Artificial Intelligence for Game Developers

Supplemental Workbook

Introduction

Our final lab project in this course is going to be a 3D extension of the previous project. For the most part, the code is going to be nearly identical, so you should find it relatively easy to follow. The main goal is to demonstrate to you that you can extend the concepts we have been talking about into a 3D world with very few changes to the underlying AI framework we have developed together in this course.

In this demo, our objective will be to get a four man squad to move from point to point in the world using the navigation schemes we have discussed in earlier lessons. The squad leader (in this case, the player) will be able to issue commands to the team and have them attempt to get to different locations in a 3D environment using the waypoint and pathfinding concepts introduced in the Chapter 5 demo.

All of the 3D rendering, collision detection, animation, etc. will be tucked away in a small library developed specifically for this demonstration. The code for this library and the assets for this demo were both adapted from the 3D Graphics Programming Module II course available right here at the Game Institute. The 3D world and NPCs we will use should be instantly recognizable to students who are currently taking that course or who have taken it in the past.

Since the code in the demo is basically going to be the same as the code we covered previously, we will focus mostly on the changes that were made to take our ideas from 2D into 3D.

One thing to note is that this demonstration will not be using MFC. To keep things simple, we are going to be running our code in a standard Windows application. This is more likely the manner in which your own game applications will run, so it is a worthwhile way to construct our final demonstration. With that said, we can begin at the beginning and look at the function that initializes our application and runs the simulation. As you probably guessed, this will be WinMain (see AI Demo.cpp):

WinMain

```
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPTSTR lpCmdLine,  int nCmdShow )
{
      MSG msg;
      int last, current, elapsed;

      // Create our window
      if ( !BuildWindow (hInstance, nCmdShow) )
            return FALSE;

      // Seed random number generator
      srand(timeGetTime());

      // initialize our scripting engine
      cPythonScriptEngine::Instance().Initialize();

      // Create our world
```

```
	if ( CreateWorld( m_hWnd, hInstance, &g_pWorld ) )
	{
		if ( g_pWorld->Load( "Data\\LandscapeII.iwf" ) )
		{
			if (SetupWorld())
			{
				// Initialize starting time
				last = timeGetTime();

				// Start main loop
				while (1)
				{
					// Did we recieve a message, or are we idling ?
					if ( PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) )
					{
						if (msg.message == WM_QUIT) break;
						TranslateMessage( &msg );
						DispatchMessage ( &msg );
					}
					else
					{
						// Get elapsed time
						current = timeGetTime();
						elapsed = current - last;

						// Step the simulations
						StepSimulation( (float)elapsed / 1000.0f );

						// Record current time
						last = current;

					} // End If messages waiting

				} // Until quit message is receieved
			}
		}
	}

	// Release our world
	if( g_pWorld ) g_pWorld->Release();
	g_pWorld = NULL;

	if ( g_pEntityWorld ) delete g_pEntityWorld;
	g_pEntityWorld = NULL;

	// shutdown our scripting engine
	cPythonScriptEngine::Instance().Finalize();

	return 0;
}
```

As you can see, there is nothing terribly complicated happening here. We start off by creating our window and seeding our random number generator.

```
	// Create our window
	if ( !BuildWindow (hInstance, nCmdShow) )
		return FALSE;

	// Seed random number generator
	srand(timeGetTime());
```

Next we initialize our scripting engine and create an instance of our 3D wrapper library's IWorld class interface. We will see how to use the functions exposed by this interface as we go along. Once we have successfully initialized the 3D world interface (g_pWorld), we will load in our environment data. Once again, the wrapper library handles all of this on our behalf, so we have little to do other than specify the IWF file we wish to use. IWF is the standard world file format here at Game Institute and is fully supported by the GILES™ world editor. In fact, this level was actually assembled using GILES™, as were all of the waypoints and points of interest (as we will see later). Once the world is fully loaded (which also handles assembling spatial partitioning trees for collision queries, building the render scene graph, etc.) we call our local SetupWorld function to manage our AI entity loading. We will look at that function in just a few moments. For now, just know that when SetupWorld returns, all of our AI entities and their 3D representations in the wrapper library will be loaded and configured.

```
	// initialize our scripting engine
	cPythonScriptEngine::Instance().Initialize();

	// Create our world
	if ( CreateWorld( m_hWnd, hInstance, &g_pWorld ) )
	{
		// Load stuff here...
		if ( g_pWorld->Load( "Data\\LandscapeII.iwf" ) )
		{
			if (SetupWorld())
			{
```

Now that the world is set up and our AI entities are in place, we are ready to begin running the simulation. The next section of code is a standard Windows message pump and should be familiar to all of you at this point in your training -- we basically loop forever until we receive a quit message from the OS. When there are no messages to process, we calculate the time delta since the last loop iteration and step our simulation forward by another frame. We will examine the code to the StepSimulation call in a moment.

```
				// Initialize starting time
				last = timeGetTime();

				// Start main loop
				while (1)
				{
					// Did we recieve a message, or are we idling ?
					if ( PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) )
					{
						if (msg.message == WM_QUIT) break;
						TranslateMessage( &msg );
						DispatchMessage ( &msg );
					}
```

```
                        else
                        {
                              // Get elapsed time
                              current = timeGetTime();
                              elapsed = current - last;

                              // Step the simulations
                              StepSimulation( (float)elapsed / 1000.0f );

                              // Record current time
                              last = current;

                        } // End If messages waiting

                  } // Until quit message is receieved
            }
      }
}
```

Once the user has decided that they have had enough of our demo, they will close the application. This will generate the quit message that breaks us out of the infinite message loop above. All that is left to do is cleanup any resources we may have allocated and shut the application down.

```
      // Release our world
      if( g_pWorld )
            g_pWorld->Release();
      g_pWorld = NULL;

      if ( g_pEntityWorld )
            delete g_pEntityWorld;
      g_pEntityWorld = NULL;

      // shutdown our scripting engine
      cPythonScriptEngine::Instance().Finalize();

      return 0;
```

Before we look at the code to the StepSimulation function, which is our high level game logic update function, we will finish looking at our initialization code by examining the code to SetupWorld.

SetupWorld

The SetupWorld function is responsible for creating and populating our squads and configuring our squad leader as well. Unlike our last project, in this demo, the squad leader is going to be a dual-nature entity. It will still be an entity that is assigned to the squad as before, and it will include some commands that control a number of behaviors revolving around commanding the squad, but it will also allow for total player control when desired. The player is basically going to take on the role of squad leader in this demonstration and they will assume full control of the squad and issue orders directly. They can also

maneuver the squad leader to various locations in the world using keyboard and mouse input. We will learn more about how this will all work later when we begin looking at the actual AI code.

Let us now walk through the key sections of the setup code one at a time and discuss what is happening. (You should be following along with the source code to the project opened up at this point if you are not already.)

The first thing we will do in this function is create a new instance of our own cWorld class. This class is responsible for storage and management of our squads and their respective members, as well as a number of other simple housekeeping tasks in our AI system. Once done, we grab a copy of our waypoint network and our points of interest. Both of these are stored in the IWF file that was loaded earlier and are nicely packaged up for us by the wrapper library which handled the file load. Finally, since the player is going to act as our squad leader, we want to make sure that we know where the library initially positioned him in the world. We can always change this later, but this is as good a way as any to get started as it will help us place the rest of our team in the 3D world later on in this function.

```
BOOL SetupWorld( )
{
      // Create the entity world
      g_pEntityWorld = new cWorld;

      // Get our waypoint network
      g_pWaypointNetwork = const_cast<cWaypointNetwork*>
                                          (g_pWorld->GetWaypointNetwork());
      if (!g_pWaypointNetwork)
            return FALSE;

      // Get our points of interest
      ClearPointsOfInterest(g_PointsOfInterest);
      POI_Array pois = g_pWorld->GetPointsOfInterest();
      for (POI_Array::iterator it = pois.begin(); it != pois.end(); ++it)
            AddPointOfInterest(g_PointsOfInterest, *(*it));

      // get the squad leader data
      D3DXVECTOR3 start = GetWorld( )->GetPlayerPosition( );
      D3DXVECTOR3 dir   = GetWorld( )->GetPlayerRight( );
```

Now it is time to start assembling our squad. We begin by allocating a new squad (a cGroup) and adding it to our world.

```
      // Create a new squad (group)
      cGroup *squadGroup = new cGroup(*g_pEntityWorld);
      g_pEntityWorld->Add(*squadGroup);
```

The next step is making sure that we have our pathfinding behavior available since our squad members are going to need it to get around the 3D world using the waypoint network we just loaded in. We have also added a new behavior type for this demo called a formation behavior. We will discuss this new behavior a bit later when we examine our squad entity updates. The constants that we pass into the behavior constructors can be seen in the full version of the source code. Feel free to tweak these settings to suit your own application.

```
	// Make pathfinding behavior
	cBehavior *beh = new cPathfindBehavior(kPathfindingMaxRateChange,
	                                       kGoalReachedRadius,
	                                       kSeparationDist,
	                                       kMaxTimeBeforeAgitation,
	                                       kMaxTimeBeforeLineOfSight,
	                                       upVector,
	                                      *g_pWaypointNetwork);

	// Make formation behavior
	cBehavior *form = new cFormationBehavior(kGoalReachedRadius,
	                                         kFormationRadius);
```

Now we can create our squad leader. As in the last demo, this will involve giving the squad leader a copy of the waypoint network and points of interest, loading and setting up his state machine, assigning him a position in the world, and adding him to the squad. This code is basically the same as the code we saw in the 2D case except the squad leader will now take on the 3D world position of the player.

```
	g_pSquadLeader = new cSquadLeaderEntity
	                                    (
	                                          *g_pEntityWorld,
	                                          kSquadType,
	                                          kSenseDistance,
	                                          kMaxVelocityChange,
	                                          kMaxSpeed,
	                                          kMaxSpeed * 0.4f,
	                                          kMoveXScalar,
	                                          kMoveYScalar,
	                                          kMoveZScalar
	                                    );

	// Give squad leader the waypoint network/POI list
	g_pSquadLeader->SetWaypointNetwork(g_pWaypointNetwork);
	g_pSquadLeader->SetPointsOfInterest(&g_PointsOfInterest);

	// Load squad leader's state machine
	cStateMachine *statemachine = new cSquadStateMachine(g_pSquadLeader);
	ifstream squadleaderfsm("Data\\SquadLeader.stm");
	try
	{
	      if (statemachine->UnSerialize(squadleaderfsm) == FALSE)
	      {
	            ::MessageBox(NULL, "Unable to Load SquadLeader state machine.",
	                         "Bummer.", MB_ICONEXCLAMATION);
	            delete statemachine;
	            statemachine = NULL;
	            return FALSE;
	      }
	}
	catch(error_already_set)
	{
	      delete statemachine;
	      statemachine = NULL;
	      return FALSE;
	}
```

```
	statemachine->Reset();
	g_pSquadLeader->SetStateMachine(statemachine);

	// Set the leader's starting position and waypoint network
	g_pSquadLeader->SetPosition(start);

	// Assign leader a squad ID of 0
	g_pSquadLeader->SetSquadID( 0 );

	// Add the leader to the current squad
	squadGroup->Add(*g_pSquadLeader);
```

The only bit of code above that would look unfamiliar is the call to SetSquadID. This is a simple function that was added in this demo that just assigns an integer identifier to the squad member (there is a new member variable in our squad entity class called mSquadID that this maps to). Later on, this ID will allow us to identify the individual squad members so that we can assign them certain types of behaviors according to who they are.

Now that we have our squad leader fully configured, we can begin adding new squad members for him to command. In this demo, in addition to the squad leader, we will have four squad members, so we set up a loop to iterate four times and create and initialize our remaining squadmates. The code for each new entity is basically the same as the code we saw above for the squad leader.

```
	D3DXVECTOR3 position( 8190.0f, 200.0f, 8160.0f );
	D3DXVECTOR3 rotation( 0.0f, 0.0f, 0.0f );
	int nNPCCount = 4;

	for (int j = 0; j < nNPCCount; j++)
	{
		// Create a new squad member
		cSquadEntity *squadmate = new cSquadMemberEntity
									(
										*g_pEntityWorld,
										kSquadType,
										kSenseDistance,
										kMaxVelocityChange,
										kMaxSpeed,
										kMaxSpeed * 0.4f,
										kMoveXScalar,
										kMoveYScalar,
										kMoveZScalar
									);

		// Create a new state machine for this squad member
		statemachine = new cSquadStateMachine(squadmate);
		ifstream ar("Data\\SquadMember.stm");
		try
		{
			if (statemachine->UnSerialize(ar) == FALSE)
			{
				::MessageBox(NULL,
							"Unable to Load SquadMember state machine.",
							"Bummer.", MB_ICONEXCLAMATION);
```

```
                        delete statemachine;
                        statemachine = NULL;
                        return FALSE;
                  }
            }
            catch(error_already_set)
            {
                  delete statemachine;
                  statemachine = NULL;
                  return FALSE;
            }
            statemachine->Reset();
            squadmate->SetStateMachine(statemachine);

            // Set the behaviors and waypoint network
            squadmate->AddBehavior(*form);
            squadmate->AddBehavior(*beh);
            squadmate->SetWaypointNetwork(g_pWaypointNetwork);

            // Calculate starting position
            position = start + dir * ( (j + 1) * 165.0f );
            squadmate->SetPosition(position);

            // Assign squadmate an ID
            squadmate->SetSquadID( 1 + j );

            // Add squad member to the group
            squadGroup->Add(*squadmate);

            // Let squad leader know about new squad member
            g_pSquadLeader->AddSquadMember(squadmate);
```

As you can see, apart from the fact that the squadmates do not get access to the points of interest (that is a leader-only concept) and that they are assigned behaviors, the rest of the code is virtually identical. At this point we now have our squad fully assembled and all of the AI components are ready to go.

The last thing we must do for our squadmates is load up some avatars to represent them in the world. The animated 3D characters we will use are all going to be selected, loaded, and managed by the wrapper library, so all we need to do is request that they be loaded. The world interface exposed by the wrapper library provides a CreateNPC call to do just this. All we need to do is pass in a position and orientation for our squadmate and it will assign the pCharacter pointer passed in to a fully configured animated character ready for us to use.

```
            // Create the character
            ICharacter *pCharacter = NULL;
            g_pWorld->CreateNPC( position, rotation, &pCharacter );
            cNPC *npc = new cNPC( pCharacter, squadmate );

            // Add character to NPC list
            AI_NPCs.push_back( npc );
      }
```

The ICharacter type you see above is the wrapper library's exported character management interface. What we have done in this demo is create a second small wrapper class around all of these concepts, which we called cNPC. The cNPC class will essentially store and manage pointers to both the 3D rendering library's character type and our AI entity type. In a sense you can think of the cNPC type as having a body (ICharacter) and a brain (cSquadEntity) that will work together. The cNPC class declaration is shown below for convenience.

```
class cNPC
{
public:

       cNPC( ICharacter *pCharacter, cSquadEntity *pEntity );
      ~cNPC( );

      void Update          ( float elapsed );

private:

      ICharacter   *m_pCharacter;
      cSquadEntity *m_pEntity;
};
```

To finish off our world setup, we set a default command for the squad leader which basically tells the squad members to rally to his position. How this gets carried out in practice will be the subject of a later discussion.

```
      g_pSquadLeader->SetCommand( sc_RallyToLeaderPosition );
      return TRUE;
}
```

Our system is now fully configured and we are ready to look at what happens during the simulation.

StepSimulation

StepSimulation is the function that advances the game logic. It is responsible for making sure that the entities all update themselves and that the scene is animated and rendered given the elapsed time delta passed in. Since almost all of the logic is tucked away inside our entities and wrapper classes, this function has little to do except invoke their respective update calls.

We begin by updating our cWorld object, which in turns triggers all of our AI updates. We saw this in the last demo as well, so there is nothing that has changed here.

```
void StepSimulation( float dt )
{
      // update the entity world, this steps the AI simulation
      try
      {
            if ( g_pEntityWorld ) g_pEntityWorld->Iterate( dt );
      }
```

```
	catch(error_already_set)
	{
		PyErr_Print();
	}
```

The next step is making sure that our NPCs get a chance to update their physics in response to the artificial intelligence that was executed during the entity updates. We will look at the cNPC::Update call in a moment.

```
	// update the NPCs
	for(unsigned i = 0; i < AI_NPCs.size(); i++)
		AI_NPCs[i]->Update( dt );
```

Now we can call into our library to update the state of the simulation. This will apply all requested game physics and collision detection, animate the characters according to the commands issued by the AI, and render the scene into the frame buffer and present it to the viewer.

```
	// Advance game frame
	if( g_pWorld ) g_pWorld->Tick( );
```

Since the player ultimately controls the position of the squad leader avatar via user input collected in the Tick call above, we call back into our wrapper library to get the current player position after the physics updates have been run and update our squad leader entity. The same logic also applies to our squad mates since they too would have had their final positions calculated by the library.

```
	// Make sure that all of our entities get a copy
	// of their post-physics simulation positions
	if ( g_pSquadLeader )
		g_pSquadLeader->SetPosition(g_pWorld->GetPlayerPosition());

	for(unsigned i = 0; i < AI_NPCs.size(); i++)
		AI_NPCs[i]->GetEntity()->SetPosition(
					AI_NPCs[i]->GetCharacter()->GetPosition());

}
```

Now that we have an idea of what is happening at the top level, let us begin to look into what is going on in the individual update calls. As we saw in the last demo, our cWorld::Iterate method is responsible for making sure that all of the AI entities in the system update themselves properly. While most of the core logic remains identical to what we developed previously, we have added a few new items to make the demo more interesting.

But before we move on to look at our AI changes, let us quickly wrap up this section by examining the new NPC class update call and see what is happening there. This should give you an idea for what the AI is trying to achieve as its end goal. The cNPC class is essentially our bridge between the wrapper library and our AI code, so its job in this demo is to make sure that it communicates to the 3D library exactly what the AI has determined will be required in terms of position, orientation, and animation. So the Update call is going to be very simple indeed. All it basically needs to do is pass along the orientation and velocity settings that were determined by the AI during the world update so that the

objects can be properly updated in terms of their physics and subsequently rendered. It also needs to determine whether or not the AI wants the entity's weapon raised or lowered so that the wrapper library can play the proper animation. We will see how the weapon update takes place in the AI code a bit later in the text.

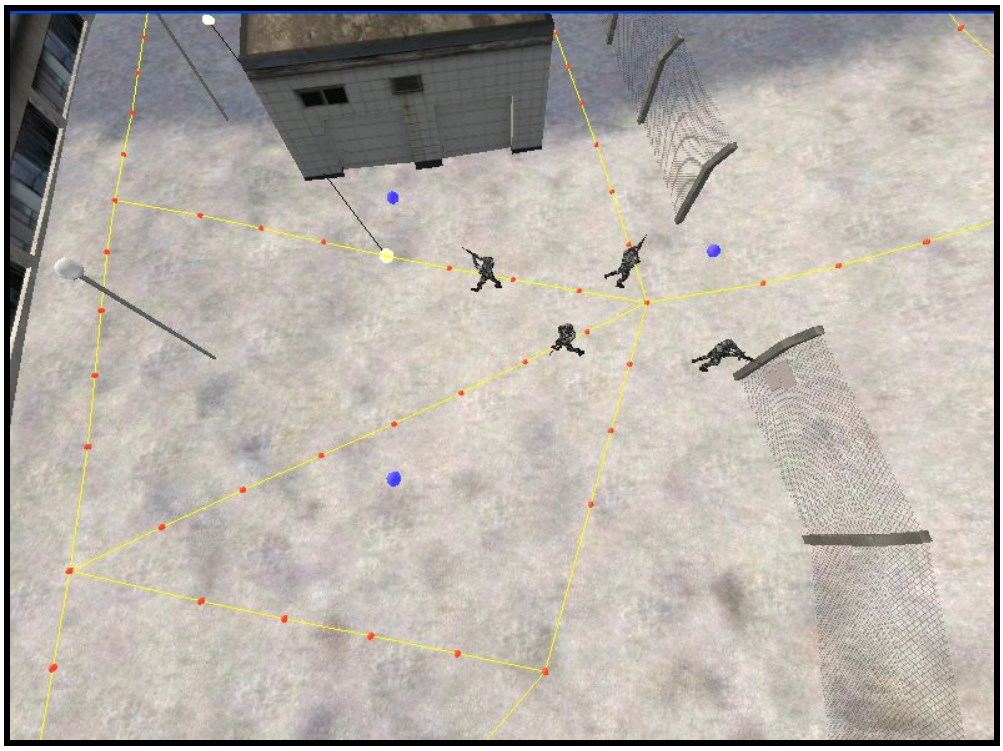
```
void cNPC::Update( float elapsed )
{
      //raise/lower the weapon based on AI direction
      if( m_pEntity->GetWeaponStatus( ) ) m_pCharacter->WeaponUp( );
      else  m_pCharacter->WeaponDown( );

      //apply the updated rotation
      m_pCharacter->SetOrientation(m_pEntity->Orientation());

      //set the movement on the character
      m_pCharacter->ApplyForce( m_pEntity->Velocity() * 125.0f );
```

Note that we request that the library apply a force to our entity (we use our velocity vector as the force direction and just scale up by a magic number), but we cannot be sure where it will wind up in the end because the position will likely be adjusted to account for obstacle collisions, terrain height, etc. This is why we made sure to copy back the position after the Tick call, as we saw above.

Finally, we do a bit of debug drawing to help visualize the pathfinding process. The next code block simply asks the library to draw a transparent sphere around the next waypoint the entity is heading towards and his ultimate destination. A thin black line will also be drawn between these two spheres to better portray the tracing along the path. (This can all be disabled via a menu option). If there is no "next waypoint", this would indicate that the NPC is not currently using the waypoint network and is instead heading directly to his goal. In this case, we draw a sphere-capped line between the current NPC position and its destination.

```
      //draw destination and next waypoint
      cWaypoint *pWaypoint = GetWorld( )->GetWaypointNetwork( )->
                  FindWaypoint( m_pEntity->GetNextWaypoint( ) );
      if( pWaypoint )
            GetWorld( )->DrawCappedLine( pWaypoint->GetPosition( ),
                                         m_pEntity->GetGoal( ) );
      else  GetWorld( )->DrawCappedLine( m_pEntity->Position( ),
                                         m_pEntity->GetGoal( ) );
}
```

In the image that follows, we can see the waypoint network with the goal and next waypoint both highlighted. Note the line that connects the next waypoint to the goal, passing through the small building.



We are finally ready to begin examining the changes that were made to the underlying AI source code. Please make sure that you have the project opened up so that you can follow along with the actual source as we discuss the updates and new additions.

A.I. Updates

Although the following two methods have not changed, it will be helpful to remind ourselves of what happens when the world class is updated during each frame. As you will recall, the only thing that the world needs to worry about is making sure that the AI entities are all updated. This all takes place within the respective iterate methods, shown below.

```
void cWorld::Iterate(float timeDelta)
{
	// iterate the world!
	tGroupList::iterator it;
	for (it = mGroups.begin(); it != mGroups.end(); ++it)
		(*it)->Iterate(timeDelta);
}
```

```
void cSquadEntity::Iterate(float timeDelta)
{
	if (mStateMachine)
		mStateMachine->Iterate();

	cEntity::Iterate(timeDelta);
}
```

So we can see above that the world tells every squad member to update itself. This involves making sure that the state machine is updated and that a call to the base class Iterate method is called. As you know, the state machine update handles all of the transitions between different action modes that the squad member needs to worry about. We will look at how the state machine affects the underlying squad leader and squadmate behavior in just a moment.

But first, recall that much of the high level AI management takes place within the base class Iterate method. For example, it is responsible for determining where the group members are situated and whether or not enemies are in the vicinity. It also makes sure that all of the relevant behaviors are applied that have been assigned for this entity type (we will examine those shortly). Finally, after the behaviors have been run, we will calculate our desired movement vector which, as we now know, will ultimately be passed along as a request for a position update in the wrapper library's physics simulation.

```
void cEntity::Iterate(float timeDelta)
{
	mPosition += mVelocity * timeDelta;

	// visibility check
	mVisibleGroupMembers.clear();
	mVisibleEnemies.clear();
	UpdateGroupVisibility();
	UpdateEnemyVisibility();

	// apply behaviors
	tBehaviorList::iterator it;
	for (it = mBehaviors.begin(); it != mBehaviors.end(); it++)
	{
		cBehavior *beh = *it;
		beh->Iterate(timeDelta, *this);
	}

	// clamp velocity rate of change (acceleration)
	float velChange = D3DXVec3Length(&mDesiredMoveVector);
	if (velChange > mMaxVelocityChange)
	{
		// clamp to max velocity change
		D3DXVec3Normalize(&mDesiredMoveVector, &mDesiredMoveVector);
		mDesiredMoveVector *= mMaxVelocityChange;
	}

	// apply change
	mVelocity += mDesiredMoveVector;

	// scale velocity (in the event we want to restrict movement)
	// scalars greater than 1.0 aren't a good idea if we are to
	// keep this system stable
	mVelocity.x *= mMoveXScalar;
	mVelocity.y *= mMoveYScalar;
	mVelocity.z *= mMoveZScalar;

	// clamp actual velocity
	float speed = D3DXVec3Length(&mVelocity);
	if (speed > mMaxSpeed)
	{
```

```
            // clamp to max speed
            D3DXVec3Normalize(&mVelocity, &mVelocity);
            mVelocity *= mMaxSpeed;
      }
}
```

So far, most of the code we have covered has been identical to the code in last demonstration. Things will start to change a bit in the next section as we examine the modifications to our squadmates' behaviors and to the way the squad leader does his job. In keeping with the flow of the application, we will begin with the squad leader and look at how commands are now going to be issued to the squad.

The Squad Leader

As mentioned earlier, our squad leader is going to be primarily player controlled. However, we have retained the ability to use a state machine in order to manage some simple tasks and to leave open the possibility that you could go either way with your own demos. For non-player controlled squad leaders, you will obviously want to have pure AI control of everything, so all of the prior squad leader functionality will remain intact. In this demo, we will simply allow the player to pass along commands to the squad via the keyboard and mouse and to control the world position and orientation of the squad leader.

If you have a look at the message processing callback function (see AI Demo.cpp) you will note that there are five different commands that the player can issue to the squad through the squad leader via key presses or mouse clicks. These commands are represented in the eSquadCommand enumeration that can be found in the file SquadEntity.h:

```
enum eSquadCommand
{
      sc_PatrolWaypointNetwork,
      sc_PatrolPointsOfInterest,
      sc_RallyToLeaderPosition,
      sc_StandGround,
      sc_AttackTarget,
      sc_NumCommands
};
```

As you can see, the squad can be instructed to randomly patrol the waypoint network or to go to random points of interest (both of which will be determined by the squad leader, as we will see in a moment). They can also be instructed to rally to the squad leader's position, stand their ground wherever they may be, or to form up and go after a target. These are commands that are common in lots of action games.

When the user presses any of the following number keys (1 – 4), they can issue one of the commands from the above enumeration. The callback function processing code looks as follows:

```
            case '1':
                  // Patrol waypoint network
```

```
                if (g_pSquadLeader)
                {
                        g_pSquadLeader->SetCommand(sc_PatrolWaypointNetwork);
                        g_pSquadLeader->SetSquadFormation(sf_SingleFile);
                }
                break;
        case '2':

                // Patrol random points of interest
                if (g_pSquadLeader)
                {
                        g_pSquadLeader->SetCommand(sc_PatrolPointsOfInterest);
                        g_pSquadLeader->SetSquadFormation(sf_Diamond);
                }
                break;
        case '3':

                // Rally to player
                if (g_pSquadLeader)
                {
                        g_pSquadLeader->SetCommand(sc_RallyToLeaderPosition);
                        g_pSquadLeader->SetSquadFormation(sf_SingleFile);
                }
                break;
        case '4':
                // Stop in place and idle
                if (g_pSquadLeader)
                {
                        g_pSquadLeader->SetCommand(sc_StandGround);
                        g_pSquadLeader->SetSquadFormation(sf_Abreast);
                }
                break;
```

Although you can probably guess what the call to SetSquadFormation does, for now just ignore it. We will discuss the squad formation behaviors shortly.

In addition to the random selection of waypoints and POIs, we have also provided the player with the ability to use the mouse to select targets in the world where the squad members should go (and theoretically attack, but we will leave the combat logic for you to implement as you see fit). Thus, when we process the right mouse button down message, we will ask the wrapper library to screen pick with the mouse cursor and determine the corresponding 3D world space point. The result will serve as a dynamic point of interest that will exist outside of the list of POIs that we loaded on application startup (this particular POI is just a module level variable stored in AI Demo.cpp).

Once we have our player-requested POI , we make sure the squad leader has it set as his current point of interest and ask him to issue the ‘attack target’ command to all entities in his squad. We will look at this function, along with the other squad leader command functions next.

```
        case WM_RBUTTONDOWN:
                SetCapture( hWnd );
                if (g_pWorld && g_pWorld->PickScreen( pick ))
                {
                        poi.SetPosition( pick );
```

```
                        poi.SetRadius( 100.0f );
                        g_pSquadLeader->SetSelectedPointOfInterest( &poi );
                        g_pSquadLeader->SetCommand( sc_AttackTarget );
                        g_pSquadLeader->SetSquadFormation(sf_Diamond);
                  }
                  break;
```

Issuing Commands

The squad leader is responsible for issuing commands to the troops in his assigned squad. In our current demo, the player is the squad leader and thus controls how the squad will behave. As you will recall, during application setup, we set an initial command for our squad leader that will request that all squadmates rally to the player's position. This will remain the state of the squad until another type of command is issued. The result is that unless otherwise ordered, the squad will attempt to follow the player around in the world.

The squad leader's SetCommand call seen above has been slightly modified to include support for the 'attack target' state. Because this state is going to be issued by the player via a mouse click and not by the squad leader's state machine, when this command is issued, we will manually handle the request. The updated SetCommand function is shown below.

```
void cSquadLeaderEntity::SetCommand(eSquadCommand command)
{
      mLastCommand = GetCommand();
      cSquadEntity::SetCommand(command);

      if(command == sc_AttackTarget)
      {
            SendSquadToTarget( );
      }
      else
      {
            for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
                 it != mSquadMembers.end(); ++it)
            {
                  cSquadEntity *entity = *it;
                  entity->SetCommand(command);
            }
      }
}
```

As you can see, this function is basically the same; it just commands the entities in the squad leader's group using their SetCommand calls to get them to take some action (we will look at how the squadmates work in the next section). The only real change was the addition of the call to SendSquadToTarget which handles processing the attack target command that was input by the player.

Let us have a look at the SendSquadToTarget function now since it is going to be very similar to the rest of the command functions we will look at in this section. As you can see in the next listing, the code is

actually very straightforward. After making sure that we have squad members that we can command, we grab the point of interest that was just set by the player when they clicked the right mouse button and pass it along to our squad members. This POI is going to be the goal that our squad is going to attempt to reach.

```
void cSquadLeaderEntity::SendSquadToTarget()
{
      mLastCommand = GetCommand();
      if ( mSquadMembers.size() == 0 )
            return;

      cPointOfInterest *poi = GetSelectedPointOfInterest();
      if (!poi)
            return;

      tPath pathToWP;
      for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
           it != mSquadMembers.end(); ++it)
      {
            cSquadEntity *entity = *it;
            entity->SetNextWaypoint(GUID_NULL);
            entity->SetPath(pathToWP);
            entity->SetGoal(poi->GetPosition());
      }
}
```

One important item to note in the above code is that, unlike the previous demo, the squad leader does *not* calculate the path to the goal for the squad members. All the squad leader needs to do is initialize each squad entity so that they know about the goal position. In this demonstration, squadmates will calculate their own paths on the fly (we will see how and why in the next section), and thus we set an empty path and a NULL next waypoint.

Since all of the squadmates will calculate their own navigation data, the same basic logic we see above holds true across all of the commands that will be issued to the squad. The remaining command functions are listed below so that you can see the similarities to the previous function. The only real difference is how the squad leader chooses the point in the world that the squad should attempt to reach.

```
void cSquadLeaderEntity::SendSquadToRandomPOI()
{
      mLastCommand = GetCommand();
      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;
	for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
	     it != mSquadMembers.end(); ++it)
	{
		cSquadEntity *entity = *it;
		entity->SetNextWaypoint(GUID_NULL);
		entity->SetPath(pathToWP);
		entity->SetGoal(poi->GetPosition());
	}
}

void cSquadLeaderEntity::SendSquadToRandomWaypoint()
{
	mLastCommand = GetCommand();
	if (mSquadMembers.size() == 0)
		return;

	tWaypointID wpID = SelectRandomWaypoint(*GetWaypointNetwork());
	cWaypoint *wp = GetWaypointNetwork()->FindWaypoint(wpID);
	if (!wp)
		return;

	SetSelectedWaypoint(wp);

	tPath pathToWP;
	for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
	     it != mSquadMembers.end(); ++it)
	{
		cSquadEntity *entity = *it;
		entity->SetNextWaypoint(GUID_NULL);
		entity->SetPath(pathToWP);
		entity->SetGoal(wp->GetPosition());
	}
}

void cSquadLeaderEntity::CommandSquadToRallyOnLeader()
{
	mLastCommand = GetCommand();
	if (mSquadMembers.size() == 0)
		return;

	tPath pathToWP;
	for (vector<cSquadEntity*>::iterator it = mSquadMembers.begin();
	     it != mSquadMembers.end(); ++it)
	{
		cSquadEntity *entity = *it;
		entity->SetNextWaypoint(GUID_NULL);
		entity->SetPath(pathToWP);
		entity->SetGoal(Position());
	}
}
```

While the code allows the player to send the squad to random POIs or random waypoints on the network by issuing keyboard commands, those same functions will continued to be called by the squad leader state machine (just as before) after the initial command is given, as long as the player does not change commands. Recall that this happens when all squadmates reach their intended target.

With the exception of one addition change to one squad leader function, the rest of the code in the squad leader class is the same as the last demo. So for now, we have covered everything we need to know about how the squad leader does its job. We will examine that final code adjustment to the squad leader momentarily. Now it is time to start examining the squad members and see how things have changed.

The Squad Members

Most of the major changes in this application have taken place at the squad member level. In this demo, our squad members are granted a bit more independence and they are also going to be somewhat more coordinated than they were before. As we saw in the last section, the squad leader is no longer responsible for calculating paths for the squad members. This is now going to be done by the individual squad members as and when they need to. All they need to know is where they are instructed to go and they will try to find the best path possible to get there.

Following Orders

The best way to begin the examination of our squad member AI is to go back for a moment and recall the means by which the entities are updated. Recall that in our cEntity::Iterate method, the first thing that happens is the update of the entity's state machine. As in the last demo, our entities' state machines are quite simplistic; they are really only concerned with waiting for commands, being in transition to a particular location, and doing something (or not) when they reach that location. In the last demo, whenever a squad member reached a waypoint or point of interest, we automatically updated the color of the circle that represented that entity. This time around we will actually use the blind data information that is stored at the waypoints to update the animation state of the entity. The animation that we will control via the blind data will raise or lower the entity's weapon. While not exactly thrilling, it does demonstrate the point. Of course, you will obviously experiment with your own ideas for what sort of blind data you want to store and how you want to react to it when you implement your own games.

The function that will handle this transition is listed below. It is actually the only function that is called from our Python script that has changed since the last demo. As you can see, all we have added is a call to extract the blind data, test its value, and update a Boolean flag that indicates whether we want our weapon raised or lowered when we reach this particular waypoint. The actual animation update request takes place in the cNPC::Update call that we looked at earlier.

```
void cSquadEntity::OnWaypointReached()
{
      if (!mCurrentWaypoint.IsEqual(GUID_NULL) && mWaypointNetwork)
      {
```

```
            cWaypoint *wp = mWaypointNetwork->FindWaypoint(mCurrentWaypoint);

            ULONG weapon_status = 0;
            wp->GetBlindData( 0, weapon_status );
            if ( weapon_status > 0 )
                  mWeaponUp = true;
            else
                  mWeaponUp = false;
      }
}
```

After the state machine updates have taken place and all of the scripted code executes, recall that the next thing that the cEntity::Iterate method does is apply any behaviors that have been assigned to this entity. This is where most of our new code will live, so we will look at the behavior system next.

Behavior Updates

In our last demonstration, our squad members were each assigned paths to the same destination by the squad leader. To make things a bit more interesting this time around, we have decided to try something new. Rather than send every entity to the same location in the world, we are going to try to organize our squad into simple formations and have it operate in a more uniform and more visually pleasing manner.

We are also going to make our entities a little smarter with respect to their navigation. Not only will each entity be responsible for figuring out how to get to its goal (which we will discuss shortly), but they will also be able to short-circuit the network traversal process and head straight to their goal as soon as they are able to see it.

Entity Pathfinding

We will begin with the updates to our pathfinding behavior (see PathfindBehavior.cpp). While much of the function that handles our entity world navigation is identical to the previous version, some significant changes have been made. Let us step through the updated Iterate function for our pathfinding behavior a few lines at a time to get a feel for the new way that the AI will work.

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

      // get the squad member's position, path and target waypoint
      tPath &path = squadmate.GetPath();
      tWaypointID wpID = squadmate.GetNextWaypoint();
      D3DXVECTOR3 entityPos = entity.Position();
      D3DXVECTOR3 desiredMoveAdj(0.0f, 0.0f, 0.0f);
```

So far, everything is pretty much as it was before. We cast our entity to the squad type and extract its current path, current position, and the destination waypoint that we are trying to get to.

The next section of code is new. It attempts to determine whether or not the entity can currently see the goal that it wants to get to by using a line of sight test provided by the wrapper library. If the entity has a clear path to its goal (i.e., no environmental obstructions), it will go "off the network" and head straight for it. This will involve clearing its current path and setting its next waypoint to NULL.

```
	cWorldVisibilityFunctor functor;

	//if we can see the destination,
	//short circuit the path finding and go straight there
	squadmate.IncrementTimeSinceLOSTest(timeDelta);
	if ( squadmate.GetTimeSinceLOSTest() > mMaxTimeBeforeLineOfSight )
	{
		squadmate.SetLOSStatus(functor.IsVisible( entityPos,
							squadmate.GetGoal() ) );
		if ( squadmate.GetLOSStatus() )
		{
			path.clear();
			wpID = GUID_NULL;
			squadmate.SetNextWaypoint( wpID );
			squadmate.SetPath( path );
		}
		squadmate.ResetTimeSinceLOSTest();
	}
```

To minimize CPU costs, we do not bother running the line of sight test every frame. Instead, each entity maintains its own internal timer and we trigger its line of sight test every *n* seconds and cache the result. In general, once an entity has a direct line of sight to its target, it will not lose it. However, it is possible that somewhere along the way, before the goal is reached, something goes wrong or the environmental conditions change and the entity loses track of the target. In this case, as we will see below, the entity will need to rebuild a path to the target since it wipes out any previous path once the line of sight is established. It would not do much good to maintain the previous path once the entity goes off the network because the 'next' waypoint that was previously stored might be quite a distance away by the time the entity veers off course and learns that the line of sight has been lost.

For this demo, the entity just rebuilds a path when the target is no longer in sight. If you feel that this is too expensive a cost, you could easily modify the system to accommodate *n* line of sight test failures before a new path is built. This way, if the entity happens to reacquire a direct line to the target, it can just continue on its way without needing to get back on the network.

The next section of code deals with the case where the next waypoint to travel to is NULL, but a path has been built. In this case, we simply pop the next waypoint off the path list, set it as our entity's next waypoint, and reset the timer that keeps track of how long it takes to travel between waypoints (recall that we use this to determine when to agitate the entity, just in case it is having trouble getting to its desired location).

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

Alternatively, if we have no next waypoint and we also have no path that has been calculated, we enter the next code block.

```
	else if (wpID == GUID_NULL || path.empty())
	{
```

The first thing we need to do is check to see whether the entity might have already arrived at its goal. If it has, then we can come to a halt at this point and wait for the next command to be issued. As in our last demo, we can optionally apply a little bit of avoidance logic to try to keep the entities from potentially bunching up. After resetting the waypoint timer, we simply return since there is nothing left to do.

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

If the entity is not at its goal, does not have a path, and does not have a next waypoint to travel to, then it is possible that it is currently on a straight line path to the goal (as a result of an earlier successful line of sight test). If this is the case, it can just continue on its way. However, if there is no current line of sight that has been established (perhaps a previously valid LOS has been lost), then there is no choice but to calculate a new path to the goal using our pathfinding routine (A* in this demo, just like the last).

```
		else
		{
			if (!squadmate.GetLOSStatus())
			{
				cWaypointVisibilityFunctor functor2;
				mWaypointNetwork.FindPathFromPositionToPosition(entityPos,
										squadmate.GetGoal(),
										functor2, path);
				squadmate.SetPath(path);
				wpID = squadmate.GetNextWaypoint();
			}
		}
	}
```

The next section of code is basically the same as it was in the prior version (albeit in 3D space now). We begin moving to our target by either selecting the next waypoint and adjusting our movement vector to take us there, or we assume that if the path is empty, we have reached the end of the path and are close enough to our goal that we can go off the network and head straight for it.

```
	cWaypoint *wp = mWaypointNetwork.FindWaypoint(wpID);
	if (wp)
	{
		D3DXVECTOR3 wppos = wp->GetPosition();
		desiredMoveAdj = wppos - entityPos;
		if (D3DXVec3Length(&desiredMoveAdj) < wp->GetRadius())
		{
			// cache the current waypoint
			squadmate.SetCurrentWaypoint( wp->GetID() );

			// 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
			{
				// no more waypoints, start walking toward our target
				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();
```

For the most part, the combination of our line of sight tests and the waypoint network pathfinding algorithm should keep our squad members on track to their respective goals. However, there is always

the possibility that a squad mate can veer off course or find himself hopelessly stuck somewhere in the level. As we did in the last demo, we include the concept of agitation to allow the entity to choose a new path to try to get out of the jam and hopefully find a way to get back on course. This behavior only kicks in when some fixed amount of time has passed without successfully reaching the goal. Essentially, by taking the 3D cross product of the desired direction vector and the world up vector, we generate a new direction vector that is perpendicular to our desired move vector and set it as our new desired movement.

```
      // do we need to agitate a bit to get back on track?
      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();
      }
```

The final piece of this function sets our newly calculated desired movement vector and applies some avoidance to try to keep the entities from bunching up too much when in close proximity. Recall that this desired movement vector will ultimately translate into a velocity vector in the entity's Iterate method and then be forwarded on to the game physics system for final position adjustment.

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

Squad Formation

The final behavior that we need to examine is a new one that we have introduced in this lab project. When dealing with multiple entities that are supposed to be exhibiting group properties, it is often preferable to apply some degree of organization with respect to the manner in which they spatially arrange themselves. Until now, we have pretty much left our entities to their own devices regarding where they wind up traveling in the world. While we did include some degree of avoidance logic to minimize the cases of entities randomly wandering into one another (and our wrapper library provides coarse entity-to-entity collision detection and response to handle the worst cases), the end result was still a bit chaotic on screen.

To make our squad look more like a squad, we decided to model a very basic set of formations using simple shapes (line, diamond, and pentagon). Using this approach, we will assign each of our squad members specific positions in the formation and those positions will be their eventual individual goals. In a moment, we will look at one of the simpler formation behaviors to see how it all works. First however, we will examine the Iterate method for the formation behavior as this is where most of the logic happens.

```
void cFormationBehavior::Iterate( float timeDelta, cEntity &entity )
{
      cSquadEntity &squadmate = dynamic_cast<cSquadEntity&>(entity);

      // safety test -- the player controls the squad leader, so we can just return
      if ( squadmate.GetSquadID( ) == 0 )
            return;
```

As you can see above, first we cast our input entity to our squad type used in this demo. This will give us access to data that we may want to have handy when setting the formation. Then we do a quick test to see if the entity passed in is the one representing the squad leader. In this demo, the player controls the squad leader, so there is little point in wasting cycles trying to position him in the world. Of course, if you are using an AI controlled squad leader, you would want to remove this line (or give him a different ID) and process that leader as you see fit.

The formation positions that we will be creating rely on the fact that we have assigned a “point man” in our squad. Generally, this will be the squad member tasked with leading the squad from the front in certain types of formations. All other squad members will use this point man as their means for determining where they need to position themselves within the formation. In this demo, with the exception of the pentagonal formation which uses the squad leader directly, we have arbitrarily assigned the point man job to squad member 1 (mSquadID == 1).

In the next section of code, we iterate through our squad list and attempt to cache pointers to both our point man and our squad leader. The squad leader pointer is going to be helpful for creating formations that are based on where the squad leader is situated (e.g., following a rally on leader command).

```
      mSquadLeader = NULL;
      mPointMan    = NULL;

      cWorld &world = squadmate.World();
      for (tGroupList::iterator git = world.Groups().begin();
           git != world.Groups().end(); ++git)
      {
            cGroup *grp = *git;
            tEntityList::iterator eit = grp->Entities().begin();
            for ( ; eit != grp->Entities().end(); ++eit )
            {
                  cSquadEntity *sm = dynamic_cast<cSquadEntity * >(*eit);

                  if ( sm->GetSquadID( ) == 0 )
                        mSquadLeader = sm;

                  if ( sm->GetSquadID( ) == 1 )
                        mPointMan = sm;
            }
      }

      //bail if we did not find the point man (error)
      if( !mPointMan || !mSquadLeader ) return;
```

Now that we have our squad leader and point man, we will extract their current orientation vectors (look and right are all we really need since up will always be <0,1,0> in our demo). We will be using these orientation vectors to help place our other squad members in their appropriate positions in the formation.

```
	D3DXMATRIX mtxOrient;
	D3DXMatrixRotationQuaternion(&mtxOrient, &mPointMan->Orientation( ));
	mPointManLook  = D3DXVECTOR3(mtxOrient._31, mtxOrient._32, mtxOrient._33);
	mPointManRight = D3DXVECTOR3(mtxOrient._11, mtxOrient._12, mtxOrient._13);

	D3DXMatrixRotationQuaternion(&mtxOrient, &mSquadLeader->Orientation( ));
	mLeaderLook  = D3DXVECTOR3(mtxOrient._31, mtxOrient._32, mtxOrient._33);
	mLeaderRight = D3DXVECTOR3(mtxOrient._11, mtxOrient._12, mtxOrient._13);
```

We are now ready to direct the entity passed into the function to his appropriate place in the squad, so in the next section of code we enter a switch statement that checks to see what the current squad formation should be according to the squad leader and calls the appropriate function to manage the formation setup. Recall that when we were discussing the squad leader's SetCommand style calls during the user input processing handler, one of the lines that we saw was a call to SetSquadFormation. This was where the squad leader made a decision to organize the squad using one formation or another. In practice, if you are using an AI squad leader (rather than the player) then this is probably something that you will want to have more heavily tied into your squad leader's state machine so that he can determine what the best formation is for the given situation. Note as well that this is information that can easily be stored in the waypoint blind data areas if you wanted to override the formation orders issued by the leader (perhaps upon reaching a waypoint, the point man might decide that a certain scenario calls for one formation over another given the situation on the ground).

We will look at the formation setup functions in just a moment, but as you can see in the code below, they expect a reference to the current entity as well as references to two 3D vectors. These vectors will be populated by the function with the goal location where the entity should be headed and the direction that they should be facing as they head towards that goal. The goal in this case will be their position in the squad and the direction will determine whether the animation system walks the assigned character forward, backwards, or strafes left/right.

```
	D3DXVECTOR3 goal, dir;
	switch( mSquadLeader->GetSquadFormation ( ) )
	{
		case sf_Pentagon:
			Pentagon( entity, goal, dir );
			break;
		case sf_Diamond:
			Diamond( entity, goal, dir );
			break;
		case sf_Abreast:
			Abreast( entity, goal, dir );
			break;
		case sf_SingleFile:
			SingleFile( entity, goal, dir );
			break;
	}
```

The direction vector returned from the formation setup call will need to be converted into a quaternion so that our cNPC::Update call can pass it along to the wrapper library. To convert the 3D vector, we simply create a rotation matrix describing the local coordinate system for the entity and convert it to a quaternion.

```
      dir.y = 0.0f;
      D3DXVECTOR3 rt, lk, up(0.0f,1.0f,0.0f);
      D3DXVec3Normalize(&lk, &dir);
      D3DXVec3Cross(&rt, &up, &lk);
      D3DXVec3Normalize(&rt, &rt);

      D3DXQUATERNION q1; D3DXMATRIX m1;
      m1._11 = rt.x; m1._12 = rt.y; m1._13 = rt.z; m1._14 = 0.0f;
      m1._21 = up.x; m1._22 = up.y; m1._23 = up.z; m1._24 = 0.0f;
      m1._31 = lk.x; m1._32 = lk.y; m1._33 = lk.z; m1._34 = 0.0f;
      m1._41 = lk.x; m1._42 = lk.y; m1._43 = lk.z; m1._44 = 1.0f;

      D3DXQuaternionRotationMatrix(&q1, &m1);
```

Finally, we can set the new goal and orientation for the squad member and the formation behavior is complete.

```
      //update the goal and orientation
      squadmate.SetGoal(goal);
      squadmate.SetOrientation(q1);
}
```

Note that our goal position is going to be the position in the squad that is determined by the shape of the formation we are employing. In order to get to that position, our squad member will use the pathfinding behavior that we examined earlier. That is, he will attempt a line of sight to his requested position and if he can get there, he will head in that direction without using the waypoint network. If for some reason, that position is not in his direct line of sight, he will attempt to use the waypoint network to build a path to that goal.

The final piece of the formation puzzle is examining how our squad formation positions are determined. Once we take a look at one of the cases below, you should have no trouble recognizing how the other formations work. Basically they all do the same thing – determine a goal and a direction vector based on where in the formation the entity should be located. This will be typically be based on the position and orientation of the point man in the squad (which was why we extracted this information earlier).

```
void cFormationBehavior::SingleFile( cEntity &entity,
                                     D3DXVECTOR3& goal, D3DXVECTOR3& dir  )
{
      cSquadEntity &squadmate = dynamic_cast<cSquadEntity&>(entity);
      switch ( squadmate.GetSquadID( ) )
      {
            case 1:  //POINT MAN
                  goal = squadmate.GetGoal();
                  dir  = goal - squadmate.Position();
                  break;
```

```
            case 2:  //RIGHT FLANK
                  goal = mPointMan->Position() - (mPointManLook * 70.0f);
                  dir  = mPointManLook;
                  break;

            case 3:  //LEFT FLANK
                  goal = mPointMan->Position() - (mPointManLook * 110.0f);
                  dir  = mPointManLook;
                  break;

            case 4:  //REAR GUARD
                  goal = mPointMan->Position() - (mPointManLook * 160.0f);
                  dir  = mPointManLook;
                  break;

            default: //ANYONE ELSE JUST HEADS TO GOAL
                  goal = squadmate.GetGoal();
                  dir  = goal - squadmate.Position();
      };
}
```

As you can see above, if the entity being processed is the point man, he will simply set his goal to be whatever destination the squad leader commanded the squad to go to (a waypoint, a point of interest, a target, etc.). For everyone else, they will set their goal positions based on some offset from the point man. In this particular case we want them to line up single-file, so all non-point man entities will seek positions that are behind the point man, offset by some distance. They will also assume an orientation that matches the point man's orientation. This need not be the case of course. For example, if you look at the diamond shaped formation in the source, you will notice that the point man orients in the direction of the goal, squadmate 2 lines up behind and to the right of the point man and faces outwards to cover the right flank, and squadmate 3 lines up behind and to the left of the point man and orients to cover the left flank. Finally, squadmate 4 brings up the rear and faces backwards to provide cover to the squad for any attacks that may come from behind.

The only formation that presents a special case is the pentagon formation because it uses the squad leader as the point man. This formation will only apply in the case where the squad members have been commanded to rally to the leader position and the command remains in effect. Essentially the squad will assume positions and orientation that provide for a nearly 360 degree field of fire (outwards from the 5 vertices of the pentagon, assuming the leader faces forward). As you can see in the code below, we base our squad member positions on offsets from the player. The same is true for orientation, where we set up clockwise firing directions every 45 degrees.

```
void cFormationBehavior::Pentagon( cEntity &entity,
                                   D3DXVECTOR3& goal, D3DXVECTOR3& dir  )
{
      cSquadEntity &squadmate = dynamic_cast<cSquadEntity&>(entity);

      D3DXVECTOR3 v1, v2;
      D3DXMATRIX m;
      D3DXMatrixRotationY(&m, D3DXToRadian(45.0f));
      D3DXVec3TransformNormal(&v1, &mLeaderLook, &m);
      v2 = v1;
      v2.x = -v2.x;
```

```
      switch ( squadmate.GetSquadID( ) )
      {
            case 2:  //RIGHT FLANK
                  goal = mSquadLeader->Position() - (mLeaderLook * 30.0f)
                                                  + (mLeaderRight * 50.0f);
                  dir  = v1;
                  break;

            case 3:  //LEFT FLANK
                  goal = mSquadLeader->Position() - (mLeaderLook * 30.0f)
                                                  – (mLeaderRight * 50.0f);
                  dir  = v2;
                  break;

            case 1:  //POINT MAN (REAR GUARD 2)
                  goal = mSquadLeader->Position() - (mLeaderLook * 90.0f)
                                                  – (mLeaderRight * 25.0f);
                  dir  = -v1;
                  break;

            case 4:  //REAR GUARD
                  goal = mSquadLeader->Position() - (mLeaderLook * 90.0f)
                                                  + (mLeaderRight * 25.0f);
                  dir  = -v2;
                  break;

            default: //ANYONE ELSE JUST HEADS TO GOAL
                  goal = squadmate.GetGoal();
                  dir  = goal - squadmate.Position();
      };
}
```

It is also worth noting that this particular formation will only be assumed once the squad has arrived at the player position. Earlier we mentioned the fact that there was one squad leader function that was going to be left undiscussed until later. This function is actually the familiar SquadArrivedAtGoal function that we introduced in the last demo. Recall that this function is called from our Python scripted state machine to determine whether or not the squad has completed its journey to its assigned destination. The state machine uses this information to basically figure out which command should be assigned next (or which new goal should be set as the next destination). While the function still returns either a true of false status, it now has one added bit of functionality as well. In the case where the currently issued command happens to be a “rally on leader” command, if the squad has arrived at its goal (i.e., the leader position) the squad leader automatically issues a set formation order to assume the pentagon shape discussed above.

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

```
	//if rallying to leader, once arrived, assume pentagon formation
	if( GetCommand() == sc_RallyToLeaderPosition )
	{
		SetSquadFormation( sf_Pentagon );
	}

	return true;
}
```

Note that while the squad might assume any formation as it is traveling to reach the leader, once it arrives, the pentagonal formation is automatically assumed. This is obviously just a design choice and you can remove this functionality if you wish. For example, with only minor modification to the other formation behaviors, you can allow the squad to assume other shapes where the player assumes the role of point man. It is entirely up to you.

Finally, before concluding this section, it is worth noting that we have also provided the player with ability to manually change the squad formations using the keyboard. In the user input message handling function, there is some additional code that allows for dynamic updates to the current formation (shown below):

```
	. . .
	case '5':
		// Line up
		if (g_pSquadLeader)
		{
			if(g_pSquadLeader->GetCommand() == sc_RallyToLeaderPosition)
				g_pSquadLeader->SetCommand(sc_StandGround);
				g_pSquadLeader->SetSquadFormation(sf_SingleFile);
		}
	break;
	case '6':
		// Abreast
		if (g_pSquadLeader)
		{
			if(g_pSquadLeader->GetCommand() == sc_RallyToLeaderPosition)
				g_pSquadLeader->SetCommand(sc_StandGround);
				g_pSquadLeader->SetSquadFormation(sf_Abreast);
		}

	break;
	case '7':
		// Diamond
		if (g_pSquadLeader)
		{
			if(g_pSquadLeader->GetCommand() == sc_RallyToLeaderPosition)
				g_pSquadLeader->SetCommand(sc_StandGround);
				g_pSquadLeader->SetSquadFormation(sf_Diamond);
		}
	break;
	. . .
```

The only item to note above is that if the current command is the rally on leader command, we must override it in order to get the squad to assume the requested formation. Otherwise it will continue to

attempt to line up in the pentagonal formation that is assumed when the squad is following the player around the world. In this case, we simply choose to tell the squad to stand their ground once they have assumed the new formation. That wraps up all of the formation specific code in this demo. While this demo obviously maintained a very simplified approach to squad organization, hopefully some of what you have seen will summon ideas for building your own squad formations.

Conclusion

Although this brings us to the end of our final demo in this course, in many ways it is only the beginning of the road for you. Hopefully some of the concepts that we introduced here have sparked your imagination and you are now brimming with new ideas about AI concepts that you will want to implement in your own games. Certainly there are many interesting squad level behaviors that you should be able to put together in short order. For example, now that you have some idea about how to create formations, you can probably envision all sorts of scenarios for how you want to arrange the "shape" of your squads (e.g., formations for firefights, door breaches, patrols, sieges, prisoner guarding, etc.). Combining your script-capable state machines with waypoint blind data can also lead you to various means for controlling how your squads navigate the world. For example, it should not be very difficult for you to add the concept of "cover" waypoint types so that your squad members can seek cover during an advance on a target. You could also implement leap-frog behavior where one squad member advances to a point, takes aim and provides cover, while the next squad member advances to a position a bit closer to the target and then returns the favor. In short, there are plenty of very interesting behaviors and patterns that you should now be able to conjure up simply using the tools provided in this course.

While you should certainly continue to experiment with your AI and be as creative as you can possibly be, one of the things that should be fairly clear at this point is that you will need a robust 3D graphics engine at your disposal if you really want to assume full control of the simulation. The 3D Graphics Programming series offered here at the Game Institute is an excellent place to get started down that road if you have not taken those courses already. As you can see even in this very simple tech demonstration, the power and flexibility that a 3D rendering and physics system can provide makes life much easier when working out how your AI entities will behave. Plus, you will also learn about many concepts that can be used directly in your AI toolset, even if you are not interested in graphics programming. For example, in the 3D Graphics Programming Module II course that we borrowed heavily from in this project, you will examine spatial partitioning data structures like quad-trees, kD-trees, and BSP-trees. These data structures and their accompanying algorithms come in very handy, as we have seen even in this demo. Although we only used them (a quad-tree behind the scenes, actually) during navigation and for simple line of sight tests, you can certainly find other use cases (e.g., speeding up queries to find the closest enemies, hierarchical pathfinding routines, determining whether health or ammunition packs can potentially be seen from a given location in the world using PVS, etc.).

In addition, while the mathematics that we used in this demonstration is not overly complicated, it is clear that math cannot be avoided when working with AI in 3D worlds. If you are having trouble with any of the 3D math concepts that we used, please be sure to get some help in the Game Mathematics course. You will certainly need to know all of this stuff if you intend to build 3D games of any sort.

And finally, as mentioned very early on in the course, there is also a short seminar available here at the Game Institute which provides an introduction to a form of artificial intelligence which we did not avail ourselves of in this course – life systems. It is a very interesting read and certainly a nice follow up to this material if you would like to branch out a bit more and expand your skill set.

With that said, it is our hope that you have enjoyed this course and that you will take what you have learned and apply it to good ends. We wish you the very best of luck in all of your future game programming endeavors!