Chapter 5

Waypoint Networks

Overview

So far, we have talked about pathfinding algorithms, flocking algorithms, decision systems, and scripting. In this chapter we will start bringing all of these ideas together to demonstrate how you might actually use each of these systems in an integrated environment. We will start by talking about waypoint networks, a specific implementation of a pathfinding graph we briefly touched on earlier. We will talk about how we can attach data to these networks such that the decision making system, a state machine such as one we discussed previously, can make decisions based on the data in the network. We will then talk about squads and squad leaders, and how we can implement their state machines so they cooperate with one another.

In this chapter we answer the following questions:

- What are waypoint networks?
- How can we attach data to waypoints, so we can use it to make decisions?
- What are the common methods of implementing squad communication?
- How can we bring all of this together?

5.1 Waypoint Networks

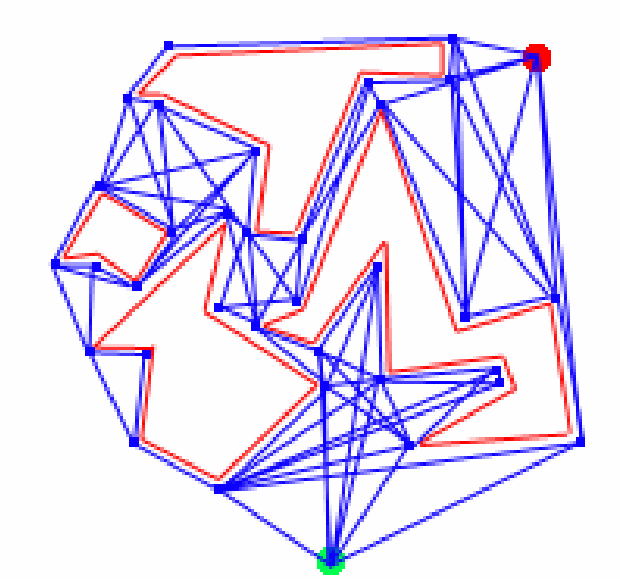

Figure 5.1

In Chapter Two we talked very briefly about pathfinding on non-gridded maps. We mentioned that one of the more popular methods of dealing with these sorts of worlds is called *waypoint networks* (or visibility points). Figure 5.1 should look familiar, since it is the same one we talked about previously when we introduced waypoint networks. The red polygons are obstacles, the small blue dots are waypoints, and the blue lines between them are the edges of the network. In this chapter, we are going to discuss this method of dealing with continuous worlds in depth. We will talk about the architecture of the network, methods to traverse the network, and some additional considerations that come up when using such a system for games.

5.1.1 Waypoints

Let us begin by talking about the waypoints themselves since they are obviously the fundamental element in a waypoint network. Ultimately, a waypoint network is just a collection of waypoints and the edges between them. If you recall some of the terminology we used during the early part of this course, a waypoint network is a graph, and the waypoints are just nodes in the graph.

So what are some important characteristics of a waypoint? Well, first of all, it has to have a position in space. It also needs to have a collection of edges to other waypoints that can be reached from it. A waypoint probably should have some sort of identifier (an simple integer ID, a GUID, etc.), although it is not strictly necessary. It should also have a radius and we will talk about why this is important in just a bit. Other useful information would be an orientation. The orientation is handy if you want to hint to the entities traversing the network that something useful might be found in the direction of this

waypoint. For example, you might set up a waypoint that is at the top of a hill, with an excellent vantage over an enemy base. You could set the orientation of this waypoint to face towards the base, so the entities traversing the network could see that it is a good sniping position.

The last thing you might want to attach to a waypoint is general blind data. General blind data is basically game related data that is attached to the waypoint, but the waypoint does not necessarily know, nor care about, what that data is. It is just holding onto it for the game entities to process when they come across it. Then, when the entity runs across the waypoint, it can peer into that blind data (with full knowledge of what is in there), and make some decisions based on it. In our chapter demo, we store a color as blind data. We use this color to set the color of the entity's arrow when we render it.

Some other examples of things you might want to use as blind data on waypoints are:

- Animation trigger data – tells the entity to play a specific animation upon reaching the waypoint
- Wait Signal – tells the entity to pause briefly upon reaching the tagged waypoint
- Look around signal – tells the entity to pause and look around for enemies
- Cover – tells the entity that crouching here would provide them with cover from the direction indicated by the orientation of the waypoint
- Defend – tells the entity that this position is a good defensive position
- Danger – tells the entity that this position is particularly dangerous
- Posture – tells the entity that movement from this waypoint should be done using a given posture (crouch, run, walk)

Discrete Simulations in Continuous Worlds

Let us now talk a little about why the radius data member of a waypoint can be helpful. The important thing to remember about continuous worlds in games is that the game is a discrete simulation, not a continuous one. You have a certain amount of time pass each frame, and you typically update the position of an entity by integrating the velocity of the entity using that time delta (using standard Euler or other numerical integration techniques).

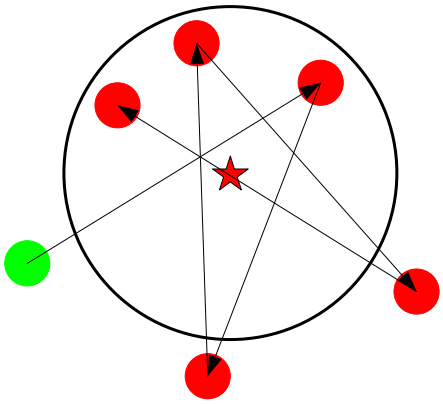

Figure 5.2

Let us assume that your entity is moving at say, 10 meters/sec, but your game only updates every 33 milliseconds (30Hz, or 30 frames per second). That means the smallest distance your entity can travel in a given frame is 3.3 meters (10 m/s * 0.033 s). So if your waypoint is a single point in space, the likelihood that your entity will land right on it is pretty slim. But if you give your waypoint a radius, and if your entity is "close enough" to the waypoint, then the entity has reached the goal. Take a look at Figure 5.2. If the entity started at the green circle, and he wanted to get to the red star, he could repeatedly jump over the star, every frame, and never reach it, resulting in the series of red circles. However, if the radius of the black circle around the star was used, the entity would have been found to have reached the waypoint after the first iteration.

The Waypoint Class

Now that we have a good understanding of the principle of the waypoint, let us take a look at the actual class we used to represent the waypoint in our demo.

```
//
// cWaypoint - a node in a waypoint network
//
class cWaypoint
{
public:
      cWaypoint(const D3DXVECTOR3 &pos, const D3DXQUATERNION &orient,
                  float radius);
      virtual ~cWaypoint();

      const tWaypointID &GetID() const { return mID; }

      D3DXVECTOR3 &GetPosition() { return mPosition; }
      const D3DXVECTOR3 &GetPosition() const { return mPosition; }
      D3DXQUATERNION &GetOrientation() { return mOrientation; }
      const D3DXQUATERNION &GetOrientation() const { return mOrientation; }
      float GetRadius() const { return mRadius; }

      void SetPosition(const D3DXVECTOR3 &position)
            { mPosition = position; }
      void SetOrientation(const D3DXQUATERNION &orientation)
            { mOrientation = orientation; }
      void SetRadius(float radius) { mRadius = radius; }

      void AllocBlindData(UINT size);
      void FreeBlindData();

      template<class T>
      void GetBlindData(UINT offset, T &data) const
      {
            assert(offset <= (mBlindDataSize - sizeof(T)));
            T *dataPtr = (T*)((char*)mBlindData + offset);
            data = *dataPtr;
      }
      template<class T>
      void SetBlindData(UINT offset, T data)
```

```
      {
            assert(offset <= mBlindDataSize - sizeof(T));
            T *dataPtr = (T*)((char*)mBlindData + offset);
            *dataPtr = data;
      }

      bool AddEdge(const cNetworkEdge &edge);
      bool RemoveEdge(const cNetworkEdge &edge);
      void ClearEdges();

      float GetCostForEdge(const cNetworkEdge &edge,
                           const cWaypointNetwork &network) const;

      tEdgeList &GetEdges() { return mOutgoingEdges; }
      const tEdgeList &GetEdges() const { return mOutgoingEdges; }

      int   Serialize(ofstream &ar);
      int   UnSerialize(ifstream &ar);

protected:
      friend class cWaypointNetwork;
      cWaypoint();

private:
      tWaypointID       mID;
      tEdgeList         mOutgoingEdges;
      D3DXVECTOR3       mPosition;
      D3DXQUATERNION    mOrientation;
      float             mRadius;
      int               mBlindDataSize;
      void              *mBlindData;
};
```

There is a lot to take in there, but most of it is accessors. First we will cover the class data members.

```
      tWaypointID       mID;
```

Each waypoint has an ID. A tWaypointID class in our demo is actually a tGUID class, which is a wrapper class for the Microsoft Windows GUID structure. GUID stands for "Globally Unique Identifier", and the GUID structure stores a 128-bit integer in a specific fashion to serve that purpose. It is used by Windows for COM object registration, among other things. GUIDs look like {D5FEE50A-625B-4b6b-B10B-FAD046F0A729}, and can be generated for us using the GuidGen tool provided with Microsoft Visual Studio, as well as the ::CoCreateGuid() method provided by the Windows API. We will talk more about what these IDs are used for when we discuss the waypoint network class.

```
      tEdgeList         mOutgoingEdges;
```

A waypoint also has a list of edges. The tEdgeList type is really just a typedef for an STL vector of network edge classes, which we will discuss shortly. This list of edges is used during the traversal to see which waypoints can reach which other waypoints.

```
      D3DXVECTOR3       mPosition;
```

```
      D3DXQUATERNION    mOrientation;
```

Waypoints also have a position in space, along with an orientation. We chose quaternions for the orientations in our demo. Quaternions are a useful way to represent rotation data because they have low memory footprint, offer smooth interpolation, and solve the problem of gimble lock. A discussion on how quaternions work is beyond the scope of this course however, but if it interests you, further discussion on the topic can be found in the 3D Graphics Programming series and the Game Mathematics course here at Game Institute.

```
      float             mRadius;
```

Waypoints also have the aforementioned radius. This value is used to determine if an entity is "close enough" to be considered as having arrived at the waypoint.

```
      int               mBlindDataSize;
      void              *mBlindData;
```

Lastly we have some data members for our blind data. In this implementation, blind data is stored as a block of bytes, which can be accessed using template functions provided. We will discuss those shortly.

Now that we know what the basic data of the class is, let us discuss the implementations of the non-trivial methods, starting with the edge management methods.

```
bool cWaypoint::AddEdge(const cNetworkEdge &edge)
{
      // look for the edge, if we find it, don't add it again
      for (tEdgeList::iterator it = mOutgoingEdges.begin();
           it != mOutgoingEdges.end(); ++it)
      {
            cNetworkEdge &e = *it;
            if (e == edge)
                  return false;
      }

      mOutgoingEdges.push_back(edge);
      return false;
}
```

The AddEdge method iterates through its list of edges, and if the edge to be added is not found, it is added to the list.

```
bool cWaypoint::RemoveEdge(const cNetworkEdge &edge)
{
      // locate and remove the edge
      for (tEdgeList::iterator it = mOutgoingEdges.begin();
           it != mOutgoingEdges.end(); ++it)
      {
            cNetworkEdge &e = *it;
            if (e == edge)
            {
```

```
                  mOutgoingEdges.erase(it);
                  return true;
            }
      }

      return false;
}
```

The RemoveEdge method iterates through its list of edges, and if the edge is found, removes it from the list.

```
float cWaypoint::GetCostForEdge(const cNetworkEdge &edge, const cWaypointNetwork
&network) const
{
      cWaypoint *dest = network.FindWaypoint(edge.GetDestination());
      if (!dest)
            return 0.0f;

      D3DXVECTOR3 vec = GetPosition() - dest->GetPosition();
      float distance = D3DXVec3Length(&vec);
      return distance * edge.GetCostModifier();
}
```

This method returns the cost for a given edge. While we have not yet discussed the implementation details of the edge class, this should still be fairly self-explanatory. First, the waypoint obtains the destination waypoint of the edge from the network. It then computes the distance from itself to the destination waypoint. This distance is the base cost of the edge. The edge also has a cost modifier associated with it, which is used to scale the base cost of the edge.

```
      template<class T>
      void GetBlindData(UINT offset, T &data) const
      {
            assert(offset <= (mBlindDataSize - sizeof(T)));
            T *dataPtr = (T*)((char*)mBlindData + offset);
            data = *dataPtr;
      }
```

This template method retrieves a value from the blind data block. It requires the caller know the byte offset into the data block so that you can locate your data, as well as the type of data you want to extract. It uses this information to get the data out of the block for you by adding the byte offset to the data block pointer and casting it to your data type for you. The usage pattern looks like this:

```
      COLORREF color;
      someWaypoint->GetBlindData(0, color);
```

Here, the template method deduced the type of data you wanted by the type of the reference you passed in. The byte offset in this case is 0.

```
      void SetBlindData(UINT offset, T data)
      {
```

```
            assert(offset <= mBlindDataSize - sizeof(T));
            T *dataPtr = (T*)((char*)mBlindData + offset);
            *dataPtr = data;
      }
```

This template method works in a similar fashion to the GetBlindData method. Again, it requires the caller to know the byte offset into the data block where the desired data is to be stored, as well as the type of data to store there. It then gets the pointer to the data block plus the offset, casts it to the type it needs, and sets the data for you. The usage pattern looks pretty much identical to the last one:

```
      COLORREF color;
      someWaypoint->SetBlindData(0, color);
```

Here the template method deduced the type of the data you wanted to set based on the parameter you passed in, and the offset again is 0.

The remaining methods of the Waypoint class are either inline accessors for data, are too trivial for remark, or are outside the scope of this conversation (namely, serialization of the data).

5.1.2 Network Edges

Now that we have discussed the waypoints in a waypoint network, let us go over the details of the edges between the waypoints. Edges are unidirectional in our chapter demo implementation, though this need not be the case. To make a bidirectional edge in our implementation, we simply create two edges, one from the source to the destination, and one from the destination to the source. As such, edges in our implementation have only a destination waypoint, and not a source. In our implementation, a waypoint owns an edge, and it has a destination that it leads to.

Edges have two other important characteristics: an open flag, and a cost modifier. The open flag determines if the edge can be traversed. This is useful in game situations that have certain environmental conditions, such as a drawbridge. If the bridge is up, the edge is closed, if the bridge is down, the edge is open, and can be traversed. The same logic would apply to doors or areas in the game world that might be temporarily off limits (perhaps a chemical weapon was used in the area). The cost modifier allows us to control how often an edge will be traversed. In our demo, the cost of an edge is the Euclidean distance from the owner waypoint to the destination waypoint, multiplied by the cost modifier. So if we want the edge to be traversed more often, we make the modifier less than one but greater than or equal to zero. To traverse less often, we can make the cost modifier greater than one.

Before we look at the implementation of the network edge class in our demo, a final note on network edges is in order. If you were so inclined, you could have blind data on the edges, just like on the waypoints. You could use that information as you traversed the network to make additional decisions for your entity. We did not do it in our demo, but it is worth remembering. The nice thing about a system like this is that it is very flexible and you can really let your creativity carry you almost as far as you want to go.

The Network Edge Class

Now that we know what is involved with the network edges, let us look at the actual implementation of the network edge class in our demo.

```
//
// cNetworkEdge - an edge from one waypoint to another
//
class cNetworkEdge
{
public:
      cNetworkEdge(const tWaypointID &destination, float cost = 1.0f,
                   bool open = true);
      virtual ~cNetworkEdge();

      tWaypointID &GetDestination() { return mDestination; }
      const tWaypointID &GetDestination() const { return mDestination; }
      float GetCostModifier() const { return mCostModifier; }
      bool IsOpen() const { return mOpen; }

      void SetDestination(const tWaypointID &destination)
            { mDestination = destination; }
      void SetCostModifier(float costmod) { mCostModifier = costmod; }
      void SetIsOpen(bool open) { mOpen = open; }
      bool operator==(const cNetworkEdge &rhs);

      int   Serialize(ofstream &ar);
      int   UnSerialize(ifstream &ar);

protected:
      friend class cWaypoint;
      cNetworkEdge() {}

private:
      tWaypointID mDestination;
      float       mCostModifier;
      bool        mOpen;
};
```

This is not quite as complicated as the waypoint class, and again, the bulk of the interface is accessors. Let us take a closer look, starting once again with the data members.

```
      tWaypointID mDestination;
```

The destination of the edge uses the waypoint ID system to identify the destination we can reach. This ID is used to look up the waypoint in the network.

```
      float       mCostModifier;
```

The cost modifier is used to scale the base cost of the edge. As mentioned before, the base cost of the edge is the Euclidean distance from the owner waypoint to the destination waypoint.

```
      bool        mOpen;
```

This flag determines if this edge is currently traversable. Again, this is useful for such things as drawbridges or other passages which can become temporarily impassible.

Believe it or not, that is basically all there is to it. The class methods are basically just accessor methods or serialization methods, so we will not need to spend any time on those. Thus, on to the network!

5.1.3 The Waypoint Network

We have now seen the waypoints and we have examined the edges, so it is time to take a look at the network itself. As mentioned earlier, a waypoint network is simply a collection of waypoints and edges (a graph). In our implementation, the waypoints own the edges, so the network really just turns out to be a collection of waypoints and the class acts primarily as a waypoint manager.

The Waypoint Network Class

The waypoint network class is very straightforward. While it does contain the actual method for traversing the network, it really is not very complicated. So let us just dive right in, and see what we are getting ourselves into.

```
//
// cWaypointNetwork - a collection of waypoints and their associated edges
//
class cWaypointNetwork
{
public:
      cWaypointNetwork();
      virtual ~cWaypointNetwork();

      bool AddWaypoint(cWaypoint &waypoint);
      bool RemoveWaypoint(const tWaypointID &waypointID);
      void ClearWaypoints();
      cWaypoint *FindWaypoint(const tWaypointID &waypointID) const;

      bool FindPathFromWaypointToWaypoint(const tWaypointID &fromWaypoint,
                                          const tWaypointID &toWaypoint,
                                          tPath &path);
      bool FindPathFromPositionToPosition(const D3DXVECTOR3 &origin,
                                          const D3DXVECTOR3 &destination,
                                          const cWaypointVisibilityFunctor
                                                &visibilityFunc,
                                          tPath &path);

      void GetExtents(D3DXVECTOR3 &minimum, D3DXVECTOR3 &maximum) const;
      const tWaypointMap &GetWaypoints() const { return mWaypoints; }
```

```
      int   Serialize(ofstream &ar);
      int   UnSerialize(ifstream &ar);

private:
      bool FindClosestValidWaypointToPosition(const D3DXVECTOR3 &origin,
                                              const cWaypointVisibilityFunctor
                                                    &visibilityFunc,
                                              tWaypointID &result);
      float GoalEstimate(const tWaypointID &fromWaypoint,
                         const tWaypointID &toWaypoint);

      tWaypointMap mWaypoints;
};
```

There it is; the waypoint network. As you can see, it really is just a collection of waypoints. Let us take a look at the specifics.

```
      tWaypointMap mWaypoints;
```

The sole data member in the waypoint network is a map of waypoint IDs to waypoints. This lets us quickly find the waypoints we want using our IDs.

```
bool cWaypointNetwork::AddWaypoint(cWaypoint &waypoint)
{
      // don't add the waypoint if its GUID already exists
      tWaypointID id = waypoint.GetID();
      if (!FindWaypoint(id))
      {
            mWaypoints[id] = &waypoint;
            return true;
      }

      return false;
}
```

The AddWaypoint method does a quick check to see if the waypoint is already in the map, and if it is not, adds it to the map.

```
bool cWaypointNetwork::RemoveWaypoint(const tWaypointID &waypointID)
{
      // if the waypoint doesn't exist don't remove it
      tWaypointMap::iterator it = mWaypoints.find(waypointID);
      if (it == mWaypoints.end())
            return false;

      // otherwise free the waypoint, and remove it
      cWaypoint *wp = it->second;
      if (wp)
            delete wp;
      wp = NULL;
      mWaypoints.erase(it);
```

```
	return true;
}
```

The RemoveWaypoint method is slightly more complex, but not by much. It looks up the waypoint in the map, and if successful, deletes the waypoint, freeing its memory. It also removes the waypoint from the map.

```
void cWaypointNetwork::ClearWaypoints()
{
	// delete all waypoints, and clear the map
	for (tWaypointMap::iterator it = mWaypoints.begin();
	     it != mWaypoints.end(); ++it)
	{
		cWaypoint *wp = it->second;
		if (wp)
			delete wp;
		wp = NULL;
	}
	mWaypoints.clear();
}
```

The ClearWaypoints method simply iterates through all of the waypoints in the map, and deletes them and frees their memory. It then clears the map of its entries.

```
cWaypoint *cWaypointNetwork::FindWaypoint(const tWaypointID &waypointID) const
{
	// if we find the waypoint, return it
	tWaypointMap::const_iterator it = mWaypoints.find(waypointID);
	if (it == mWaypoints.end())
		return NULL;
	return it->second;
}
```

The FindWaypoint method looks the waypoint up in the map, and simply returns it if found.

```
void cWaypointNetwork::GetExtents(D3DXVECTOR3 &minimum, D3DXVECTOR3 &maximum) const
{
	minimum.x = minimum.y = minimum.z = FLT_MAX;
	maximum.x = maximum.y = maximum.z = -FLT_MAX;
	for (tWaypointMap::const_iterator it = mWaypoints.begin();
		it != mWaypoints.end(); ++it)
	{
		cWaypoint *wp = it->second;
		D3DXVECTOR3 pos = wp->GetPosition() + D3DXVECTOR3(wp->GetRadius(),
		                                                  wp->GetRadius(),
		                                                  wp->GetRadius());
		if (pos.x < minimum.x)
			minimum.x = pos.x;
		if (pos.y < minimum.y)
			minimum.y = pos.y;
		if (pos.z < minimum.z)
```

```
                  minimum.z = pos.y;
            if (pos.x > maximum.x)
                  maximum.x = pos.x;
            if (pos.y > maximum.y)
                  maximum.y = pos.y;
            if (pos.z > maximum.z)
                  maximum.z = pos.z;
      }
}
```

The GetExtents method iterates through all of the waypoints, and expands the minimum and maximum vectors to build a bounding box for the waypoint network.

There is one last method to discuss in the waypoint network class before delving into the pathfinding algorithm employed in the demo.

```
bool cWaypointNetwork::FindClosestValidWaypointToPosition
(
      const D3DXVECTOR3 &origin,
      const cWaypointVisibilityFunctor &visibilityFunc,
      tWaypointID &result
)
{
      float closestDistanceSq = FLT_MAX;
      bool foundClosest = false;

      // iterate through all the waypoints
      for (tWaypointMap::iterator it = mWaypoints.begin();
           it != mWaypoints.end(); ++it)
      {
            cWaypoint *wp = it->second;
            const D3DXVECTOR3 &pos = wp->GetPosition();
            // if we have a valid waypoint, and we can see it from this position
            if (wp && visibilityFunc.IsVisible(origin, pos))
            {
                  // check our distance to the waypoint
                  D3DXVECTOR3 vec = origin - pos;
                  float distsq = D3DXVec3LengthSq(&vec);

                  // if our distance to the waypoint is the closest we've found yet
                  if (distsq < closestDistanceSq)
                  {
                        // keep track of it
                        closestDistanceSq = distsq;
                        result = wp->GetID();
                        foundClosest = true;
                  }
            }
      }

      // return if we've found any close waypoints we can see
      return foundClosest;
}
```

FindClosestValidWaypointToPosition is a useful method employed during the pathfinding traversal of the network. We will discuss those methods in detail shortly, but first let us work out what this method does.

First the parameters of the method:

```
      const D3DXVECTOR3 &origin,
```

The method takes an origin point in space, which is the location from which we want to find the closest waypoint in the network.

```
      const cWaypointVisibilityFunctor &visibilityFunc,
```

The method also takes a visibility functor. This class is used to interface to your game system to provide visibility information between waypoints. Normally this is hooked up directly to a physics simulation system, which does ray casting into the physics representation of the world to see if you can draw a line from one point to another without running into anything (often called a "line of sight" test). The result is then used to determine if you can see between the points. If you can draw the line you can see from one point to the other; if not, you cannot. Let us take a quick look at that class.

```
//
// cWaypointVisibilityFunctor - a base class for determining visibility
// for network waypoints
//
class cWaypointVisibilityFunctor
{
public:
      cWaypointVisibilityFunctor() {}
      virtual ~cWaypointVisibilityFunctor() {}

      bool IsVisible(const D3DXVECTOR3 &origin,
                     const D3DXVECTOR3 &destination) const;
};
```

This is the interface for the waypoint visibility functor. The idea is that you would derive your own type and overload the IsVisible() method. That method would then do the line of sight test from the origin point to the destination point and return success if the line could be drawn.

Let us return now to the parameters of the FindClosestValidWaypointToPosition method…

```
      tWaypointID &result
```

The last parameter of the method is an address to a waypoint ID. This result will be set to the closest waypoint to the origin point that can be seen from the origin point (assuming there is one).

Now that we know a bit about the parameters, let us discuss the algorithm.

```
      float closestDistanceSq = FLT_MAX;
```

```
bool foundClosest = false;
```

First we set the closest distance squared value to the maximum float value. Thus, any distance will be less than this distance. We also set a flag noting we have not found a closest node yet.

```
// iterate through all the waypoints
for (tWaypointMap::iterator it = mWaypoints.begin();
     it != mWaypoints.end(); ++it)
```

We then start iterating through all of the waypoints. To be fair, this is not the best design strategy. Ideally, we would have a BSP or oct-tree that would assist us in finding the waypoint closest to our point. Since it could do a binary search, it would turn this search from an O(n) search to an $O(\log_2 n)$ search. The linear search is good enough for the demo however, since we do not have many waypoints to hunt through. 3D Graphics Programming Module II here at the Game Institute covers BSP trees, oct-trees, and a host of other hierarchical spatial data structures in great detail. Be certain to check out that course at some point in the not too distant future so that you can integrate a search tree into your application and realize the benefits.

```
cWaypoint *wp = it->second;
const D3DXVECTOR3 &pos = wp->GetPosition();
```

As we get each waypoint out of the map, we get its position.

```
// if we have a valid waypoint, and we can see it from this position
if (wp && visibilityFunc.IsVisible(origin, pos))
```

We then call our visibility functor to see if we can see the waypoint from the origin point. Something that the default implementation of the visibility functor does is just check from the origin to the center of the waypoint. Another possible solution would be to check the cone from the origin point to the destination waypoint, taking into account its radius. This is a more complex test however, so it was not used for the demo.

```
// check our distance to the waypoint
D3DXVECTOR3 vec = origin - pos;
float distsq = D3DXVec3LengthSq(&vec);
```

If we can see the waypoint, we compute the distance squared to the waypoint from our origin position. We use the squared result since we are just comparing the results in terms of magnitude, and as such it saves us a square root operation. Again, we could take into account the waypoint's radius if we wanted, but our demo does not require such an approach.

```
// if our distance to the waypoint is the closest we've found yet
if (distsq < closestDistanceSq)
```

We then test this computed distance to see if it is less than the closest waypoint distance we have found so far.

```
                        // keep track of it
                        closestDistanceSq = distsq;
                        result = wp->GetID();
                        foundClosest = true;
```

If we find that the newly computed distance is less than the closest distance we have found so far, we mark it as the closest distance we have found. We also store off the ID of the waypoint, and set our flag to say that we have found a closest waypoint.

```
    // return if we've found any close waypoints we can see
    return foundClosest;
```

After we have iterated across all the waypoints, we return our flag. The caller will check this flag to see if the waypoint ID reference passed in will contain the closest waypoint or not.

5.2 Navigating the Waypoint Network

Now that we have a waypoint network, navigating it is simple. In the demo, we use the same A* algorithm as in our initial pathfinding demo, only it has been modified to traverse the waypoint network structure instead of a fixed grid. Believe it or not, this actually simplifies the code somewhat. Let us take a look at what needs to be done.

There are three separate cases that can occur when computing a path for an entity:

a. The entity is traveling from a point on the network to a point on the network.
b. The entity is traveling from a point off the network to a point on the network.
c. The entity is traveling to or from a point on the network to or from a point off the network.

When we say “on the network” versus “off the network,” we mean that an entity that is “on the network” is actually within the bounds of a waypoint; whereas “off the network” means the entity is not within the bounds of an actual waypoint.

If we are dealing with case (a), then the entity can use the FindPathFromWaypointToWaypoint method directly, using the waypoint it is starting at, and the waypoint it is going to as the parameters.

If we are dealing with case (b), then the entity uses the FindPathFromPointToPoint method, which internally finds the closest waypoints to the start and end, and then uses FindPathFromWaypointToWaypoint to compute the path between those waypoints.

If we are dealing with case (c), then the entity again uses the FindPathFromPointToPoint method passing the position of the waypoint the entity is within or the position of the waypoint that is the goal; whichever we have a waypoint for. It would be an optimization to provide a method that could fast path the finding of the known waypoints, but that was not done for this demo. Primarily in this demo, the FindPathFromPointToPoint was used.

Let us take a look at the implementation of this method.

```
bool cWaypointNetwork::FindPathFromPositionToPosition
(
      const D3DXVECTOR3 &origin,
      const D3DXVECTOR3 &destination,
      const cWaypointVisibilityFunctor &visibilityFunc,
      tPath &path
)
{
      tWaypointID closestToOrigin;
      tWaypointID closestToDestination;

      // find the waypoint closest to the starting position
      if (!FindClosestValidWaypointToPosition(origin, visibilityFunc,
                                              closestToOrigin))
            return false;
      // find the waypoint closest to the destination position
      if (!FindClosestValidWaypointToPosition(destination, visibilityFunc,
                                              closestToDestination))
            return false;

      // clear the path
      path.clear();

      // if the starting waypoint is the same as the ending waypoint,
      // we can just walk straight to the
      // destination point
      if (closestToOrigin == closestToDestination)
            return true;

      return FindPathFromWaypointToWaypoint(closestToOrigin, closestToDestination,
                                            path);
}
```

Here is the algorithm in its entirety. It is not overly complex, so let us just walk through it right here. The method takes a position for the origin, a position for the destination, a visibility functor, and a reference to a path. The tPath typedef is simply an STL list of waypoint ID objects. We first try to find the closest valid waypoint to the origin. If we do not find one, we fail to make a path. We then find the closest valid waypoint to the destination. Again, if we do not find one, we fail to make a path. Next we clear the path, and early abort in the event the closest waypoint to the origin is the same waypoint as the destination. In that case, we can just go directly from the origin to the destination without acquiring the network. If we have a closest waypoint to the origin, and a closest waypoint to the destination, and they are not the same, we call upon the FindPathFromWaypointToWaypoint method to compute a path for us. We will talk about that method in a moment.

Before we get going on the FindPathFromWaypointToWaypoint, we should discuss a helper class, the cAStarWaypointNode class.

```
//
// cAStarWaypointNode - a special node for scoring the weights for determining
// pathing through a waypoint network
```

```
// using A*
//
class cAStarWaypointNode
{
public:
	cAStarWaypointNode(const tWaypointID &id);
	cAStarWaypointNode(const tWaypointID &id, float f, float g, float h);

	float GetCost() const { return m_f; }
	void GetCosts(float &f, float &g, float &h) const
		{ f = m_f; g = m_g; h = m_h; }
	void SetCost(float cost) { m_f = cost; }
	void SetCosts(float f, float g, float h) { m_f = f; m_g = g; m_h = h; }

	cAStarWaypointNode *GetParent() const { return mParent; }
	void SetParent(cAStarWaypointNode *parent) { mParent = parent; }

	bool GetVisited() const { return mVisited; }
	void SetVisited(bool visited) { mVisited = visited; }

	const tWaypointID &GetWaypoint() const { return mWaypoint; }

	bool operator<(const cAStarWaypointNode &rhs) const;

private:
	tWaypointID mWaypoint;
	cAStarWaypointNode *mParent;
	bool		mVisited;
	float		m_f;
	float		m_g;
	float		m_h;
};
```

This class is used for managing the traversal of the network. This is done so we do not have to keep track of heuristic estimate costs and visited status on the waypoints themselves. It is very similar to the A* node classes we studied earlier in the course, but it is worth mentioning. Just as before, the class maintains the *f*, *g* and *h* values for computing the actual cost of the node, and it keeps track of the parent node that got us here. Once we find the path, we walk from end to start via the parent node pointers. Enough of the easy stuff; onto the pathfinding!

```
bool cWaypointNetwork::FindPathFromWaypointToWaypoint
(
	const tWaypointID &fromWaypoint,
	const tWaypointID &toWaypoint,
	tPath &path
)
{
	// ye verily do traversal of network here
	tAStarWaypointNodePriorityQueue open;
	tAStarWaypointNodeList closed;
	cAStarWaypointNode *n;
	cWaypoint *nwp;
	tNodeMap nodeMap;
```

```
	// seed the search with the starting point
	float g = 0.0f;
	float h = GoalEstimate(fromWaypoint, toWaypoint);
	float f = g + h;
	cAStarWaypointNode *node = new cAStarWaypointNode(fromWaypoint, f, g, h);
	nodeMap[fromWaypoint] = node;
	open.push_back(node);

	// now iterate the search
	while(!open.empty())
	{
		n = open.front();
		nwp = FindWaypoint(n->GetWaypoint());
		open.pop_front();

		if (n->GetWaypoint() == toWaypoint)
		{
			path.clear();
			node = n;
			while (node)
			{
				path.push_front(node->GetWaypoint());
				node = node->GetParent();
			}
			return true;
		}

		for (tEdgeList::iterator it = nwp->GetEdges().begin();
		     it != nwp->GetEdges().end(); ++it)
		{
			cNetworkEdge &edge = *it;
			cWaypoint *dest = FindWaypoint(edge.GetDestination());
			if (!dest || !edge.IsOpen())
				continue;

			n->GetCosts(f, g, h);
			float newg = g + nwp->GetCostForEdge(edge, *this);

			// first check the node map, if we don't have an
			// entry in the node map, we've never been here
			// before.
			tNodeMap::iterator nodeIt = nodeMap.find(edge.GetDestination());
			bool wasInMap = true;
			if (nodeIt == nodeMap.end())
			{
				// never been here
				wasInMap = false;

				// add it to the map
				node = new cAStarWaypointNode(edge.GetDestination());
				nodeMap[edge.GetDestination()] = node;
			}
			else
				node = nodeIt->second;

			node->GetCosts(f, g, h);
```

```
                  if
                  (
                        wasInMap &&
                        (open.contains(node) || closed.contains(node))
                        && g <= newg
                  )
                  {
                        // do nothing... we are already in the queue
                        // and we have a cheaper way to get there...
                  }
                  else
                  {
                        node->SetParent(n);
                        g = newg;
                        h = GoalEstimate(node->GetWaypoint(), toWaypoint);
                        f = g + h;
                        node->SetCosts(f, g, h);

                        if(closed.contains(node))
                        {
                              // remove it
                              closed.remove_item(node);
                        }

                        if(!open.contains(node))
                        {
                              open.add_item(node);
                        }
                        else
                        {
                              // update this item's position in the queue
                              // as its cost has changed
                              // and the queue needs to know about it
                              open.sort();
                        }
                  }
            }

            closed.add_item(n);
      }

      // unable to find a route
      return false;
}
```

This is obviously the core of our pathfinding on the waypoint network. This algorithm should look familiar, as it is the A* algorithm we have already discussed in this course. Even so, there are some small changes. Most importantly, it does the traversal all in one fell swoop rather than one iteration at a time like our last demo did. Let us take a look at what is going on.

```
bool cWaypointNetwork::FindPathFromWaypointToWaypoint
(
      const tWaypointID &fromWaypoint,
      const tWaypointID &toWaypoint,
```

```
	tPath &path
)
```

The method takes a waypoint ID of the waypoint to start at, a waypoint ID of the waypoint to end at, and a reference to a path. The method returns true if a path was found, and false it no path from the starting waypoint to the ending waypoint can be found.

```
	// ye verily do traversal of network here
	tAStarWaypointNodePriorityQueue open;
```

The algorithm begins with a priority queue of A* waypoint nodes which is the "open" queue of nodes to examine.

```
	tAStarWaypointNodeList closed;
```

Next we have a list of A* waypoint nodes which is the "closed" list of already examined waypoints.

```
	cAStarWaypointNode *n;
```

We have an A* waypoint node object which is the node we are currently traversing.

```
	cWaypoint *nwp;
```

We also have a waypoint pointer which is the waypoint associated with the A* waypoint node object we are currently traversing.

```
	tNodeMap nodeMap;
```

Last, we have a node map. This map associates waypoint IDs to A* waypoint nodes. This also helps to let us know if we have visited a node or not.

```
	// seed the search with the starting point
	float g = 0.0f;
	float h = GoalEstimate(fromWaypoint, toWaypoint);
	float f = g + h;
	cAStarWaypointNode *node = new cAStarWaypointNode(fromWaypoint, f, g, h);
	nodeMap[fromWaypoint] = node;
	open.push_back(node);
```

The first thing we do is seed the search with the starting point. We estimate the distance from the start to the goal using our heuristic, create an A* waypoint node for this waypoint, associate the waypoint ID to the A* node in our map, and push the A* node onto our open queue. This gets us ready for the main loop.

```
	// now iterate the search
	while(!open.empty())
```

So long as we have nodes to investigate in our open queue, we will perform the search.

```
n = open.front();
nwp = FindWaypoint(n->GetWaypoint());
open.pop_front();
```

During every iteration, we grab the top node off the priority queue, find the waypoint for the ID the node is associated with, and pop the node off the queue. We use a special derived version of the STL list container for our priority queue. We will not go into detail on how that works, as this is basic algorithms theory. The implementation lives in WaypointNetwork.h if you feel inclined to peruse it.

```
if (n->GetWaypoint() == toWaypoint)
{
      path.clear();
      node = n;
      while (node)
      {
            path.push_front(node->GetWaypoint());
            node = node->GetParent();
      }
      return true;
}
```

Next we see if the current node we popped off the queue happens to be the goal node. If the current node is the goal node, we clear out our path, and walk the parent pointers of the A* nodes, pushing the nodes onto the path in reverse order. This way the path has the nodes in the order they should be visited, not the other way around.

```
for (tEdgeList::iterator it = nwp->GetEdges().begin();
     it != nwp->GetEdges().end(); ++it)
```

Assuming we have not reached the goal node, we iterate across all of the edges of the current node.

```
      cNetworkEdge &edge = *it;
      cWaypoint *dest = FindWaypoint(edge.GetDestination());
      if (!dest || !edge.IsOpen())
            continue;
```

For each edge, we find the destination waypoint in the network. If we cannot find the destination, or the edge is closed, we skip this edge.

```
      n->GetCosts(f, g, h);
      float newg = g + nwp->GetCostForEdge(edge, *this);
```

Assuming we have a traversable edge with a valid destination waypoint, we get the costs for the current A* node, and compute a new g using the current g, and the cost for this edge.

```
      // first check the node map, if we don't have an
      // entry in the node map, we've never been here
```

```
                    // before.
                    tNodeMap::iterator nodeIt = nodeMap.find(edge.GetDestination());
                    bool wasInMap = true;
                    if (nodeIt == nodeMap.end())
                    {
                          // never been here
                          wasInMap = false;

                          // add it to the map
                          node = new cAStarWaypointNode(edge.GetDestination());
                          nodeMap[edge.GetDestination()] = node;
                    }
                    else
                          node = nodeIt->second;
```

Next we check to see if the destination waypoint has a representing A* node in the map yet. If it does not, we create a new A* node, and associate it with the destination waypoint ID in the map. If it does exist, we simply get the A* node from the map for destination waypoint ID.

```
                    node->GetCosts(f, g, h);
```

We then get the current costs for the destination waypoint's A* node.

```
                    if
                    (
                         wasInMap &&
                         (open.contains(node) || closed.contains(node))
                         && g <= newg
                    )
                    {
                          // do nothing... we are already in the queue
                          // and we have a cheaper way to get there...
                    }
```

Here we check to see if we should update the destination's A* node. If the node was previously in the map, and either the open or closed lists contain the node, and the existing *g* cost is less or equal the new *g* cost computed, we do nothing.

```
                          node->SetParent(n);
                          g = newg;
                          h = GoalEstimate(node->GetWaypoint(), toWaypoint);
                          f = g + h;
                          node->SetCosts(f, g, h);
```

If we have determined that we need to update the destination A* node, we set its parent to the current node. We also set its *g* to the new *g* computed, compute the heuristic estimate for this node, and set the costs.

```
                          if(closed.contains(node))
                          {
                                // remove it
```

```
                        closed.remove_item(node);
                  }
```

Then, if we find the node in the closed list, we remove it from the closed list, as it needs re-examination.

```
                  if(!open.contains(node))
                  {
                        open.add_item(node);
                  }
                  else
                  {
                        // update this item's position in the queue
                        // as its cost has changed
                        // and the queue needs to know about it
                        open.sort();
                  }
```

If the open queue does not contain the node, we add it to the open queue. This action automatically keeps the queue sorted. If the open queue does contain the node however, we simply resort the open queue. This will ensure the node is properly placed in the queue based on its priority.

```
            closed.add_item(n);
```

After each of the edges for the current node has been visited, the current node is added to the closed list.

```
      // unable to find a route
      return false;
```

Lastly, if after exhausting the open queue, we do not find a path and return above, we return false, reporting failure to build the path.

That is all there is to finding a path using a waypoint network. Given your experience with A* earlier in the course, this should be second nature to you.

From a theoretical perspective, you now have all of the information you need to take your waypoint networks and use them to find paths from place to place in the game world. In the next few sections we are going to talk about how we decided to use this new system in our chapter demo. Our discussions will take us back to other topics covered in Chapters 3 and 4 so that we can see how these pieces can fit together to produce interesting results. This should provide you with a foundation to work from so that you can begin integrating your own ideas that suit your particular game.

5.3 Flocking and Waypoint Networks

In Chapter 3 when we were talking about flocking, we discussed the concept of behavior based movement. The idea is that you have a set of separate behaviors and that each could contribute to the final desired movement of the entity. Following up on that concept, our demo makes use of the behavioral movement system developed in the flocking demo, and creates a movement behavior that follows a waypoint network path.

5.3.1 The Pathfind Behavior

```
class cPathfindBehavior : public cBehavior
{
public:
                        cPathfindBehavior(float turnRate, float goalRadius,
                                          float avoidDist,
                                          float maxTimeBeforeAgitation,
                                          const D3DXVECTOR3 &upVector,
                                          cWaypointNetwork &waypointNetwork);
      virtual           ~cPathfindBehavior(void);

      virtual void      Iterate(float timeDelta, cEntity &entity);
      void              ApplyAvoidance(cEntity &entity);

      virtual string    Name(void) { return("Pathfind Behavior"); }

protected:
      float mTurnRate;
      float mGoalRadius;
      float mAvoidDist;
      float mMaxTimeBeforeAgitation;
      D3DXVECTOR3 mUpVector;
      cWaypointNetwork &mWaypointNetwork;
};
```

Here we see our pathfind behavior. It has a decent number of data members, so we will examine those first.

```
      float mTurnRate;
```

Just as with many of our movement behaviors from the flocking demo, this behavior has a turning rate parameter which limits how quickly the entity can make turns.

```
      float mGoalRadius;
```

The goal radius parameter is used to determine if the entity has satisfactorily reached its goal. Bear in mind the goal is different than the current waypoint. The goal is the final destination, while the current waypoint is the next one.

```
	float mAvoidDist;
```

The avoid distance is the distance at which the entities strive to remain apart from one another. More on this value later.

```
	float mMaxTimeBeforeAgitation;
```

The max time before agitation is another data member we will discuss a bit later. Basically it has to do with keeping the entity from getting stuck trying to reach the next waypoint.

```
	D3DXVECTOR3 mUpVector;
```

This is the vector which is "up" in the game world. We will discuss it in more detail later, but for now just know that it is used in conjunction with the max time before agitation variable.

```
	cWaypointNetwork &mWaypointNetwork;
```

This is a reference to the network we will be navigating. We need this to look up waypoints from IDs.

Now that we have a good idea what kinds of data this behavior needs, let us look at how it does its job.

```
void cPathfindBehavior::Iterate(float timeDelta, cEntity &entity)
{
	// pathfinding only works on squad mate type entities!
	cSquadEntity &squadmate = dynamic_cast<cSquadEntity&>(entity);

	tPath &path = squadmate.GetPath();
	tWaypointID wpID = squadmate.GetNextWaypoint();
	D3DXVECTOR3 entityPos = entity.Position();
	D3DXVECTOR3 desiredMoveAdj(0.0f, 0.0f, 0.0f);

	if (wpID == GUID_NULL && path.size() > 0)
	{
		// set the next waypoint!
		wpID = path.front();
		path.pop_front();
		squadmate.SetNextWaypoint(wpID);
		squadmate.ResetTimeSinceWaypointReached();
	}
	else if (wpID == GUID_NULL || path.empty())
	{
		desiredMoveAdj = squadmate.GetGoal() - entityPos;
		if (D3DXVec3Length(&desiredMoveAdj) < mGoalRadius)
		{
			// we made it... stand around
			entity.SetDesiredMove(-entity.Velocity());
			ApplyAvoidance(entity);
			squadmate.ResetTimeSinceWaypointReached();
			return;
		}
	}
```

```
	cWaypoint *wp = mWaypointNetwork.FindWaypoint(wpID);

	if (wp)
	{
		D3DXVECTOR3 wppos = wp->GetPosition();
		desiredMoveAdj = wppos - entityPos;
		if (D3DXVec3Length(&desiredMoveAdj) < wp->GetRadius())
		{
			// close enough! next waypoint!
			if (path.size() > 0)
			{
				// set the next waypoint!
				wpID = path.front();
				path.pop_front();
				squadmate.SetNextWaypoint(wpID);
				wp = mWaypointNetwork.FindWaypoint(wpID);
				ASSERT(wp != NULL);
				wppos = wp->GetPosition();
				desiredMoveAdj = wppos - entityPos;
				squadmate.ResetTimeSinceWaypointReached();
			}
			else
			{
				// uh, no more waypoints,
				// start walking toward our target position
				desiredMoveAdj = squadmate.GetGoal() - entityPos;
				squadmate.SetNextWaypoint(GUID_NULL);
				if (D3DXVec3Length(&desiredMoveAdj) < mGoalRadius)
				{
					// we made it... stand around
					entity.SetDesiredMove(-entity.Velocity());
					ApplyAvoidance(entity);
					squadmate.ResetTimeSinceWaypointReached();
					return;
				}
			}
		}
	}

	// move in the direction of your next pathnode or your goal position
	squadmate.IncrementTimeSinceWaypointReached(timeDelta);
	D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
	D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
	desiredMoveAdj *= mTurnRate;
	currentDesiredMove += desiredMoveAdj * Gain();
	if (squadmate.GetTimeSinceWaypointReached() > mMaxTimeBeforeAgitation)
	{
		// agitate the movement to get the guy moving properly
		D3DXVECTOR3 agitationVector;

		// nudge the desired move vector by using a vector perpendicular to the
		// current desired move's direction.
		D3DXVec3Cross(&agitationVector, &currentDesiredMove, &mUpVector);
		currentDesiredMove = agitationVector;
		squadmate.ResetTimeSinceWaypointReached();
	}
```

```
	entity.SetDesiredMove(currentDesiredMove);
	ApplyAvoidance(entity);
}
```

That was a fairly big method, so we will tackle it a bit at a time.

```
void cPathfindBehavior::Iterate(float timeDelta, cEntity &entity)
```

We begin with the prototype. The behavior requires the time passed since the last iteration as well as the entity upon which to iterate.

```
	// pathfinding only works on squad mate type entities!
	cSquadEntity &squadmate = dynamic_cast<cSquadEntity&>(entity);
```

This particular implementation only works on the special squad entities derived for this demo. So we make sure that is the case here.

```
	tPath &path = squadmate.GetPath();
	tWaypointID wpID = squadmate.GetNextWaypoint();
	D3DXVECTOR3 entityPos = entity.Position();
	D3DXVECTOR3 desiredMoveAdj(0.0f, 0.0f, 0.0f);
```

First, we get the squad mate's path, his next waypoint ID, his position, and initialize the desired movement adjustment to be nil.

```
	if (wpID == GUID_NULL && path.size() > 0)
	{
		// set the next waypoint!
		wpID = path.front();
		path.pop_front();
		squadmate.SetNextWaypoint(wpID);
		squadmate.ResetTimeSinceWaypointReached();
	}
```

If the waypoint ID is NULL, and we have some nodes left in our path, we get the next waypoint from the path, and set it on the squad mate. We also reset a timer which keeps track of how long it has been since we reached a waypoint.

```
	else if (wpID == GUID_NULL || path.empty())
	{
		desiredMoveAdj = squadmate.GetGoal() - entityPos;
		if (D3DXVec3Length(&desiredMoveAdj) < mGoalRadius)
		{
			// we made it... stand around
			entity.SetDesiredMove(-entity.Velocity());
			ApplyAvoidance(entity);
			squadmate.ResetTimeSinceWaypointReached();
			return;
		}
	}
```

If our waypoint ID is NULL, or our path is empty, we are moving directly towards our goal position. So we compute our desired move adjustment to be the vector to the goal from our position. If that vector's length is less than the goal's radius, we have reached the goal. In that case, we set our velocity to bring us to a stop by negating it, apply some avoidance measures (which we will cover later), and again reset the amount of time it has been since we last reach a waypoint. We then return out of the method.

```
	cWaypoint *wp = mWaypointNetwork.FindWaypoint(wpID);
```

Assuming we have not reached our goal, we find the waypoint for our current waypoint ID.

```
	if (wp)
```

Assuming we have a waypoint…

```
		D3DXVECTOR3 wppos = wp->GetPosition();
		desiredMoveAdj = wppos - entityPos;
```

We get the position of the waypoint, and set our desired movement adjustment to be the vector from our current position to the waypoint's position.

```
		if (D3DXVec3Length(&desiredMoveAdj) < wp->GetRadius())
```

If the magnitude of that vector is less than our waypoint's radius, we have reached it. So…

```
			// close enough! next waypoint!
			if (path.size() > 0)
			{
				// set the next waypoint!
				wpID = path.front();
				path.pop_front();
				squadmate.SetNextWaypoint(wpID);
				wp = mWaypointNetwork.FindWaypoint(wpID);
				ASSERT(wp != NULL);
				wppos = wp->GetPosition();
				desiredMoveAdj = wppos - entityPos;
				squadmate.ResetTimeSinceWaypointReached();
			}
```

If we have path nodes left in our path, we grab the next waypoint from the path, and set the squad mate's next waypoint to be that ID. We then find the waypoint for this new waypoint ID, and set our desired movement vector to be the vector from our current position to this new waypoint's position. We then reset the amount of time it has been since this squad mate last reached a waypoint.

```
			else
			{
				// uh, no more waypoints,
				// start walking toward our target position
				desiredMoveAdj = squadmate.GetGoal() - entityPos;
				squadmate.SetNextWaypoint(GUID_NULL);
```

```
                        if (D3DXVec3Length(&desiredMoveAdj) < mGoalRadius)
                        {
                              // we made it... stand around
                              entity.SetDesiredMove(-entity.Velocity());
                              ApplyAvoidance(entity);
                              squadmate.ResetTimeSinceWaypointReached();
                              return;
                        }
                  }
```

Otherwise, if we have no more nodes left in our path, we start walking towards our goal. We set the desired movement adjustment to be the vector from our current position to our goal. We also NULL out our squad mate's next waypoint ID. If the magnitude of the vector from our current position to the goal is greater than the goal radius, we have reached the goal. At that point, we again negate our velocity and set it to be our desired movement adjustment to bring us to a halt. We apply some avoidance, and we reset the time since this squad mate last reached a waypoint. We then return from the method.

```
      // move in the direction of your next pathnode or your goal position
      squadmate.IncrementTimeSinceWaypointReached(timeDelta);
      D3DXVECTOR3 currentDesiredMove = entity.DesiredMove();
      D3DXVec3Normalize(&desiredMoveAdj, &desiredMoveAdj);
      desiredMoveAdj *= mTurnRate;
      currentDesiredMove += desiredMoveAdj * Gain();
      //
      // some mojo here…
      //
      entity.SetDesiredMove(currentDesiredMove);
      ApplyAvoidance(entity);
```

If we have not aborted from reaching a goal, we actually apply some clamps to the desired movement adjustments. But first, we increment the amount of time it has been since this squad mate reached a waypoint. We get our current desired move, normalize our desired movement adjustment, and apply the turn rate scalar. Then we add our desired move adjustment, scaled by the behavior gain, to the current desired move.

At this point, we apply some mojo. We will go over exactly what that is shortly.

Lastly, we set the desired move to the newly computed desired move and we apply some avoidance. We will talk about how that works very shortly.

Getting Stuck

There is a little complication that must be taken into account when applying turning radius limits. Imagine driving a car at 10 mph. The car corners pretty well and you can almost turn on a dime. Now start driving 50 mph. You cannot turn on a dime anymore because the car has a turning radius.

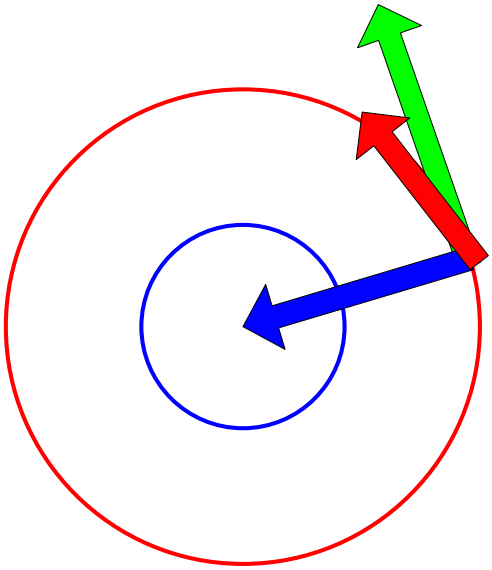

Figure 5.3

Take a look at Figure 5.3. If the car was trying to get into the blue circle, and its current velocity was in the direction of the green arrow, then the adjusted velocity would be the blue arrow. However, since the car has a turning radius, the red arrow is the adjusted velocity. This rate limit just keeps carrying it around the red circle! We never get to the blue circle because we keep trying to turn the same amount, while not slowing down.

Sadly, our entities can suffer from this same problem. But there is a cure! By keeping track of how long it has been since we last reached the waypoint, we can detect if it has been an unacceptably long period of time since we made progress. If it has been too long, we can do something about it -- perturb their velocity.

Enter the mojo…

```
	if (squadmate.GetTimeSinceWaypointReached() > mMaxTimeBeforeAgitation)
	{
		// agitate the movement to get the guy moving properly
		D3DXVECTOR3 agitationVector;

		// nudge the desired move vector by using a vector perpendicular to the
		// current desired move's direction.
		D3DXVec3Cross(&agitationVector, &currentDesiredMove, &mUpVector);
		currentDesiredMove = agitationVector;
```

```
            squadmate.ResetTimeSinceWaypointReached();
        }
```

Right before we set our desired movement, we check to make sure that the amount of time it has been since this entity last reached a waypoint is not greater than the max time before we agitate his motion. If it is, we cross the desired movement vector with the up vector, and get a vector perpendicular to our current motion. We then use that vector as our desired movement vector, for one and only one tick. This perturbs the motion of the entity just enough, and he can recover from the circular pattern we just witnessed.

5.3.2 A Word on Avoidance

There is one last bit remaining unexplained in the pathfinding behavior, and that is the ApplyAvoidance behavior. While we could have just used the avoidance behavior from the flocking demo, it did not provide exactly the same kind of avoidance that we wanted in this application. One thing to bear in mind is that avoidance is *not* supposed to keep the entities from ever running into each other. In order to do that, we would need to run a full scale simultaneous solve of the positions and velocities of the entities, and resolve interpenetrations. We did not want to worry about that in this demo since it introduces systems that are beyond the scope of this course. The idea here is that a proper collision system would keep you from being on top of one another, while the avoidance behavior would attempt minimize the number of times they keep running into each other. So the systems work in tandem. Discussion of a full blown collision system can be found in 3D Graphics Programming Module II.

In a nutshell, the avoidance behavior checks to see if any entities are too close, and if so, computes an adjustment vector to move the entity on a travel path tangent to the other entities' periphery.

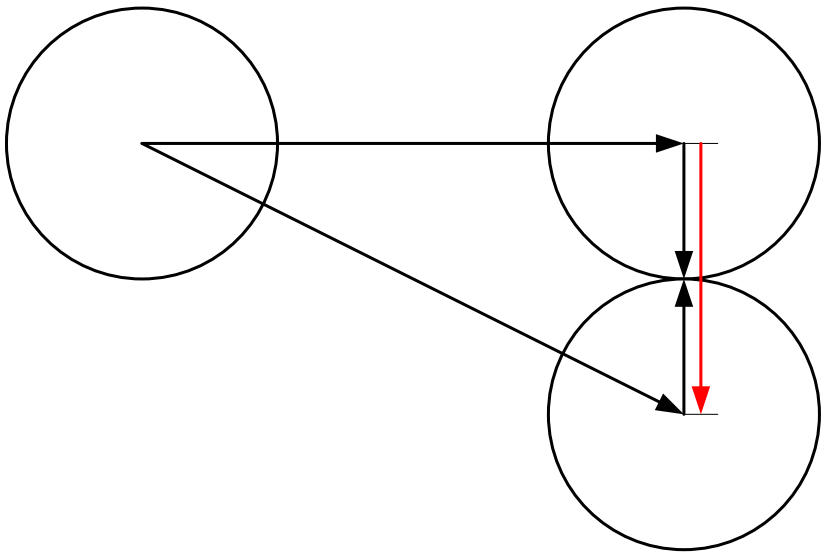

Figure 5.4

Looking at Figure 5.4, we compute the vector from Entity 1 to Entity 2. We then know if we want Entity 1 to pass by the side of Entity 2 without colliding, we should move in the direction from Entity 1 to Entity 1b. We can compute this by calculating $\theta = \tan^{-1}\frac{2r}{\|v\|}$. We then rotate v by θ to get the vector pointing in the direction we need to go. Let us look at how the code does this.

```
void cPathfindBehavior::ApplyAvoidance(cEntity &entity)
{
      // pathfinding only works on squad mate type entities!
      cSquadEntity &squadmate = dynamic_cast<cSquadEntity&>(entity);

      D3DXVECTOR3 entityPos(entity.Position());
      D3DXVECTOR3 currentDesiredMove(entity.DesiredMove());

      // let's make sure we aren't bound to hit anyone else
      cWorld &world = squadmate.World();
      for (tGroupList::iterator git = world.Groups().begin();
          git != world.Groups().end(); ++git)
      {
            cGroup *grp = *git;
            for (tEntityList::iterator eit = grp->Entities().begin();
                eit != grp->Entities().end(); ++eit)
            {
                  cEntity *e = *eit;
                  if (e == &entity)
                        continue;

                  D3DXVECTOR3 otherEntityPos(e->Position());
                  D3DXVECTOR3 toEntity = otherEntityPos - entityPos;
                  if (D3DXVec3Length(&toEntity) < mAvoidDist)
                  {
                        // let's apply some avoidance.
                        D3DXVec3Normalize(&toEntity, &toEntity);
                        float zRotation = atan2f(toEntity.y, toEntity.x);
                        D3DXMATRIX rotationMat;
                        D3DXMatrixRotationZ(&rotationMat, zRotation);
                        D3DXVECTOR4 rotatedDesiredMove;
                        D3DXVec3Transform(&rotatedDesiredMove,
                                          &currentDesiredMove, &rotationMat);
                        D3DXVECTOR3 newDesiredMove(rotatedDesiredMove.x,
                                                   rotatedDesiredMove.y,
                                                   rotatedDesiredMove.z);
                        D3DXVECTOR3 desiredMoveAdj = newDesiredMove -
                                                     currentDesiredMove;
                        desiredMoveAdj *= mTurnRate;
                        currentDesiredMove += desiredMoveAdj * Gain();
                        entity.SetDesiredMove(currentDesiredMove);
                  }
            }
      }
}
```

The method looks long, but the iteration takes up the bulk of the space.

```
	// pathfinding only works on squad mate type entities!
	cSquadEntity &squadmate = dynamic_cast<cSquadEntity&>(entity);
```

First off, this method only works on our special derived squad entity type of entity. So check that first.

```
	D3DXVECTOR3 entityPos(entity.Position());
	D3DXVECTOR3 currentDesiredMove(entity.DesiredMove());
```

Next we get the current position and current desired move of the entity.

```
	// let's make sure we aren't bound to hit anyone else
	cWorld &world = squadmate.World();
	for (tGroupList::iterator git = world.Groups().begin();
	     git != world.Groups().end(); ++git)
	{
	      cGroup *grp = *git;
	      for (tEntityList::iterator eit = grp->Entities().begin();
	           eit != grp->Entities().end(); ++eit)
	      {
```

We then iterate through all the groups in the world, and each of the entities in each group.

```
	                cEntity *e = *eit;
	                if (e == &entity)
	                      continue;
```

We then perform a sanity check, to ensure we are not trying to avoid ourselves.

```
	                D3DXVECTOR3 otherEntityPos(e->Position());
	                D3DXVECTOR3 toEntity = otherEntityPos - entityPos;
```

Now we compute a vector from this entity to the other entity.

```
	                if (D3DXVec3Length(&toEntity) < mAvoidDist)
```

If the magnitude of the vector is less than our avoid distance, we need to try to avoid this entity.

```
	                      // let's apply some avoidance.
	                      D3DXVec3Normalize(&toEntity, &toEntity);
	                      float zRotation = atan2f(toEntity.y, toEntity.x);
	                      D3DXMATRIX rotationMat;
	                      D3DXMatrixRotationZ(&rotationMat, zRotation);
```

First we normalize the vector to the entity, and compute the rotation based on the vector. We then build a rotation matrix using this vector. Note that this method is useful only in avoiding things in 2d, where Z is up.

```
	                      D3DXVECTOR4 rotatedDesiredMove;
	                      D3DXVec3Transform(&rotatedDesiredMove,
```

```
                                &currentDesiredMove, &rotationMat);
                    D3DXVECTOR3 newDesiredMove(rotatedDesiredMove.x,
                                               rotatedDesiredMove.y,
                                               rotatedDesiredMove.z);
```

Next we perform a little data hoop jumping. D3DXVec3Transform puts the results in a D3DXVECTOR4, because it wants to preserve the *w* value. That is fine, but there are no convenient operators to turn a vector 4 back to a vector 3 without constructing one by hand. So we rotate our current desired move vector by the rotation matrix we computed and then we put it into a usable vector.

```
                    D3DXVECTOR3 desiredMoveAdj = newDesiredMove -
                                                 currentDesiredMove;
                    desiredMoveAdj *= mTurnRate;
                    currentDesiredMove += desiredMoveAdj * Gain();
                    entity.SetDesiredMove(currentDesiredMove);
```

Now we get a new desired move adjustment vector by getting the vector from our current desired move to new current desired move, apply our turn rate limit, and then add it into the our current desired move, making use of the behavior gain. Finally, we then set the desired move using computed vector.

That is it for the behavioral movement component in the demo. Now we are ready to see how the squad members and squad leaders make their decisions about where to go in the world.

5.4 Squads and State Machines

So now we have waypoint networks, ways to pathfind through them, and a movement behavior to get our entities to follow a path. All that is left are the squad members themselves and how they decide what to do.

In the last chapter, we discussed state machines as decision systems, and scripting as a means to extend what our games can do in a data driven way. We now take our next step, and integrate it into the demo with pathfinding and waypoint networks.

5.4.1 Methods of Squad Communication

In our demo, we will have a squad leader and three squad members. The squad leader tells the squad members what to do, and they do it. In truth, it does not get much simpler than this. So how do they manage this communication? There are a few typical ways to implement squad communication.

Direct Control

In this method, the squad leader directly calls methods to modify variables or cause transitions in the squad member's state machine (or whatever decision system you are using). It is a simple approach, and the one we are using for our demo, but it is not the most flexible. The squad leader has to know all about the squad members, and make them do the right things to get them to behave as he wants them to. Basically the squad leader needs to know what to do, and how to do it.

Poll the Leader

Another approach is to have the squad members poll the leader to find out what he wants them to do. This pushes the responsibility of knowing how to do it onto the squad member. This is a slightly better design than the last one since it gives the squad members more autonomy and allows for more variety at the squad member level. The downside here is the squad members now need to know all about the squad leader. Although, in fairness, 'many to one' is often easier to manage than 'one to many' since the group members only need to know about one type of object – the leader.

Events

The last approach worth mentioning is an event system. The idea is that the squad leader decides what he wants to have happen, and sends an event to the squad members. They receive the event, process it, and decide how to do what the squad leader wants. When they are done, or if something comes up that requires them to seek redirection from the squad leader, they send an event back to the squad leader, who receives it, processes it, and sends an event back. It is a more complicated system, but fairly general and flexible.

5.4.2 The Squad Member

First, let us talk about the squad member itself, and the class that drives it. We will then discuss its state machine which gets it to do the things we want.

The Squad Entity Class

The squad entity class derives directly from our entity class from the flocking demo. It is specialized in a few ways, to give us the ability to recall the waypoints we were traversing, the network we are on, etc. Let us take a look at that class now.

```
class cSquadEntity : public cEntity
{
public:
                                          cSquadEntity
                                          (
                                                cWorld &world,
                                                unsigned type,
                                                float senseRange,
                                                float maxVelocityChange,
                                                float maxSpeed,
                                                float desiredSpeed,
                                                float moveXScalar,
                                                float moveYScalar,
                                                float moveZScalar
                                          );
      virtual                             ~cSquadEntity(void);

      virtual void Iterate(float timeDelta);

      void SetPath(const tPath &aPath) { mPath = aPath; }
      tPath &GetPath(void) { return mPath; }
      const tPath &GetPath(void) const { return mPath; }
      void SetGoal(const D3DXVECTOR3 &goal) { mGoalPosition = goal; }
      D3DXVECTOR3 &GetGoal(void) { return mGoalPosition; }
      const D3DXVECTOR3 &GetGoal(void) const { return mGoalPosition; }
      void SetNextWaypoint(const tWaypointID &wp) { mNextWaypoint = wp; }
      tWaypointID &GetNextWaypoint(void) { return mNextWaypoint; }
      const tWaypointID &GetNextWaypoint(void) const { return mNextWaypoint; }

      cWaypointNetwork *GetWaypointNetwork(void) { return mWaypointNetwork; }
      const cWaypointNetwork *GetWaypointNetwork(void) const
            { return mWaypointNetwork; }
      void SetWaypointNetwork(cWaypointNetwork *network)
            { mWaypointNetwork = network; }

      cStateMachine *GetStateMachine(void) { return mStateMachine; }
      const cStateMachine *GetStateMachine(void) const { return mStateMachine; }
      void SetStateMachine(cStateMachine *machine) { mStateMachine = machine; }

      COLORREF GetColor(void) { return mColor; }
      void SetColor(COLORREF color) { mColor = color; }

      bool WaypointReached();
      void OnWaypointReached();
      void ResetTimeSinceWaypointReached()
            { mTimeSinceNextWaypointReached = 0.0f; }
      void IncrementTimeSinceWaypointReached(float deltaTime)
            { mTimeSinceNextWaypointReached += deltaTime; }
      float GetTimeSinceWaypointReached() const
            { return mTimeSinceNextWaypointReached; }

      bool GoalReached();
      void OnGoalReached();

      void OnWaitingForCommand();
```

```
      bool HasValidWaypoint() { return !mNextWaypoint.IsEqual(GUID_NULL); }
      bool HasValidPath() { return mPath.size() > 0; }

      cWorld &World() { return mWorld; }

protected:
      cWaypointNetwork        *mWaypointNetwork;
      tWaypointID                   mNextWaypoint;
      D3DXVECTOR3                   mGoalPosition;
      tPath                         mPath;
      COLORREF                      mColor;
      float                         mTimeSinceNextWaypointReached;
      cStateMachine                 *mStateMachine;
};
```

Most of this interface is accessors, so it is larger and more complicated looking than it really is. We will start our examination with the data members, and then some of the less obvious methods.

```
      cWaypointNetwork        *mWaypointNetwork;
```

First off, we have the waypoint network that this entity is traveling on.

```
      tWaypointID                   mNextWaypoint;
      D3DXVECTOR3                   mGoalPosition;
```

Next, we have the waypoint ID of the next waypoint we are traveling to, and our ultimate goal position.

```
      tPath                         mPath;
```

Here we have the path we will be using to find our way through the network to our ultimate goal.

```
      COLORREF                      mColor;
```

Here we have the color we are going to draw this entity in the waypoint network view. This gets changed when we walk over waypoints.

```
      float                         mTimeSinceNextWaypointReached;
```

Here we have the timer that keeps track of how long it has been since this entity has reached a waypoint. If this timer value gets to be too large, the entity has his velocity agitated in order to get him to his waypoint.

```
      cStateMachine                 *mStateMachine;
```

Last, we have the state machine for this entity.

Now that we have seen the data this class uses, let us look at the non-trivial methods.

```
void cSquadEntity::Iterate(float timeDelta)
{
	// iterate our state machine
	if (mStateMachine)
		mStateMachine->Iterate();

	cEntity::Iterate(timeDelta);

	// Update our orientation
	const float kEpsilon = 0.01f;
	if (D3DXVec3Length(&mVelocity) > kEpsilon)
	{
		D3DXVECTOR3 vec;
		D3DXVec3Normalize(&vec, &mVelocity);
		float zRotation = atan2f(vec.y, vec.x);
		D3DXQuaternionRotationYawPitchRoll(&mOrientation, 0.0f,
						0.0f, zRotation);
	}
}
```

Here we have the iterate method, which is called every frame. Its job is to execute the movement behaviors on the entity, and move the entity through the world. We do two things above and beyond the default implementation. We first update our state machine for this frame, then we call the default Iterate implementation, and finally we compute a new orientation vector which is compatible with this demo's coordinate system.

```
#define ENTITY_RADIUS 0.75f
bool cSquadEntity::WaypointReached()
{
	if (!mNextWaypoint.IsEqual(GUID_NULL) && mWaypointNetwork)
	{
		cWaypoint *wp = mWaypointNetwork->FindWaypoint(mNextWaypoint);
		if (wp)
		{
			D3DXVECTOR3 vec = wp->GetPosition() - Position();
			float distToWP = D3DXVec3Length(&vec);
			if ((distToWP - ENTITY_RADIUS) < wp->GetRadius())
			{
				return true;
			}

			return false;
		}
	}

	return false;
}
```

The WaypointReached method is called to check to see if the next waypoint has been reached. If we have no next waypoint, or no waypoint network, we return false. Otherwise, we find the next waypoint in the network, and compute our distance to the waypoint. In this case, we take into account the radius of the entity to determine if we have reached the waypoint.

```
#define GOAL_REACH_THRESHOLD 1.5f

bool cSquadEntity::GoalReached()
{
	D3DXVECTOR3 vec = mGoalPosition - Position();
	if (D3DXVec3Length(&vec) < GOAL_REACH_THRESHOLD)
	{
		return true;
	}

	return false;
}
```

The GoalReached method works similarly to the WaypointReached method, only it computes the distance to the goal position rather than to the next waypoint.

```
void cSquadEntity::OnWaypointReached()
{
	if (!mNextWaypoint.IsEqual(GUID_NULL) && mWaypointNetwork)
	{
		cWaypoint *wp = mWaypointNetwork->FindWaypoint(mNextWaypoint);
		if (wp)
		{
			COLORREF color;
			wp->GetBlindData(0, color);
			SetColor(color);
			return;
		}
	}

	SetColor(RGB(0, 0, 0));
}
```

The OnWaypointReached method gets called when the squad member has reached his next waypoint. It queries the blind data for the color value stored in it, and sets the color of the entity using that value.

```
void cSquadEntity::OnGoalReached()
{
	SetColor(RGB(255, 150, 150));
}
```

The OnGoalReached method gets called when the squad member has reached his goal. It simply changes the color of the entity.

```
void cSquadEntity::OnWaitingForCommand()
{
	SetColor(RGB(150, 255, 150));
}
```

The OnWaitingForCommand method gets called when the squad member is waiting for a command. It simply changes the color of the entity.

The Squad Member State Machine

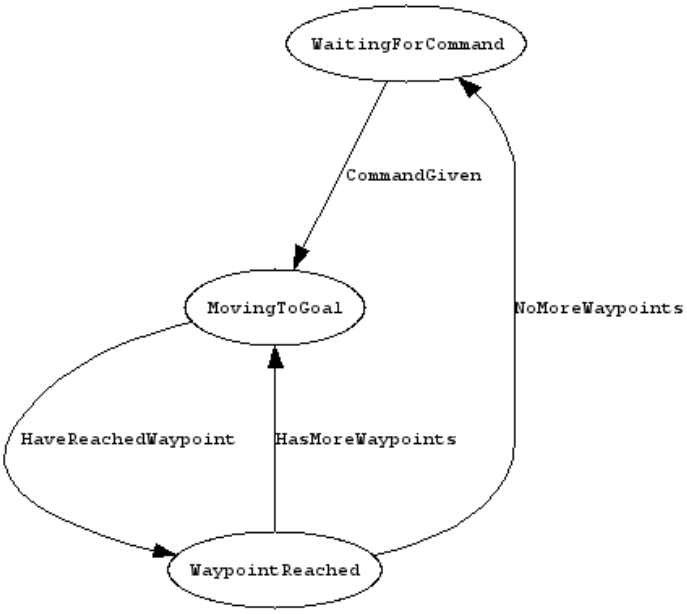

Figure 5.5

As the state transition diagram shows (see Figure 5.5), the squad member has a simple state machine. He starts waiting for a command, and once he gets one, he moves to the goal using the network. Each time he reaches a waypoint (or the goal), he goes to the waypoint reached state. Once he has reached the goal point, and he has no more waypoints, he goes back to waiting for a command.

Let us take a look at the Python scripted actions and transitions that get this job done for us.

```
from GI_AISDK import *

class WaitingForCommandAction(PythonScriptedAction):

      def execute(self):

            self.state().entity().on_waiting_for_command();
```

First we have the waiting for command action. It simply calls the entity's on_waiting_for_command handler. This action is executed when the waiting for command state is entered.

```
from GI_AISDK import *

class CommandGiven(PythonScriptedTransition):

      def should_transition(self):

            return self.source().entity().has_valid_path() and \
                   not self.source().entity().has_valid_waypoint();
```

The CommandGiven transition checks to see if the entity has a valid path and not a valid waypoint. This means a new target has been set, but the entity has not yet begun walking that path.

```
from GI_AISDK import *

class HaveReachedWaypoint(PythonScriptedTransition):

      def should_transition(self):

            return self.source().entity().waypoint_reached() or \
                  self.source().entity().goal_reached();
```

The HaveReachedWaypoint transition checks to see if the entity has either reached a waypoint, or the goal.

```
from GI_AISDK import *

class WaypointReachedAction(PythonScriptedAction):

      def execute(self):

            if self.state().entity().waypoint_reached():
                  self.state().entity().on_waypoint_reached();
            else:
                  self.state().entity().on_goal_reached();
```

The WaypointReachedAction checks to see if the waypoint or the goal has been reached, and calls the appropriate handler. This action is performed when the waypoint reached state is entered.

```
from GI_AISDK import *

class HasMoreWaypoints(PythonScriptedTransition):

      def should_transition(self):

            return self.source().entity().has_valid_path() or not \
                  self.source().entity().goal_reached();
```

The HasMoreWaypoints transition simply evaluates if the entity has a valid path, or has not yet reached the goal.

```
from GI AISDK import *

class NoMoreWaypoints(PythonScriptedTransition):

      def should_transition(self):

            return self.source().entity().goal_reached() and not \
                  self.source().entity().has_valid_path()
```

The NoMoreWaypoints transition simply returns if the goal has been reached and the entity does not have a valid path.

That does it for the squad members. Now we just have to investigate how the squad leader works and we will be ready to set up this demo.

5.4.3 The Squad Leader

The squad leader derives from our squad entity class. This allows him to navigate the world if he so desires. He also has some specialized functionality, so let us take a look at that.

The Squad Leader Class

```
class cSquadLeaderEntity : public cSquadEntity
{
public:
                                        cSquadLeaderEntity
                                        (
                                              cWorld &world,
                                              unsigned type,
                                              float senseRange,
                                              float maxVelocityChange,
                                              float maxSpeed,
                                              float desiredSpeed,
                                              float moveXScalar,
                                              float moveYScalar,
                                              float moveZScalar
                                        );
      virtual                           ~cSquadLeaderEntity(void);

      void SendSquadToRandomPOI();
      bool SquadArrivedAtGoal();

      cPointOfInterest *GetSelectedPointOfInterest() const
            { return mSelectedPointOfInterest; }
      void SetSelectedPointOfInterest(cPointOfInterest *poi)
            { mSelectedPointOfInterest = poi; }

      tPointOfInterestMap     *GetPointsOfInterest()
            { return mPointsOfInterest; }
      void SetPointsOfInterest(tPointOfInterestMap *pointsOfInterest)
            { mPointsOfInterest = pointsOfInterest; }

      void AddSquadMember(cSquadEntity *member);
      void RemoveSquadMember(cSquadEntity *member);
      void ClearSquadMembers();

protected:
      vector<cSquadEntity*>   mSquadMembers;
```

```
      tPointOfInterestMap     *mPointsOfInterest;
      cPointOfInterest        *mSelectedPointOfInterest;
};
```

One of the first things you will notice is the point of interest class. A point of interest is a lot like a waypoint, only it is not connected to the network. Typically, you create a point of interest for anything in the game world that is going to be important to the decision maker.

Consider the example of a stealth action game, where the hero is trying to sneak into a compound filled with bad guys. If the hero makes a noise, the game could create a "noise" point of interest at the position where he made the noise. The bad guys would then see the point of interest, and go investigate.

In the demo, the squad leader has a large set of points of interest, and he directs his squad to investigate them at random. Again, the points of interest are *not* on the network. This demonstrates that most of the time in a continuous world environment, the things you care about will not be on the pre-computed network (although certainly, you can include points of interest on the waypoint network if desired). Ultimately, we need to be able to get on and off the network without mishap.

Getting back to the squad leader class, you will notice that there are a few methods and some data that need a little explaining.

```
      vector<cSquadEntity*>   mSquadMembers;
```

First we have the vector of squad members in our squad. This allows the leader to keep track of them, and tell them what to do.

```
      tPointOfInterestMap     *mPointsOfInterest;
```

Next we have a map of point of interest IDs to points of interest. This is basically just like the waypoint map the waypoint network has, only with points of interest instead.

```
      cPointOfInterest        *mSelectedPointOfInterest;
```

Last, we have the point of interest we are currently directing our squad to investigate.

There are two methods of interest in the squad leader class. Let us take a look at them.

```
void cSquadLeaderEntity::SendSquadToRandomPOI()
{
      if (!mPointsOfInterest && mSquadMembers.size() > 0)
            return;

      tPointOfInterestID closestPOI = FindPointOfInterestNearestPosition
                                      (
                                           *mPointsOfInterest,
                                           mSquadMembers[0]->GetGoal()
                                      );
      tPointOfInterestID poiID = SelectRandomPointOfInterest(*mPointsOfInterest,
```

```
                                                                  closestPOI);
      cPointOfInterest *poi = FindPointOfInterest(*mPointsOfInterest, poiID);
      if (!poi)
            return;

      SetSelectedPointOfInterest(poi);

      tPath pathToWP;
      cWaypointVisibilityFunctor functor;
      for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
            it != mSquadMembers.end(); ++it)
      {
            cSquadEntity *entity = *it;
            entity->SetNextWaypoint(GUID_NULL);
            pathToWP.clear();

            GetWaypointNetwork()->FindPathFromPositionToPosition
                                  (
                                        entity->Position(),
                                        poi->GetPosition(),
                                        functor,
                                        pathToWP
                                  );
            entity->SetPath(pathToWP);
            entity->SetGoal(poi->GetPosition());
      }
}
```

First we have the SendSquadToRandomPOI method. This method selects a random point of interest, and sends every squad member to investigate. Let us take a closer look.

```
      if (!mPointsOfInterest && mSquadMembers.size() > 0)
            return;
```

For starters, if we have no points of interest, or no squad members, we do not have any work to do.

```
      tPointOfInterestID closestPOI = FindPointOfInterestNearestPosition
                                      (
                                          *mPointsOfInterest,
                                          mSquadMembers[0]->GetGoal()
                                      );
```

Next we find the closest POI to the squad's current goal position. The idea here is that we do not want to select this point of interest again, since they are already going there, or are already there.

```
      tPointOfInterestID poiID = SelectRandomPointOfInterest(*mPointsOfInterest,
                                                             closestPOI);
```

Now we select a random point of interest, ignoring the one we just found as the closest. This method is pretty simple, so we will not go into it here (see PointOfInterest.cpp).

```
      cPointOfInterest *poi = FindPointOfInterest(*mPointsOfInterest, poiID);
```

```
	if (!poi)
		return;
```

Next, we find the point of interest in the map using the ID we got back from the random selector. If we do not find it, we bail out.

```
	SetSelectedPointOfInterest(poi);
```

We then select this as our current point of interest.

```
	tPath pathToWP;
	cWaypointVisibilityFunctor functor;
	for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
		it != mSquadMembers.end(); ++it)
	{
```

Next, we iterate through all of our squad members. We will be finding a path for each of them, so we will need a path, and a visibility functor.

```
		cSquadEntity *entity = *it;
		entity->SetNextWaypoint(GUID_NULL);
		pathToWP.clear();
```

For each entity, we will null out their current waypoint, and clear our path that we are going to find for them.

```
		GetWaypointNetwork()->FindPathFromPositionToPosition
				(
					entity->Position(),
					poi->GetPosition(),
					functor,
					pathToWP
				);
		entity->SetPath(pathToWP);
		entity->SetGoal(poi->GetPosition());
```

We then find the path from the entity's current position to the position of the point of interest we selected. We set that path on the entity, and set his goal to the position of the point of interest we selected. It is worth noting here, that we could easily modify this method to select a different point of interest for each squad member by moving the random waypoint selection code inside the loop.

```
bool cSquadLeaderEntity::SquadArrivedAtGoal()
{
	for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
		it != mSquadMembers.end(); ++it)
	{
		cSquadEntity *entity = *it;
		if (!entity->GoalReached())
			return false;
	}
```

```
      return true;
}
```

The other method we should talk about is the SquadArrivedAtGoal method. This method returns success if all of the squad members have arrived at their goals. It is done by simply iterating through all the squad members, and if any one of them has not reached their goal, we return false. Otherwise, we return true.

That is everything we need to discuss about the squad leader class, so let us take a look at the state machine the squad leader uses.

The Squad Leader State Machine

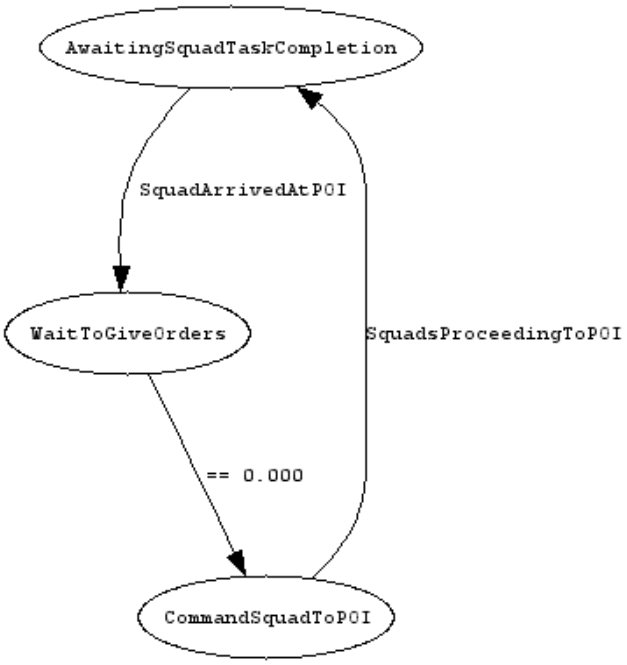

Figure 5.6

The squad leader has three states: the awaiting squad task completion state, the wait to give orders state, and command squad to POI state. He begins in the command squad to POI state, and directs the squad to a point of interest. He then waits for the squad to arrive at their destination, whereupon he waits a second, then commands them to another point of interest. Let us take a look at the Python scripted actions and transitions that make this work.

```
from GI_AISDK import *

class SquadArrivedAtPOI(PythonScriptedTransition):
```

```
      def should_transition(self):
            return self.source().entity().squad_arrived_at_goal();
```

The SquadArrivedAtPOI transition simply returns if the squad has arrived at the goal.

```
from GI_AISDK import *

class CommandSquadMembersToPOI(PythonScriptedAction):

      def execute(self):
            self.state().entity().send_squad_to_random_poi();
```

The CommandSquadMembersToPOI action sends the squad to a random point of interest using the method provided by the squad leader. This action is performed when the state is entered.

```
from GI_AISDK import *

class SquadsProceedingToPOI(PythonScriptedTransition):

      def should_transition(self):
            return True
```

The SquadsProceedingToPOI transition simply returns true. Since the squad leader issues the orders for the squad to proceed to their goal upon entering the CommandSquadToPOI state, there is no need to remain in the state afterwards.

That is really all there is to the squad leader. Obviously there is much more behavior you are probably thinking about adding, and with the tools provided you will be able to develop some very cool and interesting AI. To wrap things up in this chapter, let us see how we set up our demo to bring it all together.

5.5 Setting up the Demo

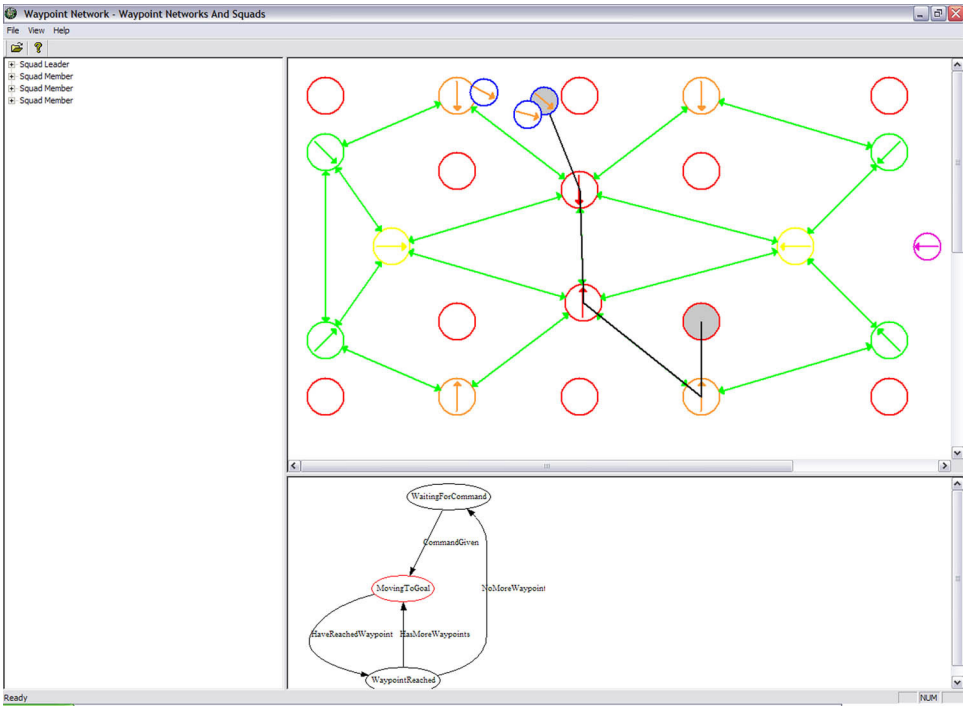

Figure 5.7

The waypoint network and squads demo is shown in Figure 5.7. On the left, we have a tree view that represents the state machines of the squad leader and his squad members. This tree view is just like the one in the state machine demo, but does not allow editing of the state machines. If you want, you can use the state machine demo to edit your state machines, since this demo uses those files to load the squad leader and squad member state machines.

The top pane on the right is the waypoint network view. It displays the waypoints, their edges, and the points of interest, along with the squad members. The arrows inside the waypoints show the orientation of the waypoint. While this demo makes no use of that information, as mentioned before, orientation on waypoints can be very useful, so it deserves attention. The circles without the arrows are the points of interest. The purple circle with the arrow is the squad leader, and the blue circles with the (in the case of this diagram orange) arrows are the squad members. As the squad members cross the waypoints, their arrow will change to the color of the waypoint crossed. The currently selected entity will be filled in gray, and his path to the currently selected point of interest (also filled in gray) will be shown in black. You may select squad members by clicking on them in this view, or in the state machine tree on the left.

The pane on the bottom right is the state machine view. It is similar to the state machine view in the state machine demo, as it shows the currently active squad member's state machine. The state with the red border is the state the squad member is currently residing in.

Let us quickly go over the initialization code for the demo, and then you should be ready to jump in and have some fun with it!

```
BOOL CWaypointNetworksAndSquadsDoc::OnNewDocument()
{
      if (!CDocument::OnNewDocument())
            return FALSE;

      // free the old network and world
      if (mWaypointNetwork)
            delete mWaypointNetwork;
      mWaypointNetwork = NULL;

      if (mWorld)
            delete mWorld;
      mWorld = NULL;

      if (mPFbeh)
            delete mPFbeh;
      mPFbeh = NULL;

      ClearPointsOfInterest(mPointsOfInterest);

      // allocate a new ones
      mWaypointNetwork = new cWaypointNetwork();
      mWorld = new cWorld;
      cGroup *squadGroup = new cGroup(*mWorld);
      mWorld->Add(*squadGroup);

      const int kSquadType = 0x1;
      const float kSenseDistance = 20.0f;
      const float kMaxVelocityChange = 1.0f;
      const float kMaxSpeed = 5.0f;
      const float kMoveXScalar = 1.0f;
      const float kMoveYScalar = 1.0f;
      const float kMoveZScalar = 0.0f;    // don't allow z moment
      const float kSeparationDist = 1.5f;
      const float kPathfindingMaxRateChange = 0.2f;
      const float kGoalReachedRadius = 1.0f;
      const float kMaxTimeBeforeAgitation = 5.0f; // seconds to reach a
                                                  // path node before the entity
                                                  // gets agitation stimulus
      D3DXVECTOR3 upVector(0.0f, 0.0f, 1.0f);   // The entities in this demo
                                                // live in the XY plane
      const int kNumSquadMembers = 3;

      // make pathfinding behavior
      mPFbeh = new cPathfindBehavior(kPathfindingMaxRateChange, kGoalReachedRadius,
                                     kSeparationDist, kMaxTimeBeforeAgitation,
                                     upVector, *mWaypointNetwork);

      // load a default waypoint network
      ifstream wpnfile("demo.wpn");

      mWaypointNetwork->UnSerialize(wpnfile);
      ClearPointsOfInterest(mPointsOfInterest);

      UnSerializePointsOfInterest(mPointsOfInterest, wpnfile);
```

```
	tPointOfInterestID startPoiID = SelectRandomPointOfInterest(
	                                    mPointsOfInterest, GUID_NULL);
	cPointOfInterest *startPoi = FindPointOfInterest(mPointsOfInterest,
	                                                 startPoiID);
	D3DXVECTOR3 origin = startPoi->GetPosition();
	tPointOfInterestID endPoiID = SelectRandomPointOfInterest(mPointsOfInterest,
	                                                          startPoiID);
	cPointOfInterest *endPoi = FindPointOfInterest(mPointsOfInterest, endPoiID);
	D3DXVECTOR3 destination = endPoi->GetPosition();

	// setup the squad leader
	mSquadLeader = new cSquadLeaderEntity
	                                (
	                                      *mWorld,
	                                      kSquadType,
	                                      kSenseDistance,
	                                      kMaxVelocityChange,
	                                      kMaxSpeed,
	                                      kMaxSpeed * 0.4f,
	                                      kMoveXScalar,
	                                      kMoveYScalar,
	                                      kMoveZScalar
	                                );
	mSquadLeader->SetPointsOfInterest(&mPointsOfInterest);
	D3DXQUATERNION squadLeaderOrientation(0.0f, 0.0f, 1.0f, 0.0f);
	mSquadLeader->SetOrientation(squadLeaderOrientation);
	mSquadLeader->SetSelectedPointOfInterest(endPoi);
	cStateMachine *statemachine = new cSquadStateMachine(mSquadLeader);
	ifstream squadleaderfsm("SquadLeader.stm");
	try
	{
	      if (statemachine->UnSerialize(squadleaderfsm) == FALSE)
	      {
	            // error handling mojo...
	            delete statemachine;
	            statemachine = NULL;
	      }
	}
	catch(error_already_set)
	{
	      // error handling mojo...
	      delete statemachine;
	      statemachine = NULL;
	}
	statemachine->Reset();
	D3DXVECTOR3 squadleaderpos(34, 10, 0.0f);
	mSquadLeader->SetPosition(squadleaderpos);
	mSquadLeader->SetStateMachine(statemachine);
	mSquadLeader->SetWaypointNetwork(mWaypointNetwork);
	squadGroup->Add(*mSquadLeader);

	// setup some squad mates to roam the network
	for (int i = 0; i < kNumSquadMembers; ++i)
	{
	      D3DXVECTOR3 pos(0.0f, (float)i, 0.0f);
	      cSquadEntity *squadmate =     new cSquadMemberEntity
```

```
                                                      (
                                                            *mWorld,
                                                            kSquadType,
                                                            kSenseDistance,
                                                            kMaxVelocityChange,
                                                            kMaxSpeed,
                                                            kMaxSpeed * 0.4f,
                                                            kMoveXScalar,
                                                            kMoveYScalar,
                                                            kMoveZScalar
                                                      );
            statemachine = new cSquadStateMachine(squadmate);
            ifstream ar("SquadMember.stm");
            try
            {
                  if (statemachine->UnSerialize(ar) == FALSE)
                  {
                        // error handling mojo...
                        delete statemachine;
                        statemachine = NULL;
                  }
            }
            catch(error_already_set)
            {
                  // error handling mojo...
                  delete statemachine;
                  statemachine = NULL;
            }
            squadmate->AddBehavior(*mPFbeh);
            squadmate->SetPosition(startPoi->GetPosition() + pos);
            squadmate->SetWaypointNetwork(mWaypointNetwork);
            statemachine->Reset();
            squadmate->SetStateMachine(statemachine);
            squadGroup->Add(*squadmate);
            mSquadLeader->AddSquadMember(squadmate);

            if (!mSelectedEntity)
                  mSelectedEntity = squadmate;

            cWaypointVisibilityFunctor visibilityFunctor;
            tPath thePath;
            if (mWaypointNetwork->FindPathFromPositionToPosition(origin,
                                                                 destination,
                                                                 visibilityFunctor,
                                                                 thePath))
            {
                  squadmate->SetPath(thePath);
                  squadmate->SetGoal(endPoi->GetPosition());
            }
      }

      mPause = false;

      return TRUE;
}
```

As with most of our prior demos, this demo is implemented in MFC, so all of the data lives in the document class. When a new document is created, this method is called. Let us go over it to see how it is put together.

```
	// free the old network and world
	if (mWaypointNetwork)
		delete mWaypointNetwork;
	mWaypointNetwork = NULL;

	if (mWorld)
		delete mWorld;
	mWorld = NULL;

	if (mPFbeh)
		delete mPFbeh;
	mPFbeh = NULL;

	ClearPointsOfInterest(mPointsOfInterest);
```

First, if we have a waypoint or world or pathfinding behavior or any points of interest around, we free them so we can start with a clean slate.

```
	// allocate a new ones
	mWaypointNetwork = new cWaypointNetwork();
	mWorld = new cWorld;
	cGroup *squadGroup = new cGroup(*mWorld);
	mWorld->Add(*squadGroup);
```

We then allocate a new waypoint network, a new world, create a group for our squad, and add it to the world.

```
	const int kSquadType = 0x1;
	const float kSenseDistance = 20.0f;
	const float kMaxVelocityChange = 1.0f;
	const float kMaxSpeed = 5.0f;
	const float kMoveXScalar = 1.0f;
	const float kMoveYScalar = 1.0f;
	const float kMoveZScalar = 0.0f;    // don't allow z moment
	const float kSeparationDist = 1.5f;
	const float kPathfindingMaxRateChange = 0.2f;
	const float kGoalReachedRadius = 1.0f;
	const float kMaxTimeBeforeAgitation = 5.0f; // seconds to reach a
                                                // path node before the entity
                                                // gets agitation stimulus
	D3DXVECTOR3 upVector(0.0f, 0.0f, 1.0f);   // The entities in this demo
                                               // live in the XY plane
	const int kNumSquadMembers = 3;
```

We set up some constants for use in creating the behaviors and entities in the world.

```
	// make pathfinding behavior
	mPFbeh = new cPathfindBehavior(kPathfindingMaxRateChange, kGoalReachedRadius,
```

```
                                        kSeparationDist, kMaxTimeBeforeAgitation,
                                        upVector, *mWaypointNetwork);
```

We then create a pathfinding behavior. This behavior is shared by all of the entities.

```
	// load a default waypoint network
	ifstream wpnfile("demo.wpn");

	mWaypointNetwork->UnSerialize(wpnfile);
	ClearPointsOfInterest(mPointsOfInterest);

	UnSerializePointsOfInterest(mPointsOfInterest, wpnfile);
```

Next we load in the default waypoint network file (feel free to open that file up in a text editor and try modifying it). We also load in the default points of interest from that file.

```
	tPointOfInterestID startPoiID = SelectRandomPointOfInterest(
	                                    mPointsOfInterest, GUID_NULL);
	cPointOfInterest *startPoi = FindPointOfInterest(mPointsOfInterest,
	                                                 startPoiID);
	D3DXVECTOR3 origin = startPoi->GetPosition();
	tPointOfInterestID endPoiID = SelectRandomPointOfInterest(mPointsOfInterest,
	                                                          startPoiID);
	cPointOfInterest *endPoi = FindPointOfInterest(mPointsOfInterest, endPoiID);
	D3DXVECTOR3 destination = endPoi->GetPosition();
```

We then select a random starting point of interest and a random goal point of interest to start off with.

```
	// setup the squad leader
	mSquadLeader = new cSquadLeaderEntity
	                                (
	                                      *mWorld,
	                                      kSquadType,
	                                      kSenseDistance,
	                                      kMaxVelocityChange,
	                                      kMaxSpeed,
	                                      kMaxSpeed * 0.4f,
	                                      kMoveXScalar,
	                                      kMoveYScalar,
	                                      kMoveZScalar
	                                );
	mSquadLeader->SetPointsOfInterest(&mPointsOfInterest);
	D3DXQUATERNION squadLeaderOrientation(0.0f, 0.0f, 1.0f, 0.0f);
	mSquadLeader->SetOrientation(squadLeaderOrientation);
	mSquadLeader->SetSelectedPointOfInterest(endPoi);
	cStateMachine *statemachine = new cSquadStateMachine(mSquadLeader);
	ifstream squadleaderfsm("SquadLeader.stm");
	try
	{
	      if (statemachine->UnSerialize(squadleaderfsm) == FALSE)
	      {
	            // error handling mojo...
	            delete statemachine;
```

```
                statemachine = NULL;
        }
    }
    catch(error_already_set)
    {
        // error handling mojo...
        delete statemachine;
        statemachine = NULL;
    }
    statemachine->Reset();
    D3DXVECTOR3 squadleaderpos(34, 10, 0.0f);
    mSquadLeader->SetPosition(squadleaderpos);
    mSquadLeader->SetStateMachine(statemachine);
    mSquadLeader->SetWaypointNetwork(mWaypointNetwork);
    squadGroup->Add(*mSquadLeader);
```

Now we set up the squad leader. We set his points of interest, give him an initial position and orientation, and set his initial selected point of interest. We also load his state machine from a file (again, feel free to experiment with this machine using the State Machine demo from the last chapter) and set it. Finally, we give him the waypoint network and finally add him to the squad.

```
    // setup some squad mates to roam the network
    for (int i = 0; i < kNumSquadMembers; ++i)
```

Here we create a set of squad members.

```
        D3DXVECTOR3 pos(0.0f, (float)i, 0.0f);
        cSquadEntity *squadmate =     new cSquadMemberEntity
                                                (
                                                    *mWorld,
                                                    kSquadType,
                                                    kSenseDistance,
                                                    kMaxVelocityChange,
                                                    kMaxSpeed,
                                                    kMaxSpeed * 0.4f,
                                                    kMoveXScalar,
                                                    kMoveYScalar,
                                                    kMoveZScalar
                                                );
        statemachine = new cSquadStateMachine(squadmate);
        ifstream ar("SquadMember.stm");
        try
        {
            if (statemachine->UnSerialize(ar) == FALSE)
            {
                // error handling mojo...
                delete statemachine;
                statemachine = NULL;
            }
        }
        catch(error_already_set)
        {
            // error handling mojo...
            delete statemachine;
```

```
            statemachine = NULL;
      }
      squadmate->AddBehavior(*mPFbeh);
      squadmate->SetPosition(startPoi->GetPosition() + pos);
      squadmate->SetWaypointNetwork(mWaypointNetwork);
      statemachine->Reset();
      squadmate->SetStateMachine(statemachine);
      squadGroup->Add(*squadmate);
      mSquadLeader->AddSquadMember(squadmate);

      if (!mSelectedEntity)
            mSelectedEntity = squadmate;

      cWaypointVisibilityFunctor visibilityFunctor;
      tPath thePath;
      if (mWaypointNetwork->FindPathFromPositionToPosition(origin,
                                                           destination,
                                                           visibilityFunctor,
                                                           thePath))
      {
            squadmate->SetPath(thePath);
            squadmate->SetGoal(endPoi->GetPosition());
      }
```

For each squad member, we load his state machine from a file (which can be edited in the State Machine Editor from the previous chapter). We add the pathfinding behavior, initialize his position, set his waypoint network, set his state machine, and add him to the group. We also inform the squad leader about the new squad mate and find a path for him from his current position to the goal position of the initial goal point of interest.

That is it! The demo is now initialized. There is one other place that needs noting -- the UpdateWorld method in the document. This method calls the Iterate method on the world object, which ensures the entities get iterated (making sure that their behaviors and state machines get iterated as well). This method gets called by a timer set in the waypoint network view, once every 33 milliseconds.

Conclusion

In this chapter, we discussed how to bring together the various AI components we learned about in this course to get them to cooperate in a single project. We talked about how we can build a waypoint network for traversing a continuous world, and a movement behavior for traversing the network. We looked at squad entities, and saw how we can use scripted state machines to drive their behavior. We even talked about squad leaders, and some different ways that we can get them to command their squads. Ultimately we built a practical example that demonstrated a squad examining points of interest using a waypoint network, while using scripting to extend our state machines.

With the material discussed in this course, it should be possible to build a robust AI system for your games using the framework provided as a starting point. Hopefully you have found our discussions and demonstrations enjoyable. There is obviously a lot to learn as a game developer, but for many of us, AI

programming is certainly one of the most fun and exciting. This is because you really get a chance to be as creative as you want to be (within reason!) and see the results of your work acted out by the little virtual characters in your world. It is a very satisfying feeling to watch your team of AI entities travel from place to place in the world in a realistic manner, seeming to communicate and cooperate with one another, doing things that make you feel that these guys are really thinking for themselves. You will get the chance soon enough to experience this for yourself.

And thus we have reached the end of our course, but certainly not to the end of the road. In this course we have covered a lot of the core AI topics that you will need to understand if you want to develop AI for games. But as mentioned right at the beginning, there is a lot more to study in the field of AI then what we were able to talk about in our short time together. At this point you have a very solid set of working code that can serve as the basis for further exploration and enhancement. If you enjoyed the materials we encountered and remain interested in taking your game AI even further, there are plenty of books and internet tutorials available for you to examine. And of course, if you do create some interesting AI for your game using the systems we discussed, please come by and let us know. We would love to hear about it and see your AI in action!

We wish you the very best of luck in your future game programming adventures!