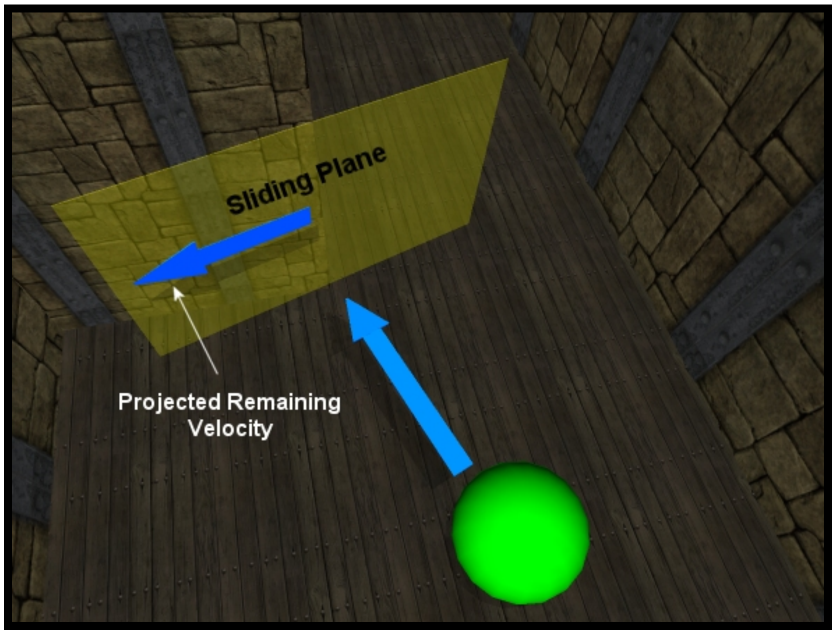
Chapter Twelve

Collision Detection and Response



Introduction

Since the first lesson in this series we have been implementing visualization applications that permit the camera and other entities to move through solid objects in the game world. But realism in games requires a sense of physical presence and adherence to some simple laws of physics in addition to the visual experience. In the real world, two objects should not be able to simultaneously share the same location. Doing so would indicate that the objects are interpenetrating one another. Since this is not a valid real world state, it cannot be a valid game state either if we want the player to feel as though they inhabit a physical and interactive game space. What our application needs is a means for determining when this situation occurs as well as a means for doing something about it when it does. These two roles fall into the aptly named categories: collision detection and collision response, respectively.

Collision detection and response is actually one of the more heavily researched areas in academic and industrial computer science because there are so many beneficial applications. For example, automobile manufacturers can develop simulations that model various impact phenomena and apply what they learn to improve safety. Of course, as game programmers, we obviously do not have to worry about life and death collision issues in our simulations, and thus our game requirements will be far less demanding. Indeed we will even allow ourselves to occasionally bend or even break the laws of physics and do things that would be virtually impossible in the real world because the results just look better on screen.

More importantly, given the steady increase of geometric complexity in 3D models and environments, it is critical that fast and precise collision determinations and corrections can be made for real-time applications to be able to benefit. Not surprisingly, speed and precision are inversely proportional concepts. The more precise we want our simulation to be, the longer it will take to perform the calculations. Testing every polygon of every object against every polygon of every other object will certainly provide a good amount of precision, but it will be expensive. Modeling proper physics equations in our response engine will generate very precise results, but it will not be as quick as using decent looking (or feeling) approximations. What is required is an appropriate degree of balance between our need for speed and our desire for accuracy.

There are a number of collision systems available to the game developer. Some are free and others are available for purchase and/or licensing. Some systems focus exclusively on the detection portion (SOLID™, I-COLLIDE™, V-COLLIDE™, etc) and require the developer to implement the response phase, while others will include the response side as well (rigid body physics libraries like Havok™ fall into this category and they often include lots of wonderful extra features). The system we will implement in this chapter will include both collision detection and collision response.

This is going to be challenging subject matter, but it is a system that involves concepts that every game programmer should be familiar with. Many of you may even have to implement a system like this at some point in your career. It is probably fair to say that, given the complexity of such systems, many of you may find this chapter to be the most demanding in the training series so far. This is due in large part to a heavy reliance on a significant number of mathematical formulas that must be understood to achieve accurate intersection determination. Rest assured though that we will make every effort in this chapter to try to leave no stone unturned. Hopefully you will find that when all is said and done, it was not so bad after all.

12.1 Our Collision System (Overview)

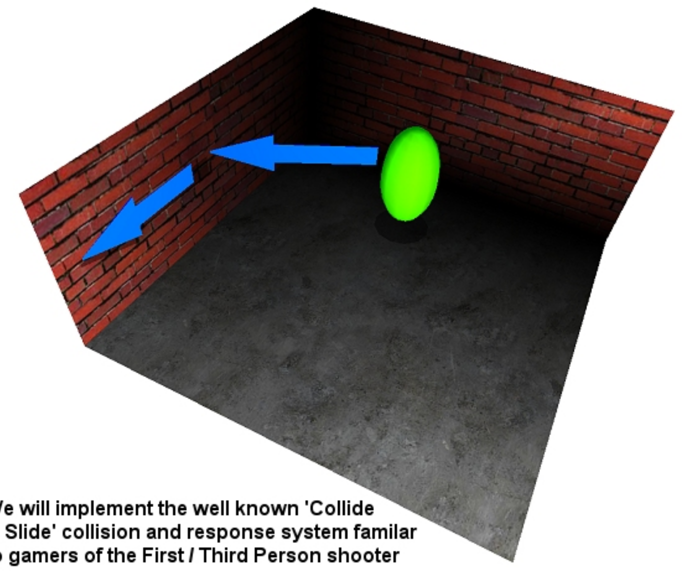
To more accurately model a physical environment, if the user wishes to move in a certain direction at a certain speed, our input code can no longer simply obey this request as it did before. This would allow the player to move through solid objects or embed himself in walls, floors, etc. The same would hold true for other moving objects. Therefore, before any moving entity in our game world can set its final position, it must feed the collision system with information about where it is currently located and where it would like to go. The collision system is tasked with determining whether a clear and unobstructed path is available. If there is a path, then it will let the caller know that it can move the entity to the desired location. If the path is blocked by one or more polygons, then the collision system will calculate and return a final position for the entity that will be different than the requested destination. Suffice to say that the moving entity ultimately ends up deferring to the collision system when it comes to navigating around in the game world. When a clear path is not available, the collision system will essentially say to the entity, "You cannot move where you wanted to move because something was in the way. However, you can move to this other location instead, which has been tested and found to be free from obstruction". Ultimately, the collision detection and response system is responsible for calculating positions that are guaranteed not to be embedded in any solid geometry that comprises the scene.

So our task in this chapter is to figure out a way to prevent moving objects from passing through the polygons that comprise our environment – easier said than done, of course. In Lab Project 12.1 we will use the camera (or more correctly, its attached CPlayer object) as the moving entity we wish to navigate through the level with simulated collision detection and response. We will also have the ability to switch to a third person mode where we will see a skinned character using the collision system in the same way. This will demonstrate that our collision system can be used by any moving entities in the game world to achieve the desired results.

To further clarify the interaction between the moving entity and the collision system we will implement in this chapter, imagine that our moving entity is an ellipsoid and that we wish to move it forward from its current position by 25 units along the world Z axis (velocity vector V = <0, 0, 25>). The entity's current position and its desired velocity vector would be fed into the collision system as a request for a position update. The collision system would then check whether any of the game world polygons would block the displacement of the ellipsoid along this path. For example, let us assume that there is a wall at a distance of 10 units in front of our moving entity. In such a case, our object should only be allowed to move as far as the wall with respect to traveling along its velocity vector (a distance of 10 units).

The collision detection system would initially detect that a collision would occur with the wall. The position that it gives back would be one that corresponded to the entity being just in contact with the wall (i.e., it is moved as far as it can go in the direction described by its velocity vector). Indeed, given this bit of knowledge, we could even say that the collision system *prevents* interpenetration from ever happening. But what about the remaining velocity? We originally wanted to travel 25 units forward, but only managed to travel 10 units before a collision was determined and additional forward movement was prevented. So in a sense, we have an 'energy surplus' that could theoretically carry us for another 15 units of travel. How we handle this remaining velocity is very much a function of the type of collision response system we decide to implement (i.e., our game physics).

In this chapter, the response system we implement will model those that are typically seen in 3D computer games of the 1st/3rd person genre. When a collision is detected, the moving entity will not simply stop at the obstructing polygon and have its excess velocity discarded. Instead, it will slide around such obstructions where possible and use up its remaining velocity. Such a response system will allow us to slide along walls, go up and down stairs and ramps, and smoothly glide over recesses in walls and floors. This is a very common response system in games because it maintains a sense of fluid movement and prevents the player (or other NPCs) from getting stuck on surfaces and edges during fast-paced gameplay.



We will implement the well known 'Collide & Slide' collision and response system familar to gamers of the First / Third Person shooter genre.

Figure 12.1

Our collision system will consist of a *detection phase* and a *response phase*. Both will be called either iteratively or recursively until the energy contained in the initial velocity vector has been completely spent and the entity rests in its final non-penetrating position (more on this later). When a request is made to the collision system to update the position of a moving entity and a collision occurs, the detection phase will determine a position that is flush against the closest colliding polygon (a wall for example). The system response phase will then project any remaining velocity onto what we call the *slide plane*. In the simplest collision cases, the slide plane is simply the plane of the polygon that was collided with (the wall plane, using our current example). There are more complex cases where the moving entity collides with the edge of a polygon (such as on an external corner where two polygons meet or on the edge of a step) where the slide plane will be determined independently from the plane of the collision polygon, and we will discuss such cases later. In the end, this system will allow the moving entity to slide up or down those steps and around such corners.

In Figure 12.1 we see the simple case where the movement of an ellipsoid would cause a collision with a wall. The ellipsoid could only be moved as far as the wall given the information returned by the detection step. Any remaining velocity would be projected onto the slide plane (the wall polygon) in the first iteration of the response step. This results in yet another velocity vector (called the *slide vector*) along which to move the ellipsoid during the next iteration of the system. Moving the ellipsoid along this new velocity vector would result in the appearance of our ellipsoid sliding along the wall.

So we now know that the information returned from the collision step allows us to move the entity into a guaranteed unobstructed position along the velocity vector, and we will spend a lot of time later in this chapter learning how the collision detection phase does this. We also know that the response portion of our system will then project any remaining velocity onto the potential sliding plane to create a new velocity vector along which to slide (starting from the new position determined by the detection phase results). If the detection phase was able to determine that the entity could move to its requested position without a collision occurring, then there will obviously be no remaining velocity for the response phase to handle. When this is the case, the collision system can simply return immediately and notify the moving entity that it can indeed move into its requested position along the velocity vector. Likewise, even if a collision did occur, if the remaining velocity that is projected onto the sliding plane is so small as to be considered negligible, then we can also return from the collision system with the new position that has been calculated. Remember, the detection phase will always calculate the closest position such that the entity is *not* interpenetrating (or passing through) any geometry. So even if we hit a wall head-on and the projected remaining velocity was zero, we can still return this new position back to the application so that it can update the old position of the entity to this new position (e.g., snug against a wall).

Not surprisingly, the real challenge occurs when the response step successfully projects the remaining velocity vector onto the sliding plane and the result is a slide vector of significant length. In these cases, we find ourselves with an iterative situation. The detection step detects an intersection along the original velocity vector and we calculate a new position such that the entity is moved as far along that vector as possible until the moment the collision occurred. The response step then projects any remaining velocity onto the sliding plane to create a new velocity vector. So in effect, the true position of the entity after the collision should be calculated by adding the new projected velocity (calculated in the response step) to the new position of the entity (calculated by the detection step). Imagine the simple case of an entity hitting a wall at an angle. The collision detection phase would move the entity up as far as the wall and return this new non-intersecting position. The response step would then project any remaining velocity onto the wall to create a new velocity vector pointing along the wall, describing the sliding direction. Therefore, the correct position of the entity after collision processing should be the combination of moving it up to the wall (detection phase) and then sliding it sideways along the wall using its new projected velocity vector (response phase).

> **Note:** It is important to make clear at this point that the terminology and descriptions we are using here are for illustration only. For example, the detection phase does not actually "move" anything. What it does, taking into account the entity's original position, velocity, and bounding volume, is determine the point beyond which the entity cannot go because something in the environment blocks it. It does not physically move the entity, but rather simulates what would happen if we did move the entity according to the requested velocity. However, for the purposes of our current high level discussion, it can be helpful to think of the breakdown of responsibility between the two system phases in the way just described. Later on we will see the precise inner workings of both system components and the details will become clearer to you.

At first, what we just described does not sound like a problem at all. After all, we have the new position and the new projected velocity vector that we have just calculated, so surely we can just do:

Final Position = New Position + New Velocity

In this case, New Position is the guaranteed non-intersecting position of the entity along the original velocity vector calculated during the detection phase and New Velocity is the remainder of the original velocity vector projected onto the sliding plane, starting from New Position.

While this is true, the problem is that we do not know whether the path to this Final Position is free from obstruction. When the collision detection phase was first invoked, it was searching for intersections along the original velocity vector (which it correctly determined and returned a new position for). But certainly we cannot simply slide the entity along the projected velocity vector calculated by the response step without knowing that this path is also free from obstruction.

As you might have guessed, our solution will need to involve either a recursive or iterative process. If the projection of the remaining velocity vector onto the sliding plane is not zero, we must feed our new position (returned from the detection phase) and our new velocity vector (calculated in the response phase) into the detection phase all over again. The detection phase will then determine any intersections along the new velocity vector and calculate a new position for the entity. This is passed on to the response step and the whole process repeats (potentially many times). It is only when the entity has been successfully moved into a position without any intersections or there is no remaining velocity left to spend beyond the point of intersection, that we can be satisfied that our task is complete. Then we can return a final resting position for our moving entity that we are confident is free from obstruction.

Given the iterative nature of the system, performance is obviously going to be a very real concern. This is especially true when we consider that the detection phase will theoretically need to check every polygon in the scene every time it is invoked in order to accurately determine unobstructed locations. Given the polygon budget for current commercial game environments, testing every scene polygon during every call to the detection phase is not an acceptable design. As a result, commercial collision systems will implement some manner of hierarchical spatial partitioning that encapsulates scene polygons into more manageable datasets for the purpose of faster queries. Indeed, once we have a working collision detection and response system of our own, we will integrate our own spatial partitioning technique (a quad-tree, oct-tree, kD-tree, etc.) to reduce the number polygons that need to be tested for intersection when the detection phase is invoked. We will be examining spatial partitioning techniques and data structures in the next chapter with an eye towards speeding up the detection phase. For now though, we will forego any concerns about optimization and concentrate solely on the core system components and how they work.

Before moving on, let us summarize some of the ideas presented:

1. The collision system will be invoked by the application whenever it wishes to move an object.
2. The collision system inputs will be the current position of the moving entity and a velocity vector describing a direction and distance over which to move the entity. We will also want to input information describing the shape/volume of the entity.
3. The position and velocity are initially passed into the **collision detection phase** of the collision system. This goal of this phase is finding the closest colliding polygon. If no such polygon exists, the system can return the sum of the position and velocity vectors. If a colliding polygon is found along the path of the moving entity:
    a. Calculate and return a new position that it is as far as possible along the input velocity vector, stopping at or before the point where an intersection would occur.
    b. Calculate and return a normal at the point of intersection. In our system, this will act as the normal of a plane that the entity should slide along if there is any velocity remaining beyond the point of intersection.
4. The updated position, the remaining velocity, and the slide plane normal (i.e. intersection normal) are input into the **collision response phase**. The goal in this phase is to apply whatever physics are needed to calculate the contact response of the entity. In our system, we will project the remaining velocity vector onto a sliding plane to generate a new velocity vector that points in

the direction that the entity should slide. We will then pass the updated position and slide velocity vector back into the system to begin the process anew.

Note: Steps 3a and 3b above are not always necessarily going to be tasks assigned to the collision detection phase in every collision system on the market. In fact, most collision detection systems literally just tell you the polygon(s) intersected, and both when and where those intersections occurred. Our detection phase will do a bit more internal housekeeping to make things a little easier to process. Theoretically, these jobs could just as easily have been assigned to the response phase of the system. How you decide to do it in your own implementations is totally up to you.

Later we will learn that the projection of the slide vector will be scaled by the angle between the original velocity vector and the normal of the sliding plane. This way, if we collide with a wall at a very shallow angle, much of the remaining velocity will survive the projection onto the sliding plane and our moving entity will slide along the wall quite quickly. However, if we collide with the wall at a very steep angle (almost perpendicular) then we are hitting the object virtually head-on and the projected velocity vector will be very small, if not zero. In this case we significantly reduce all forward movement and do not slide very much (if at all). This is the response behavior that is implemented in most commercial first-person shooter games.

Figure 12.2 demonstrates the dependency between the projection of the original velocity vector onto the sliding plane and the angle between the original velocity vector and the slide plane normal. In this image, Source is the position of a sphere prior to the update. (We will assume that the sphere is a bounding volume used to describe our moving entity inside the collision system). This current sphere position is one of the initial inputs to the collision system. The red dotted line describes the original velocity vector. This vector tells the collision system both the direction and distance we wish to move the entity. By adding the velocity vector and the Source vector, we get the Original Dest vector which describes the position we wish to move our sphere to (provided that no collisions occur).

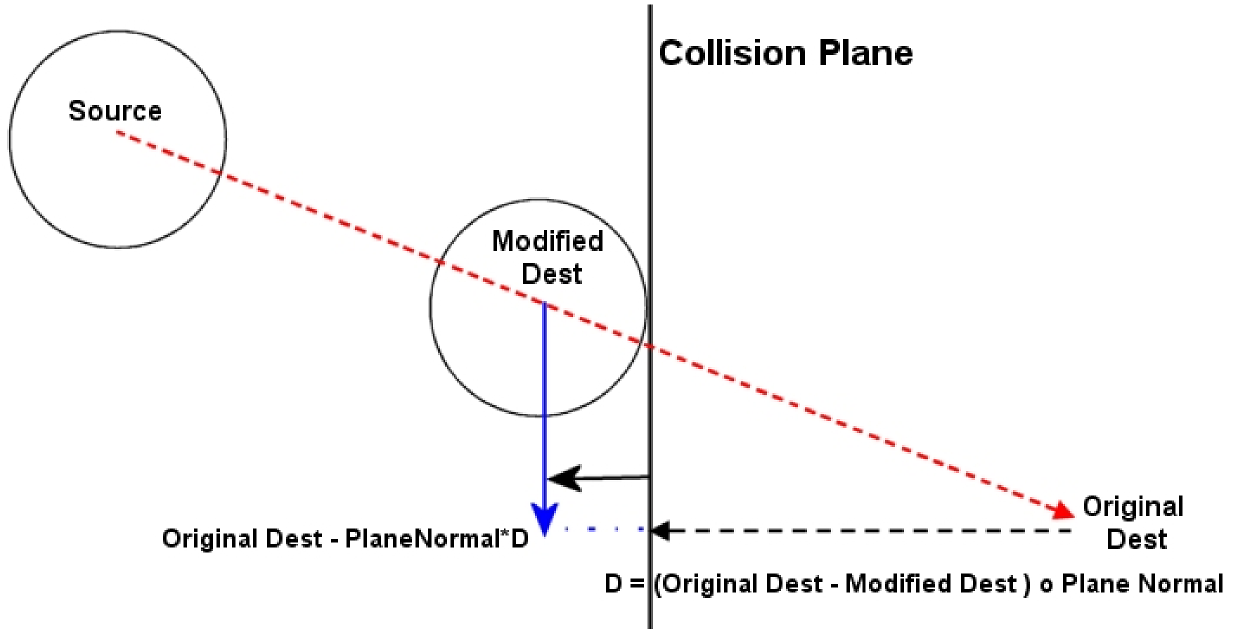

Collision Plane
Source
Modified Dest
Original Dest
Original Dest - PlaneNormal*D
D = (Original Dest - Modified Dest) o Plane Normal

Figure 12.2

We know that the first thing we do is feed this information into the collision detection phase. We are imagining in this instance that a polygon (whose plane is marked as Collision Plane) was found to be blocking the intended path. The detection phase would thus need to return the new sphere position, called Modified Dest, since Original Dest is unreachable. Note that Modified Dest is a sphere position that it is touching, but not intersecting (i.e. does not penetrate), the Collision Plane. We can see clearly that the center of the original sphere only traveled a little less than halfway along the requested velocity vector before hitting the plane, so there is certainly some "leftover" velocity that we can use for sliding. The remaining velocity vector in this example is the vector from Modified Dest (the new center position of the sphere) to Original Dest (the final position of the sphere as originally intended).

When the collision detection phase returns, we have a new sphere position (Modified Dest) and a slide plane normal (the normal of Collision Plane). We can calculate any remaining velocity as:

Remaining Velocity Vector = OriginalDest – Modified Dest

If we perform a dot product between the remaining velocity vector and the unit length plane normal, we get D. D is the length of the adjacent side of a triangle whose hypotenuse is given by the remaining velocity vector (the length of the black dotted line in Figure 12.2). By subtracting D from the Original Dest vector, along the direction of the plane normal, we create the vector shown as the blue arrow in the diagram. This is the projected velocity vector, which forms the opposite side of the triangle. As you can see, this is the direction the sphere should move (i.e., along the plane) to use up the remainder of its velocity. It should also be noted that as the angle of the original velocity vector gets closer to perpendicular with the plane normal, the projected velocity is scaled down more and more until hardly any sliding happens at all (i.e., when the sphere it hitting the plane head-on).

There are times when the slide plane normal returned from the detection routine will not be the same as the normal of the polygon the entity intersected. This is typically the case when the entity collides with a polygon edge (see Figure 12.3).

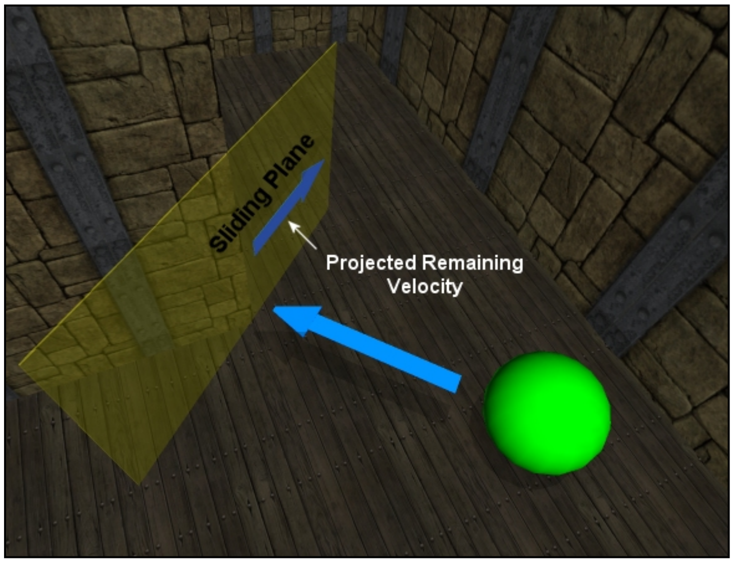


Figure 12.3

In Figure 12.3 we can see that if the green sphere was moved in the direction of the blue arrow, it would collide with the edge where two polygons meet to form an external corner. When this is the case, we want the angle at which the sphere intersects with the edge to determine the orientation of the slide plane. As it happens, this is quite simple to calculate.

When the detection phase calculates the intersection between the edge of a polygon and an entity (a sphere in this example), it will generate the plane normal by building a unit length vector from the intersection point on the edge to the sphere center point. If you imagine drawing a line from the center of the sphere to the point on the edge where the sphere would hit the corner, you would see that this vector would be perpendicular to the slide plane shown in Figure 12.3. Normalizing this vector gives us the slide plane normal and indeed the slide plane on which to project any remaining velocity that may exist. We can see that in this example, when the sphere hits the edge, it would slide around the corner to the right. Because the slide plane normal returned from an edge intersection is dependant on the position of the entity (the sphere center point in this example), we can see that a different sliding plane would be generated from a collision against the same edge if the sphere used a different angle of approach (see Figures 12.4 and 12.5).

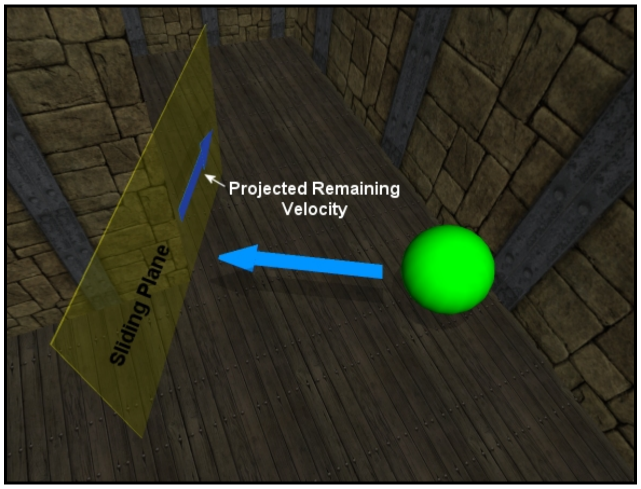


Figure 12.4

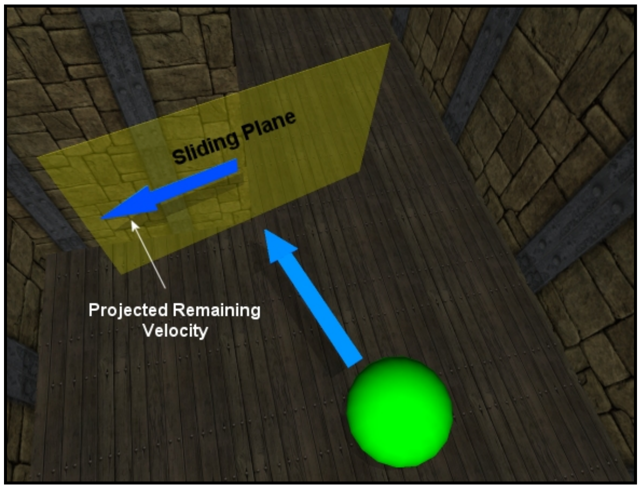
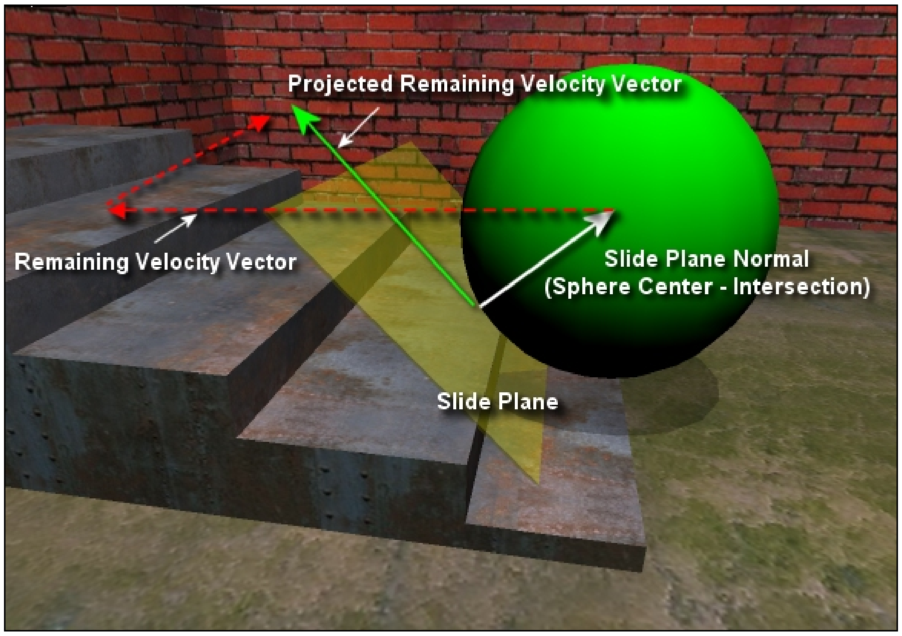


Figure 12.5

It is the sliding plane (and the projection of the remaining velocity vector onto this plane) that also allows the system to automatically handle the navigation of stairs and ramps as well. For example, in Figure 12.6 we see the same sphere attempting to navigate a flight of stairs.


Projected Remaining Velocity Vector
Remaining Velocity Vector
Slide Plane Normal
(Sphere Center - Intersection)
Slide Plane
Figure 12.6

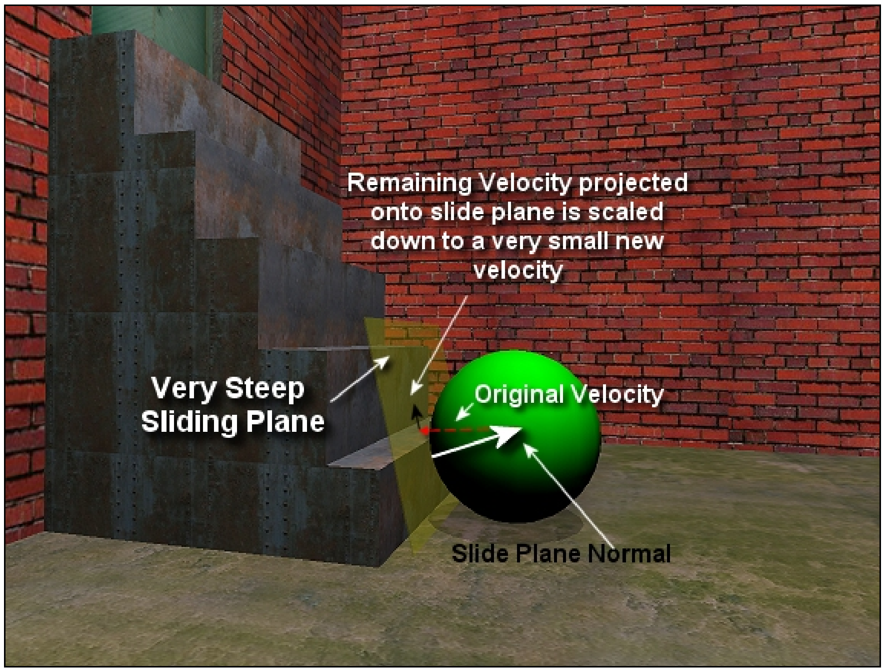
In this case, the sphere is already beyond the edge the first step and is trying to travel forward from its current position when it intersects the edge of the second step. The long red dashed line traveling from the sphere center and pointing to the left is the remaining velocity vector after we have first collided with the second step edge.

The small red dashed line shows us the direction of the projection of the remaining velocity vector onto the slide plane.

The collision detection step would determine that somewhere in the bottom left quadrant of the sphere an intersection would happen with the edge of the second step. The slide plane normal would be generated by calculating a vector from this intersection point on the step edge to the sphere center point and normalizing the result. The yellow transparent quad shows the slide plane that would be created for this normal and the plane. This is the plane onto which the response step will project any remaining velocity.

The dot product between the remaining velocity vector (the section of the vector that starts at the new sphere center position calculated by the detection phase) and the slide plane normal creates a new vector tangent to the plane describing the direction in which the sphere should continue to move. This is shown as the green arrow labeled Projected Remaining Velocity Vector. Suffice to say, it is by moving the sphere from its current position (calculated by the detection phase) along this slide vector, that the sphere ends up in its correct final position for the update. We will have moved the sphere as much as possible in the original direction and then diverted all remaining velocity in the slide direction.

As mentioned previously however, the response step cannot simply assume that it can add the slide vector to the position of the new sphere center without testing for obstruction. That is why the response phase must invoke the detection phase again, using the new slide vector as the input velocity vector to the system. This in turn invokes another response phase, and so on. Only when the remaining velocity vector has zero (or near zero) magnitude, do we consider the sphere to be in its final position. Of course, this is always the case as soon as the detection phase determines that it can move the entity along its input vector (which may be the slide vector calculated in a previous response step) without obstruction, since there will be no remaining velocity vector in this instance when the detection phase returns.


Remaining Velocity projected onto slide plane is scaled down to a very small new velocity
Very Steep Sliding Plane
Original Velocity
Slide Plane Normal
Figure 12.7

Note as well that the height of the step with respect to the entity is very important. The shallower the step, the shallower the slide plane generated. Therefore, any remaining velocity vector will be more strongly preserved when projected onto the sliding plane. For example, in Figure 12.7 we can see the same sphere bounding volume trying to climb a much steeper flight of steps with larger step sizes. The point of intersection with the step is much higher up on the surface of the sphere than before. As such, the slide plane normal calculated from the intersection point to the sphere center position is almost horizontal. In this case, the slide plane is so steep that when any remaining velocity vector is projected onto it, it is almost scaled away to nothing. Depending on your implementation what you might see in this situation is the sphere unsuccessfully attempting to mount the step. Perhaps you would even see a small upwards movement before the sphere came back down again.

Note that this is generally going to be the correct response for this situation. If we imagine the sphere in Figure 12.7 to be the bounding primitive for a character, one step alone would be about half the character's height. We would not expect a character in our game to simply slide over such large obstructions without having to perform some special maneuver (e.g. jumping or climbing). This is an extremely useful side effect of using the sliding plane style of collision response. Indeed it makes the system almost totally self-contained; it will correctly slide over small obstructions (such as steps and debris) and automatically generate very steep sliding planes for larger obstructions, where intersections with the sphere happen a fair way up from its base. Again, the taller the obstruction, the higher the intersection point between the sphere and the polygon will be. As such, the slide plane normal generated by subtracting the intersection point from the sphere center will be almost, if not totally, horizontal. As we have seen, very steep sliding planes zero out any remaining velocity during projection. Therefore, assuming the bounding volume is suitably tall to be consistent with the scale of the geometry, the response step will automatically create shallow sliding planes for collisions against small obstructions like stairs but stop the entity in its tracks when it collides head-on with taller obstructions.

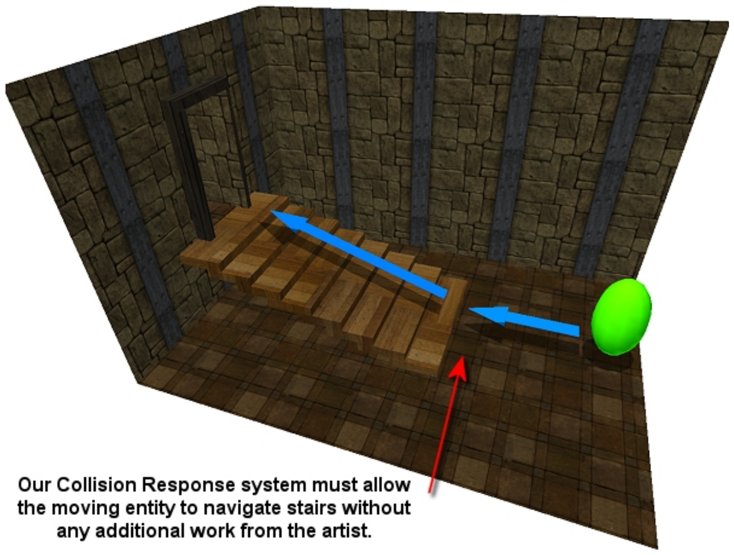

Our Collision Response system must allow the moving entity to navigate stairs without any additional work from the artist.

Figure 12.8

You may wish to verify this behavior now by running Lab Project 12.1. Notice how the camera smoothly glides up stairs and ramps and, with the integration of gravity with the velocity vector, how it falls to the ground when the character walks off a steep ledge. In truth, the gravity vector is not really part of the collision system (it is implemented as part of the CPlayer class movement physics). Every time the position of the CPlayer (which the camera is attached to) is updated, it adds a gravity vector to its current velocity vector. Gravity acts as a constant force, accelerating the entity downwards along the negative world space Y axis during every frame. Since this velocity vector is an input to the collision system (technically, gravity is an acceleration, but we integrate it into a velocity vector for our physics calculations), the entity will automatically fall downwards when there is not a polygon underneath blocking the way. Without the floor of the level for it to collide against, the moving entity would continue to fall forever. While we will talk about this in much more detail later, it demonstrates that from the perspective of the collision system, even the floor of our level is just another polygon to collide and slide against.

As mentioned, stairs and ramps are not the only obstructions that the sliding plane response can handle. It also allows us to slide over small obstructions placed in the game world using the same logic. We certainly would not want the player to be killed in the middle of an intense firefight because his character got stuck while running over a 2-inch thick wood plank. Indeed we want our artists to be able to populate the floor space of our game worlds with lots of interesting details like cracks, dips, splits and piles of debris; we just do not want these geometric details to prevent us from navigating across the floor (unless they have been designed specifically to be large enough for that task). Our collision system will automatically handle such small obstructions (see Figure 12.9) using the sliding plane response mechanism.

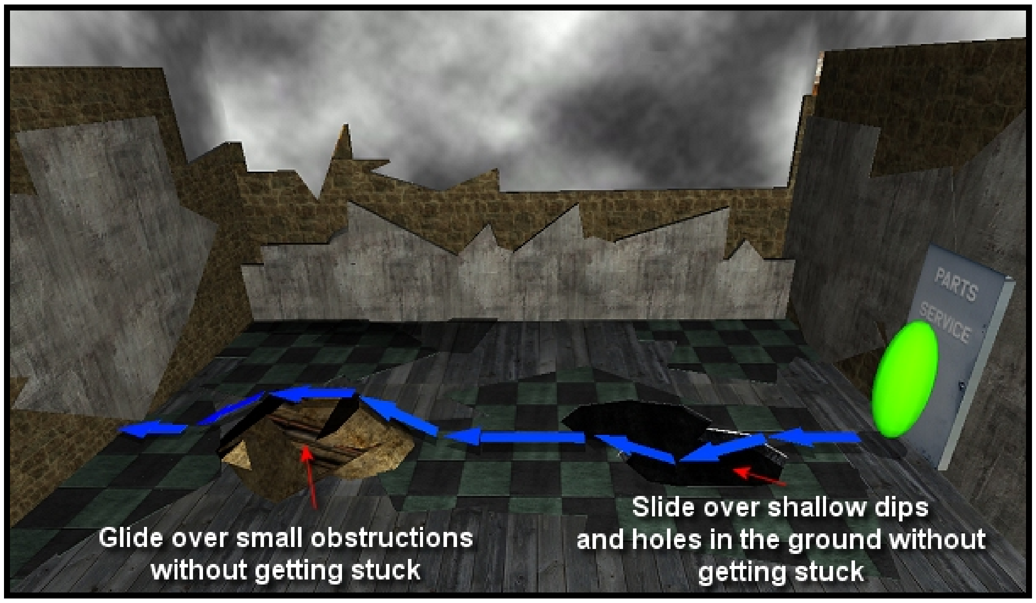

PARTS
SERVICE
Glide over small obstructions without getting stuck
Slide over shallow dips and holes in the ground without getting stuck

Figure 12.9

The sliding plane response concept will be familiar to gamers who enjoy titles in the first and third person game genres, but it is worth noting that this behavior is easily replaced or augmented according to application requirements. For example, you could replace the response system so that instead of projecting any remaining velocity, you calculate a reflection vector with respect to the collision plane. This would cause the objects to bounce off the polygons in the environment and might be a useful response system when implementing an outer space themed game. Alternatively, the response phase could be simplified so that the position returned from the first iteration of the detection phase of the collision system is used 'as is'. This would mean that when objects collide with the environment, they stop moving. This would feel a little bit like navigating around a world where all polygons are coated with a strong adhesive. While not very useful in most games, it would at least provide a sense of physical presence to your polygons. The larger point here is that your response physics can be as simple or as complicated as you want them to be. As long as you maintain separate detection and response sub-systems, you can configure your results as needed.

In actual practice, the (sliding) response phase we use in our collision system, which features most of the behavior we have discussed so far, can be implemented using only a few lines of code. It involves nothing more than projecting (using a dot product) the remaining velocity vector onto the sliding plane returned from the detection phase and then re-calling the detection phase with new input parameters. Technically, there are a few additional tests we will do in the response phase to reduce certain artifacts. For example, a jittering effect can occur when forcing an entity into an area where two polygons form an internal corner. This jittering results from trying to solve multi-polygon collisions one at a time rather than simultaneously (which is much more computationally expensive). Fortunately, jittering artifacts can be mostly eliminated with another dot product. This will make sure that during any iteration, the response step does not try to slide the entity in the direction opposite the initial input velocity vector for a given position update. We will cover all of this in detail when we discuss the full implementation later in the chapter. For now, just know that the response phase is quite straightforward and will not require significant effort to implement.

The detection phase is much more complex. If all we had to do was detect was whether or not an entity's bounding volume currently intersects polygons in our scene database, then that would be fairly easy to do (relatively speaking – there is still a good amount of math involved, as we will see later). However, the objects we are concerned with are not standing still, so we have to detect whether the bounding volume will intersect any geometry in the scene *at any time* along its movement vector, before we actually move it. The mathematics for this are a little more involved. Our collision detection phase will also be responsible for returning a new position describing the furthest point the entity can move along that vector before the intersection occurred.

Before we begin our detailed examination of the detection phase and look at the various intersection routines we must implement that make it all possible, we will discuss our high level objectives for the system design and talk about some of the key components that will be necessary for the final system to come together. Note that this chapter is going to be a little different from previous ones in that we will actually be including a good amount of discussion of the lab project source code here in the textbook rather than leaving it all for the workbook. Looking at the source code should help cement a lot of the theory we will be introducing, so keep this in mind as you move forward.

12.2 The Collision System Design

As stated, our primary goal in this chapter is the implementation of a robust and reusable collision system. Along the way, we will progress through the development of a set of class code (CCollision) that can be used by moving entities in our game world to provide accurate collision detection and response. This collision system will be able to be plugged into all of our future applications and can be easily modified and extended to provide different response behaviors.

The collision detection phase will incorporate several static intersection routines. These routines will be included in the system's namespace. Because they are static, these intersection routines can also be called from any module outside the collision system. This will provide future applications with a useful library of intersection functions that can be used with or without the main collision system

NOTE: Because our intersection routines are static and rely on no member variables of the CCollision class, they may be invoked using the CCollision namespace even if a CCollision object has not been instantiated.

Finally, the collision system we develop will place no burden on the project artists to create their geometry in a particular way to make it 'collision compliant'. A primary goal is for the system to work with arbitrarily arranged polygons and meshes (i.e., a polygon soup). This is not to say that you might not want to include artist-placed collidable geometry for particular game purposes (or to solve minor artifact problems should they occur), but that on the whole, the system should be able to work with anything we can throw at it.

12.2.1 Core Components

Our collision system will contain four main components:

Component 1: The Collision Geometry Database

Many tests will need to be made against the geometry of the environment to determine a clear path for a moving entity. Often, the triangles comprising the game world will reside somewhere that would make them either inefficient or impossible to access for collision testing. For example, the vertex data may reside in either local or non-local video memory, or it may have been compressed into a proprietary format that only the driver understands. Having to lock such vertex buffers will cause stalls in the pipeline if the geometry has to be un-swizzled into a format the application can read back and work with. If the buffer is located in video memory, we know that reading back data from an aliased pointer to video memory will result in very serious performance degradation.

Of course, we could create managed vertex buffers (i.e., a resident system memory copy maintained by the D3D memory management system) and lock this buffer with a read-only flag. This would allow our collision system to lock and read back that data without causing a pipeline stall or a possible re-compression step. But this is still far from ideal. When an entity moves, we do not want to have to lock vertex and index buffers and determine which indices in the index buffer and vertices in the vertex buffer form the triangles the entity must collide with. Large scenes would potentially require locking and unlocking many vertex and index buffers, and would prove to be terribly inefficient.

Thus, it is usually the case that a collision system will maintain its own copy of the scene geometry. That data can be stored in an application-friendly format in system memory, making it very efficient for the CPU to access. This is very important because the collision detection will be done by the host CPU and not by the graphics hardware.

Our CCollision class will expose a series of functions that will allow an application to register geometry with the collision system. Therefore our CCollision class will maintain its own **geometry database**. An application loading geometry for scene rendering can just pass the same triangle data into the collision system's geometry registration methods. As our collision system will allow us to pass triangle data to its geometry database as a triangle list, this makes it very convenient to use. Both X files and IWF files export triangle lists, so when our application loads them in, we can simply pass the triangle lists directly into the collision system so that it can store an internal copy for its own use. For convenience, the collision system we develop in this chapter will also allow us to pass in an ID3DXMesh or even a CActor object. It will handle registration of the triangle data contained with these object types automatically. Thus our application could load a single X file containing the main game level (using D3DXLoadMeshFromX for example) and just pass it into the CCollision geometry registration function. The collision system would then have its own copy of the scene geometry and all the information it needs about the environment.

So we will accept that the triangle data used for rendering and the data used for the collision system, while related, are totally independent from one another. While this obviously increases our memory requirements, the collision geometry need not be an exact copy of the triangle data used to render the game world. For starters, we can reduce memory footprint by eliminating unnecessary vertex components like normals and texture coordinates. Additionally, our geometry can be defined at a much coarser resolution to speed up collision queries and to conserve memory. For example, instead of our moving entity having to test collisions with a 2000 triangle sphere, the collision detection geometry version of this sphere may be made up of only 100 polygons approximating a rougher bounding volume around the original geometry. This allows us to speed up collision detection by reducing the number of triangles that need to be tested and it lowers the memory overhead caused by maintaining of an additional copy of the original scene data. If you consider that a wall in your game world might have been highly tessellated by the artist to facilitate some fancy lighting or texturing technique, the collision system's version of that wall could be just a simple quad (eliminating pointless testing). Therefore, while not required by the collision system, the game artists may decide to develop lower polygon versions of the game environment for collision purposes or you may elect to use a mesh simplification algorithm to accomplish the same objective.

One additional benefit can be derived from the separation of rendering and collision geometry: the collision database can contain geometry that the actual renderable scene does not. For example, we can insert invisible walls or ramps in our level simply by adding new polygons only to the collision data. We have all seen this technique employed in games in the past, where the player suddenly hits an invisible wall to stop him wandering out of the action zone.

Component 2: Broad Phase Collision Detection (Bulk Rejection)

The collision detection portion of most systems will distribute the intersection determination load over two stages or phases: a broad phase and a narrow phase. The goal of the **broad phase** is to reduce the number of *potentially* collidable triangles to as few as possible. The goal of the subsequent narrow phase (discussed next) is to test those triangles for intersection and return the results to the response handler.

We simply do not have enough processing power to spare on today's machines to test every polygon in the collision database. While this is obviously true for very large levels, it is also important for small levels (there is no point in wasting CPU cycles unnecessarily). Before intensive calculations are performed at the per-triangle level to determine whether our entity intersects the scene geometry, the broad phase will attempt to (hopefully) eliminate 95% or more of the geometry from consideration. It will accomplish this through the use of any number of different techniques -- hierarchical scene bounding volumes and/or spatial partitioning are the most common solutions. These techniques divide the collision geometry database into some given number of volumes (or 'regions' or 'zones'). We know that if our moving entity has an origin and a destination in only one region, then all other regions (and the polygons stored in those regions) cannot possibly be intersecting our entity along its desired path and require no further testing in the main collision detection phase.

The output of broad phase testing is (hopefully) a very small list of triangles which *might* block the path of the entity, because they essentially 'live in the same neighborhood'. This list is often called the *potential colliders* list as we have been unable to definitively rule out whether or not any of these triangles are blocking the path of the entity. These triangles (i.e., the potential colliders) become the inputs to the narrow phase where the more time consuming intersection tests are done between the moving entity and each triangle (one triangle at a time).

This component of the system is often separate from the detection and response steps that comprise the core of the collision system discussed earlier in the chapter. In fact, the broad phase may not even be present in a collision system. Although this is the first functional phase in the collision process that is executed whenever a moving entity requires a position update, and indeed it is the one that is most crucial to the performance of our own collision system, this is the phase that we will be adding to our collision system last. In Lab Project 12.1, the broad phase will be absent from our collision system. In fact, it will not be added until the next chapter.

Of course, we would not make such a design decision without good reason. The broad phase we will add to our system in the next chapter will be optionally a quad-tree, an oct-tree, or a kD-tree. These data structures will subdivide space such that the collision geometry database is partitioned into a hierarchy of bounding boxes. Each box is called a *leaf* and will contain some small number of polygons. Because of their hierarchical nature, these trees make per-polygon queries (such as intersection tests) very fast since they allow us to quickly and efficiently reject polygons that could not possibly be intersecting our entity along its desired path.

The broad phase can also be implemented using less complex schemes that rely purely on mesh bounding volume testing (e.g. sphere/sphere, sphere/box, etc.). But even in this case, some manner of hierarchy is preferable. We talked a little bit about such a system design towards the end of Chapter Nine, but we will not give any serious attention on this type of scheme until we examine scene graphs in Module III of this series. Instead we will focus on the highly efficient and very important spatial partition trees just mentioned. Because spatial partitioning is a significant topic all by itself (which needs to be studied in great depth by all game programmers), we have dedicated an entire chapter to it in this course. So we will leave broad phase testing out of the equation for now so as not to lose our focus on the core collision detection and response handling routines.

Component 3: Narrow Phase Collision Detection (Per-Polygon Tests)

In the **narrow phase** of the collision detection step, we will determine the actual intersections with our scene geometry. In our particular system, during the narrow phase we will also attempt to calculate a new non-intersecting position for the entity.

> **NOTE:** It is very important to emphasize that our narrow phase will always return a position that is non-intersecting. This is not always the case in every collision system, but it common enough that we decided to do it in our system as well.

This phase is ultimately responsible for determining if the requested movement along the velocity vector can be achieved without an environment collision taking place. If a collision will occur at some point along the path described by the velocity vector, the narrow phase will determine the maximum position the entity can achieve along that vector before a penetration occurs. It will then return this new position along with the normal at the point of intersection on the polygon (treated as, a sliding plane normal in our case) to the response phase of the system. This data will be used by the response phase to calculate a slide vector which will be fed in once again to the collision detection system to test for obstruction along the path of the slide vector, and so on. Only when the detection phase is executed without a collision occurring (or if the response phase considers the remaining velocity vector to be of insignificant length) is the iteration stopped and the position returned from the last execution of the detection phase. We can then set the final position of the entity to this new location.

Component 4: Collision Response

The responsibility of our system's **collision response** component is the generation of a new velocity vector (a slide vector) if a collision occurred in the detection phase. This slide vector is computed by projecting any remaining velocity from the previous velocity vector onto a sliding plane returned to the response step by the narrow phase collision detection step. If the projection of the remaining velocity from a previous intersection results in a vector magnitude of insignificant length, then the response step will simply return the position generated during the detection phase back to the application. If a collision occurred in the detection phase and the remaining velocity vector produces a vector of significant length when projected onto the slide plane, the response phase will call the detection phase again with the new position and the slide vector acting as the new velocity vector. The response step will continue this pattern of calling back into the detection phase until no collision occurs or until the length of a projected slide vector is considered insignificant.

The possibility exists with certain types of complex geometry and a large enough initial velocity vector that an expensive series of deep recursions (a slide loop) could occur. In such rare cases, the detection phase would initially detect a collision, the response phase would calculate a new slide vector and pass it to the detection phase, which would detect another collision, and so on. We run the risk of getting mired in a very time consuming recursion waiting for the velocity vector to be completely whittled away. Therefore, our system will put a maximum limit on the number of times the response phase can call the detection phase. If we still have remaining velocity after a maximum number of calls to the detection phase, we will simply discard any remaining velocity and return the non-intersecting position returned from the last detection step. This will all be revisited later when we examine the response step from a coding perspective.

12.2.2 Bounding Volume Selection

In most real-time collision systems, entities are represented using simple bounding volumes like boxes, spheres, cylinders, etc. During broad phase tests, this means that we typically wind up computing very quick volume/volume intersections to reject large areas of our geometry database. Narrow phase collision detection is also often reduced to relatively inexpensive testing for intersections between a bounding volume and a triangle, making this phase suitably efficient for real-time applications.

Note: Many collision detection systems (especially those used in academia) will compute polygon-polygon collisions. These tests are more accurate and thus more expensive. For our purposes, we will generally not require such detailed collision information. Volume-polygon testing is typically all that is needed for most real-time game development projects, although this is obviously determined by application design requirements.

While there are many different bounding volumes a collision system can use to represent its moving entities, the system we will develop in this course uses ellipsoids as the primary bounding volume. All of our moving entities, regardless of type (a character, a motor car, a bouncing ball), will be assigned a bounding ellipsoid which encompasses the object. When our collision system is asked to calculate the new position of an entity, its ellipsoid will be tested for intersection with the geometry database and used to calculate a new position for that entity. Our decision to use ellipsoids was based on their simplicity of construction and use and their ability to produce a decent fitting bounding volume around the most common types of moving objects we will encounter.

An ellipsoid is the three-dimensional version of an ellipse. It can have three uniquely specified radii along its X, Y and Z extents. Ellipsoids tend to look like stretched or elongated spheres when their radii are not equal. Indeed in the study of quadric surfaces (a family of surfaces to which spheres, cylinders and ellipsoids belong), we find that the sphere is classified as a special form of ellipsoid – one in which the radii along the X, Y and Z extents are equal, giving a single uniform radius about the center point for every point on the sphere's surface.

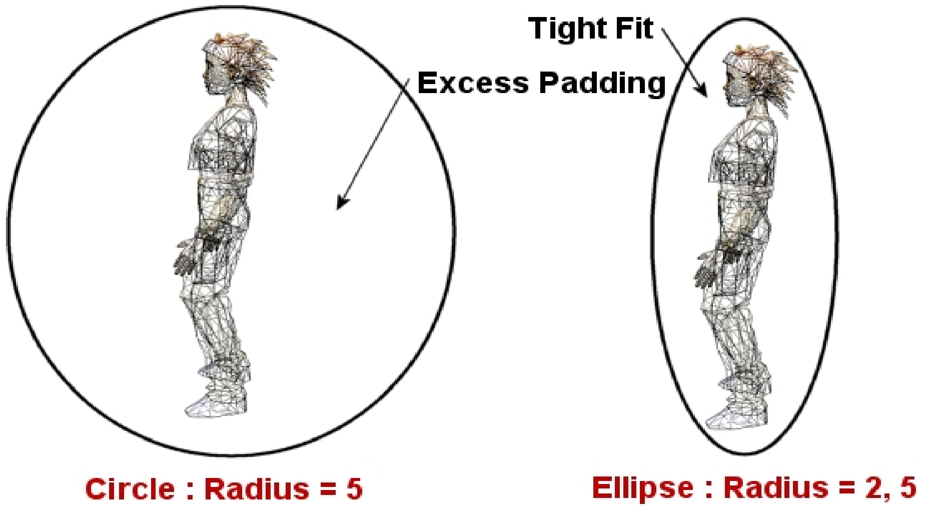


Figure 12.10

We will see later that our collision detection routines will work with unit spheres (spheres with a radius of 1) rather than ellipsoids. Intersection testing is simpler mathematically and more computationally efficient when working with spheres.

However, spheres are not a very friendly bounding volume for most shapes (although they would be a great choice for a beach ball!). If we consider a humanoid character (see Figure 12.10), we can see that if we use a sphere to bound the character, we will get a very loose fit both in front and behind (as well as left and right in the 3D case). This is due to the fact that the sphere's radius has to be sufficiently large to contain the full height of the humanoid form. However, humanoids are generally taller than they are wide, which results in the excess empty space we see depicted. If we imagine the bounding sphere in Figure 12.10 being used by our collision detection system, we would expect that even when the front of the character is quite far away from a polygon (a wall for example), its bounding sphere's surface would intersect it. The character would be prevented from getting any closer to the wall even though it would be clear that adequate room to maneuver exists. The situation is even more problematic when the character wants to walk through a standard sized doorway. The sphere would simply be too wide to accommodate this movement, even when it is clear that the character should easily fit through the opening.

In Figure 12.10 we see that because the height radius can be different than the width/depth radii, we can use an ellipsoid to describe a stretched sphere that better bounds the volume of the entity we are trying to approximate. In this example, the ellipse (2D here for ease of demonstration) has a width radius of 2 and a height radius of 5. An ellipsoid can also be used to tightly bound an animal form (a quadruped) by making the ellipsoid width/depth radii larger than its vertical radius. For example, if you look again at the ellipse shown in figure 12.10 and imagine that the radii were switched so that the ellipse had a width of 5 and a height of 2, we would have a sphere that has been stretched horizontally instead of vertically. This is ideal for bounding a quadruped or even some types of vehicles and aircraft.

So it would seem that from our application's perspective, using the more flexible ellipsoidal bounding primitive for our moving entities is preferable to the use of generic spheres. However, as mentioned, the detection phase would rather use spheres as the bounding primitive to keep the intersection testing mathematics as simple and efficient as possible.

As it happens, we can have the best of both worlds. We can use ellipsoids as the bounding primitives for our collision system's moving entities and, by performing a minor transformation on the scene geometry prior to execution of the detection routines, we can allow our detection routines to use unit sphere mathematics. How does this work?

Figure 12.11 shows a small scene and the position of a bounding ellipsoid. In this example, we will assume that the vertices of the geometry and the ellipsoid position are both specified in world space. When we describe the position of the ellipsoid in world space, we are actually describing the world space position of the ellipsoid's center point. In Figure 12.11 we will assume that the ellipsoid has width and depth radii of 1.0 and a height radius of 4.0. Thus, in contrast to a sphere whose radius can be defined by a single value, the extents of the ellipsoid are described by three values (X, Y, and Z radii). They are usually stored in a 3D vector referred to as the radius vector. In our example, the radius vector for the ellipsoid in Figure 12.11 would be:

Radius Vector = (1.0 , 4.0 , 1.0)

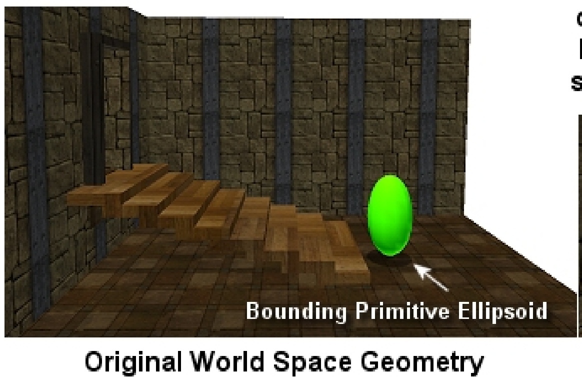


Figure 12.11 : Polygon data in world space

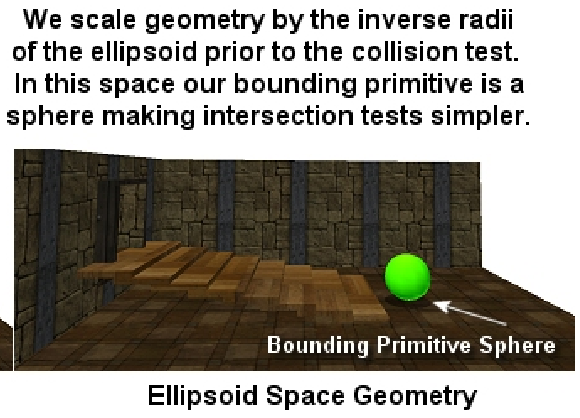

We scale geometry by the inverse radii of the ellipsoid prior to the collision test. In this space our bounding primitive is a sphere making intersection tests simpler.
Bounding Primitive Sphere
Ellipsoid Space Geometry

Figure 12.12 : Polygon data squashed into ellipsoid space

12.2.3 Inputs/Outputs

The collision system is activated by the application when it wishes to move an object in the scene. This is done with a call to the function shown next. This is the only CCollision method our application will need to call when it wishes to update the position of a moving object.

> **Note:** It should be noted that the *actual* CollideEllipsoid function exposed by our collision system takes a few extra parameters beyond the prototype shown here. These additional parameters will be described later in the chapter when the timing is more appropriate.

```
bool CCollision::CollideEllipsoid (   const D3DXVECTOR3& Center,
                                      const D3DXVECTOR3& Radius,
                                      const D3DXVECTOR3& Velocity,
                                      D3DXVECTOR3& NewCenter,
                                      D3DXVECTOR3& IntegrationVelocity )
```

The function is passed the position of the center of the ellipsoid, the ellipsoid radius vector (describing its three radii) and a velocity vector describing the direction we wish to travel. All three are given in world space units. The length of the velocity vector describes the distance we wish to travel, so we can think of the velocity vector parameter as having both a direction of movement and a desired distance for this particular frame update. The next parameter is a 3D vector which, on function return, will contain the updated position for the ellipsoid. This position will be used to update the position of the entity represented by the ellipsoid. Our final parameter is a 3D vector output parameter called IntegrationVelocity which will contain the 'new' velocity of the object on function return. This final parameter requires a little more explanation.

As discussed, we will pass into the function a velocity that describes the direction and distance we wish to move the object. Our collision system will then repeatedly iterate to move the ellipsoid along this velocity vector until it winds up in a position where no more collisions occur and no velocity remains. We also mentioned earlier that every time the detection phase returns true for an intersection, the remaining velocity (the portion of the velocity vector beyond the point of intersection) is projected onto the slide plane and used to call the detection phase again. This continues until we have spent all the energy in the original velocity vector. It is here that we encounter a problem.

The application will be controlling the velocity of the object using some manner of physics calculations. In our Lab Project for example, when the player presses a movement key, a force is applied to the player. In the player update function, this force is combined with air resistance, drag, and friction to ultimately produce an acceleration which is in turn, integrated into a new velocity vector. This is useful because it means we can apply a large initial force when the user presses a movement key, which results in a vector which can be decelerated to zero over a number of frames (as gravity, drag, etc. operate on the object's mass). When the user releases the movement key, the player will smoothly slow to a halt rather than abruptly stop in their tracks. So what does this have to do with our collision system?

Each time the velocity is calculated in the player's update function, it is later fed into a collision system which whittles it away to nothing during execution. Remember that the system will only return once that velocity has been totally spent. The problem is that our collision system should really not alter the speed of our player in this way. For example, imagine we were making a space combat simulation where our object (a spaceship) was supposed to be moving through a vacuum. Once the player pressed a key to apply a force/thrust to the ship, even if that key was only pressed for a short time, the object would continue to move at that velocity forever. This is because in a vacuum there are no environmental forces (like resistance and drag) that act to decelerate the object over time. So we know in this case that as soon as the force was applied and the velocity vector was determined to be of a certain magnitude, it should remain at that magnitude for all future frame updates. But if we let our collision system whittle away our actual velocity, every time we called the CollideEllipsoid method for our spaceship, the final velocity would be zero (even if no intersections occur). Since the application needs to apply velocity during every frame update (assuming no forces counteract it) and a zero velocity would clearly have no effect, this is not very helpful.

Now you might think that the obvious solution to the problem is to let the collision system work with a temporary velocity vector and let the object maintain its own copy that is never tampered with by the collision system. But that only works if our object never hits anything. The problem with this approach is that while the collision system should not tamper unnecessarily with the object's speed, it absolutely needs to be able to modify the direction of the velocity vector according to collision events.

Imagine that the player collides with a wall and the collision response step projects the remaining velocity onto the wall and we eventually move the sphere into its final position. At this point the velocity vector will be spent and the collision system can return. However, the collision system needs to let the application know that the object is now traveling in a completely different direction. When the collision system was invoked, the player's velocity vector had a direction that carried it straight into the wall. The system detected this and the response step ultimately modified the direction of travel. Indeed, if several intersections occur in a single movement update, the direction might have to be changed many times before the object comes to rest in its final position. Our collision system will need to return this new direction vector back to the application so that it can use it to overwrite the previous velocity vector and integrate it into physics calculations during future updates. Our collision system will also try to maintain as much of the original input velocity's magnitude as possible. It is this integration velocity vector that is returned from CollideEllipsoid and it represents the new velocity vector for the object.

Calculating the integration velocity is easy enough as it is done in nearly the same way that we project the remaining velocity onto the slide plane. Every time a collision occurs, the response phase receives a slide plane and an intersection point. It uses this information to project the remaining velocity onto the slide plane to generate a new velocity vector as input to the next detection call. All we have to do now is repeat this process during iteration to generate the integration velocity. There is one big difference however. When the integration velocity is projected onto the slide plane, we use the *entire* vector and not just the portion of the vector that lies behind the slide plane. We can think of both the slide plane and the integration velocity as being located at the origin of the coordinate system. A simple direction change is achieved by projection onto the new plane to create a new integration velocity for the next iteration. When the collision detection phase finally returns false (meaning no more responses have to computed), we will have the final integration velocity to return to the application. Note that the continual projection of the integration velocity onto different planes during the iterative process will mean that its length (i.e., speed) will still degrade by some amount. However, when only a few shallow collisions occur (e.g. clipping a corner or sliding into a wall at a small angle), most of the object's original speed will be maintained.

From the application's perspective, using our collision system could not be easier. We just tell it where we are currently situated (Center), how big we are in the world (Radius) and the direction and distance we wish to move from our current position (Velocity). On function return, we will have stored in the parameter (NewCenter) the new position we should move our entity to. Of course, this may be quite different from the position we intended to move to if a collision or multiple collisions occurred. We may have requested a movement forward by 10 units but the detection phase may have detected a collision against a wall. The response phase may have then calculated that it needs to slide to the right by 5 units and called the detection phase again to test that the slide vector path is clear. The detection phase may have discovered another intersection along the slide vector and therefore the position of the ellipsoid may have only been moved partially along the first slide vector. Any un-spent velocity of the slide vector will then be handed back to the response system which will use the remaining velocity of the previous slide vector to calculate a new slide vector from this new polygon that has been hit during the first sliding step. The response system would then call the detection phase again passing in this new (2^{nd}) slide vector and so on until the collision detection finally slides the entity into a position which does not cause an intersection.

12.2.4 Using Collision Ellipsoids

In the last section we learned that our application passes an ellipsoid radius vector into the collision system when it wishes to update the position of an entity in the game world. We know that the detection step will initially use both a broad and narrow phase to determine triangles which may cause obstructions along the path of the velocity vector. Once done, a new step will need to be taken. The narrow phase will transform each triangle it intends to test for intersection into something called *ellipsoid space*.

In ellipsoid space, the ellipsoid is essentially 'squashed' into a unit sphere and the surrounding triangles that are to be tested for intersection are also 'squashed' by that same amount. Therefore, after we have scaled any potentially colliding triangles into ellipsoid space, they now have the same size/ratio relationship with the unit sphere as they did with the ellipsoid before it was transformed. In this manner, our collision system can now perform intersection tests between a unit sphere and the transformed ellipsoid space geometry to efficiently find intersections. Once an intersection is found, the new unit sphere position can be transformed back into world space and returned to the application for final update of the moving entity's position.

To understand this concept of ellipsoid space, take another look at Figure 12.11. We stated earlier that the ellipsoid in this image was assumed to have a radius vector of (1.0, 4.0, 1.0). If we wished to convert this ellipsoid into a unit sphere (a sphere with a radius of 1.0) we would simply divide each component of its radius vector by itself:

Unit Sphere Radius $= \left(\frac{X}{X}, \frac{Y}{Y}, \frac{Z}{Z}\right)$

$$= \left(\frac{1.0}{1.0}, \frac{4.0}{4.0}, \frac{1.0}{1.0}\right)$$

$$= (1.0,\ 1.0,\ 1.0)$$

$= 1.0$ **Radius**

Now this may seem quite obvious because if we divide any number by itself we always get 1.0. However, as can be seen in Figure 12.12, if we scale the X, Y, and Z components of a triangle's vertices by the (reciprocal) components of the ellipsoid's radius vector as well, we essentially squash that triangle into the same space. This is a simple scaling transformation, just like any scaling transform we have encountered in the past. By applying the scale, we wind up bringing both the triangle and ellipsoid into a new shared space (coordinate system).

Note that as far as position and orientation are concerned, we do not consider the ellipsoid center to be the origin of the coordinate system or its radii to be the axes of the system. While it would certainly be possible to transform everything into the local space of the ellipsoid (i.e., use the ellipsoid as a standard frame of reference) it turns out to be an unnecessary step. The world origin and axes can continue to serve as the origin/orientation in this new space, but we will scale everything in that space by the same amount, thus preserving the initial (world) spatial relationships between the ellipsoid and the triangles. In that sense, all we have done here is scale the world coordinate axes using the (reciprocal) ellipsoid radii as our scaling factor. So, ellipsoid space is ultimately just a scaled world space.

In Figure 12.12 we can see that we have scaled the ellipsoid by a factor of 1.0 / 4.0 (its Y radius) along the Y axis and we have also scaled the surrounding geometry (the potential colliders) by the same amount. We can now safely perform intersection tests against the surrounding triangles with a unit sphere since both the triangles and the ellipsoid have been transformed into ellipsoid space. For example, imagine a quad comprising the front of a step that had a height of 6 (its top two vertices had a world space position of 6). We would simply divide the Y components of the vertices of the quad by 4, scaling the height of the step by the same factor that we scaled the height of the ellipsoid to turn it into a sphere. The height of the transformed step quad in ellipsoid space remains just as high proportionally to the unit sphere as the untransformed step quad was to the original ellipsoid.

Therefore, all our collision system will have to do before performing intersection tests between a given triangle and the unit sphere, is divide the components of each of the three triangle vertices by the components of the ellipsoid's radius vector:

```
Ellipsoid Space Position.x =  Center.x / Ellipsoid Radius.x;
Ellipsoid Space Position.y =  Center.y / Ellipsoid Radius.y;
Ellipsoid Space Position.z =  Center.z / Ellipsoid Radius.z;

Ellipsoid Space Velocity.x = Velocity.x / Ellipsoid Radius.x;
Ellipsoid Space Velocity.y = Velocity.y / Ellipsoid Radius.y;
Ellipsoid Space Velocity.z = Velocity.z / Ellipsoid Radius.z;

For ( each vertex in triangle )
{
        Ellipsoid Space Vertex.x = World Space Vertex.x / Ellipsoid Radius.x;
        Ellipsoid Space Vertex.y = World Space Vertex.y / Ellipsoid Radius.y;
        Ellipsoid Space Vertex.z = World Space Vertex.z / Ellipsoid Radius.z;
}
```

NOTE: In practice, we would likely calculate the reciprocal of the ellipsoid radii as a first step and then change all of the divisions into multiplications.

As can be seen in the above pseudo-code, it is very important that the center position of the ellipsoid and the velocity vector passed in by the application be transformed into ellipsoid space using the same method. We need to make sure that all of the positional information about to be used for the intersection testing (entity position, velocity and triangle vertex coordinates), exist in the same space. If we do not do this, the system will not function properly and we will generate incorrect results.

So, when our collision system is invoked by the application to update the position of a moving object, it will transform the input position and velocity vectors into ellipsoid space (from here on in called eSpace). This must take place prior to performing any per-polygon intersection testing. It is worth noting that the system will not use eSpace during the broad phase. Collection of potential triangle colliders will remain a world space operation (we will see this in the next chapter). However, once we have assembled the list of possible colliders, all of them will need their vertex positions transformed into eSpace. We do this by dividing the X, Y and Z components of each vertex by the corresponding radius component of the ellipsoid. The collision system will examine every triangle in the list and test for intersections with a unit sphere at the eSpace input position moving with the eSpace input velocity. Actual intersection positions are recorded along with a parametric distance along the velocity vector indicating when the intersection took place.

NOTE: The collision system cannot simply return after finding the first colliding triangle because the velocity vector may pass through multiple triangles if it is sufficiently long. The detection phase must test all potential colliders and only calculate the new position of the sphere after it has found the *closest intersecting triangle* (if one exists) along the velocity vector. This will be the first triangle that the sphere will hit and must be prevented from passing through.

This should give you some indication as to why the broad phase is so critical to the overall performance of the system. Without some means for quickly rejecting triangles which could not possibly be intersecting the ellipsoid along the velocity vector, our detection routine will have no choice but to test every single triangle in the environment. This will include the added cost of transforming all triangle vertices into eSpace prior to the intersection tests.

Finally, once the collision system is satisfied that it has found the new position, this position will need to be transformed from eSpace back into world space before being returned to the application for entity position update. The transformation of a position vector from eSpace back into world space can be done by simply reversing the transformation we performed earlier. Since dividing the components of a world space vector by the corresponding components of the ellipsoids radius vector transformed it into eSpace, multiplying the components of an eSpace position vector by the components of the ellipsoid's radius vector will transform it from eSpace to world space.

12.2.5 System Summary

We now have a good idea about the type of collision system components we are going to implement and the end results we should expect them to generate. As discussed, the performance enhancing broad phase component will be deferred until the next chapter so that we can focus on the core collision system in our first implementation.

The geometry database component of our CCollision class will be simple in our first implementation and will not require detailed discussion. The CCollision object will maintain an STL vector of triangle structures containing the three triangle indices, the triangle normal, and the material used by the triangle. A material in this case does not refer to a rendering material, but rather to a set of physical properties so that concepts like friction can be calculated. There will also be an STL vector of vertices. This will just be an array of the <X, Y, Z> position components since the collision geometry database has no use for additional vertex components like color or texture coordinates. Our system will provide a simple API for the purposes of adding triangle data to these arrays. The collision detection system will work exclusively with the geometry stored in these vectors, so the application must register any polygons it considers to be 'collidable' before collision testing begins (e.g., when the scene is created/loaded). Please refer to the workbook for details about the functions that add triangles to the collision database.

12.3 Collision Response

The response phase of the collision system is going to be very simple for us to implement; it requires only a few lines of code in its basic form. The collision detection step is where most of our work will occur and it is where we will find ourselves getting into some mathematics that we have not discussed before in this course series. Fortunately, you have likely already encountered most of this math in high school, so it should not be too difficult to follow. Our collision detection routines will do a fair amount of algebraic manipulation and will utilize the quadratic equation quite extensively. Students who may be a little rusty with these concepts need not worry. We have included a very detailed discussion of quadratics and will walk through the algebra one step at a time.

The collision detection phase is where we will focus most of our energies in this first section of the textbook. But before covering the detection phase code in detail, it will be useful to first look at the context in which it is used. We said earlier that the application need only call a single method of the CCollision object in order to update the position of any moving entity. This method is called

CollideEllipsoid. For now, just know that this single function is the heart of our collision system, called by the application to update the position of an entity in the scene. This function iteratively calls the detection phase and executes the response phase until a final position is resolved. We will have a look at this function in a moment and get a real feel for the flow of our system.

The code to the CollideEllipsoid method is quite simple to follow. The caller passes the position vector, radius vector and the velocity of the ellipsoid that is to have its position updated. It then calls a single collision detection function which completely wraps the detection phase of the collision system. This function will return either true on false indicating whether an intersection has happened along the path described by the velocity vector. If no collision has taken place, the CollideEllipsoid function will immediately return the new position to the application. This new position will be Position + Velocity, where both Position and Velocity are the vectors input to the routine by the application.

If an intersection does occur, then the collision detection function (called CCollision::EllipsoidIntersectScene) will return the new position for the object (in a non-intersecting location) and an intersection (slide plane) normal. CollideEllipsoid will then perform the response step and project any remaining velocity onto the slide plane before calling the EllipsoidIntersectScene function again to invoke another detection pass.

Before we look at the code to the CollideEllipsoid function, let us first examine the parameter list to the EllipsoidIntersectScene. Although we will not yet know how the detection phase works, as long as we know the parameters it takes and the values it should return, we will be able to look at the code to the rest of the collision system and understand what is happening.

The collision detection component of our system may sound like a complicated beast, but from a high level it is just a single method that is called by the collision system when a position update is requested. The prototype for the parent function that wraps the entire detection phase is shown below. This function is called by CCollision::CollideEllipsoid (our collision system interface) to determine if the path of the ellipsoid is clear. If not, it returns the furthest position that the ellipsoid can be moved to along the velocity vector such that it is not intersecting any geometry. If this function returns false then no intersection occurred and the entity can be safely moved to the position that was originally requested by the calling application (NewPos = Center + Velocity).

```
bool  CCollision::EllipsoidIntersectScene  (const D3DXVECTOR3& Center,
                                            const D3DXVECTOR3& Radius,
                                            const D3DXVECTOR3& Velocity,
                                            CollIntersect Intersections[],
                                            ULONG & IntersectionCount,
                                            bool bInputEllipsoidSpace = false,
                                            bool bReturnEllipsoidSpace = false );
```

We have quite a lot of work to do before we find out how this function (and the functions it calls) works, but a good place to start is by examining the parameter list and the way that it is used by the collision system in general.

const D3DXVECTOR3& Center
const D3DXVECTOR3& Radius
const D3DXVECTOR3& Velocity
These three parameters tell the detection phase the position of the center point of the ellipsoid we wish to move, its radius vector, and the magnitude and direction in which we wish to move it. The application passes this information into the parent function of the collision system (CCollision::CollideEllipsoid) which passes it straight through to the detection function (EllipsoidIntersectScene) via these parameters.

CollIntersect Intersections[]
The collision system passes in an array of CollIntersect structures. The CollIntersect structure is defined in CCollision.h and is contained in the CCollision namespace. It is the structure that the collision detection phase uses to transport intersection information back to the main collision system (for use in the response phase for example). This information will only be available if a collision occurred. The CollIntersect structure describes the intersection information for a single colliding triangle and is defined as:

```
struct CollIntersect
{
   D3DXVECTOR3 NewCenter;
   D3DXVECTOR3 IntersectPoint;
   D3DXVECTOR3 IntersectNormal;
   float       Interval;
   ULONG       TriangleIndex;
};
```

This structure's members are explained below.

D3DXVECTOR3 NewCenter
This member will contain the new position that the ellipsoid can be moved to without intersecting the triangle. This position can be used in the response phase or it can be passed back to the application (depending on your system's design needs) to update the position of the moving entity which the ellipsoid is approximating. In our system, we will not just return this new position back to the application if there is significant velocity left over beyond the point of intersection. In such cases, the response phase will create a new velocity vector from the remaining velocity by projecting it onto the sliding plane. The response phase will then call the EllipsoidIntersectScene function again to re-run the detection phase and test that the path described by the slide vector is also clear. This variable (NewCenter) will be the new position input as the first parameter in that next iteration of the detection phase. It essentially describes how far we managed to move along the velocity vector fed into the previous invocation of the detection phase.

D3DXVECTOR3 IntersectPoint
This vector will contain the actual point of intersection on the surface of the triangle. This will obviously also be a point on the surface of the ellipsoid (since they are in contact at the same physical point in space), so we now exactly what part of the ellipsoid skin is currently resting against the triangle we intersected with.

D3DXVECTOR3 IntersectNormal

This vector describes the normal at the point of intersection (described above). If the collision detection function determines that the ellipsoid collided somewhere in the interior of the triangle, this vector will simply store the normal of the triangle that was hit. If the ellipsoid collided with the edge of a triangle, then this will be a normal describing the direction to the center of the ellipsoid from the intersection point (returned above). Either way, this is the normal that describes to the response phase the orientation of the plane we wish to slide along if there is any left over velocity beyond the point of intersection. We will treat the IntersectPoint and the IntersectNormal as describing a sliding plane onto which the response phase will project any remaining velocity.

float Interval

This floating point member stores the *t* value for intersection along the velocity vector passed into the detection phase. As you will see in a moment when we start covering the mathematics of intersection, the *t* value describes the time (or distance, depending on context) along the velocity vector when the intersection occurred. The *t* value is in the range [0.0, 1.0] where a value of 0.0 would describe an intersection at the very start of the velocity vector and a value of 1.0 describes an intersection at the very end of the velocity vector. A value of 0.5 describes the collision as happening exactly halfway along the velocity vector. In effect, it describes the time of collision as a parametric percentage value. The point of intersection returned in the IntersectPoint member described above, is calculated as:

IntersectPoint = Center + (Velocity * Interval)

As our collision detection function does this calculation for us and conveniently returns the intersection point in a separate variable, there is no need for us to do this. But this is how we would do it if we were required to calculate the intersection point ourselves.

ULONG TriangleIndex

This value will store the index of the triangle in the collision system's triangle array (STL vector) that the collision has occurred with. This is often a useful piece of information to return to the caller, so we will include it in our structure.

ULONG & IntersectionCount

You will notice that our collision system passes an array of CollIntersect structures into the EllipsoidIntersectScene function to be filled with intersection information. This is because it is possible that when the new position is calculated, it is actually touching two or more triangles. This may seem strange at first when we consider that we are only interested in finding the first (i.e., closest) intersection time along the velocity vector, and indeed, our collision detection function will only return collision information for the closest impact. Ordinarily this is going to be a single triangle and only the first element in the passed CollIntersect array (Intersections[0]) will contain meaningful values on function return. However, it is possible that the ellipsoid could come into contact with multiple triangles at exactly the same time (see Figure 12.13).

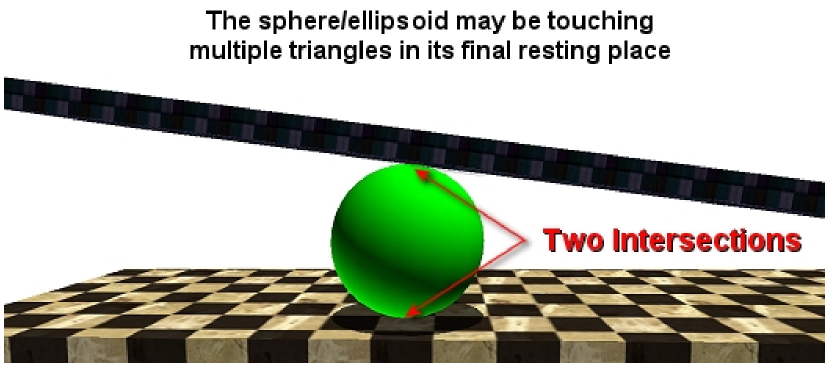

The sphere/ellipsoid may be touching multiple triangles in its final resting place
Two Intersections

Figure 12.13

As you can see in this image, the sphere is being moved into a wedge shaped geometry arrangement and as such, will intersect with both the floor and ceiling at exactly the same time. In such a case, the detection function would return two pieces of intersection information which have exactly the same time interval. That is, collision information will be stored in the first two elements of the array (Intersections[0] and Intersections[1]) describing the information discussed previously (triangle index, intersection point, etc.). Remember, this will only happen when both intersections happen at exactly the same time along the velocity vector since our collision detection function is only interested in returning collision information for the first time of impact.

The IntersectionCount parameter will contain the number of simultaneous intersections that have occurred. This describes the number of triangles that the detection phase has determined the ellipsoid will be touching in its final position. It obviously also describes the number of elements in the Intersections array that was passed into the detection function.

One thing might be concerning you at this point. When we discussed the CollIntersect structure, we said that it contains a variable called NewCenter which describes to the response step (and possibly the application) the new non-intersecting position of the ellipsoid along the velocity vector. If we have multiple CollIntersect structures returned, which one's NewCenter member should we use to update the position of the entity? As it happens, it does not matter, because our detection function will only return information for the closest intersection. If that intersection involves multiple triangles, the NewCenter vector in each structure will be exactly the same. If you take a look at Figure 12.13 you can see that whether the collision information for the floor or the ceiling is stored first in the intersection array, the same center position for the sphere will be contained in both items. This is still the furthest the sphere can move in the desired direction before the intersections occur. Therefore, it is the only position that can be returned for all intersections that may be happening at that first point(s) of contact along the velocity vector. So while the intersection points and normals will obviously be different, the sphere center position will always be the same amongst simultaneous colliders.

Thus, when the detection phase returns, we can always use the new position stored in the first intersection structure in the array for our response step. The detection phase returns the array of intersection information mostly out of courtesy. Our particular system implementation only needs the new position, so we need not return information about all intersections. However, this might be useful if you intend to extend the collision system later on. After all, we can certainly imagine a situation when it might be useful to know not only if a collision occurred, but have a list of the actual triangles that were hit (perhaps to apply a damage texture to them for example).

Now that we know what the EllipsoidIntersectScene function will expect as input and return as output, we can now examine the code to a CCollision::CollideEllipsoid function that implements the system we have discussed thus far. Remember, this is the only function that needs to be called by the application in order to update the position of a moving entity. It is also the function that contains all the code for the response step. The only code we will not see just yet is the code to the detection phase. This is tucked away inside the EllipsoidIntersectScene function whose parameter list we have just discussed. Understanding this function will require some study, but now we know how the detection function is expected to work with the rest of the system, so we can understand how everything else in the system works.

The CCollision::CollideEllipsoid function is shown below. This is a slightly more basic version than the one we will finally implement in the workbook. It has had all application specific tweaks removed so that we can concentrate on the core system. The function we are about to examine is really the entire collision system. We should understand everything about our system after studying this code except for how the actual detection phase determines intersections along the velocity vector. This will be discussed in a lot of detail later on in the chapter.

In this first section of code we are reminded that this function is called by the application and passed the position of the ellipsoid that is to be moved, the radius vector, and the velocity vector describing both the direction and distance of movement we desire. At the moment, all three use world space units. For the final parameters, the application passes a reference to two 3D vectors which on function return will contain the new position the application should move the entity to and its new velocity.

```
bool CCollision::CollideEllipsoid ( const D3DXVECTOR3& Center,
                                    const D3DXVECTOR3& Radius,
                                    const D3DXVECTOR3& Velocity,
                                    D3DXVECTOR3& NewCenter,
                                    D3DXVECTOR3& IntegrationVelocity )
{
    D3DXVECTOR3 vecOutputPos, vecOutputVelocity, InvRadius, vecNormal
    D3DXVECTOR3 eVelocity, eInputVelocity, eFrom, eTo;
    ULONG       IntersectionCount, i;
    float       fDistance;
    bool        bHit = false;

    vecOutputPos       = Center + Velocity;
    vecOutputVelocity  = Velocity;

    // Store ellipsoid transformation values
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );
```

The first thing we do is add the velocity vector to the position passed in and store the result in the vecOutputPos local variable. If no intersections occur then this describes the new position that should be returned to the application. We calculate it here so that if no intersections occur we can simply return it without any additional work. If intersections do occur then this value will be overwritten with the final position determined by the detection and response steps. We also copy the input velocity vector into the local variable eOutputVelocity. This vector will be used to store the integration velocity that will be returned to the application.

Next we create a vector called InvRadius where each component equals the reciprocal of the matching component in the ellipsoid's radius vector. We will need to convert the information (Center, Velocity, etc.) into eSpace by dividing each vector component by the matching component in the radius vector. Because we will need to do this quite a bit (especially in the detection phase where it will be done for each triangle vertex), we create a vector that when component multiplied with another vector will perform this division. Remember, A / B = A * (1 / B) and therefore, if we create a vector that has the components, 1/Radius.x, 1/Radius.y, and 1/Radius.z, we have a vector that scales by the radius of the ellipsoid to transform points into eSpace.

In the next section of code we will see this transformation being done. As discussed, the detection phase wants to work in eSpace so we must send it the ellipsoid position and its velocity in eSpace, not world space. We accomplish this by multiplying the world space vectors (Velocity and Center) by the inverse radius vector we just computed:

```
    // Calculate values in ellipsoid space
    eVelocity = Vec3VecScale( Velocity, InvRadius );
    eFrom     = Vec3VecScale( Center, InvRadius );
    eTo       = eFrom + eVelocity;

    // Store the input velocity for jump testing
    eInputVelocity = eVelocity;
```

Notice that we store the eSpace velocity vector in eVelocity and the starting position of the ellipsoid in eSpace in the eFrom vector. We also calculate the eSpace position we would like to move the sphere to by adding eVelocity to the eFrom point. If no collisions occur, this is exactly where we would like our sphere to end up in eSpace after travelling the full extent of the velocity vector. We need to know this position in order to calculate any left over velocity in the event of an intersection. This remaining velocity can then be projected onto the slide plane as will be seen in a moment. A copy of the initial velocity vector (in eSpace) is stored in the eInputVelocity local variable. You will see in a moment how this will be used if a satisfactory position cannot be found by the system in the requested number of slide iterations. When this is the case, we will call the detection phase one final time, passing in the initial velocity and simply return the position generated to the application. We need a backup of the initial velocity vector passed into the function in order to do this since the eVelocity vector will be continually overwritten in the response step with a new slide vector (to pass it back into the detection function in the next iteration).

In the above code, we use a helper function called Vec3VecScale which performs a per-component multiply between two vectors. Its code is a CCollision method and implemented inline as shown below.

```
inline static D3DXVECTOR3 Vec3VecScale (const D3DXVECTOR3&v1,const D3DXVECTOR3&v2 )
             { return D3DXVECTOR3( v1.x * v2.x, v1.y * v2.y, v1.z * v2.z ); }
```

In the next section of code we enter the actual collision detection and response phase. As discussed previously, there are times when the ellipsoid could get stuck in a sliding loop before the velocity vector is projected away to nothing. We control the maximum number of times the response phase will call the detection phase using a member variable called m_nMaxIterations. In virtually all cases, the new position of the ellipsoid will be found after only a few iterations (a few slides) at most. However, by putting this limit on the number of iterations the collision system will run, we can control the case where the ellipsoid is sliding repeatedly back and forth between two or more surfaces.

As you can see in the next code snippet, the maximum number of iterations is governed by a for loop. Inside this loop, the detection and response phases will be executed per iteration. If, during any iteration, the final position of the ellipsoid is determined (i.e., there is no remaining velocity vector), we can exit the loop and return this position back to the caller. If the loop finishes all iterations, it means a position could not be determined in the maximum number of iterations such that there was no remaining velocity. In this (rare) case an additional step will have to be taken that we will see at the end of the function. Let us have a look at the first section of the loop:

```
    // Keep testing until we hit our max iteration limit
    for ( i = 0; i < m_nMaxIterations; ++i )
    {
        // Break out if our velocity is too small
        if ( D3DXVec3Length( &eVelocity ) < 1e-5f ) break;

        // Attempt scene intersection
        // Note : We are working totally in ellipsoid space at this point
        if ( EllipsoidIntersectScene( eFrom, Radius, eVelocity,
                                      m_pIntersections, IntersectionCount) )
        {
```

Inside the loop we first test to see if the currently length of eVelocity is zero (with tolerance). If so, we do not have to move the ellipsoid any further and we can exit from the loop. We know at this point that the eTo local variable will contain the current position the entity should be moved to and can be returned to the caller. If this is the first iteration of the loop, then eVelocity will contain the initial velocity vector passed in by the application (in eSpace) and eTo will contain the desired destination position as calculated above (eFrom + eVelocity). However, if this is not the first iteration of the loop then eVelocity will contain the slide vector that was calculated in the last response step (performed at the bottom of this loop). If this is zero, then the response step projected the remaining velocity from a previous detection step onto the sliding plane and created a new eVelocity vector with such insignificant length that it can be ignored. In this case, we can still exit from the loop since the eTo local variable will contain the eSpace position of the ellipsoid returned from the previous call to the detection function.

Next, in the above code, we feed the eSpace position and velocity into the collision detection function EllipsoidIntersectScene. Notice that we also pass in the radius vector of our ellipsoid which was passed in by the caller. Why do we need to do this if we are already passing in the position and velocity vectors in eSpace (where the ellipsoid is a bounding sphere)? Remember that the detection function will need to transform the vertices of each triangle it tests for intersection into eSpace also, and for that, it will need to divide the components of each vertex of each triangle by the radius of the ellipsoid. The detection function is also passed in an array of CollIntersect structures (this array is allocated in the CCollision constructor to a maximum size of 100) which on function return will contain the intersection information for the closest collision between the unit sphere and the environment (in eSpace). The variable passed by reference as the final parameter will contain the number of triangles that have been simultaneously intersected and thus, the number of elements in the m_pIntersections array that contain meaningful information.

The next few sections of code demonstrate what happens when the EllipsoidIntersectScene function returns true (i.e., an intersection has been determined at some point along the velocity vector). This means the response phase needs to be executed. Since we talked about the response phase as being a separate component of the collision system, you might have assumed it would be a large and complex function. But in fact, the code executed in this next block comprises the entire response phase of the system we have discussed thus far.

```
            // Retrieve the first collision intersections
            CollIntersect & FirstIntersect = m_pIntersections[0];

            // Set the sphere position to the collision position
            // for the next iteration of testing
            eFrom = FirstIntersect.NewCenter;

            // Project the end of the velocity vector onto the collision plane
            // (at the sphere centre)
            fDistance = D3DXVec3Dot( &( eTo - FirstIntersect.NewCenter ),
                                     &FirstIntersect.IntersectNormal );

            eTo -= FirstIntersect.IntersectNormal * fDistance;
```

In the above code we alias the first element in the returned m_pIntersections array with the local variable FirstIntersect for ease of access. Although multiple elements may exist in this array containing the information for several triangles that may have been intersected, all of these triangles will have been intersected at the same time along the velocity vector and as such, each element in this array will contain the same eSpace sphere position in its NewCenter member. Remember, this is the furthest position the detection phase could theoretically move the sphere along the velocity vector before it intersects one or more triangles. Since we are only interested in getting the new sphere position, we can just extract this information from the first element in the array.

Notice how we copy the new position of the sphere into the eFrom vector, which previously described the position we wish to move our sphere from prior to the detection function being called. However, because the response step is about to be executed, this position is going to be the new 'from' position when the detection function is called again in the next iteration of the loop. The velocity vector eVelocity will also be overwritten with the new slide vector that will be generated. Therefore, it is as if we are saying, "Let us now run the collision detection function again, starting from this new position. We will test to see if we can slide along the new velocity vector (the slide vector) the next time an iteration of this loop is executed."

In the next line of code (shown above) we project any remaining velocity onto the sliding plane. eTo currently describes the position we wanted to reach before the detection function was called. The NewCenter member of the first CollIntersect structure contains the position we are able to move to along the velocity vector. If we subtract the new position vector from the intended destination, we get the remaining velocity vector for the sphere. If we perform a dot product between this vector and the slide plane normal (returned from the detection phase also), we get the distance from the original eTo position (Original Dest in Figure 12.14) that we wanted to travel to, to the center of the sphere along the plane normal (D in Figure 12.14). That is, if the remaining velocity vector forms the hypotenuse of a right angled triangle and the new position of the sphere describes the point at which the hypotenuse and the opposite side of the triangle meet, dotting the vector **eTo - NewCenter** (Original Dest – Modified Dest in Figure 12.14) with the intersection plane normal, describes the length of the adjacent side of the triangle. As you can see in the final line of the code shown above, once we have calculated this distance, we can simply move eTo back along the plane normal (negative direction) so that it describes the position where the opposite side meets the adjacent side (the blue arrow in Figure 12.14). This is a vector that is situated in the direction tangent to the slide plane that our sphere should slide along. Therefore, if we subtract this new eTo position from the new position of our sphere returned by the detection function, we have a new velocity vector describing the direction and magnitude we wish to slide along the plane when the collision detection phase is called in the next iteration of the loop.

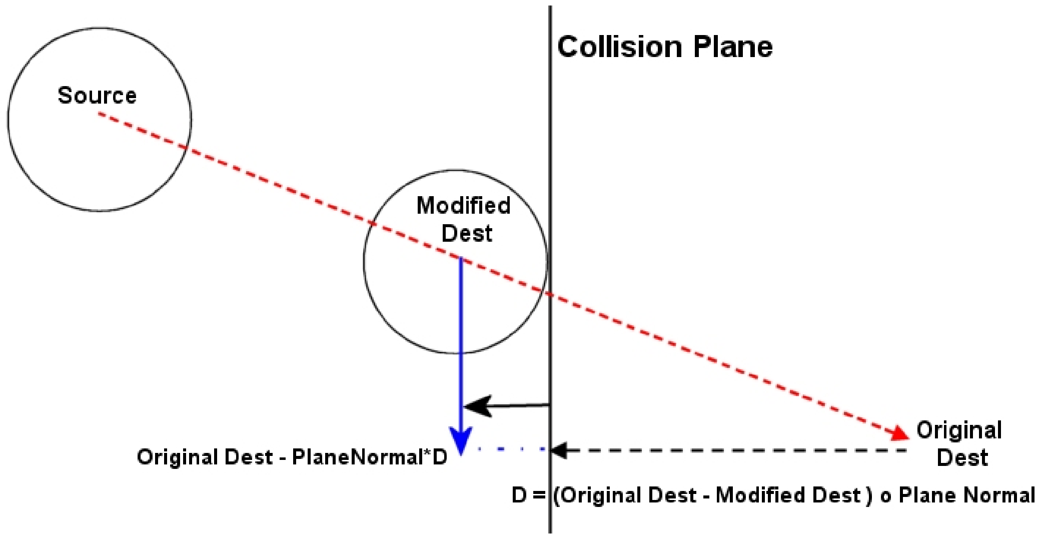

Collision Plane
Source
Modified Dest
Original Dest
Original Dest - PlaneNormal*D
D = (Original Dest - Modified Dest) o Plane Normal

Figure 12.14

In the next section of code you will see how we calculate the eVelocity vector for the next iteration of the detection phase by doing what we just described -- subtracting the new position of the sphere from the projection of the remaining velocity vector onto the slide plane.

```
             // Update the velocity value
            eVelocity = eTo - eFrom;

            // Calculate new integration velocity
            fDistance = D3DXVec3Dot( &vecOutputVelocity,
                                     &FirstIntersect.IntersectNormal );
            vecOutputVelocity -= FirstIntersect.IntersectNormal * fDistance;

            // We hit something
            bHit = true;
```

In the above code we also project the current integration velocity (which in the first iteration will just be the input velocity) onto the slide plane. Notice that this is done in almost an identical way to the previous projection except that we project the whole velocity vector and not just the remaining portion behind the point of intersection. That is, we imagine that vecOutputVelocity describes a point relative to the origin of the coordinate system, and the slide plane normal passes through the origin. The dot product will produce fDistance which is the distance from that point to the plane along the plane normal. We then move the point (vecOutputVelocity) onto the plane by moving it towards the direction of the plane for a distance that places that point on the plane. This is our new integration velocity unless it gets changed in the next iteration. As you can see, this vector is continually projected onto the new slide plane in its entirety to preserve as much of the original velocity vector's length as possible.

Notice that in the above code we also set the local Boolean variable bHit to true so that we can tell, at the bottom of the function, whether at least one collision occurred. We will see how this is used in a moment.

Take a moment to study what we have done above. To reiterate, we have recalculated the eVelocity vector to store the slide vector, the eFrom vector now contains the new modified position of the sphere butted up against the colliding polygon. We have also updated the eTo vector describing the position we wish to travel to in the next iteration of the loop. We have essentially overwritten all the inputs that will be fed into the next iteration of the detection function. As far as the detection function is concerned, in the next iteration, these values could have been passed in by the application. They have of course been calculated by the response step to force a new movement into another location along the slide vector. The integration velocity is calculated using the same projection, but the entire vector is projected, not just the remaining velocity. Remember, this is the new velocity vector that will be returned to the caller, so we do not want this to be whittled away to zero.

We have essentially just covered the entire response step. There is just one additional line we put in to help reduce jittering that might occur when the player is trying to force the entity into an internal corner as shown in Figure 12.15.

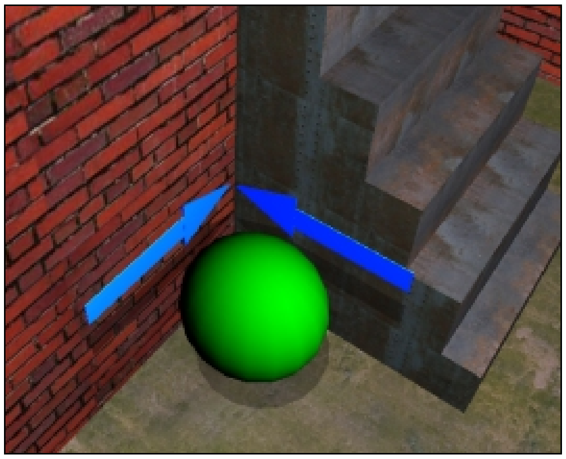


Figure 12.15

If the entity is forced into an internal corner or lip where polygons meet at an angle of 90 degrees or less (see Figure 12.15) a vibration artifact can occur until the initial velocity vector is completely spent. We might imagine that the sphere in Figure 12.15 first hits the wall in the corner which the response step slides into the corner and then into the side of the steps. When the sphere intersects with the side of the steps in the next detection step, it gets slid back into the wall it just slid along causing it to slide back in the opposite direction. When the sphere hits the wall again in the next detection phase, the response phase once again generates the original slide vector pushing it back into the side of the steps again, and so on. The sphere would seem to bounce between the two faces that form the corner in a cyclical pattern producing an unattractive vibration. To help resolve this, the response step can test if the slide vector (now stored in eVelocity ready for the next iteration) is pointing in the opposite direction to the original velocity vector that was passed in. That is, we can detect whether we are generating a slide vector that forms an angle greater than 90 degrees with respect to the original velocity vector that was first input by the application. If so, then we know that we are trying to move the sphere in a direction opposite of what was originally intended and have successfully detected where such a pattern of bouncing might be about to be carried out. When this is the case, we can simply use the current sphere position (returned from the last call of the detection function) as the final resting place of the sphere and exit the loop. The final line of code for the response step is shown below and performs this test.

```
            // Filter 'Impulse' jumps
            if ( D3DXVec3Dot( &eVelocity, &eInputVelocity ) < 0 )
                { eTo = eFrom; break; }
```

```
        } // End if we got some intersections
        else
        {
            // We found no collisions, so break out of the loop
            break;
        } // End if no collision

    } // Next Iteration
```

Studying the above code you can now see why we originally made a backup of the original velocity vector in eSpace (eInputVelocity) at the start of the function -- so we can filter out any vibration artifacts. And that is basically all there is to our main collision detection and response loop.

In the above code we also show the 'else' code block. This is executed when no collisions between the velocity vector and the scene are detected (i.e., EllipsoidIntersectScene returns false). In such a case, we break from the loop immediately since we already have the final resting place of the sphere in eSpace in the eTo variable. Either way, eTo will always contain the final resting place of the sphere in eSpace when this loop is exited. Notice that when we detect a looping pattern in the bottom of the response step in the above code, we copy the current position stored in eFrom to the eTo variable. Remember that this variable contains the new sphere position that was returned from the previous call to the detection function. Since we have discovered a looping pattern, we will decide to stop doing any further response processing and set this as the final resting place of the sphere.

At this point we have exited the loop and one of two conditions will be true; we will have exited the loop prematurely (the usual case) and we have the final resting position of the sphere in eSpace stored in the eTo variable, or the loop will have exited naturally after its full number of iterations. The second case is bad news since it means a very rare case has arisen where we were unable to spend the entire velocity in the maximum number of iterations to resolve a final resting place for the sphere. As discussed earlier, this can happen very infrequently when the geometry forms an angle that causes the sphere to repeatedly slide between opposing surfaces without significantly diminishing the velocity vector each time it is projected onto the sliding plane. If the maximum number of iterations was executed, then we have decided we no longer wish to spend any more processing time sliding this sphere around.

In the first case, we have our final position in eTo, so all we have to do is transform it back into world space. Just as we transform a position vector into eSpace by dividing its components by the radius vector of the ellipse, we transform an eSpace vector back into world space by multiplying its vector components by the radius vector of the ellipse (see below). This code is only executed if the loop variable is smaller than the maximum number of iterations (i.e., a final resting place for the sphere was resolved in the given number of loops and the loop exited prematurely). The final world space position for the ellipsoid is stored in the local variable vecOutputPos.

```
    // Did we register any intersection at all?
    if ( bHit )
    {
        // Did we finish neatly or not?
        if ( i < m_nMaxIterations )
        {
            // Return our final position in world space
            vecOutputPos = Vec3VecScale( eTo, Radius );

        } // End if in clear space
```

When this is not the case and we have executed the maximum number of iterations of the loop, we have to 'hack' around the problem. We will simply ignore everything we have done thus far and simply call the collision detection routine one more time with the original eSpace values. We will then take the position returned by the detection function (first point of impact along the original velocity vector) and return that as the final position of the sphere.

```
        else
        {
            // Just find the closest intersection
            eFrom = Vec3VecScale( Center, InvRadius );

            // Attempt to intersect the scene
            IntersectionCount = 0;
            EllipsoidIntersectScene( eFrom, Radius, eInputVelocity,
                                     m_pIntersections, IntersectionCount )
            vecOutputPos = Vec3VecScale( m_pIntersections[0].NewCenter, Radius );

        } // End if bad situation
    } // End if intersection found
```

As you can see in the above code, we reset the eFrom vector to be the eSpace version of the original position vector passed into the function and run the detection routine again. We then transform the intersection position into world space before storing it in vecOutputPos.

Our job is now done. Whatever the outcome, we now have the position the ellipsoid should be moved to stored in the local variable vecOutputPos and the new integration velocity vector stored in vecOutputVelocity. As a final step we will copy these values into the NewCenter and IntegrationVelocity variables which were passed by reference by the caller. The application will now have access to this new position and velocity and can use it to update the position of the entity which the ellipsoid is approximating. We then return either 'true' or 'false' to indicate whether any intersection and path correction took place.

```
    // Store the resulting output values
    NewCenter           = vecOutputPos;
    IntegrationVelocity = vecOutputVelocity;

    // Return hit code
    return bHit;
}
```

And there we have it. With the exception of our examination of the EllipsoidIntersectScene method which actually calculates the new non-intersecting position of the unit sphere, we have walked through the entire collision system. Hopefully you will find this to be relatively straightforward, especially the response step which, while very effective, was actually quite small.

Note: If you are comparing the code described above with the version in Lab Project 12.1's source code, you will likely notice that the version in the source code looks much more complicated and quite different in places. This is because the version supplied with Lab Project 12.1 has had a lot of extras added, such as the ability to provide collision detection and response against moving meshes in the environment (lifts, automated doors, etc). At the moment, we are just concentrating on the core system that deals with a static collision environment. As we progress through this chapter and the more advanced features are added to our collision system, you will see the code snippets start to look more like those in the accompanying lab project.

Of course, there is still a large portion of the collision system that requires explanation. We have not yet discussed how the EllipsoidIntersectScene function determines intersections with the sphere along its velocity vector and how it returns the new position of the sphere at the point of intersection. Moving forward, we will begin to focus on the implementation of the collision detection phase (i.e., EllipsoidIntersectScene and its helper functions).

12.4 Collision Detection

Our collision detection phase will be responsible for testing the movement of a sphere along a velocity vector and determining the distance that the sphere can travel along that vector before an intersection occurs between the surface of the sphere and the environment. The collision detection phase will comprise several intersection tests between rays and various geometric primitive types. While it might not be obvious why this is the case at the moment, we will need to cover how to intersect a ray with a triangle, a cylinder, and a sphere. These intersections tests will be performed one after another to determine whether the path is clear. Only when all intersection tests fail for every potential collider do we know that the velocity vector is clear to move the sphere along.

Also, several of the intersection tests may return different collision times along the velocity vector for a given triangle. This is because each uses a different technique to test the different parts of a triangle with the sphere. The edges, the vertices, and the interior of each triangle will all need to be tested separately to make sure they do not intersect the sphere at some point along its velocity vector. If several of the tests return different intersection positions along the velocity vector for a single triangle, our collision system will only be interesting in recording the intersection information with the smallest *t* value (the first collision along the path). In other words, we are only interesting in finding the furthest point the center of the sphere can move to along the velocity vector until its first impact with the environment occurs.

The detection process is complex because our sphere is (theoretically) moving along the velocity vector. Simple tests that could be performed with a static sphere cannot be used here because our entity is assumed to be in motion. We cannot perform a sphere/triangle test at both the start and end points of the velocity vector since that would ignore collisions that might have happened between the endpoints. We can easily imagine a situation where a sphere starts off in front of a triangle and is not intersecting it. Assume that we wish to move to a position (given a velocity vector) completely behind the triangle, which we know is also non-intersecting. Intuitively we see that in order for the sphere to get from one side of the triangle to the other, it must pass right through the triangle and thus must collide with it at some point along that path. Simply testing the start and end positions of the sphere would prove negative for intersection, which might lead us to believe the sphere can indeed be moved across the full extent of the velocity vector to its desired destination.

Solving this problem requires that we account for the time aspect of the velocity vector. We will use a technique referred to as swept sphere collision detection, which accounts for the full travel path, including endpoints. A swept sphere is shown in Figure 12.16 and as you can see, it is like we have stretched (i.e., swept) the sphere along its velocity vector to create a capsule shape. Source describes the initial center position of the sphere and Dest describes the desired destination for the center of the sphere (Center + Velocity). In between, we see a solid area representing the integration of the sphere at each moment in time in which the sphere is traveling from origin to destination.

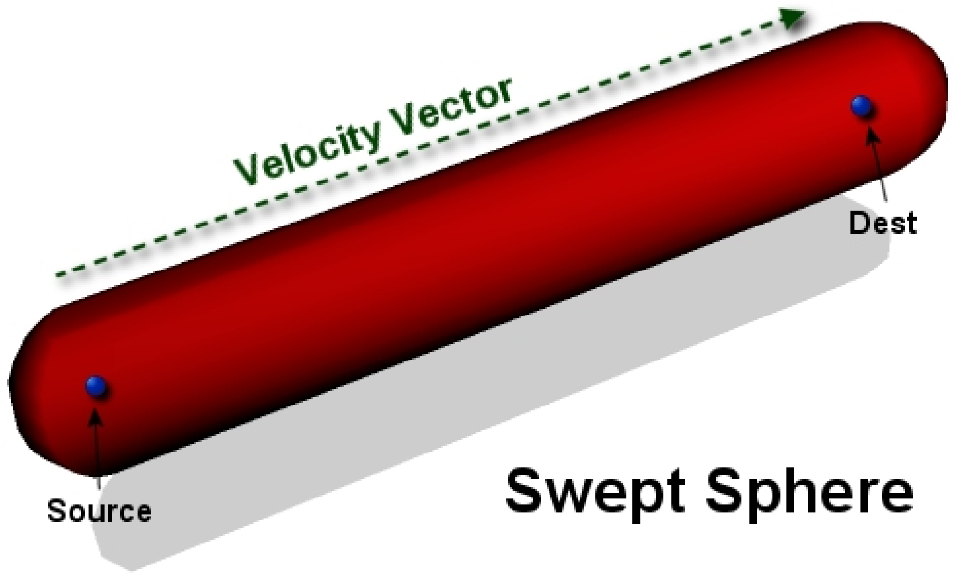


Figure 12.16

We know that if any geometry intersects the swept sphere then it must intersect the sphere at some point along the velocity vector. It is as if we have thickened the velocity vector ray by the radius of the sphere and are now testing this against the scene instead of the sphere itself. It covers all the space in the environment that the sphere will touch when moving along the velocity vector.

When we visualize the swept sphere as a primitive (as shown above) we can see that it bounds all the space that will be touched by the sphere on its path along the velocity vector. While this is vaguely interesting we still have not covered how to intersect such a shape with the environment. As you will see in a moment, we can greatly simplify this process by performing a transformation on any triangle that is to be tested against the swept sphere. Instead of creating a capsule shape as shown in Figure 12.6 describing the velocity vector thickened by the sphere radius, we can approach this problem from the other angle. We will see later that if we instead 'inflate' the geometry of our environment by the radius of the sphere, we can treat the sphere's movement along the velocity vector, not as a capsule, but as a simple ray. We can test for intersections between this ray and the 'inflated' environment to determine the point at which the sphere surface intersects the environment. This technique will be used for testing the sphere's path against the center of the polygon, the polygon edges, and the polygon vertices. It should be clear then that our collision detection techniques will essentially be a collection of methods that intersect a ray with other geometric primitive types. Therefore, before we try to understand the collision detection system as a whole, let us spend some time defining what a ray is and look at some of the various intersection techniques we can perform with rays. Once we understand how to intersect a ray with several different primitive types, we will then see how these intersection techniques can be assembled to create our core collision detection system.

12.4.1 Rays

In this section we will discuss rays and how they can be represented and used to perform intersection tests. It is important that we initially get our terminology right because in classical geometry there is a distinct difference between a line, a line segment, and a ray.

- A **Line** has no start and end point and is assumed to extend infinitely in both directions along a given slope.
- A **Line Segment** is a finite portion of a line in that it has a start and an end position in the world. To visualize a line segment we would draw a mental line between the two position vectors describing the start and end points. Thus, it is quite common for a line segment to be represented by two positional vectors.
- A **Ray** has a starting position (an origin) and a direction. It is assumed to extend infinitely in the direction it is pointing. We can think of a Ray as being one half of a line. Whereas a line would extend infinitely from the starting position of the ray in both the positive and negative directions, a Ray will start at some origin and extend infinitely in one direction only -- the direction the ray is pointing.

Figure 12.17 demonstrates the differences between these three geometric primitives.

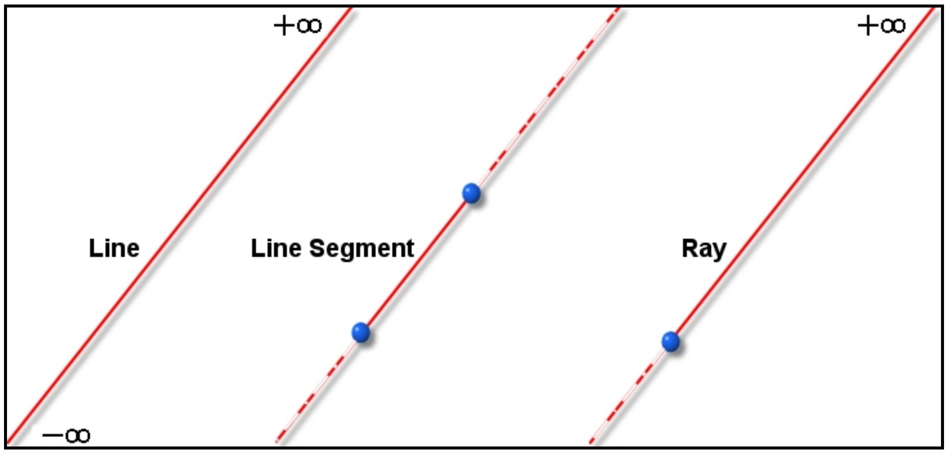


Figure 12.17 : A Line, A Line Segment and a Ray

In computer science, we will sometimes take some liberties with these definitions, and this is especially true in the case of a ray. While it is often useful to think of a ray has having a starting position and a direction in which the ray extends infinitely, it is also useful to use a ray definition that allows for a finite length from the origin. If we think about the velocity vector passed into our collision routines, we can think of this as being a ray of finite length when coupled with the center position of the sphere. The sphere center point describes the origin of the ray and the velocity vector describes the direction of the ray and its magnitude. This is the ray representation we will be using most in this chapter and throughout our ray intersection routines. Since we can add the ray's direction vector to its origin to get the end point of the ray, we can see that our ray is really just a directed line segment. In other words, it is a line segment that is described not as two position vectors, but as a single position vector (the ray origin) and a vector describing the direction the ray is traveling from the origin and the distance it travels in that direction. Therefore, in our discussions we will use the following definition:

A **Ray** is a directed line segment.

Let us now take a look at how we will define our ray. A 3D ray is often specified in its parametric form as shown below using three functions.

Parametric Form of a Ray (3D)

$$x(t) = x_o + t\Delta x$$
$$y(t) = y_o + t\Delta y$$
$$z(t) = z_o + t\Delta z$$

where

$(x_o \;\; y_o \;\; z_o)$	=	**Ray Origin (Start Position)**
$(\Delta x \;\; \Delta y \;\; \Delta z)$	=	**Direction (Delta) Vector**

The parameter to each of these functions is *t*, where *t* = [0.0, 1.0]. The combined results of each function will form a vector that describes a position somewhere along the ray where *t* is between 0.0 and 1.0. Note that *t* is the same input to each function. Each function shown above is simply describing a movement along the direction vector (Δi) from the ray origin (i_o) for each component x, y, and z. The result of each component calculated can then be linearly combined into the final position vector (x, y, z) describing the point along the ray given by *t*. It is often nicer to represent a 3D ray parametrically using its more compact vector form:

$$p(t) = p_o + td$$

where

p_o = A 3D vector describing the position of the start of the ray (the origin)

d = A non-unit length 3D vector (or a unit length vector if you wish the length of the ray to be 1.0) describing the direction and length of the ray.

We can describe any position on that ray by specifying a function input value of *t* between 0.0 and 1.0. For example, imagine that the ray origin **p** is a vector describing a ray origin <100, 100, 100>. Also imagine that the direction and magnitude of the ray described in vector **d** sets the end of the ray at a position offset 10 units along the X, Y and Z axes from the ray origin. That is, <10, 10, 10>. To calculate a position exactly halfway along that ray we can use a *t* value of 0.5, as shown below.

$$\begin{aligned}
\textbf{Position} \quad &= \mathbf{(\,100, 100, 100\,)} \quad + \quad \mathbf{t \quad (\,10, 10, 10\,)}\\
&= \mathbf{(\,100, 100, 100\,)} \quad + \quad \mathbf{0.5\,(10, 10, 10\,)}\\
&= \mathbf{(\,100, 100, 100\,)} \quad + \quad \mathbf{(\,5, 5, 5\,)}\\
&= \mathbf{(\,105, 105, 105\,)}
\end{aligned}$$

Using a *t* parameter of 0.5, we have correctly calculated that the exact position of a point situated exactly halfway along the ray is (105, 105, 105).

In Figure 12.8 we see the same ray and the calculation of other points along the ray. We can see for example that a *t* value of 0.25 gives us a position situated ¼ of the way along the ray from the ray origin. We can also see that using a *t* value of 0.75 provides us with a position situated exactly ¾ of the length of the ray from the origin. Finally, we can see that the end point of the ray can be calculated using a *t* value of 1.0. As the function for calculating the end position (*t* = 1) will simply scale the ray direction vector by 1.0, we can omit the *t* value altogether when we wish to calculate the end of the ray by simply adding the direction vector to the ray origin:

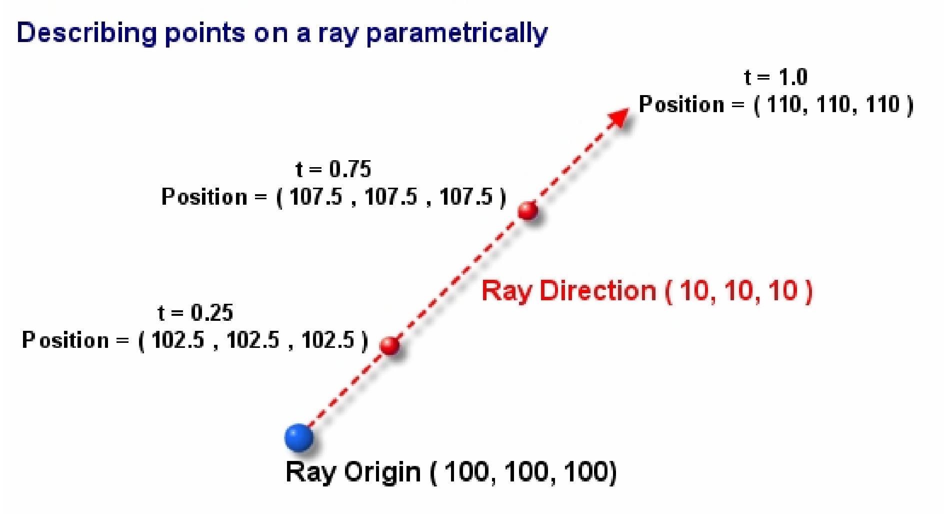

Describing points on a ray parametrically
t = 1.0
Position = (110, 110, 110)
t = 0.75
Position = (107.5 , 107.5 , 107.5)
Ray Direction (10, 10, 10)
t = 0.25
Position = (102.5 , 102.5 , 102.5)
Ray Origin (100, 100, 100)

Figure 12.18

$$\begin{aligned}
\textbf{Position} \quad &= \mathbf{(\,100, 100, 100\,)} \quad + \quad \mathbf{1.0 \quad (\,10, 10, 10\,)}\\
&= \mathbf{(\,100, 100, 100\,)} \quad + \quad \mathbf{(10, 10, 10\,)}\\
&= \mathbf{(\,110, 110, 110\,)}
\end{aligned}$$

Alternatively, we can think of *t* as describing how many units we would like to move along the ray measured in units of 'ray length'. A value of 1.0 means we would like to travel one whole ray length and a value of 0.5 would describe us as wanting to move along the ray ½ of the ray length. Even when representing a ray in its classical geometry form, this same relationship holds true, although it may not be immediately obvious. The classical geometry representation of a ray is that it has a position and a direction and an infinite length. Such a ray definition can be represented in the same parametric vector form we have been using thus far:

$$p(t) = p_o + td$$

The only difference between this representation and the one we have been using is that it is implied that ***d*** is a unit length vector. It describes a direction only, but no magnitude since the length of the ray is infinite (controlled by *t* which in this case has an infinite range). When using such a definition we will usually have some other variable that determines legal values for points along the ray. For example, we might have a variable *k* that contains the length of the ray. Positional vectors on the ray that have a distance from the origin that is larger than *k* are considered to be beyond the legal end point of the ray.

When using the classical representation of a ray, the *t* value still essentially means the same thing -- it describes the number of units (in vector length) we would like to move along the ray from the origin. However, with the classical definition, the *t* value will not be restricted between 0.0 and 1.0, but rather between 0.0 and *k*, where *k* is some arbitrary value used to describe the length of the ray. Using the classical definition we would use a *t* value to describe a position exactly *t* units along the infinite ray. A value of *t* = 7 for example would describe a point along the ray seven units away from the origin, when in the case of the classical definition, the units of measurement are the units of the coordinate system for which it is defined (e.g., world space). Obviously, this seems quite different from the ray definition we have been using thus far, where units are based on 'ray length'. However, if we think about the classical ray representation where a unit length direction vector is used, we could hold the view that instead of the ray being infinite, it has a length of 1 (because the direction vector ***d*** is unit length). In this case, we can see that a value of *t* = 7 does indeed still describe a position along the ray measured in units of 'ray length'. In the classical case, ray length is 1.0, so the mapping between the ray length and the units of the coordinate system for which it is defined is 1:1.

Although we will be using the definition of a ray which involves a delta vector, that does not mean that we cannot create a ray in this form from information stored in another form. Very often we may not have the delta vector at hand and may have only the start and end points that we would like the build the ray between (i.e. the line segment). That is not a problem since we can create the delta value by subtracting the start position from the end position. The resulting vector ***d*** will describe the direction from the first vector to the second and will have the length of that ray encoded as its magnitude. The example below shows how easy it is to calculate the delta vector for our ray when the initial information is in the form of two positional vectors:

RayStart	**=**	**(100, 100, 100)**
RayEnd	**=**	**(110, 110, 100)**
d	**=**	**RayEnd – RayStart = (10, 10, 10)**

Exactly the same technique can be used to create the unit length direction vector for the classical geometry representation of the ray. An additional step is required to normalize vector ***d***. The following example shows how we could create a ray in classical form and then use an additional variable *k* to describe the length of the ray. Any *t* value larger than *k* would be considered past the end of the ray.

RayStart	=	**(100, 100, 100)**		
RayEnd	=	**(110, 110, 100)**		
d	=	**RayEnd - RayStart = (10, 10, 10)**		
d	=	**Normalized (d)**		
k	=	$\sqrt{10^2 + 10^2 + 10^2}$	=	**17.3205**

We now know what a ray is and how we can represent one using two vectors (a ray origin and a direction). Because we will be using the non-classical geometry definition of a ray, our direction vector will also have magnitude and will be referred to as a delta vector. When using delta vectors, points on the ray can be described parametrically using *t* values between 0.0 and 1.0. Any *t* value that is smaller than zero is said to exist on the same line as the ray, but exists before the ray origin. Any *t* values that are larger than 1.0 describe points that exist on the same line as the ray but after the ray end point.

With the description of a ray behind us, let us now move on and look at some intersection tests that can be performed using rays. This will help us uncover why rays are so useful in 3D computer graphics (and in particular, in our collision detection system).

12.4.2 Ray / Plane Intersection Testing

It only takes a small amount of experience to realize that the dot product is the veritable Swiss Army Knife of 3D graphics programming. It is used to perform so many common tasks in 3D graphics programming that it would be impossible to make a 3D game without it. The dot product comes to the rescue again when trying to determine whether a ray intersects a plane. The process is remarkably simple as long as you understand how the dot product operation works.

We have discussed several times in the past (see Graphics Programming Module I) the properties of the dot product and we even examined ray intersections with a plane in our workbooks. We approached the understanding of this technique by analyzing the geometry and applying the properties of the dot product to solve the problem. But this method is not always ideal when handling more complex intersections since drawing diagrams to show geometric solutions can become a very ungainly process.

It is generally much simpler to come up with a pure mathematical solution. That is, we can find solutions by substituting the ray equation into the equation for the primitive we are testing against and then perform some algebraic manipulation to solve for *t* (the interval along the ray where the ray intersects the primitive). This is a technique we managed to steer clear of in the past because other solutions were available. We did not have to worry about your ability to solve equations using algebra. But we can put this off no longer. So we will start with the ray/plane intersection technique as a refresher for the math we (hopefully) learned in high school. We will start with the more familiar geometry perspective and then will solve the same problem algebraically to get us warmed up for what is to come later in the chapter. Do not fear if your algebra skills are a little rusty. We will review everything in detail and you will quickly see just how easy it can be once you get the hang of it again.

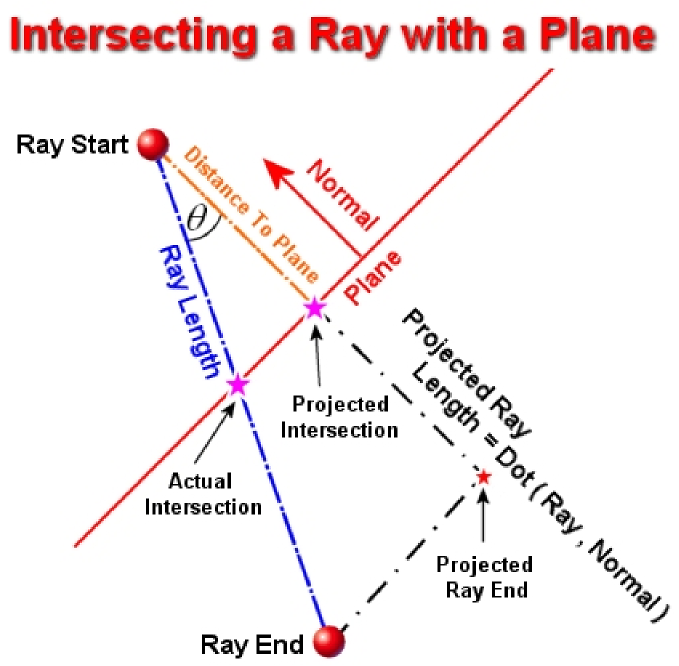


Figure 12.19

Study the diagram in Figure 12.19. Ray Start describes the position of the ray origin and the blue dashed line describes the delta vector. We can see that if we add the delta vector to the ray origin we would get the ray end position. We wish to find the position along the ray where intersection with the plane occurs. This can be done quite simply by applying the properties of the dot product and projecting the ray delta vector along the plane normal.

Our first task is to calculate the distance from the ray origin to the plane. Recall that the plane equation is:

$$ax + by + cz + d = 0$$

In the plane equation **a**, **b,** and **c** are the x, y and z components of the plane normal and **d** is the signed distance from the origin of the coordinate system to the plane along the plane normal. The **x**, **y**, and **z** variables in the plane equation describe the 3D components of the point on the plane. For the equation to be satisfied, the point (x,y,z) must lay on the plane described by (a,b,c,d). As both the position of the point and the plane normal are 3D vectors, we can write the plane equation as a dot product between the point and the plane normal with the addition of the plane distance. The plane equation in its more compact vector form looks like this:

$$P \bullet N + d = 0$$

where **P** is the point on the plane and **N** is the plane normal. It should also be noted that you could store the plane coefficients in a 4D vector and represent the point on the plane as an homogeneous 4D coordinate with w=1. The classification of the point against the plane is then simplified to a 4D dot product. As can be seen, the plane equation will return the distance from the point to the plane (which will be zero for any point on the plane). Therefore, if the plane equation is satisfied, the point **P** is located on plane. For any point not on the plane, the result will be the signed distance to the plane.

While it is most common for a plane to be stored in **a**, **b**, **c**, **d** format, it is also fairly common to find a plane represented using the plane normal and a point known to be on the plane. When this is the case, the **d** plane coefficient can be calculated by taking the negative dot product of the plane normal and the point on the plane. This can be seen by looking at the plane equation. To solve for **d** we have to leave **d** on its own on the left hand side of the equation. We can do this by subtracting $P \bullet N$ from the left and right side of the equation. Remember that anything we do to one side of the equation must be done to the other, so that the equation remains balanced. So we start with:

$$(P \bullet N) + d = 0$$

If P is our point on the plane, then we wish to subtract $P \bullet N$ from both sides of the equation. Subtracting it from the LHS (left hand side of the equals sign) leaves us with just **d** and we have successfully isolated **d**. We then have to perform the same operation to the RHS of the equation to keep it balanced. Subtracting $P \bullet N$ from zero gives us:

$$d = -(P \bullet N)$$

This can also be written as:

$$d = -N \bullet P$$

This is commonly how you will see **d** calculated.

Sometimes people will use the plane equation in the form:

$$ax + bx + cz - d = 0$$

In this case the vector form is obviously:

$$(P \bullet N) - d = 0$$

When using this form of the equation to solve for a distance, the **d** coefficient must be calculated slightly differently. Let us manipulate this form of the plane equation to see how **d** must be calculated. To isolate **d** we add **d** to both sides of the equation. This will cancel out **d** on the LHS and introduce it to the RHS.

$$P \bullet N = d$$

or in the more familiar form:

$$d = P \bullet N$$

So as you can see, if we intend to use the ax+by+cz+d version of the equation, d must be calculated by negating the result of the dot product between the plane normal and the point on the plane. If we intend to use the ax+by+cz-d version of the equation, d must be calculated by taking the dot product of the plane normal and the point on plane and not negating the result. This subtle difference in the calculation of d is what makes both forms of the equation essentially the same. Please refer back to Module I in this series for more detailed discussion on the positive and negative versions of the equation if you need more in depth coverage.

Getting back to our problem, we first must calculate the distance from the ray origin (Ray Start) to the plane. We do this as follows:

PlaneDist = $RayStart \bullet Normal + D$

This gives us the distance from the ray origin to the plane along the direction of the plane normal. This is shown as the orange dashed line in Figure 12.19 labelled Distance To Plane. We can also calculate the distance to the plane from the origin in another way. We can simply dot the plane normal with a vector formed from a point known to be on the plane to the ray origin. You will see us using this technique many times in our code.

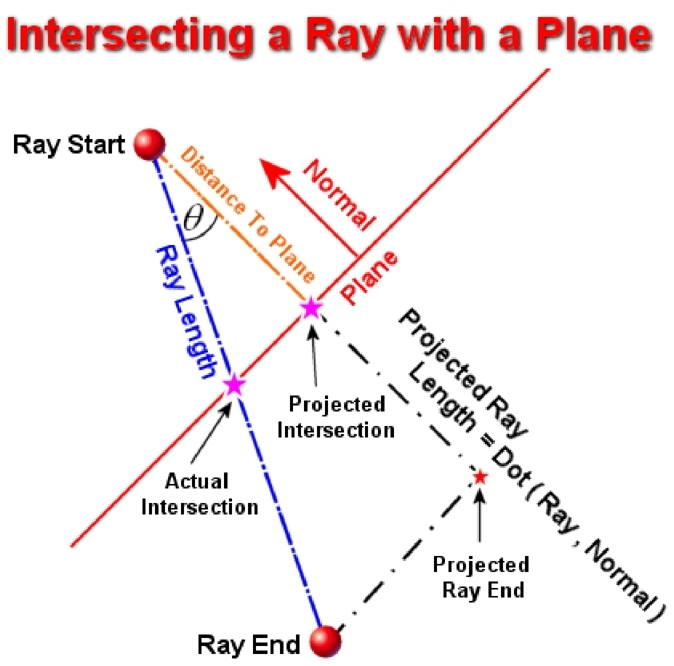


Figure 12.19 (again for convenience)

Our next task is to make sure we have the ray delta vector. Our intersection routines will use the ray in the form of an origin and a delta vector, so if the only information at hand is the start and end positions (as can sometimes be the case) we can calculate the delta vector by subtracting the ray start position from the ray end position. If we imagine that the line labelled Distance To Plane is our ray (instead of the actual ray) we would instantly know the distance to the intersection -- it is simply the distance to the plane from the start point which we have already calculated. However, our ray is travelling in an arbitrary direction and we need to know the time of intersection along this ray, not along the direction of the plane normal. But if we take the dot product of the negated plane normal and the ray delta vector ***d*** to get them both facing the same way (see the blue dashed line in Figure 12.19), we will get the length of the delta vector projected onto the plane normal. This describes the length of the ray delta vector as if it had been rotated around the ray origin and aligned with the plane normal.

Projected Line Length = $d \bullet -Normal$

Remember that the plane normal is a unit length vector but the ray delta vector is not. The result of a dot product between a unit length vector and a non unit length vector is:

$$\cos(\theta)|Ray| = d \bullet -Normal$$

In other words, the result is the length of the non unit length vector (the ray delta vector) scaled by the cosine of the angle between them. If you look at the image, we can see that this would give us the length of a vector from the ray origin to the Projected Ray End point along the direction of the negated plane normal. If we imagine the ray delta vector to be the hypotenuse of a right angled triangle, the result describes the length by which we would need to scale the unit length normal vector such that it correctly described the adjacent side of the same right angled triangle. If we were to scale the negated normal of the plane by this result, we would get the projected ray end point shown in the diagram. If we were to draw a straight line from the actual ray end point to the projected ray end point we can see that this would form the opposite side of a right angled triangle also. So, by dotting the ray delta vector with the negated normal, we end up with the length of the ray projected along the plane normal.

So what does this tell us? Quite a lot, actually. With the projected ray length, we can measure the intersection between the projected ray and the plane much more easily. We know from looking at the diagram that the point at which the projected ray intersects the plane in this new projected direction is simply the distance to the plane from the ray start point. This is the orange line in the diagram labelled Distance to Plane, which we have already calculated.

We know the length of the projected ray and the distance at which the plane intersects the projected ray because now they both share the same orientation. The intersection point is simply the distance to the plane from the ray origin. If we divide the distance to the plane (the distance to the intersection along the projected ray direction) by the length of the projected ray, we get back a float *t* that represents the intersection along the projected ray as a percentage between 0.0 and 1.0. It essentially describes the distance to the intersection in units of overall ray length. So a *t* value of 0.5 would mean the intersection is exactly halfway along the ray's delta vector.

You may see versions of the calculation of the *t* value that negate the sign of the Distance to Plane when performing such a calculation. It depends on whether you calculate your distance to plane value as a positive or negative value when a point is in front of the plane. You may also see versions of this calculation where the Projected Ray Length is negated before the divide. This boils down to whether it was projected by performing $d \bullet -Normal$ as we have, or by using $d \bullet Normal$. In the second case, the dot product will return a negative value for the projected line length because the delta vector ***d*** and the non-negated plane normal are facing in opposite directions. All that is happening is that the negation is not being applied to the normal; it is being applied later to the result. However, for the time being, we are assuming that we have calculated the distance to the plane and the projected ray length as positive values by flipping the plane normals where applicable.

t = Distance To Plane / Projected Ray Length

If Projected Ray length ($d \bullet -Normal$) equals zero then it means no intersection occurs because there is a 90 degree angle formed between the plane normal and the ray delta vector. This means the ray is parallel with the plane and cannot possibly intersect it. If the distance to the plane from the ray origin is a negative value, it means the ray origin is already behind the plane. You may or may not want to consider intersections with the back of a plane depending on your application's needs, so you may or may not wish to return false immediately when this is the case. If $t < 0$ then the ray origin is situated behind the plane and will never intersect the back of it. If $t > 1$ then the intersection with the plane happens past the end point of the ray and is not considered to be a valid intersection. A *t* value in the range of 0.0 to 1.0 describes the intersection with the plane along the projected ray and the intersection along the actual ray parametrically. Thus, if we wanted our Ray/Plane intersection method to return the position of intersection (and not just the *t* value) we could easily calculate it as follows:

Intersection Point = Ray Origin + (Ray Delta * t)

As you can see, we are just scaling the ray delta vector by the *t* value to reduce the delta vector's length such that it correctly describes the length of a vector from the ray origin to the intersection with the plane. We then add this to the ray origin to get the actual position of the intersection with the plane.

Taking what we have just learned, we can now implement a function that will calculate ray/plane intersections. It will take a ray origin, a ray delta vector, a plane normal, and a point on that plane as input parameters. We also have two output parameters: a 3D vector which will store the intersection point (if one occurs) and a float to store the *t* value of intersection.

```
bool RayIntersectPlane( const D3DXVECTOR3& Origin,
                        const D3DXVECTOR3& Direction,
                        const D3DXVECTOR3& PlaneNormal,
                        const D3DXVECTOR3& PlanePoint,
                        float& t,
                        D3DXVECTOR3& IntersectionPoint )
{
    // Calculate Distance To plane
    float PlaneDistance = D3DXVec3Dot( &(Origin - PlanePoint), &PlaneNormal );

    // Calculate Projected Ray Length
    float ProjRayLength = D3DXVec3Dot( &Direction, &-PlaneNormal );

    // If smaller then zero then it is either running parallel to the plane
    // or intersects the back of the plane. We choose to ignore in this version.
    if ( ProjRayLength < 1e-5f ) return false;

    // Calculate t
    t = PlaneDistance / ProjRayLength;

    // Plane is either before the start of the ray or past the end of the ray so
    // not intersection
    if ( t < 0.0f || t > 1.0f ) return false;

    // Calculate the intersection point
    IntersectionPoint = Origin + ( Direction * t );

    // We're intersecting
    return true;
}
```

In this version of the function we calculate the distance to the plane from the ray origin by projecting a vector from the point on the plane to the ray origin along the plane normal. This step would not be necessary if the function was passed all the plane coefficients and not just the normal.

Next we see another version of the function that does the same thing, only this time, the distance to the plane from the ray origin can be calculated using ax+by+cz+d. In this version, we pass the plane in as a single parameter (instead of a normal and a point) stored inside a D3DXPLANE structure. This is a structure with four float values labeled a, b, c, and d as expected.

```
bool RayIntersectPlane( const D3DXVECTOR3& Origin,
                        const D3DXVECTOR3& Direction,
                        const D3DXPLANE& Plane,
                          float& t,
                          D3DXVECTOR3& IntersectionPoint )
{
    // The D3DXPlaneDotCoord function performs the plane equation calculation.
    // It could be written manually as :
    // Plane Distance = Plane.a * Origin.x + Plane.b * Origin.y +
    //                  Plane.c * Origin.Z + Plane.d
    float PlaneDistance = D3DXPlaneDotCoord( &Plane , &Origin );

    // Calculate Projected Ray Length using the D3DXPlaneDotVector function
    // It performs ax + by + cz. A simple dot product between the plane normal
    // and the passed vector
    float ProjRayLength = D3DXPlaneDotVector( &Plane, &Direction );

    // if smaller then zero then it is either running parallel to the plane
    // or intersects the back of the plane we choose to ignore in this version.
    if ( ProjRayLength > -1e-5f ) return false;

    // Sign must be negated because Plane Normal was not inverted
    // prior to dot product above
    t = - ( PlaneDistance / ProjRayLength );

    // Plane is either before the start of the ray or past the end of the ray so
    // not intersection
    if ( t < 0.0f || t > 1.0f ) return false;

    // Calculate the intersection point
    IntersectionPoint = Origin + ( Direction * t );

    // We're intersecting
    return true;
}
```

This version of the function will probably appeal more to the math purists. This will make some sense when we try to arrive at an algebraic solution in a moment. Here we are using the standard plane equation **ax + by + cz + d** to calculate the distance to the plane from the ray origin. This is performed inside the function D3DXPlaneDotCoord. It essentially performs a 4D dot product between the plane coefficients and a 3D vector with an assumed *w* coordinate of 1.0 (1**d*).

We then calculate the projected line length by doing a dot product between the plane normal and the ray delta vector. We use the D3DXPlaneDotVector function for this which performs a 3D dot product between the a, b, and c components of the plane (the normal) and the x, y and z components of the input vector. You will recall that in the previous version of the function we negated the plane normal before we performed this dot product so that both the ray delta vector and the plane normal were facing the same way. In this version we have not done this (i.e., negating the a, b, and c coefficients of the plane prior) so the projected line length returned will be a negative value. Dividing the plane distance by this value will also generate a negative *t* value, which is not desirable. Notice how we take care of this later in the function simply by negating the result the of the *t* value calculation. This nicely demonstrates what was said earlier about the *t* value calculation potentially needing to have its result negated depending on the sign of the inputs to its equation.

The Algebraic Solution

In the last section we analyzed ray / plane intersection from a geometric perspective and arrived at our final *t* value calculation. As we proceed through this chapter however, we will have to be ready to find algebraic solutions since the problems we will attempt to solve will become more challenging and less easily represented from a geometric perspective. In order to refresh ourselves in the subject of algebra, we will start by finding a solution for Ray / Plane intersection using algebraic manipulation.

When using the standard plane equation in the last section, we discovered that to determine the value of *t*, the calculation was:

$$t = -\frac{\text{Distance To Plane}}{\text{Projected Ray Length}}$$

Let us see if we come up with the same equation using an algebraic approach.

Our ray **R** is defined parametrically and has an origin vector **O** and a delta vector **V**. *t* values in the range of 0.0 to 1.0 represent points on the ray.

$$R(t) = O + tV$$

Our plane is represented using the standard form of the plane equation:

$$ax + by + cz + d = 0$$

In vector form this can be written as the dot product between a point **P** (x,y,z), a plane normal **N** (a,b,c) and the addition of the d coefficient (the distance to the plane from the origin of the coordinate system).

$$P \bullet N + d = 0$$

Substituting the Ray into the Plane Equation

In the vector form of the plane equation, **P** represents a point on the plane if the equation is true. However, we wish to find a point on the ray which intersects the plane. Of course, if the ray intersects the plane, the point where it intersects is obviously on the plane. Thus, we are going to want to find the *t* value for our ray which would produce a point which makes the plane equation true. So we substitute **P** in the plane equation with our ray definition and we get:

$$(O + tV) \bullet N + d = 0$$

As the term (**O** + *t***V**) is dotted by **N**, this is equivalent to dotting both vectors (**O** and *t***V**) contained inside the brackets by **N** individually (the dot product is distributive). This means we can instead write **O** dot **N** + *t***V** dot **N** allowing us to expand out the brackets:

$$O \bullet N + tV \bullet N + d = 0$$

In order to find the point on the ray which intersects the plane, we have to find *t*. If we solve for *t* in the above equation, then we are finding the point on the ray which returns a distance of zero (look at the = 0 on the RHS) from the plane.

Solving such equations is actually quite simple. We just have to figure out a way to move everything we do not want to know about over to the right hand side of the equals sign, leaving us with the variable we do want to know about isolated on the left hand side of the equals sign (or vice versa). At this point, we will have solved the equation and will know how to calculate *t*.

Looking at the above equation we can see that the first thing we can get rid of on the LHS is '+*d*'. If we subtract *d* from the LHS of the equation it will cancel it out. We must also make sure that anything we do to the LHS must be done to the RHS to keep the equation balanced. If we subtract *d* from both sides of the equation, our equation now looks like this :

$$O \bullet N + tV \bullet N = -d$$

We also see that *t* is multiplied by ($V \bullet N$) on the LHS. If we divide the LHS by $V \bullet N$ we can cancel that out from the LHS. Of course, we must also divide the RHS by $V \bullet N$ as well to keep the equation balanced. Performing this step gives us the following:

$$\frac{O \bullet N}{V \bullet N} + t = \frac{-d}{V \bullet N}$$

t is very nearly isolated on the LHS now. All that is happening to *t* now on the LHS is that it is having $O \bullet N$ added to it. Therefore, we can completely isolate *t* on the LHS by subtracting $O \bullet N$ from both sides of the equation. This leaves us with the final solution for *t* as shown below:

$$t = \frac{-d - O \bullet N}{V \bullet N}$$

Of course, this can also be written by placing the numerator in brackets and distributing a -1 multiplication:

$$t = \frac{-(O \bullet N + d)}{V \bullet N}$$

This is the equivalent of the more common form:

$$t = -\frac{O \bullet N + d}{V \bullet N}$$

And here we see the final solution for *t*. Note that is it exactly how the last version of the RayIntersectPlane function calculated its *t* value:

```
t = - ( PlaneDistance / ProjRayLength );
```

Looking at our final equation, we know that $O \bullet N + d$ calculates the distance to the plane from the ray origin **O**. We also know that $V \bullet N$ computes the length of the ray projected onto the normal. However, because the normal **N** and the delta vector **V** are pointing in opposite directions, this will produce a negative length value. When the positive plane distance is divided by a negative projected length, we will get back a negative *t* value. This is why we negate the final result. As you can see, this is exactly what we have in the equation for *t*.

The solution for *t* could also be written by negating $V \bullet N$ before the divide like so.

$$t = \frac{O \bullet N + d}{-V \bullet N}$$

These are all identical ways of calculating the final positive *t* value with a negative projected length. You may see us using any of the forms to solve for *t* in our code from time to time. They all are equivalent.

12.4.3 Ray / Polygon Intersection Testing

Determining whether or not a ray intersects a plane is very a useful technique. It is used in polygon clipping for example, where each edge of the polygon can be used as a ray which is then intersected with the clip plane. We will cover polygon clipping in the next chapter when we discuss spatial partitioning techniques and you will see this all taking place. Ray/Plane intersection testing is also used when determining whether or not a ray is intersecting a polygon. This is important because ray/polygon intersections will be used by our collision detection system and by a host of other techniques we will employ later in this training program.

We will see one application of Ray/Polygon intersection testing later in this course when writing a Lightmap Compiler tool. A Lightmap Compiler is an application that will calculate and store lighting information in texture maps which can later be applied to polygons to achieve excellent per-texel lighting results. In such an application, a ray is used to represent a beam of light between each light source in the scene and the texels of each polygon's texture. If the light source ray reaches the polygon currently being tested then the lighting information is added to the light map texture. If the ray is intersected by other polygons in the environment on its way to the polygon currently having its lightmap computed, then the polygon is said to be at least partially in shadow with respect to that light source and the light source does not contribute lighting information to that particular texture element. We will look at how to build our Lightmap Compiler later in the course. In fact, Ray/Polygon intersection tests will be used in all manner of lighting techniques and not just light mapping.

Another application of Ray/Polygon intersection is a technique called picking. When writing an application such as GILES™, we want the user to be able to select a polygon in the scene using the mouse as an input device. To accomplish this, the screen coordinates of the mouse can be converted into world space coordinates and then used to create a ray that extends into the scene from the camera position. The polygon intersection producing the smallest *t* value is assumed to be the first polygon that is hit by the ray and is the polygon that is selected. This is pretty much the same technique that is used by simple ray tracing renderers as well. In a simple ray tracer, the near view plane is essentially carved up into a pixel grid and rays are cast from the eye point (located some distance behind the plane) through each pixel location into the scene to find the point of closest intersection. The lighting information at that point is then computed and stored in the image pixel.

As you are no doubt convinced by now, the need to be able intersect rays with polygons is of paramount importance in 3D computer graphics.

Testing whether a ray intersects a polygon is a two step process. First we can perform a ray/plane intersection test to see whether the ray intersects the infinite plane that the polygon is on. The ray/plane test is done first because testing for a ray/plane intersection is computationally inexpensive. If a ray does not intersect the plane that a polygon lies on, it cannot possibly intersect the polygon itself. Therefore, performing the ray/plane intersection test first provides an early out mechanism for many of the polygons that we need to test which are not going to be intersecting the ray.

If a polygon passes the initial ray/plane test, the next test looks to see if the ray intersects the actual polygon itself. Since planes are infinite, even if the ray does intersect the plane of the polygon, the intersection with the plane may have happened well outside the borders of the polygon. So our second test will need to determine if the plane intersection point returned from the ray/plane intersection test is within all the edges of the polygon. If this is the case, then the point is indeed inside the polygon and we will have successfully determined that the ray has intersected the polygon. Since polygons can take on arbitrary shapes it might appear that a generic solution would be very hard to create. However, with our trusty dot product and cross product operations (see Module I), the task is mostly trivial.

Once a plane intersection point has been found, our job is to see whether the intersection point actually resides within the interior of the polygon. Since a polygon is just a subset of its parent plane, the testing process is really nothing more than finding out whether the intersection point is contained within all edges of the polygon. So how do we test this? Usually, your polygons will store their plane normals, but if this is not the case, you can refer back to Module I to see how to generate a polygon normal by performing the cross product on two of its edge vectors and normalizing the result. We will assume for the sake of this discussion that the polygon you wish to test for intersection also contains a polygon normal.

The "Point in Polygon" testing process is broken down into a set of simple tests (one for each edge of the polygon). For each edge we create an edge normal by performing a cross product between the edge vector and the polygon normal. For this test, the edge normal does not need to be normalized. This will return to us the normal of the edge and since we know that any vertex in the edge will suffice for a point that lay on the plane described by this edge normal, we have ourselves a complete description of the plane for that edge. We then simply classify the plane intersection point against the edge plane.

In the example shown in Figure 12.20, the edge planes are created with their normals facing outwards from the center of the polygon. Therefore, if the point is found to be in front of **any** edge plane, it is outside the polygon and we can return false immediately. If the point is behind the edge currently being tested, then we must continue to test the other edges. Only when the intersection point is contained behind all edge planes is the intersection point considered to be inside the polygon.

Figure 12.20 shows this process by demonstrating two example intersection points (P0 and P1) which are being tested for "Point in Poly" compliance. P0 is clearly inside the polygon because it lies behind all the planes formed by the edges of the polygon. P1 is not inside because it does not lay behind all edge planes (although it does lay behind some). We will use this diagram to step through the testing process using these two example intersection points.

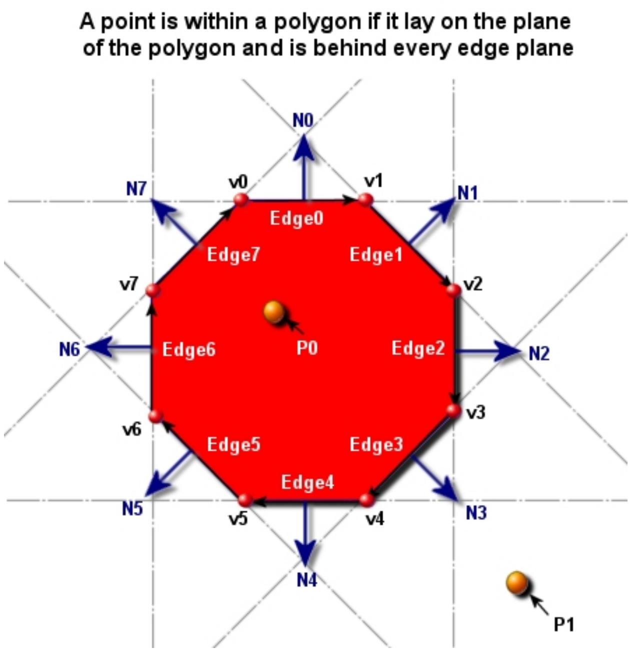

A point is within a polygon if it lay on the plane of the polygon and is behind every edge plane
N0
N1
N2
N3
N4
N5
N6
N7
v0
v1
v2
v3
v4
v5
v6
v7
Edge0
Edge1
Edge2
Edge3
Edge4
Edge5
Edge6
Edge7
P0
P1

Figure 12.20

In the first example we will start with intersection point P0 which has been found to be on the plane of the polygon by a prior ray/plane intersection test. The polygon being tested for intersection is an octagon with 8 vertices labelled v0 through v7. We start by entering a loop that will iterate through each of the eight edges of the polygon. In the first iteration of this loop, we test the first edge formed by the first two vertices v0 and v1. We can create an edge vector by subtracting v0 from v1 which give us the black arrow running along the first edge in the diagram from left to right connecting those two vertices:

```
EdgeVector = v1 – v0
```

Now, while the polygon normal is not illustrated in the diagram because it is directly facing us (coming out of the page), we can imagine it is sticking out from the polygon in our direction such that it is perpendicular to the polygon. So we have an edge vector and a polygon normal at our disposal with which to generate the plane normal for the first edge (N0 in the diagram).

We know that performing the cross product between two vectors will return a third vector which is orthogonal to the two input vectors. The edge vector and the polygon normal are already perpendicular to each other, so crossing these two vectors will return vector N0 forming an orthogonal set of vectors (like a coordinate system axis). Vector N0 will be the normal to the plane for that edge. Looking at Figure 12.20 and at edge 0 in particular, we can see that the normal N0 generated by the cross product is the normal to an infinite plane aligned with the edge and shown as the gray dashed line running horizontally through that edge. We can imagine this plane to continue infinitely both to the left and right of the edge and can further imagine the plane continuing infinitely both into and out of the page on which you are viewing the diagram. Remember, these edge planes are perpendicular to the plane of the actual polygon.

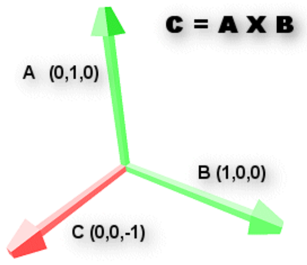


Figure 12.21

Once we have the edge plane normal for the edge we are testing, we simply classify the point P0 against that plane. If the point is in front of the edge plane, then it is clearly outside the polygon. In this example however, we can see that intersection point P0 is behind the first edge plane and we must continue through to the next iteration of the loop and check the next edge in the polygon. So, as P0 has now passed the test for edge 0, we move on to the second iteration of the edge loop where we test edge 1. Once again, we create an edge vector from the vertices of that edge (v1 and v2) like so:

```
EdgeVector =  v2 - v1
```

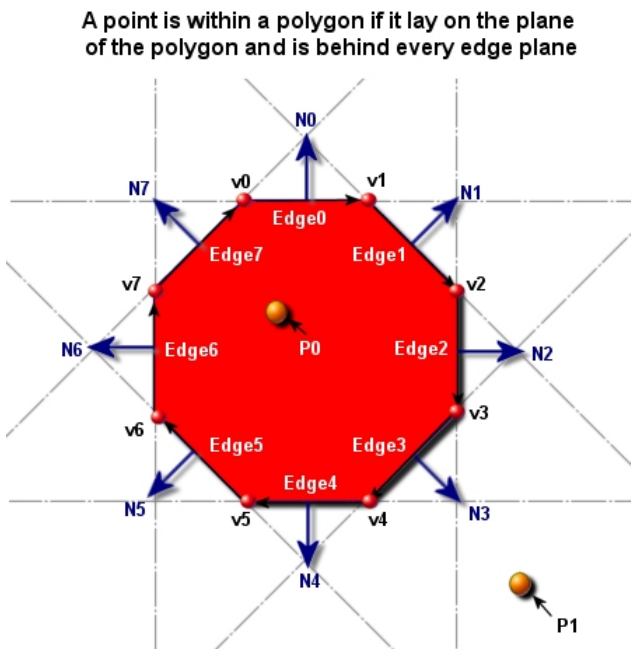

A point is within a polygon if it lay on the plane of the polygon and is behind every edge plane
N0
N1
N2
N3
N4
N5
N6
N7
v0
v1
v2
v3
v4
v5
v6
v7
Edge0
Edge1
Edge2
Edge3
Edge4
Edge5
Edge6
Edge7
P0
P1

Figure 12.20 (again for convenience)

We then perform the cross product with the polygon normal and edge 1 to generate the normal N1 (the normal of the second edge). We classify the intersection point P0 against this plane and once again find it to lay behind the plane for this edge. If it lay in front of this plane we could return false from the function and skip the edge tests which have not yet been performed. In this case however, the point has passed the test for edge 1 also so we must continue to the third iteration of the loop and test the third edge.

You should be able to see by looking at the diagram that for any point that is actually inside the polygon, each edge will need to be tested and the point P0 would be found to lay behind all of the polygon's edge planes. If we reach the end of the process and have not yet returned false, it means the point is situated behind all edge planes and is therefore inside the polygon. Thus, the ray for which the intersection point with the plane was generated does indeed impact the polygon also.

Let us now see how intersection point P1 fares using this technique. We can clearly see by looking at it that it is not within the bounds of the polygon, but let us quickly walk through the testing process to see at which point it gets rejected and the process returns false.

First we create the plane for edge 0 (N0) and see that intersection point P1 is behind the plane and cannot be rejected at this point. As far as we are concerned right now, P1 may very well be inside the polygon, so we continue on to edge 1. Once again, we can see that when we test P1 against the second edge plane (edge 1), P1 is also found to be behind edge plane N1 and therefore, at this point may still be inside the polygon. When testing edge2 however, we can see that when we create the plane for this edge and classify P1 against it, P1 is found to be in front of the plane described by normal N2. As such, we can immediately return false from this procedure eliminating the need to test any more edges.

So this process does indeed correctly tell us if the point is inside the polygon, provided that we know beforehand that the point being tested is on the plane of the polygon. This all works out quite nicely because the point in polygon test is much more expensive than the ray/plane intersection test. We only have to perform the point in polygon test for a given polygon when the ray/plane intersection test was successful. When performing ray/polygon intersection tests on a large number of polygons, most of those polygons will be rejected by the cheaper ray/plane test first. This avoids the need to carry out the point in polygon test for polygons which cannot possibly intersect the ray. Once we have reduced our polygon list to a list of potential intersecting polygons, we can move to the second step and perform point in polygon tests on this subset. Note that usually only a few, at most, of the potential intersecting polygons will actually be intersected by the ray. So it is only for those polygons that every edge will need to be tested. But even the majority of these polygons tested will exit from the process early as soon as the first edge plane is found that contains the intersection point in its front space.

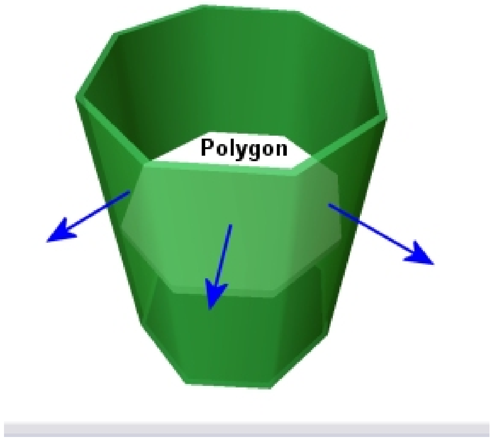


Figure 12.22

It is sometimes difficult to picture the orientations of the edge planes given the 2D image we saw earlier. But we can imagine the planes in the above example forming an infinitely long 3D cylinder based on our octagon polygon. Figure 12.22, while showing the planes as having finite length (for the sake of illustration), hopefully demonstrates this concept. The green cylinder surrounding the polygon in this example shows the planes of each edge. These planes would be infinitely tall in reality. The polygon is at the center of these planes. As we see in the diagram, the plane normals face outwards. Any point found to be behind every plane is by definition inside the space of the cylinder formed by these planes. Since we already know that the point we are testing shares the same plane as the polygon (determined by the ray/plane intersection test) we know that the point is actually on the polygon itself.

Let us now look at how we might implement a "Point in Polygon" function. We will return true if the positional vector passed in is found to be contained inside the polygon and false otherwise. Remember, this function should be used in conjunction with a ray/plane intersection test. Whenever it is called, we are always passing in a vector that describes a position on the plane of the polygon being tested. This will be the intersection point returned from the ray/plane intersection test in a prior step. If the ray/plane test returned false for a given polygon, then there is no need to call this next function.

This simple version of the function assumes that the polygon being tested is an N-gon where all points are assumed to lay on the same plane in a clockwise winding order.

```
BOOL PointInPoly ( D3DXVECTOR3 * Point , CPolygon * pPoly)
{
   D3DXVECTOR3 EdgeNormal, Direction, Edge;
   D3DXVECTOR3 FirstVertex, SecondVertex;

   // Loop through each vertex ( edge ) in the polygon
   for ( int a = 0; a < VertexCount; a++ )
   {
       // Get First vertex in edge
       FirstVertex      = pPoly->m_vecVertices[a];

       // Get second vertex in edge  ( with wrap around)
       SecondVertex = pPoly->m_vecVertices[ (a+1)%VertexCount ] ;

        // Create edge vector
        Edge            = SecondVertex - FirstVertex;

        // Generate plane normal
        D3DXVec3Cross   ( &EdgeNormal, &pPoly->m_vecNormal, &EdgeNormal );

        //Create vector from point to  vertex ( point on plane )
        Direction       = FirstVertex-*Point;

        float d = D3DXVec3Dot( &Direction, &EdgeNormal );

        if ( d < 0 ) return FALSE;

    } // Next Edge

    return TRUE;
}
```

The code sets up a loop to test each edge (each pair of vertices). Notice how we use the modulus operator to make sure that the second vertex of the final edge wraps back around again to index the first vertex in the polygon's vertex list. We then create Edge, the edge vector of the current edge being tested.

After generating the edge vector, we create the normal for the edge plane by performing a cross product between the edge vector we just calculated and the polygon normal. In this example, we are assuming that the polygon data is represented as a CPolygon structure which contains the normal. If the normal is not present then you will have to generate it by crossing two edge vectors.

At this point we have the normal for the edge currently being tested and we also know a point that exists on that plane (i.e., either vertex in the edge). We use the first vertex in the edge, but it could be the second also, as long as the point is on the edge plane. We then calculate a vector (Direction) from the point being tested to the point on the plane (the first edge vertex) and take the dot product of this vector with the edge normal. If the angle between these two vectors is greater than 90 degrees (i.e., the result is larger than zero) then the point is in front of the plane. As soon as this happens for any edge we know the point is not contained inside the polygon and we can exit.

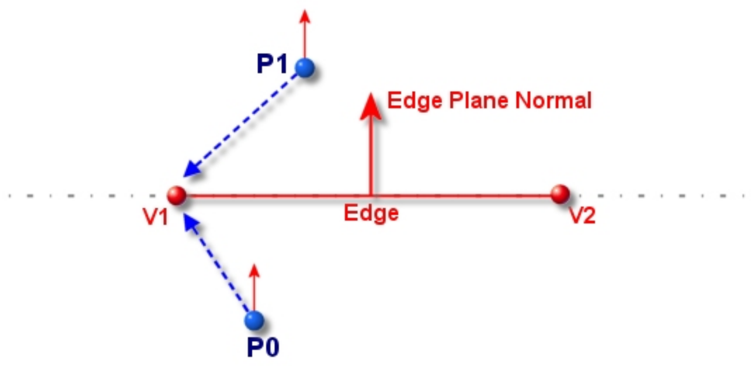


Figure 12.23

Figure 12.23 demonstrates testing to see if a point is in front or behind the edge plane. In this diagram we show two example points (P0 and P1) and perform the test on a single edge (v2 – v1). Starting with P0 we can see that creating a vector from P0 to V1 (called Direction in the above code) forms an angle with the plane normal that is less than 90 degrees. Remember that we bring vector origins together when we perform the dot product. Therefore, the dot product will return a positive result indicating that the point is behind the edge and cannot be removed from consideration without checking the other edges. The point should be tested against other edge planes as it may be inside the polygon. However, when we perform the same edge test on point P1, we can see that the vector created from P1 to V1 forms an angle with the edge normal that is larger than 90 degrees, returning a negative dot product result. When this happens, we know that the point is in front of the edge plane and cannot possibly be contained inside the bounds of the polygon. We can return false immediately and refrain from performing any more edge tests.

Bringing together everything we have learned, a function could be written to test for the intersection between a ray and a polygon by combining the ray/plane intersection test and the point in polygon test. In the following code we implement a function called RayIntersectPolygon which does just this. This function is assumed to understand the data type CPolygon which is expected to contain a polygon normal. You should be able to substitute this for your own custom polygon structures. This version of the function will return the intersection point and the intersection *t* value if the ray hits the polygon.

```
bool RayIntersectPolygon( const D3DXVECTOR3& Origin,
                          const D3DXVECTOR3& Direction,
                          const CPolygon  &Polygon,
                          float& t,
                          D3DXVECTOR3& IntersectionPoint )
{
    D3DXVECTOR3 EdgeNormal, Direction, Edge;
    D3DXVECTOR3 FirstVertex, SecondVertex;

    // Phase One : Test if Ray intersects polygon’s plane
    // Calculate Distance to polygons plane using first vertex as point on plane
    float PlaneDistance = D3DXVec3Dot( &(Origin - Polygon.VertexList[0]),
                                       &Polygon.Normal );

    // Calculate Projected Ray Length onto polygons plane normal
    float ProjRayLength = D3DXVec3Dot( &Direction, &-Polygon.Normal );

    // if smaller then zero then it is either running parallel to the plane
    // or intersects the back of the plane we choose to ignore in this version.
    if ( ProjRayLength < 1e-5f ) return false;
    // Calculate t
    t = PlaneDistance / ProjRayLength ;

    // Plane is either before the start of the ray or past the end of the ray so
    // not intersection
    if ( t < 0.0f || t > 1.0f ) return false;

    // Calculate the intersection point
    IntersectionPoint = Origin + ( Direction * t );

    // Phase Two : Point in Poly
    // Loop through each vertex ( edge ) in the polygon
    for ( int a = 0; a < VertexCount; a++ )
    {
        // Get First vertex in edge
        FirstVertex     = pPoly->m_vecVertices[a];

        // Get second vertex in edge  ( with wrap around)
        SecondVertex = pPoly->m_vecVertices[ (a+1)%VertexCount ] ;

        // Create edge vector
        Edge            = SecondVertex - FirstVertex;

        // Generate plane normal
        D3DXVec3Cross   ( &EdgeNormal, &EdgeVector , &pPoly->m_vecNormal);

        //Create vector from point to  vertex ( point on plane )
        Direction       = FirstVertex-IntersectionPoint;

        float d = D3DXVec3Dot( &Direction, &EdgeNormal );

        if ( d < 0 ) return FALSE;

    } // Next Edge

    return TRUE; // Yes it must intersect the polygon

}
```

Most often in 3D graphics we are dealing with triangles and not with N-gons. The point in polygon test can be optimized somewhat when being used specifically for “Point in Triangle” tests because we can unwind the loop and lose a bit of overhead. The following snippet of code shows how we might write a function called PointInTriangle whose code could also be substituted into the function shown above to create a RayIntersectTriangle method.

A triangle will always have three vertices, so we will pass these in along with the normal for the triangle.

```
bool PointInTriangle(const D3DXVECTOR3& Point,  const D3DXVECTOR3& v1,
                     const D3DXVECTOR3& v2,     const D3DXVECTOR3& v3,
                     const D3DXVECTOR3& TriNormal )
{
    D3DXVECTOR3 Edge, EdgeNormal, Direction;
    // First edge
    Edge      = v2 - v1;
    Direction = v1 - Point;
    D3DXVec3Cross( &EdgeNormal, &EdgeNormal , TriNormal );

    // In front of edge?
    if ( D3DXVec3Dot( &Direction, &Edge ) < 0.0f ) return false;

    // Second edge
    Edge      = v3 - v2;
    Direction = v2 - Point;
    D3DXVec3Cross( &EdgeNormal, &Edge, &TriNormal );

    // In front of edge?
    if ( D3DXVec3Dot( &Direction, &EdgeNormal ) < 0.0f ) return false;

    // Third edge
    Edge      = v1 - v3;
    Direction = v3 - Point;
    D3DXVec3Cross( &EdgeNormal, &Edge, &TriNormal);

    // In front of edge?
    if ( D3DXVec3Dot( &Direction, &EdgeNormal ) < 0.0f ) return false;

    // We are behind all planes
    return true;
}
```

The cross product is not commutative so you have to make sure to feed the Edge and Plane normal vectors into the cross product in the order shown above if your edge normals point outwards. If the normals are generated so that they point inwards, then the sign of the dot product should also be changed to match. With inward facing normals, a point is considered inside the triangle if the angle formed by the vector from that point to the vertex in the edge and the edge normal is greater than 90 degrees (greater than 0.0). Alternatively, you could flip the direction of the vector from the point to the vertex and use (Point – Vertex) instead of (Vertex – Point). In that case, both vectors will have been flipped for the comparison and point into the same halfspace, so we can go back to the < 0.0 case for failure.

You will completely invalidate your results if you calculate your vectors to point in opposite directions and forget to change the sign of the dot product test. This is worth remembering since you will see instances of this in our source code from time to time.

12.4.4 Swept Sphere / Plane Intersection Testing

As discussed earlier, an application that uses our collision system will call the CollideEllipsoid method to request a position update for a moving entity. When examining a simplified version of this function, we saw the invocation of the detection phase via EllipsoidIntersectScene. We will get to study the code to this function a bit later in the chapter, but first we will concentrate on understanding the separate intersection tests which must be performed by this function in order to determine a new position for the ellipsoid.

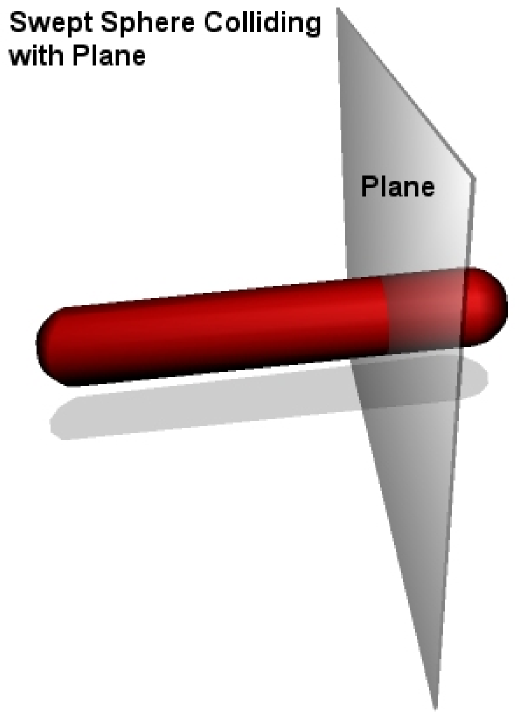


Figure 12.24

EllipsoidIntersectScene is passed the velocity vector and the ellipsoid center position in eSpace so that it can determine scene geometry intersections using unit sphere tests. The function will have to test the swept sphere against each potential colliding triangle to see if an intersection occurs. It will then return a final eSpace position based on the closest intersecting triangle (if one exists).

In order to use our unit swept sphere for intersections, this function must transform the vertices of each triangle it tests into eSpace prior to performing the intersection test. This is absolutely essential since we must have our ellipsoid and our triangles in the same space in order for the intersection results to have meaning. Over the next several sections of the lesson we will discuss the core intersection techniques that will be used by this function to determine intersections between the swept sphere and a triangle. These intersection tests will have to be executed for every triangle that is considered a potential candidate for intersection. As we currently have no broad phase in our system, this will be every triangle registered with the collision geometry database. Once we have covered the various components of the detection phase we will finally see how they are used in combination by looking at source code to the EllipsoidIntersectScene function.

When covering ray/triangle (ray/polygon) intersection, the first test we performed was ray/plane. If the ray did not intersect the plane, then it could not intersect the triangle and we can skipped any further tests. This was beneficial because this is an inexpensive test that allows us to reject most triangles very early on in the process. For the same reasons, when determining whether a swept sphere intersects a triangle, our first test is also going to be determining if the swept sphere intersects the plane of the triangle. If this is not the case then it cannot possibly intersect the actual triangle and we can bail out early from the test. Intersecting a swept sphere with a plane is going to be the cheapest intersection test we will have to perform for each triangle, so the benefit is the same for spheres as it was for rays (i.e., many of the potential colliders will be removed from consideration with very little computational overhead). Figure 12.24 demonstrates the intersection of a swept sphere and a plane. As can be seen, the sphere does indeed intersect the plane at some point along its velocity vector and we must write a function that will return that time *t* of intersection.

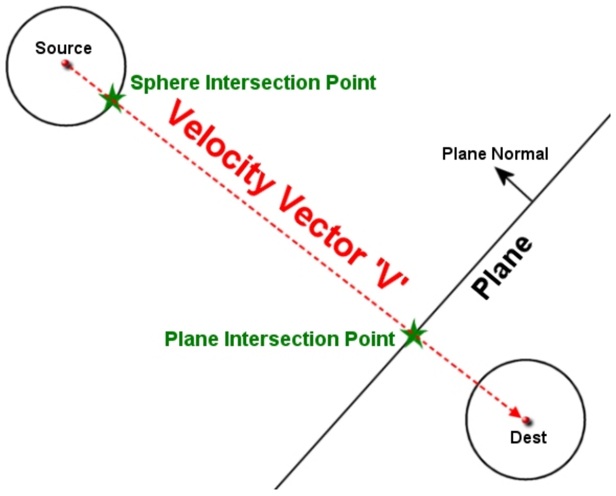


Figure 12.25

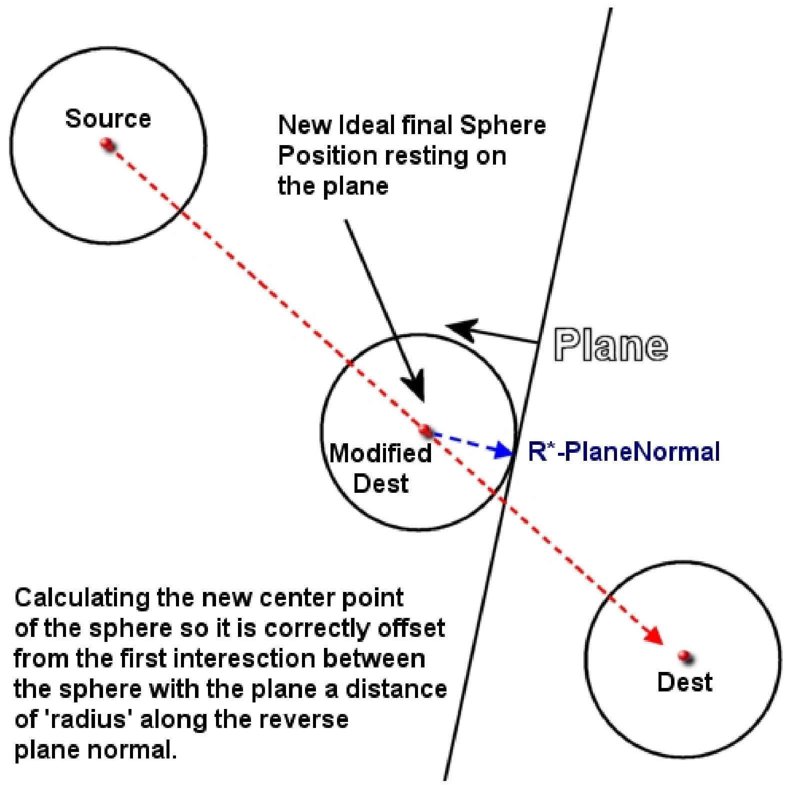
In Figure 12.25 you see the intersection we ultimately wish to find marked off. Source shows the center of the sphere in its starting position and Dest shows the sphere in its desired final position. The red dashed line is the velocity vector V describing the path we intend to move the sphere along during this update.

We want to find the *t* value for the time when the surface of the sphere first intersects the plane. The diagram shows both the intersection point on the plane and the matching point on the sphere surface that will intersect the plane first.

Unfortunately, we are not interested in finding the *t* value for the moment when the center of the sphere (Source) intersects the plane. If we were, this could be achieved by performing a simple ray/plane intersection test. You should be able to see in this diagram that if the center of the sphere Source was assumed to be the origin of a ray and the velocity vector V was the ray's delta vector, performing a ray/plane intersection test would give us the *t* value for the time when the center of the sphere would intersect the plane along the velocity vector. But this information is not helpful to us as is, because when this *t* value is later used to create the new center position of the sphere, the sphere would already be halfway embedded in the plane (and thus the triangle, assuming one exists at that location on the plane).

But as it happens, analyzing this problem sheds some light on a relationship that exists between the plane normal, the sphere radius and the intersection point on the sphere surface. By exploiting this relationship we will be able to rearrange the problem into one that allows us to use a simple ray/plane intersection test again.

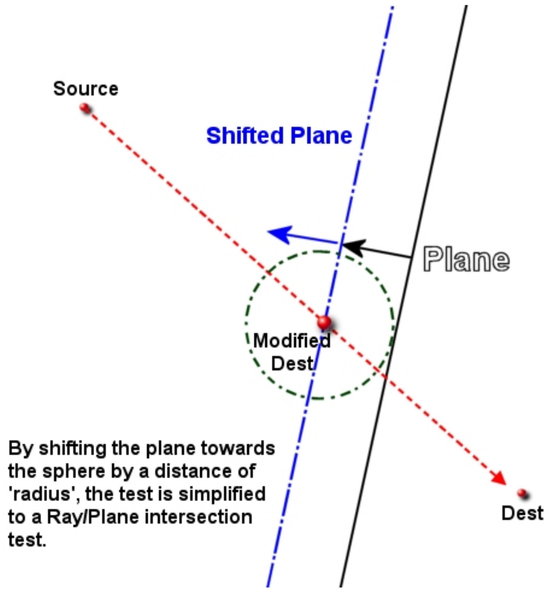
In Figure 12.26 we see another example of a sphere of radius R intersecting the plane. We also see the sphere in its desired final resting place on the plane (Modified Dest). It is this center position we are attempting to find.



Source
New Ideal final Sphere Position resting on the plane
Plane
Modified Dest
R*-PlaneNormal
Dest
Calculating the new center point of the sphere so it is correctly offset from the first interesction between the sphere with the plane a distance of 'radius' along the reverse plane normal.

Figure 12.26

Notice that if we study the center point of the sphere resting against the plane (Modified Dest), it is offset from the plane by a distance equal to its radius. That is, it is offset along the plane normal by a distance R. Notice also how the intersection point on the surface of the sphere is found by traveling out from the center of the sphere Modified Dest by a distance of R along the negated plane normal. This is very important because it just so happens that regardless of the orientation of the plane and the orientation of the velocity vector V, if an intersection happens, the first point of intersection on the surface of the sphere will *always* be the point that is offset from the center of the sphere by a distance of R along the negated plane normal.



Source
Shifted Plane
Plane
Modified Dest
By shifting the plane towards the sphere by a distance of 'radius', the test is simplified to a Ray/Plane intersection test.
Dest

Figure 12.27

While this might not sound like a great discovery by itself, study Figure 12.26 again and imagine that you could pick up the plane and move it along the plane normal by a distance of R units. You would have essentially shifted the plane along the plane normal by a distance of –R such that the shifted plane now passes straight through the center of the sphere in its new position (Modified Dest). This is clearly shown in Figure 12.27.

As mentioned, it is the new center position of the sphere resting up against the plane we are searching for. If we shift the plane along its normal by a distance of -R, we get a plane that the center point of the sphere will intersect at exactly the same time that the surface of the sphere intersects the actual (unshifted) plane. This means we can find the new center position of the sphere by simply performing a ray/plane intersection test using the shifted plane. The ray will have the initial sphere center point

Source as its origin and it will have velocity vector V as its delta vector. As can be seen in Figure 12.27, the point at which this ray intersects the shifted plane describes the exact position where the center of the sphere should be when the surface of the sphere first intersects the actual plane. By simply shifting the plane, we have reduced the swept sphere/plane intersection test to a ray/plane intersection test (something we already know how to do very easily).

Let us put this in more formal terms:

We have the actual triangle plane defined using the plane equation (shown here in vector form):

$$P \bullet N + d = 0$$

N is the plane normal, *d* is the plane distance from the origin of the coordinate system and **P** is any point on the plane. However, we know that we are not trying to find a point on the triangle plane per se, but rather, we are trying to find the position (the center of our sphere) that is at a distance R from the plane. This is the position we would like to find since it describes the position of the center of the sphere when the surface of the sphere is touching the plane. So let us modify our plane equation to reflect this desire:

$$P \bullet N + d = R$$

For this equation to be true, the point **P** must be a distance of R from the original plane. If we subtract R from both sides of the equation, we zero out the RHS and introduce –R on the LHS:

$$P \bullet N + d - R = 0$$

As you can see, we now have the equation for the shifted plane (the original plane shifted by –R).

Next, let us substitute our ray into this equation (in place of **P**) and solve for *t*. Notice that we have placed (d – R) in brackets. This helps to more clearly identify that this is now a term describing the distance of the **shifted** plane from the origin of the coordinate system:

$$(O + tV) \bullet N + (d - R) = 0$$

where:

O = Initial Sphere Center
V = Sphere velocity vector

Now we will manipulate the equation using a bit of algebra and solve for *t*. Remember, we want to remove everything except *t* from the LHS of the equation.

$$(O + tV) \bullet N + (d - R) = 0$$

Let us first multiply out the brackets surrounding the ray by dotting the terms ***O*** and ***tV*** by ***N***:

$$O \bullet N + tV \bullet N + (d - R) = 0$$

Next we will subtract the term (*d* - *R*) from both sides, which removes it from the LHS and negates it on the RHS:

$$O \bullet N + tV \bullet N = -(d - r)$$

Now we will subtract $O \bullet N$ from both sides to remove it from the LHS. This obviously introduces the term $-O \bullet N$ on the RHS.

$$tV \bullet N = -(O \bullet N) - (d - R)$$

t is still multiplied by $V \bullet N$ on the LHS, so let us divide both sides by $V \bullet N$ leaving *t* alone on the LHS:.

$$t = \frac{-(O \bullet N + d) - R))}{V \bullet N}$$

Notice that we tidied up the numerator a little bit by moving ***+d*** from one set of negated brackets on the right into the other on the left. This way the coefficients for the original plane (the un-shifted plane) are grouped together as we are used to seeing them.

This can alternatively be written as:

$$t = -\frac{(O \bullet N + d) - R}{V \bullet N}$$

So what does this equation tell us? It tells us that all we have to do is calculate the distance from the ray origin to the triangle's plane and then subtract the sphere radius from this distance. This gives us the distance to the shifted plane in the numerator. We then divide this shifted plane distance by the length of the ray projected onto the plane normal. Of course, if you compare our final equation to the equation for the ray/plane intersection discussed in the previous section, you will see that the two are almost identical with the exception of the subtraction of R in the numerator. This small change allows the numerator to calculate the distance from the ray origin to the shifted plane instead of to the actual triangle plane.

Our swept sphere/plane intersection routine will be responsible for returning a *t* value. The calling function can then calculate the new position of the center of the sphere by performing:

New Sphere Center = Sphere Center + (t * Velocity)

where *t* is the value returned from the intersection routine we are about to write. It can be seen that because this intersection point lies on the shifted plane, it is at a distance of exactly Radius units from the actual plane and thus the sphere's surface is resting on the plane of the triangle at this time. Obviously, if the ray does not intersect the shifted plane, then the function should return false and the above step skipped.

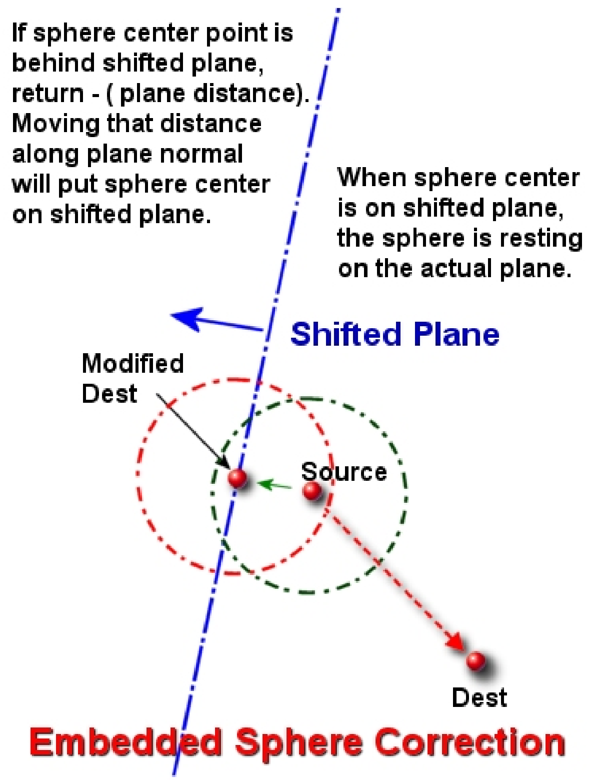

If sphere center point is behind shifted plane, return - (plane distance). Moving that distance along plane normal will put sphere center on shifted plane.
When sphere center is on shifted plane, the sphere is resting on the actual plane.
Shifted Plane
Modified Dest
Source
Dest
Embedded Sphere Correction

Figure 12.28

We are going to have to be vigilant about attempting to detect cases where things might not go as smoothly as planned. With floating point accumulation and rounding errors being what they are, it is possible that the sphere may have been allowed to become slightly embedded in the plane such that the ray origin actually lies behind the shifted plane of the triangle prior to the test being executed. While our collision detection system exists to stop the sphere ever passing through a triangle, there may be times when it has been moved a miniscule amount and it is partially embedded in the plane, but by a value larger than epsilon such that it is not considered to be on the plane by the intersection routines. We certainly do not want our collision detection system to fail when such an event occurs. It should instead calculate the value we need to move the sphere's center position back along the plane normal so that on function return, it is resting on the shifted plane. We will have then corrected the problem and properly moved the sphere center position back from behind the shifted plane to resting on it, as desired.

Figure 12.28 shows this additional test in action. The original center of the sphere (Source) is already behind the shifted plane when the ray intersection with the shifted plane is carried out. Usually, if the distance to the plane from the ray origin is negative in a ray/plane test, we return false for intersection. However, as Figure 12.28 demonstrates, in this particular case, we wish to calculate the distance from the center point to the plane and move it along the plane normal by that amount so that the center of the sphere is sitting on the shifted plane and we have the intersection point we were looking for. However, how do we know if the center of the sphere is legitimately behind the shifted plane of the triangle and should be corrected?

At this point we do not. Only if the intersection point with the shifted plane is later found to be within the edges of a polygon (in a separate point in polygon test) do we know if this is the case. Then we can update the position of the sphere so that it is no longer embedded. Remember, the ray origin will legitimately be behind thousands of planes as each triangle comprising the scene will lie on a plane. As planes are infinite, they will span the entire game level. We certainly do not want to move the ray origin back onto the plane for every one it is behind, as most will be nowhere near the sphere. What we will do is narrow this down so that we only shift the ray origin back onto the shifted plane if the shifted plane itself is behind the sphere center position by a distance of no more than the radius of the sphere. In the worst cases it will never embed itself in a polygon by more than a fraction of that amount in a single update. This way we will not be performing this step needlessly for every shifted plane of every triangle which the sphere center position lays behind. Of course, we will still be needlessly moving it back for many planes since we really do not know, until we perform the point in poly tests later, whether this intersection point is within the edges of a polygon on that plane and whether the position of the sphere should really be updated in this way.

Let us now put to the test everything we have learned and examine the code to a function that performs an intersection test between a moving sphere and a plane. The following code is the exact code used in our collision system for this type of intersection. As discussed a moment ago, our intersection routines do not return the actual intersection point (a vector) but instead return only the *t* value. You will see in a moment that it is the parent function of our collision detection phase that uses the *t* value returned from its intersection routines to generate the final position of the sphere at the end of the process.

Notice first how we calculate the numerator of our equation (the distance to the shifted plane). We calculate the distance to the plane by dotting a vector from the sphere center to the point on the plane (passed as a parameter) and then subtract the radius of the sphere (just as our equation told us to do).

```
bool CCollision::SphereIntersectPlane( const D3DXVECTOR3& Center,
                                       float Radius,
                                       const D3DXVECTOR3& Velocity,
                                       const D3DXVECTOR3& PlaneNormal,
                                       const D3DXVECTOR3& PlanePoint,
                                       float& tMax )
{
    float numer, denom, t;

    // Setup equation
    numer = D3DXVec3Dot( &(Center - PlanePoint), &PlaneNormal ) - Radius;
    denom = D3DXVec3Dot( &Velocity, &PlaneNormal );

    // Are we already overlapping?
    if ( numer < 0.0f || denom > -0.0000001f )
    {
        // The sphere is moving away from the plane
        if ( denom > -1e-5f ) return false;

        // Sphere is too far away from the plane
        if ( numer < -Radius ) return false;

        // Calculate the penetration depth
        tMax = numer;

        // Intersecting!
        return true;

    } // End if overlapping

    // We are not overlapping, perform ray-plane intersection
    t = -(numer / denom);
    // Ensure we are within range
    if ( t < 0.0f || t > tMax ) return false;

    // Store interval
    tMax = t;

    // Intersecting!
    return true;
}
```

Notice the conditional code block in the middle of the function that handles the embedded case. As you can see, if the numerator is smaller than zero it means the distance to the shifted plane from the sphere center is a negative value. Also, if the denominator is larger than zero (with tolerance) it means the velocity vector is moving away from the plane and we just return false. We also return false if the distance from the ray origin to the shifted plane is larger than the radius (< -Radius). As discussed, planes that are this far away can safely be removed from consideration since the sphere could not have possibly embedded itself into a polygon this much in a single frame update. We then store the negative distance to the plane (the numerator) in the referenced parameter tMax so the caller will have access to this distance. This will describe to the caller the distance the sphere center should be moved backwards to rest on the shifted plane. In other words, it tells us how far back to move the sphere position so that it is no longer embedded in this plane.

Outside the embedded case code block we perform the usual calculation of the *t* value to generate a parametric value along the ray for the point of intersection. This is then returned to the caller via the tMax reference. Once again, in this section you can see that if t < 0.0 then we return false, since the intersection is behind the ray. Notice however that we also return false if tMax is smaller than the *t* value we just calculated. As you can see, tMax is not only an output, but it is also an input to the function.

To understand this, remember that the detection process will be run for each triangle and that we are only interested in discovering the closest intersection. Therefore, every time this function is called, it will only return true if the *t* value is smaller (closer to the sphere center point) than a previously calculated *t* value generated in a test with another triangle's plane. Thus, when we have tested every triangle, tMax will contain the *t* value for the closest colliding triangle.

Finally, if the sphere is not embedded in the plane, tMax will contain the parametric *t* value (0.0 to 1.0 range) on function return. But if the sphere *is* embedded, tMax will contain the actual distance to the shifted plane along the plane normal. The calling function will need to make sure it uses this information correctly when generating the intersection point on the plane, as shown next:

```
    if ( !SphereIntersectPlane( Center, Radius, Velocity,
                                TriNormal, v1, t )) return false;

    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);
```

In the above code, v1 is the first vertex in the triangle currently having its plane tested (i.e., the point on plane).

If input/output parameter *t* is negative on function return then it contains the actual distance to the plane. The point of collision on the shifted plane is calculated by moving the sphere center *t* units along the plane normal (we negate *t* to make it a positive distance). This moves it backwards so that the sphere center is now on the shifted plane and its surface is no longer embedded. If *t* is not negative then it is within the 0.0 to 1.0 range and describes the position of intersection (the new center point of the sphere) parametrically. Therefore, we simply multiply the velocity vector with the *t* value and add the result to the initial position of the center of the sphere. The end result is the same -- a position on the shifted plane which correctly describes what the position of the sphere should be such that its surface is resting on, but not intersecting, the triangle plane.

12.5 The SphereIntersectTriangle Function: Part I

Our collision system will invoke the collision detection phase by calling the CCollision::EllipsoidIntersectScene method. We will see later how this method will transform all the potential intersecting triangles into eSpace. It will then loop through each triangle and test it for intersection with the swept sphere. To test each triangle, it calls the CCollision::SphereIntersectTriangle method, which we will begin to construct in this section. This function will be a collection of different intersection techniques that we will implement. Below, we see the first part of this function using the intersection routine we discussed in the previous section.

```
bool CCollision::SphereIntersectTriangle( const D3DXVECTOR3& Center,
                                          float Radius,
                                          const D3DXVECTOR3& Velocity,
                                          const D3DXVECTOR3& v1,
                                          const D3DXVECTOR3& v2,
                                          const D3DXVECTOR3& v3,
                                          const D3DXVECTOR3& TriNormal,
                                          float& tMax,
                                          D3DXVECTOR3& CollisionNormal )
{

    float       t = tMax;
    D3DXVECTOR3 CollisionCenter;
    bool        bCollided = false;

    // Find the time of collision with the triangle's plane.
    if ( !SphereIntersectPlane( Center, Radius, Velocity, TriNormal, v1, t ))
       return false;

    // Calculate the sphere's center at the point of collision with the plane
    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);

    // If this point is within the bounds of the triangle,
    // we have found the collision
    if ( PointInTriangle( CollisionCenter, v1, v2, v3, TriNormal ) )
    {
        // Collision normal is just the triangle normal
        CollisionNormal = TriNormal;
        tMax            = t;

        // Intersecting!
        return true;

    } // End if point within triangle interior

    // We will need to add other tests here

}
```

This function is called for each collidable triangle and will be passed (among other things) a reference to a variable tMax which will be used to test for closer intersections. It will only return true if the *t* value of intersection (if it occurs) is smaller than the one already stored in tMax. The three vertices of the triangle (just 3D vectors) are passed in the v1, v2, and v3 parameters.

The first thing the function does is call the SphereIntersectPlane function we just wrote to test if the sphere intersects the plane of the triangle. Notice how we pass in the triangle normal and the first vertex of the triangle as the point on the plane (any vertex in the triangle will do). If the SphereIntersectPlane function returns false, then the SphereIntersectTriangle function can return false as well. If the sphere does not intersect the plane, it cannot possibly intersect the triangle on that plane. If the SphereIntersectPlane function returns true, the sphere did intersect with the plane and we continue.

The next thing we do is calculate the intersection point on the shifted plane. We can think of this position as the new position we should move the center of the sphere to so that its surface is resting on, but not intersecting, the triangle's plane. Of course, we still do know yet whether this collision point is within the triangle itself, so we must perform a Point in Triangle test. If it is inside, then we have indeed found the closest intersecting triangle so far and can return this new *t* value to the caller, which it can then use to update the sphere's position. Keep in mind that this function is called for each triangle, so we may find a triangle in a future test which is closer, overwriting the tMax value we currently have stored. We can also return the triangle normal as the intersection normal (for later use when creating the slide plane).

One thing that may have struck you as we have gone through this discussion is that we use the Point in Triangle function to test if the intersection point on the shifted plane is contained inside the polygon's edges. But the intersection point is on the shifted plane, not on the actual plane of the triangle. We can think of the intersection point as hovering above the actual polygon's plane at a distance of radius units (see Figure 12.29).

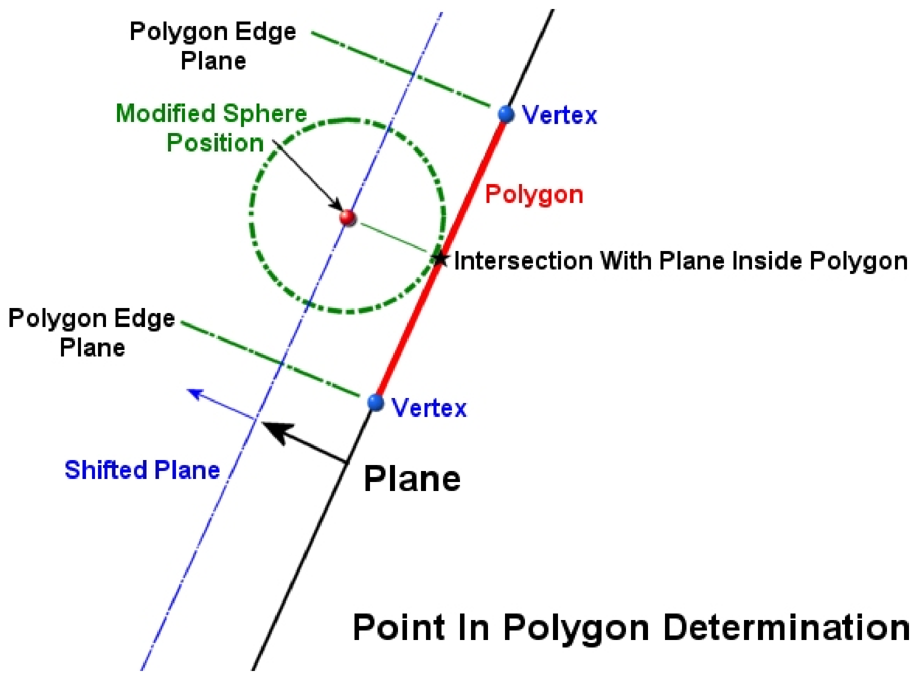


Figure 12.29

In Figure 12.29, the intersection point on the shifted plane describes the new non-intersecting position of the sphere center point. But it is not actually *on* the polygon's plane, so it raises the question as to whether or not we can use our Point in Triangle routine? The answer is yes. Recall that our Point in Polygon/Triangle routine returns true if the point is enclosed within the infinite planes formed by each edge of the polygon. Therefore, it is not necessary for the point to actually lie on the plane for the function to return true. If you refer back to Figure 12.22 you can see that a point hovering above the actual plane of the hexagonal polygon would still be inside the bounds of the infinite cylinder formed by the edge planes. In Figure 12.29, we see that the intersection point on the shifted plane (Modified Sphere Position) is indeed between the two edge planes, even though the point itself does not rest on the polygon's plane. Therefore, our Point in Triangle function would still correctly return true.

Unfortunately, our SphereIntersectTriangle method currently only handles the simplest and most common intersection case – when the sphere surface collides with the interior of the polygon. However, this is not the only case we must test for, and you will notice at the bottom of the code listing shown above we have highlighted a comment that suggests additional tests will need to be performed before we can rule out a collision between the sphere and the triangle currently being tested.

Figure 12.30 demonstrates the case where the sphere does intersect the plane and does intersect a polygon on that plane. However, the sphere does not intersect the *interior* of the polygon; it collides with one of its edges. Thus, edge tests are going to be required as well. What this essentially means is that our routine will go as follows… First we test to see if the sphere intersects the plane. If it does not, we can return false as shown in the above code. Then test to see if the intersection point on the plane is within the interior of the polygon (also shown above). If it is, then we have found the collision point and no further tests need to be performed. We simply return the *t* value of intersection. However, if the sphere did intersect the plane, but the intersection point is not within the interior of the polygon, we must test for an edge intersection (see Figure 12.30) because a collision may still occur.

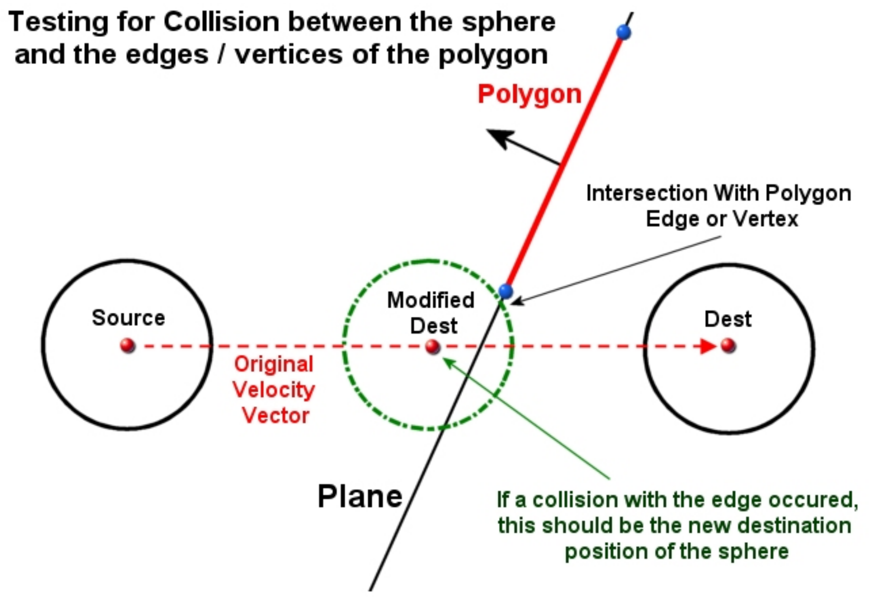

Testing for Collision between the sphere and the edges / vertices of the polygon
Polygon
Intersection With Polygon Edge or Vertex
Source
Modified Dest
Dest
Original Velocity Vector
Plane
If a collision with the edge occured, this should be the new destination position of the sphere

Figure 12.30

Unfortunately, testing for collisions between a swept sphere and a polygon edge is not as simple as our other cases. The problem is that we can no longer rely on the fact that the first point of intersection on the surface of the sphere will be found at a distance of Radius units along the negated plane normal from the sphere center point. As Figure 12.30 demonstrates, at the time the sphere intersects the edge of the polygon, the sphere has (legally) passed through the plane. While the intersection point is still clearly at a distance of Radius units from the center of the sphere, it is no longer determined by tracing a line from the sphere center (Modified Dest) in the direction of the negated plane normal. That was how our previous test worked, but it will obviously not work here. When we call the SphereIntersectPlane function it will calculate the point at which the sphere first touches the plane. This point will not be inside the polygon, so the Point in Polygon test will return false. Thus, if the initial Point in Polygon test returns false, we must perform some more complicated edge tests before we can safely conclude that no collision has occurred.

We will see later that tackling such a problem naturally leads us to a quadratic equation that will need to be solved. Because many students have not brushed up on quadratic equations since high school, the next section provides a quick refresher on quadratic equations and related ideas (including some very rudimentary mathematics like fraction multiplication and division, negative numbers, etc.). If you feel confident with respect to your ability to manipulate quadratic equations, get them into the standard form, and then solve them, please feel free to skip the next section. You should bear in mind that the remaining intersection techniques we examine will require that you have a good understanding of quadratic equations, so reading the next section is highly recommended if you are a little rusty.

12.6 Quadratic Equations

In this section we will discuss what quadratic equations are and how to solve them. If you have never encountered quadratic equations before then you will have to take a leap of faith for the time being and just believe us when we say that you will find the ability to solve them incredibly useful in your programming career. But before we get too far ahead of ourselves, we first have to discuss what a quadratic equation is and talk about a generic means for solving them. Only then will you be able to see why they are so useful in our collision detection code. Therefore, although this section will be talking about quadratic equations in an abstract manner initially, please stay with us and trust that their utility will become apparent as we progress with the lesson.

Hopefully we all recall that an equation is a mathematical expression stating that two or more quantities are the same as each other. You should also remember from high school algebra that a polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients (i.e., constants). For example, the following polynomial involves the sum of powers in one variable *t*. Polynomials are usually written such that the terms with the highest powers are arranged from left to right. This allows the degree of the polynomial to be more easily recognizable. In this example, we have a polynomial of the 3rd order because it involves a term that raises the unknown variable *t* to a maximum power of 3.

$$t^3 + at^2 + bt + c = 0$$

This is referred to as a *cubic polynomial* because the highest power of the unknown *t* is 3. **a**, **b** and **c** are referred to as *coefficients* (or constants) and contain the 'known' values, whatever they may be. Solving the above equation essentially means finding a value for *t* (i.e., the 'unknown') which makes the equation true. As you can imagine, it is no longer as easy to isolate *t* on the LHS of the equation (as it was with a linear equation such as the plane equation for example) since the unknown is now involved in multiple terms where it is being multiplied by coefficients.

A **quadratic** equation is a polynomial equation in which the highest power of the **single** unknown variable *t* is 2 and the coefficient of the squared term **a** is not zero. The **standard form** of the quadratic equation is the arrangement when the RHS is set to zero and the terms are stated as:

$$at^2 + bt + c = 0 \quad \textit{\textbf{(Standard Form)}}$$

As you can see, if **a** was set to zero, this would cancel out the squared term (**t^2**) reducing the equation to a linear equation in the form:

$$bt + c = 0$$

Thus, the coefficient of the squared term must be non zero in order for it to be a quadratic equation.

Note: The name 'Quadratic' given to such an equation is often the subject for much confusion. We know that the word 'Quad' means four and yet it is used to describe an equation which involves the 2nd power and not the 4th. But there is another meaning for the word 'quad' which helps us make sense of why an equation of this type is called a Quadratic. Although the Latin word 'Quadri' does indeed mean four, the word 'Quadrus' means 'a square' -- a shape that has four sides. Quadratic equations arose from geometry problems involving squares, and as we know, the second power is also referred to as a square. Another Latin word 'Quadratus' means 'Squared', and several other words derive from this. For example, another name for the 'Square Dance' is the 'Quadrille' and the word 'Quadrature' (through the French language) describes the process of constructing a square of a certain area. Therefore, 'Quadratic' refers to an equation involving a single variable in which the highest power is raised to the power of two and is therefore said to be squared.

Not all quadratic equations are readily arranged into the shape the of the standard form $at^2 + bt + c = 0$. For example, the following list shows some quadratic equations which look initially to be using very different shapes. Notice however, that the single unknown (**x**) is still raised to the second power in one of the terms.

$$x^2 = 81$$

$$6x^2 + x = 100$$

$$-x^2 - 100 = 20$$

$$(x+3)^2 = 144$$

While these equations are all quadratic equations in **x**, they are not currently in the standard form. Later we will discuss the *quadratic formula*. This is a technique which can always be used to find the solutions to a quadratic equation that has been arranged in the standard form. Thus, in order to use the quadratic formula, we must first apply some algebraic manipulation to a quadratic equation to get it into the shape of the standard form $at^2 + bt + c = 0$. In the standard form, information we know about is stored in the coefficients **a**, **b** and **c** and t is the value we are trying to find. We need to be able to feed the correct **a**, **b** and **c** values into the quadratic formula so that it can be used to solve for the unknown t (**x** in the above list of examples).

Let us see how the above equations could be arranged into the standard form so that we could solve them using the quadratic formula. For now, do not worry about what the quadratic formula is or how we solve the above equations; for the moment, we are just going to focus on taking a quadratic equation in various forms and manipulating it into the standard form. Later on we will address the quadratic formula and find out how it can be applied to solve such equations.

<u>**Example 1:**</u> **Representing $x^2 = 81$ in standard form.**

The standard form has zero on the RHS of the equals sign, so let us first subtract 81 from both sides of the equation to make that true. Our equation now looks like this:

$$x^2 - 81 = 0$$

Notice that in our equation **x** is used as the label for the unknown, while in the standard form ***t*** is generally used. This really is just a mater of preference, so you can think of **x** in our equation as representing ***t*** in the standard form. Sometimes, **x** is also used to represent the unknown in the standard form. As you recall, in algebra, these are all just placeholders, so we can use whatever letters we wish (although it is helpful to follow convention, since it makes it easier for others to pick up on what is happening).

Now, if we look at the standard form $at^2 + bt + c = 0$, you should see that we essentially have what we are looking for. We know that in our current equation the squared term is not multiplied by anything (it has no **a** coefficient like the standard form). We also know that this is equivalent to having an **a** coefficient of 1 in the standard form.

$$1x^2 - 81 = 0$$

So we can see that **a=1** in this example. Our equation also has no **bt** (**bx**) term like the standard form does. This is equivalent to having a **b** coefficient of zero. Obviously, having a **b** value of zero would cancel out any contribution of the middle term, as shown below.

$$1x^2 + 0x - 81 = 0$$

We now have our equation in the standard form where **a**=1, **b**=0 and **c**= -81 as shown below. Compare it to the standard form:

$$1x^2 + 0x - 81 = 0$$

$$at^2 + bt + c = 0$$

If we wanted to use the quadratic formula to solve this equation we could simply feed in the values, **a**=1, **b**=0 and **c**=-81. Of course, $\mathbf{1x^2}$ is the same as just $\mathbf{x^2}$, and in the middle term, **0x** evaluates to nothing and the term is completely cancelled out. This can also be written as follows, which is the original equation we had:

$$x^2 - 81 = 0$$

or

$$x^2 = 81$$

To be sure, this is a very simple quadratic equation to solve, even without using the quadratic formula. But if you wish to solve it using the quadratic formula, you could use the inputs **a**=1, **b**=0 and **c**=-81, as we have just discovered.

Example 2: Representing $6x^2 + x = 100$ in standard form.

First, let us subtract the 100 from both sides of the equation to leave zero on the RHS as the standard form dictates:

$$6x^2 + x - 100 = 0$$

This example is actually a little easier to spot the relationship with the standard form. The standard form starts with at^2 and therefore we can see right away that **a** = 6. We can also see that the second term of the standard form is bt (***bx*** in our equation). Since bx is the same as x if **b** = 1, we know that the lack of a **b** coefficient in our current equation's middle term must mean that **b** = 1. We can also see that the removal of 100 from the RHS has introduced a **c** coefficient of –100 on the LHS, giving us the values **a**=6, **b**=1 and **c**= -100 in the standard form:

$$6x^2 + 1x - 100 = 0$$

$$at^2 + bt + c = 0$$

Therefore, we could solve the quadratic equation $6x^2 + x = 100$ by plugging the values **a=6**, **b=1** and **c=-100** into the quadratic formula to solve for **x**.

Example 3: Representing $-x^2 - 100 = -60$ in standard form.

Once again, the first thing we do is add 60 to both sides of the equals sign to leave us with zero on the RHS, as the standard form dictates.

$$-x^2 - 100 + 60 = 0$$

Currently, we have a **c** coefficient of (-100+60) and we can instantly simplify by adding +60 to –100 to get an equation with an equivalent **c** coefficient of -40:

$$-x^2 - 40 = 0$$

The lack of any middle term *bt* (or in our case, **bx**) means that **b** = 0, canceling that term out altogether. The lack of any coefficient next to the squared term must also mean **a** = -1 (as -1*x^2 is just –x^2). Therefore, in the standard form, our equation looks like the following (once again, shown alongside the standard form template for comparison):

$$-1x^2 + 0x - 40 = 0$$

$$at^2 + bt + c = 0$$

Thus, in order to solve our original equation $-x^2 - 100 = -60$ using the quadratic formula, we would feed in the values **a**=-1, **b**=0, **c**=-40.

<u>Example 4:</u> Representing $(x+3)^2 = 144$ in standard form.

This example looks quite different from any we have seen so far. Is this still a quadratic equation? Well the answer is yes even though it is being shown in its squared form on the LHS. To understand why it is indeed a quadratic, let us see what happens to the LHS if we multiply out the square. This will bring to our attention two **very important** algebraic identities.

We know that if we have the term i^2, then the term can be evaluated by performing $i * i$ .You could say that we have multiplied the square out of the term, since we now have the value of $i * i$ and no longer have a power in that term. So how do we multiply the square out of a term like $(o+e)^2$? Well, we just treat the content inside the brackets (the squared term) as the single term '*i*'. Therefore, instead of doing *i* * *i*, we do:

$$(o+e)(o+e)$$

The next question concerns how we multiply the expression (o+e) by the expression (o+e). Hopefully most of you will remember learning about multiplying bracketed terms in high school. In particular you may remember the acronym FOIL. FOIL is used as an aid to remember the multiplication order for bracketed terms. The acronym stands for the words "**F**irst **O**uter **I**nner **L**ast and describes the paired terms that have to be multiplied. Let us do what it says and see what happens.

First we multiply the First terms in each bracket:

(o + e) (o + e) = o * o = o^2

Next we multiply the Outer terms of each bracket:

(o + e) (o + e) = oe

Then we multiply the Inner terms of each bracket:.

(o + e) (o + e) = eo (which is the same as **oe)**

And finally, we multiply the Last terms in each bracket:

(o + e) (o + e) = e^2

Adding the results of each separate multiplication together, we get the final expanded version of $(o+e)^2$:

$$o^2 + oe + eo + e^2$$

Since **oe** and **eo** are equivalent, we are just adding this value to the expression twice. Therefore, we can collect those terms and rewrite as follows:

$$o^2 + 2eo + e^2$$

And there you have it. We have expanded the original squared term out of its brackets. We have not changed the value of the expression in any way, but we have now discovered one very important algebraic identity. That is, we have found an equivalent way of writing the expression $(o+e)^2$. The three term polynomial $o^2 + 2eo + e^2$ is referred to as being a *perfect square trinomial* (PST) as it can instantly be written in its 'easier to solve' squared form of $(o+e)^2$.

Identity 1: $(o+e)^2 = (o+e)(o+e) = o^2 + 2eo + e^2$

We will see later when deriving the quadratic formula that this identity is very important. Remember it and make sure you understand it. It will become critical for you to know that when you have a LHS of an equation that looks like this: $o^2 + 2eo + e^2$, that it can also be written like this: $(o+e)^2$ without changing its evaluation. Being able to recognize patterns like this will help you understand how quadratic equations are solved and how to manipulate quadratic equations into the standard form.

Another very important identity that is used often in solving quadratic equations is very similar to the last one except it has a negative in the initial bracketed expression:

$$(o-e)^2$$

As we know, this is equivalent to multiplying the term (o-e) by itself:

$$(o-e)(o-e)$$

FOIL can once again be applied to multiply out the brackets.

First we multiply the First terms in each bracket.

$\mathbf{(o - e) (o - e) = o * o = o^2}$

Next we multiply the Outer terms of each bracket. Here we are multiplying positive **o** with negative **e**. A positive times a negative gives us a negative, so:

$\mathbf{(o - e) (o - e) = -oe}$

Now we multiply the Inner terms of each bracketed expression. This time we are multiplying negative **e** with positive **o** and once again we get a negative:

(o - e) (o - e) = -eo (which is the same as **-oe)**

Finally, we multiply the Last terms in each bracket. This time we are multiplying negative **e** by negative **e** which we know produces a positive:

(o - e) (o - e) = e^2

Adding the results of each separate multiplication together we get the final expanded version of $(o-e)^2$:

$$o^2 - oe - eo + e^2$$

Since **oe** and **eo** are equivalent, we are essentially subtracting **oe** from the expression twice, so we can collect those terms and rewrite as:

$$o^2 - 2eo + e^2$$

This is another very important algebraic identity. Learn it and remember it. It is important to recognize the shape of an expression such as $o^2 - 2eo + e^2$ and know that it can also be written as $(o-e)^2$. That is, $o^2 - 2eo + e^2$ is also a perfect square trinomial (PST) that can instantly be represented in its 'easier to solve' squared representation $(o-e)^2$.

Identity 2: $(o-e)^2 = (o-e)(o-e) = o^2 - 2eo + e^2$

Now we will get back to the task at hand and determine how we can represent our final example equation $(x+3)^2 = 144$ in standard form.

First, we know from the algebraic identities we just discovered that $(x + 3)^2$ can also be rewritten as shown in Identity 1. In our example, x is equivalent to the o variable in the identity and 3 is equivalent to the e variable.

Therefore, if

$$(o+e)^2 = (o+e)(o+e) = o^2 + 2eo + e^2$$

then

$$(x+3)^2 = (x+3)(x+3) = x^2 + 2x3 + 3^2$$

Our equation now looks as follows, when compared to the standard form:

$$x^2 + 2x3 + 3^2 = 144$$

$$at^2 + bt + c = 0$$

We are not quite there yet. However, our middle term (**2*x*3**) can also be written as **2*3*x**, which is the same as **(2*3)*x**. So we can simplify our term to **6x**. We also have no **a** coefficient, which means it is equivalent to having an **a** coefficient of 1. So let us reorder things a little and see what we get:

$$1x^2 + 6x + 3^2 = 144$$

$$at^2 + bt + c = 0$$

Nearly there! We currently have a **c** term of 3^2, which evaluates to 3*3=9:

$$1x^2 + 6x + 9 = 144$$

$$at^2 + bt + c = 0$$

The only difference now between the current form of our equation and the standard form is that we have a value of 144 on the RHS and the standard form has a value of zero. Therefore, we must subtract 144 from both sides of the equation so that we are left with 0 on the RHS:

$$1x^2 + 6x + 9 - 144 = 0$$

When shown along side the standard form we can see that we currently have a **c** term of (+9-144). Therefore, our **c** coefficient evaluates to 9-144 = -135. Below, we see the final version of our equation in the standard form.

$$1x^2 + 6x - 135 = 0$$

$$at^2 + bt + c = 0$$

If we wanted to use the quadratic formula to solve the quadratic equation $(x+3)^2 = 144$, we would feed in the known values: **a=1 , b=6 c=-135.**

This may all sound very abstract at the moment, since we are simply manipulating an equation into the standard form. But the standard form of the quadratic equation is very important to us because if we can get the quadratic equation we need to solve into this shape, we can use the quadratic formula to solve it.

Thus, if we wish to solve a problem that naturally gives rise to a quadratic equation, and we can manipulate the information we *do* know about the problem into the **a**, **b** and **c** coefficients, then we can use the quadratic formula to solve for the unknown *t*. This means we can use the quadratic formula to solve *any* quadratic equation in exactly the same way, so long as we mold the problem into the shape of the standard form first.

We shall see later that our collision system must perform an intersection test between a ray and a sphere. The equation of a sphere (which belongs to a family known as Quadric Surfaces) involves squared terms and as such, when we substitute the ray equation into the sphere equation of the sphere to solve for *t* (time of intersection along the ray), this naturally unravels into an equation where the unknown *t* is in a squared term. Therefore, we will have ourselves a quadratic equation that needs to be solved. When this is the case, we can manipulate the equation into the standard form and then solve for *t* using the quadratic formula giving us the time of intersection along the ray. Essentially, solving a quadratic equation boils down to nothing more than isolating t^2 on the LHS and finding the square roots of that term. We will learn more about this later.

12.6.1 Solutions to Simple Quadratic Equations

The quadratic formula we will study later in this lesson is not the only method for solving quadratic equations. Other approaches include *factoring* and a method called *completing the square*. Since factoring will not work with all quadratic equations, we will not be using this method. Instead we will concentrate on the methods that work reliably in all circumstances. It is also worth noting that we do not have to manipulate a quadratic equation into the standard form to solve it; we only need to do this if we wish to use the quadratic formula. Many quadratic equations are simple to solve and as such, manipulating them into the standard form and then using the quadratic formula would be overkill. So before we look at how to use the quadratic formula and examine its derivation, we will look at some simple quadratic equations that can be solved in a very simple way. This will allow us to examine the properties of a quadratic equation and the solutions that are returned.

Example 1: $x^2 = 81$

We examined this exact equation earlier and discussed how to represent it in standard form so that the quadratic formula could be used to solve it. Of course, this quadratic is extremely simple to solve without manipulation into standard form and indeed using such a technique would certainly be taking the long way round.

We know that we intend to find the value of x and we currently have x^2 on the LHS. We also know that the square root of x^2 is simply x. Therefore, to isolate x on the left hand side, we can just take the square root of both sides of the equation. Remember, it is vitally important that whatever you do on one side of the equals sign must be done on the other side too.

Step 1: Original equation is in its squared form

$$x^2 = 81$$

Step 2: Remove the squared sign from x by taking its square root. Remember to take the square root of the RHS as well.

$$\sqrt{x^2} = \sqrt{81}$$

Step 3 : The square root of x^2 is x and the square root of 81 is 9, so we have:

$$x = 9$$

This is correct, but we have forgotten one important thing. Every positive real number actually has two square roots -- a positive root and a negative root. If it has been a while since you covered this material in high school then that might come as a surprise. After all, if you punch 81 into a calculator and take its square root, you will have one answer returned: 9. Likewise, if you use the C runtime library's sqrt function, it too returns a single root. In both instances, we are getting the positive root of the number. However, we must remember that when we multiply a negative number by a negative number, we get a positive number. Thus, when we square a negative number we get a positive result. This leads to the conclusion that a positive number must have a negative square root as well.

For example, we know that 9 is the square root of 81 because:

$$9^2 = 9 * 9 = 81$$

Clearly 9 is the square root of 81 because 9 squared is 81. But as mentioned, this is actually only one of two square roots that exist (the positive root). –9 is also a square root of 81 because –9 multiplied by itself is 81:

$$-9^2 = -9 * -9 = 81$$

So, we can see that the square roots of 81 are in fact, 9 and –9. This means there are actually two possible solutions for **x** in our equation: 9 and –9. This duality is often indicated using the $\pm$ sign which is basically a plus and a minus sign communicating that there is a negative and positive solution to the square root. Therefore, we should correctly show our solution as:

$$x^2 = 81$$

$$x = \pm\sqrt{81}$$

$$x = \pm 9$$

Thus, there are two answers: **x** = +9 and **x** = –9.

The fundamental theorem of algebra tells us that because a quadratic equation is a polynomial of the second order, it is guaranteed to have two roots (solutions). However, some of these may be degenerate and some of them may be complex numbers. We are only interested in real number solutions in this lesson, so we will consider a quadratic that returns two complex roots as having no roots (and thus no solutions for the equation). If you have never heard of complex numbers, do not worry because we are not going to need to cover them to accomplish our goals. However, you can learn more about complex numbers in our Game Mathematics course if you are interested in pursuing the subject further.

Example 2: $x^2 + 25 = 25$ (A Degenerate Root Example)

This example is interesting because it present a case where only one root exists. Actually, when this happens, we can perhaps more correctly say that two roots are returned, but both the negative and positive roots are the same. Let us solve this equation step by step.

Step 1: Let us first isolate x^2 on the LHS by subtracting 25 from both sides of the equation.

$$x^2 = 0$$

Step 2 : Now to find x we simply take the square root of both sides as shown below.

$$x = \pm\sqrt{0}$$

Now you can probably see where we are going with this. The solutions for x (the roots) are the negative and positive square root of zero. Since the square of zero is always just zero, negative zero and positive zero is just zero. Therefore, we do have two roots, but they are exactly the same. If ever we get to a point when solving a quadratic equation where we have isolated x^2 on the LHS and have zero on the RHS, we know we have a situation where we have multiplicity of the root and both roots evaluate to zero.

Example 3: $x^2 + 25 = 0$ (An example with no **real** solutions.)

This example demonstrates the case where no real numbered solutions can be found for the equation. You can probably spot why this the case right away just by looking at the equation, but if not, let us try to solve it and see what happens.

Step 1: Isolate x^2 on the LHS by subtracting 25 from both sides of the equation. This will cancel out the '+25' term on the LHS.

$$x^2 = -25$$

Step 2: Now we have to take the square root of both sides of the equation to get x isolated on the LHS.

$$x = \pm\sqrt{-25}$$

And here in lies our problem. We cannot take the square root of a negative number and expect a real number result. Just try punching a negative number into a calculator and hitting the square root button; it should give you an error. This is actually pretty obvious when only a few moments ago we said that multiplying a negative by a negative gives us a positive. Therefore, if squaring a negative number gives us a positive number then there can be no real number such that when it is squared results in a negative number. Therefore, a negative number cannot possibly have real square roots. This is actually one of the main reasons that complex numbers were invented. They actually fix this problem through the use of something called imaginary numbers. However, we are not interested in complex numbers in this lesson so we will consider any quadratic equation for which real roots cannot be found, to have no solutions. This is detectable because we know this situation arises as soon as we have x^2 on the LHS and a negative number on the RHS.

<u>**Example 4:**</u> $(x+3)^2 = 144$

This is also an interesting example because it is the first we have tried to solve in which the LHS is a squared expression instead of a single value. It just so happens that a quadratic equation such as this is very simple to solve. Let us go through solving it step by step.

Step 1: The expression x+3 is currently squared on the LHS, so we wish to remove the square. We do this by simply take the square root of both sides of the equation. The square root of $(x+3)^2$ is simply x+3 (we just remove the square sign) and the square root of the RHS (144) is ±12 .

$$x+3 = \pm\sqrt{144}$$

Step 2: This evaluates to:

$$x+3 = \pm 12$$

Step 3: **x** is still not isolated on the LHS, so we can fix that by subtracting 3 from both sides of the equation:

$$x = -3 \pm 12$$

Therefore, the two solutions for **x** are

$$x = -3 + 12 = 9$$

and

$$x = -3 - 12 = -15$$

So:

x = *9* and ***x*** = *-15*

At this point you may be wondering how we know which solution is the one we actually need to use. For example, when we solve a quadratic equation to find a *t* value along the ray at which an intersection occurs with a sphere, we will only be interested in the positive solution. If we think of an infinite ray passing through the center of the sphere however, we can see how it will intersect with the surface of the sphere twice; once on the way in and once again on the way back out. In fact, this is a property that defines a surface as being a quadric surface (a family of surfaces to which spheres, ellipsoids and cylinders belong).

If we imagine stepping along a ray, we would be interested in using the smallest solution, since this will be the first point of intersection along that ray. That is to say, we are interested in the time the ray first intersects the sphere. We will generally have no interest in the second solution which gives a *t* value for the ray passing out the other side of the sphere.

However, it is worth noting that we are interested in the smallest positive root. A negative *t* value describes an intersection occurring behind the ray origin. If we imagine a situation where our ray origin is inside the sphere to begin with, we will have a positive and a negative solution. The positive solution will be the point at which our ray intersects the far side of the sphere, and the negative solution will describe the distance to the surface backwards along the ray. However, it really does depend on what you are trying to do and the quadratic you are trying to solve. For example, you may well be interested in finding both intersection points with an infinite ray and a sphere. However, if the above example was returning intersection distances between our ray and the sphere surface, our collision system will only be interested in using the positive solution and we will discard information about intersections that have occurred behind the ray origin (more on this later).

Example 5: $x^2 + 6x + 9 = 144$ (Solving a Perfect Square Trinomial)

This example looks a lot harder to solve than the previous one where the LHS was already represented as a squared expression. Now you will find out why we discussed those two algebraic identities earlier in such detail and why it is vital that you recognize them.

Take a look again at Identity 1 that we discussed earlier:

Identity 1: $o^2 + 2eo + e^2 = (o + e)^2$

Now let us look at $o^2 + 2eo + e^2$ along side the LHS of our equation.

$$o^2 + 2eo + e^2$$
$$x^2 + 6x + 9$$

We can see that **x** in our equation is equivalent to **o** in the identity. We can also see that **2e** in the identity is equal to **6** in our equation. Of course, if **2e = 6** then **e** must equal 3. The identity has $\mathbf{e^2}$ as its third term and we know that in our equation **e = 3** and that $\mathbf{3^2 = 9}$. So our equation perfectly matches the shape of $(o + e)^2$ when expanded out of its brackets. Thus, our equation is a perfect square trinomial that can be readily written in its squared form:

If

$$o^2 + 2eo + e^2 = (o + e)^2$$

then

$$x^2 + 6x + 9 = (x + 3)^2$$

This means that our original equation:

$$x^2 + 6x + 9 = 144$$

can also be written as

$$(x + 3)^2 = 144$$

While this equation initially looked quite different from the last example, we now see that it is exactly the same equation. We can prove this by performing FOIL on (x+3)^2:

First we multiply the First terms in each bracket.

(x + 3) (x + 3) = x * x = x^2

Next we multiply the Outer terms of each bracket

(x + 3) (x + 3) = 3x

Now we multiply the Inner terms of each bracket.

(x + 3) (x + 3) = 3x

And finally, we multiply the Last terms in each bracket.

(x + 3) (x + 3) = 9

Adding the results of each separate multiplication together we get the final expanded version of $(x + 3)^2$:

$$x^2 + 3x + 3x + 9$$

As we are adding **3x** to the LHS expression twice, we can collect these like terms and rewrite the LHS as the following:

$$x^2 + 6x + 9$$

As you can see, this is the original LHS of the equation we started with.

Now that we have identified (and re-checked using FOIL) that we can write the LHS in its squared form, solving the equation becomes no different from the previous example with the steps shown below.

Step 1: Our original equation.

$$x^2 + 6x + 9 = 144$$

Step 2: Identify that the equation is a PST (perfect square trinomial) and can be readily written in squared form on the LHS.

$$(x + 3)^2 = 144$$

Step 3: Take the square root of both sides of the equation, removing the power from the LHS.

$$x + 3 = \pm\sqrt{144}$$

Step 4: This evaluates to + and – 12 on the RHS

$$x + 3 = \pm 12$$

Step 5 : Subtract 3 from both sides to isolate **x** on the LHS

$$x = -3 \pm 12$$

This yields two solutions for **x**:

$$x = -3 + 12 = 9 \text{ and } x = -3 - 12 = -15$$

In this example we have illustrated the importance of recognizing when the LHS can be written in its squared form. This boils down to nothing more than pattern matching if your LHS is already a PST. As long as you can make this connection with the two algebraic identities discussed earlier, you can immediately represent the LHS of the equation in its much simpler squared form. This makes it much easier to find the solution and isolate the unknown.

12.6.2 Completing the Square

We have seen that it is easy to find the solution to a quadratic equation that is in the form $(x+3)^2$ on the LHS. We have also demonstrated how the LHS of a quadratic equation in the form $o^2 + 2eo + e^2$ is a perfect square trinomial and can be written as $(o+e)^2$ and easily solved. For example, we identified that the LHS of an equation in the form of $x^2 + 6x + 9 = 144$ can readily have its LHS written as $(x+3)^2 = 144$ because the LHS represents a polynomial that can be readily written as a squared expression.

The next question is, how do we solve an equation whose LHS is not a perfect square trinomial and therefore cannot be readily written as an expression in squared form? For example, how would we a find the solutions to a quadratic equation whose LHS is in the form: $2x^2 + 12x$. This equation's LHS is clearly not a PST in the form of $o^2 + 2eo + e^2$ and thus can not be written in $(o+e)^2$ form.

<u>**Example 6:**</u> $2x^2 + 12x = 0$ (Solving a Quadratic Equation that is not a square.)

A quadratic equation in this form is not so easy to solve because the LHS is not a perfect square. Therefore, we have to manipulate the equation so that the LHS becomes a perfect square trinomial allowing us to then represent it in its squared form. This process is known as **Completing the Square**.

> **Note:** Many sources cite the Babylonians as the first to solve quadratic equations (around 400BC). However, the Babylonians had no concept of an *equation*, so this is not quite fair to say. What they did do however was develop an approach to solving problems algorithmically which, in our terminology, would give rise to quadratic equations. The method is essentially one of completing the square. Hindu mathematicians took the Babylonian methods further to give almost modern methods which admit negative quantities. They also used abbreviations for the unknown, usually the initial letter of a colour was used, and sometimes several different unknowns occurred in a single problem.

In order to be able to represent the LHS in Example 6 in its squared form, we must somehow get it into a form that makes it a perfect square. That is to say, we must try to mold it into the form: $o^2 + 2eo + e^2$.

Let us see step by step how the process of completing the square is performed. We will start by showing the LHS of our equation alongside the form of the perfect square trinomial we are trying to mold it into:

$$o^2 + 2eo + e^2 = 0$$

$$2x^2 + 12x = 0$$

Step 1: In our equation **x** is the unknown, which corresponds to **o** in the square trinomial. We also have an **x** coefficient of 2. In the square trinomial the $\mathbf{o^2}$ term is on its own (it has no coefficient) so we want to get $\mathbf{x^2}$ on its own also. Since **x** is currently multiplied by 2 in our equation, we can cancel it out by dividing the equation (both the LHS and the RHS) by 2:

$$x^2 + \frac{12}{2}x = \frac{0}{2}$$

As you can see, dividing $\mathbf{2x^2}$ by 2 yields $\mathbf{x^2}$, removing the coefficient of the $\mathbf{x^2}$ term as we intended. Actually, we could say that the $\mathbf{x^2}$ term now has an implied coefficient of 1. We must also divide every other term of the equation by 2 so that the equation stays balanced. We divide the term **12x** by 2 reducing the term to **6x.** The term on the RHS is divided by 2 also, but as it is currently is set to zero, dividing by anything will still result in a zero and as such, the RHS remains unchanged. This evaluates to the following equation (shown again along side the trinomial whose form we are trying to achieve):

$$o^2 + 2eo + e^2 = 0$$
$$x^2 + 6x = 0$$

Step 2: Remembering that **x** in our equation is equivalent to **o** in the PST, we can see that we have correctly reduced the $\mathbf{x^2}$ term on the LHS so that it has no coefficient (or an implied coefficient of 1). This is consistent with the $\mathbf{o^2}$ term in the PST. We can also see that in the middle term of the square trinomial, the unknown **o** has a coefficient of **2e.** In the middle term of our equation, **x** has a coefficient of 6. Therefore, we can say that in our equation **2e = 6** and thus, **e = 3**. All we have to do to make our equation a PST is add $\mathbf{e^2}$ to both sides of the equation (LHS and RHS must remain balanced). We have just discovered that if **2e = 6** then **e = 3** and thus $\mathbf{e^2 = 3*3 = 9}$. So, in this step we add 9 to both the LHS and the RHS of the equation, resulting in the following:

$$x^2 + 6x + 9 = 9$$

Now we have a perfect square trinomial where **o** is equivalent to **x** and **e** is **3**. Identity 1 tells is that the LHS of this equation can now be written in its squared form for simplicity:

$$(x+3)^2 = 9$$

Step 3: Now we can just solve as before. First we take the square root of both sides:

$$x + 3 = \pm\sqrt{9}$$

Step 4: Subtract 3 from both sides to leave **x** isolated on the LHS:

$$x = -3 \pm \sqrt{9}$$

This results in the two solutions for **x** shown below:

$$x = -3 + 3 = 0$$

and

$$x = -3 - 3 = -6$$

We have successfully found the solutions to the equation $2x^2 + 12x = 0$ by first completing the square and then solving as usual.

The technique of completing the square can be reliably used to solve any quadratic. Let us look at one more example of completing the square before moving on.

<u>**Example 7:**</u> $4x^2 + 12x - 100 = 0$ (Solving a Quadratic Equation that is not a square.)

Step 1: In this example, our equation is a trinomial to begin with, but not a perfect square. In a perfect square trinomial we need the 3rd term to be something very specific (e^2), so we really want to lose our current 3rd term on the LHS so that we can put something else there that completes the square. So let us first add 100 to both sides of the equation, which will remove the 3rd term (-100) from the LHS and add 100 to the RHS. After this step, our equation now looks as follows:

$$4x^2 + 12x = 100$$

Step 2: Our next job is to remove the coefficient of the squared term ($\mathbf{x^2}$) so that it matches its like term in the identity. If we divide every term by 4, this will eliminate the coefficient of the squared term. More correctly, it reduces it to an implied coefficient of 1, as shown below. Notice how dividing $\mathbf{4x^2}$ by 4 leaves us with $\mathbf{x^2}$, exactly as we want:

$$x^2 + \frac{12}{4}x = \frac{100}{4}$$

If we carry out the remaining divisions indicated above we can see that 12x/4 = 3x and 100/4 = 25, reducing the equation to the following:

$$x^2 + 3x = 25$$

Step 3: Now we have the LHS reduced to two terms with no leading coefficient accompanying the squared term. So we find ourselves with an equation in the same form as the previous example. We know from the identity that the coefficient of **o** in the middle term is **2e**, and we can see that as **x = o** in our equation, its coefficient **3** must be equal to **2e**. Therefore we have correctly identified that **e = 3/2 = 1.5**. We know that we can now complete the square by adding e^2 to both sides of the equation. Since **e = 1.5** and $\mathbf{e^2 = 2.25}$, we simply add **2.25** to both the LHS and the RHS to complete the square:

$$x^2 + 3x + 2.25 = 27.25$$

Step 4: Now that we have a PST on the LHS and we know that **o = x** and **e = 1.5**, we can write the LHS in (o + e) ^{2} form, as shown below:

$$(x + 1.5)^2 = 27.25$$

Step 5: We take the square roots of both sides of the equation, leaving us with x + 1.5 on the LHS and the positive and negative square roots of 27.25 on the RHS (which we will leave under the square root sign for now).

$$x + 1.5 = \pm\sqrt{27.25}$$

Step 6: Finally we can isolate **x** by subtracting 1.5 from the both sides of the equation, introducing –1.5 on the RHS.

$$x = -1.5 \pm \sqrt{27.25}$$

We now have the two values of **x** shown below. The first shows the calculation using the positive root of 27.27 (5.2201) .

$$x = -1.5 + 5.2201 = 3.7202$$

The second value of x is determined using the negative root of 27.25 (-5.2201)

$$x = -1.5 - 5.2201 = -6.7202$$

Therefore, the solutions are **x = 3.7202** and **x = –6.7202**.

Over the past few sections we have learned how to solve both simple and complex quadratic equations using a method known as completing the square. When completing the square we first mold the LHS into a perfect square trinomial. But early on, we discussed how to take a quadratic expression in any form an manipulate it into the standard form $at^2 + bt + c = 0$.

Since we learned how to solve standard and non-standard form quadratic equations, you may be wondering why we even bothered with the discussion of manipulating an equation into the standard form. We certainly do not need it in that form to solve it as we have shown. But while that is technically true, we do need to be able to solve quadratic equations in a consistent manner using a single method. This is especially true for us as programmers as we will need to write code that solves quadratic equations. We certainly do not want to have lots of conditionals testing the form of the quadratic equation so that a way to solve it can be determined.

As it happens, any quadratic equation that is in the standard form can be solved using a simple calculation called the quadratic formula. Thus if we want to write a function that can solve any quadratic equation with a minimum of code, we can just insist the equation passed into the function that needs to be solved is in the standard form. We can then use the quadratic formula to find the roots of the equation every time (assuming they exist). Whether the quadratic equation is being solved to determine the intersection between a ray and a sphere, a ray and a cylinder, or even for calculating cost analysis, provided it has been manipulated into the standard form, our code can always solve it using this simple technique. Naturally then, our next and final discussion on solving quadratic equations will be on the quadratic formula, since it is the method we will be using for solving our standard form quadratic equations.

12.6.3 The Quadratic Formula

It was determined quite a long time ago that the quadratic formula could be used to solve any quadratic equation in the standard form. Indeed, the quadratic formula is so useful that it has become required learning in high schools around the world. It really is an essential tool in the mathematicians' toolkit and can be used to solve quadratics very easily.

Provided a quadratic equation is in the form $at^2 + bt + c = 0$, we can solve for *t* using the quadratic formula, which is shown below.

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The Quadratic Formula

It is well worth remembering this formula so that you can recall it at will. It will come in extremely handy down the road and will be the method we use when solving for intersections between a ray and a quadric surface (sphere or cylinder). Remember that the $\pm$ symbol essentially says that there is a positive and negative root of $\mathbf{b^2 - 4ac}$. This indicates that as long $\mathbf{b^2 - 4ac}$ is not zero, there will be two real solutions for *t*. If we call the two roots *t1* and *t2*, we see the two solutions for *t* below.

$$t1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$t2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

The section of the formula under the square root sign $b^2 - 4ac$ has special properties which allow us to determine how many *real* roots (if any) can be found prior to performing the full calculation. For this reason, it is called the *discriminant* of the polynomial. We can see for example, that if the discriminant is zero, then both *t1* and *t2* will be identical and thus we might say that one real root exists.

We can also see that if the discriminant is a negative value, then the equation has no real roots. This lets us avoid applying the full quadratic formula logic since we know that we cannot take the square root of a negative number and expect a real result. This suggests to us that we should calculate the discriminant first and determine if it is a negative number. If so, no real roots exist.

Below we see how we might implement a new function called SolveRoots. It is passed the **a**, **b**, and **c** coefficients of a quadratic in standard form and will return the real roots of the equation, if they exist, in the output variables t1 and t2.

```
bool CCollision::SolveRoots( float a, float b, float c, float& t1, float &t2 )
{
    float d, one_over_two_a;

    // Calculate Discriminant first
    d = b*b - 4*a*c;

    // No discriminant is negative return false
    // as no real roots exist
    if (d < 0.0f) return false;

    // Calculate square root of b^2-4ac ( discriminant )
    d = sqrtf( d );

    // Calculate quadratic formula denominator ( 2a )
    one_over_two_a = 1.0f / (2.0f * a);

    // Calculate the two possible roots using quadratic formula.
    // They be duplicate is d = 0
    t1 = (-b - d) * one_over_two_a;
    t2 = (-b + d) * one_over_two_a;

    // Solution found
    return true;
}
```

These few lines of code are pretty much all there is to solving a quadratic equation in standard form using the quadratic formula. The actual function that solves the quadratic equations in our CCollision class is in a method called SolveCollision which is almost identical to this. The only difference is that the SolveCollision function will only return the smallest positive root. This is because the quadratic equation passed in (as we will see later) will represent the intersection of a ray with a sphere or cylinder. As negative *t* values will represent the distance to the surface behind the ray origin we are not interested in returning them. Additionally, since we wish to find the first point of intersection between the ray and the quadric surface (there may be two intersections since a ray can pass through a sphere's surface twice), we are only interested in the nearest intersection. Therefore, we are interested in the **smallest positive** *t* value.

Below we see the actual code to our CCollision::SolveCollision function.

```
bool CCollision::SolveCollision( float a, float b, float c, float& t )
{
    float d, one_over_two_a, t0, t1, temp;

    // Basic equation solving
    d = b*b - 4*a*c;

    // No root if d < 0
    if (d < 0.0f) return false;

    // Setup for calculation
    d = sqrtf( d );
    one_over_two_a = 1.0f / (2.0f * a);

    // Calculate the two possible roots
    t0 = (-b - d) * one_over_two_a;
    t1 = (-b + d) * one_over_two_a;

    // Order the results
    if (t1 < t0) { temp = t0; t0 = t1; t1 = temp; }

    // Fail if both results are negative
    if (t1 < 0.0f) return false;

    // Return the first positive root
    if (t0 < 0.0f) t = t1; else t = t0;

    // Solution found
    return true;

}
```

Notice that after we have found the two solutions, we store them in local variables t0 and t1. We then order them (possibly involving a swap) so that t0 always holds the smallest *t* value and t1 holds the larger one. We then test to see if t1 is smaller than zero. If it is then both *t* values are negative (because t1 is the largest value) and the intersections between the ray and the quadric surface happen behind the ray origin and are of no interest to us. Finally, we test if t0 (which now contains the smallest *t* value) is negative. If it is, then we return t1 which will be the smallest positive root. If t0 is not negative, then it must contain the smallest positive root, since we ordered them this way a few lines before. If this is the case, we return the value of t0.

Of course, at this point you are not supposed to understand how this function gets called and why we need to call it. This will all be discussed in a moment. However, hopefully you can see that this function is essentially doing nothing more than being passed the coefficients of a quadratic equation in standard form and then solving for the unknown *t*. If you follow the lines of code, you should see how we are performing the steps exactly as dictated by the quadratic formula. Regardless of whether we wish to intersect a ray with a sphere, a cylinder, or any other quadric surface, provided we can substitute the ray equation into the equation of the surface (discussed in a moment) and then manipulate that equation into the standard form, we can use this function to find the *t* values (i.e., the points of intersection parametrically along the ray).

Thus far you have taken for granted that the quadratic formula works but we have not discussed why it works and how it was arrived at in the first place. Therefore, let us see what happens if we try to solve a generic quadratic equation in standard form. You will find that we end up with the quadratic formula. This means you will understand not only what the quadratic formula is, but also why it works.

Step 1: We first determine that we wish to solve a quadratic equation in the standard form. Our objective is to isolate *t* on the LHS thus finding solutions for it. We will do this by manipulating the LHS into a perfect square in the form $\mathbf{o^2 + 2eo + e^2}$ which can then be written as $\mathbf{(o+e)^2}$ and easily solved. So let us start with the standard form quadratic equation template.

$$at^2 + bt + c = 0$$

Step 2: The first simple thing we can do in our quest to remove everything from the LHS (other than **t**) is to subtract **c** from both sides. This will eliminate the third term from the LHS and introduce **–c** on the RHS as shown below.

$$at^2 + bt = -c$$

Step 3: To make our perfect square trinomial, we know that the squared term $\mathbf{t^2}$ must have an implied coefficient of 1. Therefore, just as we have done many times before, we will divide the rest of the terms of the equation by **a**. This will set the coefficient of the squared term to 1 and leave just $\mathbf{t^2}$.

$$\frac{a}{a}t^2 + \frac{b}{a}t = \frac{-c}{a}$$

We know that a/a is just 1 and cancels itself out, so our equation now looks like what we see next. It is shown along side the PST whose form we are trying to replicate.

$$t^2 + \frac{b}{a}t = \frac{-c}{a}$$

$$o^2 + 2eo + e^2 = 0$$

Step 4: Looking at our equation we can see that **t** is equivalent to **o** in the PST. We can also see that the coefficient of **t** in the middle term of our equation is **b/a** which is equal to **2e** (the coefficient of **o** in the PST's middle term). Therefore, we have determined that **2e = b/a** in our equation and thus, **e** must equal **b/a** divided by 2.

When we multiply a fraction by a single value, we essentially treat that single value as a fraction with a denominator of 1. Therefore, multiplying **b/a** by 2 would look like this.

$$\frac{b}{a} x \frac{2}{1} = \frac{b*2}{a*1} = \frac{2b}{a}$$

As you can see, we multiply the two fractions by multiplying the numerators and denominators separately. When we wish to divide a fraction by a real number we essentially do the same thing. That is, we add a denominator of 1 to the single value to get it into fraction form. However, because we wish to divide one fraction by another instead of multiply, we still perform a multiply but will swap the numerator and the denominator of the second fraction (i.e., multiply by the *reciprocal*).

Therefore, since we know that **2e = b/a** in our equation, we can find the value of **e** by multiplying the fraction **b/a** with the fraction ½ as shown below.

$$e = \frac{b}{a} x \frac{1}{2} = \frac{b}{2a}$$

Step 5: Now that we know the value of **e**, we can add $\mathbf{e^2}$ to both sides of the equation to form a perfect square trinomial on the LHS which is ready to be written in its squared form. As we know that **e=b/2a**, $\mathbf{e^2}$ must be equal to multiplying the fractions (**b/2a**)*(**b/2a**) as shown below.

$$e^2 = \frac{b}{2a} x \frac{b}{2a} = \frac{b^2}{4a^2}$$

So we must add this result to both the LHS and the RHS of our equation to complete the square. Our equation now looks like this:

$$t^2 + \frac{b}{a}t + \frac{b^2}{4a^2} = -\frac{c}{a} + \frac{b^2}{4a^2}$$

Step 6: As we now know that we have a perfect square trinomial on the LHS and that **o = t** and **e = b/2a**, we can write the LHS of the equation in its squared form.

$$\left(t + \frac{b}{2a}\right)^2 = \frac{-c}{a} + \frac{b^2}{4a^2}$$

Step 7: At this point the RHS could do with a little cleaning up. It is currently represented as the addition of two fractions, which is not very nice. But we could merge these two fractions together if they both had the same denominator. For example, if the left fraction on the RHS had a denominator of $4a^2$ also, we could represent the RHS simply as

$$\frac{-c+b^2}{4a^2}$$

Of course, we cannot do that because the left fraction does not have a denominator of 4a^2. However, we could multiply the left fraction (both the numerator and the denominator) so that it is scaled up to a fraction of the same proportion but with a denominator of 4a^2. Many of you may remember this from high school as finding the lowest common denominator. The goal is to have all fractions use the same denominator to simplify our calculations.

What can we multiply the first fraction (**–c/a**) by such that its numerator and denominator stay proportional to one another and yet we wind up with a fraction whose denominator has 4a^2? That is simple enough -- we multiply **–c/a** by the fraction **4a/4a** since multiplying the first fraction's denominator (**a)** by **4a** will scale it to **4a^2** as shown below.

$$\frac{-c}{a} * \frac{4a}{4a^2} = \frac{-4ac}{4a^2}$$

As you can see, multiplying the numerator **–c** by **4a** results in a numerator of **–4ac**. Multiplying the denominator **a** by **4a** results in the denominator we were after: **4a^2**. This now matches the denominator of the other fraction on the RHS.

$$\left(t+\frac{b}{2a}\right)^2 = \frac{-4ac}{4a^2} + \frac{b^2}{4a^2}$$

Since both the fractions on the RHS now have the same denominator, we can add the two fractions together to get a single result on the RHS:

$$\left(t+\frac{b}{2a}\right)^2 = \frac{-4ac+b^2}{4a^2}$$

What we will do now is rearrange the numerator of the RHS (we generally prefer to have the terms sorted by power going from left to right) to give:

$$\left(t+\frac{b}{2a}\right)^2 = \frac{b^2-4ac}{4a^2}$$

Those of you paying close attention will have noticed that the numerator on the RHS is actually the discriminant we discussed earlier.

Step 8: We currently have our equation in squared form. As we have done many times before, we now wish to take the square root of both sides of the equation. This will remove the squared brackets on the LHS leaving us with just **t + b/2a**, making **t** easier to isolate.

$$\sqrt{\left(t+\frac{b}{2a}\right)^2}=\sqrt{\frac{b^2-4ac}{4a^2}}$$

The square root of the LHS is simple. It just removes the squared brackets like so.

$$t+\frac{b}{2a}=\sqrt{\frac{b^2-4ac}{4a^2}}$$

Taking the square root of the RHS is easy also. We simply take the square root of the denominator and the numerator separately.

$$t+\frac{b}{2a}=\frac{\sqrt{b^2-4ac}}{\sqrt{4a^2}}$$

Looking at the numerator of the RHS first; the square root of **b^2 -4ac** is something we just do not know since it cannot be reduced any further. So we will just leave the numerator as is with the square root sign still in place. However, the denominator is no problem at all. We know that:

$$2a*2a=4a^2$$

Thus, the square root of $4a^2$ is 2a. After we have taken the square roots of both sides, our equation now looks like this.

$$t+\frac{b}{2a}=\frac{\pm\sqrt{b^2-4ac}}{2a}$$

Step 10: We nearly have **t** isolated on the LHS. All that is left to do now is subtract **b/2a** from both sides of the equation:

$$t = \frac{\pm\sqrt{b^2 - 4ac}}{2a} - \frac{b}{2a}$$

Because both fractions on the RHS share a common denominator, we can add them and rearrange to look a little nicer. The result is the quadratic formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This wraps up our review of quadratic equations. As explained at the beginning of this section, the discussion has been completely abstracted from the topic of collision detection and as such, you may be wondering what any of this has to do with finding intersections with triangles. In the next section, you will finally get the answers to your questions and see that everything you have learned here will be put to good use.

12.7 The SphereIntersectTriangle Function: Part II

Earlier we discussed the progress we had made with respect to writing a function that would correctly detect the intersection point between a swept sphere and a polygon. As discussed, we currently have only implemented the first and most common case – testing against the plane. If the sphere did not intersect the plane of the triangle, the function can immediately return false. If an intersection with the plane has occurred then we tested the intersection point (on the plane) to see whether it was inside the polygon/triangle edges. If it was, then our task was complete. We have our intersection point on the shifted plane (the potential new sphere center position) and we have our intersection normal (the normal of the polygon which we can use for sliding). However, if we have intersected the plane but the intersection point is not in the triangle interior, then we must test for other cases. For starters, we must test if the swept sphere has collided with one of the three triangle edges (Figure 12.31).

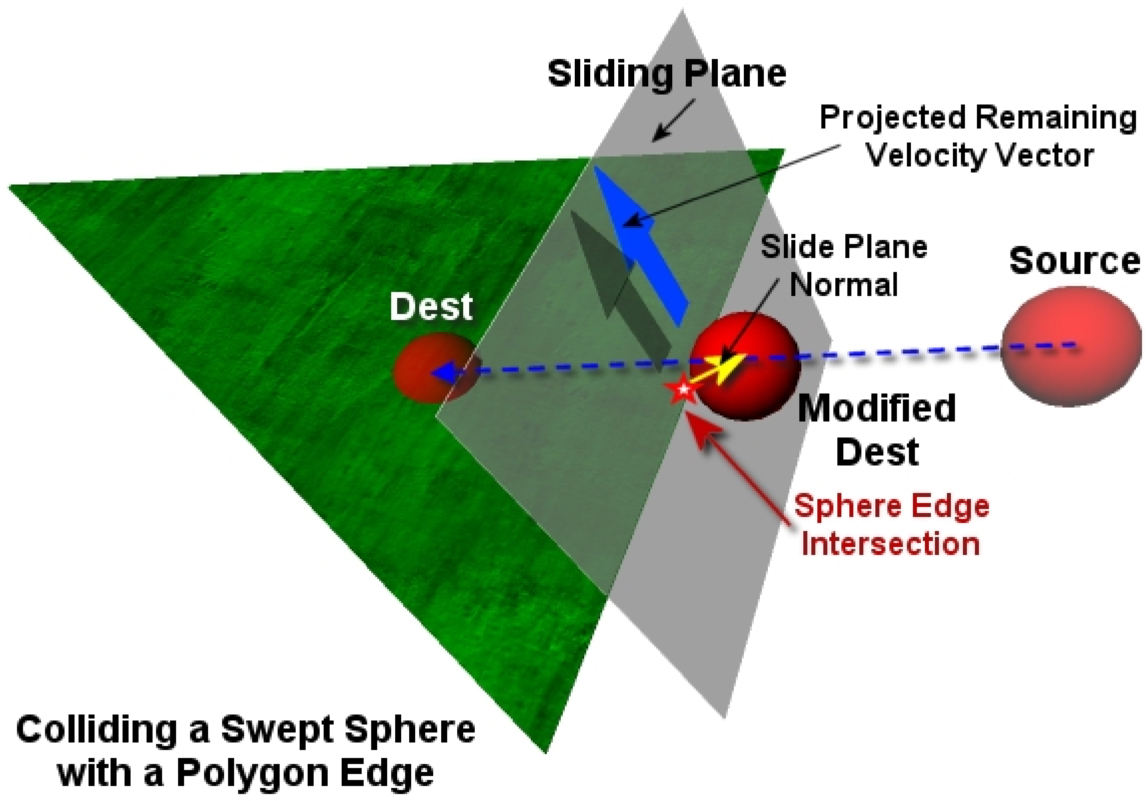

Sliding Plane
Projected Remaining Velocity Vector
Slide Plane Normal
Source
Dest
Modified Dest
Sphere Edge Intersection
Colliding a Swept Sphere with a Polygon Edge

Figure 12.31

Figure 12.31 shows an intersection test between our swept sphere and a single edge. We will need to perform this test for each edge to make sure we catch all collision cases and return the closest intersection.

In Figure 12.31, **Source** and **Dest** describe the initial center position of the sphere and the desired destination of the sphere after the movement update. The blue dashed line connecting them describes the velocity vector we wish to move the sphere along to make that so. **Modified Dest** is the new center position we need to calculate which will position the sphere such that its surface is touching (but not intersecting) the triangle edge.

Once we have found this new position, we also have to return an intersection normal for later use as our slide plane normal. As Figure 12.31 shows, this will be calculated by creating a vector from the intersection point on the edge to the sphere center position and then normalizing the result. This is different from the normal returned when the sphere collides with the interior of a polygon. In the other case, the triangle normal is returned. But when we collide with an edge, we wish to slide around that edge smoothly. You should be able to see that the angle of approach made by the sphere with the intersected edge influences the orientation of the slide plane generated. If the sphere collides with the edge right in the center of its surface a very steep plane will be produced. With such a steep slide plane, any remaining velocity vector will be scaled down quite a bit when projected onto the slide plane (in the response phase) and only a little slide will happen. This is like hitting the edge head on. If however, the intersection occurs via a more shallow angle of approach (almost as if we are just clipping the edge accidentally as we walk by it), the plane will not be as steep and the edge will be easily slid around without losing to much velocity during the velocity vector projection process in the response step.

While we do not yet know how to write a function that determines the intersection between a swept sphere and a triangle edge, for now we will stipulate that this test will be performed in a method called CCollision::SphereIntersectLineSegement. This will be the method that will be called for each edge of the triangle to test for intersection between the sphere and the line segment formed by the vertices of each edge. While we will discuss how to implement this function in a moment, we now have enough information to understand the final version of our CCollision::SphereIntersectTriangle function. The final code is shown below. The only difference from the version we saw earlier will be the three function calls that have been added to the bottom of the function (highlighted in bold).

```
bool CCollision::SphereIntersectTriangle(  const D3DXVECTOR3& Center,
                                           float Radius,
                                           const D3DXVECTOR3& Velocity,
                                           const D3DXVECTOR3& v1,
                                           const D3DXVECTOR3& v2,
                                           const D3DXVECTOR3& v3,
                                           const D3DXVECTOR3& TriNormal,
                                           float& tMax,
                                           D3DXVECTOR3& CollisionNormal )
{
    float       t = tMax;
    D3DXVECTOR3 CollisionCenter;
    bool        bCollided = false;

    // Find the time of collision with the triangle's plane.
    if ( !SphereIntersectPlane( Center, Radius, Velocity, TriNormal, v1, t ))
        return false;

    // Calculate the sphere's center at the point of collision with the plane
    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);

    // If this point is within the bounds of the triangle,
    // we have found the collision
    if ( PointInTriangle( CollisionCenter, v1, v2, v3, TriNormal ) )
    {
        // Collision normal is just the triangle normal
        CollisionNormal = TriNormal;
        tMax            = t;

        // Intersecting!
        return true;

    } // End if point within triangle interior

    // Otherwise we need to test each edge
    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v1, v2, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                           v2, v3, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity, v3, v1,
                                              tMax, CollisionNormal );

    return bCollided;
}
```

As you can see, if the sphere is intersecting the plane of the triangle and the intersection point is inside the triangle we return true. However, if the intersection point is not within the interior of the triangle we perform an intersection test between the swept sphere and each edge. We pass into the SphereIntersectLineSegment function the eSpace position of the center of the ellipsoid, the radius vector of the ellipsoid, and the eSpace velocity vector describing the direction and distance we wish to move the sphere. We also pass in the two vertices that form the edge we are testing. Obviously a triangle has three edges, so the function is called three times (once to test each edge). Each time a different vertex pair is used to describe the edge.

We also pass (by reference) the tMax variable that was itself passed in as a parameter to the function. Remember from our earlier discussions that this will always hold the smallest intersection distance (*t* value) that we have found so far. We are only interested in finding intersections that are closer than this value since if the intersection is further away, it means we have previously found another triangle that is closer to our sphere. Remember, the SphereIntersectTriangle function is called for every triangle in the collision database and the same tMax variable is overwritten as soon as we find an intersection that is closer than one we previously stored.

As the final parameter, we pass (by reference) CollisionNormal, which itself is a vector that was passed by reference into the function. If an intersection is found with the edge that is closer than any found so far, the function will store the collision normal in this variable. When the SphereIntersectLineSegment function returns, if an intersection has been determined to be closer than any found with previously tested triangles, the calling function will have access to both the intersection *t* value and the collision normal.

So the question that begs to be asked is, how do we detect an intersection between a moving sphere (swept sphere) and a polygon edge? The answer can be found in the next section.

12.7.1 Swept Sphere / Triangle Edge Intersection

Performing an intersection test between a swept sphere and a triangle edge sounds like it might be quite complicated. When we tested the swept sphere against a plane, we were able to reduce the entire test to a ray / plane test simply by shifting the plane along its normal by the radius of the sphere. The point at which the ray intersected the shifted plane described the exact center position of the sphere at the time the surface of the sphere will make contact with the actual plane. Certainly it would be nice if a similar technique could be employed to reduce the swept sphere / edge test to that of an intersection test between a ray and some other primitive type. As it turns out, there is indeed a way to do this.

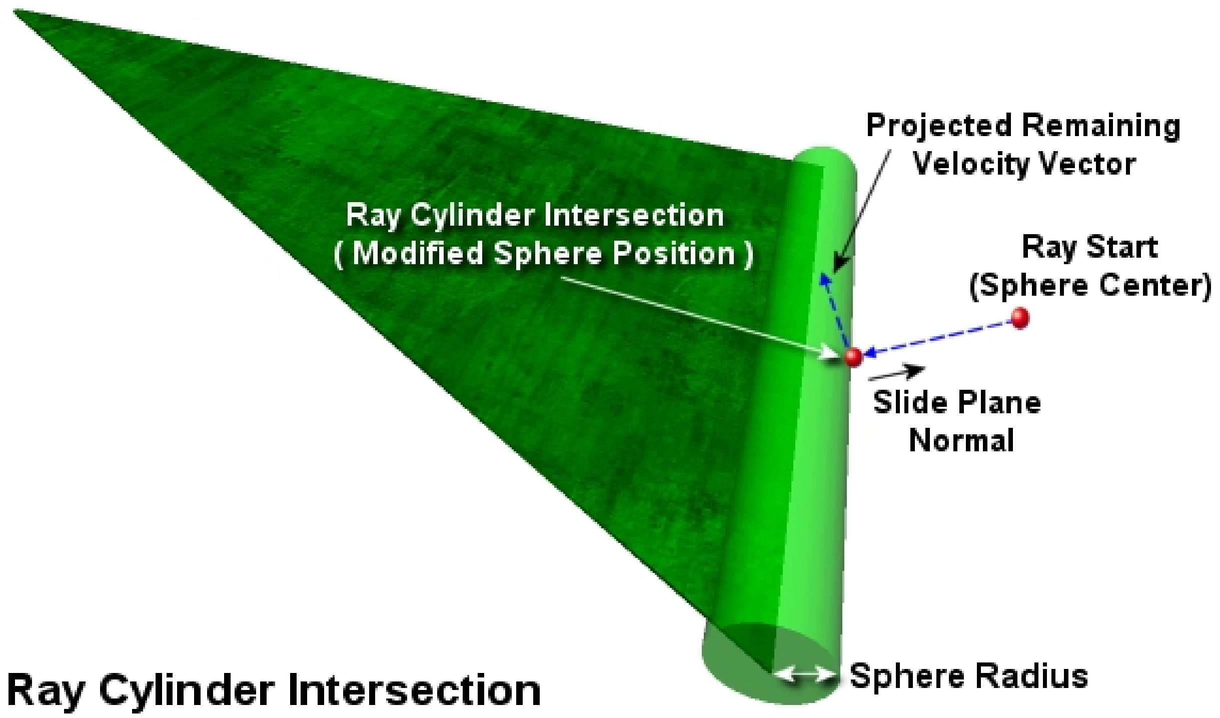

Projected Remaining Velocity Vector
Ray Cylinder Intersection (Modified Sphere Position)
Ray Start (Sphere Center)
Slide Plane Normal
Sphere Radius
Ray Cylinder Intersection

Figure 12.32

In the sphere / plane intersection test we were able to reduce the swept sphere to a ray by shifting the plane along the normal by the sphere radius. Imagine now, that we were to take the edge of the triangle that we currently want to test and expanded it outwards in all directions by the radius of the sphere. That is, we would give the edge a thickness equal to twice the sphere's radius. If we could do this, we would have essentially grown the edge into a cylinder, where any point on the surface of the cylinder will be located at a distance of sphere radius from its central spine.

As Figure 12.32 illustrates, if the edge becomes a cylinder, the swept sphere can once again be treated as a ray, where the original eSpace center position of the sphere becomes the ray origin (Ray Start) and the eSpace velocity vector becomes the ray's delta vector. If we find the intersection point along the ray with the surface of the cylinder, it will be at a distance of sphere radius from the actual edge and will thus describe the exact location where the center of the sphere should be positioned at the time the surface of the sphere is touching the actual edge (the time of intersection). Remember, the true edge is the central spine of the cylinder which is at a distance of radius units from any point on the surface of the cylinder. All we have to do is implement a function that will find the *t* value along a ray which intersects the surface of a cylinder. This will provide us with the new position for the center of the sphere at the time of intersection between the sphere's surface and the triangle edge.

Once we have this intersection point on the surface of the cylinder, we can also calculate the slide plane normal by generating a vector from the intersection point on the actual edge to the new sphere center we just calculated (and normalizing the result).

The equation for a cylinder is.

$$\|(P - V1) \times E\|^2 = Radius^2$$

Let us analyze this formula. **P** is a point on the surface of the cylinder and **V1** is the first vertex in the edge that forms the cylinder's central spine (axis). This is the point at the base of the cylinder. **E** is the unit length direction vector of the cylinder's central spine. Therefore, if the edge of the triangle is given by vertices **V1** and **V2**, **E** is created by normalizing the vector (**V2 - V1**) as shown in Figure 12.33.

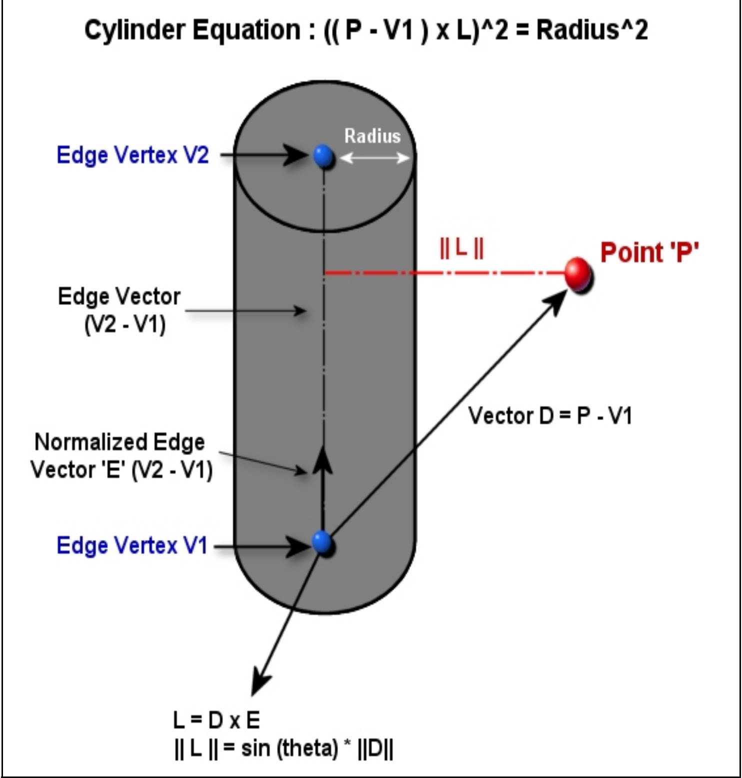

Cylinder Equation : ((P - V1) x L)^2 = Radius^2
Edge Vertex V2
Radius
|| L ||
Point 'P'
Edge Vector (V2 - V1)
Vector D = P - V1
Normalized Edge Vector 'E' (V2 - V1)
Edge Vertex V1
L = D x E
|| L || = sin (theta) * ||D||

Figure 12.33

The cylinder equation says that if the squared length of a vector, (generated by crossing a vector from **V1** to the point **P** with the unit length direction vector of the cylinder's central axis **E**) is equal to the squared radius of the cylinder, the point **P** is on the surface of the cylinder. In Figure 12.33, the point **P** is clearly not on the surface of the cylinder, but what is the relationship between the squared length of the vector we have created from that cross product (**(P-V1)xE**) on the LHS of the equation and the radius of the cylinder?

In this example, **P** in the cylinder equation is **Point P** in the diagram. We can see that it is not on the surface of the cylinder, but this was done intentionally to clarify the relationships we are about to discuss. We can also see that if we create a vector from **V1** at the bottom of the cylinder (first vertex in the edge) to the point **P**, we have vector **D** in the diagram. If we look at unit vector **E**, we can imagine how this forms the adjacent side of a triangle where the vector **D** forms the hypotenuse. What we need to know is the distance from **Point P** to the edge (red dashed horizontal line in Figure 12.33). To calculate this distance we start by performing the cross product between vectors **D** and **E**. We know that when we perform the cross product between two vectors we get a vector returned that is perpendicular to the two input vectors. This is vector **L** in the diagram (assumed to be pointing out of the page). The direction of this vector appears to have nothing to do with what we are trying to accomplish, and indeed it does not. However, our interest in is the length of this vector, not its direction. Why? Because the length of the vector returned from a cross product given as:

$$\|D \times E\| = \|D\|\|E\| \sin\alpha$$

The length of the vector returned from crossing **D** and **E** is equal to the sine of the angle between them multiplied by the length of vector **E** and the length of vector **D**. In our cylinder equation, **E** is a unit length vector and thus this equation simplifies to:

$$\|D \times E\| = \|D\| \sin \alpha$$

So when we cross **D** and **E**, we are returned a vector whose length is equal to the sign of the angle between the two input vectors scaled by the length of vector **D** (the hypotenuse).

Now, if you refer back to Figure 12.33 and envisage **D** as the hypotenuse and **E** as the adjacent of a right angled triangle, we can see that what we actually wish to determine is the length of the opposite side of that triangle. The opposite side is the line from the point **P** to the adjacent side formed by the central spine of the cylinder. How do we get the length of the opposite side of a right angled triangle?

The helpful acronym **SOH CAH TOA** tells us that we can find the sine of the angle by dividing the length of the opposite side by the length of the hypotenuse.

$$\sin \alpha = \frac{Opposite}{Hypotenuse}$$

Therefore, in order to find the length of the opposite side, we simply multiply each side of this equation by the length of the hypotenuse leaving the opposite side isolated on the RHS.

$$\sin \alpha * Hypotenuse = Opposite$$

or more nicely arranged:

$$Opposite = Hypotenuse * \sin \alpha$$

So we know that we can find the length of the opposite side of a triangle by multiplying the length of the hypotenuse by the sine of the angle. Does that help us? Yes it does, because that is exactly what the cross product just gave us. Remembering once again that vector **D = (P – V1)** is the hypotenuse of our triangle and vector **E** is unit length, the cross product of **DxE** returns a vector **L** with the following length.

$$\|L\| = \|D \times E\|$$

The above states that our vector **L** is the result of crossing vectors **D** and **E**. The length of vector **L** (||**L**||) is then obviously the length of the vector returned from crossing **D** and **E**. Since vector **D** is generated by subtracting vector **V1** from point **P**, using substitution for **D**, this can also be written as:

$$= \|(P - V1) \times E\|$$

The result of the cross product is a vector whose length is equal to the sine of the angle between them multiplied by the lengths of each input vector. Since input vector **E** is unit length, the result is simply the sine of the angle scaled by the length of the non-unit vector **D** (i.e., **P - V1**).

$$= \|(P - V1)\| \sin \alpha$$

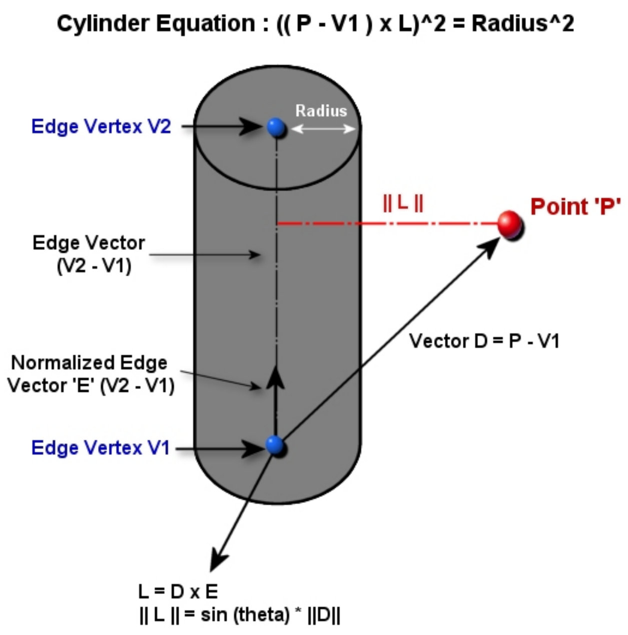
Since vector **P-V1** forms the hypotenuse of our triangle, the length of this vector is equal to the length of the hypotenuse, so we can write this as:

$$= \|Hypotenuse\| \sin \alpha$$

As we just discovered, scaling the length of the hypotenuse of a right angled triangle by the sine of its angle gives us the length of the opposite side,

$$= \|Opposite\|$$

And there we have it. As **D** is the hypotenuse and **E** is unit length, the length of the resulting vector **L** is equal to the length of vector **D** multiplied by the sine of the angle between **D** and **E**. This is the length of the hypotenuse multiplied by the sine of angle, which trigonometry tells us is the length of the opposite side. This is the length of the red dashed horizontal line in the diagram and is the distance from point **P** to the cylinder's central spine (i.e., the triangle edge).



Cylinder Equation : ((P - V1) x L)^2 = Radius^2
Edge Vertex V2
Radius
|| L ||
Point 'P'
Edge Vector (V2 - V1)
Vector D = P - V1
Normalized Edge Vector 'E' (V2 - V1)
Edge Vertex V1
L = D x E
|| L || = sin (theta) * ||D||

Figure 12.34

In Figure 12.34 we can now see exactly why we compare the squared length of vector **L** against the squared radius. The length of vector **L** describes the distance from the point **P** to the edge (the central axis). Since the cylinder's surface is a distance of Radius units from its central axis, if the squared distance from point **P** to the edge (||**L**||) is equal to the Radius squared, the point must be on the surface of the cylinder. In Figure 12.34 we can see that the length of vector **L** would be much greater than the radius of the cylinder and therefore the point **P** is not considered to be on the surface of the cylinder.

Of course, if we were only interested in determining whether or not a point is on the surface of the cylinder, we would be all set. However, we wish to find a ***t*** value along a ray which describes a point on the surface of the cylinder (i.e., the *t* value at which the ray intersects the cylinder). Just as we did when intersecting a ray with a plane, we must substitute the equation of our ray into the cylinder equation in place of point **P**.

If the cylinder equation is:

$$\|(P - v1) \times E\|^2 = Radius^2$$

and the equation of our ray is defined by an origin **O**, a delta vector **V**, and a parametric value ***t*** describing a point along that ray:

$$O + tV$$

Substituting our ray into the cylinder equation in place of **P** gives us:

$$\left\|(O + tV - V1) \times E\right\|^2 = Radius^2$$

Because we are dealing with vectors from an algebraic point of view, we can treat the double bars || as brackets, since we know that they mean the length of a vector. Because this length is squared in the equation, we also know that we can get the squared length of a vector by dotting it with itself (a vector multiply). Therefore, let us re-write our equation in bracket form. Many of you may consider this approach a little relaxed, but we find it helpful.

$$((O + tV - V1) \times E)^2 = Radius^2$$

This is not a very attractive equation at the moment since we want to isolate ***t*** as much as possible. Well, we know that subtracting vector **V1** from the vector (**O + tV**) is equivalent to subtracting vector **V1** from vector **O** and then adding vector **tV** afterwards, so we can swap them around.

$$((O - V1 + tV) \times E)^2 = Radius^2$$

We also know, that because the vector (**O-V1**) and the vector **tV** are inside brackets that are crossed with vector **E**, this is equivalent to removing those inner brackets and crossing the vector (**O-V1**) and **tV** with **E** separately. As you can hopefully see, we are slowly breaking this equation down into smaller more manageable and components.

$$((O - V1) \times E + (tV \times E))^2 = Radius^2$$

We also know that *t* is a scalar not a vector. Scaling vector **V** by *t* and then crossing with **E** is equivalent to crossing **V** and **E** and then scaling the result. Therefore, we can move *t* outside the brackets, isolating it even more.

$$((O - v1) \times E + (V \times E)t)^2 = Radius^2$$

This is all looking good so far. However, everything on the LHS is still in squared form. Thus, everything on the LHS can be removed from the squared brackets by multiplying the LHS with itself as we discussed earlier (refer back to multiplying an expression in squared brackets). Since the expression in both brackets is a vector, and the vector form of a multiply is a dot product, we should use the dot symbol as shown below when we multiply the expression by itself:

$$((O - v1) \times E + (V \times E)t) \bullet ((O - v1) \times E + (V \times E)t) = Radius^2$$

This is starting to look a little more complex, but that is no problem. We just have a set of bracketed terms that we have to perform **FOIL** on. For the time being, we will just concentrate on the LHS of the equation.

The **First** term in each bracket is $(O - v1) \times E$. Therefore, we are multiplying vector $(O - v1) \times E$ by itself. This gives us a new first term of:

$$((O - v1) \times E) \bullet ((O - v1) \times E) \ = \ (O - v1) \times E)^2$$

The **Outer** term of the first bracket is $(O - v1) \times E$ whilst the outer term of the second bracket is $(V \times E)t$. Both of these are vectors and therefore, our second new term is calculated as:

$$((O - v1) \times E) \bullet ((V \times E)t)$$

The **Inner** terms of the brackets are $(V \times E)t$ and $(O - v1) \times E$ for the first and second brackets respectively, giving us an identical term to the last one if we swap them around (which we can do without altering its evaluation):

$$((O - v1) \times E) \bullet ((V \times E)t)$$

Finally, we multiply the last terms of each bracket, which are the same for both $(V \times E)t$:

$$(V \times E)t \bullet (V \times E)t \quad = \quad (V \times E)^2 t^2$$

Let us now add the four results together and see our new expanded equation. We have put each FOIL result on its own line for clarity at this stage, but this is all just the LHS of our equation.

$$\begin{aligned}
&((O - v1) \times E)^2 + \\
&((O - v1) \times E) \bullet (V \times E)t + \\
&((O - v1) \times E) \bullet (V \times E)t + \\
&(V \times E)^2 t^2
\end{aligned}$$

One thing is immediately obvious. Since *t* is the unknown we wish to find and it is raised to a power of two in one of the expressions, we have ourselves a quadratic that needs solving. We know how to solve quadratic equations using the quadratic formula, but we need our equation in the standard form.

$$at^2 + bt + c = 0$$

Looking at our equation and searching for the squared term, we can immediately identify the value of our **a** coefficient. It is the expression that shares the term with $\mathbf{t^2}$, $(V \times E)^2$. Therefore, let us move that entire expression to the front of the equation so that it resembles its position in the standard form.

$$\begin{aligned}&(V \times E)^2 t^2 + \\ &((O - v1) \times E)^2 + \\ &((O - v1) \times E) \bullet (V \times E)t + \\ &((O - v1) \times E) \bullet (V \times E)t\end{aligned}$$

We have the squared term exactly where it should be now and we can quickly see that

$$a = (V \times E)^2$$

In other words, **a** should be set to the squared length of the vector produced by crossing the velocity vector (the ray delta vector) with the unit length direction vector of the edge (the cylinder's central spine).

We have to find the **a**, **b**, and **c** coefficients in order to solve our quadratic, and we now have one of them. So let us now work on finding the **b** coefficient.

We know from studying the standard form that **b** must be calculated from expressions that accompany *t* (not t^2). There are actually two terms in our above equation that contain *t* in its non-squared form – currently, the third and fourth terms (which are actually identical terms). Therefore, we know that we must be able to construct **b** by isolating *t* from these terms. The two terms of our equation that contain *t* in our current equation are shown below.

$$((O - v1) \times E) \bullet (V \times E)t + ((O - v1) \times E) \bullet (V \times E)t$$

Since these two expressions are the same, we are essentially adding $((O - v1) \times E) \bullet (V \times E)t$ twice to the LHS of our equation. In each term we are taking the dot product of two vectors **(O-v1)xE** and **VxE** and scaling the result by *t*. Since the result of the dot product in each of the above terms is scaled by *t*, we can scale the sum of the combined dot products by *t* in a single step instead of separately in each term. What that means (in generic terms) is, instead of doing something like **A.Bt+A.Bt,** we can instead write **2(A.B)t**. Therefore, the above two terms can be combined into a single term like so:

$$2(((O - v1) \times E) \bullet (V \times E))t$$

Let us swap the positions of the two bracketed terms since it looks a bit nicer (our humble opinion). Thus, we have simplified the two terms that involve *t* to the following:

$$2((V \times E) \bullet ((O - v1) \times E))t$$

We have now isolated *t* on the right hand side of this expression and can see that the **b** coefficient that accompanies *t* in the middle term of the standard form must be:

$$b = 2((V \times E) \bullet ((O - v1) \times E))$$

Our equation now looks like this:

$$\begin{aligned}&(V \times E)^2 t^2 + \\ &((O - v1) \times E)^2 + \\ &2((V \times E) \bullet ((O - v1) \times E))t\end{aligned}$$

We are getting there! However, we know that in the standard form the t^2 term goes on the far left, the t term goes in the middle and the constant term **c** is written to the right. Therefore let us be consistent and write our equation that way too. Currently we have the t term (the middle term in the standard form) at the end of our equation rather than the middle, so let us swap the last two terms so that we have the term involving the squared unknown on the left and the non-squared unknown in the middle:

$$(V \times E)^2 t^2 + 2((V \times E) \bullet ((O - v1) \times E))t + ((O - v1) \times E)^2 = Radius^2$$

What is left must be the constant term **c** of the standard form, but we can see now that in its current form **c** must be:

$$c = ((O - v1) \times E)^2$$

Do not forget that in the standard form, we must have a zero on the RHS of the equals sign. That is no problem either. We can just subtract the squared radius from both sides of the equation leaving zero on the RHS and introducing –**Radius^2** on the LHS.

We now have our ray/cylinder intersection equation in the standard quadratic form:

$$(V \times E)^2 t^2 + 2(V \times E)((O - v1) \times E)t + ((O - v1) \times E)^2 - Radius^2 = 0$$

Finally, we have the **a**, **b**, and **c** coefficients of our quadratic in standard form and can simply plug them into our SolveCollision function to generate the t value of intersection (if one exists) between the ray and the cylinder. Below we show our final calculations for the coefficients in the more correct vector form. That is, that the terms inside the brackets are vectors, not single values, and as such a multiply must be replaced with the vector version of a multiply (a dot product operation). As you can see, any vectors that were in a squared form in the previous version of the equation have had there square sign removed by multiplying out the square. Therefore, **(VxE)^2** becomes **(VxE)$\bullet$(VxE)** for example. This yields the final results of our coefficients exactly as we will calculate them in code.

$$\begin{aligned}a &= (V \times E) \bullet (V \times E) \\ b &= 2((V \times E) \bullet ((O - v1) \times E)) \\ c &= ((O - v1) \times E) \bullet ((O - v1) \times E) - (Radius * Radius)\end{aligned}$$

You now know how to determine when the swept sphere collides with the edge of a triangle. If it does, the *t* value returned will describe (indirectly) the new position the center of the sphere should be moved to so that its surface is touching (but not intersecting) the edge.

You could say that for the test between the swept sphere and our triangle, the triangle was inflated by the radius of the sphere. The plane was shifted during the initial tests with the interior of the polygon, so we can liken this to inflating the depth of the polygon away from the plane so that it would have a height of Radius when viewed from any point on that plane. We have also learned that to test each edge, we should also inflate that edge by the same amount, forming a cylinder. It would seem that with these two techniques, we have fully thickened our polygon in all directions by a distance equal to the sphere's radius. This allows us safely reduce the swept sphere to a ray in our intersection tests. However, our inflating technique is still missing one thing which will cause gaps to be created. Treating the swept sphere as a ray without fixing these flaws would fail (causing the sphere to get stuck on the geometry). Let us now examine where these gaps exist that would cause collision testing to fail.

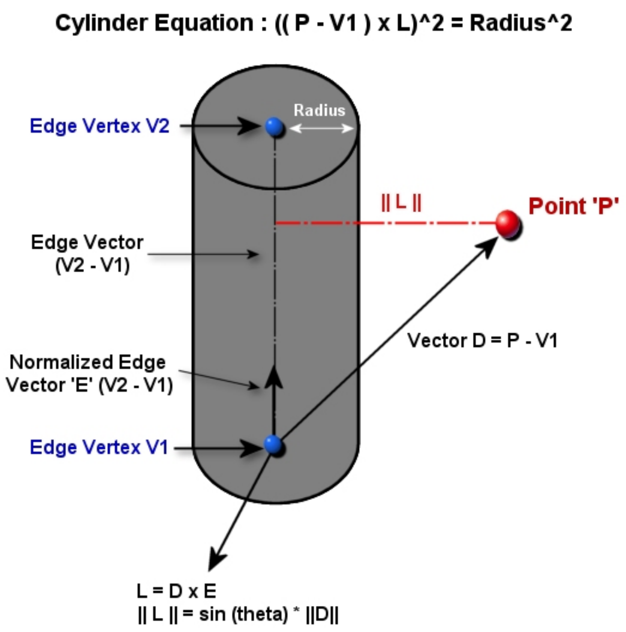

Cylinder Equation : ((P - V1) x L)^2 = Radius^2
Edge Vertex V2
Radius
|| L ||
Point 'P'
Edge Vector (V2 - V1)
Vector D = P - V1
Normalized Edge Vector 'E' (V2 - V1)
Edge Vertex V1
L = D x E
|| L || = sin (theta) * ||D||

Figure 12.35

When we look at how we constructed a cylinder from our triangle edge, we must pay close attention to the type of cylinder we have created. The cylinder has no caps; it is just an infinite tube with no top or bottom. However, when testing our ray against this cylinder, we will project the intersection point on the surface of the cylinder onto the edge itself (the central spine of cylinder) to get that actual point on the infinite line where the intersection has occurred. If the intersection point projected onto vector **E** (the adjacent side) is outside the line segment (**V2-V1**) then it will be considered to have not intersected the edge. This means we have only bloated the edge in two dimensions and not three.

To understand the problem that we have, imagine that point **P** in Figure 12.35 was raised up so that it was only slightly higher than vertex **V2**. When the ray intersection point is projected onto the infinite line on which the edge is defined, it will be found to be higher than vertex **V2** and therefore, considered to be outside the line segment (**V2-V1**). In other words, the opposite side of the triangle (||L|| in Figure 12.35) is connecting with the central spine of the cylinder higher than vertex **V2**. The intersection with the infinite line is considered to occur beyond the edge extents when this is the case. However, we have not taken into consideration that the ends of each edge (the vertices themselves) should also be inflated by a distance equal to the sphere's radius to give complete padding around the edge end points. Where our edges end at the vertices, we should have domes filling these gaps. Figure 12.26 shows the gaps we currently have in our inflated triangle technique causing intersection tests to fail if not corrected. As you can see, we have not entirely padded the perimeter of the triangle as we should.

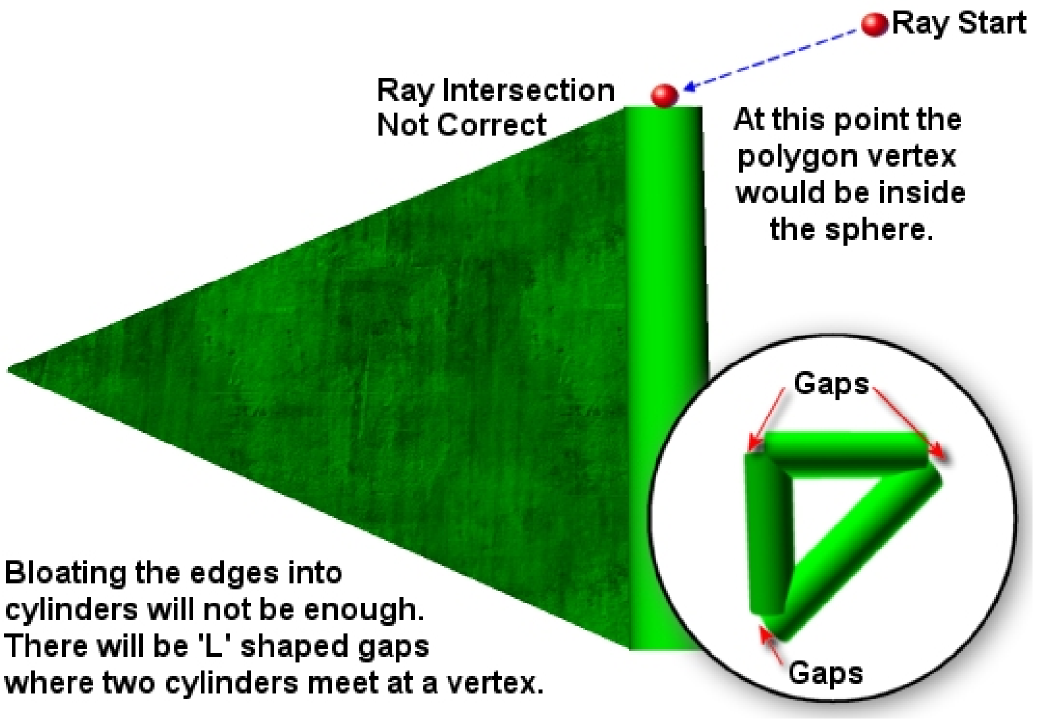

Ray Start
Ray Intersection Not Correct
At this point the polygon vertex would be inside the sphere.
Gaps
Gaps
Bloating the edges into cylinders will not be enough. There will be 'L' shaped gaps where two cylinders meet at a vertex.

Figure 12.36

In Figure 12.36 we see a ray that is intersecting the top vertex of the edge currently being tested. Since we essentially discount any intersection with the surface of the cylinder that (in this diagram) is higher or lower than the two vertices that form the edge, the ray/cylinder test would return false for intersection in this case. However, it is clear that the ray collides right with the top of the cylinder, where the cap should be. Unfortunately the top vertex in the diagram has had no inflating applied and the intersection is undetected. Remembering that the *t* value returned from our intersection tests is used to calculate the new position of the **center** of the sphere, we can see that if we imagine the position at the end of the ray in Figure 12.36 to be allowed to be the new position of the sphere center, the sphere surface would be embedded in the polygon when its radius is taken into account.

If you look at the circular inset in Figure 12.36, you can see that were we to do nothing other than the edge tests so far described, there would be gaps at the ends of each edge. This can cause the swept sphere to become embedded around the areas of the vertices.

To fix this problem we have to perform an additional test with the vertices when the ray does not intersect the cylinder (i.e., the swept sphere did not intersect the interior of the edge). To remove the gaps at our vertex positions, we need to inflate those areas too so that we have a consistent radius thick buffer around the entire triangle perimeter.

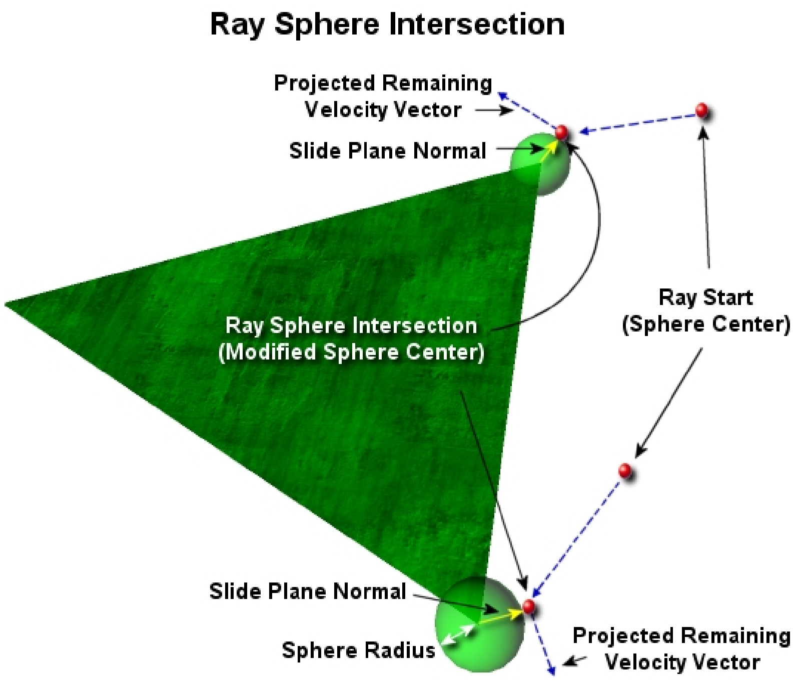

Ray Sphere Intersection
Projected Remaining Velocity Vector
Slide Plane Normal
Ray Start (Sphere Center)
Ray Sphere Intersection (Modified Sphere Center)
Slide Plane Normal
Sphere Radius
Projected Remaining Velocity Vector

Figure 12.37

We can do this using a similar method to the previous test. We can just inflate each vertex by the radius of our sphere (in all directions). This turns each vertex in the edge into a sphere of the correct radius and allows us to treat our swept sphere as a simple ray. We can then calculate the intersection between the swept sphere and a vertex using a ray/sphere intersection test.

Thus, for each edge, we will first test the ray against the inflated edge (a cylinder) as previously discussed. If no intersection occurs, we will then test the same ray against each inflated vertex (a sphere) in that edge. This fully buffers the edge of the triangle currently being tested. In fact, if we examine the buffering effect with respect to both tests, we see that each edge is transformed into a capsule. In Figure 12.38, we see how the edge looks from the perspective of our ray when all intersection tests for that edge are taken into account. The first edge test inflates the edge into a cylinder and that is followed by a test on each vertex of the edge. As the vertices are inflated into spheres, we form a smooth capsule around the edge for our response system to slide around.

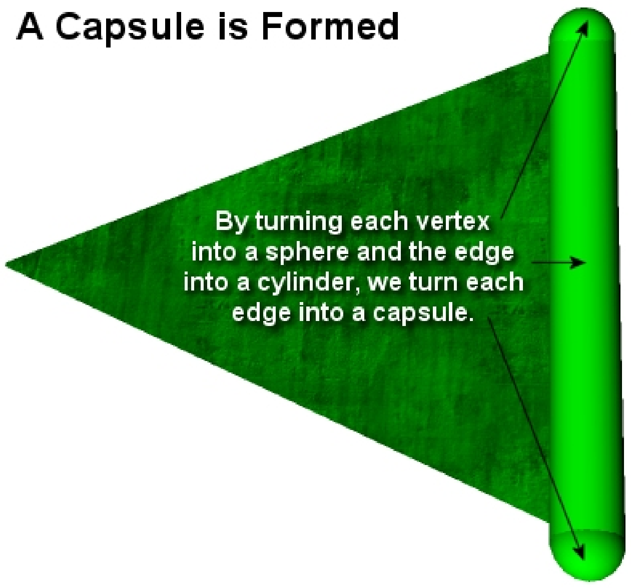

A Capsule is Formed
By turning each vertex into a sphere and the edge into a cylinder, we turn each edge into a capsule.

Figure 12.38

You will see in a moment that our edge intersection function will now have to solve (potentially) three quadratic equations. The first quadratic equation is for the intersection between a ray and a cylinder. Then we will have to perform two more intersection tests to determine if the ray (the swept sphere) intersects the spheres (the vertices) at each end of the capsule. As mentioned previously, the sphere tests only need to be performed if the ray has not collided with the cylinder.

When an intersection has occurred with one of the edge vertices (instead of the interior of the edge), the collision normal will be calculated by generating a normalized vector from the vertex (which is the center of the sphere) to the ray intersection point which is on the surface of the sphere (inflated vertex) as shown in Figure 12.39. This will generate a slide plane for the response phase allowing the ellipsoid to slide smoothly around the vertex after a collision.

Before we can look the code that determines these intersections, we still have a little more math to do. In order to test the ray against the sphere formed by each vertex, we must first learn how to substitute the ray equation into the sphere equation and manipulate the resulting quadratic equation into the standard form. Once we have learned how to do that, we will finally look at the complete set of code for the CCollision::SphereIntersectLineSegment function that performs all of these tests. This is the function that we called from CCollision::SphereIntersectTriangle for each edge in our triangle.

It should be noted that while the SphereIntersectLineSegment function is the main function call, for clarity the ray/sphere tests are performed in a separate helper function called SphereIntersectPoint. This function will be called from SphereIntersectLineSegment which directly handles only the ray/cylinder test.

12.7.2 Swept Sphere / Vertex Intersection

Figure 12.39 illustrates what we are trying to achieve with this intersection test. **Source** and **Dest** describe the initial and desired destination points for our swept sphere in this update. If we inflate the vertices into spheres equal to the radius of our swept sphere, we can treat our swept sphere as a ray. The ray will therefore intersect the sphere around the vertex giving us a new destination (**Modified Dest**) position for the center of the sphere that will be at an exact distance of Radius units from the actual vertex. This means, in reality, the surface of our sphere will be touching (but not intersecting) the vertex at this time. Therefore, the *t* value of intersection between the ray and the sphere surrounding the vertex provides the furthest point along the velocity vector that the center of the sphere can move to before penetrating the vertex.

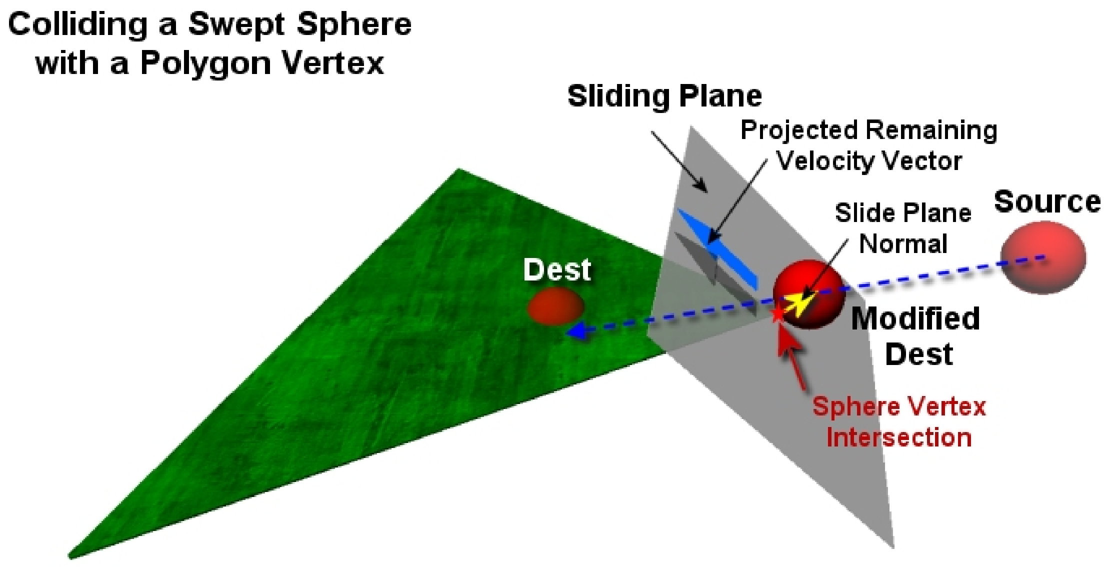

Colliding a Swept Sphere with a Polygon Vertex
Sliding Plane
Projected Remaining Velocity Vector
Slide Plane Normal
Source
Dest
Modified Dest
Sphere Vertex Intersection

Figure 12.39

As can also be seen in Figure 12.39, the response step will require a sliding plane. The slide plane normal (collision normal) is calculated by returning a normalized vector pointing from the vertex position to the intersection point (**Modified Dest**) that serves as the new position of the center of the sphere.

Before coding can commence, we need to examine the ray/sphere intersection test. Therefore, we must start with the sphere equation, substitute the ray equation into that sphere equation, and then manipulate the resulting quadratic into the standard form so that we can call SolveCollision method to solve for *t*.

The equation for a sphere that we will use is shown below. **P** is any point that is on the surface of the sphere if the equation is true. **C** is the position of the center of the sphere and R is the radius of the sphere.

$$\left\|(P - C\right\|^2 = R^2$$

It is a rather simple equation to understand. If the squared length of a vector created from the position of the center of the sphere to an arbitrary point **P** is equal in length to the squared radius of the sphere, the point must be on the surface of the sphere. For any point not on the surface of the sphere, the equation will not be true.

We do not wish to simply test a point **P** using this equation; instead we want to find the position *t* along a ray at which the equation becomes true. If we can do that, we have found the point on the ray that is at a distance of Radius units from the sphere center and is therefore the exact time at which the ray pierces the sphere surface.

We need to substitute the equation of our ray:

$$O + tV$$

into the sphere equation in place of **P** and then solve for *t*. After substituting **O+tV** for **P** in the sphere equation we have the following equation:

$$\left\| O + tV - C \right\|^2 = R^2$$

We must first expand the LHS so it is no longer a squared expression. We know that the squared length of the vector **O+tV-C** can be obtained by multiplying this vector by itself, so let us multiply out the square on the LHS:

$$(O + tV - C) \bullet (O + tV - C) = R^2$$

Now we have to multiply two bracketed terms again, but this time each bracket contains three terms instead of two. Although the equations we used FOIL on previously had only two terms, it still works the same way. We just start with the first term in the left brackets working from left to right, and for each term on the left, we multiply it with each term on the right and finally sum the results of each test to get the final result.

Step 1: First we multiply the first term in the left brackets with the first term in the right brackets.

$$O \bullet O = O^2$$

Step 2: Now we multiply the first term in the left brackets with the second term on the right. In this step we rearranged the result to have *t* on the right. This is really a matter of preference thing at this stage. As *t* is the unknown, we will eventually be trying to isolate it as much as possible, and moving it to the right keeps the vectors (**O** and **V**) together in the term and keeps *t* separate. Remember *t* is just a scalar so we have not altered the evaluation of the expression by doing this; we simply rearranged the term more to our liking:

$$O \bullet tV = O \bullet Vt$$

Step 3: Multiply the first term in the left brackets with the last term in the right brackets. Since **O · –C** is the same as **–O · C**, we write it like this to make a tidier term with the sign on the left.

$$O \bullet -C = -O \bullet C$$

Let us have a look at what our equation looks like so far by summing the three results we have:

$$O^2 + O \bullet Vt - O \bullet C$$

Step 4: We have now performed all steps for the first term in the left brackets, so we do it all again now for the second term. That is, we will multiply each term in the right brackets with the second term in the left brackets (***t*V**).

$$tV \bullet O = O \bullet Vt$$

We rearrange the order of the result again to make it easier later when simplifying the equation. Since we already have an **O · Vt** term (see above), collecting like terms will be very straightforward.

Step 5: Now we multiply the second term in the left brackets with the second term in the right brackets.

$$tV \bullet tV = tV^2 = t^2V^2 = V^2t^2$$

It is pretty clear now that this is definitely a quadratic-- it has a term that involves the unknown *t* raised to the second power.

Step 6: Next we multiply the second term in the left brackets with the last term in the right brackets, yielding the following:

$$tV \bullet -C = -C \bullet Vt$$

At this point we have multiplied the first and second terms in the left brackets with all terms in the right brackets. We could say at this point that we have two thirds of our equation's LHS complete. After the six steps we have completed, we have six terms which, when added together, yields this current LHS:

$$O^2 + O \bullet Vt - O \bullet C + O \bullet Vt + V^2t^2 - C \bullet Vt$$

Step 7: Finally, we have to multiply the final term in the left brackets with each term in the right brackets, giving us the final three terms for our expanded equation. First, we multiply the last term in the left brackets with the first term in the right brackets. We rearrange this to **–O · C** since we already have a term in this shape:

$$-C \bullet O = -O \bullet C$$

Step 8: Next we multiply the last term on the left with the middle term on the right.

$$-C \bullet tV = -C \bullet Vt$$

Step 9: And lastly, we multiply the two final terms of each bracket. This is the multiplication of two negatives **–C** and **–C** giving the positive result $\mathbf{C^2}$.

$$-C \bullet -C = C^2$$

Let us now see the final (and quite large) left hand side of our fully expanded quadratic equation:

$$O^2 + O \bullet Vt - O \bullet C + O \bullet Vt + V^2t^2 - C \bullet Vt - O \bullet C - C \bullet Vt + C^2$$

That initially looks pretty daunting, but there are lots of like terms we can collect. For example, look at the third term **(-O ·C**) and the and seventh term (also **–O · C**). As these are identical and we are just subtracting **–O · C** from the LHS twice, the equation can be collected in the third term like so:

$$O^2 + O \bullet Vt - 2(O \bullet C) + O \bullet Vt + V^2t^2 - C \bullet Vt - C \bullet Vt + C^2$$

We also have many terms containing *t* (not t^2) and we should collect these to form the middle term of our quadratic in standard form. If we collect *t* terms and then isolate *t*, we will also have our **b** coefficient for our standard form quadratic.

Let us just collect all the terms with *t* in them and line them up together:

$$O \bullet Vt + O \bullet Vt - C \bullet Vt - C \bullet Vt$$

We can see that we are adding together two (**O · Vt**) terms on the LHS and also subtracting the term (**C · Vt**) twice from the LHS. Thus, we can collapse these four terms into two terms:

$$2(O \bullet Vt) - 2(C \bullet Vt)$$

That looks a lot nicer, but we still need this arranged into a single term with an isolated *t*, so that we have our **b** coefficient available. Finding a common factor will solve all of our problems. With common factors, if there is some shared multiple across all terms, that multiple can be moved outside the brackets. In our case, we see that both terms have the number **2** and both terms have ***t***. Thus, both **2** and ***t*** are common factors which can be placed outside a set of brackets that contains the rest of the expression (we call this "factoring out" the 2 and the t):

$$2t(O \bullet V - C \bullet V)$$

Using the same logic, we can see that the two terms inside the brackets both contain **V**. Therefore, we can move **V** outside a new set of brackets that contain **O - C**.

$$2t(V \bullet (O - C))$$

Notice how we are careful to preserve the dot product sign since both **O - C** and **V** are vectors and should be vector multiplied.

Finally, let us put *t* over on the right hand side of the outer brackets to isolate it more clearly. This does not change the evaluation of the expression, but it does provide clarity for showing us what the **b** coefficient is going to be in standard form (**2 * (V · (O – C))**:

$$2(V \bullet (O - C))t$$

Plugging this back into our original equation in place of all the *t* terms we had before gives us the following LHS:

$$O^2 + 2(V \bullet (O - C))t - 2(O \bullet C) + V^2t^2 + C^2$$

Looking at this equation we can instantly see what our **a** coefficient is going to be. It is the value accompanying $\boldsymbol{t^2}$. We can see that this is $\mathbf{V^2}$. Let us move that term to the left of the equation so that it assumes its rightful position in the standard form:

$$V^2t^2 + O^2 + 2(V \bullet (O - C))t - 2(O \bullet C) + C^2$$

We can also see what the **b** coefficient in standard form is going to be. It is the expression accompanying ***t*** in the non-squared term: **2(V · (O - C))**. Therefore, let us move the *t* term to its rightful position in the standard form:

$$V^2t^2 + 2(V \bullet (O - C))t + O^2 - 2(O \bullet C) + C^2$$

Thus, we can see that our **c** coefficient (so far) must be $\mathbf{O^2 - 2(O \cdot C) + C^2}$. That expression is a little lengthy. But wait just a moment. Do we recognize any pattern in the term? Indeed we do. It matches the identity:

$$O^2 - 2OE + E^2 = (O - E)^2$$

In our current constant term we can see that **O** = **O** and **E** = **C** and thus, the identity tells us that it can it also be written in its more compact form $\mathbf{(O - C)^2}$. Below we show the full equation (RHS as well) so far:

$$V^2t^2 + 2(V \bullet (O - C))t + (O - C)^2 = R^2$$

We are almost there! The standard form quadratic must have zero on the RHS, so we must subtract $\mathbf{R^2}$ from both sides of the equation. This gives us the final ray/sphere intersection quadratic equation in standard form:

$$V^2t^2 + 2(V \bullet (O - C))t + (O - C)^2 - R^2 = 0$$

We can write this slightly differently to reflect a pattern that is more convenient for programmers (it is the same equation though):

$$(\mathrm{V} \bullet \mathrm{V})\mathrm{t}^2 + 2(\mathrm{V} \bullet (\mathrm{O} - \mathrm{C}))\mathrm{t} + (\mathrm{O} - \mathrm{C}) \bullet (\mathrm{O} - \mathrm{C}) - \mathrm{R} * \mathrm{R} = 0$$

Therefore, the final a, b, and c coefficients that we were searching for (which will be fed into our SolveCollision function to solve for *t*) are:

$$a = V \bullet V$$

$$b = 2 * (V \bullet (O - C))$$

$$c = (O - C) \bullet (O - C) - R * R$$

12.8 The SphereIntersectLineSegment Function

We are now ready to put everything we have learned in the last two sections into practice and write a function that will determine if a swept sphere intersects a triangle edge. Remember, this function is called three times from the SphereIntersectTriangle function (once per edge). Since the function is passed the current closest *t* value that has been found so far (tMax), it will only return true if an intersection happens with the edge at a distance closer than one already found. If this happens, the current *t* value will be overwritten with the new *t* value.

This first version of the function shows the code for the equations we have discussed. You will see in a moment that we will add some additional code to this function to correct the situation where the sphere may already be embedded in the edge prior to the intersection test being performed. Since this will complicate things slightly, we will first show the pure code and then worry about the special case handling.

```
bool CCollision::SphereIntersectLineSegment( const D3DXVECTOR3& Center,
                                             float Radius,
                                             const D3DXVECTOR3& Velocity,
                                             const D3DXVECTOR3& v1,
                                             const D3DXVECTOR3& v2,
                                             float& tMax,
                                             D3DXVECTOR3& CollisionNormal )
{
    D3DXVECTOR3 E, L, X, Y, PointOnEdge, CollisionCenter;
    float       a, b, c, d, e, t, n;

    // Setup the equation values
    E = v2 - v1;              // E = triangle Edge V2-V1
    L = Center - v1;          // L = Hypotenuse ( Vector from V1 to Ray Origin )
    e = D3DXVec3Length( &E );  // e = length of triangle edge

    if ( e < 1e-5f ) return false;  // Safety test:
                                    // If triangle edge has no length return false

    E /= e;             // Normalize E ( Unit length Adjacent )

    // Generate cross values
    D3DXVec3Cross( &X, &L, &E );       // ||X|| = Length of Opposite side
                                       // ( Hypotenuse * sin a )
                                       //    = Distance from Ray Origin to Edge
    D3DXVec3Cross( &Y, &Velocity, &E );  //  Y = Velocity x E

    // Setup the input values for the quadratic equation
    a = D3DXVec3LengthSq( &Y );            // ( VxE ) dot ( VxE )
    b = 2.0f * D3DXVec3Dot( &X, &Y );      // 2 *(RayOrigin - v1 x E) dot ( V x E )
    c = D3DXVec3LengthSq( &X ) - (Radius*Radius); // ( RayOrigin -v1 x E )^2 – R*R

    // Solve the quadratic for t
    if ( !SolveCollision(a, b, c, t) ) return false;
```

If you cross reference the way we calculate the a, b and c coefficients above, you should see that this matches exactly what we determined they should be when we arranged the ray/cylinder equation into a standard form quadratic equation. We simply plug these values into the SolveCollision function which solves the quadratic equation using the quadratic formula. Remember that the SolveCollision function will return in its output parameter *t*, the smallest positive solution to the quadratic. Therefore, if the function returns true, the ray did intersect the cylinder and local variable *t* will contain the first point of intersection in front of the ray origin. If no intersection is found, we can return false from this function immediately and pass program flow back to the calling function SphereIntersectTriangle (which will then proceed to test the other edges for intersection).

Notice that we have added a test that checks the length of the edge vector (*e*). If *e* is zero (with tolerance) then we have been passed a degenerate edge and return immediately.

The last half of this function is shown below. It first tests to see if the *t* value returned from SolveCollision is larger than the value passed in the tMax parameter. At this point, tMax will contain the closest *t* value found while testing all previous triangles/edges. We are only interested in finding the closest intersecting triangle, so if we have previously detected a collision with a triangle that happens closer to the ray origin than is described by the current *t* value we have just had returned, we can return immediately since we are not interested in this intersection. In that case, we have already hit something that is in front of it.

```
     if ( t > tMax ) return false; // Intersection distance is larger
                                   // than one already found so return false
```

If no closer intersection has yet been found, then we need to calculate the intersection point along the ray. This will also be the point of intersection on the surface of the cylinder. After testing all remaining triangles, if our detection phase determines that this is the closest intersection, this position (stored in the local CollisionCenter variable) will describe the new center position of our sphere (i.e., ellipsoid) such that the surface is just contacting the edge. Calculating this position along the ray is easy now that we have our *t* value.

```
    // Use returned t value to calculate intersection position along velocity ray
    CollisionCenter = Center + Velocity * t;
```

Now that we have a temporary position on the cylinder surface where intersection will occur, we create a vector from the base of the cylinder (v1) to this position. This forms the hypotenuse of a triangle with the edge as its adjacent leg. If we dot this vector with unit length edge vector E, we will scale the length of the hypotenuse by the cosine of the angle between them. This gives us the length of the adjacent leg. This length will describe the exact distance along the edge from vertex v1 to the point where the intersection with the edge has occurred. This allows us to test whether this point is between vertices v1 and v2. All we are essentially doing is projecting the collision point on the cylinder surface onto the infinite line on which the edge is contained.

```
    // Project this onto the line containing the edge ( v2-v1 )
    d = D3DXVec3Dot( &(CollisionCenter - v1), &E );

    if ( d < 0.0f ) // The intersection with the line happens before
                    // the first vertex v1, so test vertex v1 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v1, tMax, CollisionNormal);
    else
    if ( d > e )      // The intersection with line happens past
                    // the last vertex v2 so test vertex v2 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v2, tMax, CollisionNormal);
```

Once we have *d* (the distance from v1 to the point of intersection along the infinite line that contains the edge), we test to see if it is smaller than zero. If it is, then the intersection with the edge has occurred before the first vertex and therefore, the ray is considered non-intersecting with our main cylinder. In this case, we call the SphereIntersectPoint point function to test for intersection against vertex v1. However, if the distance to the point of edge intersection (*d*) from vertex v1 is larger than the length of the edge (*e*), an intersection may have occurred with the second vertex (v2), so the same intersection test is done for vertex v2. Notice that either way we return at this point.

The SphereIntersectPoint function is passed all the information it needs to perform its tasks properly. It is passed the tMax value so that it will store a new *t* value and return true only if an intersection occurs that is closer than the current value in tMax. It is also passed the output parameter CollisionNormal so that it can calculate a collision normal and return it back to the caller.

Finally, if we get past this part, we know we have an intersection with the edge cylinder closer than any that has previously been found. All our collision detection phase wants us to return at this point (and the same was true for the calling function SphereIntersectTriangle) are two pieces of information: the *t* value tMax and a collision normal (slide plane normal). We will calculate the normal by building a normalized vector from the point of intersection on the actual edge, to the point at which the ray intersected the surface of the cylinder (as described earlier). What we are actually doing then, is generating a vector from the point of intersection on the edge, to the center of the swept sphere at the time of intersection.

We calculate the point of intersection on the edge by scaling the unit length edge vector E by the distance to the point of intersection from the start of the edge (*d*) and adding the resulting vector to the edge start vector (v1).

```
    // Calculate the actual Point of contact on the line segment
    PointOnEdge = v1 + E * d;

    // We can now generate our normal, store the interval and return
    D3DXVec3Normalize( &CollisionNormal, &(CollisionCenter - PointOnEdge) );
    tMax = t;

    // Intersecting!
    return true;

}
```

With the collision normal generated, we also overwrite the value currently stored in tMax with the new closer *t* value we have found and then return true. We will look at the complete code to this function in one block in a moment after we discuss something else that must be added.

12.8.1 Ray / Cylinder Intersection: Embedded Object Correction

Recall that the first intersection test we call in SphereIntersectTriangle is the SphereIntersectPlane function. This is a simple ray/plane test where the swept sphere is reduced to a ray by shifting the plane. However, recall that we performed tests within that function to determine if the sphere was already embedded in the plane prior to the test being performed. When this is the case, we have very few options other than to calculate a new position for the sphere so that it is no longer intersecting. To address this issue we calculated the distance from the ray origin to the plane (that was behind it) and then moved the ray origin back along the normal so that it was sitting on the plane instead of behind it. This worked well because the ray origin represents the center of the sphere, and when the sphere center is located on the shifted plane, its surface is touching the actual plane (and is no longer embedded).
Note that when this embedded case correction had been performed, the *t* value returned was not a positive parametric ray position in the range [0.0, 1.0]. It was a negative value describing the actual world space distance to the plane. In other words, the *t* value describes to the detection phase how far the sphere should be moved backwards so that it is no longer embedded.

It seems reasonable that we should perform these same embedded tests and corrections when performing our cylinder and sphere tests. For example, it is quite possible that our swept sphere's surface may already be embedded in the triangle edge prior to the intersection test being performed.

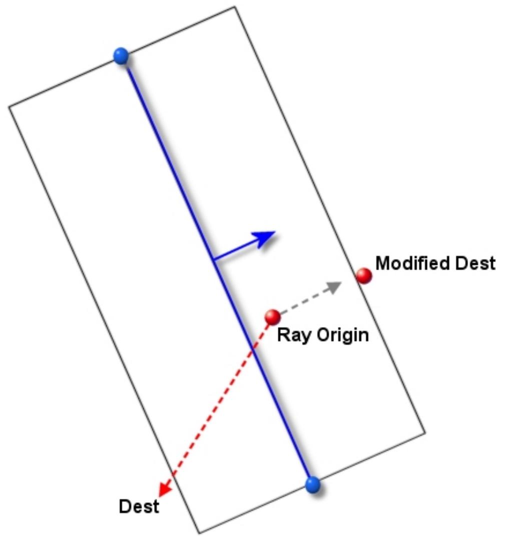


Figure 12.40

In Figure 12.40 we can see that the **Ray Origin** is inside the cylinder formed from the edge prior to the test being performed. **Dest** illustrates the position that the sphere is being requested to move to. In this case (like the plane case), we just ignore the velocity vector and move the sphere center (ray origin) in the direction of the edge normal so that it is no longer inside the cylinder and is instead located on the surface. Although we do not have the normal of the edge at this point, we can easily calculate it by projecting the ray origin onto the edge. We can then create a vector from the projected point on the edge to the original ray origin and make it unit length. Once we have the unit length edge normal, we can shift the ray origin along it. The distance we need to shift is equal to the radius of the cylinder minus the distance from the ray origin to the edge.

When this is the case, our SphereIntersectLineSegment function will not even have to bother solving the quadratic equation. We simply calculate the negative distance from the ray origin to the surface of the cylinder and return this as the *t* value to the detection phase. The collision normal returned will be the normal of the edge.

> **Note:** It is important to remember that none of our intersection techniques return or update the actual sphere position. That is done by the parent detection function (which we have not yet covered) that is called by our CollideEllipsoid function. All the intersection functions return is a slide plane normal and a *t* value of intersection. If negative, the *t* value will describe the actual world space distance the sphere must be moved so it is no longer embedded.

The question has to be asked, how do detect whether our ray origin is already within the cylinder? As it happens, we already have this answer stored in the **c** coefficient of our quadratic equation. Recall that the **c** coefficient was calculated like so:

$$c = ((O - v1) \times E) \bullet ((O - v1) \times E) - (Radius * Radius)$$

Where:

O	=	Sphere Center (Ray Origin)
v1	=	First vertex in edge
E	=	Unit length edge vector

Therefore,

((O-v1)xE) $\bullet$ **((O-v1)xE)** = Squared Distance from ray origin to edge

Since we subtract the squared radius when calculating **c**, this value will be zero if the ray origin is on the surface of the cylinder, positive if the ray origin is outside the cylinder, and negative if the ray origin is inside the cylinder (i.e., distance between the ray origin and the edge is smaller than the radius).

So after we have calculated the coefficients for the ray/cylinder quadratic equation, but before we have solved it, we can test **c**. If it is negative, we can execute the following code in lieu of solving the quadratic equation.

```
    if ( c < 0.0f )
    {
        // Product Ray Origin onto Edge 'E'
        d = D3DXVec3Dot( &L, &E );

        // Is this before or after line start?
        if (d < 0.0f)
        {
            // The point is before the beginning of the line,
            // test against the first vertex instead
            return SphereIntersectPoint( Center, Radius, Velocity,
                                         v1, tMax, CollisionNormal );

        } // End if before line start
        else if ( d > e )
        {
            // The point is after the end of the line,
            // test against the second vertex
            return SphereIntersectPoint( Center, Radius, Velocity,
                                         v2, tMax, CollisionNormal );

        } // End if after line end
```

The first section of this code calculates *d*. This is calculated by projecting the length of the vector from v1 to the ray origin (L) onto the unit length edge vector E. This gets us the distance from vertex v1 to the ray origin (hypotenuse) projected onto vector E (adjacent). The length of this adjacent side L describes the distance from vertex v1 to the ray origin projected onto the line containing the edge. If this distance value is smaller than zero or larger than the length of the edge (*e*), then the sphere is not embedded in the actual cylinder, but may be embedded in the vertices (the spheres). Therefore, if *d* is negative, we perform an embedded correction test for v1, or if *d* > *e*, we perform the embedded correction test against v2. In both cases, we will let the SphereIntersectPoint function sort out the embedded correction.

If *d* contains a distance value between v1 and v2 then the ray origin is inside the cylinder and we need to correct it. As shown below, we first calculate the point on the actual edge. This is the ray origin projected onto the line that contains edge (v2-v1). Since we already have *d* (the distance to this projected point along the edge from v1), we can calculate the actual point on the edge by adding the vector E**d* to v1.

```
        else
        {
            // Point within the line segment
            PointOnEdge = v1 + E * d;

            // Generate collision normal
            CollisionNormal = Center - PointOnEdge;
            n = D3DXVec3Length( &CollisionNormal );
            CollisionNormal /= n;

            // Calculate negative t value as actual distance
            // by subtracting distance from edge to ray origin
            // ( which is smaller than radius ) the radius.
            t = n - Radius;

            if (tMax < t) return false;

            // Store t and return
            tMax = t;

            // Edge Overlap
            return true;

        } // End if inside line segment

    } // End if sphere inside cylinder
```

After we have the point on the edge, we now have to calculate the edge normal. We do this by creating a vector from the point on the edge to the ray origin. We then record the length of this vector in *n*, which describes the distance from the ray origin to the actual edge. Next we divide the collision normal by *n* to normalize it and we have the vector we will use as our slide plane normal later on. Now we just have to calculate the *t* value (shown above). Since *n* describes the distance from the ray origin to the edge, and this is now guaranteed to be smaller than the radius of the cylinder, subtracting the radius from *n* will give us a negative *t* value describing the distance from the ray origin to the surface of the cylinder. Because this *t* value is negative, it is guaranteed to overwrite any non-embedded *t* value that has been recorded in tMax while checking previous triangles. We record this value in tMax and return. Our function will have returned the *t* value and collision normal back to the parent detection function.

Let us now merge all of these concepts together to create the final code to our CCollision::SphereIntersectLineSegment function (complete with embedded case correction).

12.8.2 The Final SphereIntersectLineSegment Function

```
bool CCollision::SphereIntersectLineSegment( const D3DXVECTOR3& Center,
                                             float Radius,
                                             const D3DXVECTOR3& Velocity,
                                             const D3DXVECTOR3& v1,
                                             const D3DXVECTOR3& v2,
                                             float& tMax,
                                             D3DXVECTOR3& CollisionNormal )
{
    D3DXVECTOR3 E, L, X, Y, PointOnEdge, CollisionCenter;
    float       a, b, c, d, e, t, n;

    // Setup the equation values
    E = v2 - v1;              // E = triangle Edge V2-V1
    L = Center - v1;          // L = Hypotenuse ( Vector from V1 to Ray Origin )
    e = D3DXVec3Length( &E ); // e = length of triangle edge

    if ( e < 1e-5f ) return false;  // Safety test…
                                    // if triangle edge has no length return false

    E /= e;             // Normalize E ( Unit length Adjacent )

    // Generate cross values
    D3DXVec3Cross( &X, &L, &E );      // ||X|| = Length of Opposite side
                                      // ( Hypotenuse * sin a )
                                      //    = Distance from Ray Origin to Edge
    D3DXVec3Cross( &Y, &Velocity, &E );   //  Y     = Velocity x E

    // Setup the input values for the quadratic equation
    a = D3DXVec3LengthSq( &Y );          // ( V x E ) dot ( V x E )
    b = 2.0f * D3DXVec3Dot( &X, &Y );    // 2*( RayOrigin-v1 x E ) dot ( V x E )
    c = D3DXVec3LengthSq( &X ) - (Radius*Radius); // ( RayOrigin-v1 x E )^2 - R*R

    if ( c < 0.0f )
    {
        // Product Ray Origin onto Edge 'E'
        d = D3DXVec3Dot( &L, &E );

        // Is this before or after line start?
        if (d < 0.0f)
        {
            // The point is before the beginning of the line,
            // test against the first vertex instead
            return SphereIntersectPoint( Center, Radius, Velocity, v1,
                                         tMax, CollisionNormal );

        } // End if before line start
        else if ( d > e )
        {
            // The point is after the end of the line,
            // test against the second vertex
            return SphereIntersectPoint( Center, Radius, Velocity,
                                         v2, tMax, CollisionNormal );

        } // End if after line end
        else
        {
            // Point within the line segment
            PointOnEdge = v1 + E * d;

            // Generate collision normal
            CollisionNormal = Center - PointOnEdge;
            n = D3DXVec3Length( &CollisionNormal );
            CollisionNormal /= n;

            // Calculate negative t value as actual distance
            // by subtracting distance from edge to ray origin
            // ( which is smaller than radius ) the radius.
            t = n - Radius;

            if (tMax < t) return false;

            // Store t and return
            tMax = t;

            // Edge Overlap
            return true;

        } // End if inside line segment

    } // End if sphere inside cylinder

    // If we are already checking for overlaps, return
    if ( tMax < 0.0f ) return false;

    // Solve the quadratic for t
    if ( !SolveCollision(a, b, c, t) ) return false;

    if ( t > tMax ) return false; // Intersection distance is larger
                                  // than one already found so return false

    // Use returned t value to calculate intersection position along velocity ray
    CollisionCenter = Center + Velocity * t;

    // Project this onto the line containing the edge ( v2-v1 )
    d = D3DXVec3Dot( &(CollisionCenter - v1), &E );

    if ( d < 0.0f ) // The intersection with the line happens
                    // before the first vertex v1 so test vertex v1 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v1, tMax, CollisionNormal);
    else
    if ( d > e )     // The intersection with line happens
                     // past the last vertex v2 so test vertex v2 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v2, tMax, CollisionNormal);

    // Calculate the actual Point of contact on the line segment
    PointOnEdge = v1 + E * d;

    // We can now generate our normal, store the interval and return
    D3DXVec3Normalize( &CollisionNormal, &(CollisionCenter - PointOnEdge) );
    tMax = t;

    // Intersecting!
    return true;

}
```

One line worth drawing your attention to is highlighted in bold:

```
  // If we are already checking for overlaps, return
  if ( tMax < 0.0f ) return false;
```

If we have gotten to this point in the function, the swept sphere is not embedded in the edge currently being tested. Normally, a quadratic equation would then be solved to find the *t* value of intersection along the ray. However, if the tMax value passed in is already negative, then we will not waste time determining the intersection point along the ray since we have already found a triangle (in a previous test) in which the swept sphere is embedded. From that point on, we are only interested in finding triangles for which the swept sphere is potentially *more* embedded.

You can think about our intersection routines as working in one of two modes. Until we find an embedded triangle, we solve quadratic equations to find a closest *t* value. In this mode, the *t* value will be a parametric distance along the ray in the range of [0.0, 1.0]. However, as soon as we find an embedded triangle, tMax is set to a negative distance value which describes how to correct the situation by moving the sphere. At this point, the intersection tests are switched into what we might call 'Embedded Correction Mode'. For the remainder of our tests, we are only interested in other triangles that the sphere is embedded in. Therefore, tMax becoming negative switches into this Embedded

Correction Mode, and this can be done by the SphereIntersectPlane, SphereIntersectLineSegment and SphereIntersectPoint functions.

We have just one more function to write before we have completed the methods needed to sweep a sphere against a triangle. The SphereIntersectLineSegment function calls a function called SphereIntersectPoint. This is the final missing piece we need to discuss. We will then have covered every separate intersection test called directly or indirectly by the SphereIntersectTriangle function.

12.9 The SphereIntersectPoint Function

SphereIntersectPoint is called by the previous function if no intersection is found between the ray and the cylinder formed by the edge. This additional test looks to see if the ray intersects the sphere that we construct around each vertex (which we do to fully pad the triangle edge). The code just calculates the **a**, **b**, and **c** coefficients of the quadratic equation and passes them into our SolveCollision. It will return either true or false depending on whether the ray intersects the sphere. If the *t* value returned is greater than the one we currently have in the tMax parameter, we ignore it and return false for intersection; otherwise, we overwrite the value stored in tMax with our new, closer intersection *t* value so that it can be returned to the caller. We then calculate this new position along the ray (which is on the surface of the inflated sphere at a distance of Radius units from the vertex). Finally, we calculate the collision normal by generating a unit length vector from the vertex to the intersection point on the surface of the sphere.

As with the previous function, if that was all we had to worry about then the function would be extremely small. However, we also have to cater for the case where the ray origin is already inside the sphere surrounding the vertex. If it is then we will not bother solving the quadratic equation and will simply calculate the distance we need to move the sphere center position (the ray origin) along the collision normal so that it is no longer penetrating the vertex. In other words, just as we have done in all of our intersection functions, we will return a negative *t* value which describes the actual distance (not parametric) we have to move the sphere center position back so that it is correctly positioned at a distance of Radius from the vertex, just contacting the inflated sphere surrounding that vertex.

We will look at this function in a few sections. In the first part of the function we calculate the **a**, **b**, and **c** coefficients of the quadratic equation using the mathematics we discussed earlier.

```
bool CCollision::SphereIntersectPoint( const D3DXVECTOR3& Center,
                                       float Radius,
                                       const D3DXVECTOR3& Velocity,
                                       const D3DXVECTOR3& Point,
                                       float& tMax,
                                       D3DXVECTOR3& CollisionNormal )
{
    D3DXVECTOR3 L, CollisionCenter;
    float       a, b, c, l, l2, t;

    // Setup the equation values
    L  = Center - Point;
    l2 = D3DXVec3LengthSq( &L );

    // Setup the input values for the quadratic equation
    a = D3DXVec3LengthSq( &Velocity );
    b = 2.0f * D3DXVec3Dot( &Velocity, &L );
    c = l2 - (Radius * Radius);
```

We only need to solve this equation if the **c** coefficient is non-negative. The **c** coefficient contains the squared length of a vector from the ray origin to the vertex, minus the squared radius of the sphere. If it is negative, then the distance from the vertex to the ray origin is smaller than the radius of the inflated sphere surrounding the vertex. This tells us that the ray origin is already inside the sphere. In reality, this means the vertex is embedded inside our swept sphere and we need to back the swept sphere away from the vertex so that this is no longer the case. The code that calculates the *t* value to accomplish this is shown below.

```
    // If c < 0 then we are overlapping, return the overlap
    if ( c < 0.0f )
    {
        // Remember, when we're overlapping we have no choice
        // but to return a physical distance (the penetration depth)
        l = sqrtf( l2 );
        t = l - Radius;

        // Outside our range?
        if (tMax < t) return false;

        // Generate the collision normal
        CollisionNormal = L / l;

        // Store t and return
        tMax = t;

        // Vertex Overlap
        return true;

    } // End if overlapping
```

We store the distance from the ray origin (the center of the swept sphere) to the vertex (the center of our inflated sphere) in local variable *l*. Since we know that this value is already smaller than the radius of the inflated sphere (otherwise we would not be in this code block) we can subtract the radius from this value to get the negative distance (*t*) we need to move the ray origin so that it is located on the surface of the inflated sphere (i.e., the vertex is now just touching the surface of the swept sphere). If this *t* value is larger than the one already stored in tMax, it means that we have found another intersection in which the swept sphere is more deeply embedded. So we can ignore the current one and return false for intersection. Otherwise, we calculate the collision normal as the unit length vector from the vertex to the ray origin. We then store our *t* value in tMax and return true for intersection. From this point on, our detection phase will only be intersected in finding embedded polygons that are more deeply embedded than this one.

Beneath this code block is the code that gets executed when the current vertex is not already embedded in our swept sphere. Before solving the quadratic, we first test whether the current value stored in tMax is negative. If it is, it means a previous intersection test found an embedded situation and we are no longer interested in anything other than finding another triangle, edge, or vertex that is more deeply embedded than the current one. So we return false for intersection.

If that is not the case, we are still searching for a closer intersection along the ray and we call our SolveCollision function to see whether the ray intersects the sphere surrounding the vertex. If not, we return false; otherwise, we test the returned *t* value and return false if it is found to be greater than the value we already have stored in tMax. We do this because, although we have found an intersection, it is happening at a further distance along the ray from the origin than an intersection we found on a previous test. Remember, our collision detection phase is only interested in finding the closest colliding triangle.

```
    // If we are already checking for overlaps, return
    if ( tMax < 0.0f ) return false;

    // Solve the quadratic for t
    if (!SolveCollision(a, b, c, t)) return false;

    // Is the vertex too far away?
    if ( t > tMax ) return false;
```

Finally, we calculate the point of intersection along the ray and use this to create a collision normal. The normal is generated by normalizing a vector from the vertex to the intersection point on the surface of the inflated sphere surrounding the vertex.

```
    // Calculate the new sphere position at the time of contact
    CollisionCenter = Center + Velocity * t;

    // We can now generate our normal, store the interval and return
    D3DXVec3Normalize( &CollisionNormal, &(CollisionCenter - Point) );

    tMax = t;

    // Intersecting!
    return true;
}
```

As can be seen above, we store the new *t* value in tMax so the calling function can pass it back to the collision detection phase and use it for comparisons in future triangle intersection tests.

And there we have it. We have discussed all the intersection techniques that we need to understand in order to implement this collision system. Primarily we have described exactly how we test a swept sphere against a triangle. First, we test against the plane of the triangle, then we test against the interior of the triangle, and finally tests are performed against each edges and vertices of the triangle.

Before leaving this section, here is another look at the SphereIntersectTriangle function:

```
bool CCollision::SphereIntersectTriangle(  const D3DXVECTOR3& Center,
                                           float Radius,
                                           const D3DXVECTOR3& Velocity,
                                            const D3DXVECTOR3& v1,
                                            const D3DXVECTOR3& v2,
                                            const D3DXVECTOR3& v3,
                                            const D3DXVECTOR3& TriNormal,
                                            float& tMax,
                                            D3DXVECTOR3& CollisionNormal )
{

    float       t = tMax;
    D3DXVECTOR3 CollisionCenter;
    bool        bCollided = false;

    // Find the time of collision with the triangle's plane.
    if ( !SphereIntersectPlane( Center, Radius, Velocity, TriNormal, v1, t ))
        return false;

    // Calculate the sphere's center at the point of collision with the plane
    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);

    // If this point is within the bounds of the triangle,
    // we have found the collision
    if ( PointInTriangle( CollisionCenter, v1, v2, v3, TriNormal ) )
    {
        // Collision normal is just the triangle normal
        CollisionNormal = TriNormal;
        tMax            = t;

        // Intersecting!
        return true;

    } // End if point within triangle interior

    // Otherwise we need to test each edge
    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v1, v2, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v2, v3, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v3, v1, tMax, CollisionNormal );

    return bCollided;
}
```

Keep in mind that while at several points in the various intersection techniques we temporarily calculate the actual intersection point, the only thing this function actually returns to the caller is a *t* value and a collision normal. It is the caller that will use this information to return the actual position of the closest intersection back to the CollideEllipsoid function as the new position of the moving entity.

We have spent the last several sections discussing how to detect the intersection between a swept sphere and a triangle. However, we have not yet introduced the function which serves as the glue that binds the detection phase together. Earlier we covered the code to the CollideEllipsoid function, which was called by the application each time it wanted to update a moving entity. This function would invoke the collision detection phase to determine an intersecting position for the entity and generate a collision normal if an intersection occurred. This would serve as the data for a new sliding plane, upon which we would project our velocity and attempt to shift positions. The function would go back and forth between detection and response until the initial velocity was all used up (via projection onto the slide plane) or until a slide vector was calculated that the sphere could move along and spend the remainder of its velocity without any further intersections. CollideEllipsoid would iteratively make the call to the EllipsoidIntersectScene function, which is the main function in the detection phase. It is this function that will call the SphereIntersectTriangle function for every potential colliding polygon in the collision system's geometry database. It is this function that, after calling the SphereIntersectTriangle function for every triangle, will calculate the updated position for the ellipsoid and return it back to the response phase. Let us look at this function then so that we can get a final view of our overall collision system.

12.10 The EllipsoidIntersectScene Function

This function is the front end for the collision detection phase and is ultimately responsible for returning collision data back for use in the response phase. You will recall from our earlier discussions that it was passed the eSpace position of the center of the ellipsoid, the ellipsoid's radius, and the eSpace velocity vector. Hopefully, you also remember that our CCollision class maintains an array of **CollIntersect** structures which are used to transport collision information back to the response phase. Although the detection phase is only interested in returning the new position of the ellipsoid based on the closest intersection, it may be the case that the sphere intersects several triangles at exactly the same time. When this is the case, the information about each intersection will be returned in this array.

```
struct CollIntersect
{
    D3DXVECTOR3     NewCenter;
    D3DXVECTOR3     IntersectPoint;
    D3DXVECTOR3     IntersectNormal;
    float           Interval;
    ULONG           TriangleIndex;
};
```

The CollIntersect structure stores the new position of the ellipsoid so that the application can update the position of the moving entity. It also stores the actual intersection point on the surface of the ellipsoid and the collision (slide plane) normal. It should be noted that these vectors are all currently in eSpace and should be transformed back into world space before being handed back to the application. We saw that this is exactly what the CollideEllipsoid function did before returning the final position and integration velocity back to the application. The *t* value of intersection is also stored in the Interval float. If multiple collisions occur simultaneously, the information will be returned in multiple CollIntersect structures. When this is the case, keep in mind that the new position of the ellipsoid in eSpace and the interval will be exactly the same in each structure. Since we only return information for the nearest intersection, they must share the same *t* value.

Each structure also returns the index of the triangle in the collision system's geometry database that was intersected. This can be very useful to an application. For example, imagine you were using this collision system to track a ball around a pinball table. When the ball hits certain triangles, our application may want to know which ones they were since this may increase or decrease the player's score. Alternatively, you might decide to check some application associated triangle material property and play a certain sound effect (maybe a different footstep sound depending on the material underfoot) or trigger a scripted event upon a particular collision.

Although we have not focused much on how the collision geometry will be stored in our collision system (this is covered in detail in the workbook), for now just know that we use registration functions such as CCollision::AddIndexedPrimitive to register all the triangles in a mesh with the collision system. In fact, there is even a CCollision::AddActor function which steps though the frame hierarchy of an actor, transforms each mesh into world space, and then adds the triangles of each transformed mesh to the collision database. This means that we could load an entire scene from a hierarchical X file into an actor, and then register all the triangles in that scene with the collision system with a single function call.

Ultimately, each triangle that is registered with the collision system has its vertices added to an internal array of D3DXVECTOR3s. In our case, we are only interested in the positional information at each vertex and there is no need to store other per vertex properties. The CCollision class uses an STL vector to manage this data. The following line of code is located in CCollision.h and defines an STL vector of D3DXVECTOR3 structures called CollVertVector. The vertices of every triangle registered with the collision system will be added to this single array.

```
typedef std::vector<D3DXVECTOR3>    CollVertVector;
```

A vector of this type is then declared as a member variable of the CCollision class as shown below.

```
CollVertVector      m_CollVertices;
```

When the vertices of a triangle are added to the vector, its indices are also stored inside a CollTriangle structure (also defined in CCollision.h) along with other per triangle properties such as its normal and a surface material index which we might find useful. This structure is shown below.

```
struct CollTriangle
{
        ULONG       Indices[3];
        D3DXVECTOR3 Normal;
        USHORT      SurfaceMaterial;
}
```

We called our triangle data a material index since it is common for collision systems to use surface material properties to determine responses (e.g., the application of friction). Of course, you could decide to make this a more generic data storage concept. As discussed a moment ago, you might use this member to store a value that can be added to the player's score or health when a triangle of this type is collided with. Or you might even decide to use this value to index into an array of callback functions. Basically, the idea is to provide us the ability to store a per-triangle code that might be useful in our application.

Every triangle that is added to the collision system is stored in one of these structures. Its three indices reference into the CollVertVector array describing the vertices that comprise the triangle. Triangle storage will take the form of a dynamic array (another STL vector) of CollTriangle structures as shown below. This array is called CollTriVector.

```
typedef std::vector<CollTriangle>   CollTriVector;
```

A vector of this type is declared as a member of the CCollision class as shown below.

```
CollTriVector       m_CollTriangles;
```

It is not important in this discussion to understand how the triangle data gets added to these arrays; the workbook will discuss such application specific topics. However, we do know that our EllipsoidIntersectScene function can access all the triangle and vertex data in these two arrays. Indeed our collision geometry database at this point is basically just these two STL vectors.

Now that we know how the collision data is stored inside the CCollision class, let us have a look at the parameter list to the function.

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG &IntersectionCount )
```

The Center parameter and the Velocity parameter describe the center position of the ellipsoid and the direction and magnitude for the movement request. Recall that before this function was called by the CollideEllipsoid function, these values were already in eSpace. We also pass in the radius vector of our ellipsoid, which is used to convert each triangle into eSpace prior to testing it for intersection. As the fourth parameter, the CCollision object's CollIntersect array is passed so that the detection function can fill it with intersection information and return it to the response phase. The final parameter is passed by reference from the CollideEllipsoid function. It is a DWORD value which will keep track of how many CollIntersect structures in the passed array contain valid intersection information. Remember, this array is re-used to transport information every time this function is called.

Note: If necessary, please refer back to our discussion of the CollideEllipsoid function. This is the function that calls EllipsoidIntersectScene and retrieves the information from the CollIntersect array.

Let us now look at this final function one section at a time. It is responsible for testing each triangle in the database for intersection and recording the nearest intersection results in the input **CollIntersect** array.

The first part of the function calculates the inverse radius vector of the ellipsoid. This will be needed to scale the vertex of each triangle into eSpace. We also copy the passed parameters Center and Velocity into two local variables called eCenter and eVelocity.

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG & IntersectionCount )
{
    D3DXVECTOR3 eCenter, eVelocity, InvRadius;
    D3DXVECTOR3 ePoints[3], eNormal;
    D3DXVECTOR3 eIntersectNormal, eNewCenter;
    ULONG       Counter, NewIndex, FirstIndex;
    bool        AddToList;
    float       eInterval;
    ULONG       i;

    // Calculate the reciprocal radius to minimize division ops
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );

    eCenter   = Center;
    eVelocity = Velocity;

    // Reset ellipsoid space interval to maximum
    eInterval = 1.0f;

    // Reset initial intersection count to 0 to save the caller having to do this.
    IntersectionCount = 0;
```

Notice above that the local variable eInterval is initially set to one. It is this variable that we will be passing into the intersection routines as tMax. As closer intersections are found, the value of eInterval will be overwritten to contain the nearer *t* values. Initializing it to 1.0 describes an intersection time at the very end of the ray. That is, the ray is completely clear from obstruction unless our intersection routines detect a collision somewhere along the ray and update it with a smaller value. We also set the passed IntersectCount variable to zero because we have not yet found any intersections and thus, have not added anything useful to the passed CollIntersect array.

Now we have to loop through each triangle in the database. We create an STL iterator to step though the CollTriVector one triangle at a time. Then we use the triangle's indices array to fetch the correct vertices from the m_CollVertices array. We scale the vertex positions (using theVec3VecScale function) by the Inverse ellipsoid radius vector, transforming them into eSpace, and store them in the local vector array ePoints.

```
    // Iterate through our triangle database
    CollTriVector::iterator Iterator = m_CollTriangles.begin();
    for ( Counter = 0; Iterator != m_CollTriangles.end(); ++Iterator, ++Counter )
    {
        // Get the triangle descriptor
        CollTriangle * pTriangle = &(*Iterator);
        if ( !pTriangle ) continue;

            // Get points and transform into ellipsoid space
            ePoints[0] = Vec3VecScale( m_CollVertices[ pTriangle->Indices[0] ],
                                       InvRadius );
            ePoints[1] = Vec3VecScale( m_CollVertices[ pTriangle->Indices[1] ],
                                       InvRadius );
            ePoints[2] = Vec3VecScale( m_CollVertices[ pTriangle->Indices[2] ],
                                       InvRadius );

            // Transform normal and normalize
            // (Note how we do not use InvRadius for the normal)
            eNormal = Vec3VecScale( pTriangle->Normal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );
```

We convert the normal of the current triangle into eSpace as well, by scaling it with the radius of the ellipsoid and normalizing the result. Note that we do not use the inverse radius as is the case with the vertices; it works the other way around for direction vectors.

At this point we have the three eSpace vertices for the current triangle we are about to test and also have its eSpace normal. We now have everything we need to call our very familiar SphereIntersectTriangle function:

```
        // Test for intersection with a unit sphere
        // and the ellipsoid space triangle
        if ( SphereIntersectTriangle( eCenter, 1.0f, eVelocity,
                                      ePoints[0], ePoints[1], ePoints[2], eNormal,
                                      eInterval, eIntersectNormal ) )
```

When we call this function we pass in the eSpace center of the ellipsoid (the ray origin) and the radius of the ellipsoid in eSpace. We know that this is always 1.0 since the ellipsoid in eSpace becomes a unit sphere. We also pass in the eSpace velocity vector (ray delta vector) followed by the three eSpace vertices of the triangle. The eighth parameter is the eInterval local variable which was set to 1.0 at the start of the function. If an intersection occurs between the swept sphere and this triangle, this variable will contain the new (smaller) *t* value on function return. As the final parameter we also pass in a local 3D vector variable (eIntersectNormal) which will store the collision normal if the function returns true for intersection.

The next section of code is inside the conditional code block that is executed if an intersection occurs (i.e., SphereIntersectTriangle returns true). Let us look at this code a few sections at a time.

The first thing we do is test the value currently stored in the local eInterval variable. If we are in this code block, then SphereIntersectTriangle not only determined an intersection, but it also found one that occurs closer to the ray origin then any previous found. Since eInterval is passed into this function for *t* value storage, the value stored in the variable at this point will be the new *t* value of intersection along the velocity vector.

If this is a positive value, then eInterval contains the parametric distance of the intersection point along the ray and will be in the 0.0 to 1.0 range. When this is the case, the new sphere position can be calculated by scaling the velocity vector by this amount and adding it to the current center position of the sphere. If eInterval is a negative value then it means the sphere was embedded in the triangle. When this is the case, eInterval will store the actual world space distance by which the sphere's center must be moved back in the direction of the triangle plane normal. This will assure that the surface of the sphere is no longer embedded.

```
    {
           // Calculate our new sphere center at the point of intersection
           if ( eInterval > 0 )
               eNewCenter = eCenter + (eVelocity * eInterval);
           else
               eNewCenter = eCenter - (eIntersectNormal * eInterval);
```

At this point we now have the new eSpace position of the sphere. Of course, we have many triangles to test, so we may find closer intersections and overwrite the new sphere center position with an even closer one in future iterations of the loop.

As discussed, the intersection information will be returned from this function using the Intersections array that was passed in. At first the IntersectCount value will be zero and we will store this new intersection information at the start of the array. If we find that our new intersection *t* value is smaller than the information we currently have stored in the Intersections array, we will overwrite it with the new information. Remember, all we wish to return in the Intersections array is the closest collision. If intersection happens simultaneously with multiple triangles, this will require us to store the information for each of these collisions in its own CollIntersect structure.

If we do find an intersection which element in the array should we store it in? We first test to see if IntersectionCount equals zero. If it does, then this is the first intersection we have found and therefore, we will store this information at index zero in the Intersections array. If the interval is smaller than the interval currently stored in the first element in the array, it means that our new one is closer and we should discard all the others and add this new one in their place. To do this we simply say that if the new intersection interval is smaller than the intervals currently stored in the intersection array, set the index to 0 so that it overwrites the first element in the array. Then set the number of intersections to 1 so all older information (that may still be stored in elements 2, 3, and 4, etc.) are ignored.

```
           // Where in the array should it go?
           AddToList = false;
           if ( IntersectionCount == 0 || eInterval < Intersections[0].Interval )
           {
               // We either have nothing in the array yet,
               // or the new intersection is closer to us
               AddToList         = true;
               NewIndex          = 0;
               IntersectionCount = 1;
           } // End if overwrite existing intersections
```

The else code block associated with the conditional above is executed only if the new interval returned for this latest intersection matches (with tolerance) the interval values currently stored in the intersection array. Essentially we are saying that if our intersection interval is the same as the intervals we currently have stored in our intersection array, then we have found another collision that happens at the exact same time as those already in the list. Thus, the size of the array should be increased and the new data added to the end.

```
           else if ( fabsf( eInterval - Intersections[0].Interval ) < 1e-5f )
           {
               // It has the same interval as those in our list already, append to
               // the end unless we've already reached our limit
               if ( IntersectionCount < m_nMaxIntersections )
               {
                   AddToList         = true;
                   NewIndex          = IntersectionCount;
                   IntersectionCount++;

               } // End if we have room to store more

           } // End if the same interval
```

Studying the two code blocks above you should be able to see that if the intersect array contained 10 CollIntersect structures describing collisions that happen at time *t* = 0.25, as soon as a intersection happens with a *t* value of 0.15 (for example), the structure at the head of the array would have its data overwritten and the array would be snapped back to an effective size of 1 element.

Provided one of the above cases was true, the local AddToList Boolean will be set to true. Checking that this is the case, we then finally add the collision data to the intersection array. NewIndex was calculated in the above code blocks and describes whether we are adding the data to the back of the array, or whether we are overwriting the head of the array.

```
            // Add to the list?
            if ( AddToList )
            {
                Intersections[ NewIndex ].Interval  = eInterval;
                Intersections[ NewIndex ].NewCenter = eNewCenter +
                                                (eIntersectNormal * 1e-3f);
                Intersections[ NewIndex ].IntersectPoint = eNewCenter –
                                                          eIntersectNormal;
                Intersections[ NewIndex ].IntersectNormal = eIntersectNormal;
                Intersections[ NewIndex ].TriangleIndex = Counter;

            } // End if we are inserting in our list

        } // End if collided

    } // Next Triangle

    // Return hit.
    return (IntersectionCount > 0);
}
```

We do this for every triangle and finally return true or false depending on whether at least one intersection was found (i.e., IntersectionCount > 0).

Notice in the above code how interval, the collision normal (eIntersectNormal) and the triangle index (Counter) are copied straight into the relative fields of the CollIntersect structure. Also note how the new non-intersecting position of the center of the sphere (moved as far along the velocity vector as possible) is slightly pushed back from the plane by a distance of 0.0001. This is something we added to provide a little more stability when working in the world of floating point inaccuracies. Although we know that our intersection functions have correctly determined an interval value that was used to create the new position of the center of the ellipsoid, it is positioned such that its surface is just contacting the triangle. We do not want floating point rounding errors to turn this into an embedded situation, so we slightly shift the sphere away from the triangle by a very small amount to give us a little bit of space between the surface and the ellipsoid.

Finally, notice that we return the intersection position (in eSpace). If you refer back to our discussion about how the slide plane normal is generated, you will see that moving the new sphere position along the slide plane normal (which is unit length and also equal to the radius of the unit sphere) will provide us with the intersection point in eSpace on the surface of the unit sphere. Although our particular response implementation may not require all of this data, it seemed better to provide as much information about the collision as we can and let the caller choose what they need to accomplish their specific objectives.

Believe it or not, at this point, we have now finished our core collision detection and response system implementation discussion. Moving forward, we now have a robust way to collide and slide our entities as they intersect with static scene geometry. This will certainly make our demos feel much more 'game-like' than before.

12.11 Final Notes on Basic Detection/Response

We have obviously discussed a good deal of code in this textbook, but we have tried to keep that discussion as tied into the theory and the mathematics as possible. Utility functions that add polygons to the collision system's database and such minor players have not been covered here. These will fall into the domain of the chapter workbook and as such, a more complete description of the source code can be found there.

We have now learned how to implement a collision detection and response system that works nicely with moving entities and static scene geometry. However, some of you may have crossed referenced the code shown in this chapter with the code in Lab Project 12.1 and encountered some differences. For example, while the core intersection routines are the same, the CollideEllipsoid function and the EllipsoidIntersectScene functions probably look a bit more complicated in Lab Project 12.1. Of course, there is a perfectly good reason for this -- in Lab Project 12.1, we have implemented a more fully featured system that supports terrains, actors, and even referenced actors. In other words, the collision system supports animated frame hierarchies. This means that we could register an actor with the collision system where that actor has moving doors, lifts, and conveyor belts that are updated every time the actor's AdvanceTime function is called. Our collision system will track the movement of these dynamic objects and make sure that the entity that is having its position updated correctly intersects and responds to this dynamic geometry. For example, if the player stands on an animated lift platform, when the lift moves upwards, the player's ellipsoid should also be pushed upwards with it. Doors that swing or slide open should correctly knock the player back, and so on.

This is quite a complex system and its code could not possibly be tackled until the foundation of a more simplistic collision detection and response phase was put into place. Therefore, the majority of this chapter has been focused on presenting a system that deals only with static geometry. That is, once the mesh/actor/terrain/etc. triangles are registered with the collision system's database in world space, that is how they will remain. This works perfectly for many situations and definitely has allowed us to come to grips with implementing a 'collide and slide' system.

Now the time has come to take the next step along this road and look to include support for dynamic collision geometry. All of the work we have done so far has not been in vain; supporting dynamic objects will involve adding features to the code we currently have, not replacing it. Our discussions of adding dynamic object management to our collision system will be from a relatively high level here in the textbook and we will save the detailed source code examination for the workbook.

12.12 Dynamic Collision Geometry

Our collision system currently stores its geometry using two dynamic arrays. Every single triangle added to the system will have its vertices stored in the CCollision::m_CollVertices array and its triangle information (indices, normal, etc.) stored in the CCollision::m_CollTriangles array. As we have seen, it is these two arrays that contain all the geometry that we wish to be considered as collidable with respect to our moving objects. We will see in the workbook that when we add a mesh to the collision system, all its vertices and indices will be added to these arrays.

We can add an entire CActor to the collision system also. The CCollision::AddActor function is passed an actor and will automatically traverse the frame hierarchy looking for mesh containers. For each mesh container it finds, it will grab its vertices and indices and add them to these two arrays. This single function could be used to register your entire scene with a single function call were it all to be stored in a single frame hierarchy loaded into a CActor object. We even have functions to register our CTerrain objects with the collision database.

All of these functions are simple helper functions that will take the passed object (CActor, CTerrain, etc.) and collect its polygons, transform them into world space, and add them to the two collision geometry arrays. This means we can integrate our collision system into our games with only a few simple calls to such registration functions. Once we have done this for each object we wish to register, our CCollision geometry database (m_CollVertices and m_CollTriangles) will contain every triangle we need to test against whenever a moving entity is updated. As we have seen, it is these two arrays that are looped through in the EllipsoidIntersectScene function. One at a time, each triangle is fed into the SphereIntersectTriangle function to determine if a collision occurred.

So far, so good. But the collision database currently has no capability to manage dynamic scene objects. Earlier in the course we spent a good deal of time covering the D3DX animation system and our CActor class fully supports it. If we register an actor with our collision system and that actor contains animation data, if we use its animation functions after we have registered the actor with the current collision database, we will be in for some real trouble. After all, as we just mentioned, those triangles are going to be converted to world space during collision registration and remain static.

Consider a scene that models the interior of a building. Animations might be triggered that make doors open and close, lifts go up and down, etc. As soon as an animation is played that moves the position of any of these meshes, the triangle data that the collision system has in its static arrays will be out of date and will no longer represent what the scene actually looks like to the player. Keep in mind that our collision and render geometry are separate things. So a door that opens for example, would visually appear to present a new path for the player to explore, but if it was registered with the collision system in an initially closed state, it would remain so and prevent the player from crossing the threshold (and vice versa if the door was registered in an open state).

Clearly, we need some way of letting the collision system know that some of its geometry is dynamic. As such, a new mechanism will have to be added to our collision system for such objects. Processing dynamic scene objects will place an even greater burden on our CPU (more so than testing collisions against our static geometry arrays), so we should only use the dynamic object system for meshes or hierarchies that we intend to animate. Geometry that will never change should still be added to the collision system as before for storage in the static arrays. What we will add is a secondary sub-system that will maintain a list of dynamic objects which the collision system needs to be kept informed about. This way it can track positional and/or rotational changes.

After reading this section, you should go straight to the workbook where we will finally implement everything we have discussed. This section of the textbook will focus primarily on the type of system design we need to think about and also discuss how to sweep a sphere against moving scene objects.

12.12.1 Sweeping a Sphere against Dynamic Objects

A dynamic object's data structure will contain the model space vertex and index data and a pointer to the object's world matrix. Because this is a matrix pointer, the collision system will always have access to the current object matrix, even when its contents are modified from outside the collision system by the application.

Our EllipsoidIntersectScene function must additionally perform the sphere/triangle tests against the geometry of each dynamic object. Nothing much has changed with respect to our intentions in the detection step -- we are still searching for the closest intersecting triangle. Whether that triangle exists in the static triangle array or in the triangle array of the one of the dynamic objects is for the most part irrelevant. What is quite different however is how we sweep the sphere against the dynamic objects. We can no longer just sweep the sphere as we did previously since the dynamic object is theoretically moving as well.

It would seem that we have to sweep the sphere along its velocity vector and the dynamic object along its velocity vector and then determine the intersection of the two swept volumes. Indeed that is one possible approach, but it is made complicated by the fact that the shape of the dynamic object might be something as simple as a cube or as complex as an automobile mesh. We know that we can sweep a sphere/ellipsoid easily (which is why it was used to bound our moving entity), but we hesitate to use crude approximations like this when sweeping the dynamic scene objects since we are after exact triangle intersections, not just the intersections of two swept bounding volumes.

Our system will take a different view of the problem. In fact, we will not have to sweep the dynamic scene object at all. What we will do is cheat a little bit by taking its movement into account and extending the swept sphere to compensate. That is, for each dynamic object we will maintain its current world matrix and its previous world matrix. Every time the application updates the position or orientation of a registered dynamic object, it will inform the collision system and the system will copy the currently stored matrix into the previous matrix variable. These two matrices will tell the collision system where the object is now and where it was in the previous collision update. By inverting the previous matrix and multiplying it with the current matrix we essentially end up with a relative movement matrix (we call this a *velocity matrix* in our system). This matrix describes just the change in position and orientation from the previous collision update to the current one for the dynamic object. We can use this information to grow the size of our swept sphere so that it compensates for how much the dynamic object has moved since the previous update. Since we have grown our sphere by the amount the object has moved from its previous position, we can now sweep it against the triangles of the dynamic scene object in its previous position. This once again reduces our intersection testing to a simple sphere/triangle test for each triangle in the dynamic object, and we already have functions written to do that.

Let us have a look at some diagrams to clarify this point. In Figure 12.40 we see our sphere about to be moved along its velocity vector. As we know, our collision system will use this information to create a swept sphere which can then be intersected against each of the triangles in the scene database. In Figure 12.40, we also see a blue cube, which we will assume is a dynamic object, which is currently stationary.

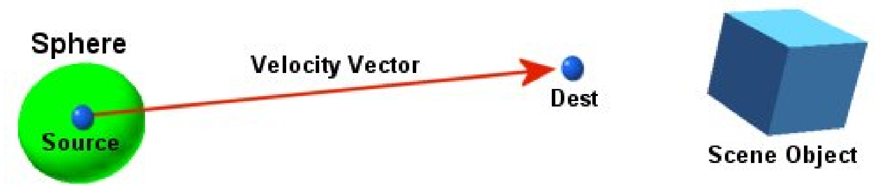


Figure 12.40

When this is the case, our job is no different then detecting collisions between our swept sphere and any other object that has been added to the static geometry database.

We know that if the dynamic object is not currently being treated as dynamic, then we can just use the SphereIntersectTriangle function to sweep the sphere against each triangle in its triangle array. Remember, each dynamic object will store its own array of triangles and vertices and therefore, even when not currently in motion, our collision system will have to loop through each dynamic object and test its triangles against the swept sphere. This is in addition to the triangle tests performed on the static geometry array. At the end of the day, we are just trying to find the closest intersecting triangle so that we can adjust the sphere's position so that it only moves along the velocity vector up to the time the sphere is in contact with the surface. This position and the corresponding slide plane normal are then returned back to the collision response phase.

In Figure 12.41, the sphere will not intersect the cube while moving along its current velocity vector. We know that in this case, the final position returned to the application for the

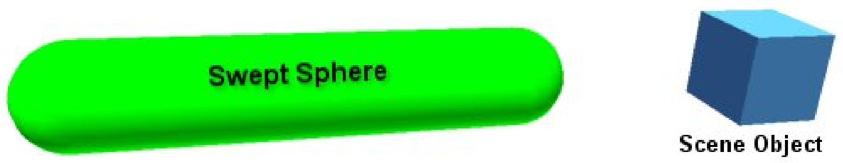


Figure 12.41

ellipsoid will be the original one that was requested (Sphere Center + Velocity Vector). This is shown as Dest in Figure 12.40.

Complications arise as soon as we start moving these dynamic objects around in our application. As discussed, each dynamic object will also maintain a pointer to the matrix that the application is using to position it in world space. This means our collision system always has access to the current position of any of its dynamic objects. Figure 12.42 depicts the problem we need to solve when the dynamic object is moving. In this example, the position the sphere wants to move to bypasses the position the dynamic object wants to move to. As such, a head on collision would occur somewhere along their respective paths.

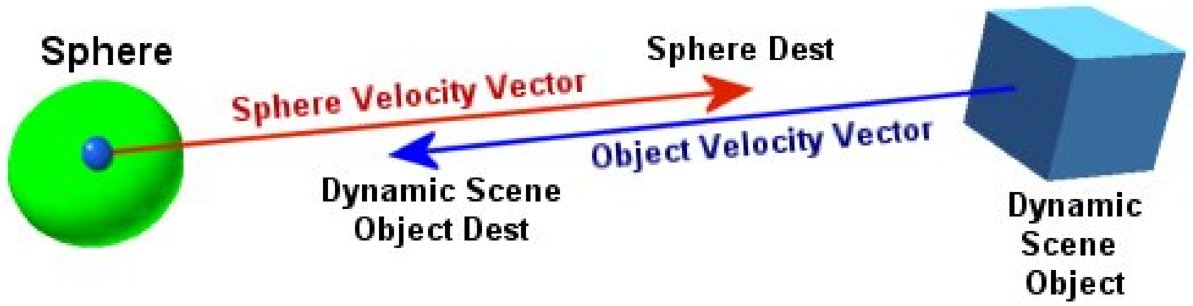


Figure 12.42

We can see clearly that there is a section of overlap between the two projected movement paths and a collision is definitely going to happen. The question is, what should our response be? Should the sphere and the cube stop at the point of intersection? Should they both bounce backwards as a result of impact?

With some caveats, in our implementation, we have decided to provide the dynamic scene objects with movement precedence. Since our collision system only has control over positioning the sphere, it was simplest just to give the dynamic scene object total right of way. However, this does not mean a user cannot implement complex responses, like allowing for various principles of physics to come into play. Our system does have a pointer to the object's world matrix and as such, it could overwrite the values in that matrix with anything it pleases. Of course, while this is certainly true, we would have to proceed very carefully.

Imagine for example a situation where we have registered an actor as a dynamic object with our collision system. The CCollision::AddActor function will have a Boolean parameter used to indicate whether we would like the meshes in the actor's frame hierarchy to be added to the collision system's static database or whether we would like to add each mesh in the hierarchy as a dynamic object. Obviously, if we intend to play animations on our actor, we will want it registered as set of dynamic objects (assuming multiple meshes). Now imagine that our cube in Figure 12.42 was one of many dynamic objects that were created from that single actor. The matrix pointer that the collision detection system stored would point directly at the combined frame matrix for the mesh. When an animation is played, since those animations are pre-recorded, we cannot simply move the cube backwards when an impact occurs. Even if our collision system were to alter the matrix of the dynamic object, the values it placed in there would be overwritten in the next frame update when the application advanced the animation time and the absolute frame matrices were all rebuilt. Therefore, only for a split second would these matrices, stored external to the collision system, contain the values placed in them by the collision system. As you can see, at least in this type of situation, we are going to have to forget about updating the matrices of the dynamic objects and update the position of the sphere instead.

This is not necessarily such a bad thing. Usually we want the animations that were recorded to play out the same way regardless of where the player may be. For example, we might load a scene which has animations that control large pieces of machinery in some part of the level. If the player was to walk into a swinging piece of the machinery, they should probably be knocked out of the way, leaving the machinery to carry out its animation unimpeded. This is also generally true when dealing with lifts. When the player is standing on the lift and animation is played which raises the lift up, we are relying on this behavior to make sure that the lift always moves the player up and does not bounce off the feet of the player when the two collide. Therefore, when the sphere collides with a moving object, we will move the sphere back so that its new position is simply contacting the object. It might even be the case that the new center position of the sphere is actually behind its original starting position if the sphere has been pushed backwards by a moving scene object (given some amount of force that might be applied).

To be fair, this approach is not always going to be the ideal way to handle every single scenario you and your design team might come up with. For very heavy objects like the machinery we just mentioned, this probably works well enough, but for objects with little mass or momentum, you might decide to go ahead and actually work out a way to modify the animation on the fly or take some other action. For example, you might decide to add a mechanism to the system which allows you to pass in a callback function at registration time. That callback could be triggered by the system when an object collides with the dynamic object. The callback might be passed information about the colliders such as their masses, linear and angular velocities, etc. You could then make some decisions based on the relative physical differences between the two colliders. If the dynamic object was a ceiling fan for example, you might decide to immediately stop the pre-recorded animation and take no action on the moving entity. Different situations will require different responses, but you get the idea. In our case, we will simply allow the dynamic objects to proceed with their animations and limit our response results to the moving entity only. This will make things much easier to implement and understand in the short tem.

Figure 12.43 illustrates this concept using the previous example. In Figure 12.42 we saw how the sphere and the cube wanted to move along paths that would cause a collision. In Figure 12.43 we see the result of such an impact. The cube is allowed to move from its previous position into its new position and the sphere is moved back so that it is now resting against the new position of the cube.

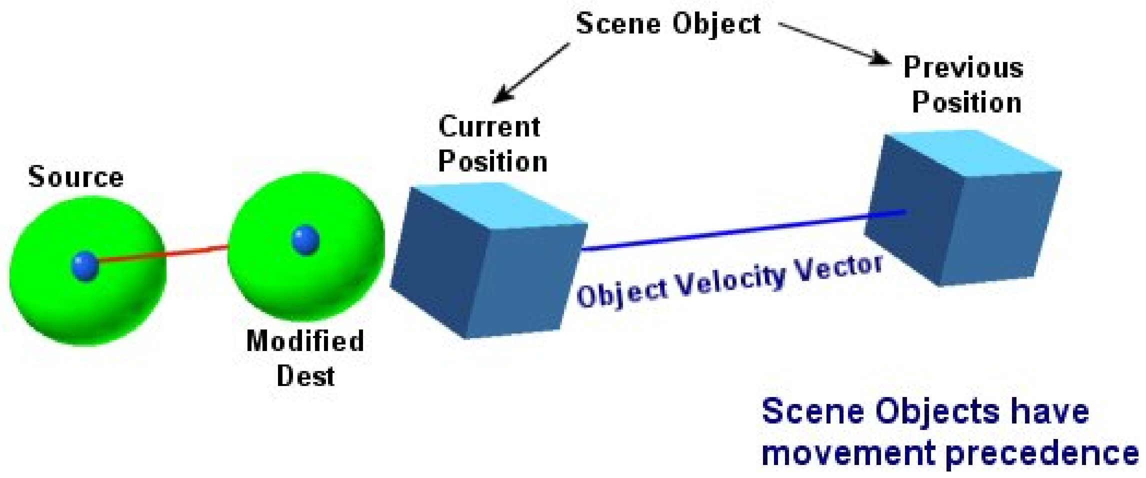


Figure 12.43

So we know how we want our ellipsoid to intersect with moving objects. But how do we find the final position for the sphere in the above situation? As discussed, the cube will have its matrix updated by the application. So what we have is the new cube position and the velocity we wish to move along. Now, you might assume that all we have to do it sweep the sphere against the cube in its current position. This would certainly seem to work in the above diagram and it would also mean that the test is once again reduced to a static test between the swept sphere and the triangles of the cube in its new position. However, imagine a situation where the cube was moving so fast, that while its start position was in front of the swept sphere (as in Figure 12.43), its final position is actually behind the swept sphere. In other words, with a single matrix update, the dynamic object has leapt from one side of the swept sphere to the other. Neither the previous or current positions of the cube intersect the swept volume, so our collision system would determine that the sphere could move to its desired location. The problem is that between these two frame updates, the cube has just passed straight through our sphere with no ill effect. So what should have happened instead? The sphere should have been knocked backwards by the moving cube (via some amount of applied force).

We discussed previously that a dynamic object will contain a pointer to its current application controlled matrix and we will cache its previous matrix as well. When we add a dynamic object to our collision system, a new DynamicObject structure will be allocated to contain this information along with some other data. If an actor is registered with the collision system, a DynamicObject structure will be created for every mesh in its hierarchy. Our DynamicObject structure will look like this:

Excerpt from CCollision.h

```
    struct DynamicObject
    {
        CollTriVector   *pCollTriangles;
        CollVertVector  *pCollVertices;
        D3DXMATRIX      *pCurrentMatrix;
        D3DXMATRIX       LastMatrix;
        D3DXMATRIX       VelocityMatrix;
        D3DXMATRIX       CollisionMatrix;
        long             ObjectSetIndex;
        bool             IsReference;
    };
```

CollTriVector *pCollTriangles
This is a pointer to an STL vector of CollTriangle structures. This vector contains one entry for each triangle in the mesh of the dynamic object. A CollTriangle contains the indices into the pCollVertices vector and the triangle normal.

CollVertVector *pCollVertices
This is a pointer to an STL vector of D3DXVECTOR3 structures. This vector will contain all of the *model space* vertex positions for the dynamic object's mesh.

D3DXMATRIX *pCurrentMatrix
When a dynamic object is added to the collision system via the CCollision::AddDynamicObject method, we must also pass in the address of the dynamic object's world matrix. This matrix will be owned and altered by the application. A pointer to this matrix is stored in the pCurrentMatrix member so that the collision system can always have immediate access to the matrix and any changes the application may have made to it. Our collision system will never write any values to this matrix; it will instead use it only to copy the values from this matrix into the dynamic object's LastMatrix member when the collision system is notified that the matrix has changed. If the dynamic object is part of a frame hierarchy, this member will point to the associated frame's combined matrix.

D3DXMATRIX LastMatrix
Every time the application alters the position of an object, it has to inform the collision system that the object has been updated and its matrices need recalculating inside the collision system. This is done via the CCollision::ObjectSetUpdated method. The pCurrentMatrix pointer will always point to the matrix of the object with the updated values, so we will use this matrix to copy the values into the LastMatrix member. Thus, the LastMatrix member will always contain the current world matrix of the object during collision tests.

D3DXMATRIX CollisionMatrix
When the ObjectSetUpdated function is called, the matrix that was previously the current position/orientation of the object is stored in LastMatrix. This is now about to be overwritten by the new values in the matrix pointed to by the pCurrentMatrix pointer. Before we do that, we copy the previous matrix into the CollisionMatrix member so that at the end of the update, LastMatrix contains the new current position of the object and CollisionMatrix holds the previous position of the object. The reason it is called the collision matrix and not something like 'PreviousMatrix' is something that will become clear shortly.

D3DXMATRIX VelocityMatrix
The velocity matrix is calculated for the dynamic object every time its position is updated by the application and the ObjectSetUpdated function is called. We need this matrix to describe the direction and magnitude the object has travelled from its previous position since the last collision frame update. That is to say, this matrix will describe the current position/orientation of the object relative to the previous position/orientation currently stored in Collision Matrix. How do we calculate it? We take the previous object world matrix and invert it. We then multiply this with the current object world matrix. Because we inverted the previous world matrix prior to the multiply, this has the effect of subtracting the position and orientation stored in the previous matrix from the position and orientation stored in the current matrix. The result is our velocity matrix which describes the new position and orientation of the

object relative to its previous one. You will see in a moment how we will use this matrix to stretch the swept sphere to compensate for the movement of the dynamic object from its previous position. This will then allow us to sweep this extended sphere against the triangles of the dynamic object in their previous positions (described in CollisionMatrix) using the usual static sphere/triangle intersection code we developed for our static scene geometry tests.

long ObjectSetIndex

Every time a dynamic object is registered with the collision system it will return an ObjectSetIndex. This is just a numeric value that uniquely identifies this object to the system. You can think of it as a way for the application to tell the collision system which object had its matrix updated. When adding a single dynamic object to the system, the dynamic object will be added to the collision system's dynamic object array and its internal dynamic object counter incremented. This will be returned as the object set index for the newly added object. The application should store this away so that it can always identify which object needs to be updated in the collision system. What we do in Lab Project 12.1 is add this as a new member in our CObject structure, which we have been using in one shape or form since the beginning of this course series. The updated CObject structure is shown below.

```
class CObject
{
public:
               CObject( CTriMesh * pMesh  );
               CObject( CActor   * pActor );
               CObject( );
    virtual    ~CObject( );

    D3DXMATRIX                m_mtxWorld;
    CTriMesh                 *m_pMesh;
    CActor                   *m_pActor;
    LPD3DXANIMATIONCONTROLLER m_pAnimController;
    long                      m_nObjectSetIndex;
};
```

Notice that even if the object contains an actor comprised of many meshes, only one object set ID is assigned by the collision system for all the meshes contain within. This is by design. When an actor is registered with the collision system, every mesh in the hierarchy will have a dynamic object created for it and added to the collision system's dynamic object database. However, every object created from the hierarchy will also be assigned the same object set index. This ID is used to identify a set of objects that have some relationship. To see why this is necessary, imagine the situation where you have registered an actor containing 50 meshes with the collision system. Then imagine that the application moves the entire actor to a new position in the world. When the actor's hierarchy is updated, all the absolute matrices in the hierarchy will be rebuilt to describe new world space positions of all 50 meshes contained within. The 50 dynamic objects in the collision database that we created from that hierarchy have just all had their current matrices changed. Because we have assigned every dynamic object in that hierarchy the same ID, we can call the ObjectSetUpdated function, passing in the actor's ID, and it will re-cache the matrices for every dynamic object with a matching ID (i.e., the 50 dynamic objects that were originally created from the hierarchy we just updated). Thus the object set ID does not just describe the unique ID of a single dynamic object, it can also represent a family of dynamic objects that will always need to be updated together.

bool IsReference
The final member of the DynamicObject structure is a boolean that indicates whether or not this dynamic object should be a reference. Referencing will be discussed in the workbook, but essentially this just provides a way for multiple dynamic objects to share the same triangle and vertex data.

We have now covered all the members of the DynamicObject structure. To understand the collision system, we first should understand how the application informs the collision system that a dynamic object has been updated. We will assume at this point that the objects have already been registered with the collision system and have been assigned object set index numbers. The registration functions are fairly simple and as such will be covered in the workbook. All we are interested in discussing here is how the dynamic system is going to work from a high level.

The CScene::AnimateObjects function is another familiar method. It is called from CGameApp::FrameAdvance function to give the scene class a chance to update the positions of any objects it wishes to animate. Since learning about animation, we have always used this function to advance the timer of each actor being used by the scene. Below we see the first part of the function from Lab Project 12.1. There is nothing new in this first section; it loops though each CObject and tests to see if an actor lives there. If not, then there is nothing to animate so it continues to the next iteration of the loop. If an actor does live there, then it attaches the CObject's animation controller to the actor.

Excerpt from CScene.h
```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;
        if ( !pActor ) continue;

        // If we are instancing, make sure that the cloned controller
        // is re-attached (no need to resynchronize the animation outputs
        // as we're about to call 'AdvanceTime'.)
        if ( pObject->m_pAnimController )
             pActor->AttachController( pObject->m_pAnimController, false );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false );
```

The next section is new.

If the object has a value in the m_nObjectSetIndex member of -1, then it means this object has not been registered with the collision system and we can simply continue on to process the next object. If this is not the case and we have a valid collision ID stored in m_nObjectSetIndex, then we know that the actor we are animating is also a dynamic object in the collision system and the system needs to be informed of the changes we have made to its position/orientation. We first set the actor world matrix so that all its absolute frame matrices get built, and then we call the CCollision::ObjectSetUpdated to inform the collision system about the event. The collision system can then grab the current state of the matrices for each dynamic object created from this hierarchy and calculate the collision and velocity matrices.

```
        // If the object has a collision object set index,
        // update the collision system.
        if ( pObject->m_nObjectSetIndex > -1 )
        {
            // Set world matrix and update combined frame matrices.
            pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );

            // Notify the collision system that this set of dynamic objects
            // positions, orientations or scale have been updated.
            m_Collision.ObjectSetUpdated( pObject->m_nObjectSetIndex );

        } // End if actor exists

    } // Next Object

}
```

Let us now take a look at the CCollision::ObjectSetUpdated function. Although we do yet understand how the collision detection is performed, looking at this function will show us how the collision matrix and velocity matrix are created every time the object is updated. It is important that we understand this process since these two matrices will be used by our EllipsoidIntersectScene function when sweeping the sphere against the triangles of dynamic objects.

This function is very small and it has a simple task to perform. It is passed an object set index which it will use to locate dynamic objects in the collision database that share the same ID. If you look in CCollision.h you will see that the CCollision class stores its dynamic object structures in an STL vector.

```
typedef std::vector<DynamicObject*> DynamicObjectVector;
DynamicObjectVector m_DynamicObjects;
```

As you can see, the member variable m_DynamicObjects is an STL vector that stores DynamicObject structures. Let us now have a look at the function itself.

Excerpt from CCollision.cpp

```
void CCollision::ObjectSetUpdated( long Index )
{
    D3DXMATRIX mtxInv;

    // Update dynamic object matrices
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Skip if this doesn't belong to the requested set.
        if ( pObject->ObjectSetIndex != Index ) continue;
```

In this first section of code we create an iterator to allow us to loop through and fetch each DynamicObject structure from the STL vector. We store a pointer to the dynamic object in the pObject local variable for each of access. We then check to see of this object's ObjectSetIndex is the same as the index passed into the function. If it is not, then it means the current object does not belong to the family of objects we are updating, so we continue on to test the rest of the list.

This next section of code is only executed if we have determined that the index of the current object matches the index passed in by the application. If this is the case, then the application has updated the matrix of the dynamic object and we need to do some matrix recalculation inside the collision system.

The first thing we do is access the matrix that is currently stored in the dynamic object's LastMatrix member. This matrix describes the world matrix of the object before the application recently updated it. Inverting it gives us a matrix that essentially subtracts this transformation from any matrix. Thus, if we multiply it with the dynamic object's current matrix, we essentially subtract the old position and orientation from the new one to leave us with a matrix that describes only the relative motion that has taken place since the last update. Remember, pCurrentMatrix is a pointer to the object's actual matrix and we have immediate access to it. We store the relative movement result in the VelocityMatrix member.

```
        // Generate the inverse of the previous frame's matrix
        D3DXMatrixInverse( &mtxInv, NULL, &pObject->LastMatrix );

        // 'Subtract' the last matrix from the current to give us the difference
        D3DXMatrixMultiply( &pObject->VelocityMatrix, &mtxInv,
                            pObject->pCurrentMatrix );
```

Next we copy over the previous matrix of the object from the LastMatrix member into the Collision Matrix member before we update the LastMatrix member with the new current position of the object. We now have everything we need. We have the velocity matrix calculated and the previous world matrix stored in CollisionMatrix which describes the world space position of the object that we will collide against. Further, we have updated LastMatrix with the current matrix so that this whole pattern can repeat again the next time UpdateObjectSet is called.

```
        // Store the collision space matrix
        pObject->CollisionMatrix = pObject->LastMatrix;

        // Update last matrix
        pObject->LastMatrix = *pObject->pCurrentMatrix;

    } // Next Object
}
```

Once again, please do not concern yourself at the moment with how the dynamic objects are created and how the scene registers the dynamic objects with the collision database. This is all pretty routine code which will be discussed in detail in the workbook. For now, all you need to know is that when our application calls the CollideEllipsoid method to move an object about the game world, the collision system will have a velocity matrix describing its movement from its previous position, and a matrix that describes this previous position (CollisionMatrix). This will be stored for every dynamic object.

In the EllipsoidIntersectScene function we now know that in addition to testing the swept sphere against the static scene database, we also have to test for collisions against the triangles of each dynamic object. We have at our disposal the previous transform state of each dynamic object (CollisionMatrix) and the velocity matrix that describes how much that state has changed since the last collision test. We now have to use this information to determine if the dynamic object intersected the swept sphere during its state change. If so, the position of the sphere will need to be adjusted to account for this collision.

Take a look at Figure 12.44. Source and Dest represent the amount the user wishes to move the ellipsoid (in eSpace). The red arrow represents the velocity vector for the ellipsoid (in eSpace) describing the distance and direction we wish to travel. Before testing each triangle of a dynamic object for collision, we will first transform it by the collision matrix so that the triangle is placed in the world at its previous position. Since this is a per-triangle process, we will describe it with respect to a single triangle for now.

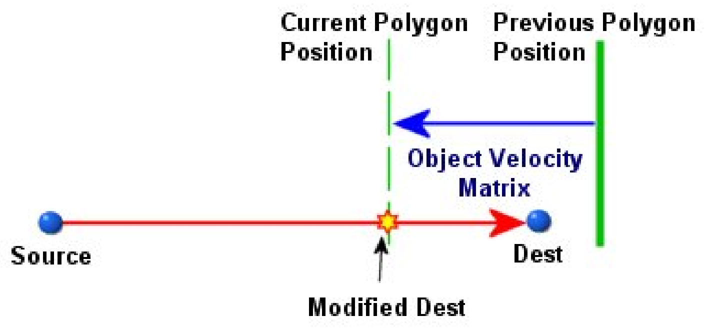


Figure 12.44

The solid green line illustrates the current triangle to be tested from the dynamic object. It is transformed into its previous position. The green dashed line shows where the triangle is in its current position. The blue arrow shows the distance the triangle has moved since the last collision test. Since we know that our triangle intersection routines will inflate the triangle by the radius of the sphere, we know the swept sphere can now be treated as a ray. This is the red line in Figure 12.44. Although the ray does not intersect the triangle in its previous position, it certainly does in its current position. We need to extend the ray such that it compensates for the movement of the triangle from its previous position to the current position. If we do this, we can simply test the ray against the triangle in its previous position. Once we have the intersection point, we can then shift it back by the distance the triangle has moved. This will give us the final intersection point (Modified Dest) along the ray such that the sphere is now resting against the polygon in its current position. More importantly, it will also catch the awkward case where the movement of the triangle was so large between frames, that it is completely bypassed by the ray. That is, we will develop a method that will always reliably catch the situation when the moving object intersects the sphere (even if the sphere is not moving).

The process we are about to discuss will be used to alter the swept sphere's velocity vector for a given dynamic object. Once we have discussed these intricacies, we will then examine the complete modified EllipsoidIntersectScene function with dynamic object support.

As the EllipsoidIntersectScene function is working with eSpace variables, we have to temporarily undo this to calculate the amount of shift we have to apply to the velocity.

```
vecEndPoint = (Vec3VecScale(eCenter, Radius) + Vec3VecScale(eVelocity, Radius));
```

All we are doing here is calculating Dest in the above diagram in world space. By scaling the eSpace sphere center position and the eSpace velocity vector by the radius of the ellipsoid, we scale these back into world space. We then add the world space velocity vector that was input into the function to the world space sphere center position to get the position that our ellipsoid would like to move to in the world (if no collisions block its path). We store this desired destination position in the local variable vecEndPoint. This is the world space position that the application (or the previous response step) would like to move the ellipsoid to.

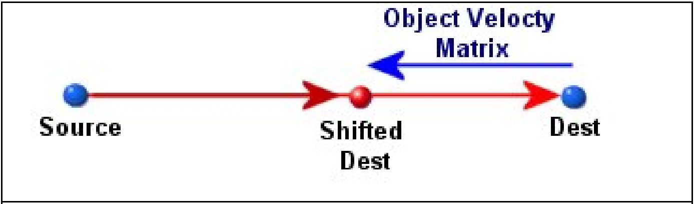


Figure 12.45

Our next step will be to take this world space destination position and transform it by the velocity matrix of the current dynamic object we are testing. In the case of our example shown in Figure 12.44, the velocity matrix describes the movement illustrated by the blue arrow. By transforming the destination position by this matrix we will shift the destination back along the ray. This shifted destination is illustrated in Figure 12.45. Notice how we are ultimately subtracting the movement of the object currently being tested from the end of our ray even though the position of the triangle in its previous location is past the end of the ray. This shifted destination position does not tell us anything quite yet, but bear with us. Below we show the code that shifts the ray end point back along the ray using the objects velocity matrix.

```
// Transform the end point
D3DXVec3TransformCoord( &eAdjust, &vecEndPoint, &pObject->VelocityMatrix );
```

At this point, Shifted Dest in Figure 12.45 is stored in the local variable eAdjust. We will now subtract the original end point of the ray from this shifted dest.

```
// Translate back so we have the difference
eAdjust -= vecEndPoint;
```

What we have a this point is a vector pointing in the opposite direction to the original velocity vector of our sphere whose length is equal to the distance moved by the dynamic object between its previous and current positions. We illustrate this in Figure 12.46. **Source** shows the original start point of the ray (the center of the ellipsoid) and **Shifted Dest** describes the position stored in eAdjust prior to the above line being executed. When we subtract the original ray end position from eAdjust we are subtracting the entire length of the original velocity vector from the shifted ray end point. This gives us an adjustment vector pointing in the opposite direction to the ray. This is shown in Figure 12.46 as the short blue arrow pointing from the ray origin (Source) to the left.

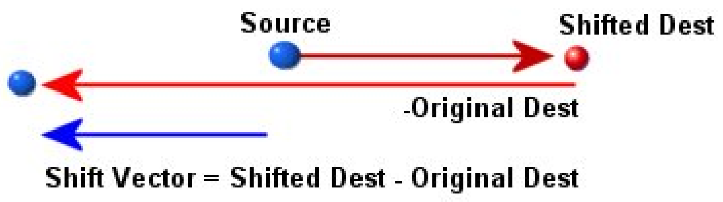


Figure 12.46

The next phase is the trick. We take this adjustment vector, transform it back into eSpace and then subtract it from the original velocity vector. Since the velocity vector and the adjustment vector are facing in opposite directions, subtracting the adjustment vector actually *adds it* to the original velocity vector. This extends the velocity vector of our swept sphere by the distance and direction the object has moved since the last collision test. Although we are illustrating just translational movement here this also works with rotations of the dynamic object.

```
// Scale back into ellipsoid space
eAdjust  = Vec3VecScale( eAdjust, InvRadius );

// Add to velocity
eVelocity-=eAdjust;
```

Figure 12.47 shows what our new velocity vector looks like. As we can see, it does indeed intersect the triangle in its previous position. We have altered the direction and length of velocity vector so that we can extrude it in the direction the object has moved since the last update and perform collision tests in its previous position.

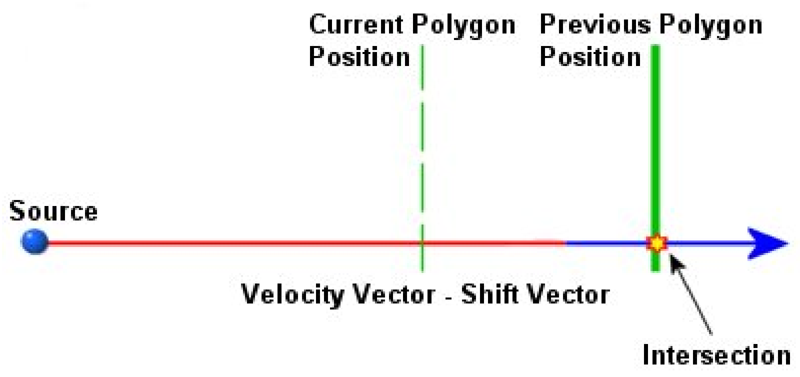


Figure 12.47

Now that we have the modified velocity vector, we can pass it into the SphereIntersectTriangle function for every triangle in the dynamic object. We do this for each dynamic object, searching for the intersection with the smallest interval. Once we have this new center position, what we actually have is the new center position of the unit sphere such that its surface is touching but not penetrating the closest intersecting triangle in the current dynamic object being tested. Of course, this is the position of the sphere resting against the triangle in its previous position and not its current position. All we have to do to correct this is take the resulting sphere center position and subtract from it the adjustment vector we initially added on. Remember, the adjustment vector is facing in the opposite direction from the velocity vector (in this example), so we actually add the adjustment vector to the resulting sphere center position to subtract it back.

```
// Translate back
Intersections[i].NewCenter      += eAdjust;
Intersections[i].IntersectPoint += eAdjust;
```

Above you can see that after we have added the new center position and the actual intersection point on the surface of the sphere to an intersection structure (to send back to the response step), we transform them back along the ray by the adjustment vector. Since this is the distance the triangle has moved from its previous position (the one we tested against), this shifts the sphere center position back so that it is now positioned on the shifted plane of the triangle in its current position instead of on the shifted plane of the triangle in its previous position. This gives us our final sphere center position (Modified Dest in Figure 12.47). As you can see, by calculating the adjustment vector, performing the intersection in the object's previous space, and then subtracting that adjustment vector from the intersection point, we essentially find out how much the sphere has to be moved back out of the way if a collision does occur.

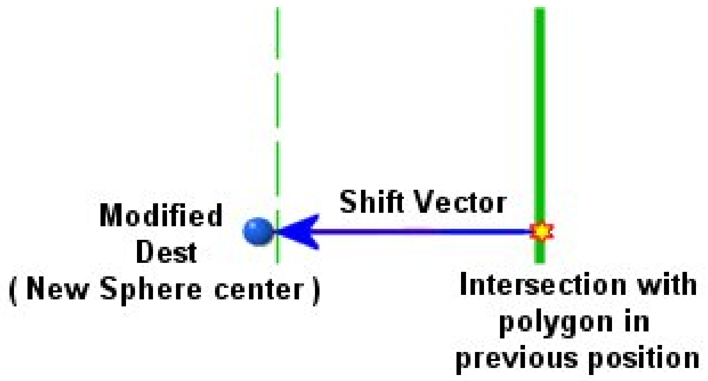


Figure 12.47

Once we have performed these same steps for every dynamic object and have found the closest intersecting triangle (or not), we then move on and test the original swept sphere against the static collision geometry using the methods explained earlier in the chapter. At the end of doing both the dynamic and static geometry tests, we will have a new ellipsoid position that we can return to the response step.

12.12.2 Integrating Dynamic Objects

In this final part of the textbook we will examine the code to the modified collision detection and response phases. The utility functions that add objects to the collision database and the way in which the application intersects with the collision system will all be discussed in the workbook. In this section, we will focus on updating the CollideEllipsoid and EllipsoidIntersectScene functions to handle both dynamic and static objects. Couple this knowledge with the fact that we have already discussed how the ObjectSetUpdated function works, and you should be able to understand everything in the workbook with very little trouble.

The first thing we will do is add a new generic function to our CCollision class that will remove the redundancy that would otherwise creep into the EllipsoidIntersectScene function now that we have many vertex and triangle buffers to process. The vertex and triangle data for each static triangle will all be stored in two STL vectors within the class namespace. These are the arrays that the EllipsoidIntersectScene function will need to extract geometry from when sweeping the sphere against the static scene geometry. However, each dynamic object will also have its own buffers of triangle and vertex data and each of these will have to be tested too. Therefore, instead of repeating this code in the EllipsoidIntersectScene function, we will write a generic function called EllipsoidIntersectBuffers. This function can then be passed the buffer of information that needs to be processed by the EllipsoidIntersectScene function. For example, we will call this function for each dynamic object in turn, passing in a pointer to its vertex and triangle buffers. The function will loop through each triangle in the buffer and call our SphereIntersectTriangle function. It will return the new position of the sphere based on the closest collision interval. The function can be called once for each dynamic object, each time being passed the vertex and triangle buffers of the dynamic object for testing. All the while it will be keeping track of the closest intersection. Once this function has been called to process the buffer for each dynamic object, it can be called one final time and passed the vertex and triangle buffers containing all the static collision geometry.

The CCollision class has many helper functions to make registering objects with the collision database fast and easy. Below, we will look at just the member variables in the CCollision class so you get a feel for how it all works. Note that all the structures we have discussed so far are defined in the class.

```
class CCollision
{
public:

    //-------------------------------------------------------------------------
    // Public, class specific, Typedefs, structures & enumerators
    //-------------------------------------------------------------------------
    struct CollTriangle
    {
        ULONG       Indices[3];           // Indices referencing the triangle
                                          // vertices from our vertex buffer
        D3DXVECTOR3 Normal;               // The cached triangle normal.
        USHORT      SurfaceMaterial;      // The surface material index
                                          // assigned to this triangle.
    };

    // Vectors (STL Arrays) to contain triangle and vertex collision structures.
    typedef std::vector<CollTriangle>   CollTriVector;
    typedef std::vector<D3DXVECTOR3>    CollVertVector;

    struct DynamicObject
    {
        CollTriVector  *pCollTriangles;   // The triangle data referencing
                                          // the vertices below.
        CollVertVector *pCollVertices;    // The vertices defining
                                          // the level geometry.
        D3DXMATRIX     *pCurrentMatrix;   // A pointer to the actual matrix,
                                          // updated by external sources
        D3DXMATRIX      LastMatrix;       // The matrix recorded on the latest test
        D3DXMATRIX      VelocityMatrix;   // The difference for this frame
                                          // between our two matrices.
        D3DXMATRIX      CollisionMatrix;  // The space in which the collision
                                          // against this object will be performed.
        long            ObjectSetIndex;   // The index of the set of objects
                                          // this belongs to.
        bool            IsReference;      // Is this a reference object?
    };

    // Vector of a type to contain dynamic objects
    typedef std::vector<DynamicObject*> DynamicObjectVector;

    struct CollIntersect
    {
        D3DXVECTOR3     NewCenter;
        D3DXVECTOR3     IntersectPoint;
        D3DXVECTOR3     IntersectNormal;
        float           Interval;
        ULONG           TriangleIndex;  // The index of the triangle
                                        // we are intersecting
        DynamicObject * pObject;        // If we hit a dynamic object,
                                        // it will be stored here.
    };

private:
    CollTriVector       m_CollTriangles;        // The triangle data referencing
                                                // the vertices below.
    CollVertVector      m_CollVertices;         // The vertices defining
                                                // the level geometry.
    DynamicObjectVector m_DynamicObjects;       // The dynamic object structures.
    USHORT              m_nMaxIntersections;    // The total number of
                                                // intersections we should record
    USHORT              m_nMaxIterations;       // The maximum number of collision
                                                // test iterations we bail
    CollIntersect      *m_pIntersections;       // Internal buffer for storing
                                                // intersection information
    D3DXMATRIX          m_mtxWorldTransform;    // When adding primitives, this
                                                // transform will be applied.
    long                m_nLastObjectSet;       // The last object set index used.
};
```

As you can see, the member variables are declared at the bottom. Let us quickly discuss them.

CollTriVector m_CollTriangles
CollVertVector m_CollVertices
These two STL vectors store the static triangle and vertex data. Every object that is registered with the collision system that is not flagged as a dynamic object will have its triangle and vertex data added to these two vectors. The first of the two shown above is a vector containing CollTriangle structures and the second contains the D3DXVECTOR3 structures. Feeding these two vectors into our EllipsoidIntersectBuffers function (which we will write in a moment) will test the swept sphere against every triangle in the static scene geometry database.

DynamicObjectVector m_DynamicObjects
This is an STL vector that contains DynamicObject structures. Every object that is registered with the collision system as a dynamic object will be added to this vector. Each dynamic object structure contains its own triangle and vertex arrays (STL vectors). When a frame hierarchy (an actor) is registered with the collision system as a dynamic object, a dynamic object structure will be created and added to this vector for every mesh in the frame hierarchy. Each dynamic object added in this way will belong to the same object set and will be assigned the same object set index number.

USHORT m_nMaxIntersections
CollIntersect *m_pIntersections
These two variables control the size of an array of CollIntersect structures that are used temporarily by the detection phase to return information back to the response step about the closest intersecting triangles.

USHORT m_nMaxIterations
This member contains the maximum number of iterations of detection/response we wish to execute before we settle on the closest intersection and forcibly spend the remaining velocity. There are occasions when the ellipsoid could get stuck in a sliding loop chewing up CPU cycles as it bounces between surfaces. When this number of iterations is reached, we admit defeat and settle for pushing the ellipsoid to the closest intersect point.

D3DXMATRIX m_mtxWorldTransform
This matrix will describe a world transform to the collision system that should be used to transform one or more model space static meshes we are registering. The system will use this matrix as it is filling its vertex array with data from that model space mesh. This can be useful when loading referenced mesh from a file for example. You should be mindful to set this back to its default state of identity after you have finished using it since all future objects you register will also be transformed by this same matrix (i.e., it is a system level matrix).

long m_nLastObjectSet
This variable contains the number of the last object set index that was issued to a dynamic object. Every time a new object set is added, this value is incremented and assigned to the object as a means of identification. This does not necessarily describe the number of dynamic objects currently registered with the collision system. It can also apply to groups of dynamic objects that should be updated together.

As you can see, for such a complex system, it really does not have many variables to manage. Let us now see the code that has been modified to facilitate dynamic object integration. Luckily, our core intersection routines are exactly the same. The SphereIntersectTriangle function and all the sub-intersection routines it calls are unchanged. From their point of view, we are still just sweeping the sphere against a triangle, even when dealing with dynamic objects. The functions that have changed are the CollideEllipsoid and the EllipsoidIntersectScene functions. We have added an additional helper function which we will look a first. It is called EllipsoidIntersectBuffers.

12.13 The EllipsoidIntersectBuffers Function

The EllipsoidIntersectBuffers function will handle the intersection determination between a swept sphere and an arbitrary buffer of triangle data. Most of this code will be instantly recognizable as it existed in the previous EllipsoidIntersectScene function that we covered earlier. All we have done is extract the inner workings of that function into a separate function that can be used to sweep the sphere against triangles in the static scene database and the triangle buffers of each dynamic object. This function will be called by the EllipsoidIntersectScene function, so let us first take a look at its parameter list. Most of these parameters will be familiar to us as they are simply the values that were passed into the EllipsoidIntersectScene function.

```
ULONG CCollision::EllipsoidIntersectBuffers( CollVertVector& VertBuffer,
                                             CollTriVector& TriBuffer,
                                             const D3DXVECTOR3& eCenter,
                                             const D3DXVECTOR3& Radius,
                                             const D3DXVECTOR3& InvRadius,
                                             const D3DXVECTOR3& eVelocity,
                                             float& eInterval,
                                             CollIntersect Intersections[],
                                             ULONG & IntersectionCount,
                                             D3DXMATRIX * pTransform /*= NULL*/ )
```

The first two parameters are new. They allow us to pass in the vertex and triangle buffers (STL vectors) we would like to sweep the sphere against. In our collision system, these parameters will be used either to pass the vertex and triangle buffers of the static scene geometry or the vertex and triangle buffers of a given dynamic object. The eCenter parameter is the current eSpace position of the ellipsoid that is about to be moved. The Radius and InvRadius vectors contain the radii and inverse radii of this ellipsoid. The InvRadius vector will be used to scale the vertex positions of each triangle into eSpace while the Radius vector will be used to scale the normal of each triangle into eSpace (remember the transform into eSpace works the opposite way for direction vectors). We also pass in the eSpace velocity vector describing the direction and distance we would like to move the sphere. We pass by reference the current *t* value of intersection. This will be passed to each of the intersection routines and collisions will only be considered if they have a smaller *t* value than what is currently stored. When the function returns, if a closer intersection was found than the one already stored in this variable, it will be overwritten with the new *t* value for the intersection.

As the eighth and ninth parameters we pass in the CollIntersect array which intersection results will be stored in and an output variable for a count of those items. As we have seen, this is the transport mechanism for returning the information back to the response phase. As discussed earlier, we must be mindful of the fact that this array may already hold intersection information. Recall that as soon as we find a closer intersection, we add this to the first element in the array and reset the counter to 1.

The final parameter is a pointer to a matrix that is used when processing the buffers of dynamic objects. The vertices in a dynamic object's vertex list are in model space, so before this function performs sphere/triangle tests on dynamic object geometry buffers, it must transform the triangle vertices into world space. When this function is called to perform collision testing against the static arrays, this parameter will be set to NULL. For dynamic object intersection testing, we will pass in the object's CollisionMatrix. As previously discussed, this is a world space transformation matrix that will transform the vertices of the dynamic object into their previous state.

Finally, this function will return an integer that describes the index into the intersection array where the intersection structures added by this function begin. For example, if this function was passed a CollIntersect array that contained five elements and another three were added, this function would return 5 to the caller and increment the intersection count to 8 (5 is the index of the first element in the array that was added by this function assuming zero-based offsets). If this function finds an intersection that is smaller than the interval values of the intersections currently stored in the array, the function will return 0 since the array will be completely overwritten from the beginning with the new closer intersection information.

Let us now have a look at this function a few sections at a time.

The first thing we do is store the current intersection count that was passed in (FirstIndex). This contains the current end of the list and the index where any additional structures will be added by this function, assuming that a closer intersection time is not found. It is the value of FirstIntersect that will be returned from this function. It will be set to zero later in the function if we determine a closer intersection and start building the list from the beginning again.

```
{
    D3DXVECTOR3 ePoints[3], eNormal;
    D3DXVECTOR3 eIntersectNormal, eNewCenter;
    ULONG       Counter, NewIndex, FirstIndex;
    bool        AddToList;

    // FirstIndex tracks the first item to be added the intersection list.
    FirstIndex = IntersectionCount;

    // Iterate through our triangle database
    CollTriVector::iterator Iterator = TriBuffer.begin();
    for ( Counter = 0; Iterator != TriBuffer.end(); ++Iterator, ++Counter )
    {
        // Get the triangle descriptor
        CollTriangle * pTriangle = &(*Iterator);
        if ( !pTriangle ) continue;

        // Requires transformation?
        if ( pTransform )
        {
            // Get transformed points
            D3DXVec3TransformCoord(&ePoints[0], &VertBuffer[pTriangle->Indices[0]],
                                   pTransform );
            D3DXVec3TransformCoord(&ePoints[1], &VertBuffer[pTriangle->Indices[1]],
                                   pTransform );
            D3DXVec3TransformCoord(&ePoints[2], &VertBuffer[pTriangle->Indices[2]],
                                   pTransform );

            // Get points and transform into ellipsoid space
            ePoints[0] = Vec3VecScale( ePoints[0], InvRadius );
            ePoints[1] = Vec3VecScale( ePoints[1], InvRadius );
            ePoints[2] = Vec3VecScale( ePoints[2], InvRadius );

            // Transform normal and normalize
            // (Note how we do not use InvRadius for the normal)
            D3DXVec3TransformNormal( &eNormal, &pTriangle->Normal, pTransform );
            eNormal = Vec3VecScale( eNormal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );

        } // End if requires transformation
```

Looking at the above code you can see that we create an iterator to access the elements of the STL vector and begin to loop through every triangle in the passed buffer. We then store a pointer to the current triangle in the local pTriangle pointer for ease of access. If for some reason this is not a valid triangle, we continue to the next iteration of the loop (a simple safety test).

We then test to see if the pTransform parameter is not NULL. If it is not, then it means that a matrix has been passed and the vertices of each triangle will need to be transformed by it prior to being intersection tested against the sphere. As you can see, we fetch the three vertices of the current triangle from the passed vertex buffer and transform them by this matrix. This code will only be executed if we are calling this function to process a dynamic object's geometry buffer. We store the transformed results of each world space vertex in the ePoints local array.

Next we scale each of the triangle vertices by the inverse radius of the ellipsoid, transforming these three vertices into eSpace. We then fetch the normal of the triangle and scale it from model space into world space using the passed transformation matrix (the CollisionMatrix of the dynamic object). Once we have the normal in world space, we scale it by the radius of the ellipsoid, transforming it into eSpace as well. We finally normalize the result.

If a valid transformation matrix was not passed, such as when processing the static geometry buffers, the triangles in those buffers are assumed to already be in world space and thus require only the transformation into eSpace.

```
        else
        {
            // Get points and transform into ellipsoid space
            ePoints[0] = Vec3VecScale( VertBuffer[ pTriangle->Indices[0] ],
                                       InvRadius );
            ePoints[1] = Vec3VecScale( VertBuffer[ pTriangle->Indices[1] ],
                                       InvRadius );
            ePoints[2] = Vec3VecScale( VertBuffer[ pTriangle->Indices[2] ],
                                       InvRadius );

            // Transform normal and normalize
            // (Note how we do not use InvRadius for the normal)
            eNormal = Vec3VecScale( pTriangle->Normal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );

        } // End if no transformation
```

At this point we have the triangle in eSpace, so we use our SphereIntersectTriangle function to perform the intersection test. If the function returns true, an intersection was found and we are returned the collision normal in eSpace (eIntersectNormal) and the new *t* value (eInterval).

```
        // Test for intersection between unit sphere and ellipsoid space triangle
        if ( SphereIntersectTriangle( eCenter, 1.0f, eVelocity,
                                      ePoints[0], ePoints[1], ePoints[2],
                                      eNormal, eInterval, eIntersectNormal ) )
        {
            // Calculate our new sphere center at the point of intersection
            if ( eInterval > 0 )
```

```
                eNewCenter = eCenter + (eVelocity * eInterval);
            else
                eNewCenter = eCenter - (eIntersectNormal * eInterval);
```

If the function returns true we calculate the new sphere center in eSpace. If eInterval is larger than zero then it means the intersection happened along the ray and the position can be calculated by scaling the velocity vector by the interval and adding the resulting vector to the original sphere center. If the collision interval is negative, then the sphere was embedded in the triangle and needs to be moved backwards such that this is no longer the case. Under these circumstances, the actual (negative) distance to the shifted plane is stored in eInterval and we can move the sphere center position back on to the shifted plane by moving along the plane normal by that distance.

In the next section of code we test to see if the interval is smaller than the intervals of the collision structures currently stored in the array. If so, it means we have found a triangle that intersects the ray closer to the origin than any currently stored in the array. When this is the case we set NewIndex to zero (which contains the index of the element in the array where the information about this intersection will be stored). As we have just invalidated all the information currently stored in the array, we store this in element zero and reset the intersection count back to 1. We also set FirstIndex to zero. Remember, this will be returned from the function as the index of the first structure added by this function to this array.

```
            // Where in the array should it go?
            AddToList = false;
            if ( IntersectionCount == 0 || eInterval < Intersections[0].Interval )
            {
                // We either have nothing in the array yet,
                // or the new intersection is closer to us
                AddToList         = true;
                NewIndex          = 0;
                IntersectionCount = 1;

                // Reset, we've cleared the list
                FirstIndex        = 0;

            } // End if overwrite existing intersections
```

If the interval is equal (with tolerance) to the intervals of the structures already stored in the array, we will add this new intersection information to the back of the array.

```
            else if ( fabsf( eInterval - Intersections[0].Interval ) < 1e-5f )
            {
                // It has the same interval as those in our list already, append to
                // the end unless we've already reached our limit
                if ( IntersectionCount < m_nMaxIntersections )
                {
                    AddToList         = true;
                    NewIndex          = IntersectionCount;
                    IntersectionCount++;

                } // End if we have room to store more

            } // End if the same interval
```

If the collision interval for the current triangle was either smaller or the same as the intervals currently stored, the AddToList boolean would have been set to true; otherwise, it will still be false, nothing will happen, and the following code block will be skipped. This is the code block that adds the collision information to the array (explained earlier in the EllipsoidIntersectScene discussion).

```
            // Add to the list?
            if ( AddToList )
            {
                Intersections[ NewIndex ].Interval = eInterval;
                Intersections[ NewIndex ].NewCenter = eNewCenter +
                                                (eIntersectNormal * 1e-3f);
                Intersections[ NewIndex ].IntersectPoint = eNewCenter -
                                                   eIntersectNormal;
                Intersections[ NewIndex ].IntersectNormal = eIntersectNormal;
                Intersections[ NewIndex ].TriangleIndex = Counter;
                Intersections[ NewIndex ].pObject = NULL;

            } // End if we are inserting in our list

        } // End if collided

    } // Next Triangle

    // Return hit.
    return FirstIndex;
}
```

12.14 The Revised EllipsoidIntersectScene Function

We will now examine the code to the modified EllipsoidIntersectScene function that adds support for dynamic objects and uses the previous function to perform the intersection with the various geometry buffers. This function is essentially the detection phase of our collision system. It is called by CollideEllipsoid for each movement of the ellipsoid along the initial or slide vectors. Our CollideEllipsoid function first converts the sphere center and velocity into eSpace before passing them in.

While the detection function will stay pretty constant as its job is fairly generic, the CollideEllipsoid function is very application specific. You may wish to add different response handling functions to the CCollision class to make the ellipsoid act in a different way. Further, you may not even be using ellipsoids at all; instead you might be using a sphere of arbitrary size to bound your objects. In such a case, you would not want to needlessly perform the eSpace transformation code. Additionally you might not wish to use the CCollision system's response phase at all, but instead allow your application to make direct calls to EllipsoidIntersectScene and handle you own responses. When this is the case, you would not want to have to concern your application with feeding in eSpace values for the ellipsoid center and velocity. Preferably, you could use the EllipsoidIntersectScene function as a black box that does this for you. The same is true for output. You are not likely to want to get back eSpace values for the intersection point, integration velocity, and the new sphere position. Ideally you would to feed EllipsoidIntersectScene world space values and get back world space values. So to make the function a little more generic, we will add two Boolean parameters that express whether the function is being passed eSpace input values and whether it should return eSpace values or perform the transformation back into world space prior to returning the information. Here is the new parameter list:

```
bool CCollision::EllipsoidIntersectScene(const D3DXVECTOR3& Center,
                                         const D3DXVECTOR3& Radius,
                                         const D3DXVECTOR3& Velocity,
                                         CollIntersect Intersections[],
                                         ULONG& IntersectionCount,
                                         bool bInputEllipsoidSpace /* = false */,
                                         bool bReturnEllipsoidSpace /* = false */ )
```

The final two parameters default to false, where the function will assume that it is being passed world space values and perform the transformation into eSpace using the passed radius vector. Because the final parameter is also set to false, this means the function will also convert the intersection information (stored in the passed CollIntersect array) back into world space prior to the function returning. While this makes the function a nice black box for third party calls, it should be noted that our CollideEllipsoid function (the function that calls this one) will transform the input values into eSpace and also requires eSpace values to be returned exactly as it did before. Therefore, when our CollideEllipsoid function calls this function, it will pass true for the final two parameters (the Center and Velocity vectors passed in will already be in eSpace).

The fourth and fifth parameters are the array and count of the CCollision object's CollIntersect array. It may seem strange to pass this array as a parameter to a CCollision function when it is already a class variable that the function would automatically have access to. We do this so that the caller can get back the results in their own array, should they wish not to use the complete collision detection and response system, but rather just the EllipsoidIntersectScene function.

Let us now talk about the new version of this function. The first thing we do is create the inverse radius vector that will be used to scale the passed center position and velocity into eSpace. If the bInputEllipsoidSpace parameter is set to false, the input variables are in world space and need to be transformed before we can continue. We copy the eSpace results into the local variables eCenter and eVelocity. If the bInputEllipsoidSpace parameter is set to true, the input variables are already in ellipsoid space and we simply copy them into the eCenter and eVelocity variables.

```
{
    D3DXVECTOR3 eCenter, eVelocity, eAdjust, vecEndPoint, InvRadius;
    float       eInterval;
    ULONG       i;

    // Calculate the reciprocal radius to prevent the many
    // divides we would need otherwise
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );

    // Convert the values specified into ellipsoid space if required
    if ( !bInputEllipsoidSpace )
    {
        eCenter   = Vec3VecScale( Center, InvRadius );
        eVelocity = Vec3VecScale( Velocity, InvRadius );
    } // End if the input values were not in ellipsoid space
    else
    {
        eCenter   = Center;
        eVelocity = Velocity;

    } // End if the input values are already in ellipsoid space
```

At this point we have our sphere center position and the velocity vector in eSpace. Next we set the eInterval variable to its maximum intersection interval of 1.0. This local variable will be fed into the EllipsoidIntersectBuffers function and will always contain the *t* value of the closest intersection we have found so far. When a closer intersection is found, the EllipsoidIntersectBuffers function will overwrite it. The initial value of 1.0 basically describes a *t* value at the end of the ray. If this is not altered by any of the collision tests, this means the velocity vector is free from obstruction. We also set the initial intersection count to zero as we have not identified any collisions yet.

```
    // Reset ellipsoid space interval to maximum
    eInterval = 1.0f;

    // Reset initial intersection count to 0 to save the caller having to do this.
    IntersectionCount = 0;
```

Now we will loop through each dynamic object in the m_DynamicObjects vector. For each one we will use its velocity vector to create the shift vector that is used to extend the length and direction of the velocity vector. This is the exact code we saw earlier when first discussing dynamic intersections. This process is also illustrated in Figures 12.43 through 12.47.

```
    // Iterate through our triangle database
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Calculate our adjustment vector in world space.
        // This is our velocity adjustment for objects
        // so we have to work in the original world space.
        vecEndPoint = ( Vec3VecScale( eCenter, Radius ) +
                        Vec3VecScale( eVelocity, Radius ));

        // Transform the end point
        D3DXVec3TransformCoord( &eAdjust, &vecEndPoint, &pObject->VelocityMatrix );

        // Translate back so we have the difference
        eAdjust -= vecEndPoint;

        // Scale back into ellipsoid space
        eAdjust  = Vec3VecScale( eAdjust, InvRadius );

        // Perform the ellipsoid intersect test against this object
        ULONG StartIntersection = EllipsoidIntersectBuffers(
                                              *pObject->pCollVertices,
                                              *pObject->pCollTriangles,
                                                eCenter,
                                                Radius,
                                                InvRadius,
                                                eVelocity - eAdjust,
                                                eInterval,
                                                Intersections,
                                                IntersectionCount,
                                                &pObject->CollisionMatrix );
```

Notice how once we have calculated the shift vector (eAdjust), we subtract it from the end of the velocity vector to extend the ray in the opposite direction of the dynamic object's movement. We do this while passing the velocity vector into the EllipsoidIntersectBuffers function as the sixth parameter. Note also that for the first two parameters we pass in the vertex and triangle arrays of the current dynamic object being processed for collision. The function will return the index of the first intersection structure that the call added to the CollIntersect array. Remember, this function will be called for each dynamic object and therefore, many different calls to the function may result in intersection information being cumulatively added to the CollIntersect array. If the function found no additional intersections, StartIntersect will be equal to IntersectionCount on function return.

Now it is time to loop through any structures that were added by the above function call and translate them back by the adjustment vector. By doing this little trick, we are sure to catch the sphere if it is between the previous and current position of the moving triangle. We can then translate the sphere back by the movement of the triangle so it is pushed out of the way by the dynamic object.

```
        // Loop through the intersections returned
        for ( i = StartIntersection; i < IntersectionCount; ++i )
        {
            // Move us to the correct point (including the objects velocity)
            // if we were not embedded.
            if ( Intersections[i].Interval > 0 )
            {
                // Translate back
                Intersections[i].NewCenter      += eAdjust;
                Intersections[i].IntersectPoint += eAdjust;

            } // End if not embedded

            // Store object
            Intersections[i].pObject = pObject;

        } // Next Intersection

    } // Next Dynamic Object
```

Notice that at the bottom of the dynamic object loop, if an intersection has occurred with the object's buffers, we also store a pointer to the dynamic object in each CollIntersect structure. This might be useful if the calling function would like to know more information about which object was actually hit.

With all the dynamic objects tested for intersection and the closest intersection recorded in eInterval, we now have to test the static scene geometry for intersection with the swept sphere. We do this by simply calling the EllipsoidIntersectBuffers function one more time. This time we pass in the vertex and triangle buffers containing the static scene geometry along with the CCollision object's m_CollVertices and m_CollTriangles members.

```
    // Perform the ellipsoid intersect test against our static scene
    EllipsoidIntersectBuffers(
                        m_CollVertices,
                        m_CollTriangles,
                        eCenter,
                        Radius,
                        InvRadius,
                        eVelocity,
                        eInterval,
                        Intersections,
                        IntersectionCount );
```

At this point, if any intersections have occurred with any geometry, the CollIntersect array will contain information about the closest intersection. The array may contain multiple entries if collisions occurred with multiple triangles at the exact same time. Either way, the first structure in this array will contain the new center position of the sphere. It is automatically returned to the response phase in CollideEllipsoid.

Before we return there is one final step that we will take. If bReturnEllipsoidSpace is set to false, then the function should transform all the eSpace values stored in the CollIntersect array into world space before returning. If this is the case, we loop through each intersection structure and apply the ellipsoid radius scale to make that so. Notice once again how we transform a positional vector from eSpace to world space by scaling it by the radii of the ellipsoid, but we transform a direction vector from eSpace to world space by scaling it by the inverse radii of the ellipsoid.

```
    // If we were requested to return the values in normal space
    // then we must take the values back out of ellipsoid space here
    if ( !bReturnEllipsoidSpace )
    {
        // For each intersection found
        for ( i = 0; i < IntersectionCount; ++i )
        {
            // Transform the new center position and intersection point
            Intersections[ i ].NewCenter =
                              Vec3VecScale( Intersections[ i ].NewCenter, Radius );
            Intersections[ i ].IntersectPoint =
                              Vec3VecScale( Intersections[ i ].IntersectPoint,
                                            Radius );

            // Transform the normal
            // (again we do this in the opposite way to a coordinate)
            D3DXVECTOR3 Normal =
                     Vec3VecScale( Intersections[ i ].IntersectNormal, InvRadius );
            D3DXVec3Normalize( &Normal, &Normal );

            // Store the transformed normal
            Intersections[ i ].IntersectNormal = Normal;

        } // Next Intersection

    } // End if !bReturnEllipsoidSpace
    // Return hit.
    return (IntersectionCount > 0);
}
```

The function simply returns true if the number of intersections found is greater than zero.

12.15 The Revised CollideEllipsoid Function

The CollideEllipsoid has also received a minor upgrade due to the integration of dynamic objects. The only major change involves the calculation of the integration vector that is returned back to the application.

As discussed earlier, the integration velocity vector parameter is an output parameter that will contain the new direction and speed of the object on function return. If no collisions have occurred, this will be a straight copy of the velocity vector passed in. However, while the input velocity vector is continually split at intersection planes and recalculated by projecting any remaining velocity past the point of intersection onto the slide plane, we do not wish the integration velocity to be diminished in this way. We should retain as much of its length as possible so that we can more accurately communicate the new direction and speed of the moving entity back to the application on function return.

Now we will have to take collisions with dynamic objects into account when calculating this vector. When the slide plane is on a dynamic object, we must once again calculate the adjustment vector in order to extend the integration velocity vector so that it can be projected onto the slide plane. However, remember that the intersection was performed between our velocity vector and the triangle of the dynamic object by adjusting the length of the ray to compensate for object movement. Since we now wish to project our integration velocity onto this same plane, we must also extend that vector in the same manner. Recall that we returned the intersection normal based on the dynamic object's previous position instead of its current one. The velocity matrix that describes the transition from the dynamic object's previous position to its current one may contain rotation as well. In short, we need to know that when we project the integration velocity vector onto the slide plane that the velocity vector is in the same *space* as the slide plane; otherwise the projection of the velocity vector onto this slide plane will be incorrect. This probably all sounds very complicated, but when you see the code you will realize that this is the exact same code we used to extend the velocity vector when testing for intersection against dynamic objects.

We have also added another feature to our CollideEllipsoid function. As we know, the CollIntersect array that is returned from the EllipsoidIntersectScene function contains the information about the closest collisions. If this array contains more than one element, then it means the ellipsoid intersected multiple geometry constructs at the same time. We also know that each CollIntersect structure contains within it, a vector describing the point of intersection on the surface of the ellipsoid. It is sometimes handy to have two vectors returned that describe the maximum and minimum collision extents on the surface of the ellipsoid. For example, our application will know that if the player is walking on the ground, then we should always have at least of one collision (the collision between the bottom of the ellipsoid and the floor polygon). We know that if the minimum y component of all the intersection points that occurred on the surface of the ellipsoid is very near to -Radius.y, the height of this point places it at the ground level with respect to the ellipsoid. Our application uses this to set a state that indicates that the player is on the ground and friction should be applied to diminish the velocity of the player.

This state is also tested during user input when the user presses the CTRL key to see whether the player should be allowed to jump -- a player that is in mid-air and currently falling should not be able to jump. Remember that an intersection point with a y component of zero would describe a point that is aligned with the center of the ellipsoid along the y axis. The top and bottom of the ellipsoid is described by +Radius.y and –Radius.y from the center of the ellipsoid, respectively.

Similarly, we know that if the y component of the lowest intersection point on the surface of the ellipsoid is zero, then the intersection point is aligned vertically with the center of the ellipsoid. This would mean that the player is currently in the air, and if gravity is being applied, the player will be falling. The collision point would also suggest to the application that an object has collided into a wall as the player is falling.

Finally, we know that when we have an intersection point nearing a y value of zero, the intersection point is aligned with the center of the sphere along the y axis. If you think about this for a moment, you will see that this means the player is hitting something almost head on, and its velocity should be set to zero. In fact, as we walk our ellipsoid into steeper inclines, we will find intersection points that are near the center of the ellipsoid vertically. Our application will use this information to slow the velocity of the player so that the steeper the hill, the slower the speed of the player as it ascends.

How these minimum and maximum collision extents are used by the application will be discussed in the workbook since it is not part of the collision detection system proper. However, the CollideEllipsoid function will record the minimum and maximum extents of the intersection points and return them via two output vectors that are passed as parameters.

Let us have a look at the new parameter list to this function. The only difference is that we have added these two new parameters to the end. These two vectors will be filled by the function with the minimum and maximum x, y, and z components of all intersection points. Thus these two vectors describe a bounding box within which all the intersection points are contained.

```
bool CCollision::CollideEllipsoid( const D3DXVECTOR3& Center,
                                   const D3DXVECTOR3& Radius,
                                   const D3DXVECTOR3& Velocity,
                                   D3DXVECTOR3& NewCenter,
                                   D3DXVECTOR3& NewIntegrationVelocity,
                                   D3DXVECTOR3& CollExtentsMin,
                                   D3DXVECTOR3& CollExtentsMax )
{
```

This is the function that is called by the application to calculate a movement update for an entity. The first parameter is the world space position of the center of the ellipsoid and the second is its radius vector. The Velocity vector describes the direction and distance the application would like to move the ellipsoid. On function return, NewCenter will describe the new world space position of the ellipsoid that is free from obstruction. It can be used by the application to update the position of the entity which the ellipsoid is approximating. On function return, the NewIntegrationVelocity parameter will contain the new direction and speed which the object should be heading after collisions and their responses have been taken into account. The final two parameters will contain two vectors describing the bounding box of intersection points.

We will look at the function a bit at a time, although most of this code will be familiar from the earlier version. First we add the passed velocity to the passed sphere center point to describe the world space position we would like to move the sphere to. We store the resulting position in the local variable vecOutputPos. We also take the passed velocity and store it in the local variable vecOutputVelocity. It is this output velocity that will contain the integration velocity we wish to return to the application. At the moment we set it to the current velocity because if no intersections occur, the velocity should not be changed.

```
    D3DXVECTOR3 vecOutputPos, vecOutputVelocity, InvRadius, vecNormal, vecExtents;
    D3DXVECTOR3 eVelocity, eInputVelocity, eFrom, eTo;
    D3DXVECTOR3 vecNewCenter, vecIntersectPoint;
    D3DXVECTOR3 vecFrom, vecEndPoint, vecAdjust;
    ULONG       IntersectionCount, i;
    float       fDistance, fDot;
    bool        bHit = false;

    vecOutputPos        = Center + Velocity;
    vecOutputVelocity   = Velocity;
```

Next we initialize the minimum and maximum extents vectors to the initial extremes. That is, we set the minimum vector to the maximum extents of the ellipsoid and set the maximum vector to the minimum extents of the ellipsoid. This will make sure that minimum extent starts as large as possible and will be set to the first intersection point we initially find, and vice versa. We have seen code like this when initializing bounding boxes in the past.

```
    // Default our intersection extents
    CollExtentsMin.x  = Radius.x;
    CollExtentsMin.y  = Radius.y;
    CollExtentsMin.z  = Radius.z;
    CollExtentsMax.x  = -Radius.x;
    CollExtentsMax.y  = -Radius.y;
    CollExtentsMax.z  = -Radius.z;
```

Now we create the inverse radius vector and scale the sphere center position and the velocity vector into eSpace. We store the current eSpace sphere position in eFrom, the eSpace velocity in eVelocity, and the eSpace desired destination position in eTo.

```
    // Store ellipsoid transformation values
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );

    // Calculate values in ellipsoid space
    eVelocity = Vec3VecScale( Velocity, InvRadius );
    eFrom     = Vec3VecScale( Center, InvRadius );
    eTo       = eFrom + eVelocity;
    // Store the input velocity for jump testing
    eInputVelocity = eVelocity;
```

Notice that we copy the eSpace velocity passed into the function into the eInputVelocity local variable. As discussed earlier, we may need this later if, after the maximum number of collision iterations, we still cannot resolve a final position. When this is the case, we will just use the initial velocity vector passed in to run the detection phase one last time to get a position which we will settle for (no sliding).

We now loop through the maximum number of iterations we wish to spend finding the correct position. Inside the loop, we see that if the current length of our velocity vector is zero (with tolerance) we have no more sliding left to do and can exit. Otherwise, we call the EllipsoidIntersectScene function to test for collisions along the current velocity vector. Notice how we pass true for the two boolean parameters, indicating that we will be passing in eSpace values and would like to have eSpace values returned.

```
    // Keep testing until we hit our max iteration limit
    for ( i = 0; i < m_nMaxIterations; ++i )
    {
        // Break out if our velocity is too small
        // (optional but sometimes beneficial)
        //if ( D3DXVec3Length( &eVelocity ) < 1e-5f ) break;

        if ( EllipsoidIntersectScene( eFrom, Radius, eVelocity,
                                m_pIntersections, IntersectionCount, true, true ) )
        {
            // Retrieve the first collision intersections
            CollIntersect & FirstIntersect = m_pIntersections[0];

            // Calculate the WORLD space sliding normal
            D3DXVec3Normalize( &vecNormal,
                               &Vec3VecScale( FirstIntersect.IntersectNormal,
                                              InvRadius )
                             );
```

If the function returns true then the sphere collided with something along the velocity vector and we should access the new position of the sphere from the first CollIntersect structure in the array. We also transform the returned eSpace collision normal into world space and normalize the result. We will need to do some work with this when projecting the integration vector.

In the next section of code we would usually just project the current integration velocity vector onto the slide plane, but if this collision has occurred with a dynamic object, then we have to calculate the force and direction at which the dynamic object struck the ellipsoid so that we can add that to the current integration velocity. As it happens, we already know how to do this. When we performed intersection tests between the sphere and triangles of a dynamic object, you will recall how we create an adjustment vector that allows us to extend the ray and collide with the previous position of the object. This adjustment vector described the direction the object has moved since the previous update. If we calculate this adjustment vector in the same way, we have a vector that describes the speed and distance of movement the dynamic object has moved as shown below.

```
            // Did we collide with a dynamic object
            if ( FirstIntersect.pObject )
            {
                DynamicObject *pObject = FirstIntersect.pObject;

                // Calculate our adjustment vector in world space. This is for our
                // velocity adjustment for objects so we have to work in the
                // original world space.
                vecFrom     = Vec3VecScale( eFrom, Radius );
                vecEndPoint = vecFrom + vecOutputVelocity;

                // Transform the end point
                D3DXVec3TransformCoord( &vecAdjust, &vecEndPoint,
                                        &pObject->VelocityMatrix );

                // Translate back so we only get the distance.
                 vecAdjust -= vecEndPoint;
```

The code above is not new to us -- we created a shift vector (eAdjust) in exactly the same way when performing the intersection tests. The adjustment vector describes the movement of the dynamic object from its previous state to its current state; the period in which the collision with the sphere happened. This vector tells us the speed which the dynamic object was traveling from its previous position to its new position and we can use this to approximate an applied force on our sphere as a result of the impact. After we have projected the current integration velocity vector onto the slide plane, we have a new integration vector facing in the slide direction:

```
                // Flatten out to slide velocity
                fDistance = D3DXVec3Dot( &vecOutputVelocity, &vecNormal );
                vecOutputVelocity -= vecNormal * fDistance;
```

We will then calculate the angle between the adjustment vector and the slide plane normal. Since the slide plane normal describes the orientation of the point of impact on the triangle, and the adjustment vector describes the speed and direction of the movement of the triangle when it hit the sphere, performing the dot product between them will give us with a distance value that gets larger the more head-on the collision between the two objects. We will use this distance value to add a force to the integration velocity that pushes the object away from the triangle in the direction of the triangle normal.

```
                // Apply the object's momentum to our velocity along the
                // intersection normal
                fDot = D3DXVec3Dot( &vecAdjust, &vecNormal );
                vecOutputVelocity += vecNormal * fDot;

            } // End if colliding with dynamic object
```

Remember, even before we calculate the integration vector, the sphere has been correctly repositioned so that it has moved back with the colliding triangle and thus is not intersecting. By adding this force to the integration velocity that is returned to the application, it means that our application will continue to move the sphere back away from the point of impact for the next several frames. This is certainly not state of the art physics here, but it certainly provides a convincing enough effect to be suitable for our purposes in this demo.

The next code block is executed if the intersection information does not relate to a dynamic object. In this case we just project the current integration velocity onto the side plane, just as we did in the previous version of the function.

```
            else
            {
                // Generate slide velocity
                fDistance = D3DXVec3Dot( &vecOutputVelocity, &vecNormal );
                vecOutputVelocity -= vecNormal * fDistance;

            } // End if colliding with static scene
```

We have now modified the integration velocity vector and accounted for collisions with a static or scene triangle. The application can now be informed of the new direction the object should be heading when the function returns.

In the next section of code we grab the new eSpace position of the sphere and store it in the eFrom local variable. This is the new position of the sphere as suggested by the collision detection phase. We then calculate the slide vector that will become the velocity vector passed into the EllipsoidIntersectScene function in the next iteration of the collision loop. We subtract the actual position we were allowed to move to from the previous desired destination position which provides us with the remaining velocity vector after the point of intersection. This is then dotted with the plane normal, giving us the distance from the original desired destination position to the slide plane. We then subtract this distance from the original intended destination along the direction of the plane normal, thus flattening the remaining velocity onto the slide plane. This is stored in the eTo local variable which will be used in a moment to calculate the new slide velocity that will be one of the inputs for the next collision detection loop.

```
            // Set the sphere position to the collision position
            // for the next iteration of testing
            eFrom = FirstIntersect.NewCenter;

            // Project the end of the velocity vector onto the collision plane
            // (at the sphere centre)
            fDistance = D3DXVec3Dot( &( eTo - FirstIntersect.NewCenter ),
                                      &FirstIntersect.IntersectNormal );
            eTo -= FirstIntersect.IntersectNormal * fDistance;
```

We will now get the new sphere position and the intersection point on the surface of the sphere and transform them into world space. This will be done for the purpose of testing the intersection point against the minimum and maximum extents currently recorded. We adjust the relevant extents vector if the intersection point is found to outside the bounding box (i.e., we will grow the box to fit). To calculate the new world space sphere center position, we just transform the new eSpace sphere position returned from the collision detection function using the ellipsoid radius vector. To calculate the intersection point on the surface of the ellipsoid we simply scale the slide plane normal into world space by the radius vector and subtract the world space sphere position from it.

```
            // Transform the sphere position back into world space,
            // and recalculate the intersect point
            vecNewCenter       = Vec3VecScale( FirstIntersect.NewCenter, Radius );
            vecIntersectPoint  = vecNewCenter - Vec3VecScale( vecNormal, Radius );
```

If the calculation of the intersection point is bothering you, remember that in eSpace the sphere has a radius of one. Since a unit vector also has a length of one we know that in eSpace, the intersection point on the sphere can be found simply by subtracting the slide plane normal from the sphere center position. This will always generate a vector at a distance of 1.0 unit from the sphere center in the direction of the intersection point. Because we are transforming the unit normal by the radius vector, we also scale its length as well. The resulting vector will have a length equal to the world space distance from the center of the ellipsoid to the point on the surface of the ellipsoid. If we subtract this from the world space ellipsoid position, we have the intersection point in world space on the surface of the ellipsoid.

In the next section of code we subtract the ellipsoid center position to leave us with an intersection point that is relative to its center. We can now test the x, y, and z components of the intersection point against the corresponding components in the extents vectors and grow those vectors if we find a component of the intersection point vector that is outside the box. The result is a bounding box where there ellipsoid center is assumed to be at its center.

```
            // Calculate the min/max collision extents around the ellipsoid center
            vecExtents = vecIntersectPoint - vecNewCenter;
            if ( vecExtents.x > CollExtentsMax.x ) CollExtentsMax.x = vecExtents.x;
            if ( vecExtents.y > CollExtentsMax.y ) CollExtentsMax.y = vecExtents.y;
            if ( vecExtents.z > CollExtentsMax.z ) CollExtentsMax.z = vecExtents.z;
            if ( vecExtents.x < CollExtentsMin.x ) CollExtentsMin.x = vecExtents.x;
            if ( vecExtents.y < CollExtentsMin.y ) CollExtentsMin.y = vecExtents.y;
            if ( vecExtents.z < CollExtentsMin.z ) CollExtentsMin.z = vecExtents.z;
```

Next we create the slide vector that will become the velocity vector for the next iteration of the collision detection phase. In the eTo vector, we currently have the original desired destination point projected onto the slide plane. This describes the location we want to travel to in the next iteration. In eFrom we also have the new non-interpenetrating position of the ellipsoid returned from the previous call to EllipsoidIntersectScene. Therefore, by subtracting eFrom from eTo, we have our new velocity vector.

```
            // Update the velocity value
            eVelocity = eTo - eFrom;

            // We hit something
            bHit = true;

            // Filter 'Impulse' jumps (described in earlier version of function)
            if ( D3DXVec3Dot( &eVelocity, &eInputVelocity ) < 0 )
            { eTo = eFrom; break; }

        } // End if we got some intersections
        else
        {
            // We found no collisions, so break out of the loop
            break;

        } // End if no collision

    } // Next Iteration
```

Above we see the final section of code in the inner loop. Hopefully, we will have found a collision and broken from the above loop before the maximum number of iterations. Notice how we use the additional dot product to filter out back and forth vibration. Now you see why we made that additional copy of the initial velocity vector that was input into the function. We would need it later to test if we ever generate a slide vector in the response phase that is bouncing the ellipsoid back in the opposite direction than was initially intended. If we find this is the case, we set eTo equal to eFrom, making the slide vector zero length so that no more sliding is performed.

The next section of code is executed if an intersection occurred. If at any point in the detection loop above EllipsoidIntersectScene returned true, the local boolean variable bHit will have been set to true. Inside this conditional is another one that tests whether the number of loop iterations we executed is equal to the maximum number of iterations. If not, then it means we successfully resolved the response to our satisfaction and simply have to scale the new position of the sphere into world space.

```
    // Did we register any intersection at all?
    if ( bHit )
    {
        // Did we finish neatly or not?
        if ( i < m_nMaxIterations )
        {
            // Return our final position in world space
            vecOutputPos = Vec3VecScale( eTo, Radius );

        } // End if in clear space
```

However, if the loop variable *i* is equal to the maximum number of iterations allowed, it means we could not find a satisfactory position to slide to in the given number of iterations. When this is the case we simply call EllipsoidIntersectScene again with the original input values to get the closest non-intersecting position and store this in vecOutputPos after transforming it into world space. As you can see, if this section of code is executed, we have encountered the undesirable situation where the response step was repeatedly bouncing the sphere between two triangles. When this is the case, we forget about the slide step and just move the sphere along the velocity vector as much as possible and discard any remaining velocity.

```
        else
        {
            // Just find the closest intersection
            eFrom = Vec3VecScale( Center, InvRadius );

            // Attempt to intersect the scene
            IntersectionCount = 0;
            if ( EllipsoidIntersectScene( eFrom, Radius, eInputVelocity,
                                          m_pIntersections, IntersectionCount,
                                          true, true ) )
            {
                // Retrieve the intersection point in clear space,
                // but ensure that we undo the epsilon
                // shift we apply during the above call.
                // This ensures that when we are stuck between two
                // planes, we don't slowly push our way through.
                vecOutputPos = m_pIntersections[0].NewCenter -
```

```
                                   (m_pIntersections[0].IntersectNormal * 1e-3f);

                // Scale back into world space
                vecOutputPos = Vec3VecScale( vecOutputPos, Radius );

            } // End if collision
            else
            {
                // Don't move at all, stay where we were
                vecOutputPos = Center;
            } // End if no collision

        } // End if bad situation

    } // End if intersection found

    // Store the resulting output values
    NewCenter              = vecOutputPos;
    NewIntegrationVelocity = vecOutputVelocity;

    // Return hit code
    return bHit;
}
```

The final few lines of code simply store the new world space ellipsoid position in the NewCenter variable and store the integration velocity in the NewIntegrationVelocity vector. Since both of these vectors were passed by reference, the application will be able to access these values when the function returns. We conclude by returning the value of bHit, which will be set to either true or false depending on whether EllipsoidIntersectScene call ever returned true.

Conclusion

This has probably been a very challenging lesson for many of you as it was certainly one of our most complicated and most mathematically intense systems to date in the series. Do not be concerned if you find that you need to re-read parts of this chapter again in order to fully grasp what is happening. It is recommended that you take your time with the material and try to keep the actual source code in hand.

On the bright side, we have developed a collision system that can be easily plugged into our future applications. We will expand the coverage of our CCollision class in the accompanying workbook, where the changes to the application and the CPlayer class will be discussed. We will also discuss all of the CCollision utility member functions, such as those responsible for registering terrains, dynamic objects, and static meshes with the collision database.

The core runtime query functions of the collision system have all been discussed in this chapter, so they will not be covered again. The final EllipsoidIntersectScene, EllipsoidIntersectBuffers and CollideEllipsoid functions just examined are almost identical to the ones used by the CCollision class in Lab Project 12.1. In the lab project we add a few extra lines to efficiently deal with terrains, but this will be explained in the workbook. The SphereIntersectTriangle, SphereIntersectPlane, SphereIntersectPoint, and SphereIntersectLineSegment intersection routines (called from SphereIntersectTriangle) are also the exact implementations used in Lab Project 12.1. With all the intersection routines covered, this leaves us with a workbook that can concentrate on how geometry is registered with the system and how the collision system is used by our framework.

Your next course of action should be to read the workbook that accompanies this chapter so that you will finally have a full understanding of every aspect of the collision system. As we will be using this collision system in all future lab projects, it is a good idea to make sure you understand it very well.