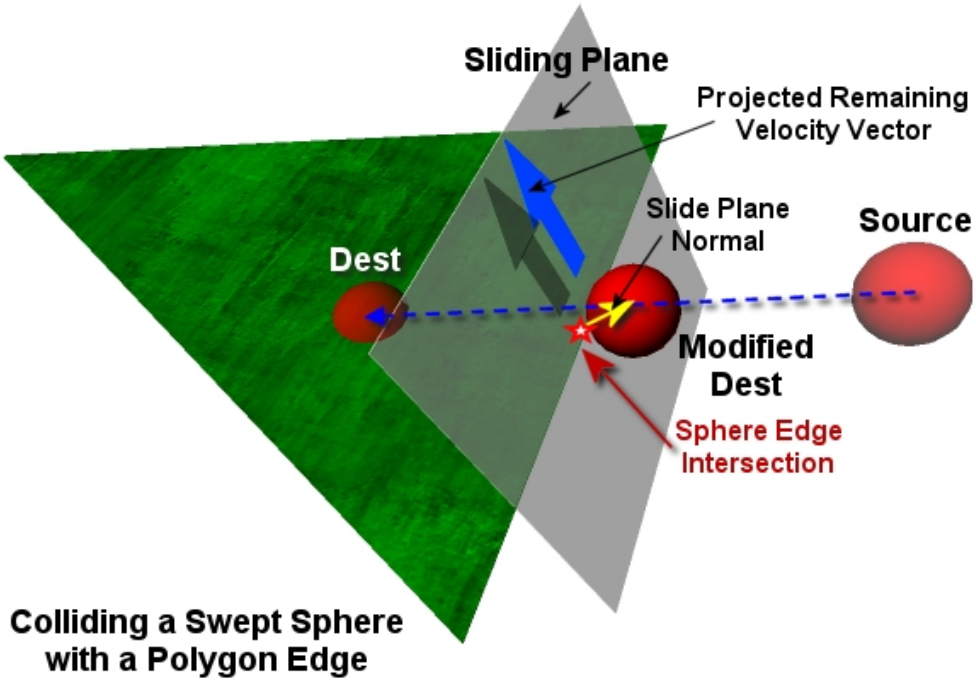
Workbook Thirteen:
Collision Detection and Response



Lab Project 13.1: The CollisionSystem

In this workbook we will discuss the remaining code in Lab Project 13.1 which was not discussed in the accompanying textbook. It is very important that you read the accompanying textbook before reading this workbook because the intersection routines in the CCollision class will not be discussed again in detail here.

This workbook will focus on the utility functions exposed by the CCollision class which allow an application to easily register various types of static and dynamic objects with the collision system. We will discuss the functions that allow the application to register static meshes with the collision geometry database as well as how to register complete frame hierarchies (contained in a CActor) with the collision system. Furthermore, we will cover the functions that allow the application to register single mesh constructs and entire frame hierarchies as dynamic scene collision objects. This will allow an application to register fully animated actors with the collision system such that collision determination and response between moving entities and animated scene geometry can happen in real time.

Figure 13.1

We will also add a special method tailored for the registration of terrain data that will help keep the memory overhead of the collision geometry database minimized. This will ensure that our CTerrain objects too can benefit from the collision system. The process will involve our collision system building triangle data on the fly using the terrains height map whenever that terrain is queried for collision. This allows us to completely eliminate the need for us to make a copy of every triangle comprising the terrain in order to add it to the static geometry database. Whenever a terrain is queried for intersection, the swept ellipsoid will be transformed into the image space of the terrain's height map. The starting location and the desired destination location of the ellipsoid will then be used to construct an image space bounding box on the height map describing the region of the terrain that could possibly be collided with in this update. This will then be used to build temporary vertex and triangle lists containing only triangles in the area of the terrain that fall within that box (see Figure 13.1). These buffers will then be tested for intersection using our standard tests. The end result is that our collision system will provide detection and response for our CTerrain objects without the need to store copies of every terrain triangle. This makes collision detection against height map based terrains very memory efficient as only a handful of triangles will need to be temporarily stored, and only for the lifetime of the collision test. These triangles will then be discarded.

Our collision system will also be extended to deal with the registration of referenced actors. This means that dynamic objects inside the collision system will not always have their own vertex and triangle lists. We may, for example, have a frame hierarchy loaded into a CActor object that contains 20 meshes. This actor might even be used to model a geometric construct that appears many times in our scene; a street lamp for example. We already know that from a rendering perspective, rather than having to load 20 separate actors into memory, all with the same geometric detail, we can just create 20 CObjects that all point to the same actor but have different world matrices (and possibly animation controllers). As long as we inform the actor of the objects world matrix so that it can update itself before it is rendered, we can have just one actor in memory and render it from in 20 different places in our scene.

We also took this concept a step further earlier in the course when we added support for referenced *animated* actors. To do this we expanded the CObject structure to store both a pointer to the actor it is referencing and a pointer to the animation controller for that reference. Before we render the object, we pass its world matrix into the actor so that it can update all its frame matrices correctly for that reference. We also attach the object's controller to the actor. Attaching the instance controller allowed the actor to update its frame matrices so that they reflect the positions and orientations described by the animation controller for that reference.

We have to consider this idea with respect to our collision database. When we add an actor to our collision database, the CCollision class will traverse the hierarchy and add a new dynamic object to the internal list of dynamic objects for every mesh container found in the hierarchy. For each mesh container we find, we will copy over its vertex and index data into a new dynamic object. Clearly, if we have a single actor that contained 10 meshes and we would like to place it in the scene 100 times in different world space locations, we would not want to create 1000 dynamic objects in our collision system (100*10) that all had their own copies of the same vertex data. To address this, we will allow an actor to be added as a reference to an actor that has previously been registered with the collision system. We know from the textbook discussion that when an actor is registered with the collision system, every mesh in that hierarchy will have a dynamic object created for it and each will be assigned the same object set index. Using the previous example of a 10 mesh hierarchy, when the actor is added for the first time, 10 dynamic objects would be created and added to the collision database. However, each of these dynamic objects would be assigned the same object set ID. This is how the collision system knows that all these objects belong to the same hierarchy or group and need to be updated together.

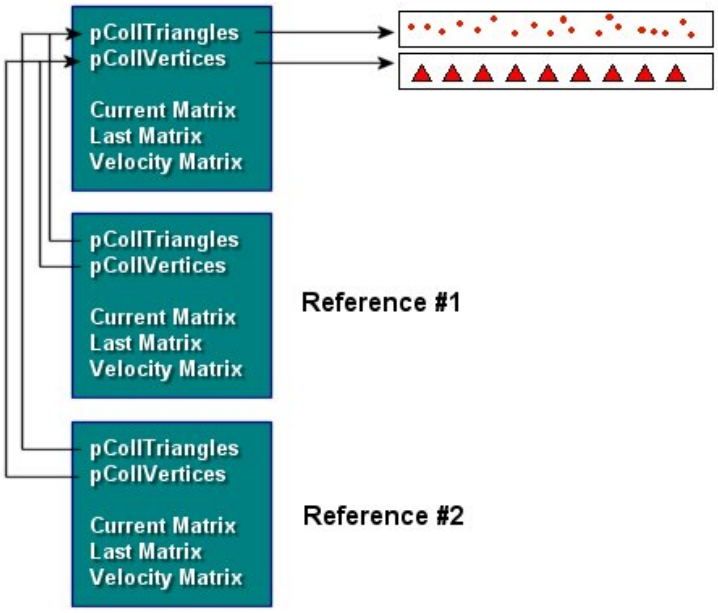


Figure 13.2

The next time we wish to register the same actor with the collision system we can use the CCollision::AddActorReference function. This function will be passed the object set index that was returned by the collision system when the initial actor was added. This function will then loop through the collision systems dynamic object array searching for all dynamic objects that have a matching object set index. It knows that each one it finds belongs to the original actor that is now being referenced. As such, it can just make a new copy of each dynamic object and have the vertex and triangle buffer pointers point to the original dynamic objects buffers.

For example, if we initially add an actor that contains 20 meshes to the collision system, 20 dynamic objects will be created. Each will contain actual vertex and index information from the original meshes in the hierarchy that was registered with the collision system. The registration of this actor would cause

the collision system to issue an object set index back to the application that will have been assigned to each dynamic object created from that actor. If the application subsequently passed this object set index into the CCollision::AddActorReference method, the 20 dynamic objects would be duplicated such that there would now be 40 dynamic objects registered with the collision system. However, the vertex and triangle buffers of the second 20 objects would be pointers to the buffers of the original 20 objects from which they were cloned. Remember from the textbook that a non-referenced dynamic object will contain its geometry data in model space. Since each dynamic object (normal or referenced) contains its own world matrix, the collision system has everything it needs to take that geometry and transform it into world space prior to performing any intersection tests on that dynamic object

We will also examine how the CScene IWF loading functions (e.g., ProcessMeshes and ProcessReference) have been altered so that they can register each mesh, actor, and terrain they create with the collision system. The CScene class owns the CCollision object as a member variable.

Finally, we will discuss how the application (and especially the CPlayer object) uses the collision system in a real-time situation. We will add a better physics model to the movement of our CPlayer so that now concepts like friction and gravity are factored when computing its velocity. The velocity of the CPlayer will ultimately be controlled by how much force is being applied to it. The CGameApp::ProcessInput function will also be altered slightly so that instead of moving the player directly in response to key presses, these now cause an application of force to the player. We then let our simple physics model determine the velocity of the player based on the force being applied, taking into account other factors such as resistant forces (aerodynamic drag, for example).

In this workbook will cover the following:

- Adding geometry registration functions to CCollision
- Adding referenced data to the collision system
- Adding memory efficient terrain registration and collision testing functions for height-map based terrains
- Modifying our IWF and X file loading functions to automate geometry registration with the collision system
- Interfacing the application with our collision system
- Applying a simple Newtonian physics model to our player to accommodate movement that accounts for forces (friction, drag, gravity, etc.)

The CCollision Class

The CCollision class will be a member of our CScene class. It will be charged with the task of determining collisions with scene geometry and generating appropriate responses. The CCollision class has its core intersection functions declared as static members so that they can be used directly by external sources in the event that the full suite of functionality offered by this class is not required by the user. All of the query functions have been discussed in detail in the textbook, but there will be a minor upgrade to the EllipsoidIntersectScene function presented in the textbook to accommodate CTerrain objects that may have been registered with the collision system. So we will have to look at that later.

The CCollision class is declared in CCollision.h and its implementation is in CCollision.cpp. Below we see the CCollision class declaration. We have removed the list of member functions to improve readability, so check the source code for a full list of methods. All we are interested in at the moment are the member variables and structures declared within the class scope. If you have read the textbook, then all of these structures and nearly all of the member variables will be familiar to you. We have added one or two (in bold below) that were not shown in the textbook version of the class to facilitate support for height mapped terrains.

```
class CCollision
{
public:

    //-----------------------------------------------------------------------
    // Public, class specific, Typedefs, structures & enumerators
    //-----------------------------------------------------------------------
    struct CollTriangle
    {
        ULONG       Indices[3];         // Triangle Indices
        D3DXVECTOR3 Normal;             // The cached triangle normal.
        USHORT      SurfaceMaterial;    // The material index of this triangle.
    };

    // Vectors
    typedef std::vector<CollTriangle>   CollTriVector;
    typedef std::vector<D3DXVECTOR3>    CollVertVector;

    struct DynamicObject
    {
        CollTriVector  *pCollTriangles;   // Triangle List
        CollVertVector *pCollVertices;    // vertex List
        D3DXMATRIX     *pCurrentMatrix;   // Pointer to dynamic objects
                                          // application owned external matrix
        D3DXMATRIX      LastMatrix;       // The matrix recorded on the last test
        D3DXMATRIX      VelocityMatrix;   // Describes movement from previous
                                          // position to current position.
        D3DXMATRIX      CollisionMatrix;  // Space in which collision is performed
        long            ObjectSetIndex;   // The index of the set of objects this
                                          // belongs to inside the collision
                                          // systems geometry database.
        bool            IsReference;      // Is this a reference object?
    };

    // Vectors
    typedef std::vector<DynamicObject*> DynamicObjectVector;

    struct CollIntersect
    {
        D3DXVECTOR3     NewCenter;      // The new sphere/ellipsoid centre point
        D3DXVECTOR3     IntersectPoint; // Collision point on surface of ellipsoid
        D3DXVECTOR3     IntersectNormal;// The intersection normal (sliding plane)
        float           Interval;       // The Time of intersection (t value)
        ULONG           TriangleIndex;  // The index of the intersecting triangle
        DynamicObject * pObject;        // A pointer to the dynamic object that has
                                        // been collided with.
    };
```

```
    // Vectors
    typedef std::vector<CTerrain*>      TerrainVector;

    // Constructors & Destructors for This Class.
             CCollision();
    virtual ~CCollision();

 private;

    // Private Variables for This Class.
    CollTriVector       m_CollTriangles;
    CollVertVector      m_CollVertices;
    DynamicObjectVector m_DynamicObjects;

    TerrainVector       m_TerrainObjects;
    USHORT              m_nTriGrowCount;
    USHORT              m_nVertGrowCount;

    USHORT              m_nMaxIntersections;
    USHORT              m_nMaxIterations;
    CollIntersect      *m_pIntersections;

    D3DXMATRIX          m_mtxWorldTransform;
    long                m_nLastObjectSet;
};
```

We have added only three new member variables beyond the version covered in the textbook. Let us briefly discuss them.

TerrainVector m_TerrainObjects

In the last lesson we introduced our CTerrain class, which encapsulated the creation of terrain geometry from height maps. We also added support in our IWF loading code for the GILES™ terrain entity. We can use GILES™ to place terrain entities in our IWF level and have those terrains created and positioned by our CTerrain class at load time. Each CTerrain object contains a height map and the actual geometric data created from that height map.

We will develop a memory efficient way for these terrain objects to be registered with the collision system by using the terrain height map to build only a temporary list of triangles in the region of the swept sphere as and when they are needed. As terrains are usually quite large and are comprised of many triangles, we certainly want to avoid storing a copy of every terrain triangle in the static geometry database. All our collision system will need is an array where it can store CTerrain pointers. The CCollision interface will expose an AddTerrain method which will essentially just add the passed CTerrain pointer to this internal list of terrain objects. That is all it does. No geometry is added to the collision system. When the EllipsoidIntersectScene function is called in the detection step, these CTerrain pointers will be used to access the height map of each terrain. These height maps will then be used to build only triangles that are within the same region as the swept sphere.

This member variable will be used to store any pointers to CTerrain objects that are registered with the collision system. It is of type TerrainVector, which is a typedef for an STL vector of CTerrain pointers.

typedef std::vector<CTerrain*> TerrainVector;

USHORT m_nTriGrowCount
USHORT m_nVertGrowCount
STL vectors are used throughout the collision system for storing vector and triangle data. As you know, STL vectors encapsulate the idea of a dynamic array. The STL vector has methods to easily add and delete elements to/from an array as well as methods to resize an array if we have added elements up to its maximum capacity.

The collision system has a vector for storing static triangle structures and a vector for storing static vertices. Additionally, each dynamic object we create will also have its own vectors to store its model space triangle and vertex data. Finally, with our dynamic terrain mechanism, we will be building triangle data on the fly and adding that triangle and vertex data to temporary vectors so that they can be passed into the intersection methods of the collision system. Suffice to say, that during the registration stages especially, we will be adding a lot of triangles and vertices to these vectors. This is especially true of the collision systems static triangle and vertex buffers, since they will have geometry data cumulatively added to them as multiple files are loaded and multiple meshes registered with the static database.

We never know how much space we are going to need in these vectors until we have loaded and registered all the geometry, so we usually set the initial capacity of these buffers to some initial small value. If we wish to add a vertex or a triangle to one of these vectors but find that it is full, we can simply resize the array and add an additional element on the end to place our new data. While the STL vector has methods to allow us to do that easily, we must remember that underneath the hood an array resize is still happening and they can be quite costly. Usually an array resize results in a totally new memory block being allocated that is large enough to store the new capacity before the data from the old memory block is copied into it. The memory of the original array is then deleted and the variable adjusted to point at the new memory block. Of course, all of this happens behind the scenes, but if we imagine that we wish to add 50,000 vertices to the collision databases static vertex vector, as each one is added, the array would have to be resized so that in the worst case there is only enough space for that new vertex. In this worst case, that would mean that while registering those 50,000 vertices, we have also performed 50,000 memory allocations, 50,000 block memory copies and 50,000 memory de-allocations. Again, that is a worst case scenario as most STL vector implementations are far more efficient than this. Still, we wish to avoid slowing down our registration of geometry too much and would like to try to avoid excessive memory fragmentation.

The solution is simple if we are prepared to spare a little memory. We can instead make the decision that when a vector is full and we wish to add another element to it, we can increase its capacity by a fixed size (e.g., 500). We can now add another 500 vertices before we fill it up and have to increase its capacity again. This would cut down our (worst case) 50,000 resize example down to 100. To be sure, at the end of the loading process, this vector could have a few hundred unused elements at the end of its array. But even this can be corrected after the loading process is complete by resizing the vector so that the capacity of the vector matches its number of contained elements.

The m_nTriGrowCount and m_nVertGrowCount variables contain the resize amount for times when vertices are triangles are being added to a data buffer that is currently full. If you set m_nTriGrowCount to 250, when the m_nCollTriangles vector reaches capacity and more space is needed, it will be resized to accommodate another 250 triangles (even though we might only be adding just one). We have seen all of this logic many times before, so it should be quite familiar to you.

D3DXMATRIX m_mtxWorldTransform

The m_mtxWorldMatrix member is used when adding of triangle data to the static database. When we load meshes from a file, those meshes will typically be defined in model space and accompanied by a world transformation matrix that describes the location of that mesh within the scene. When we are adding such mesh data to our static collision geometry database, we are only interested in the world space vertex positions. Therefore, the vertices of the mesh should be transformed by that matrix prior to being stored in the static collision database. The CCollision class exposes a SetWorldTransform method to set this matrix member to the world transform for the mesh that is about to have its vertex data registered. When the vertices of that mesh are passed into the CCollision::AddIndexedPrimitive method to register them with the static database, the vertices will be transformed by this matrix into world space prior to being added to the static collision geometry list. Thus, application registration of three static model space meshes with the collision system would take on the following form:

```
pCollision->SetWorldTransform(…);
pCollision->AddIndexedPrimitive(…);

pCollision->SetWorldTransform(…);
pCollision->AddIndexedPrimitive(…);

pCollision->SetWorldTransform(…);
pCollision->AddIndexedPrimitive(…);
```

Let us now discuss the functions to the CCollision class that we have not yet seen. We will start by looking at the simple initialization functions (the constructor, etc.) and will then discuss the geometry registration functions over several sections. We will then look at the upgraded EllipsoidIntersectScene function and see how we have added intersection testing for registered CTerrain objects.

CCollision::CCollision()

In our application, we will only need to use one instance of a CCollsion object. To keep things simple, this object will be a member of the CScene class. The only constructor is the default one, which simply initializes the member variables of the collision system at application startup.

```
CCollision::CCollision()
{
    // Setup any required values
    m_nTriGrowCount     = 100;
    m_nVertGrowCount    = 100;
    m_nMaxIntersections = 100;
    m_nMaxIterations    = 30;
    m_nLastObjectSet     = -1;

    // Reset the internal transformation matrix
    D3DXMatrixIdentity( &m_mtxWorldTransform );

    // Allocate memory for intersections
    m_pIntersections    = new CollIntersect[ m_nMaxIntersections ];
}
```

The default STL vector resize quantity is set to 100 for both vertex and triangle buffers. Every time our collision system tries to add another triangle or another vertex to an STL vector that is currently filled to capacity, the vector will have its capacity increased by 100. We also set the size of the intersection buffer to 100. This is the CollIntersect buffer that is used by the EllipsoidIntersectScene function to return collision information back to the response phase. We use a default value of 100 as it is unlikely a moving entity will collide with 100 triangles in a single update. Either way, the collision system is really only interested in the first intersection, so even if the buffer is not large enough to return all intersection information structures back to the response step, it will not adversely effect the collision detection or the response phase. The remaining intersection information is only used to compile a bounding box that is returned to the application. Feel free to change this value if needed.

We also set the maximum number of iterations that will be executed by the collision system to a default of 30 loops. If our CollideEllipsoid cannot deplete the slide vector and calculate the final resting place the of the ellipsoid after 30 tries, the collision detection phase will be called one more time and the first non-intersecting position returned will be used. This should rarely happen, but it allows us to cover ourselves when it does.

There is also a member variable called m_nLastObjectSet which is used by the system to store the last object set index that was issued. This is set to -1 initially because we have not yet added any dynamic objects and our collision system has not yet issued any object set indexes. Every time a new group of dynamic objects is registered with the collision system, this value will be incremented and assigned to each dynamic object in that group.

We also set the collision object's world transformation matrix to an identity matrix. The application can set this to a mesh world matrix when it wishes to register a mesh with collision database.

Finally we allocate the array of CollIntersect structures that will be used as the transport mechanism to return triangle intersection information from the detection phase (the EllipsoidIntersectScene function) back to the response phase (the CollideEllipsoid function).

CCollision::SetTriBufferGrow / SetVertBufferGrow

These two functions are simple utility functions that allow you to alter the amount that the capacity of either a vertex or triangle STL vector is expanded when it becomes full and new data is about to be added.

```
void CCollision::SetTriBufferGrow( USHORT nGrowCount )
{
    // Store grow count
    m_nTriGrowCount = max( 1, nGrowCount );
}
```

```
void CCollision::SetVertBufferGrow( USHORT nGrowCount )
{
    // Store grow count
    m_nTriGrowCount = max( 1, nGrowCount );
}
```

Notice that in each function we do not simply store the passed nGrowCount value in the respective CCollision member variables because we must not ever allow these values to be set to zero. If we did, it would break our collision system. We use these values to make room for the extra elements that are about to be added, so it must be set to at least 1.

CCollision::SetMaxIntersections

This function allows you to alter the size of the buffer that is used to transport triangle collision information from the detection phase to the response phase. It essentially defines how many intersections in a given update we are interested in recording information for. This must be set to a size of at least 1 since the response phase uses the first element in this buffer to extract the new ellipsoid position, intersection point, and collision normal. Any additional elements in this array are collisions with other triangles at the exact same *t* value. The application may or may not wish to know which other triangles were hit and as such, we can adjust the size of this array to fit our needs. The default size is 100, but the application can change this default value by calling this function.

```
void CCollision::SetMaxIntersections( USHORT nMaxIntersections )
{
    // Store max intersections
    m_nMaxIntersections = max( 1, nMaxIntersections );

    // Delete our old buffer
    if ( m_pIntersections ) delete []m_pIntersections;
    m_pIntersections = NULL;

    // Allocate a new buffer
    m_pIntersections = new CollIntersect[ m_nMaxIntersections ];
}
```

Notice how the function will not allow the value of m_nMaxIntersections to be set to a value of less than 1 because the system needs to record at least one collision in order to employ the response step. The function then deletes the previous array and allocates a new array of the requested size to replace it.

Collision::SetMaxIterations

This function allows the application to tailor the maximum number of response iterations that will be executed while trying to resolve the final position of the ellipsoid in a collision update (see earlier discussions). The default is 30 iterations (see constructor).

```
void CCollision::SetMaxIterations( USHORT nMaxIterations )
{
    // Store max iterations
    m_nMaxIterations = max( 1, nMaxIterations );
}
```

Once again, we must have at least a single iteration or our collision detection and response code would never be called at all.

CCollision::SetWorldTranform

This simple function can be used by the application to send a world transformation matrix to the collision system before registering any vertex and triangle data for a mesh that needs to be converted into world space.

```
void CCollision::SetWorldTransform( const D3DXMATRIX& mtxWorld )
{
    // Store the world matrix
    m_mtxWorldTransform = mtxWorld;
}
```

When we load internal meshes exported from GILES™, the vertices are already defined in world space, so we can pass them straight into the AddIndexedPrimtive function and leave this matrix set at identity (default). When this is not the case, we must set this matrix prior to registering each model space mesh.

Geometry Registration Methods

This section will discuss the geometry registration methods exposed by the CCollision interface. These are the functions an application will use to register pre-loaded or procedurally generated geometry with the collision system prior to entering the main game loop.

CCollision::AddBufferData

Many of the registration functions that are exposed to the application by the CCollision class will typically involve adding vertex data and triangle data to STL vectors. Whether these vectors are the ones that store the static vertex and triangle geometry buffers or whether they represent the vertex and triangle geometry buffers for a single dynamic object, we will wrap this functionality in a private generic function that can be called to perform this core task for all registration functions. Now that we are adding specialized terrain support to our collision system, we will need to build triangle data for that terrain on the fly when the terrain is queried for collision. These terrain triangles and vertices will also need to be added to temporary STL vectors so that they too can be passed into the EllipsoidIntersectBuffers function which performs the core intersection processes.

Before we discuss the geometry registration functions exposed by the CCollision class, we will first examine the code to the CCollision::AddBufferData method. This is a generic function that is passed two STL vectors (one stores vertices and another stores triangles) and arrays of vertex and index data that should be added to these STL vectors. This function is called from many places in the collision system to take vertex and index data, create triangle information from it, and then add the triangle data to the two passed buffers. By placing this code in a generic function like this, we can use the same code to add geometry to the static geometry STL vectors, the STL vector of each dynamic object, and the temporary STL vectors that are used at runtime to collect relevant terrain data in response to a collision query.

This function will convert the input data into the format that our vertex and triangle STL vectors require. This will involve constructing triangles from the passed indices and generating the triangle normals. This information can then be stored in a CollTriangle structure and added to the passed triangle buffer. The function is also responsible for recognizing when an STL vector is full and expanding its capacity by the values stored in the m_nTriGrowCount and the m_nVertGrowCount member variables.

Finally, remember that an application will never directly call this function; it is private. It is simply a helper function that is used internally by the collision system to cut down on the amount of redundant code that would otherwise have to be added to each registration function. All of these registration functions need to add geometry to buffers, so there would be no sense in duplicating such code in every function.

Let us look at this function a few sections at a time, starting first with its parameter list.

```
bool CCollision::AddBufferData( CollVertVector& VertBuffer,
                                CollTriVector& TriBuffer,
                                LPVOID Vertices,
                                LPVOID Indices,
                                ULONG VertexCount,
                                ULONG TriCount,
                                ULONG VertexStride,
                                ULONG IndexStride,
                                const D3DXMATRIX& mtxWorld )
{
    ULONG        i, Index1, Index2, Index3, BaseVertex;
    UCHAR       *pVertices = (UCHAR*)Vertices;
    UCHAR       *pIndices  = (UCHAR*)Indices;
    CollTriangle Triangle;
    D3DXVECTOR3  Edge1, Edge2;

    // Validate parameters
    if ( !Vertices || !Indices || !VertexCount || !TriCount ||
          !VertexStride || (IndexStride != 2 && IndexStride != 4) )
       return false;
```

The first two parameters are the STL vectors that are going to receive the vertex and triangle data passed into this function. These buffers may be the static scene buffers or the buffers of a single dynamic object. In this function, we are not concerned with who owns these buffers or what they are for, only with filling them with the passed data.

The third and fourth parameters are void pointers to the vertex and index data that we would like to format into triangle data that our collision system understands. These might be the void pointers returned to the application when it locked the vertex and index buffers of an ID3DXMesh, or just pointers to blocks of system memory that contain vertex and index data. The VertexCount and TriCount parameters instruct the function as to how many vertices and indices are contained in these passed arrays. For example, the VertexCount parameter tells us how many vertices are in the Vertices array and the TriCount parameter tells us how many triplets of indices exist in the Indices array.

The VertexStride parameter tells the function about the size of each vertex (in bytes) in the passed vertex array. Although our function will only be interested in the positional X, Y and Z components of each vertex stored in the first 12 bytes of each vertex structure, each vertex may contain other data that our collision system is not interested (e.g., texture coordinates or a vertex normal). We will need to know the size of each vertex in the function so that we can iterate though each vertex in the array and extract only the information we need.

The IndexStride parameter should contain a value describing the size (in bytes) of each index in the indices array. As an index will be either a WORD or DWORD in size, this value will be either 2 or 4 respectively. We need to know this so that we know how to traverse the array.

The final parameter is a matrix that should be used to transform the vertices in the vertex array (into world space) prior to adding them to the STL vector.

Notice how we cast the vertex and index arrays pointers to local unsigned char pointers (pVertices and pIndices). These pointers will allow us to step through those memory blocks one byte at a time so that we can extract the information we need. We also allocate a single CollTriangle structure that will be used to temporarily build each triangle we are going to add to the CollTriangle STL vector.

Our first real task is to test the current maximum capacity of the passed STL vector that will receive the vertex data. If it is not large enough to contain all the vertices it currently contains (VertBuffer.Size) plus the number of vertices we are about to add (VertexCount) then we will have to grow the vector. We grow the vector by the number of elements stored in the m_nVertGrowCount variable (by default set to 100). Since even this may not be enough, we execute this code in a while loop so the vector will continually have its capacity increased until such a time as it is large enough to accept all the vertices we intend to add.

```
    // Catch template exceptions
    try
    {
        // Grow the vertex buffer if required
        while ( VertBuffer.capacity() < VertBuffer.size() + VertexCount )
        {
            // Reserve extra space
            VertBuffer.reserve( VertBuffer.capacity() + m_nVertGrowCount );

        } // Keep growing until we have enough
```

We also do exactly the same thing for the STL vector that is going to receive the triangle information. We resize its capacity until it is large enough to contain the triangles it already contains plus the ones we wish to add to it in this function.

```
        // Grow the triangle buffer if required
        while ( TriBuffer.capacity() < TriBuffer.size() + TriCount )
        {
            // Reserve extra space
            TriBuffer.reserve( TriBuffer.capacity() + m_nTriGrowCount );

        } // Keep growing until we have enough
```

Before we add any vertices to the vertex vector, we must record the current number of vertices that are stored there. We will need to offset the indices we are about to add by this amount so that when the are placed into the CollTriangle structure and added to the triangle vector, they still reference the (original) vertices that were passed in the accompanying vertex array.

We store the current number of vertices in the vector in a local variable called BaseVertex. When we store the indices in the triangle structure, we will add BaseVertex to each index. This will ensure that the triangle indices we add index into the correct portion of the vertex vector and maintain their relationship with their associated vertices.

```
        // Store the original vertex count before we copy
        BaseVertex = VertBuffer.size();
```

Now we will loop through each vertex in the passed vertex array using the vertex stride to move our byte pointer from the start of a vertex to the next. Our byte pointer pVertices will always point to the start of a vertex in this array. Since we are only interested in the positional 3D vector stored in the first 12 bytes, we can cast this pointer to a D3DXVECTOR3 structure to extract the values into a temporary D3DXVECTOR3. We will then transform this vertex position by the input world transformation matrix before adding the transformed position to the vertex vector (using the vector::push_back method).

```
        // For each vertex passed
        for ( i = 0; i < VertexCount; ++i, pVertices += VertexStride )
        {
            // Transform the vertex
            D3DXVECTOR3 Vertex = *(D3DXVECTOR3*)pVertices;
            D3DXVec3TransformCoord( &Vertex, &Vertex, &mtxWorld );

            // Copy over the vertices
            VertBuffer.push_back( Vertex );

        } // Next Vertex
```

At this point we have successfully transformed all the vertices in the passed array into world space and added them to the passed vertex STL vector. Now we must add the triangles to the passed CollTriangle vector.

We set up a loop to count up to the value of TriCount. Since the data is assumed to represent an indexed triangle list, we know that there should be 3*TriCount indices in the passed array. For each iteration of the loop, we will extract the next three indices in the array and make a triangle out of them. We will also generate the triangle normal before adding the triangle structure to the CollTriangle vector. Our collision system will need to know the triangle normal if the triangle has its interior collided with as this will be used as the slide plane normal.

First we will examine the section of code that extracts the three indices from the array. This may initially look a little strange, but we will explain how it works in a moment.

```
        // Build the triangle data
        for ( i = 0; i < TriCount; ++i )
        {
```

```
            // Retrieve the three indices
            Index1 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices)
                                        : *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index2 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices)
                                        : *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index3 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices)
                                        : *((ULONG*)pIndices) );
            pIndices += IndexStride;
```

We currently have a byte sized pointer to the index array, so we know that we are going to have to cast the index pointer to either a USHORT or a ULONG (WORD or DWORD) pointer in order to dereference that pointer and grab a copy of the actual index in the correct format (depending on whether we have been passed 16 or 32-bit indices). However, our CollTriangle structures *always* store indices as ULONGs (32-bit values), so in the case where we have a byte pointer to 16-bit indices, a double cast is necessary.

To clarify, if the index stride is 2, then we have an array of 16-bit (USHORT) indices. When this is the case we must first cast the byte pointer (pIndices) to a pointer of the correct type (USHORT):

```
(USHORT*) pIndices
```

We then want to dereference this pointer to get the actual value of the index pointed to:

```
*((USHORT*)pIndices)
```

We now have a 16-bit value, but our collision system wants it as a 32-bit value so we do an additional cast to a ULONG on the de-referenced 16 bit value.

```
(ULONG)*((USHORT*)pIndices)
```

If the index stride parameter is not set to 2, we have a byte pointer to an array of 32-bit (ULONG) values. When this is the case, we can simply cast the byte pointer to a ULONG pointer and deference the result.

```
*((ULONG*)pIndices)
```

Notice that as we extract the information for each index into the local variables (Index1, Index2, and Index3), we increment the byte pointer by the stride of each index (IndexStride) so that it moves the correct number of bytes forward to be pointing at the start of the next index in the array.

In the next section of code we start to fill out the local CollTriangle structure with the information about the current triangle we are about to add. In this function, we set the surface material index of each triangle to zero as this can be set by a higher level function. The surface material index allows you to assign some arbitrary numerical value to a collision triangle so that your application can determine what it should do if a collision with a triangle using that material occurs. For example, if the triangle has a

toxic material or texture applied, the application may wish to degrade the health of the player after contact is made.

In the following code, we also add the three indices we have fetched from the indices array to the indices array of the CollTriangle structure. As we add each index, we remember to add on BaseVertex to account for the fact that the original indices were relative to the start of the passed vertex array and not necessarily the start of the vertex STL vector.

```
            // Store the details
            Triangle.SurfaceMaterial = 0;
            Triangle.Indices[0]      = Index1 + BaseVertex;
            Triangle.Indices[1]      = Index2 + BaseVertex;
            Triangle.Indices[2]      = Index3 + BaseVertex;
```

We now need to generate a normal for this triangle. This will be used by the collision detection phase as the collision (slide plane) normal when direct hits with the interior of the triangle occur.

In order to calculate the normal of a triangle we need to take the cross product of its two edge vectors. In order to calculate the edge vectors we use our new indices to fetch the triangles vertices from the vertex vector.

```
            // Retrieve the vertices themselves
            D3DXVECTOR3 &v1 = VertBuffer[ Triangle.Indices[0] ];
            D3DXVECTOR3 &v2 = VertBuffer[ Triangle.Indices[1] ];
            D3DXVECTOR3 &v3 = VertBuffer[ Triangle.Indices[2] ];
```

We now create an edge vector from the first vertex to the second vertex and another from the first vertex to the third vertex and normalize the result.

```
            // Calculate and store edge values
            D3DXVec3Normalize( &Edge1, &(v2 - v1) );
            D3DXVec3Normalize( &Edge2, &(v3 - v1) );
```

We then perform a safety check to make sure that we have not been passed a degenerate triangle. We take the dot product of the two edge vectors we just calculated. If the absolute result is equal to 1.0 (with tolerance) we know that the three vertices of the triangle exist on the same line. This triangle will provide no benefit in our collision system as it has zero volume and cannot be collided with. When this is the case, we skip to the next iteration of the triangle generation loop and avoid adding the current triangle to the triangle vector.

```
            // Skip if this is a degenerate triangle, we don't want it in our set
            float fDot = D3DXVec3Dot( &Edge1, &Edge2 );
            if ( fabsf(fDot) >= (1.0f - 1e-5f) ) continue;
```

We now take the cross product of the two edge vectors and normalize the result to get our triangle normal. We store the result in the CollTriangle structure before finally adding the triangle structure to the back of the CollTriangle vector.

```
            // Generate the triangle normal
            D3DXVec3Cross( &Triangle.Normal, &Edge1, &Edge2 );
            D3DXVec3Normalize( &Triangle.Normal, &Triangle.Normal );

            // Store the triangle in our database
            TriBuffer.push_back( Triangle );

        } // Next Triangle

    } // End Try Block

    catch ( ... )
    {
        // Just fail on any exception.
        return false;

    } // End Catch Block


    // Success!
    return true;
}
```

CCollision::AddIndexedPrimitive

This is a public method that can be used to register mesh vertex and index data with the collision system. Lab Project 13.1 uses this function when loading the internal meshes from an IWF file. It will take the loaded vertex and index lists and pass them in along with information about the size of the vertex and index structures. This function is used to register mesh data with the **static** scene geometry database.

AddIndexedPrimitive is just a wrapper around the previous function. You should take note of the variables that are passed by this function to the AddBufferData method. For the first and second parameters it passes in the m_CollVertices and m_CollTriangles member variables. These are the STL vectors used by the CCollision class to contain all its static vertex and triangle geometry. In other words, the AddBufferData function will add the passed geometry data to the static buffers of the collision system. Notice also that the member variable m_mtxWorldTranform is passed in as the final parameter. This is the collision system's current world transformation matrix that will be applied to the static geometry being registered. Now you can see how setting this matrix (via the SetWorldTransform method) before registering a mesh with the collision system transforms the vertices of that mesh into world space before adding them to the static database.

```
bool CCollision::AddIndexedPrimitive( LPVOID Vertices,
                                      LPVOID Indices,
                                      ULONG VertexCount,
                                      ULONG TriCount,
                                      ULONG VertexStride,
                                      ULONG IndexStride )
{
    // Add to the standard buffer
    return AddBufferData(     m_CollVertices,
                              m_CollTriangles,
```

```
                                Vertices,
                                Indices,
                                VertexCount,
                                TriCount,
                                VertexStride,
                                IndexStride,
                                m_mtxWorldTransform );
}
```

CCollision::AddIndexedPrimitive (Overloaded)

This next method overloads the previous one and allows you to specify a numerical material index that will be assigned to each triangle. For example, if you wanted every triangle of this a mesh that was registered with the collision system to have a material index that matches its subset ID in the original D3DX mesh, you would call this AddIndexedPrimitive version multiple times for the mesh, once for each subset. In each call, you would pass only the vertices and indices of the current subset you are rendering and the subset ID as the material index.

```
bool CCollision::AddIndexedPrimitive( LPVOID Vertices,
                                      LPVOID Indices,
                                      ULONG VertexCount,
                                      ULONG TriCount,
                                      ULONG VertexStride,
                                      ULONG IndexStride,
                                      USHORT MaterialIndex )
{
    ULONG i, BaseTriCount;

    // Store the previous triangle count
    BaseTriCount = m_CollTriangles.size();

    // Add to the standard buffer
    if ( !AddBufferData( m_CollVertices,
                         m_CollTriangles,
                         Vertices,
                         Indices,
                         VertexCount,
                         TriCount,
                         VertexStride,
                         IndexStride,
                         m_mtxWorldTransform ) ) return false;

    // Loop through and assign the specified material ID to all triangles
    for ( i = BaseTriCount; i < m_CollTriangles.size(); ++i )
    {
        // Assign to triangle
        m_CollTriangles[i].SurfaceMaterial = MaterialIndex;
    }

    // Success
    return true;
}
```

The above function first records how many triangles are in the scene's static triangle vector before it adds the new buffer data, and stores the result in the BaseTriCount local variable. The AddBufferData function is then called as before to add the vertex and triangle data to the collision system's static geometry database. When the function returns, BaseTriCount will contain the index of the first triangle that was added to the buffer in this call. We can then start a loop through every triangle from the first new one that was added to the end of the triangle buffer. This is the range of triangles that were just added. We set the SurfaceMaterial member of each triangle in this range to the material index passed into the function by the caller.

Collision::AddTerrain

Although we have not yet explained exactly how our collision detection system will dynamically generate triangle data from terrain height maps for collision purposes, we know that it will need access to the CTerrain objects which the application intends to use as collision geometry. The CCollision class stores its registered terrains as a simple vector of CTerrain pointers.

This function is used by application to register CTerrain objects with the static collision database. This function is extremely simple since the collision system will use the CTerrain object pointer to generate the actual collision geometry during the intersection tests. All we need to do in this function is add a pointer to the passed terrain to the collision system's CTerrain pointer vector. We must also call the CTerrain->AddRef method to let the terrain object know that another object has a pointer to its interface. Remember, we recently added the COM reference counting mechanism to the CTerrain class and it must be used (otherwise the terrain object could delete itself from memory prematurely).

```
bool CCollision::AddTerrain( CTerrain * pTerrain )
{
    // Validate parameters
    if ( !pTerrain ) return false;

    // Catch Exceptions
    try
    {
        // We own a reference, make sure it's not deleted.
        pTerrain->AddRef();

        // Store the terrain object
        m_TerrainObjects.push_back( pTerrain );

    } // End Try Block

    catch ( ... )
    {
        // Simply bail
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

CCollision::AddDynamicObject

In the textbook we discussed how dynamic objects would be integrated into the collision system. In terms of our collision system, a dynamic object is a scene object that cannot be stored in the usual static database because its world space position/orientation can change at any moment. A simple example is the mesh of a sliding door. We would want the player to be able to collide with this door so that he/she is not allowed access to a certain area when the door is closed. The door mesh cannot be transformed into world space during registration because the application may wish to alter its world matrix in response to a game event.

When we register a dynamic object with the collision system, a new DynamicObject structure is allocated and added to the collision system's dynamic object STL vector. A dynamic object stores its own model space vertex and triangle data and a pointer to the matrix that the application can update. Whenever the application updates the world matrix for a dynamic object, it should call the CCollision::ObjectSetUpdated method to allow the collision system to recalculate the velocity matrix and collision matrix used by the intersection functions. This was all discussed in the textbook, so we will not review that material here.

This particular function should be used if the caller would like to register a single mesh as a dynamic object. We will not be using this function in Lab Project 13.1 as all the dynamic objects we register will be actors containing complete frame hierarchies of meshes. There are separate functions to aid in the registration of actors.

An application that would like to register a single mesh object with the collision system as a dynamic object should pass this function the model space vertex and index data and a pointer to the application owned world matrix for the dynamic object. A copy if this pointer will be cached in the dynamic object structure so that both the collision system and the application have direct access to it. The application will be responsible for updating the world matrix and the collision system will be able to work with those updates.

Before we look at the code remember that whenever a dynamic object is added to the collision database, it is assigned a collision ID which is referred to as an object set index. This is like the handle for the object within the collision system and is the means by which the application can inform the collision system that the matrix of that object has been updated. Because we may want to add multiple dynamic objects as part of the same object group (they share the same object set index) we have a final boolean parameter to this function. If set to true, the CCollision::m_nLastObjectSet member will be increased, thus creating a new object group assigned to the dynamic object. If this boolean is set to false then the m_nLastObjectSet index will not be increased and the object will be assigned the same index as a previous object that has been registered. This allows us to assign multiple dynamic objects the same ID, much like when an actor is registered with the collision system. If we assigned multiple objects the same ID, we can reference and update all those dynamic objects as a single group using a single collision index. If you want each dynamic object you register to have its own object set index (most often the case) then you should pass true as this parameter.

When this function is called to add the first dynamic object to the collision system, the current value of m_nLastObjectSet will be set to -1, which is not a valid object set index. So you should never pass false

to this function when registering the first dynamic object. Essentially, you are stating that you would like the dynamic object to be added to a previously created group, which makes no sense in the case of the first dynamic object when no groups have yet been created. Just in case, the function will force this boolean to true if this is the first dynamic object that is being added to the system.

The first section of code creates an identity matrix. You will why this is the case in a moment. The function tests to see if the current value of m_nLastObjectSet is smaller than zero and if so, then we know this is the first dynamic object that has been added to the system and the NewObjectSet Boolean is forced to true. The value of the Boolean parameter is then checked and if set to true, the current value of the member variable m_nLastObjectSet is incremented to provide a new object set index for this object. The function then allocates a new DynamicObject structure and initializes it to zero for safety. We then allocate the vertex and triangle STL vectors which will be pointed at by the dynamic object structure and used to contain the model space geometry of the dynamic object's mesh

```
long CCollision::AddDynamicObject(  LPVOID Vertices,
                                    LPVOID Indices,
                                    ULONG VertexCount,
                                    ULONG TriCount,
                                    ULONG VertexStride,
                                    ULONG IndexStride,
                                    D3DXMATRIX * pMatrix,
                                    bool bNewObjectSet /* = true */ )
{
    D3DXMATRIX      mtxIdentity;
    DynamicObject * pObject = NULL;

    // Reset identity matrix
    D3DXMatrixIdentity( &mtxIdentity );

    // Ensure that they aren't doing something naughty
    if ( m_nLastObjectSet < 0 ) bNewObjectSet = true;

    // We have used another object set index.
    if ( !bNewObjectSet ) m_nLastObjectSet++;

    try
    {
        // Allocate a new dynamic object instance
        pObject = new DynamicObject;
        if ( !pObject ) throw 0;

        // Clear the structure
        ZeroMemory( pObject, sizeof(DynamicObject) );

        // Allocate an empty vector for the buffer data
        pObject->pCollVertices  = new CollVertVector;
        if ( !pObject->pCollVertices ) throw 0;

       pObject->pCollTriangles = new CollTriVector;
        if ( !pObject->pCollTriangles ) throw 0;
```

At this point we have a new dynamic object which also has pointers to two new (currently empty) STL vectors for the model space the vertices and triangles of the mesh. We set the dynamic object's matrix

pointer to point at the matrix passed by the application. This will describe the current world space transformation of the dynamic object at all times. We also copy the values of this matrix into the LastMatrix and CollisionMatrix members of the structure since the object is currently stationary. These values will be set correctly the first time the CCollision::ObjectSetUpdated method is called prior to any collision tests.

```
        // Store the matrices well need for
        pObject->pCurrentMatrix  = pMatrix;
        pObject->LastMatrix      = *pMatrix;
        pObject->CollisionMatrix = *pMatrix;
        pObject->ObjectSetIndex  = m_nLastObjectSet;
        pObject->IsReference     = false;
```

Notice in the above code that we then assign the value of the m_nLastObjectSet member variable as the dynamic object's ID. If the Boolean parameter was set to false, this will not have been incremented and the object will be assigned the same ID as the last dynamic object that was created (i.e., adding this dynamic object to a pre-existing object set). If the Boolean parameter was set to true, then the value would have been incremented, creating a new and currently unique ID, making this object the first to be added to this new object set. We also set the reference member of the structure to false since this is not a referenced dynamic object (it has its own vertex and triangle data).

Our next task is to format the passed model space vertices and indices of the dynamic object's mesh and add them to its STL geometry vectors. This is no problem since we can once again use our AddBufferData function for this task.

```
        // Add to the dynamic objects database
        if ( !AddBufferData(  *pObject->pCollVertices,
                              *pObject->pCollTriangles,
                              Vertices,
                              Indices,
                              VertexCount,
                              TriCount,
                              VertexStride,
                              IndexStride,
                              mtxIdentity ) ) throw 0;

        // Store the dynamic object
        m_DynamicObjects.push_back( pObject );

    } // End try block
```

The first and second parameters are the dynamic object's geometry vectors to be filled with the formatted data. The final parameter is the identity matrix we created at the head of the function. We do this because we know that the AddBufferData function will transform all the vertices we pass it by this matrix to transform them into world space. In the case of a dynamic object, we want the vertices to remain in model space since they will be transformed into world space on fly by the collision detection routines. After the AddBufferData function returns, our dynamic object structure will be fully populated and its geometry vectors will contain all the relevant model space vertices and triangles. As a final step, we then add this dynamic object structure to the collision system's dynamic object array.

If anything went wrong in the process and an exception was thrown, we free any memory we may have allocated in the catch block before returning an error value of -1.

```
    catch (...)
    {
        // Release the object if it got created
        if ( pObject )
        {
            if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
            if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;
            delete pObject;

        } // End if object exists

        // Return failure
        return -1;

    } // End Catch Block
```

If we get this far then everything worked and we return the object set index that was assigned to the dynamic object. Our application will store the object set indices of each dynamic object we register in the CObject structure that owns the mesh. When we update the matrix of a CObject, we can then use this object set index to notify the collision system that the matrices of the dynamic object need to be re-cached using the CCollision::ObjectSetUpated method. This was discussed in the textbook.

```
    // Return the object index
    return m_nLastObjectSet;
}
```

CCollision::AddActor

Adding single meshes as dynamic objects is certainly useful, but most of our previous demos have used our CActor class. As we are by now intimately aware, our CActor class encapsulates the loading, rendering and animation of complete frame hierarchies that may contain multiple meshes. We will definitely want our collision system to expose a way to also register these multi-mesh constructs with our collision system.

An actor that has no animation data or that is not intended to be moved throughout the scene is essentially a static actor. Registering it with the collision system will mean storing the triangles of each mesh in that hierarchy in the static collision database. This is not very difficult. We just have to traverse the hierarchy searching for mesh containers and for each one container we find, we will lock its vertex and index buffers and transform the vertices into world space using the mesh container's absolute frame matrix (not the parent relative matrix). This does mean that we will also need to be passed the current world matrix for the entire actor as it will influence the world matrices stored at each frame in the hierarchy. We will need to update the hierarchy's absolute matrices using the passed world matrix as we traverse the hierarchy. This is so we know that all absolute frame matrices in the hierarchy are correctly set to contain the world space transforms for each mesh container before using it to transform their

vertices. For each mesh container, we will transform the vertices into world space and then use the AddBufferData function to add this geometry to the static database.

If the actor contains animation or if the application intends to animate the position of the actor within the scene, then we will also need a means to register an actor as a dynamic object, or more correctly, a dynamic object set. Once again, this is not so difficult. If the actor is to be added as a dynamic object set, we will need a function that will traverse the hierarchy searching for mesh containers. For each mesh container it finds it will allocate a new DynamicObject structure and add it to the collision system's dynamic object array. We will lock the vertex and index buffers of each mesh and copy the *model space* vertex and index data into the dynamic object's geometry arrays (using AddBufferData). We will assign each dynamic object we create from a given actor the same object set ID. We know when the actor is animated and all its frame matrices are updated, calling the ObjectSetUpdated method and passing this single ID will result in all dynamic objects created from that actor having their matrices re-cached. Each dynamic object that is created from an actor will also store (as its matrix pointer), a pointer to the absolute matrix in the mesh container's owner frame. This will always store the world space transformation of the mesh whether an animation is playing or not.

So we need two strategies for registering actors. We will wrap both techniques in a single function that uses its parameters to decide whether the actor should be added as a dynamic object set or as static geometry. We pass a Boolean parameter to this function called Static which, if set to true, will cause the meshes contained inside the actor to be registered statically after having been transformed into world space. Otherwise, each mesh in the hierarchy will have a dynamic object created for it and added to the collision system's dynamic object array. The functions used to actually perform the registration process are private functions called AddStaticFrameHierarchy and AddDynamicFrameHierarchy and are for the registration of static and dynamic actors, respectively.

The wrapper function that our application calls is called AddActor and accepts three parameters. The first is a pointer to the CActor that we would like to register. The second is a matrix describing the placement of the actor within the scene (i.e., the root frame world matrix). The third parameter is the Boolean that was previously discussed that indicates whether the actor's geometry should be registered as static or dynamic triangle data.

```
long CCollision::AddActor(    const CActor * pActor,
                              const D3DXMATRIX& mtxWorld,
                              bool Static /*= true*/ )
{
    // Validate parameters
    if ( !pActor ) return -1;

    // Retrieve the root frame
    D3DXFRAME * pFrame = pActor->GetRootFrame();

    // If there is no root frame, return.
    if ( !pFrame ) return -1;

    // Add as static or dynamic geometry?
    if ( Static )
    {
        if ( !AddStaticFrameHierarchy( pFrame, mtxWorld ) ) return -1;
```

```
    } // End if static
    else
    {
        if ( !AddDynamicFrameHierarchy( pFrame, mtxWorld ) ) return -1;

    } // End if dynamic

    // We have used another object set index.
    m_nLastObjectSet++;

    // Return the object index
    return m_nLastObjectSet;

}
```

Notice that before we call the AddStaticFrameHierarchy or AddDynamicFrameHierarchy functions, we fetch the root frame of the actor to pass in as a parameter. These functions are recursive and will start at the root frame of the hierarchy and traverse to the very bottom searching for mesh containers.

It may seem odd that we increment the m_nLastObjectSet member variable at the end of the function rather than before the recursive functions are called. This is because in the actual recursive functions, this value will be incremented by assigning an object set index of m_nLastObjectSet+1 to each dynamic object we create. In other words, if the last dynamic object we added was issued an ID of 10, all the dynamic objects created by these recursive functions would be assigned an ID of 10+1=11. Remember, every dynamic object created from an actor will be assigned the same object set index and are therefore considered to belong to the same object group/set. When the recursive functions return and the dynamic objects have all been created, we then increment the m_nLastObjectSet variable so it now correctly describes the last object set index that was used. In this example, that would be a value of 11.

Of course, this function does not really do a whole lot by itself since the registration processes for both dynamic and static actors are buried away in the helper functions AddStaticFrameHierarchy and AddDynamicFrameHierarchy. Let us now have a look at these functions, starting first with the function that adds a non-animating actor to the static geometry database.

CCollision::AddStaticFrameHierarchy

This function is called by the AddActor method to register a frame hierarchy and all its contained mesh data with the static geometry database. The function recurses until all mesh containers have been found and their geometry added.

The first parameter is a pointer to the current frame that is being visited. When first called by the AddActor method, this will be a pointer to the root frame of the hierarchy. The second parameter is the world matrix that this frame's parent relative matrix is relative to. When this matrix is combined with the parent relative matrix stored at the frame, we will have the absolute world transformation matrix for the frame and the world matrix for any mesh container that it may contain. This same matrix is also passed along to each sibling frame if any should exist since all sibling frames share the same frame of reference. We can think of this matrix as being the absolute world matrix of the parent frame which,

when combined with the parent relative matrix, will provide the absolute matrix of the current frame. This matrix, once used to transform the vertices of any child mesh containers of this frame, is then passed down to the children of this frame. The matrix parameter passed into this function, when called for the root frame (from AddActor), will be the world matrix of the actor itself.

```
bool CCollision::AddStaticFrameHierarchy( D3DXFRAME * pFrame,
                                          const D3DXMATRIX& mtxWorld )
{
    D3DXMESHCONTAINER * pMeshContainer = NULL;
    LPD3DXBASEMESH      pMesh          = NULL;
    LPVOID              pVertices      = NULL;
    LPVOID              pIndices       = NULL;
    ULONG               nVertexStride, nIndexStride;
    D3DXMATRIX          mtxFrame;

    // Validate parameters (just skip)
    if ( !pFrame ) return true;

    // Combine the matrix for this frame
    D3DXMatrixMultiply( &mtxFrame, &pFrame->TransformationMatrix, &mtxWorld );

    // Retrieve the mesh container
    pMeshContainer = pFrame->pMeshContainer;
```

In the first section of the code (shown above) we first test that the frame pointer passed into the function is valid; if not, we return immediately. We then combine the passed matrix with the parent relative matrix of the current frame to generate the absolute world matrix for this frame. This is the world matrix that describes the world space placement of any meshes that are owned by this frame.

We then grab a copy of the frame's pMeshContainer pointer and store it in a local variable (pMeshContainer) for ease of access. If this pointer is NULL then there are no meshes attached to this frame and we have no geometry to add. We can just jump straight to the bottom of the function where we traverse into the child and sibling frames.

> **Note:** We are not interested in storing skinned meshes since our collision system will not directly support skinned geometry. To use skinned meshes as colliders, a good approach is to register either a geometric bounding volume that encapsulates the skinned structure or a series of low polygon meshes that bound the object and animate with it. This produces very good results with little overhead.

The next section of code loops through each mesh container attached to this frame (remember that a frame may have multiple mesh containers arranged in a linked list). Each iteration of the loop will process a single mesh in the list. If the mesh container pointer is NULL, then there are no meshes attached to this frame and the loop is never executed. Once inside the loop, if the mesh container does not have a NULL ID3DXSkinInfo pointer, then we know it contains skinning information and we skip to the next mesh in the list (see note above).

```
    // Keep going until we run out of containers (or there were none to begin with)
    for ( ; pMeshContainer != NULL;
```

```
            pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
        // Skip if this is a skin container
        if ( pMeshContainer->pSkinInfo ) continue;

        // Attempt to retrieve standard mesh
        pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pMesh;

        // Attempt to retrieve progressive mesh if standard mesh is unavailable
        if ( !pMesh ) pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pPMesh;
```

After we have made sure that the current mesh container does not contain skinning information, we then retrieve a pointer to the ID3DXMesh stored in the mesh container (via the D3DXMESHDATA member pMesh field). If this pointer is NULL then it may mean that a progressive mesh is stored here instead of a regular ID3DXMesh. When this is the case, we assign the local pMesh pointer to the pPMesh member of the container's D3DXMESHDATA structure instead. At this point, we hopefully have either a pointer to a progressive mesh or a regular mesh and we can continue. If pMesh is still NULL, then it contains a mesh data format that our system does not currently support (e.g., a patch mesh).

Now that we have a pointer to the mesh, we can use its interface to inquire about the stride of the vertices contained within. We can also use the ID3DXMesh->GetOptions method to determine whether the indices of the mesh are 16 or 32-bits wide. Notice in the following code that if the D3DXMESH_32BIT flag is set, we set the nIndexStride variable to 4 (bytes); otherwise we set it to 2. We need these pieces of information when we feed the vertices and indices of the mesh into the AddBufferData function.

```
        // If we have a mesh to process
        if ( pMesh )
        {
            try
            {
                // Retrieve the stride values
                nVertexStride = pMesh->GetNumBytesPerVertex();
                nIndexStride  = (pMesh->GetOptions() & D3DXMESH_32BIT) ? 4 : 2;
```

Now that we know how big each vertex and index is, we lock the vertex and index buffers of the mesh. The pointers to the vertex and index data are returned in the pVertices and pIndices local variables. Once we have these pointers, we can pass them straight into the AddBufferData function along with the vertex stride and index stride information, as shown below.

```
                // Retrieve the vertex buffer
                if ( FAILED(pMesh->LockVertexBuffer
                                    ( D3DLOCK_READONLY, &pVertices ) ) )
                  throw 0;

                if ( FAILED(pMesh->LockIndexBuffer
                                    ( D3DLOCK_READONLY, &pIndices ) ) )
                  throw 0;


                // Add to static database
```

```
                if ( !AddBufferData(      m_CollVertices,
                                          m_CollTriangles,
                                          pVertices,
                                          pIndices,
                                          pMesh->GetNumVertices(),
                                          pMesh->GetNumFaces(),
                                          nVertexStride,
                                          nIndexStride,
                                          mtxFrame ) ) throw 0;

                // Unlock resources
                pMesh->UnlockVertexBuffer();
                pVertices = NULL;
                pMesh->UnlockIndexBuffer();
                pIndices = NULL;

            } // End try block
```

There are a few important things to note about the AddBufferData call. First, the first two parameters we are passing in are the STL vectors of the CCollision class that contain the static vertex and geometry data. We are also using the methods of the ID3DXMesh interface to pass in the number of vertices and triangles pointed at by the pVertices and pIndices pointers, respectively. Perhaps most important is the matrix we pass in as the final parameter. This is the combined matrix of the current frame which describes the world space transformation of any meshes attached directly to this frame. The AddBufferData function will use this matrix to transform the vertices of the mesh into world space prior to adding them to the static geometry database.

After the AddBufferData function returns, the current mesh has been added to the static geometry database and we can unlock the vertex and index buffers. Our task is complete for this particular mesh.

If an exception is raised for whatever reason, the catch block below will be executed. It simply forces the unlocking of the vertex and index buffers if they were locked when the exception occurred.

```
            catch (...)
            {
                // Unlock resources
                if ( pVertices ) pMesh->UnlockVertexBuffer();
                if ( pIndices  ) pMesh->UnlockIndexBuffer();

                // Return fatal error
                return false;

            } // End catch block

        } // End if mesh exists

    } // Next Container
```

That concludes the mesh loop of this function. The above sections of code will be executed for every mesh container that is a child of the current frame being visited by our recursive function. Notice at the

top of the loop, we move along the linked list of the mesh container by setting the mesh container pointer to point at its pNextMeshContainer member with each iteration.

If we get to this point, we have processed all the meshes attached to this frame and have added them to the static geometry database. We are now ready to continue our recursive journey down the hierarchy. First we test to see if the current frame has a sibling list. If it has, then we will traverse into that list. Notice that when the function recursively calls itself in order to visit the sibling(s), it is passed the same matrix that was passed into this instance of the function. Remember, this is the combined absolute world matrix of the parent frame. Since all siblings share the same parent frame of reference, this same matrix must be passed along to each of them so that they too can combine it with their parent-relative matrices to generate their own world transforms.

```
    // Process children and siblings
    if ( pFrame->pFrameSibling )
    {
        if ( !AddStaticFrameHierarchy( pFrame->pFrameSibling, mtxWorld ) )
            return false;

    } // End if there is a sibling
```

We have now visited our sibling frames and entered their meshes into the static database. Now we must continue down to the next level of the hierarchy and visit any child frames. When the function steps into the child frame, it passes the absolute world matrix of the current frame as the matrix parameter, not the matrix that was passed into this instance of the function. We have had enough exposure to hierarchy traversals at this point to know why this is the case.

```
    if ( pFrame->pFrameFirstChild )
    {
        if ( !AddStaticFrameHierarchy( pFrame->pFrameFirstChild, mtxFrame ) )
             return false;

    } // End if there is a child

    // Success!
    return true;
}
```

That is all there is to adding an entire frame hierarchy to our static database. It is a simple recursive procedure that searches a hierarchy for meshes and adds them to the static geometry vectors.

CCollision::AddDynamicFrameHierarchy

This function is called by the AddActor method if its Boolean parameter was set to false, indicating that this is not a static actor. When this is the case a recursive procedure will be executed, much like the AddStaticFrameHierarchy function (and identical in many respects). The differences between this function and previous function occur when a mesh container is found. This time, its geometry is not transformed into world space and added to the static geometry array. Instead, a new dynamic object is created and the model space vertices are copied into the geometry arrays of the dynamic object. Also, a

pointer to the absolute matrix of the parent frame is stored inside the dynamic object also so that any changes to the hierarchy (via an animation update or an update of the position of the actor by the application), is immediately available to the collision system via this pointer. Each dynamic object is also assigned the same object set index. As the ID issued to the last dynamic object group added to the collision system will be currently stored in m_nObjectSetIndex, we can add one to this value and assign this new index to each dynamic object created from this hierarchy. As we saw when we discussed the AddActor method, the actual value of m_nObjectSetIndex in incremented when the AddDynamicFrameHierarchy function returns so that m_nObjectSetIndex is updated to store the ID we have just assigned to each dynamic object in the hierarchy.

Let us look at the code a section at a time. Note that a lot of this code is duplicated from the previous function we have just discussed, so we will move a little faster.

This function is called from AddActor and is passed the root frame of the hierarchy and the world matrix of the actor. This is the matrix describing the current position of the actor in the scene. We combine it with the parent relative matrix of the current frame being visited (initially the root) to generate the absolute world transformation matrix of the frame we are visiting. This is the world matrix of any meshes assigned to this frame.

```
bool CCollision::AddDynamicFrameHierarchy( D3DXFRAME * pFrame,
                                           const D3DXMATRIX& mtxWorld )
{
    D3DXMESHCONTAINER * pMeshContainer = NULL;
    LPD3DXBASEMESH      pMesh          = NULL;
    LPVOID              pVertices      = NULL;
    LPVOID              pIndices       = NULL;
    ULONG               nVertexStride, nIndexStride;
    D3DXMATRIX          mtxFrame, mtxIdentity;

    // Validate parameters (just skip)
    if ( !pFrame ) return true;

    // Combine the matrix for this frame
    D3DXMatrixMultiply( &mtxFrame, &pFrame->TransformationMatrix, &mtxWorld );

    // Store identity
    D3DXMatrixIdentity( &mtxIdentity );
```

Notice how we also set up an identity matrix. This will be passed into the AddBufferData function so that it does not transform the vertices of the mesh and instead copies the model space vertices straight into the geometry arrays of the dynamic object.

The next section of code is the same as the last. We set up a loop to traverse the linked list of mesh containers that may exist at this frame and skip any meshes in the list that are skins.

```
    // Retrieve the mesh container
    pMeshContainer = pFrame->pMeshContainer;

    // Keep going until we run out of containers (
    // or there were none to begin with)
```

```
    for( ;  pMeshContainer != NULL;
            pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
        // Skip if this is a skin container
        if ( pMeshContainer->pSkinInfo ) continue;

        // Attempt to retrieve standard mesh
        pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pMesh;

        // Attempt to retrieve progressive mesh if standard mesh is unavailable
        if ( !pMesh ) pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pPMesh;
```

The next section of code is executed if we have a valid mesh object. It once again retrieves the stride of both the indices and the vertices and locks the vertex and index buffers.

```
        // If we have a mesh to process
        if ( pMesh )
        {
            DynamicObject * pObject = NULL;

            try
            {
                // Retrieve the stride values
                nVertexStride = pMesh->GetNumBytesPerVertex();
                nIndexStride  = (pMesh->GetOptions() & D3DXMESH_32BIT) ? 4 : 2;

                // Retrieve the vertex buffer
                if ( FAILED(pMesh->LockVertexBuffer(  D3DLOCK_READONLY,
                                                      &pVertices ) ) ) throw 0;


                if ( FAILED(pMesh->LockIndexBuffer( D3DLOCK_READONLY,
                                                    &pIndices ) ) ) throw 0;
```

We do not wish to add this geometry to the static arrays of the collision system but instead want to create a new dynamic object from this mesh. Thus, we first allocate a new dynamic object, initialize its memory to zero, and then allocate the two STL vectors that the dynamic object will use for its vertex and triangle data.

```
                // Allocate a new dynamic object instance
                pObject = new DynamicObject;
                if ( !pObject ) throw 0;

                // Clear the structure
                ZeroMemory( pObject, sizeof(DynamicObject) );

                // Allocate an empty vector for the buffer data
                pObject->pCollVertices  = new CollVertVector;
                if ( !pObject->pCollVertices ) throw 0;

                pObject->pCollTriangles = new CollTriVector;
                if ( !pObject->pCollTriangles ) throw 0;
```

At this point, our dynamic object has its pCollVertices and pCollTriangles members pointed at the newly allocated empty vectors.

In the next section of code we will assign the pCurrentMatrix member of the dynamic object to point at the frame's combined matrix. This is the matrix that will contain the world space transform for this frame (and any of its attached meshes) when the actor is updated. We do not care what is currently stored in this matrix as we will not use it until the ObjectSetUpdated method is called to signify to the collision system that this matrix has been updated (either explicitly by the application or via an animation update). We also assign the current world space transformation of this frame (calculated above) to the LastMatrix and CollisionMatrix members. We are saying that the current position of the dynamic object in the world (described by the passed world matrix) will also be the previous position when the collision update it first called. In other words, the object has not moved yet.

We do not want to assign LastMatrix an arbitrary position even if it will be overwritten the first time ObjectSetUpdated is called; we should set it to the current position of the parent frame. If we did not do that, then for the first update we might have large values in our velocity matrix calculated between the last matrix and the current matrix which could really throw off our response system. Furthermore, our terrain collision system could end up testing thousands of triangles unnecessarily purely because the swept ellipsoid would span a great distance in its first update forcing the collision system to think that the object has moved a great distance between the last and current updates. You should remember from the textbook that it is actually the previous matrix of the dynamic object (cached in CollisionMatrix) that is used for intersection testing.

```
                // Store the matrices well need for
                pObject->pCurrentMatrix=
                                        &((D3DXFRAME_MATRIX*)pFrame)->mtxCombined;

                pObject->LastMatrix      = mtxFrame;
                pObject->CollisionMatrix = mtxFrame;
                pObject->ObjectSetIndex  = m_nLastObjectSet + 1;
                pObject->IsReference     = false;
```

As the m_nLastObjectSet will contain the last ID assigned to an actor/object that was registered, we can add one to this amount to generate the new group ID for every dynamic object created from this hierarchy. Remember, the value of m_nLastObjectSet is never altered or incremented in this function; we are assigning the same ID to every dynamic object we create from this actor. The ID of every object in this group will be one greater than the ID of the previous group that was added. We also set the IsReference member to false as this is not a reference (we will discuss adding references in a moment).

We will now add the model space vertices of the mesh to the dynamic object vectors using the AddBufferData member. Take note of the first two parameters where we are passing the dynamic object's geometry arrays and not the static scene geometry arrays as before. Also notice that as the final parameter we pass an identity matrix so that the model space vertices are not transformed into world space. We want them to be stored in the dynamic object in model space because (as we saw in the textbook), the geometry will be transformed into world space on the fly during the EllipsoidIntersectScene call.

```
                // Add to the dynamic objects database
                if ( !AddBufferData(      *pObject->pCollVertices,
                                          *pObject->pCollTriangles,
                                          pVertices,
                                          pIndices,
                                          pMesh->GetNumVertices(),
                                          pMesh->GetNumFaces(),
                                          nVertexStride,
                                          nIndexStride,
                                          mtxIdentity ) ) throw 0;
```

Now that our dynamic object structure contains all the data it needs, let us add it to the collision system's dynamic object array and unlock the vertex and index buffers of the D3DX mesh.

```
                // Store the dynamic object
                m_DynamicObjects.push_back( pObject );

                // Unlock resources
                pMesh->UnlockVertexBuffer();
                pVertices = NULL;
                pMesh->UnlockIndexBuffer();
                pIndices = NULL;

            } // End try block
```

If anything went wrong in the above code and an exception is thrown, the following catch code block will be triggered. It releases the dynamic object structure and the STL vectors we allocated to contain its geometry. We also unlock the vertex and index buffers.

```
            catch (...)
            {
                // Is there an object already?
                if ( pObject )
                {
                    if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
                    if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;
                    delete pObject;

                } // End if object created

                // Unlock resources
                if ( pVertices ) pMesh->UnlockVertexBuffer();
                if ( pIndices  ) pMesh->UnlockIndexBuffer();

                // Return fatal error
                return false;

            }
        }
    }
```

If we reach this point in the code then every mesh container that was attached to the current frame has had a dynamic object created and has had its model space geometry arrays created. We now traverse into

the sibling and child lists as before, causing a cascade effect that allows this function to recursively visit the entire hierarchy and create dynamic objects from any meshes found. Each mesh is assigned the same object set ID and as such, all the dynamic objects in the actor will belong to the same object group in the collision system.

```
    // Process children and siblings
    if ( pFrame->pFrameSibling )
    {
        if ( !AddDynamicFrameHierarchy( pFrame->pFrameSibling, mtxWorld ) )
            return false;

    } // End if there is a sibling


    if ( pFrame->pFrameFirstChild )
    {
        if ( !AddDynamicFrameHierarchy( pFrame->pFrameFirstChild, mtxFrame ) )
            return false;

    } // End if there is a child

    // Success!
    return true;
}
```

We have now covered not only how to add dynamic objects to the collision system, but have also discussed how to add entire hierarchies of dynamic objects. There is one more dynamic object registration function that we must cover which allows an actor to be registered with the collision system by referencing an actor that has previous been registered.

CCollision::AddActorReference

A referenced dynamic object shares its geometry data with the dynamic object from which it was instanced. When an actor is added to the system, multiple dynamic objects will be created. To reference an actor, we simply call the AddActorReference function passing in the object set index of the actor that was originally added. We want this function to make a copy of every dynamic object with a matching object set index. We do not need to traverse the frame hierarchy of the actor to do this. We can simply loop though the collision system's dynamic object array searching for all dynamic objects that currently exist which match the object set index passed into the function. For each one that is found, we will create a new dynamic object. However, unlike the normal creation of dynamic objects, we will not allocate these dynamic objects their own vertex and triangle vectors. Instead, we will assign their pointers to the geometry buffers of the original dynamic object we are referencing. This makes the system more memory efficient.

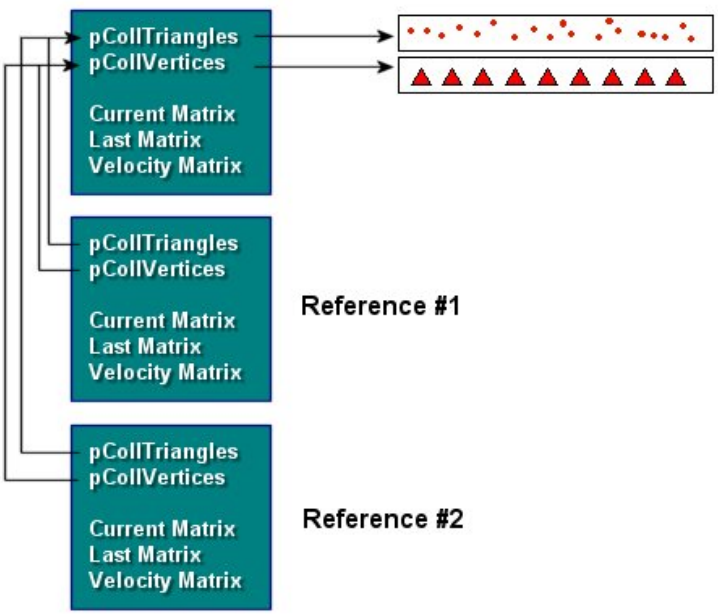


Figure 13.3

In Figure 13.3 we see how three dynamic objects might look in the collision system's dynamic object array. The topmost dynamic object was not created as a reference and therefore it has its own geometry buffers. The following two dynamic objects were registered using the AddActorReference function, passing in the object set index of the original dynamic object. As you can see, while these are dynamic objects in their own right (with their own object set index and matrix pointers), their geometry buffer pointers point at the buffers owned by the original dynamic object.

Although this makes sense, some confusion may be caused by the fact that the dynamic object's matrix pointer will be pointing to the same absolute frame matrix. The whole idea of references is to be able to place the same actor in the world in different locations and orientations without having to duplicate the geometry data. However, if each of our referenced dynamic objects points to the same world matrix (the absolute frame matrix in the actor's hierarchy) as the original dynamic object, would this not mean that all of our references will have to be in the same position in the world (negating the whole point of using references to begin with)?

While this is certainly a worthwhile observation to make, remember that the dynamic objects added to the system for the referenced actors will have different object set indices than the dynamic objects added to the system for the original actor. In the textbook we also discussed that when the application updates the position of an actor in any way (or applies an animation), it should immediately instruct the actor to update itself. This will cause the absolute frame matrices of the actor to be regenerated. If we have multiple CObjects using the same actor, we can see that this would cause the single actor to be updated many times in a given scene animation update, once for each CObject that stores a pointer to it. As we perform each update on the actor for each object that uses it (CScene::AnimateObjects), we also call the CCollision::ObjectSetUpdated function. This will update all the matrices of dynamic objects that were created from that actor or actor reference that was assigned that object set index. This function will use the matrix pointer to get the current position of the parent frame in the hierarchy which it will then use to build its LastMatrix, CollisionMatrix and VelocityMatrix members. The matrix pointer which points directly into the hierarchy is not needed by that group of dynamic objects any further in this update because its values have been cached. The frame matrix can therefore be updated by other objects in the scene after this point. The general pattern of actor updating is described below.

Imagine we have three CObjects in our scene and each one's actor pointer points to the same CActor. At the scene level, we know this actor is being referenced by three objects. That is, one set of geometry, and three instances of it. We would also want to register the same actor with the collision system as one real actor and two actor references. Assume that when we first register the actor with the collision

system we get back an object set index of 1. When we register the second actor as a reference, we get an object set index of 2. Finally, when we register the third actor (as our second reference) we get back an object set index of 3. If we also imagine that the original actor contained 10 meshes, we would now have 30 dynamic objects in our collision system. But there would only be 10 sets of geometry buffers (those from the original non-referenced actor).

We also know that each corresponding dynamic object from each of the three object groups in the collision system will store a pCurrentMatrix pointer pointing to the exact same frame matrix. However, as long as we update the position of each object separately, we will not have a problem. We just update the frames of that actor to reflect the pose of the object and then notify the collision system of a change so that it can grab a snapshot of the current values for each dynamic object's matrix at that point. The update strategy should be as follows:

1. For each CObject in CScene
    a. Apply animation to actor using this CObject's animation controller
    b. Set actor's world matrix to the CObject matrix and instruct actor to update its absolute matrices.
    c. Now that the actor is in its correct position for the reference, inform the collision system that it has been updated using CCollision::UpdateSetUpdated. We pass in the object set ID of this object reference. This function will extract the current state of each absolute matrix in the hierarchy that is being pointed at by each dynamic object and use it to generate its collision matrix, last matrix and its velocity matrix members. These will be used later in the current game loop to transform the dynamic object into world space for intersection testing.

As you can see, the fact that referenced and non-referenced dynamic objects with the same mesh point to the same physical frame matrix is not a problem, as long as we update each object group one at a time. This gives the CCollision::ObjectSetUpdated function a chance to generate its internal matrices based on a snapshot of the actor in the reference pose. From the collision system's perspective, it is almost as if we are posing the hierarchy in different poses for each object reference and recording the matrix information for each pose so that it can be used for intersection testing later.

Now that we know how the application deals with references both inside and outside the collision system, let us look at the code that allows us to add an actor reference to the collision system.

When an application calls this function it does not have to pass a pointer to an actor since this function has no need for the frame hierarchy. Every dynamic object that belongs to the original actor's object set will be duplicated in the collision system. Therefore, we just have to pass the object set index of the group we wish to reference and a world matrix describing where we would like the referenced actor to be placed in the scene. We must also pass the original world matrix that was used when we added the original actor (non-referenced) to the collision database. We will discuss why this is needed in a moment. Let us look at the first snippet of code.

```
long CCollision::AddActorReference( long ObjectSet,
                                    const D3DXMATRIX& mtxOriginalWorld,
                                    const D3DXMATRIX& mtxWorld )
{
```

```
    D3DXMATRIX      mtxInv;
    bool            bAddedData = false;

    // Generate the inverse of the original world matrix
    D3DXMatrixInverse( &mtxInv, NULL, &mtxOriginalWorld );
```

As you can see, the parameters to the function from left to right are the object set index of the group we wish to reference, the original world matrix that was used when the original group of objects was added to the collision system (the actor that was not added as a reference), and the world matrix for this reference.

Why do we need the world matrix of the original actor? In short, because we need to invert it so that we can undo the current world transform that is applied to the objects we are referencing. We will discuss this further in just a moment.

The next section of code sets up a loop to iterate through every dynamic object currently contained in the collision system's dynamic object array. It then compares the object set index assigned to each dynamic object in the array to see if it matches the object set index passed into the function. If there is no match we simply skip the dynamic object as it does not belong to the group we wish to reference. For each object that we do find that has an object set index that matches the one passed into the function, we know that it is one that we wish to reference, so we create a new dynamic object structure.

```
    // Iterate through our object database
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;
        if ( !pObject ) continue;

        // We only reference if the set indices match
        if ( pObject->ObjectSetIndex == ObjectSet )
        {
            DynamicObject * pNewObject = NULL;

            // Catch Exceptions
            try
            {
                // Allocate a new dynamic object instance
                pNewObject = new DynamicObject;
                if ( !pNewObject ) throw 0;

                // Clear the structure
                ZeroMemory( pNewObject, sizeof(DynamicObject) );
```

Unlike non-referenced dynamic objects, we do not allocate the new object its own vertex and triangle arrays. Instead, we assign its pCollVertices and pCollTriangles members to point at the geometry buffers of the object being referenced. We also set the new objects IsReference Boolean member to true.

```
                // Setup referenced data
                pNewObject->IsReference    = true;
                pNewObject->pCollTriangles = pObject->pCollTriangles;
```

```
                pNewObject->pCollVertices  = pObject->pCollVertices;
                pNewObject->pCurrentMatrix = pObject->pCurrentMatrix;

                // Store the new object set index
                pNewObject->ObjectSetIndex = m_nLastObjectSet + 1;
```

Just as in the non-referenced case, we assign the new object's matrix pointer to point to the same matrix as the dynamic object it is instancing. That is, in the case of an actor, both the dynamic object and the referenced dynamic object point to the same absolute matrix stored in the mesh's owner frame. As just discussed, this does not cause a problem because the ObjectSetUpdated function will be called when each object is individually updated, allowing the collision system to take a snapshot of the frame matrix in the correct pose for that instance.

At the bottom of the above code, we also assign the new object a new object set ID which is NOT the same as the dynamic object we are referencing. We calculate this object set ID just as before; by adding one to the ID last issued by the collision registration functions. Note that if we are referencing an actor that has multiple meshes in its hierarchy, multiple dynamic object references will be created in the collision system, but each will have the same object set ID and belong to the same group, as we would expect. After all, when an object is updated in the collision system (CCollision::ObjectSetUpdated), we wish the matrix data for each dynamic object that belongs to that group to be re-calculated.

The next step is where we use that inverted matrix we calculated earlier and stored in the mtxInv local variable. This matrix contains the inverted world space transform of the original dynamic object we are referencing (the original actor). More accurately, it is the inverted world space transformation of the root frame of the actor we are referencing. So why do we need it?

As discussed in the previous function, we want to assign the LastMatrix and CollisionMatrix members of the dynamic object to a value that has legitimate meaning when we first call our collision update function. However, we cannot really set these members to the object's previous position as it has none; it is only just being created now. We also do not want to give these matrices arbitrary values because they will be used to create the velocity matrix of the dynamic object when the ObjectSetUpdated function is first called for this object group. If we assign these matrices a position in the world that is nowhere near the current position of the object in the first collision update, we will get a huge velocity vector which will wreak havoc in our response code. Furthermore, we do not want our terrain collision system thinking that the object has moved in the first update across a huge expanse of terrain since this would mean hundreds, perhaps thousands, of triangles would need to be temporarily built and tested for intersection.

So it makes sense to assign to the dynamic object's LastMatrix and CollisionMatrix members the current transform of the frame in the hierarchy which owns this mesh. However, unlike the AddDynamicFrameHierarchy function, which traversed the hierarchy and always had access to the absolute frame matrix, in this function we are not traversing the hierarchy and we are not combining matrices. Therefore, we do not know the world space position of the dynamic objects. That is, we do not know what the absolute matrix should contain in the reference pose. We are simply looping through the dynamic objects of the collision system. Although this function was passed a world matrix for this reference, it only describes the world matrix of the root frame in the reference position. Of course, the mesh in the actor will likely be offset from the root frame by several levels of transforms. What we need

to know is not the world matrix of the root frame of the actor in the reference pose (the mtxWorld parameter), but the world matrix of the owner frame in the reference position.

Although we do not have this information at hand, we do know the current world space position of the dynamic object we are *referencing* since this information is stored in its LastMatrix parameter. We also have the original world matrix that was assigned to that actor's root frame when it was registered with the collision system since we have passed it as the second parameter to this function (mtxOriginalMatrix). Finally, we have the inverse of this matrix which undoes the transformation that was applied to the root frame of the non-referenced actor when it was registered. Therefore, if we multiply this inverse matrix with the non-referenced dynamic object's LastMatrix pointer (the dynamic object we are copying), we are essentially subtracting the position and rotation of its root frame from the world space position of the non-referenced dynamic object. This transforms the actor and all its meshes into model space (the reference pose). In other words, we are left with a matrix for that object that describes its position and orientation relative to <0,0,0> in actor space.

```
                // Transform the matrices to ensure it starts in the correct place
                D3DXMatrixMultiply( &pNewObject->LastMatrix,
                                    &pObject->LastMatrix,
                                    &mtxInv );
```

Now that we have undone the transformation that was originally applied to the non-referenced object, we can transform it back out into world space using the world space matrix of the referenced object. The result is the current world space position of the corresponding frame in the hierarchy, positioned by the reference matrix (instead of the original matrix used to position the non-referenced actor).

```
                D3DXMatrixMultiply( &pNewObject->LastMatrix,
                                    &pNewObject->LastMatrix,
                                    &mtxWorld );
                pNewObject->CollisionMatrix = pNewObject->LastMatrix;
```

At this point, both the CollisionMatrix and the LastMatrix members of the new dynamic object store the current world space position of the dynamic object using the reference's own world transform.

With our dynamic object reference now created, we finally add it to the collision system's dynamic object array. If an exception was thrown in the above code, the catch block simply deletes the dynamic object structure.

```
                // Store the dynamic object
                m_DynamicObjects.push_back( pNewObject );

                // We've added, ensure we don't release
                pNewObject = NULL;

                // We successfully added some data
                bAddedData = true;

            } // End Try Block
```

```
            catch (...)
            {
                // Release new memory
                if ( pNewObject ) delete pNewObject;

                // Return failure.
                return -1;

            } // End Catch Block

        } // End if from matching set

    } // Next object
```

At this point, if the local Boolean variable bAddedData is set to true, we know that at least one of the currently existing dynamic objects was referenced and we have added a new object group. Therefore, we increment the m_nLastObjectSet and return the new group index to the caller. Otherwise, we return -1 indicating that the requested object group could not be referenced because it seemed not to exist.

```
    // Did we find anything to reference?
    if ( bAddedData )
    {
        // We have used another object set index.
        m_nLastObjectSet++;

        // Return the object index
        return m_nLastObjectSet;

    } // End if added references
    else
    {
        return -1;

    } // End if nothing added
}
```

Thankfully, we have covered all the geometry registration functions of the CCollision class and you should have a good understanding of where the static and dynamic objects live in the collision system and how they are accessed.

CCollision::Optimize

The optimize method can optionally be called after your application has registered all static and dynamic objects with the collision system. It simply compacts the STL vectors used by both the static database and each dynamic object to their actual size, eliminating wasted space introduced during registration.

This function will test the current size and the capacity of each STL vector used by the system. This includes the two vectors that contain the static vertices and triangles and the vectors of each dynamic object used to contain its geometry. Since the size of the vector describes how many elements have been added to it, and the capacity of the vector describes how many elements can be added to it, we just have

to change the capacity of the vector to equal to its current size. This is done using the vector::reserve method which removes the wasted space from the end of each vector.

```
bool CCollision::Optimize( )
{
    // For now, we simply remove any slack from our vectors, but you could
    // (for instance) weld all the vertices here too.
    try
    {
        // Remove array slack
        m_CollTriangles.reserve( m_CollTriangles.size() );
        m_CollVertices.reserve( m_CollVertices.size() );

        // Iterate through our object database
        DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
        for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
        {
            DynamicObject * pObject = *ObjIterator;
            if ( !pObject ) continue;

            // If there is a database, remove slack
            if ( pObject->pCollTriangles )
            pObject->pCollTriangles->reserve(pObject->pCollTriangles->size());

            if ( pObject->pCollVertices  )
            pObject->pCollVertices->reserve( pObject->pCollVertices->size() );

        } // Next object

    } // End Try Block

    catch ( ... )
    {
        // Simply bail
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

While this function is not vital to proper system functioning, it is recommended that your application call it after geometry registration to keep memory requirements as minimal as possible.

Updating Collision Querying with a CTerrain Handler

In the accompanying textbook we discovered that the CCollision::EllipsoidIntersectScene function is the heart of the collision detection process. It uses the EllipsoidIntersectBuffers method to perform collision determination on both the static geometry buffers and the geometry buffers of each dynamic object registered with the system. The code was already discussed in great detail, so we will not spend a great deal of time on it (or its helper functions) in this section. However, we will discuss the special case code we will add to allow us to perform intersection tests against any CTerrain objects that have been registered with the system.

We will first look at the updates to the EllipsoidIntersectScene function where we will discover that an additional collision phase has been added. In our textbook discussion, this function performed intersection testing in two phases. It would first loop through each dynamic object and perform intersection tests on each object's geometry buffers. Then it would perform intersection testing against the collision system's static geometry buffers. We will now add a third step (which will actually turn out to be performed as the first step in our revised implementation) which will process collisions against CTerrain objects. When we examined the CCollision::AddTerrain method earlier, we saw that it simply added the passed CTerrain pointer to an internal vector of terrain object pointers. No geometry from the terrain was added to the static or dynamic geometry buffers.

While adding terrain geometry to the collision system could be done simply by registering the terrain geometry with the static database just like we do with any other static geometry (using AddIndexedPrimitive), this is obviously not the most memory efficient design. Terrains are usually quite vast and are often comprised of many thousands of triangles. As programmers, we usually have a hard enough time as it is fitting such huge terrains in memory for the purposes of rendering. If we were to store a copy of each terrain triangle in the collision database, we would effectively double the memory overhead of using that terrain. Instead, we will use a procedural geometry approach at runtime that will make using terrains based on height map data very efficient. For terrains that are not built from height map data (e.g., static meshes designed in 3D Studio MAX™) these can still be registered with the collision system using the AddIndexedPrimitive methods.

For height map based terrains which can be loaded into our CTerrain class, we will store only the CTerrain pointer in the collision system. The EllipsoidIntersectScene function will now use a new function we will implement called CollectTerrainData to temporarily build the triangles of the terrain that need to be tested for intersection. This determination will be based on the position of the ellipsoid, its velocity vector, and the data contained in the height map. The ellipsoid and its velocity vector will be mapped into the image space of the terrain's height map and used to build a bounding box describing a region on the height map that contains potentially colliding triangles. In a given update, where the ellipsoid will have moved a very small distance from its previous position, this bounding box will span only a very small area of pixels in the height map. The bounded rectangular area of pixels can then be used to build quads for the terrain that falls within that box. This is exactly the same way we build the renderable terrain data from a height map. The only difference is that instead of building quads for every pixel in the height map, we are only using a very small subsection to build a temporary mini-terrain for the area we are interested in. Once this temporary buffer has been tested for collision, the data can then be released.

The CollectTerrainData function will return the vertices and triangles for a given sub-terrain in two buffers (STL vectors) that the EllipsoidIntersectScene function can then pass into the EllipsoidIntersectBuffers function for normal intersection testing. At this point, we will not concern ourselves with how the CollectTerrainData function builds its terrain data. First we will concentrate on the additions to the EllipsoidIntersectScene function. Although the entire function is shown below, we will only discuss the new code that has been added (shown in bold). The rest of the code has been discussed in detail in the accompanying textbook.

Collision::EllipsoidIntersectScene (Version 3)

In this first section of code we see two new lines that declare two local STL vectors. One will be used to hold the vertex data returned from the CollectTerrainData function and the other used to contain the triangle data. Note that these vectors are of the same type used for the static geometry in the collision system and the model space geometry of each dynamic object. We then calculate the inverse radius vector of the ellipsoid so that we can scale the vectors in and out of eSpace as required.

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG & IntersectionCount,
                                          bool bInputEllipsoidSpace /* = false */,
                                          bool bReturnEllipsoidSpace /*= false*/ )
{
    D3DXVECTOR3 eCenter, eVelocity, eAdjust, vecEndPoint, InvRadius;
    float       eInterval;
    ULONG       i;

    // Vectors for terrain building
    CollVertVector VertBuffer;
    CollTriVector  TriBuffer;

    // Calculate the reciprocal radius
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );
```

In the next section of code we copy the passed ellipsoid position and velocity vectors into the local variables eCenter and eVelocity. If the bInputEllipsoidSpace Boolean parameter was set false it means that we would like this function to convert them into eSpace for us. In our code, the CollideEllipsoid function passes true for this parameter because it inputs both the ellipsoid position and velocity vector already in eSpace and therefore, the data is copied straight into the local variables.

```
    // 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
```

Next set the initial intersection interval along the ray (the *t* value) to 1.0, meaning that the closest intersection is at the end of the velocity vector. If this is not modified to a smaller value by the intersection routines, the path of the ellipsoid along its desired velocity vector is free from obstruction and can be moved to its desired position. We also set the initial value of IntersectionCount to zero as we have not yet found any colliding triangles.

```
    // 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 enter step one of the three detection steps. This is the new step that performs intersection testing against any CTerrain objects that have been registered with the collision system. This small section of code is virtually all the new code that has been added to this function. This is due to the fact that most of the new code is wrapped up in the CollectTerrainData method, which we will discuss in a moment.

We will first loop through every CTerrain object that has been registered with the collision system via the AddTerrain method. The pointers of each CTerrain object will be stored in the m_TerrainObjects member variable. This is an STL vector of type TerrainVector. In each iteration of the loop, we will extract the current CTerrain object being processed into the local pTerrain pointer for ease of access.

```
    // Iterate through our terrain database
    TerrainVector::iterator TerrainIterator = m_TerrainObjects.begin();
    for ( ; TerrainIterator != m_TerrainObjects.end(); ++TerrainIterator )
    {
        const CTerrain * pTerrain = *TerrainIterator;
```

We now have a pointer to the CTerrain object we want to test for intersections with our ellipsoid. The CollectTerrainData function will now be called to build and return the triangle data for the region of interest in the height map. The CollectTerrainData function must be passed the world space ellipsoid center position and velocity vector in order to do this. It uses these values to calculate the start and end positions of the ellipsoid in order to construct an image space bounding box on the height map. Because we currently have our ellipsoid position and velocity vector in eSpace, we must temporarily multiply them by the radius vector of the ellipsoid to put them back into world space. We then call the CollectTerrainData function, passing in the terrain object itself, the world space position and velocity vector, and the ellipsoid radius vector. As the final two parameters we pass the two local geometry buffers we allocated at the top of the function. When the function returns, these vectors will contain the triangle and vertex data needed to testing.

```
        // Get world space values
        D3DXVECTOR3 vecCenter   = Vec3VecScale( eCenter, Radius );
```

```
        D3DXVECTOR3 vecVelocity = Vec3VecScale( eVelocity, Radius );

        // Collect the terrain triangle data
        if ( !CollectTerrainData(   *pTerrain,
                                    vecCenter,
                                    Radius,
                                    vecVelocity,
                                    VertBuffer,
                                    TriBuffer ) ) continue;
```

If the function returns false, it means that the ellipsoid is not intersecting the overall terrain object and therefore no data could be collected from the height map. If the ellipsoid is not currently over the terrain, then mapping the bounding box of its start and end positions to image space would place it outside the entire height map. If the function returns false the ellipsoid could not possibly be colliding with this terrain and we just continue on to the next iteration of the loop and test any other terrain objects that may have been registered with the system.

If the CollectTerrainData function returns true, it means that some geometry was compiled from the terrain height map and should be tested for intersection. Luckily, we already have a function that performs all the intersection testing and interval recording -- the EllipsoidIntersectBuffers function. We can use it here as well to perform the tests against the temporary terrain buffers that were just built by CollectTerrainData.

```
        // Perform the ellipsoid intersect test against this set of terrain data
        EllipsoidIntersectBuffers(  VertBuffer,
                                    TriBuffer,
                                    eCenter,
                                    Radius,
                                    InvRadius,
                                    eVelocity,
                                    eInterval,
                                    Intersections,
                                    IntersectionCount );

        // Clear buffers for next terrain
        VertBuffer.clear();
        TriBuffer.clear();

    } // Next Terrain
```

The first and second parameters passed are the local temporary terrain geometry buffers that were just generated for this terrain object. This function will test every triangle in those buffers and record the closest colliding triangle's *t* value in the eIntersect variable. It will also store the triangle intersection information for this interval in the Intersections array.

After the EllipsoidIntersectBuffers function returns, we no longer need the temporary terrain geometry buffers, so we empty all the data they contain. They have served their purpose at this point and if any collision with the geometry did occur, the collision information will have been recorded in the Intersection array. Emptying the buffers lets us reuse them for any other terrain objects which need processing in future iterations of this loop.

At this point we have tested all the terrain objects and step one is complete. We now move on to step two where we test each dynamic object. This was all covered earlier.

```
    // 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.
        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 );
```

We now have the adjustment vector which can be added to the velocity vector to compensate for the movement of the dynamic object between collision updates. So we call the EllipsoidIntersectBuffers function to test our ellipsoid against the geometry of the dynamic object. Notice how we extend the velocity ray by the adjustment vector while passing the parameter and that the first and second parameters are the model space buffers of the dynamic object. The final parameter is the world matrix of the dynamic object in its previous position. This technique is described in the textbook.

```
        // 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 );
```

We now loop through the intersection information that was added to the array for this dynamic object in the previous function call and adjust the new eSpace position and intersection points by the adjustment vector. This gives us the new position of the ellipsoid after it has been shunted back by any dynamic object that might have collided with it. Remember, the original collision test was done using a velocity vector that was extended by the opposite amount the dynamic object has moved from its previous position.

```
        // 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
```

At this point, step two is complete and we have performed intersection tests against all terrain objects and all dynamic objects. Step three is the simplest -- we simply call EllipsoidIntersectBuffers one more time to perform intersection tests against our static geometry buffers.

```
    // Perform the ellipsoid intersect test against our static scene
    EllipsoidIntersectBuffers(      m_CollVertices,
                                    m_CollTriangles,
                                    eCenter,
                                    Radius,
                                    InvRadius,
                                    eVelocity,
                                    eInterval,
                                    Intersections,
                                    IntersectionCount );
```

We now have a CollIntersect array containing the information of all triangles that collided simultaneously at the smallest interval (eInterval). The new ellipsoid position and collision normals stored in this structure are currently in eSpace. If the caller passed false as the bReturnEllipsoidSpace boolean parameter, then it means they would like all the information stored in this array to be returned as world space vectors. When this is the case, we simply loop though each intersection structure in the compiled array and use the radius vector of the ellipsoid to scale the values from eSpace into world space. Our CollideEllipsoid function passes true as this parameter as it wants the information returned in eSpace. Thus, this code is never utilized in Lab Project 13.1.

```
    // 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
            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);
}
```

At the very bottom of the function we return true if the number of intersections found is greater than zero; otherwise we return false. How the CollectTerrainData function works is the final piece of the puzzle and will be discussed next.

CCollision::CollectTerrainData

The CollectTerrainData function is tasked with finding the region of the passed CTerrain object that falls within a bounding box described by the movement of the ellipsoid. It will then build the terrain data for this region and add the vertex and triangle data to the passed buffers (VertBuffer and TriBuffer). As we have seen in the previous discussion, these buffers are then returned to the EllipsoidIntersectScene function where they are tested for intersection before being discarded. Basically what we are after is a cheap and simple way to reject most of the triangles from consideration so we only have to build a very small amount of temporary triangle data.

Since upgrading our CTerrain class in the previous chapter, a CTerrain object can now have a world matrix that positions and orients it in the world. We will transform the ellipsoid (sort of) into terrain space and compile a terrain space bounding box that describes a region of potential colliding triangles. We know that the terrain object also has a scale vector that describes the scale of the terrain geometry in relation to the height map used to create it. For example, a scale vector of (1,1,1) would mean that neighboring pixels in the image map would represent a space of 1 world space unit. A scale vector of (10,7,12) would mean that each group of four pixels represent a quad in terrain space that is 10 units wide (x axis) and 12 units deep (z axis). The values stored in the height map for each vertex would also be scaled by 7 to produce its terrain space height. Notice that we are referring to the scaled image data as describing the terrain space dimensions of the terrain and not the world space dimensions. This is

because the world matrix is also used to potentially rotate and translate the terrain geometry to position it in the world. Therefore, a quad that is 10x10 in terrain local space may be rotated about the world Y axis by 45 degrees. That is why we must transform the start and end positions of the ellipsoid into terrain space first so that we are working in the model space of the terrain. In this space, the quads of the terrains are aligned with the X and Z axes of the local coordinate system. Figure 13.4 shows how the center and movement vectors of the ellipsoid will be used to generate a terrain space bounding box for a (typically very small) region of the overall terrain.

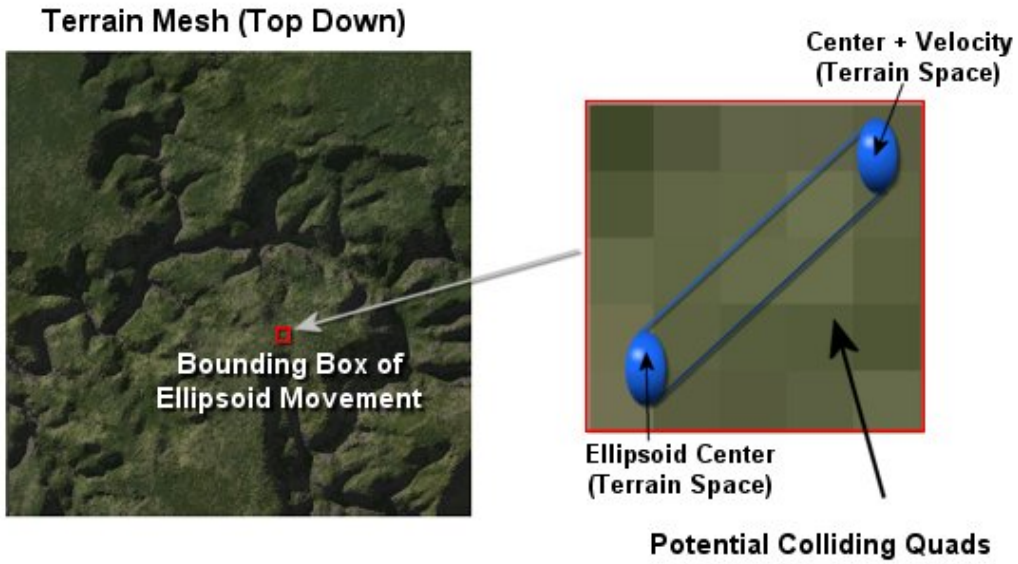


Figure 13.4

Since we know that the terrain's scale vector transforms the pixel positions in the height map into terrain space, dividing the dimensions of the terrain space bounding box by this scale vector will provide us with a bounding box in height map image space. Once we have this rectangle on the height map, we can simply loop through each row of contained pixels and build the triangle data in the exact same way we built the original terrain rendering geometry.

Let us now discuss this function a section at a time.

```
bool CCollision::CollectTerrainData(      const CTerrain& Terrain,
                                          const D3DXVECTOR3& Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollVertVector& VertBuffer,
                                          CollTriVector& TriBuffer )
{
    D3DXMATRIX   mtxInverse;
    D3DXVECTOR3  tCenter, tVelocity, tvecMin, tvecMax, Vertex;
    long         nStartX, nEndX, nStartZ, nEndZ,
                 nX, nZ, nCounterX, nCounterZ, nPitch;
    float        fLargestExtent;

    // Retrieve the various pieces of information we need from the terrain
    const float *pHeightMap = Terrain.GetHeightMap();
```

```
    D3DXMATRIX   mtxWorld   = Terrain.GetWorldMatrix();
    D3DXVECTOR3  vecScale   = Terrain.GetScale();
    long         Width      = (long)Terrain.GetHeightMapWidth();
    long         Height     = (long)Terrain.GetHeightMapHeight();

    // Retrieve the inverse of the terrains matrix
    D3DXMatrixInverse( &mtxInverse, NULL, &mtxWorld );

    // Transform our sphere data, into terrain space
    D3DXVec3TransformCoord( &tCenter, &Center, &mtxInverse );
    D3DXVec3TransformNormal( &tVelocity, &Velocity, &mtxInverse );
```

In the first section of code (shown above) we retrieve all the information about the terrain we need; a pointer to its height map, the width and height of the height map, and the world matrix of the terrain object. We also fetch the terrain's scaling vector that describes the scaling that takes place to transform a pixel in the height map into a terrain space vertex position. The world space position and velocity vector of the ellipsoid have been passed into the function as parameters, but we need them in the terrain's local space. Remember that in world space, the terrain may be arbitrarily rotated or positioned by its world matrix, so we must make sure that the ellipsoid and the terrain are in the same space prior to compiling the bounding box. The obvious choice is terrain space because we can then easily transform the terrain space box into image space using the terrain's scale vector.

In order to do this, the ellipsoid center and velocity vectors will need to be multiplied by the terrain's inverse world matrix. Thus, in the above section of code we invert the terrain matrix and transform the vector into terrain space. The terrain space vectors are stored in local variables tCenter and tVelocity. If we look at Figure 13.4, we can see that tCenter represents the bottom left blue ellipsoid and tCenter + tVelocity describes the position of the top right blue ellipsoid.

Of course, we cannot just take the start and end points of the ellipsoid into account when compiling our terrain space bounding box. The vectors describe only the extents of the ellipsoid's center point as it travels along the terrain space velocity vector. As we know, an ellipsoid has a width, height and depth described by the radius vector that was also passed into the function. Therefore, if we could transform the ellipsoids radius vector into terrain space also, we would know that the extents of the bounding box along any of its three axes can be found by adding and subtracting the terrain space radius vector from the source and destination locations of the ellipsoid and recording the smallest and largest x, y and z values. This gives us the box shown in Figure 13.4 where it bounds the start and end center points and the radii of the ellipsoid surrounding those points.

So in order to compile our terrain space bounding box we must also transform the radius vector of the ellipsoid into terrain space. Then we have everything we need to start compiling our bounding box. You would be forgiven for thinking that we can transform the radius of the ellipsoid into terrain space simply by transforming the ellipsoid radius vector by the terrain's inverse matrix. Unfortunately, this is not the case, as transforming the ellipsoid radius vector in this way will produce a very different shaped ellipsoid in terrain space.

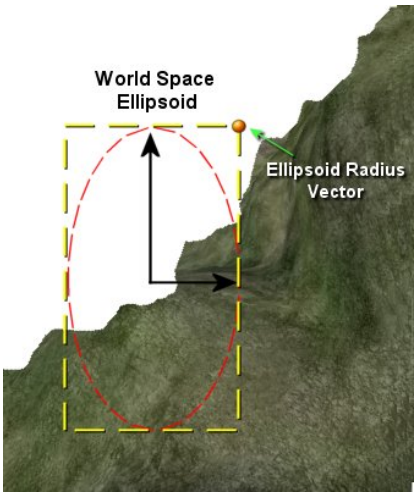


Figure 13.5

In Figure 13.5 we see both an ellipse and a terrain in world space. We will reduce the problem to 2D here for ease of explanation. The terrain's matrix has it rotated 45 degrees so that our ellipse is actually colliding into it at an angle. We know we can transform the center of the ellipsoid into terrain space using the inverse world matrix of the terrain but the ellipsoid radius is a completely different matter. If we think about the radius vector, we can see that while it is used to describe three radius values, if we think of it as a 3D vector, it actually describes a location at the tip of a bounding box that encases the ellipsoid. In Figure 13.5 the width radius is 1 and the height radius is 2 and therefore, the radius vector (1, 2) actually describes a location 1 unit along the X axis and 2 units along the Y axis. This is shown as the orange sphere at the top right corner of the bounding box. Inside the box we see the actual ellipsoid when the components of this vector are used to describe an ellipse radius.

In terrain space, the terrain will no longer be rotated; it will be perfectly aligned with the X, Y, and Z axes of the coordinate system and the ellipse shown in Figure 13.5 will be rotated forward 45 degrees. If we look at the two back arrows emanating from the center of the ellipse, we can see that they show its width and height axes. We might think that rotating the radius vector 45 degrees to the right would rotate these axes also thus providing us with a perfect terrain space bounding volume. However, this is not the case. As discussed, the world space radius vector describes only that orange sphere in Figure 13.5. When we rotate it, we are actually rotating the orange sphere by 45 degrees. In terrain space, this will now describe the top right extent of a bounding box that encloses the new terrain space ellipsoid. This is not remotely similar to the ellipsoid we were after as Figure 13.6 clearly shows.

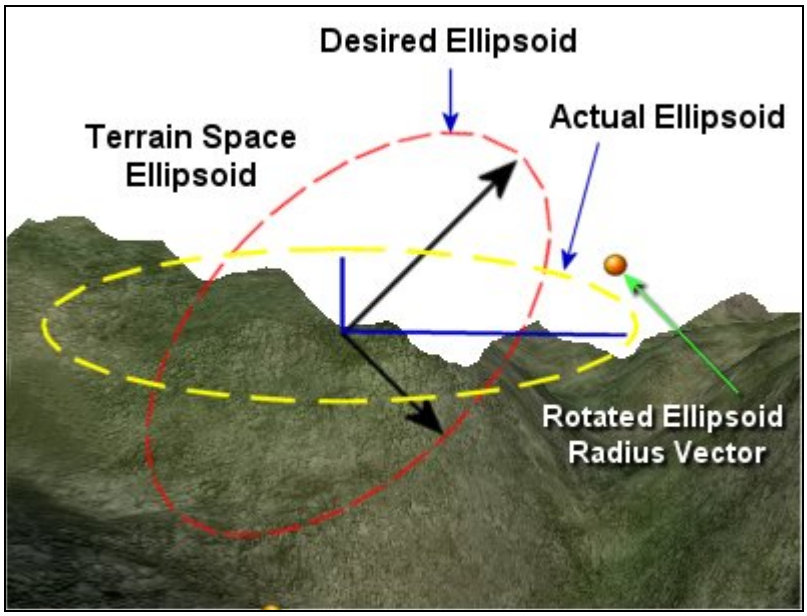


Figure 13.6

In Figure 13.6 the terrain space ellipse that we were actually hoping to get is shown as the red dashed ellipse complete with its rotated axis vectors. However, the orange sphere shows the radius vector plotted as a position describing the new top right extent of a bounding box that encases the new terrain space ellipse (post-rotation). Therefore, if we imagine this as being one corner of a box that surrounds the center of the ellipsoid, we can see the actual ellipse this describes as the yellow dashed ellipse. The ellipse has been severely squashed vertically and it is certainly not conservative. That is, the ellipsoid we ideally wanted does not fit inside the shape we actually get after rotation. Therefore, it is dangerous to use this approximation. We may reject polygons that do intersect the ellipsoid in world space but which fail to be added to the collision list because they do not intersect our terrain space ellipsoid.

The solution is simple, but comes with the cost of perhaps being a little too conservative. The problem here is the rotation of the terrain in world space and how to make sure that our ellipsoid in terrain space is at least big enough to force the compilation of a bounding box that will cause all polygons that could potentially collide with our ellipsoid to be added to the arrays. What we will do is simplify our problem by making our ellipsoid a sphere in terrain space. The rotation of the terrain in relation to a sphere is not significant as it has a radius equal in all directions. We need this sphere to be large enough to contain what our ellipsoid should like in terrain space when rotated at any angle. All we have to do then is simply take the largest component of the world space ellipsoid radius vector and essentially use a sphere that has a radius as large (see Figure 13.7).

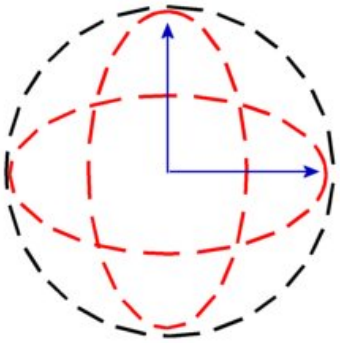


Figure 13.7

In this diagram we show two ellipsoids. The taller ellipsoid is the actual ellipsoid in world space and the wide ellipsoid shows the maximum width the ellipsoid could be in terrain space if the terrain were rotated 90 degrees. If we take the largest radius dimension of the world space ellipsoid vector and use this to build a sphere, we will have a sphere that completely encapsulates any rotation that may be applied to the original ellipsoid when transformed into terrain space.

Of course, we do not actually need to build a sphere as all we are after is the radius value. Once we have this, we can both add and subtract this value from the terrain space start and end positions of the center of the ellipsoid and record the maximum and minimum terrain space extents that we find.

The following code compiles the terrain space bounding box. It first tests each component of the world space ellipsoid radius vector to find which is the largest. This will be the radius of our hypothetical terrain space sphere.

```
    // Find the largest extent of our ellipsoid.
    fLargestExtent = Radius.x;
    if ( Radius.y > fLargestExtent ) fLargestExtent = Radius.y;
    if ( Radius.z > fLargestExtent ) fLargestExtent = Radius.z;
```

Now that we have the radius of our terrain space sphere, we will compile a bounding box by finding the minimum and maximum world space positions by adding and subtracting this radius vector from the terrain space start and end locations of the ellipsoid's center point. We start by first setting the minimum

and maximum vectors of this bounding box to values which describe an inside-out box that will be snapped to the correct size as soon as the first location tests are performed.

```
    // Reset the bounding box values
    tvecMin = D3DXVECTOR3( 9999999.0f, 9999999.0f, 9999999.0f );
    tvecMax = D3DXVECTOR3( -9999999.0f, -9999999.0f, -9999999.0f );
```

First we will add the radius of the sphere to the starting position of our ellipsoid. If this is larger than the current maximum we have recorded for that axis so far, we record the new extent. Note that this is done on a per-axis basis since we need the x, y, and z minimum and maximum extents to create a box.

```
    // Calculate the bounding box extents of where the ellipsoid currently
    // is, and the position it will be moving to.
    if ( tCenter.x + fLargestExtent > tvecMax.x )
            tvecMax.x = tCenter.x + fLargestExtent;

    if ( tCenter.y + fLargestExtent > tvecMax.y )
            tvecMax.y = tCenter.y + fLargestExtent;

    if ( tCenter.z + fLargestExtent > tvecMax.z )
            tvecMax.z = tCenter.z + fLargestExtent;
```

We next test to see if subtracting the radius of our sphere from the ellipsoid's starting position provides a new minimum extent for any of the axes.

```
    if ( tCenter.x - fLargestExtent < tvecMin.x )
            tvecMin.x = tCenter.x - fLargestExtent;

    if ( tCenter.y - fLargestExtent < tvecMin.y )
            tvecMin.y = tCenter.y - fLargestExtent;

     if ( tCenter.z - fLargestExtent < tvecMin.z )
            tvecMin.z = tCenter.z - fLargestExtent;
```

Now we test to see if adding the sphere radius to the destination location of the ellipsoid's center point (calculated as tCenter + tVelocity) provides a new maximum extent for any axis.

```
     if ( tCenter.x + tVelocity.x + fLargestExtent > tvecMax.x )
            tvecMax.x = tCenter.x + tVelocity.x + fLargestExtent;

     if ( tCenter.y + tVelocity.y + fLargestExtent > tvecMax.y )
            tvecMax.y = tCenter.y + tVelocity.y + fLargestExtent;

     if ( tCenter.z + tVelocity.z + fLargestExtent > tvecMax.z )
            tvecMax.z = tCenter.z + tVelocity.z + fLargestExtent;
```

And finally we test to see if subtracting the radius of our sphere from the destination location of the ellipsoid's center point provides us with a new minimum extent for any axis.

```
     if ( tCenter.x + tVelocity.x - fLargestExtent < tvecMin.x )
```

```
            tvecMin.x = tCenter.x + tVelocity.x - fLargestExtent;

    if ( tCenter.y + tVelocity.y - fLargestExtent < tvecMin.y )
            tvecMin.y = tCenter.y + tVelocity.y - fLargestExtent;

    if ( tCenter.z + tVelocity.z - fLargestExtent < tvecMin.z )
            tvecMin.z = tCenter.z + tVelocity.z - fLargestExtent;
```

With floating point inaccuracies being what they are, we would hate the above bounding box to miss a vertex later on just because it was outside the box by some very small epsilon value (0.000001 for example). Therefore, we will give ourselves a bit of padding by inflating the box 2 units along each axis; one unit for each axis in the positive direction and one for each axis in the negative direction.

```
    // Add Tolerance values
    tvecMin -= D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
    tvecMax += D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
```

We now have a bounding box in terrain space encompassing all the triangles that might intersect the movement of the ellipsoid. Any triangles outside this box cannot possibly collide, so sending them through our intersection routines would be unnecessary.

Next we need to transform our box into the image space of the height map. This is very easy to do. The terrain object's scale vector was used to transform a pixel in the height map into terrain space. All we had to do was multiply the x and y coordinate of the pixel by the scaling vector and we get the terrain space X and Z vertex coordinates generated from the pixel. The value stored in the pixel itself was also multiplied by the scale vector to create the height of the vertex (Y position) in terrain space. Thus, all we have to do to turn our terrain space 3D bounding box into a 2D rectangle on the height map, is reverse the process and divide the X and Z extents of this box by the scaling vector. We also snap the results to integer pixel locations on the height map as shown below.

```
    // Calculate the actual heightmap start and end points
    // (ensure we have enough surrounding points)
    nStartX = (long)(tvecMin.x / vecScale.x) - 1;
    nStartZ = (long)(tvecMin.z / vecScale.z) - 1;
    nEndX   = (long)(tvecMax.x / vecScale.x) + 1;
    nEndZ   = (long)(tvecMax.z / vecScale.z) + 1;
```

Notice how after generating the integer extents of the 2D box along both the X and Z axes, we subtract and add one to the minimum and maximum extents, respectively. This is to make sure that we always have at least four different corner points so that at least a single quad can be built. If we imagine for example that the ellipsoid is small and has no velocity, it is possible that the four extents of the terrain space bounding box, when transformed into image space and snapped to integer values, could all map to the same single pixel location in the height map. Remember that a pixel in the height map represents a vertex, so we need at least four unique points to build a quad. This addition and subtraction makes sure this is always the case.

In the next section of code we clamp the extents of the bounding box so that its extents are within the image of the height map. We certainly do not want to be trying to access pixel locations like x=600 if

the image is only 300 pixels wide. We also do not want to try and access pixel locations like x=-10 since there are no negative pixels in image space.

```
    // Clamp these values to prevent array overflow
    nStartX = max( 0, min( Width - 1, nStartX ) );
    nStartZ = max( 0, min( Height - 1, nStartZ ) );
    nEndX   = max( 0, min( Width - 1, nEndX ) );
    nEndZ   = max( 0, min( Height - 1, nEndZ ) );
```

At this point, we have four integer values which describe the corner points of our 2D bounding box where (nStartZ, nStartX) = Top Left and (nEndZ, nEndX) = Bottom Right. Remember at this point that nStartZ and nEndZ describe the image space Y axis, which increases top to bottom. Before we start stepping through these pixels and building vertices from each one, we should first make sure that the bounds of the box are not degenerate. That is, if the width or height of the box is zero then we just return true. We have no triangles to add from this terrain to the collision list.

```
    // Return if the bounds are degenerate (no op)
    if ( nEndX - nStartX <= 0 || nEndZ - nStartZ <= 0 ) return true;
```

Now it is time to start stepping through the rows and columns of pixels contained inside the rectangle on the image.

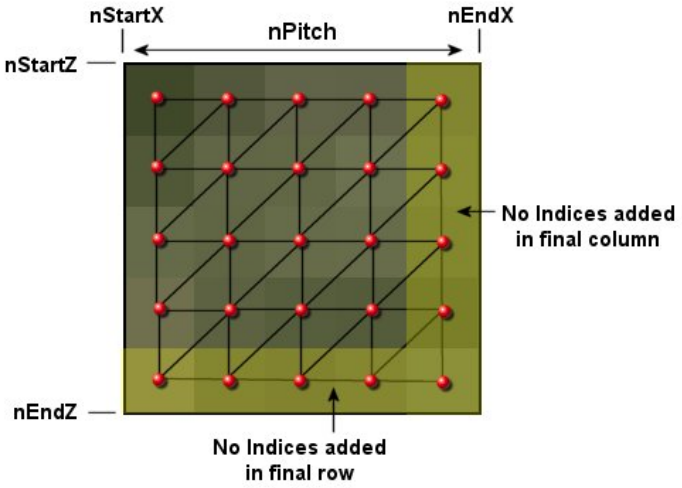


Figure 13.8

Figure 13.8 shows how we will build the vertex and triangle data for the vertices inside our box. Starting at the top left corner (nStartX , nStartY) we will work along each row and along each column. For each pixel in the height map we will add a vertex along with six indices describing the quad formed by the vertex, its right neighbor, the vertex below it and the vertex below and to the right of it. Since the top left vertex of any quad is the first to be added, when building the two triangles at the vertex, our triangles will be indexing into vertices that we have not yet added.

For example, look at the top left vertex in Figure 13.8, which shows the vertex that will be added in the first iteration of the loop. Not only will we create a vertex here, we will also create the entire top left quad (the group of four vertices in the top left corner). Although these other three vertices have not been added yet, we know where they will be positioned in the vertex buffer because we know how many vertices will be created in a given row (nPitch).

In Figure 13.8 we can see that the image space bounding box has dimensions of 5x5, so we will be adding 5 rows of 5 vertices. Therefore, if there are 5 vertices in each row, the pitch of our vertex array will be 5. When we visit the top left vertex, this will be vertex 0 in the array. Although the other three

vertices forming the quad have not yet been added (but will be added in future iterations of the loop) we do know the locations where each of these vertices will be in the vertex array and therefore we have the information with which to build the quad which uses the current vertex as the top left corner. We know for example that when processing a vertex at position *n*, the four vertex positions of all vertices forming the quad will eventually be in our vertex array at:

Top Left = n (Current Vertex)
Top Right = n+1
Bottom Left = n+pitch
Bottom Right = n+pitch+1

If we were currently processing the pixel at x=2 and y=10 in the rectangle, the vertices that would be added to the vertex buffer (assuming a pitch of 5 vertices per row) in the inner loop iteration would be:

Top Left = 2+10*pitch
Top Left = 52

In other words, the vertex we are currently processing will be added at location 52 in the vertex array. Furthermore, although they have not yet been added, the other three vertices comprising the quad will be added to the vertex array at positions:

Top Right = 2+1 + (10*pitch) = 53

Bottom Left = 2 + ((10+1)*pitch) = 67

Bottom Right = (5+1) + ((10+1)*pitch) = 68

As you can see, although we have only made it as far as processing vertex 52 (2nd vertex in the 10th row of our box), we can calculate the indices of the four vertices needed to comprise the quad. But this means that we must make sure we do not try to add any triangle data when processing the last vertex in each column and the last row of vertices. If you look at Figure 13.8 once again, you will see that the last column and the last row of vertices are highlighted yellow. When processing these vertices, we will not add any indices at all (only the vertices) because the quads that these vertices are a part of have already been added when processing the previous vertex in the column or row.

Let us now loop through each pixel in our bounding box and create the vertex and triangle data for it. First we calculate nPitch, which contains the number of vertices that will be in each row of the vertex array we will compile. We calculate this by adding 1 to the width of the image space bounding box.

```
    // Catch all exceptions
    try
    {
        // Store pitch value (to save us having to calculate each time
        nPitch = (nEndX - nStartX) + 1;
```

Notice how we add 1 to the result since this is the count of the number of vertices that will comprise each row of our vertex buffer, and we do not want it to be zero based. For example, if StartX=10 and

EndX=14 then this means there are actually 5 vertices on each row (the vertices at locations 10,11,12,13 and 14). If we were to just subtract the start dimension from the end dimensions we would get 14-10=4, which is not correct.

Now that we know the start and end positions of our bounding box along the X and Y axes of image space, we will loop through each row (outer loop) and each column (inner loop).

```
        // Build triangle data for each of the quads this falls into
        for ( nZ = nStartZ, nCounterZ = 0; nZ <= nEndZ; ++nZ, ++nCounterZ )
        {
            // For each column
            for ( nX = nStartX, nCounterX = 0; nX <= nEndX; ++nX, ++nCounterX )
            {
```

If we are not about to add the last vertex in a row or the last vertex in a column, we will allocate enough room in our triangle buffer to add two more CollTriangle structures. This is because we are about to create the quad for which the current vertex we are processing forms the top left corner. Notice how we allocate a temporary triangle structure that will be reused to add the data for each triangle of this quad to the triangle buffer.

```
                // Do not add triangle data for last column / row
                if ( nZ < nEndZ && nX < nEndX )
                {
                    CollTriangle Triangle;
                    ZeroMemory( &Triangle, sizeof(CollTriangle) );

                    // Grow the triangle buffer if required
                    if ( TriBuffer.capacity() < TriBuffer.size() + 2 )
                    {
                        // Reserve extra space
                        TriBuffer.reserve(TriBuffer.capacity()+m_nTriGrowCount );

                    } // End if should grow buffer
```

Each CollTriangle structure has a three element indices array, so we will add the indices for the first triangle. We will then add the triangle structure to the passed triangle buffer (STL vector).

```
                    // Build first triangle
                    Triangle.Indices[2] = nCounterX + nCounterZ * nPitch;
                    Triangle.Indices[1] = (nCounterX + 1) + nCounterZ * nPitch;
                    Triangle.Indices[0] = nCounterX + (nCounterZ + 1) * nPitch;

                    // Store first triangle
                    TriBuffer.push_back( Triangle );
```

Now we will reuse the CollTriangle structure to add the indices of the second triangle in the quad and add that to the triangle buffer also.

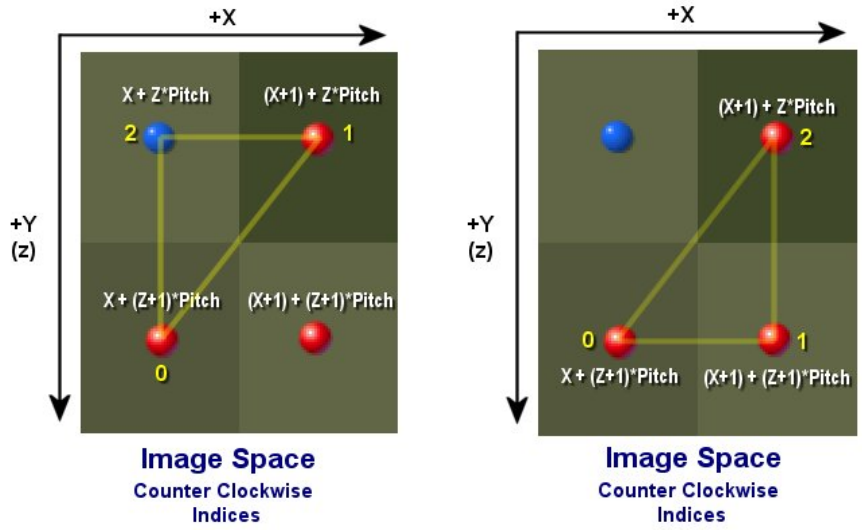
```
                    // Build second triangle
                    Triangle.Indices[2]= (nCounterX + 1) + nCounterZ * nPitch;
                    Triangle.Indices[1]= (nCounterX+1) + (nCounterZ + 1) * nPitch;
```

```
                    Triangle.Indices[0]=  nCounterX + (nCounterZ + 1) * nPitch;

                    // Store second triangle
                    TriBuffer.push_back( Triangle );

                } // End if last column / row
```

At this point we have added the two triangles that form the quad. Remember, this quad is only created and added if the current vertex we are processing (nCounterX, nCounterY) is not the final vertex in a row or the final vertex in the column (Figure 13.8).

You might have noticed that we are defining the indices of each triangle in a counter clockwise winding order. Figure 13.9 should help you visualize both the first and second triangle we add in the above code.

Figure 13.9



Triangle 1 **Triangle 2**

The vertices have been clearly indexed by each triangle in a counter clockwise order in image space. However, we must remember that the image space Y axis is equivalent to our terrain space Z axis. Furthermore, while the image space Y axis increases as it goes down the screen, in terrain space, if we were looking down on the terrain mesh from above, the Z axis would decrease as it headed down the screen. Therefore, as discussed back in 3D Graphics Module I, when we first examined terrain building using height maps, when the image is mapped to terrain space (when the image space Y axis is used as the terrain space Z axis) there is an implied 'flip' of the terrain about the image space Y axis. If we change the direction of the image space Y axis so that it is facing the opposite direction (as is the case when we build a 3D mesh from this image) you will see that the triangles are now defined with a clockwise winding order. Therefore, we define them counter clockwise in image space so that when the image is flipped during the transformation to terrain space, the winding order changes and all is well. Figure 13.10 shows the two images flipped along the image space Y axis. This is what happens when the image space Y coordinates are mapped to the terrain space Z axis.

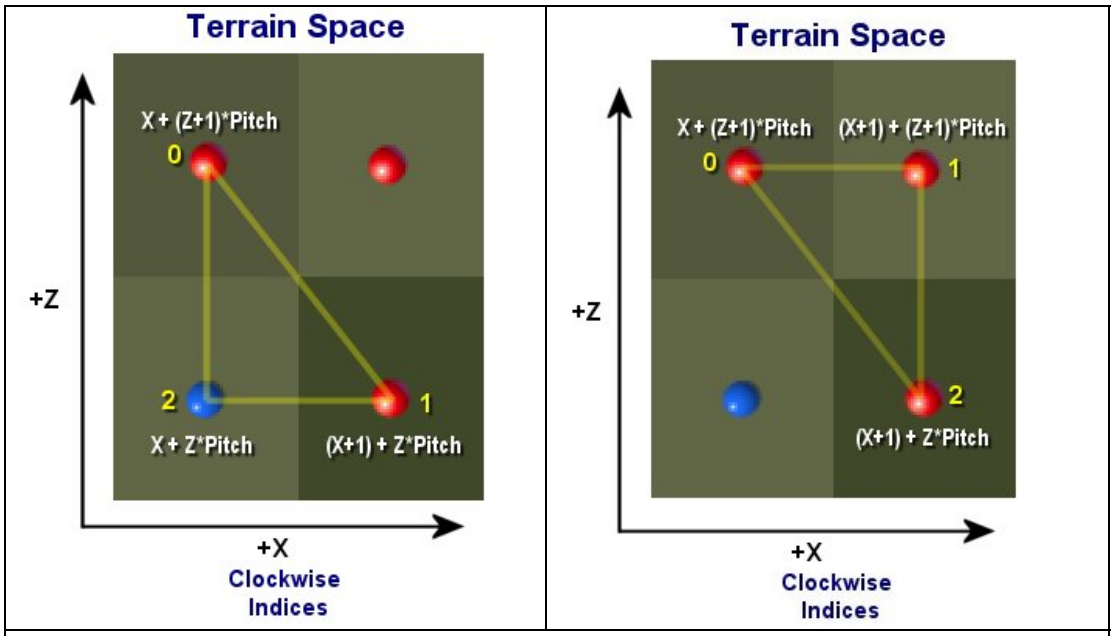


Figure 13.10 : The two triangles in terrain space

So we have added the quad (if applicable) whose top left corner is represented by the current pixel we are visiting. Now it is time to create and add the vertex itself. First we make sure there is enough room in the vertex array to add another vertex. If not, we resize the vector.

```
                // Grow the vertex buffer if required
                if ( VertBuffer.capacity() < VertBuffer.size() + 1 )
                {
                    // Reserve extra space
                    VertBuffer.reserve( VertBuffer.capacity()+m_nVertGrowCount );

                } // End if should grow buffer
```

The terrain space vertex position along the X and Z axes is simply the X and Y coordinate of the pixel scaled by the terrain object's scale vector. The Y coordinate of the vertex is the value of the pixel itself (stored in the pHeightMap array) scaled by the Y component of the terrain object's scale vector.

```
                // Calculate the vertex
                Vertex = D3DXVECTOR3( (float)nX * vecScale.x,
                                       pHeightMap[nX+nZ * Width] * vecScale.y,
                                      (float)nZ * vecScale.z  );
```

At this point, we have the terrain space vertex, but we need to return our triangle information in world space. So we multiply the vertex by the terrain object's world matrix before finally adding it to the vertex array.

```
                // Transform into world space
                D3DXVec3TransformCoord( &Vertex, &Vertex, &mtxWorld );
```

```
                // Add the vertex to the buffer
                VertBuffer.push_back( Vertex );

            } // Next Column

        } // Next Row
```

If we get to this point in the function we have successfully added a rectangular region of world space triangles to the passed vectors.

Our collision system also requires that each triangle have a normal which will be used as the slide plane normal in the case of impacts between the ellipsoid and the interior of triangle. Therefore, as a last step, we will loop through all the triangle structures we have just added to the triangle vector and generate a normal for each one. We do this using the same technique we have used many times before. That is, we use the vertices of the triangle to create two edge vectors that are tangent to the triangle surface and perform the cross product on them to generate a vector that is perpendicular to the surface. We then normalize the result.

```
        // Calculate resulting normals
        D3DXVECTOR3 v1, v2, v3, Edge1, Edge2;
        CollTriVector::iterator Iterator = TriBuffer.begin();
        for ( ; Iterator != TriBuffer.end(); ++Iterator )
        {
            // Get a REFERENCE to the underlying triangle
            CollTriangle & Triangle = *Iterator;

            // Retrieve vertices
            v1 = VertBuffer[ Triangle.Indices[0] ];
            v2 = VertBuffer[ Triangle.Indices[1] ];
            v3 = VertBuffer[ Triangle.Indices[2] ];

            // Calculate two edge vectors
            Edge1 = v2 - v1;
            Edge2 = v3 - v1;

            // Generate cross vector
            D3DXVec3Cross( &Triangle.Normal, &Edge1, &Edge2 );
            D3DXVec3Normalize( &Triangle.Normal, &Triangle.Normal );
        } // Next Triangle

    } // End Try Block

    catch ( ... )
    {
        // Return false. We failed.
        return false;

    } // End Catch

    // We added data successfully.
    return true;
}
```

At the end of the function you can see the catch block that simply returns false should an exception be thrown during execution.

Using the Collision System with Dynamic Objects

We have already discussed all the methods that allow our application to register dynamic object groups with the collision database. We discovered that when an object (or a hierarchy of objects) is registered with the collision system, an object set index is returned to the application. This index is the handle by which the application informs the collision system that it has altered the matrix (matrices) of an object (group of objects).

Some exterior entity (the application, a CObject, a D3DXFRAME, etc.) owns the matrix that contains the world space transformation for a dynamic object; the collision system simply maintains a pointer to it for access during dynamic object updates. The application can feel free to move the dynamic object about in the world by setting that matrix explicitly or by playing animations on the actor which owns the object.

CCollision::ObjectSetUpdated

Whenever the application updates the matrix of a dynamic object, it must inform the collision system by calling its ObjectSetUpdated method. The only parameter that need be passed is the object set index of the group that has had its matrix (matrices) updated. If the object set index represents a hierarchy of dynamic objects (an actor), then calling this function will cause the collision matrix of every effected dynamic object to be recalculated. Here is the code to the function.

```
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;

        // Generate the inverse of the previous frames matrix
        D3DXMatrixInverse( &mtxInv, NULL, &pObject->LastMatrix );

        // Subtract the last matrix from the current to give us the difference
        D3DXMatrixMultiply(   &pObject->VelocityMatrix,
                              &mtxInv, pObject->pCurrentMatrix );

        // Store the collision space matrix
        pObject->CollisionMatrix = pObject->LastMatrix;
```

```
        // Update last matrix
        pObject->LastMatrix = *pObject->pCurrentMatrix;

    } // Next Object
}
```

It loops though every dynamic object in the collision system's dynamic object array. For each one it finds with a matching object set index, it calculates the new velocity matrix. As discussed in the textbook, the velocity matrix describes the relative transformation from its previous position/orientation. We calculate this by inverting its previous world matrix and multiplying it with the new updated world matrix. This provides a relative movement matrix describing how the object has moved between this update and the previous one. We store this data in the velocity matrix member of the dynamic object. As we saw in the EllipsoidIntersectScene function, this matrix is used to extend the swept sphere so that intersection tests can be performed against the geometry of the dynamic object in its previous position. We then store the current previous world matrix in the collision matrix member. The collision matrix describes the previous world transform of the object and will be used alongside the velocity matrix by the intersection routines. We then copy the new current world matrix into the last matrix member so that the process can be repeated again and again every time the object group is updated. Remember, only the collision matrix and the velocity matrix are used by the intersection routines. The last matrix and current matrix members are used only to generate these two matrices.

CCollision::SceneUpdated

Rather than force the application to call the ObjectSetUpdated method for each updated object, a convenience function has been added that allows the application to tell the collision system that all dynamic objects should have their matrix states updated simultaneously with a single function call.

The SceneUpdate method of the CCollision class (shown below) is almost identical to the previous function. The difference is that it is not passed an object set index. It just updates the matrices for every dynamic object registered with the system.

```
void CCollision::SceneUpdated( )
{
    D3DXMATRIX mtxInv;

    // Update dynamic object matrices
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Generate the inverse of the previous frames matrix
        D3DXMatrixInverse( &mtxInv, NULL, &pObject->LastMatrix );

        // Subtract the last matrix from the current to give us the difference
        D3DXMatrixMultiply(   &pObject->VelocityMatrix,
                              &mtxInv,
                              pObject->pCurrentMatrix );
```

```
        // Store the collision space matrix
        pObject->CollisionMatrix = pObject->LastMatrix;

        // Update last matrix
        pObject->LastMatrix = *pObject->pCurrentMatrix;

    } // Next Object
}
```

Applications should ***never*** use this function with actors that have been registered as dynamic references. This is a simple convenience function that applied only in cases where actor references are not being used. Do you see why this must be the case?

To keep things simple, imagine a single mesh actor that has been initially registered with the collision system and then referenced 9 times (via AddActorReference). This means we will have 10 dynamic objects in the system. The first registered dynamic object allocates the geometry and its current matrix pointer would point at the object's absolute frame matrix in the hierarchy. The other 9 dynamic objects share the geometry buffers and have their matrix pointers pointing at the same frame matrix. Therefore, if you call this function, the matrix pointer of every dynamic object will point to the same matrix, the current position described by the frame. As discussed in the textbook, when actor references are used, we update the actor matrices first and then call the ObjectSetUpdated function so that the collision system can grab a snapshot of the current matrices. We then update the actor matrices for the second reference, call ObjectSetUpdated again, and so on. This is very important and has the potential to cause numerous problems if not remembered. There is no way the single actor can ever be in more than one pose at a time. If you were to call this function, all dynamic objects that were created from the same mesh in the same actor would all be assigned the same world matrix.

CCollision::Clear

There may be times when you wish to purge the collision system geometry database so that you can re-use the same collision object for a different task. The Clear method does just this. This method is also used by the CCollision destructor to release all memory before the system is deleted.

Clear resets the state of the CCollision object to its default state. It sets its internal transform matrix to an identity matrix and then loops through each dynamic object. For each dynamic object it finds that is not a reference, it will delete its geometry buffers. It does not do this for references as they do not own their own geometry. Whether a reference object or not, the dynamic object structure is then deleted from memory also. This is done for each dynamic object in the collision system's dynamic object vector.

```
void CCollision::Clear( )
{
    ULONG i;

    // Reset the internal transformation matrix
    D3DXMatrixIdentity( &m_mtxWorldTransform );
    // Release dynamic object pointers
    for ( i = 0; i < m_DynamicObjects.size(); ++i )
    {
```

```
        // Retrieve the object
        DynamicObject * pObject = m_DynamicObjects[i];

        // Delete if existing
        if ( pObject )
        {
            // We only delete our mesh data if were not a reference
            if ( !pObject->IsReference )
            {
                // Release the vectors
                if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
                if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;

            } // End if not reference

            // Delete the object
            delete pObject;
        }
    }
```

We then loop through each element in the collision system's terrain pointer vector. Before clearing this vector we must first call the Release method on each terrain pointer because our terrain object employs a COM style reference counting mechanism. You will recall how we incremented the reference count of a terrain object when its pointer was added in the AddTerrain method discussed earlier.

```
    // Release terrain objects
    for ( i = 0; i < m_TerrainObjects.size(); ++i )
    {
        // Retrieve the object
        CTerrain * pTerrain = m_TerrainObjects[i];

        // Release if existing
        if ( pTerrain ) pTerrain->Release();

    } // Next Terrain Object
```

Finally, we empty the static geometry vectors, the dynamic object vector, and the terrain vector, releasing all the memory they currently contain. We then reset the m_nLastObjectSet member variable back to its default state of -1 since there now no object sets registered with the collision system.

```
    // Empty our STL containers.
    m_CollTriangles.clear();
    m_CollVertices.clear();
    m_DynamicObjects.clear();
    m_TerrainObjects.clear();

    // Reset any variables
    m_nLastObjectSet = -1;
}
```

Registering Geometry with the Collision System

In this next section we will discuss the additions to the application's loading code that manage collision system registration. The CCollision object is actually a member of our CScene class. This is useful since this class manages loading geometry data, and the animation and rendering of that data in the main game loop. We will revisit functions such as CScene::ProcessMeshes, CScene::ProcessReferences, and CScene::ProcessEntities, which are no strangers to us. These are the functions we have used since the beginning of this training program to process the objects loaded from IWF files by the CFileIWF file object.

In this workbook we will focus on the additions to the IWF loading code in CScene. The CScene::LoadSceneFromX function will not be discussed since it has hardly changed from previous versions. It simply loads the single X file into a CActor object and then registers it with the collision system using the CCollision::AddActor function. Because adding actors to the collision system will be demonstrated in the IWF loading code, you should have no trouble noticing the necessary changes to this function.

Loading IWF Files – Recap

In virtually all of our previous projects we have used the CScene::LoadSceneFromIWF function to load in our geometry from IWF files. This function is called from the CGameApp::BuildObjects function which itself is called from CGameApp::InitInstance.

The LoadSceneFromIWF method uses the CFileIWF object contained in libIWF.lib. This library is part of the IWF SDK to automate the loading of IWF files. As we have discussed on previous occasions, the CFileIWF::Load function is used to load all the data objects contained in the IWF file into a number of vectors supplied by the library. For example, all entities contained in the IWF file are stored in the CFileIWF::m_vpEntityList vector and all internal meshes defined in the file are loaded into the CFileIWF::m_vpMeshList vector. Vectors also exist to store the materials and texture filenames. These vectors are automatically filled with data when CFileIWF::Load is called. This means all our scene object has to do on successful completion of this function is extract the data from these vectors and format it in the desired fashion. None of this is new to us as we developed this loading system way back in Module 1 and have been adding to it ever since.

The first updated function we will look at is the CScene::ProcessMeshes function. It is called from CScene::LoadSceneFromIWF to extract the data from the CFileIWF::m_vpMeshList vector. The function is passed a single parameter, the CFileIWF object that contains the meshes loaded from the file.

CScene::ProcessMeshes (Updated)

IWF files generally contain meshes in two different forms -- internal meshes and mesh references. If you have ever used GILES™ then you can think of the brushes that you place in the scene as the internal mesh type. The physical polygon data of such objects is saved out to the IWF file with a world matrix describing the position of the brush in the scene. In the case of GILES™, all brushes have their vertices

defined in world space, so the world matrix accompanying each mesh in the file will always be identity. The other type of mesh data that we have used is the reference entity. In GILES™, we can place a reference entity in the scene that contains the name of an X file. While GILES™ will physically render the geometry in the X file referenced by such an entity, this geometry is not saved out to the IWF file, only the filename of the X file is. The ProcessMeshes function is called to extract only internal geometry meshes whose geometry was stored in the IWF file. This geometry will have been loaded into the CFileIWF::m_vpMeshList array by the CFileIWF::Load function, so let us now parse this data.

We have discussed most of the code to this function before and as such we will simply step to the new code we have added.

In Lab Project 13.1, we decided only to add meshes to the collision database that are not flagged as detail objects (like a decoration), but you can easily change this behavior if you wish. After our mesh is constructed and the IWF copied, we call the collision system's AddIndexedPrimitive method and pass in the pointers to our new CTriMesh's vertex and index arrays. We use the member functions of CTriMesh to get all the information (vertex and face list pointers, the number of vertices and faces in those lists, and the stride of the vertex and indices). This single function call registers every single triangle of the CTriMesh with the collision system's static database. Notice that we do not bother setting the world transformation matrix because the internal meshes we load from GILES™ are already in world space.

```
        // Add this data to our collision database if it is not a detail object
        if ( !(pMesh->Style & MESH_DETAIL) )
        {
            // Add mesh data to collision database
            if ( !m_Collision.AddIndexedPrimitive(    pNewMesh->GetVertices(),
                                                      pNewMesh->GetFaces(),
                                                      pNewMesh->GetNumVertices(),
                                                      pNewMesh->GetNumFaces(),
                                                      pNewMesh->GetVertexStride(),
                                                      pNewMesh->GetIndexStride()))
            {
                // Clean up and fail, something bad happened
                delete pNewMesh;
                return false;

            } // End if failure to add data

        } // End if not detail object
```

CScene::ProcessReference

The CScene::LoadSceneFromIWF function calls the ProcessEntities function to process any entities that may have been loaded into the CFileIWF::m_vpEntities vector. External mesh references are stored as reference entities. The data area of a reference entity is arranged as a ReferenceEntity structure (CScene.h). This structure just contains the filename of the thing that it is referencing. When the ProcessEntities function determines it has found a reference entity, it calls the ProcessReference function. This function assumes that all reference entities are references to external X files and as such, the X files are loaded into actors along with any animation they may contain.

This function, while mostly unchanged, has a few new bits added. If the X file we are loading has already been loaded, then the actor is added to the collision system as an actor reference, otherwise it is added as a normal actor. We will move very quickly through this code, since most of it is familiar to us.

```
bool CScene::ProcessReference( const ReferenceEntity& Reference,
                               const D3DXMATRIX & mtxWorld )
{
    HRESULT                     hRet;
    CActor                    * pReferenceActor      = NULL;
    LPD3DXANIMATIONCONTROLLER   pReferenceController = NULL;
    LPD3DXANIMATIONCONTROLLER   pController          = NULL;
    long                        ObjectSetIndex       = -1;
    ULONG                       i;

    // Skip if this is anything other than an external reference.
    // Internal references are not supported in this demo.
    if (Reference.ReferenceType != 1) return true;
```

In the first section of code we test to see of the reference type member of the reference entity structure is set to 1 (external reference), if not we return as we do not currently support any other reference type. Notice how the ProcessEntities function will also pass in the world matrix of the reference which would also have been loaded from the file (as every entities is accompanied by a matrix in the IWF file).

Next we build the complete filename of the reference, and loop through each currently loaded actor. If we find an actor in the scene's CActor array that has the same name as the X file we are trying to load, then we know this X file has already been loaded into an actor and we break from the loop.

```
    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, m_strDataPath );
    _tcscat( Buffer, Reference.ReferenceName );

    // Search to see if this X file has already been loaded
    for ( i = 0; i < m_nActorCount; ++i )
    {
        if (!m_pActor[i]) continue;
        if ( _tcsicmp( Buffer, m_pActor[i]->GetActorName() ) == 0 ) break;

    } // Next Actor
```

We know at this point that if the loop variable *i* is not equal to the number of currently loaded actors, then the loop exited early because we found a matching actor. This means the X file geometry already exists in memory and we do not want to load it again. Instead, we will create an actor reference. We first get a pointer to the actor we are going to reference as shown below.

```
    // If we didn't reach then end, this Actor already exists
    if ( i != m_nActorCount )
    {
        // Store reference Actor.
        pReferenceActor = m_pActor[i];
```

We now see if any CObject exists in the scene's CObject array which is using the actor.

```
        // Find any previous object which owns this actor
        for ( i = 0; i < m_nObjectCount; ++i )
        {
            if (!m_pObject[i]) continue;
            if ( m_pObject[i]->m_pActor == pReferenceActor ) break;

        } // Next Object
```

If loop variable *i* is not equal to the number of objects in the CObject array, then we found an object that is using the actor we want to add to the collision system. Therefore, we will reference the object.

We first get a pointer to the CObject which contains the actor we wish to reference.

```
        // Add a REFERENCE dynamic object to the collision system.
        if ( i != m_nObjectCount )
        {
            CObject * pReferenceObject = m_pObject[i];
```

At this point we know that the CObject that already exists must have registered its actor with the collision system and as such, the CObject::m_nObjectSetIndex member will contain the object set index that was returned when this original actor was registered. We call the CCollision::AddActorReference function to register a new copy of this actor with the collision system at a different world space position. Notice how we pass in the object set index and the world matrix of the current reference we are processing. Provided the object set index we passed in exists, new dynamic objects will be created and added to the collision system as described by the matrix (the final parameter). These new dynamic objects (one for each mesh contained in the actor) will be added as a single new object set to the collision system and the index of that new set will be returned from the function.

```
            // Create a reference of this objects data in the collision system
            ObjectSetIndex = m_Collision.AddActorReference
                             ( pReferenceObject->m_nObjectSetIndex,
                               pReferenceObject->m_mtxWorld,
                               mtxWorld );
```

If the object we are referencing currently has no animation controller pointer, then it means no object references to this actor currently exist; it is only a single non-reference object. As soon as we add more than one CObject to the scene that uses the same actor, all CObjects that use that actor essentially become references and store their own animation controllers. We covered all of this logic before.

```
            // Is this the first reference?
            if ( !pReferenceObject->m_pAnimController )
            {
                // If this is the first time we've referenced this actor then
                // we need to detach its controller and store in the reference
                pReferenceObject->m_pAnimController =
                                          pReferenceActor->DetachController();

            } // End if first reference
```

```
            ULONG nMaxOutputs, nMaxTracks, nMaxSets, nMaxEvents;

            // Retrieve all the data we need for cloning.
            pController = pReferenceObject->m_pAnimController;
            nMaxOutputs = pController->GetMaxNumAnimationOutputs();
            nMaxTracks  = pController->GetMaxNumTracks();
            nMaxSets    = pController->GetMaxNumAnimationSets();
            nMaxEvents  = pController->GetMaxNumEvents();

            // Clone the animation controller into this new reference
            pController->CloneAnimationController( nMaxOutputs,
                                                   nMaxSets,
                                                   nMaxTracks,
                                                   nMaxEvents,
                                                   &pReferenceController );

        } // End if we found an original object reference.
```

The above section of code shows the conditional that happens when an object was found that uses the actor we wished to load. Outside that conditional code block, we test to make sure we have a valid object set index. If not, then it means that although the actor already exists (there is no need to create a new one, we can just reference it) it has ***not*** yet been assigned to a CObject or registered with the collision system. Therefore we cannot possibly register the actor with the collision system as a reference since it has not been added to the collision system in its non-referenced form. It is the first actor of this type we are registering.

When this is the case, we register the actor with the collision system as a normal actor. Notice that we register the actor with the collision system as a dynamic object group by passing false as the final parameter. Therefore, if this actor contains animation data, our application will be able to animate it and have the collision system respond to it in real time.

```
        // If the collision system could not match this object set index,
        // or we couldn't find an already existing object, add the full blown
        // actor.
        if ( ObjectSetIndex < 0  )
             ObjectSetIndex = m_Collision.AddActor( pReferenceActor,
                                                    mtxWorld,
                                                    false );

    } // End if Actor already exists
```

All the above code is executed if the file name of the external reference we are trying to load has already been loaded for a previous actor. In short, if it has, we decide we never want two copies of the same frame hierarchy in memory, so we reference it both in the scene and in the collision system.

The next section of code shows what happens when the actor has not already been loaded. When this is the case, we must create a new actor, load the X file and register its attribute callback function (CScene::CollectAttributeID). We then load the actor from the X file. The name of the X file we wish to load is the name of the external reference which is now stored in the Buffer array along with the data path.

```
    else

    {
        // Allocate a new Actor for this reference
        CActor * pNewActor = new CActor;
        if (!pNewActor) return false;

        // Load in the externally referenced X File
        pNewActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID,
                                     CollectAttributeID, this );

       HRet = pNewActor->LoadActorFromX(  Buffer,
                                          D3DXMESH_MANAGED,
                                          m_pD3DDevice );

       if ( FAILED(hRet) ) { delete pNewActor; return false; }
```

With the actor now loaded, we make room at the end of the scene's CActor array for another actor pointer and store our current actor. Our code upgrade (in bold) now registers this new actor with the collision system. We also assign the local pRefereneActor pointer to point at this actor so that outside this conditional code block we can use the same pReferenceActor pointer whether a new actor was loaded, or whether it just points to an actor that was previously loaded.

```
        // Store this new Actor
        if ( AddActor( ) < 0 ) { delete pNewActor; return false; }

        m_pActor[ m_nActorCount - 1 ] = pNewActor;

        // Add the physical actor to the collision system
        ObjectSetIndex = m_Collision.AddActor( pNewActor, mtxWorld, false );

        // Store as object reference Actor
        pReferenceActor = pNewActor;

    } // End if Actor doesnt exist.
```

At this point, regardless of whether we created and loaded a new actor or are using one that was already loaded, pReferenceActor will point at it. Let us now create a new CObject to hold this actor. Notice again that we pass the actor pointer into the CObject constructor.

```
    // Now build an object for this Actor (standard identity)
    CObject * pNewObject = new CObject( pReferenceActor );
    if ( !pNewObject ) return false;
```

We also store the world matrix of the reference (passed into the function) and a pointer to the animation controller it will use. Along the way we will store a copy of the object set index that was assigned during actor registration with the collision system. Finally, we add this object to the end of the scene CObject array and return.

```
    // Copy over the specified matrix and store the colldet object set index
    pNewObject->m_mtxWorld        = mtxWorld;
```

```
    pNewObject->m_nObjectSetIndex = ObjectSetIndex;

    // Store the reference animation controller (if any)
    pNewObject->m_pAnimController = pReferenceController;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Success!!
    return true;
}
```

CScene::ProcessEntities

The process entities function has had one line added to it to register any loaded terrains with the collision database. We will not show the entire function but only a subsection of the code that deals with the terrain entity type. The complete loading code for the terrain entity was discussed in the previous lesson. We start at the switch statement used to determine which type of entity we are processing.

```
            switch ( pFileEntity->EntityTypeID )
            {
                  …
                  …
                  Code Snipped here for other entity types
                  …
                  …

                   case CUSTOM_ENTITY_TERRAIN:

                        …
                        … Code snipped here which copies entity info
                        … in to terrain entity structure Terrain

                        // Allocate a new terrain object
                        pNewTerrain = new CTerrain;
                        if ( !pNewTerrain ) break;

                        // Setup the terrain
                        pNewTerrain->SetD3DDevice( m_pD3DDevice, m_bHardwareTnL );
                        pNewTerrain->SetTextureFormat( m_TextureFormats );
                        pNewTerrain->SetRenderMode(GetGameApp()->GetSinglePass());
                        pNewTerrain->SetWorldMatrix
                                    ((D3DXMATRIX&)pFileEntity->ObjectMatrix );
                        pNewTerrain->SetDataPath( m_strDataPath );

                        // Store it
                        m_pTerrain[ m_nTerrainCount - 1 ] = pNewTerrain;

                        // Load the terrain
                        if ( !pNewTerrain->LoadTerrain( &Terrain ) ) return false;

                        // Add to the collision system
```

```
                        m_Collision.AddTerrain( pNewTerrain );

                    } // End if standard terrain

                    break;
```

In the source code shown above, we have also removed all the code that simply copies the terrain data of the terrain from the CFileIWF object into a terrain entity structure. It still exists in the source code but has been removed here for readability.

After doing all of our usual setup for managing the terrain, as a final step, we call the CCollision::AddTerrain method to add a pointer to this terrain object to the end of the collision object's terrain list. As we have seen, the collision system can handle collisions with such terrain objects in a memory efficient way because it does not need to store a complete copy of every terrain triangle.

Updating the Geometry Database at Runtime

You should be well aware by now that the heartbeat of our application is the CGameApp::FrameAdvance function. At a high level, our runtime processing comes down to repeatedly calling this function in a loop. This function controls what happens for every single frame of the game and controls the flow in which events are processed. For example, each time it is called, it instructs the application to process any input that may have been given by the user. It then instructs the scene to apply any animations to any of its objects that it wishes for the current frame update. It also instructs the camera to update its view matrix and finally, it instructs both the scene and the CPlayer objects to render themselves.

Below we see the main snippet of the CGameApp::FrameAdvance function. We have not shown the code that tests for and recovers lost devices or the code that presents the back buffer. Instead we see all of the function calls made to transform and render the scene and the order in which states are updated in a given iteration of the game loop.

```
    // Poll & Process input devices
    ProcessInput();

    // Animate the scene objects
    m_Scene.AnimateObjects( m_Timer );

    // Update the Player ( Used to be in ProcessInput )
    m_Player.Update( m_Timer.GetTimeElapsed() );

    // Update the device matrix
    m_pCamera->UpdateRenderView( m_pD3DDevice );

    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear(0,NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x79D3FF,
                        1.0f, 0 );

    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();
```

```
    // Render the scene
    m_Scene.Render( *m_pCamera );

    // End Scene Rendering
    m_pD3DDevice->EndScene();
```

Notice in the above code snippet from Lab Project 13.1 that we now have a CPlayer::Update call after the call to AnimateObjects. This function call used to update the position of the player was originally called directly from the ProcessInput function. We will discuss why we have moved it out of that function and into the FrameAdvance function shortly.

The above code clearly shows that the CScene::AnimateObjects function is called during every iteration of the game loop. This gives the scene a chance to update the matrices of any scene objects it wishes to animate. It also allows the scene object to advance animation controllers of any actors currently participating in the scene. We need to revisit this function so that we can add the code that will notify the collision system when the scene geometry has been animated.

CScene::AnimateObjects

In this lesson we will have to inform the collision system when any objects that have been registered as dynamic objects with the collision system have been animated.

The function first loops through every CObject in the scene's CObject array. In Lab Project 13.1, we only animate actors, so we are not interested in finding any objects in this function which contain only a single CTriMesh. You can see that at the start of the object loop, we skip the current object if it does not have a valid CActor pointer.

```
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;
```

We next test to see if the object we are about to animate has a valid animation controller pointer. If the pointer is NULL, then it means either the actor has no animation to play, or this is the only object that references the actor and therefore the animation controller is owned by the actor itself. If the object does have a valid animation controller pointer, then there are multiple objects that reference this object's actor. This means we must attach the animation controller of this reference object to the actor so that we can animate the hierarchy using the reference's animation data. Once we attach the controller to the

actor, we advance its timeline. This will cause the parent relative frame matrices of the actor's hierarchy to be rebuilt in the pose described by the reference's animation controller. The parent relative matrices of the hierarchy will now describe the actor exactly as it should look for this particular reference.

```
        if ( pObject->m_pAnimController )
            pActor->AttachController( pObject->m_pAnimController, false );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false );
```

Next we get the object set index assigned by the collision system when it was registered during the CScene::ProcessReference function. We set the actor's world matrix and pass true to the CActor::SetWorldMatrix function which forces all the absolute matrices of the actor to be rebuilt in their correct world space positions. Remember, it is these absolute matrices that the dynamic objects in the collision system point to. We must remember to update the world space matrices *before* instructing the collision system to update the status of the object group.

```
        //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

}
```

Once the world matrices of the actor have been built, we call the CCollision::ObjectSetUpdated function, passing in the object set index for the reference. This will instruct the collision system to search for all dynamic objects that were spawned from this actor and recalculate their collision and velocity matrices based on the updated world space frame matrices.

Collision Geometry Management – Final Note

We have now covered everything we need to know about registering scene geometry of different types with our collision system. We have seen how to load and register static meshes, actor references, and terrain entities with the collision system. We have also talked about how we should correctly manage updates for scene geometry that is registered as dynamic.

What we have not yet discussed are the actual moving entities that will use our collision system to collide and slide in the environment. In Lab Project 13.1, we will use the CPlayer object as the moving

entity. It will support both a first and third person camera. The mesh of the player in first person mode will be our U.S. Army Ranger from the prior lessons on skinning.

By attaching the skinned mesh to the CPlayer object, we can move it about the scene and watch it collide and slide in the environment. However, our CPlayer object has always been a little rudimentary when it comes to its physics handling. Now we will need to pay a little more attention to these details because we want to implement player movement that is typical in a first/third person game. Therefore, in the next section, we will discuss the changes to our player class to see how some simple improvements to our physics model will provide better environment interaction in our collision system.

> **Note**: Our intent is not to design a proper physics engine in this course. That would be the subject of an entire course by itself. We are simply trying to provide a more realistic feel for our player as it navigates throughout the world. Although the system we use is based on some laws of Classical Mechanics, it is still going to be a very rudimentary system. Learning how to create an actual physics engine with support for rigid bodies, vehicles, and the like, falls into the domain of the Game Physics course offered here at Game Institute.

The Revised CPlayer Class

Our previous CPlayer class was pretty straightforward and much of the relationship between the CGameApp object and the CPlayer object will remain intact. However, the CPlayer::Update function, which was responsible for calculating the player velocity will be totally rewritten. For one thing, it will use more appropriate physics calculations. It will also issue an update request to the collision system via a CScene callback function.

Let us quickly recap the way things used to work so that we get a better idea for what has to change.

- The CGameApp::FrameAdvance function would call the CGameApp::ProcessInput function with each iteration of the game loop.

- The ProcessInput function would read the state of the keyboard to combine a number of CPlayer flags describing the user direction request. The mouse would also be read to determine if the user wished to rotate the view.

- If the player tried to move, the ProcessInput function would call the CPlayer::Move function passing in the direction(s) the player should move (using a combination of flags) and the distance to move. The CPlayer::Move function would build a vector pointing in the requested direction with the requested length and add it to the current velocity vector (maintained internally). This updated velocity vector would not yet be used to update the position of both the player object itself and its attached camera. That happened later. All we have done this point is added a force to the velocity vector.

- If the ProcessInput function determined that the user also wished to rotate the player object (via mouse input), it would call the CPlayer::Rotate function to perform a local rotation to the CPlayer's world matrix.

- At this point in the ProcessInput function, the player object has been potentially rotated and its velocity vector has been updated, but the position has not yet been altered based on that velocity. The ProcessInput function would finally call the CPlayer::Update function before returning to apply the movement described by the velocity vector that had just been updated.

- The CPlayer::Update function is the one that will require the most changes in this new version. Previously, it added a gravity vector to the player's velocity vector so that a constant downwards force was applied to the player each frame. This would make sure that if the player had no terrain underneath him (the only type of collidable geometry we supported), they would fall downwards. We would also downscale the velocity vector of the player each frame based on a friction constant. By shortening the length of the velocity vector based on friction each time this function is called, we allowed our player to slow to a halt when the user released the movement keys instead of carrying on forever, as would be the case in a frictionless environment. The larger the value we set for the player object's friction, the more suddenly it would stop when no movement was being applied by the user. This is because the velocity vector will be shortened the next time the update function is called as long as the user does not press another movement key.

- The CPlayer::Update function would update the actual position of the player using the velocity vector. This would physically move the player object in the world.

- Keep in mind that the player object may have updated its position with respect to another scene object. You will recall that the CPlayer object maintained an array of callback functions to help in this regard. In previous applications, the CTerrain class registered a callback function with the player. This callback was invoked from the CPlayer::Update function after the position of the player had been updated. The callback function had a chance to examine the new position of the player and modify it if it finds it improper. In our previous applications, the terrain callback function would test the position of the player against the terrain geometry. If it found that the position of the player had been moved below the height of the terrain at that location, it would modify the height of the player's position so that it sat on top of the terrain. This is what prevented our simple gravity model from pushing the player through the terrain. This callback function will now be replaced with a CScene callback function which will use our new collision system.

- As the CPlayer::Update function also updated the position of the camera, it also instructed the camera to call any callback functions which have been registered for it. This allowed the same collision detection function to be used to modify the position of the player and its attached camera before returning.

- The CGameApp::FrameAdvance function would also call the CPlayer::Render function each frame to instruct the player to render any attached mesh (such as the third person mesh).

Our CPlayer object will now have the following member variables. Not all of them are new to us since our player class has always maintained a velocity vector, a gravity vector, and a scalar used to store the amount of drag to apply to the camera position when tracking the player in third person mode. Some of the other members shown below will be new and their usefulness will be described in this section.

Except from CPlayer.h (Lab 13.1)

```
    // Force / Player Update Variables
    D3DXVECTOR3     m_vecVelocity;          // Movement velocity vector
    D3DXVECTOR3     m_vecAppliedForce;      // Our motor force
    D3DXVECTOR3     m_vecGravity;           // Gravity vector
    float           m_fCameraLag;           // Amount of camera lag in seconds
    float           m_fTraction;            // How much traction we can apply
    float           m_fAirResistance;       // Air resistance coefficient.
    float           m_fSurfaceFriction;     // Fake Surface friction scalar
    float           m_fMass;                // Mass of player
```

There are also some simple inline public member functions to set the above properties of the player.

```
void SetGravity  ( const D3DXVECTOR3& Gravity  ) { m_vecGravity = Gravity; }
void SetVelocity ( const D3DXVECTOR3& Velocity ) { m_vecVelocity = Velocity; }
void SetCamLag   ( float CamLag )                { m_fCameraLag = CamLag; }
void SetTraction ( float Traction              ) { m_fTraction = Traction; }
void SetSurfaceFriction ( float Friction       ) { m_fSurfaceFriction = Friction; }
void SetAirResistance   ( float Resistance     ) { m_fAirResistance = Resistance; }
void SetMass            ( float Mass           ) { m_fMass = Mass; }
```

Useful Concepts in Classical Dynamics

Before we start examining the implementation of our application's physics model, let us begin with a very high level overview of some of the physics concepts we would like to consider as we put together our new system. To be sure, this will be a very quick introduction to Classical Dynamics. For a much more complete discussion, it is highly recommended that you take the Game Physics course offered here at the Game Institute.

Newton's Laws of Motion

We will start with some very basic ideas: Newton's Three Laws of Motion.

First Law: *When the net force on an object is zero, the motion of an object will be unchanged. An object at rest will remain so, unless compelled to change because some amount of force is applied.*

The first law basically tells us that if an object is moving with some direction and magnitude (i.e., velocity), it will continue moving with that velocity unless some outside force acts on it causing some change.

Second Law: $\sum \boldsymbol{F} = m\boldsymbol{a}$

The second law talks about what happens to an object when forces are applied. As we can see, the law gives us a relationship between the net forces acting on the object, the object's acceleration, and the mass of that object. In other words, using this law, we can eventually figure out how fast an object will go based on how much mass it has and how much force is applied to it.

Note that in the above formula, force and acceleration are both vector quantities. Thus, they act along all three axes in the case of a three dimensional system.

It is also worth noting that the unit of measurement for force is called a newton (1 kg * m / s^2), often given by the letter N. The units of measurement might seem a little strange at first due to the s^2 concept in the denominator. But this makes more sense when you recall the relationship between position, velocity, and acceleration. We know that velocity represents a change in position with respect to time. This basically tells us how fast we are going (our speed).

$$\mathbf{v} = \Delta\mathbf{p} / \Delta t$$

Acceleration tells us how much our velocity is changing with respect to time.

$$\mathbf{a} = \Delta\mathbf{v} / \Delta t$$

This is where we see our unit for acceleration come into play. If we substitute in our equation for velocity, we get:

$$\mathbf{a} = (\Delta\mathbf{p} / \Delta t) / \Delta t$$

$$\mathbf{a} = (\Delta\mathbf{p} / \Delta t^2)$$

Since position is measured in meters and time is measures in seconds, we get our acceleration units of meters per second per second (or, given the rules of fractional division: m / s^2). When we factor in mass, which is measured in kilograms, we wind up with our newton as described above. Thus, one newton is the amount of force required to give an object with a mass of 1 kilogram an acceleration of 1 meter per second per second.

Third Law: *For every action, there exists an equal and opposite reaction.*

The third law gives us a relation between the forces that exist between two interacting bodies. What it basically states is that when an object A applies a force to another object B, object B is applying the same force to object A, just in the opposite direction. Mathematically, we can state this relationship as:

$$\mathbf{F}_B = -\mathbf{F}_A \quad \text{or} \quad \mathbf{F}_A + \mathbf{F}_B = 0$$

Thus when I am in contact with the ground, while gravity might force me downwards towards the center of the earth, the ground exerts an opposite upwards force that is equal and opposite and I remain in place on the surface. I do not get shoved through the surface, nor do I hover above the ground. I am in equilibrium when I am in contact with the surface.

Contact Forces

Now let us talk a little bit about the forces that come into play when objects are moving about in the environment.

On Earth, all solid bodies experience resistance to motion. Whether they are sliding on solid surfaces, rolling along on those surfaces, or moving through a liquid or gas, some amount of resistance will be given. For our purposes in this lesson, we will consider the general concept of resistance as falling within the domain of *friction*.

Friction forces depend on the types of surface that are in contact. Generally, the rougher the surface, the greater the friction. We can generally break down the concept of friction into two categories: *static friction* and *dynamic* (or *sliding*) *friction*. Static friction is essentially the amount of force that must be overcome before an object begins to slide. Once an object is in sliding, dynamic friction comes into play. Dynamic friction tells us how much force needs to be overcome in order to keep our object sliding along on that surface.

Consider the example of an automobile. There exists a certain amount of static friction between the rubber of the car's tire and the asphalt of the road at the point where the two come into contact. As long as the forces that are being applied to the tire (and thus the contact point) do not exceed this static friction threshold, the car tire is able to grip the road and propel the car forward. As the tire spins, it pushes down on the road, and because of static friction, the road pushes back on the tire (see Newton's Third Law) and the car continues it forward motion. But if the car were to suddenly hit a patch of ice, where the static friction between rubber and ice is significantly lower, the static friction hurdle would be much easier to overcome. If the forces were such that they exceeded the static friction threshold, the tire

would begin to slide. Although the tire is still spinning, it is not be able to get a good purchase on the surface and suddenly dynamic friction (sliding friction) comes into play.

Friction is ultimately proportional to the total amount of force pressing the objects together. In most cases, this will be the force due to gravity (a downwards acceleration) which we can describe as:

$$\mathbf{F}_G = m\mathbf{g}$$

Where m is the mass of the object and **g** is the gravitational acceleration (9.8 m/s^2). Note the use of Newton's Second Law as a means to describe gravity.

According to Newton's Third Law there must be a net equal and opposite force that counters this one. That is, if gravity forces a body downwards, assuming the body is in contact with a surface, there must be a force pressing back upwards on the bottom of the object. There is indeed such a force, and it is generally referred to as the *normal force* (**N**). The normal force is directed based on the orientation of the surface.

$$\mathbf{N} = -m\mathbf{g}$$

The proportionality between friction and the normal force can be expressed by a constant which can be introduced as a coefficient. This coefficient is represented by the Greek letter mu (μ). Thus:

$$\mathbf{F}_{(\text{sliding friction})} = \mu\mathbf{N}$$

We can substitute in our gravity values for the normal force and rewrite the equation as:

$$\mathbf{F} = -\mu\, m\mathbf{g}$$

In the case of static friction, we can describe this force using the formula:

$$\mathbf{F}_s \leq \mu_s\, m\mathbf{g}$$

μ_s is the coefficient of static friction that is associated with the two types of materials in contact. In other words, the static friction force remains below a certain threshold given by the normal force scaled by some constant.

Dynamic friction can be described by the very similar formula

$$\mathbf{F}_D = \mu_d\, m\mathbf{g}$$

Once again, we see the relationship with the normal force scaled by some constant. The following table provides some friction coefficient values that have been determined by experiment for various materials.

Material 1	Material 2	Static μ_s	Dynamic μ_d
Steel	Steel	0.74	0.57
Aluminum	Steel	0.61	0.47
Copper	Steel	0.53	0.36
Brass	Steel	0.51	0.44
Zinc	Cast Iron	0.85	0.21
Copper	Cast Iron	1.05	0.29
Glass	Glass	0.94	0.4
Rubber	Concrete (dry)	1	0.8
Rubber	Concrete (wet)	0.3	0.25

Not surprisingly, the coefficients of static friction turn out to be higher than those for dynamic friction. As you own experience tells you, it is generally easier to slide a heavy box along your kitchen floor after you have "broken it loose" from the grip of static friction.

> **Note:** The forces that exist when two bodies are touching are called *contact forces*. Thus friction and the normal force are both contact forces. The normal force is a perpendicular force that the surface exerts on the body it is in contact with. The friction force is a parallel force that exists in the direction of motion on the surface.

We used the example of an automobile earlier to introduce the concepts of static and dynamic friction. In practice, for objects that are rolling, a third type of frictional coefficient can be introduced called the *coefficient of rolling friction* (μ_r). This is sometime referred to as *tractive resistance*. The formula is pretty much the same as we see in all of our friction cases:

$$\mathbf{F} = \mu_r \mathbf{N}$$

For steel wheels on steel rails, μ_r generally equals 0.002. For rubber tires on concrete, μ_r equals ~0.2. Thus we see that the steel wheel/rail case experiences less resistance to movement than does the rubber/concrete case. This actually gives some indication as to why trains are typically more fuel efficient than automobiles.

> **Note:** We often hear the term *traction* used to describe this friction relationship as the amount of 'grip' that a wheel or tire has on the surface. The car tire on the asphalt surface has very good traction (a good grip), while the same tire on a sheet of ice has very poor traction (possibly no grip at all).

Fluid Resistance

We now understand that all bodies on Earth experience resistance to motion and we have seen friction at work with respect to movement on solid surfaces. But what about fluids (liquids and gases)? Certainly we all recognize from experience that moving through a pool of water is generally much more difficult than walking down the street. So there is obviously some manner of resistance happening there. Is this friction as well? Yes indeed.

Viscosity is a term used to describe the thickness of a fluid. Thicker fluids have higher viscosity and vice versa. Objects moving through high viscosity fluids will experience a greater amount of friction than similar movement through low viscosity fluids. However, as it turns out, when an object moves through a viscous fluid (including air), the resistance force also takes into account the speed of the object. To be fair, this will actually vary from one situation to the next and we are going to oversimplify things here a little bit, but in general, for slow moving objects we find that:

$$\mathbf{F}_R = -k\mathbf{v}$$

Where **v** is the velocity of the object and *k* is a constant (called the *coefficient of frictional drag*) that varies according to the size and shape of the body, as well as other factors having to do with the viscosity of the fluid.

For objects moving at higher rate of speed the formula becomes:

$$\mathbf{F}_R = -k\mathbf{v}^2$$

So as the speed of the object increases, so does the amount of resistance forces exerted on it by the fluid. This is the result of turbulence in the fluid due to the rapid movement of the object. The result is a higher level of friction between the moving object and the fluid.

If you were modeling slow moving vehicles in your game (cars, ships, etc.) then you would preferably use the first formula to model fluid resistance (often called *aerodynamic drag*, or just *drag*). If you were modeling high speed moving objects (racing cars, jet-skis, etc.) then the second formula would produce more accurate results.

Updating the CPlayer Class

At this point we have now covered all of the topics we will need to know about when we are updating our application's physics model. To be clear right upfront, we are *not* going to model precision physics that obey all of the formulas we have just discussed. Often it is not just practical or necessary to do so when working on a game. However, we will absolutely use the ideas that were presented as a basis for trying to simulate behavior that we find satisfying in our particular demo. We will attempt to maintain the spirit of a particular concept, but not necessarily follow the specific rules and equation. As we introduce concepts in our model, we will talk about how they relate to the theory just presented and, when we make some alterations of our own, explain why we did so.

While our new CPlayer object will work in a very similar manner to its prior incarnation, we will need to alter the CPlayer::Update function. But before we look at the code to the new CPlayer object, let us talk a little bit about the model on a high level and look at some of the member variables that were introduced to support that model. This should help set the stage for what is to come on the coding side.

Motor Force and Tractive Force (m_vecAppliedForce / m_fTraction)

Motor force is the amount of force that is applied to an object to make it move. If we think about the soles of a person's shoes in contact with the ground, when we apply motor force to the leg, that leg pushes on the ground with an applied force and in response, the ground pushes back in the opposite direction with its own force, essentially propelling the shoe (and thus the person) forward. We talked about some of these forces in the last section. For example, we know about the force pushing upwards from the ground (we called it the normal force) and we also know that some degree of resistance comes into play (friction). Our model will essentially take some liberties and combine these concepts together and we will call the resulting force the *tractive force*. That is, this tractive force will be the combination of the applied motor force(s) and the normal force (technically, the inverted gravitational force), with a dash of friction thrown in for good measure. We will discuss the friction model in more detail later. For now, the best way to understand our tractive force as it relates to friction is to think about the concept of traction introduced earlier. That is, we are going to say that there is a 'grip' factor that comes into play with respect to our surfaces as we attempt to apply accelerating forces. This will prove to be a very helpful concept when we model jumping a bit later on.

The amount of tractive force produced will depend on a traction coefficient for the surface (m_fTraction in our model). We can think of the traction coefficient as describing how good a grip the player's shoe will have with the surface and we will use a value in the range [0.0, 1.0]. The closer to 1.0 the traction coefficient is, the better the grip of the shoe on the surface. Since we will use this coefficient to scale the motor force, larger coefficient values mean the closer to the initially applied motor force the tractive force applied to the object will ultimately be. We can think of the motor force as being the force describing how much we wish to move, while the tractive force describes the amount we will actually move, once slippage between the surface and the object is taken into account. Again, gravity will play a role as well, but we will talk more about that shortly.

So our traction coefficient is actually going to act very much like a frictional coefficient in that it will scale the resulting force based on the properties of the contact surfaces. Consider an Olympic athlete's running shoes, which are specifically designed to get as near to perfect traction with the running track as possible. If we imagine the same athlete running on ice, the same amount of motor force would still be getting applied by the athlete, but because of the lack of grip between his/her shoes and the ice, the amount of tractive force actually pushing the athlete forward is going to be much less, and as such, slower running speeds are observed.

So a traction coefficient for an interacting surface/object pair of 0.5 would describe quite a slippery situation, since only half the applied motor force will be applied to the object in the forward direction.

Traction = 0.5

Let us also assume that we wish to apply a motor force of 20 to the object to move it along its velocity vector.

Applied Force = 20N

The amount of force that actually gets applied (the *tractive force*) is calculated as follows:

Tractive Force = Applied Force * Traction
= 20N * 0.5
= 10N

Again, for the moment, we are not including gravity in our model. We will address that a bit later.

In this example, you can see that although we applied a force of 20N to the object, the actual forwards force applied to the object was only half of that because of the lack of grip. As mentioned earlier, this is why traction is so important between the wheels of a racing car and the track on which it is racing. Racing cars often use tires that do not have treads in order to allow more rubber to come into contact with the surface of the track and provide a better grip for high speed racing.

Although the relationship between the player and the various surfaces in the level in real life would introduce a lot of different traction coefficients between surface types, we will simplify and assign our CPlayer object a single traction coefficient. This means, the traction between the player object and every surface in our level will be considered to be exactly the same. Obviously this could easily be extended so that a traction coefficient could be assigned at the per polygon level. That way, you could assign a polygon with an ice texture map a low traction of 0.2 but assign a much higher traction coefficient to a polygon with an asphalt texture mapped to it. In this demo, we will use a single traction coefficient for a given player object which will be set and stored in the CPlayer::m_fTraction variable shown above. The traction coefficient used by the player for its physics calculations can be set by the application using the CPlayer::SetTraction method.

Note that we will also be modeling dynamic friction, so you might wonder why we would need this traction concept if we intended to use proper friction anyway. As it turns out, in actual practice, our traction coefficient will almost always equal 1.0. The only time this changes (and drops to nearly zero) is when the player is no longer in contact with a surface (i.e., they are in the air). The idea was to give the player a little bit of ability to control the player while they are in mid-air. Since a person cannot actually walk on air, the traction concept gives them just a little bit control that they would not otherwise have using a standard model. This type of mid-air control is fairly common in games, so we decided to include it in our demonstration as well.

Applying motor force to our CPlayer object will be the primary means by which the application controls the speed of its movement. The CPlayer object will now have an ApplyForce method which allows the application to apply a motor force to the player. You will see later when we revisit the code to the CGameApp::ProcessInput function, that in response to keys being pressed by the user, we no longer call the CPlayer::Move function. Instead, we will call the CPlayer::ApplyForce method which adds motor force to the object. The more motor force we apply, the faster our object will move (assuming that our traction coefficient is not set to zero).

The CPlayer::ApplyForce function is shown below:

```
void CPlayer::ApplyForce( ULONG Direction, float Force )
{
    D3DXVECTOR3 vecShift = D3DXVECTOR3( 0, 0, 0 );

    // Which direction are we moving ?
```

```
    if ( Direction & DIR_FORWARD  ) vecShift += m_vecLook;
    if ( Direction & DIR_BACKWARD ) vecShift -= m_vecLook;
    if ( Direction & DIR_RIGHT    ) vecShift += m_vecRight;
    if ( Direction & DIR_LEFT     ) vecShift -= m_vecRight;
    if ( Direction & DIR_UP       ) vecShift += m_vecUp;
    if ( Direction & DIR_DOWN     ) vecShift -= m_vecUp;

    m_vecAppliedForce += vecShift * Force;
}
```

As you can see, ApplyForce accepts two parameters. The first is a 32-bit integer containing a combination of one or more flags describing the direction we wish to move the object. Remember, this function is called from the CGameApp::ProcessInput function, so the direction flags that are set correspond precisely to the direction keys being pressed by the user.

> **Note:** The movement flags such as DIR_FORWARD and DIR_UP are members of the Direction enumerated type that has been part of the CPlayer namespace since Chapter 4.

The function calculates a shift vector which describes the direction we wish to move the object. For example, if the forward key is pressed, then the player object's look vector is added to the shift vector. If both the up and forwards keys are pressed, then the shift vector will be a combination of the player object's up and look vectors, and so on. The result is a unit length shift vector describing the direction we wish to move the player. We then scale this vector by the passed motor force parameter before adding it to the m_vecAppliedForce member variable. Since forces are vector quantities that have both a direction and magnitude, you can see that in this particular case, we wind up with the proper result (just split over two logical code sections for ease of implementation, where the magnitude is passed in separately).

When we multiply the unit length direction vector by the motor force scalar in the above code, we generate a vector whose direction represents the direction we wish to move and whose length represents the amount of force we wish to apply to the object to move it in that direction (the motor force). Notice that we do not assign this value to the m_vecAppliedForce member variable but instead add it. Why?

The m_vecAppliedForce vector describes the direction and amount we wish to move. It contains the motor force that has been applied since the last frame. This is very important because you will see in a moment that this member is always set to a zero vector again after each CPlayer::Update call. Therefore, if it is always zero at the start of each iteration of our game loop, why are we not assigning (vecShift * Force) instead of adding it? The answer is that other places in the application may wish to apply forces to the player object in a given iteration of the game loop. As such, we will collect all forces that have been applied in this update. When we do eventually call the CPlayer::Update method to apply m_vecAppliedForce and generate the player's current velocity vector and calculate a new position, it will contain the combination of all forces applied since the last update.

We still have a ways to go yet, but now that we have seen the new ApplyForce method, let us quickly revisit the CGameApp::ProcessInput function. This is called at the start of the update process for each frame. It is called by the CGameApp::FrameAdvance function.

CGameApp::ProcessInput

Most of this function is unaltered so we will quickly skip past the bits we have already discussed in previous lessons.

```
void CGameApp::ProcessInput( )
{
    static UCHAR pKeyBuffer[ 256 ];
    ULONG        Direction = 0;
    POINT        CursorPos;
    float        X = 0.0f, Y = 0.0f;

    // Retrieve keyboard state
    if ( !GetKeyboardState( pKeyBuffer ) ) return;

    // Check the relevant keys
    if ( pKeyBuffer[ VK_UP    ] & 0xF0 ) Direction |= CPlayer::DIR_FORWARD;
    if ( pKeyBuffer[ VK_DOWN  ] & 0xF0 ) Direction |= CPlayer::DIR_BACKWARD;
    if ( pKeyBuffer[ VK_LEFT  ] & 0xF0 ) Direction |= CPlayer::DIR_LEFT;
    if ( pKeyBuffer[ VK_RIGHT ] & 0xF0 ) Direction |= CPlayer::DIR_RIGHT;
    if ( pKeyBuffer[ VK_PRIOR ] & 0xF0 ) Direction |= CPlayer::DIR_UP;
    if ( pKeyBuffer[ VK_NEXT  ] & 0xF0 ) Direction |= CPlayer::DIR_DOWN;
```

The first section of code (shown above) uses the GetKeyboardState function to read the state of each key into the local 256 byte buffer. The state of each key we are interested in (Up, Down, Left, Right, PageUp and PageDown) is checked to see if it is depressed. If so, the corresponding CPlayer::Direction flags are combined into the 32-bit variable Direction. At this point the Direction variable has a bit set for each direction key that we pressed.

Next we see if the mouse button is currently being held down, in which case our application window needs to be managing the capture of mouse input. If so, we get the current position of the mouse cursor and subtract from it the previous position of the mouse cursor for both the X and Y axes. Dividing these results by 3 (arrived at by trial and error) gives us our pitch and yaw rotation angles. We then reset the mouse cursor back to its previous position so that it always stays at the center of the screen. We must do this because if we allowed the cursor to physically move to the edges of the screen, the player would no longer be able to rotate in that direction when the operating system clamps the mouse movement to the screen edges.

```
    // Now process the mouse (if the button is pressed)
    if ( GetCapture() == m_hWnd )
    {
        // Hide the mouse pointer
        SetCursor( NULL );

        // Retrieve the cursor position
        GetCursorPos( &CursorPos );

        // Calculate mouse rotational values
        X = (float)(CursorPos.x - m_OldCursorPos.x) / 3.0f;
        Y = (float)(CursorPos.y - m_OldCursorPos.y) / 3.0f;
```

```
        // Reset our cursor position so we can keep going forever :)
        SetCursorPos( m_OldCursorPos.x, m_OldCursorPos.y );

    } // End if Captured
```

The next section of code is only executed if either the player needs to be moved or rotated. If the X and Y variables are not equal to zero then they describe the amount of pitch and yaw rotation that should be applied to the player. First we will deal with the rotation.

If the right mouse button is being held down then the left and right mouse movement should apply rotations around the player object's local Z axis (roll). Otherwise, the left and right movement should apply rotations around the player object's Y axis (yaw). In both cases, the backwards and forwards movement of the mouse will apply rotation about the player's X axis (pitch).

```
    // Update if we have moved
    if ( Direction > 0 || X != 0.0f || Y != 0.0f )
    {
        // Rotate our camera
        if ( X || Y )
        {
            // Are they holding the right mouse button ?
            if ( pKeyBuffer[ VK_RBUTTON ] & 0xF0 )
                m_Player.Rotate( Y, 0.0f, -X );
            else
                m_Player.Rotate( Y, X, 0.0f );

        } // End if any rotation
```

At this point we will have rotated the player. Now let us handle the movement. If any movement flags are set in the Direction variable then we need to apply some motor force to the player. We apply a motor force magnitude of 600 in this application but you can change this as you see fit. If the shift key is depressed, we set the applied motor force magnitude to 1000, which allows the shift key to enable a run mode for the player that moves it more quickly. We then call the CPlayer::ApplyForce function to add this motor force to the player.

```
        // Any Movement ?
        if ( Direction )
        {
            // Apply a force to the player.
            float fForce = 600.0f;
            if ( pKeyBuffer[ VK_SHIFT ] & 0xF0 ) fForce = 1000.0f;
            m_Player.ApplyForce( Direction, fForce );

        } // End if any movement

    } // End if camera moved
```

Remember that at this point the player has not had its position updated; we have simply added the force we have just calculated to its m_vecAppliedForce vector. The actual position change of the player happens in the CPlayer::Update method, which used to be called at the end of this function. This will

actually no longer be the case, since we have moved the CPlayer::Update call out of this function and into CGameApp::FrameAdvance just after the call to the CScene::AnimateObjects function.

We did this because, now that our scene geometry may be moving and the CPlayer::Update function is responsible for invoking the collision system and calculating the new position of the player, this must be called after the scene objects are updated. This will ensure that we are colliding with the position of these scene objects in their current state (instead of the previous frame state). In short, we should move the scene objects first and then try to move our CPlayer, and not the other way around. We will look at the CPlayer::Update method shortly.

The next section of code is new to the ProcessInput function and handles the reaction to the depression of the control key by the user. In this lab project, the control key will allow the player to jump up in the air. This is easily accomplished by getting the current velocity vector of the player and simply adding a value of 300 to its Y component. The code is shown below and uses two new functions exposed by CPlayer in this lab project.

```
    // Jump pressed?
    if ( m_Player.GetOnFloor() && pKeyBuffer[ VK_CONTROL ] & 0xF0 )
    {
        D3DXVECTOR3 Velocity = m_Player.GetVelocity();
        Velocity.y += 300.0f;
        m_Player.SetVelocity( Velocity );
        m_Player.SetOnFloor( false );

    } // End if Jumping Key
```

While our player is now allowed to jump, it must only be allowed to do so if its feet are on the ground. The control key should be ignored if the player is not in contact with the ground since there is no surface for their feet to push off from and thus no possible way to apply an upwards motor force. Later we will see how our traction coefficient allows for a little extra control when the player has jumped into the air.

In the above code you will notice that CPlayer now exposes a function called GetOnFloor that will return true if the player's feet are considered to be in contact with the ground. Do not worry about how this function works for the time being, as it is closely coupled with the collision detection update. For now, just know that if it returns true, our player is on the ground and should be allowed to jump. Notice that after we adjust the velocity of the player to launch our player into the air, we use the CPlayer::SetOnFloor function to inform the player object that its feet are no longer on the ground. This function will be discussed in a little while as well.

> **Note:** While you might assume that we would have modeled jumping as an upwards motor force, it turns out that it really is not worth the effort to go to so much trouble. Adjusting the velocity vector directly works just fine for our purposes and is certainly the simplest way to accomplish our objective.

Finally, if the 0 key on the num pad is pressed we adjust the player's camera offset vector to show the front view of the mesh (3rd person mode only).

```
    // Switch third person camera to front view if available.
    if ( m_pCamera && m_pCamera->GetCameraMode() == CCamera::MODE_THIRDPERSON )
    {
```

```
        if ( pKeyBuffer[VK_NUMPAD0] & 0xF0 )
            m_Player.SetCamOffset( D3DXVECTOR3( 0.0f, 26.0f, 55.0f ) );
        else
            m_Player.SetCamOffset( D3DXVECTOR3( 0.0f, 26.0f, -35.0f ) );

    } // End if
}
```

And there we have our new ProcessInput function. We have shown how it applies motor forces to the player object in response to user input. Again, note that at the end of this function, motor force has been applied to the player but no position update has been applied yet.

Introducing Resistance

We have now discussed what motor force is and how it is applied in the new CPlayer system. We have also mentioned that the amount of force we actually apply to the object to propel it forward is equal to the tractive force. The tractive force is the motor force scaled by the traction coefficient between the object and the surface (plus the addition of gravity as we will look at shortly).

But these are not the only forces we will have to apply to our object in order to calculate its new position in the CPlayer::Update function. If we were to simply add a constant tractive force to our player in each update call, our object would eventually accelerate to infinite speeds. Just think about it; if the velocity vector of the object is not diminished based on some concept like friction or drag, every time we applied a force, the velocity vector would get longer and longer. Since the (length of the) velocity vector is commonly referred to as the speed of the object, we know that our speed would increase with each update, forever. This makes perfect sense of course since we know from Newton's Second Law of Motion that

$$\mathbf{F} = m\mathbf{a}$$

If we rearrange this equation to look at the acceleration, we see that:

$$\mathbf{F}/m = \mathbf{a}$$

What this tells us is that our acceleration is always going to equal the total force applied to the object scaled by the inverse of the object's mass. Since the mass is a constant, you can see that if we assumed that mass equals 1 (for simplicity) our acceleration will equal our total applied forces. If we keep increasing our force, our acceleration will continue to increase. Since acceleration is simply the change in velocity with respect to time:

$$\mathbf{a} = \Delta\mathbf{v} / \Delta t$$

we know that over time, our velocity will simply get larger and larger (i.e., the speed increases because the object continues to accelerate).

Clearly this is not the case in real life. Just as we have tractive force propelling the object forward, we also experience forces between the surface and the moving object which counteracts the tractive force we are trying to apply to varying degrees. As we discussed earlier, these forces fall under the category of resistance forces (or contact forces for object in contact). We learned that on Earth, all solid bodies experience resistance to motion. Whether they are sliding on solid surfaces, rolling along, or moving through a liquid or gas, some amount of resistance is to be expected. We looked at multiple types of friction and also talked about viscous drag. Now let us look at how these ideas will apply in our player physics model.

Surface Friction – m_fSurfaceFriction

As mentioned earlier, friction forces depend on the types of surface that are in contact and we can generally break down the concept of friction into two categories: static friction and dynamic (or sliding) friction. In our demonstration, rather than deal with multiple friction models and how they influence forces, we are simply going to focus on two high level ideas – forces that speed us up (positive forces) and those that slow us down (negative forces). The motor forces and gravitational force fall into the category of positive forces because they will act to speed us up. Friction and drag are going to fall into the category of negative forces because they will serve to slow us down.

It is worth noting that we are not going to bother modeling static friction in our demonstration. Our goal is to slide our player on command based on user input and we decided not to worry about overcoming a friction threshold before we can begin movement. After all, while we may be sliding a sphere around the environment in our code, we are assuming our player is walking, not sliding around in the world. So we need to take some liberties with the theoretical models and focus on what works for us in this application. In a sense we are trying to model what is a nearly frictionless environment so that we can get smooth rapid movement, but be sure to include just enough dynamic friction to be able to slow us down (and maybe even do so differently depending on the surface material properties when desired).

To be fair, there is a downside to not modeling static friction which comes into play on sloped surfaces - due to the force of gravity, a motionless player on an inclined plane will slide downwards. This is not a terribly difficult problem to solve and there are a number of ways to tackle it should you decide you want static friction in your game. Keep in mind that should you decide to model actual static friction, calculating the normal force can be a little tricky. While you do get back the normals of the colliding triangles from the collision system, do not forget that you may have cases where you are intersecting more than one surface simultaneously. Averaging surface normals might be one way to tackle this, but in truth, the results are not very dependable. Rather than model pure static friction, you could simulate it far more cheaply by simply zeroing out your velocity vector (when you are not moving of course) when the surface normal is less than some particular inclination.

In this application we will stick to modeling dynamic friction only, since our primary movement model is based on sliding anyway. Our equation for dynamic friction was as follows:

$$\mathbf{F} = -\mu\, m\mathbf{g}$$

If we wanted to, we could plug in proper friction coefficients or even attempt to calculate our own based on the above formula. But for the most part, it fairly common to arrive at some fixed constant value(s) that produces a good feeling result in the game. Again, for simplicity in this lab project, rather than have every surface in our game store friction coefficients, we will store a single coefficient in our CPlayer class and apply it universally to all surfaces. Feel free to alter this behavior should you deem it necessary for your particular needs.

Our formula for calculating dynamic friction will be very straightforward. Friction will act in the opposite direction of our current movement vector, and our goal is to essentially have a force that will act to reduce the speed of the player over time. We can model idea this using the following calculation:

Friction = -m_fFriction * Velocity

> **Note:** Depending on how you decide to represent and store your friction coefficients, you could either calculate the constant as (1 – m_fFriction) or just m_fFriction.

In this case, Velocity is the current velocity vector of our player and m_fFriction is the friction coefficient. Again, we are using a single surface friction coefficient in our lab project but this could be extended and stored at the per polygon level.

Note that the friction calculation we are using is pretty much identical to the formula we learned for viscous drag for slow moving objects:

$$\mathbf{F}_R = -k\mathbf{v}$$

The resulting friction vector in the above calculation can later be added to our velocity vector. Since the sign of the friction coefficient is negative, this generates a vector pointing in the opposite direction of the velocity vector with the length scaled by the friction coefficient. Adding this vector to the velocity vector would essentially subtract some amount of length (i.e., reduce speed) from the velocity vector each frame, eventually whittling it down to zero magnitude over a number of frames (assuming no further motor forces were applied).

We can set the friction coefficient of our CPlayer object using the CPlayer::SetFriction function. This same friction coefficient is used for every surface, so there will be constant friction force working against our velocity vector in each update.

Air Resistance / Drag – m_fAirResistance

Another negative force we will consider is that of air resistance / drag. The amount of air resistance we will apply is going to be calculated using a drag coefficient just as we discussed earlier. This coefficient will ultimately describe how aerodynamic the object is. The less aerodynamic the shape of an object, the more air resistance will be experienced. Drag is an extremely important force to consider when modeling the movement of high speed objects, such as a car in a racing game.

The drag vector can be calculated with the following calculation where m_fAirResistance is a scalar value assigned to the object describing how aerodynamic it is (drag coefficient). Other factors could also

be considered when calculating this drag coefficient such as the type and density of the fluid being navigated, or even wind strength and direction (although this could be modeled as a separate motor force). In our demonstration we will actually use the higher speed version of our drag formula, but you can obviously experiment with this as you see fit. Recall that our formula was:

$$\mathbf{F}_R = -k\mathbf{v}^2$$

In code then, we will simply calculate the drag force as:

vecDragForce = -m_fAirResistance * m_vecVelocity * |m_vecVelocity|

Once again, we can see that this vector represents force acting in the exact opposite direction of the velocity vector. The length of this vector is equal to the squared length of the velocity vector scaled by our drag coefficient.

Gravity – m_vecGravity

As mentioned earlier, gravity is another force that we will include in our model. Keep in mind that gravity has been part of our CPlayer object since its first incarnation. We will represent gravity using a vector which will be directed down the -Y axis in world space. While it is not a system necessity that the gravity vector point down, this is obviously going to be preferable in most real world simulations. The magnitude of the gravity vector describes the strength of the gravitation pull of the planet on which our player is moving. To calculate the gravitational force applied to the object, we will turn to Newton's Second Law of Motion and multiply the gravity (acceleration) vector by the mass of the object.

vecGravForce = vecGravityVector*m_fMass

Updating our Velocity Vector

As discussed, we will apply motor force to our player object using the CGameApp::ApplyForce function (called from the CGameApp::ProcessInput). Regardless of how many times this function is called prior to the CPlayer::Update function, the forces will be combined in the m_vecAppliedForce vector.

After CGameApp::FrameAdvance has called the CGameApp::ProcessInput function to apply any forces, and has called the CScene::AnimateObjects function to update the scene and collision database, it next calls the CPlayer::Update function. Prior to going into this function we know that the m_vecAppliedForce member will contain a combination of all the motor forces applied in this iteration of the game loop.

As m_vecAppliedForce contains all motor forces we have applied to the object, we will first scale this vector by the traction coefficient to generate part of our final tractive force. This force vector will describe the actual force applied in the requested direction of motion. As mentioned, we will also include gravity in our tractive force. We will treat this a little differently from the motor force case and it

will not be scaled by our traction coefficient. Therefore, to generate the final tractive force we will scale the gravity vector by the mass of our player object and add the result to the applied force(s) vector.

m_vecAppliedForce = m_vecAppliedForce * m_fTraction
m_vecTractiveForce = m_vecAppliedForce + (m_vecGravity * m_fMass)

At this point, m_vecTractiveForce contains a vector describing the direction and force that should be applied to the object. Note that by giving ourselves a little bit of traction while we the player is airborne (we can set the value fairly low, but not quite 0), we are given a minor degree of control during jumps without having to write a lot of extra code. You can think of it as giving the player a bit of 'grip' on the air even though there is no actual surface to push off from.

Going back to our earlier discussion of what we are attempting to model at the high level, this vector represents our positive forces (i.e., those that will attempt to accelerate the player). Next we need to focus on generating our negative forces (i.e., the resistance forces acting against the force vector we have just created).

We start with the dynamic friction force. As discussed, this will be calculated as the negated result of the friction coefficient multiplied by the object's current velocity:

m_vecFrictionForce = - m_fSurfaceFriction * m_vecVelocity

While this approach would work fine, we decided to tweak the model a bit and take into account our traction idea as well. This could be described as a bit of a hack, but it makes it much easier to adjust the starting and stopping properties of the object given that we are using a constant friction in our demonstration. While the surface friction coefficient will be an arbitrary value, the traction coefficient will always be in the 0.0 to 1.0 range. By scaling the surface friction coefficient by the traction coefficient, we essentially make the assumption that if the friction of the surface is low (such as ice), it will take our player a while to ramp up to full speed. It would also take the player longer to stop given the difficulty getting a good grip on the surface. This friction force is going to be subtracted from the velocity vector of the player each time:

m_vecFrictionForce = - (m_fTraction * m_fSurfaceFriction) * m_vecVelocity

Next we calculate our final resistance force -- drag. Since we are decided to use the drag calculation for high speed movement, this force will be calculated by scaling the drag coefficient by a vector that is equal in direction to our velocity but with a squared length and then negating the result.

m_vecDragForce = - m_fAirResistance * (m_vecVelocity * | m_vecVelocity |)

We can now calculate the total force working on the body by summing the tractive force vector, the friction force vector, and the drag vector:

m_vecTotalForce = m_vecTractiveForce + m_vecFrictionForce + m_vecDragForce

Now that we know the total amount of force acting on the object, we use Newton's Second Law to do the rest. We divide out total force by the mass of the object to calculate the actual acceleration (or deceleration) of the object:

m_vecAcceleration = m_vecTotalForce / m_fMass

Now we must scale the acceleration by the elapsed time between frames so that we can determine the amount of acceleration to apply for this frame update. Recall that our definition of acceleration was a change in velocity with respect to time:

$$\mathbf{a} = \Delta\mathbf{v} / \Delta t$$

How can we use this knowledge to figure out what our new velocity needs to be? Well, for starters, let us consider what the change in velocity represents. If we had a previous velocity (v0) and we wanted to determine our new velocity (v1) based on our acceleration, we could say the following:

$$\Delta\mathbf{v} = \mathbf{v}1 - \mathbf{v}0$$

$$\mathbf{a} = (\mathbf{v}1 - \mathbf{v}0) / \Delta t$$

$$\mathbf{a}\Delta t = \mathbf{v}1 - \mathbf{v}0$$

$$\mathbf{v}0 + \mathbf{a}\Delta t = \mathbf{v}1$$

So we can see that all we need to do is scale our acceleration by the elapsed time, add it to our old velocity, and we will have our new velocity:

m_vecVelocity += m_vecAcceleration * TimeScale

Remember that the total force we calculated above may be acting primarily against the current velocity vector if the resistant forces are stronger than the sum of applied forces. This is what allows our player's velocity to decrease (i.e., deceleration) when the user lets go of the movement keys.

Our final important step will be resetting the applied forces vector to zero for the next frame since we have now used up all the force it contained updating the velocity vector.

We will grant that has been a very simplified introduction to physics, but it has given us all we need to implement fairly good handling in our CPlayer::Update function. You are strongly encouraged to take your knowledge further by trying out the Game Physics course, where you will get much more detail about the concepts discussed here and explore many other interesting topics in the world of physics.

We will now discuss the functions from the CPlayer class that have been updated.

CPlayer::CPlayer()

The constructor has had very few changes made to it. The friction, traction, drag coefficient, and the mass of the player object are set inside the constructor of CPlayer to default values. We use these default values in our application but you can change these values using the CPlayer methods as needed.

```
CPlayer::CPlayer()
{
    // Clear any required variables
    m_pCamera            = NULL;
    m_p3rdPersonObject   = NULL;
    m_CameraMode         = 0;
    m_nUpdatePlayerCount = 0;
    m_nUpdateCameraCount = 0;

    // Players position & orientation (independent of camera)
    m_vecPos             = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecRight           = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    m_vecUp              = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    m_vecLook            = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );

    // Camera offset values (from the players origin)
    m_vecCamOffset       = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_fCameraLag         = 0.0f;

    // The following force related values are used in conjunction with Update
    m_vecVelocity        = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecGravity         = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecAppliedForce    = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_fTraction          = 1.0f;
    m_fAirResistance     = 0.001f;
    m_fSurfaceFriction   = 15.0f;
    m_fMass              = 3.0f;

    // Default volume information
    m_Volume.Min         = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_Volume.Max         = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );

    // Used for determining if we are on the floor or not
    m_fOffFloorTime      = 1.0f;

    // Collision detection on by default
    m_bCollision         = true;
}
```

Notice that there is a new member variable called m_fOffFloorTime. This is used to track how long the player has not been making contact with the ground. It works as follows…

Every time the collision query is run on the player (shown in a moment) we will get back the bounding box describing the extents of the intersections that occurred between the player object and the environment. We can use this box to determine whether intersections have happened between the underside of the ellipsoid and a scene polygon. If so, we can assume that the player is in contact with the ground and the CPlayer::SetOnFloor method is called with a parameter of true.

```
void CPlayer::SetOnFloor( bool OnFloor )
{
    // Set whether or not were on the floor.
    if ( OnFloor )
        m_fOffFloorTime = 0.0f;
    else
        m_fOffFloorTime = 1.0f;
}
```

As you can see, this sets the value of the m_fOffFloorTime variable to zero. This happens every frame when the underside of our ellipsoid is found to be making contact with a surface.

Going further, in every subsequent frame (in the CPlayer::Update function), this variable is incremented by the elapsed time. Thus, if the ellipsoid is always making contact with the ground, it will be getting incremented by some small value and then ultimately reset to zero each frame by the collision test. However, if the collision query finds that the ellipsoid is not in contact with the ground, the SetOnFloor function is not called, but the value is still incremented so the off floor timer continues to increase.

We know that if the player is not making contact with the floor, they should not be allowed to jump. We also saw in the CGameApp::ProcessInput function that it used the CPlayer::GetOnFloor function to determine whether the player is considered to be making contact with the floor or not. It was only when this function returned true that a control key press was processed and the player's Y velocity adjusted to launch the player up into the air. The CPlayer::SetOnFloor function was then called to inform the player that it is currently in the air.

The CPlayer::GetOnFloor function is shown below.

```
bool CPlayer::GetOnFloor() const
{
    // Only return true if we've been off the floor for < 200ms
    return (m_fOffFloorTime < 0.200);
}
```

As you can see, it only returns true if the off floor timer has not been incremented past 200 milliseconds. Now, we know that in reality, as soon as this timer is not equal to zero, the player is off the ground and the function should return false for floor contact. The problem with doing it in that way is that it makes our system a little too sensitive. There may be times when our player is minimally bouncing along a surface and may be just a bit off the surface for a very small amount of time. In such cases we do not want to disable the jump ability since the player is not really supposed to be considered to be in the air; it may just be bumping over rough terrain for example. Therefore, only if the player has been off the ground for 200 milliseconds or more will we consider the character to be truly off of the ground and return false.

When we look at the constructor we are also reminded that our CPlayer object has two vectors describing its bounding box. The half length of this box will be used as the radius vector of the ellipsoid used by the collision system. That is, the bounding box describes a volume that completely and tightly encases the ellipsoid.

CGameApp::SetupGameState

The CGameApp::SetupGameState function of our framework is called to allow the application to initialize any objects before the game loop is started. This is where the player object has many of its properties set to their starting values. Some of the values shown here may vary from the actual source code in Lab Project 13.1 due to some last moment tweaking.

```
 void CGameApp::SetupGameState()
{
    // Generate an identity matrix
    D3DXMatrixIdentity( &m_mtxIdentity );

    // App is active
    m_bActive = true;

    m_Player.SetCameraMode( CCamera::MODE_FPS );
    m_pCamera = m_Player.GetCamera();
```

First we set the player object (a member of CGameApp) into first person mode. We then grab a pointer to the player's camera.

Next we set the acceleration due to gravity which will be applied to the player object during updates. Again, our gravity vector magnitude was basically determined by trial and error. Feel free to modify any of these values to better suit your own needs. We also set the offset vector describing the position of the player's camera relative to its position and set the camera lag initial to zero seconds.

```
    // Setup our players default details
    m_Player.SetGravity( D3DXVECTOR3( 0, -800.0f, 0 ) );
    m_Player.SetCamOffset( D3DXVECTOR3( -3.5f, 18.9f, 2.5f ) );
    m_Player.SetCamLag( 0.0f );
```

We then set up a suitable bounding box that encases the mesh of our CPlayer object. We use the CPlayer::SetVolumeInfo to set the player's bounding volume.

```
    // Set up the players collision volume info
    VOLUME_INFO Volume;
    Volume.Min = D3DXVECTOR3( -11,  -20, -11 );
    Volume.Max = D3DXVECTOR3(  11,   20,  11 );
    m_Player.SetVolumeInfo( Volume );
```

Next we set the camera's viewport properties and its bounding volume:

```
    // Setup our cameras view details
    m_pCamera->SetFOV( 80.0f );
    m_pCamera->SetViewport(   m_nViewX,
                              m_nViewY,
                              m_nViewWidth,
                              m_nViewHeight,
                              m_fNearPlane,
                              m_fFarPlane );
```

```
    // Set the camera volume info (matches player volume)
    m_pCamera->SetVolumeInfo( Volume );
```

Then we set the initial position of the player in the world.

```
    // Lets give a small initial rotation and set initial position
    m_Player.SetPosition( D3DXVECTOR3( 50.0f, 50.0f, 20.0f ) );

    // Collision detection on by default
    m_bNoClip = false;
}
```

CPlayer::Update

Before we look at the code to the CPlayer::Update function, you should be aware of a new line of code that has been added to the CScene::LoadSceneFromIWF function. This new line of code registers a CScene callback function with the player that will be called from the update function. We will see this in a moment.

Excerpt from CScene::LoadSceneFromIWF

```
GetGameApp()->GetPlayer()->AddPlayerCallback( CScene::UpdatePlayer, this );
```

With this knowledge in hand, we can look at the modified Update method which is now called from CGameApp::FrameAdvance just after the input has been processed and the scene objects have been animated. In this function, we will use the physics model we discussed earlier. CPlayer::Update is passed one parameter, the amount of time (in seconds) that has elapsed since the previous Update call.

The first thing we do is scale the applied motor forces by the traction coefficient to get the first part of our tractive force. We then add our gravitational force calculated by multiplying the gravity acceleration vector by the mass of the object:

```
void CPlayer::Update( float TimeScale )
{
    D3DXVECTOR3 vecTractive, vecDrag, vecFriction, vecForce, vecAccel;
    bool        bUpdated = false;
    float       fSpeed;
    ULONG       i;

    // Scale our traction force by the amount we have available.
    m_vecAppliedForce *= m_fTraction;

    // First calculate the tractive force of the body
    vecTractive = m_vecAppliedForce + (m_vecGravity * m_fMass);
```

At this point we have our total tractive force. Now it is time to calculate the resistance forces that act against this tractive force vector.

First we will calculate drag based on our model discussed earlier (for high speed objects):

```
    // Now calculate the speed the body is currently moving
    fSpeed = D3DXVec3Length( &m_vecVelocity );

    // Calculate drag / air resistance (relative to the speed squared).
    vecDrag = -m_fAirResistance * (m_vecVelocity * fSpeed);
```

Next we calculate the dynamic friction vector, once again using the model introduced earlier (similar to viscous drag for slow moving objects). We can then add the friction vector, drag vector, and the tractive force vector together and divide the resulting vector to get our total force. Dividing by the mass of the object gives us the current acceleration of the object.

```
    // Calculate the friction force
    vecFriction = -(m_fTraction * m_fSurfaceFriction) * m_vecVelocity;

    // Calculate our final force vector
    vecForce = vecTractive + vecDrag + vecFriction;

    // Now calculate acceleration
    vecAccel = vecForce / m_fMass;
```

Now we can scale our acceleration by the elapsed time to get an acceleration value for this frame update. The result is then added to our previous velocity vector and we have our final velocity for this frame. The applied forces vector is then reset to zero since we have used up all applied forces at this point.

```
    // Finally apply the acceleration for this frame
    m_vecVelocity += vecAccel * TimeScale;

    // Reset our motor force.
    m_vecAppliedForce = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
```

We have now successfully calculated our new velocity vector.

In the next step we will loop through any callback functions that have been registered for the player. In our application there will be just one -- a pointer to the CScene::UpdatePlayer function which is responsible for running the collision query. We will see this in a moment when we discuss the source code.

```
    // Only allow sources to fix position if collision detection is enabled
    if ( m_bCollision && m_nUpdatePlayerCount > 0 )
    {
        // Allow all our registered callbacks to update the player position
        for ( i = 0; i < m_nUpdatePlayerCount; i++ )
        {
            UPDATEPLAYER UpdatePlayer =(UPDATEPLAYER)m_pUpdatePlayer[i].pFunction;

            if ( UpdatePlayer( m_pUpdatePlayer[i].pContext, this, TimeScale ) )
                bUpdated = true;
        }
    } // End if collision enabled
```

At this point the callback function would have been called to run a collision query on the position and velocity of the player. It would have now set the player in its new non-colliding position. We will look at the callback function next since it is the final piece in our puzzle.

In the next section of code we call some legacy code if the collision system is not being invoked or if no callback functions have been registered to update the position of the player. When this is the case the pUpdated member will not have been set to true (see above code), so we scale the new velocity vector we just calculated by the time scale and pass this movement vector into the CPlayer::Move function to physically alter the position of the player. This next section of code will not be executed by our application in Lab Project 13.1 since we are using the collision system.

```
    if ( !bUpdated )
    {
        // Just move
        Move( m_vecVelocity * TimeScale );

    } // End if collision disabled
```

Next we call the Update method of the player's camera object so that it can also alter its position. This is important when the camera is a third person mode since it will cache the new position of the player and start to track it (with the desired amount of lag). This function is a no-op for other camera modes since they do not implement this virtual function.

```
    // Let our camera update if required
    m_pCamera->Update( TimeScale, m_fCameraLag );
```

If this is a no-op for other camera modes, you may be wondering how the position of those cameras are updated. The answer will become clear when we look at the callback function momentarily.

Next we also loop through any callback functions which may have been registered for the player's camera and call those functions as well. This allows us to register functions which will handle the collision of the camera with the scene if we wish. If this callback function calling code looks unfamiliar to you, refer back to Chapter 4 where we first implemented and discussed this callback system.

```
    // Only allow sources to fix position if collision detection is enabled
    if ( m_bCollision && m_nUpdateCameraCount > 0 )
    {
        // Allow all our registered callbacks to update the camera position
        for ( i =0; i < m_nUpdateCameraCount; i++ )
        {
            UPDATECAMERA UpdateCamera =(UPDATECAMERA)m_pUpdateCamera[i].pFunction;

            UpdateCamera( m_pUpdateCamera[i].pContext, m_pCamera, TimeScale );

        } // Next Camera Callback

    } // End if collision enabled.
```

Next we use the GetOnFloor function to adjust the traction and surface friction coefficients. If the player is not currently in contact with the floor, we will assume that there is no surface friction acting against its velocity. It would seem strange if the velocity of our player was being diminished by surface friction if the player is currently falling through the air.

As mentioned earlier, in reality the player should not have any traction in the air since there is nothing for the player's feet to push against. In this case we will set the surface friction coefficient to zero but set the traction coefficient to a very low value. Although it would be more correct to set the traction to zero too, we set it to a very low value so that the player still has limited directional control even when in mid air. This is consistent with gameplay mechanics in many first and third person titles. If the player is in contact with the floor, we set its traction and friction values to 1.0 and 10.0 respectively. Feel free to experiment with these values to change the feel of the player as it navigates the scene to something that suits your tastes.

```
    if ( !GetOnFloor() )
    {
        SetTraction( 0.1f );
        SetSurfaceFriction( 0.0f );

    } // End if not on floor
    else
    {
        SetTraction( 1.0f );
        SetSurfaceFriction( 10.0f );

    } // End if on floor
```

Finally, we increment the m_fOffFloorTimer value by the elapsed time as we must do each frame. We will see in a moment how this value is set to zero when the collision test determines that the player is in contact with the ground. Only when this variable has been incremented past 200 (milliseconds) do we consider the player to truly be off the ground for a significant enough amount of time for us to consider the player to be in the air. Thus it is only when this variable is greater than or equal to 200 does the GetOnFloor function returns false.

```
    // Increment timer
    m_fOffFloorTime += TimeScale;

    // Allow player to update its action.
    CPlayer->ActionUpdate( TimeScale );
}
```

As the final line of the function you can see us call the CPlayer::ActionUpdate function, which was added in the previous lesson. This function adjusts the current animation action of the third person object attached to the player. It sets the object's world matrix and also sets the actor action based on whether the character is idle or shooting, for example. Any other animations you wish to be played based on input or game events should be placed in this function.

We have now seen how the Update function of the player has been enhanced to include a more realistic physics model. We have not yet seen where the velocity vector we calculate in the above code is

actually used to update the position of the player object. As discussed, this is done in the CScene callback function that is registered with the player and called from the function previously discussed. Let us look at this callback function now to conclude our discussion.

CScene::UpdatePlayer

This callback function is the glue that holds this whole application together. It is called from the previous function and is passed a pointer to the player we wish to move. It is also passed the elapsed time between frames (in seconds) so that it knows how to scale that movement. As the code to the previous function demonstrated, when this function is called, the current velocity vector has been calculated using our player physics model so that at this point, the player object contains a velocity vector describing the direction and speed it would *like* to move. This function will call the collision system to see if the player can be moved along the velocity vector without obstruction, and if not, the collision system will return a new position for the player that is free from intersection. This function will then update the position of the player as dictated by the collision system. The collision system also returns a new velocity vector describing the direction and speed the player should be traveling after intersections have been factored in. We will see in a moment that we will not simply set the velocity of the player to the one returned from the collision system since this will cause some problems in our physics model. Instead, we will make a few adjustments so that our player's velocity will not diminish too significantly when trying to ascend shallow slopes.

We decided against placing the call to CCollision::CollideEllipsoid directly inside the CPlayer update function because then the CPlayer object would be reliant on the CCollision class and the design of CPlayer would be less modular. It makes sense that the callback function used for this purpose should be part of the CScene namespace since it is the scene object that contains both the scene geometry and the collision database (CCollision). The scene object also has the functions that load the scene and register it with the collision database. So it seemed a sensible design that this object should also be the one responsible for running queries on the collision database in our application.

Let us look at this function a little bit at a time. The function (like all player callback functions) accepts three parameters. The first is a void context pointer that was registered with the callback function. This data pointer can be used to point at anything, but we use it when we register the callback function to store a pointer to the CScene object. This is important because in order for the CScene::UpdatePlayer function to be a callback function, it must be static and therefore have no automatic access to the non-static members of the CScene class. In our code, this first parameter will point to the instance of the CScene object that contains the collision geometry. For the second parameter, a pointer to the calling CPlayer is passed. The third parameter will be the elapsed time since the last update.

```
bool CScene::UpdatePlayer( LPVOID pContext, CPlayer * pPlayer, float TimeScale )
{
    // Validate Parameters
    if ( !pContext || !pPlayer ) return false;

    // Retrieve values
    CScene     *pScene   = (CScene*)pContext;
    VOLUME_INFO Volume   = pPlayer->GetVolumeInfo();
    D3DXVECTOR3 Position = pPlayer->GetPosition();
```

```
    D3DXVECTOR3 Velocity = pPlayer->GetVelocity();
    D3DXVECTOR3 AddedMomentum;

    D3DXVECTOR3 CollExtentsMin, CollExtentsMax, NewPos, NewVelocity;
    D3DXVECTOR3 Radius = (Volume.Max - Volume.Min) * 0.5f;
```

In the first section of code we cast the context pointer to an object of type CScene so we can access the scene's methods and member variables. We also extract the position, velocity and bounding box information from the player into local variables. The ellipsoid that will be used to approximate the player in the collision system has its radius vector generated by using half the length of the bounding box. Remembering that the radius vector describes the distance of the ellipsoid surface from the center of the ellipsoid, we can see that this describes an ellipsoid that fits relatively tightly in the player's bounding box. We also allocate two 3D vectors called CollExtentsMax and CollExtentsMin which will be used to transport information about the extents of the collisions that occur with the ellipsoid during the detection process.

We now have all the information we need to run the collision test.

```
    // Test for collision against the scene
    if ( pScene->m_Collision.CollideEllipsoid(  Position,
                                                Radius,
                                                Velocity * TimeScale,
                                                NewPos,
                                                NewVelocity,
                                                CollExtentsMin,
                                                CollExtentsMax ) )
    {
```

When the CollideEllipsoid function returns, the local variables NewPos and NewVelocity will contain the new position and velocity of the object calculated by the collision system. The CollExtentsMax and CollExtentsMin vectors will describe the maximum and minimum points of intersection along the X, Y and Z axes that occurred in this movement update.

The first thing we do is test the minimum Y coordinate of the intersections bounding box returned by the collision system. If the minimum point of intersection along the Y axis is smaller than the Y radius of the ellipsoid, minus a quarter of the Y radius of the ellipsoid, then it means we have an intersection that occurred with the ellipsoid in its bottom quarter (roughly) as shown in Figure 13.11.

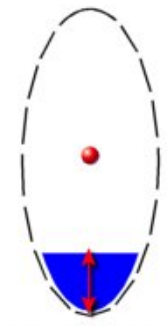


Figure 13.11

When this is the case, we consider the ellipsoid to be in contact with the floor and we call the CPlayer::SetOnFloor function to reset the player's m_fOffFloorTimer to zero. It will take at least another 200 milliseconds of non-contact between the ellipsoid and the floor for the player to be considered in the air.

```
        // If the lowest intersection point was somewhere in the
        // bottom quarter (+tolerance) of the ellipsoid
        if ( CollExtentsMin.y < -(Radius.y - (Radius.y / 4.25f)) )
        {
            // Notify to the player that were in contact with a "floor"
            pPlayer->SetOnFloor( );

        } // End if hit "feet"
```

The reasons for the existence of the next section of code will require some explanation as it is responsible for generating the new velocity vector of the player. Is it not true that the collision function returned us the new integration velocity? Well, yes and no.

It is true that the collision system does correctly calculate our new velocity, and in a perfect world we could simply set this new velocity as the player's velocity. If collisions occurred, the velocity vector will now be pointing in the slide direction, which is what allows our collision system to slide our player over bumps and steps. If we were passing in a really large initial velocity vector each time, we would cruise over most slopes and bumps with little trouble. However, now that we have integrated more complex physics into our model, our velocity also has resistant forces working against which diminish it each frame. Therefore, the amount of motor force we would usually apply to allow the ellipsoid to slide around a section of flat floor will not be enough to slide the same ellipsoid up a steep slope.

If we think about this in real terms, imagine applying enough force to a ball so that it slides at a fairly slow constant speed across your carpet. Now imagine that the ball hits a step. If you applied the same force to the ball, it would be nowhere near enough to allow you to push the ball up and over the step. The same is true here. What we consider a nice motor force for the player when moving along flat ground will not be enough to slide up most ramps with gravity and drag working against us.

We know that our collision system will project the input velocity vector onto the sliding plane so that the direction of the player changes to slide along corners and up steps. However, we also know that if our ellipsoid was to hit a step of significant size, the velocity vector would have its direction diverted so that it pointed up into the air, for the most part. The velocity vector returned will have most of its movement diverted from horizontal movement along the X and Y axes to vertical movement along the Y axis. Even if the player had enough force so that the projected velocity vector was enough to push the ellipsoid up higher than the step, we will have lost most our forward (X,Y) momentum contained in the original velocity vector. Essentially, we might have enough projected velocity to clear the step vertically, but then we would not have enough to actually move forward and onto the step.

To solve this particular problem we could use only the Y component of the velocity vector returned from the collision system and use the X and Z components of the original velocity vector. This way, the collision system can correctly inform us about how we need to move up and down when a step is hit, but the original X, Z velocity is still maintained (not diminished) so we have enough horizontal force to move not only up the step, but up and over. It is vitally important that we always obey the Y component of the velocity vector returned from the collision system since this is what prevents us from falling through floors. Remember, we might pass in a velocity vector with a massive downwards force (e.g., a very strong gravitational force has acted) of say (0, 300, 0). The collision system would detect that we cannot fall downwards as described by the velocity vector because there is geometry underneath us and

the returned velocity vector would have a Y component of zero. So, the Y component of our player's velocity must be set to the exact Y velocity returned from the collision system.

Ignoring the returned X and Z velocity components and simply using the original (pre-collision) X and Z velocity components along with the Y velocity returned from the collision system would certainly solve our problem in one respect. It would mean that the collision system can tell us the new up/down velocity that should be applied to our position to clear obstacles and steep slopes, while the original horizontal velocity would keep us moving in the XZ direction with the original forces applied. This would allow us to glide over steps and slopes with ease since our horizontal momentum would not be diminished at all by the obstacles we encounter. Indeed we would be able to slide up very steep slopes; just about any slope that was not perfectly vertical. Of course, that in itself is problematic since we do not want our player to be able to slide up near perfect vertical slopes.

To be sure, ignoring the X and Z components is certainly what we want to do to help us get over small obstacles. That is, we want to obey their direction but not necessarily the length. We need a way of knowing that if the object is fairly small, like a shallow slope or a small step, then we should use the original (pre-collision) X and Z velocity components so that we slide up **and over** the obstacle with ease. However, when we are dealing with larger obstacles or much steeper slopes, we do not want to ignore the X and Z velocity components returned from the collision system. In this case, they describe the way our horizontal momentum should be diminished, preventing us from climbing very steep slopes or large steps. Doing otherwise would seem unnatural unless you were making a Spiderman® style game and it is your intention to allow the player to traverse vertical walls.

So it seems that the degree to which we ignore the diminished X and Z components of the velocity vector returned by the collision system depends on how tall/steep the object we have collided with is. Fortunately, this is no problem; we can determine how high the object we collided with was because we have the collision extents bounding box. We will use the maximum height of the intersection to scale the amount that the horizontal velocity was diminished by the collision system.

Let us get started and see how this will work.

First, the velocity vector returned from the collision system is a per-frame velocity, but when performing our physics calculations we work with per-second values. By dividing the returned velocity vector by the elapsed time we get the new velocity vector specified as a per-second velocity. This is the velocity vector returned from the collision system describing the new direction and speed we should be traveling. We then copy the Y component of the velocity vector returned by the collision system into the velocity vector of our player object. We must retain the Y component of the collision velocity vector because it keeps us from falling through the floor.

```
        // Scale new frame based velocity into per-second
        NewVelocity /= TimeScale;

        // Store our new vertical velocity. Its important to ignore
        // the x / z velocity changes to allow us to bump over objects
        Velocity.y = NewVelocity.y;
```

We have now modified the Y velocity in accordance with the collision system. Next we will determine the maximum height of intersection and use that to scale the difference between our original undiminished horizontal velocity and the horizontal velocity returned from the collision system. It should be noted that this works well with slopes and not just steps, as shown in Figures 13.12 and 13.13

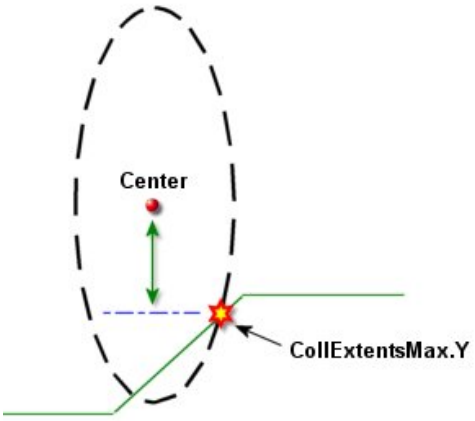


Figure 13.12

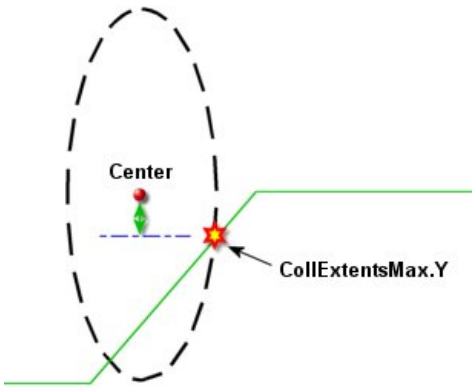


Figure 13.13

In Figure 13.12 we see an ellipsoid colliding with a fairly shallow slope and we can see that the point of intersection is further down the ellipsoid than the intersection shown in Figure 13.13 (where a taller, steeper slope is collided with). It is easy to see that the steeper the slope, the closer to being vertically aligned with the center point of the ellipsoid the intersection point will be. Therefore, we can scale the amount we want to diminish the horizontal movement of the player (as described by the collision system) based on how near to the center of the ellipsoid (vertically) the closest intersection point is. We can imagine for example, that if the ellipsoid collided with a brick wall, the intersection point would be perfectly aligned with the center of the ellipsoid and we should fully obey the collision system when it tells us we need to stop our horizontal movement immediately. On the other hand, if we find that the intersection point is closer to the bottom or top of the ellipsoid, we can ignore the new X and Z diminished velocity vector components returned by the collision system and use the original XZ velocity. This will allow us to slide up and over small obstacles or shallow slopes with ease. This works the same for collisions with both the top and bottom of the ellipsoid. We are interested in finding the intersection point that is closest to the center of the ellipsoid vertically since it is this ratio that will be used to scale the amount by which we factor in the diminished XZ velocity returned by the collision system. The next section of code will show how the new X and Z velocity components of the player's velocity vector are calculated.

First we will subtract about one quarter of the radius height from the radius. We are essentially going to say that if the closest intersection point to the vertical center of the ellipsoid is contained in the bottom quarter or top quarter of the ellipsoid's radius, this is a small obstruction. In that case we remove it completely and do not diminish our original horizontal momentum at all.

```
        // Truncate the impact extents so the interpolation below begins
        // and ends at the above the players "feet"
        Radius.y -= (Radius.y / 4.25f);
```

Having shaved a little over a quarter of the radius length (both sides of the center point), we will now find the Y coordinate of intersection that is closest to the center point. We will first take the maximum of the –Radius.y (the bottom of the new shaved ellipsoid) and the maximum Y intersection point. We will then take the minimum of the positive radius (top of the ellipsoid) and the maximum we just calculated.

```
        CollExtentsMax.y = max( -Radius.y, CollExtentsMax.y );
        CollExtentsMax.y = min( Radius.y, CollExtentsMax.y );
```

What we have now is the Y component of the intersection point that is closest to the center of the ellipsoid, or a value that is equal to the Y radius of the ellipsoid. We do this just to make sure that we get an intersection point along the diameter of the ellipsoid (-Radius to +Radius). This is important because if our player is falling though the air, it will not be colliding with anything and we need the value to be in the [–Radius, +Radius] range to perform the next section of code. Furthermore, because we shaved ¼ of the Y radius of the ellipsoid, if the maximum intersection point is in the bottom or top quadrant of the ellipsoid, it would initially be smaller than the shaved radius. That is why we clamp the values in the range of the shaved vertical diameter of the ellipsoid.

Now we will create a weighting value called fScale which will be calculated by dividing the absolute value of our closest Y point (closest to the ellipsoid center), by the radius of the ellipsoid and subtracting the result from 1.

```
        float fScale = 1 - (fabsf(CollExtentsMax.y) / Radius.y);
```

fScale is a parametric value in the range [0, 1] describing the Y difference between the ellipsoid center and our closest intersection point. If the intersection point is equal to the center of the ellipsoid (vertically), a value of 1.0 will be generated. If the closest intersection point is right at the top or bottom of the ellipsoid (or anywhere in the top or bottom zones), a value of 0.0 will be generated. We will now use this value to scale the difference between our original horizontal velocity (pre-collision) and the diminished horizontal velocity returned from the collision system.

```
        Velocity.x = Velocity.x + (((NewVelocity.x ) - Velocity.x) * fScale);
        Velocity.z = Velocity.z + (((NewVelocity.z ) - Velocity.z) * fScale);
```

As you can see, we subtract the original components from the new diminished horizontal velocity components:

(NewVelocity – Velocity)

Since NewVelocity is always smaller in the case of an intersection, this generates a vector acting in the opposite direction to that of the original velocity vector. Therefore, adding it to the original velocity vector actually subtracts the correct amount from its length such that it is equal to the diminished velocity vector. For example, if our original velocity vector was <10, 10> but the collision system returned a diminished velocity vector of <2, 2>, subtracting and adding to the original velocity vector would give:

OriginalVelocity += NewVelocity – OriginalVelocity
= <2, 2> <-10, -10>
= <-8, -8>

Therefore:

<10,10> += <-8,-8>
= <2 , 2>

That seemed like a lot of manipulation to do just to scale the original velocity so that it is equal to the velocity returned from the collision system. Why not just use the velocity returned from the collision system as the new velocity in the first place? Well, if that is all we intended to do then we would, but notice that the negative vector created from (NewVelocity – Velocity) is actually scaled by fScale. Therefore, the amount we diminish our velocity vector as dictated by the collision system, ends up being dependant on how close to the center of the ellipsoid a collision happened. As such, we have achieved our goal! For little bumps or shallow slopes, fScale will be close to zero, allowing us to continue our horizontal movement unhindered. When the slope is steep or the obstacle is tall, fScale will be closer to 1.0 and the full diminishment of horizontal momentum is applied to the player's velocity as directed by the collision system.

With our velocity vector now correctly calculated and the new position of the ellipsoid returned from the collision system, we can set them as the CPlayer's current position and velocity before returning from the function.

```
        // Update our position and velocity to the new one
        pPlayer->SetPosition( NewPos );
        pPlayer->SetVelocity( Velocity );

        // Hit has been registered
        return true;

    } // End if collision

    // No hit registered
    return false;
}
```

We have now covered the function in its entirety and have no further code to discuss. Do not expect to understand all of the code we have added in this lab project right away. Expect to have to study it again before you feel comfortable implementing such a system yourself from the ground up. The main areas of focus in this lesson have been the CCollision class and the updated CPlayer code.

Conclusion

In many ways, this lesson has been a very exciting one. We now have a reusable collision system that can be plugged into our future applications to provide collide and slide behavior with arbitrary scene geometry. We have implemented it in such a way that our existing classes such as CActor can also be registered with the collision system's geometry database. We have even employed a system that allows us to support animating collision geometry when the situation calls for it.

We also upgraded our CPlayer object with a nicer (but still admittedly simplistic) physics model which allows us much greater control over how the player interacts with the collision environment. The CCollision class will be used in all lab projects from this point onwards.

In the next lab project we will implement hierarchical spatial partitioning to improve our collision system's performance (among other things). We will add a spatial partitioning system to our CCollision class so that a fast broad phase can be introduced to quickly identify potential colliders, even when dealing with huge data sets. The broad phase component will significantly speed up our collision system by performing efficient bulk rejection of polygons that are not contained within the same region of the scene as the ellipsoid that is currently being queried. This is going to be very important since our collision system currently has to perform the full range of intersection tests on every triangle that has been registered. This will simply not be good enough as we progress with our studies and start to develop larger and more complex scenes. Before moving on, please be sure that you are comfortable with the material covered in the textbook and workbook as we will see a lot of it again in the next chapter.