3.6 Game Architecture

In This Chapter

Overview

The code necessary to create modern games is anything but simple. Gone are the days when the source code for a full game was just a couple of files and we didn't have to worry about overall structure and architecture. In today's games, with code bases exceeding a million lines of code, it is vitally important to have a well-defined architecture in order to understand the source code, add new features, and ship the game on time.

Main Structure

Most games make a distinction between *game-specific code* and *game-engine code*.

Game-specific code is, as the name implies, tailored to the current game being developed. It involves the implementation of specific parts of the game domain itself, such as the behavior of zombies or spaceships, tactical reasoning for a set of units, or

the logic for a front-end screen. This code is not intended to be generically reused in any other game in the future other than possibly direct sequels.

Game-engine code is the foundation on top of which the game-specific code is built. It has no concept of the specifics of the game being developed, and deals with generic concepts that apply to any project: rendering, message passing, sound playback, collision detection, or network communication.

Both game-specific code and game-engine code are large enough that they are often split into several modules. Depending on how the project is organized, these modules can be static libraries, dynamically linked libraries (DLLs), or sometimes simply subdirectories in a project.

Architecture Types

When discussing different architectures, we will often talk about *coupling*. Coupling is a measure of how tightly two parts of the code are connected to each other. Loose coupling means that there's only a slight connection between the two sets of code, which is the ideal situation because it makes it possible to change one without affecting the other. The greater the coupling, the harder it is to modify or replace one without affecting the other. Really tight coupling means that the two sets of code are highly dependent on each other.

Game code is often organized in one of the following ways, which are discussed in order of increasing structure and complexity:

Ad-Hoc Architecture

Code bases developed this way don't have any apparent organization. They often grow organically, with code being added as needed without looking at the big picture. Different subsystems are not identified, let alone isolated, which leads to extremely tight coupling between all parts of the code. This approach works fine for projects with very small code bases (a few dozen files), but is very limiting in large projects, making development difficult and costly, and makes it virtually impossible to reuse the code in future projects (see Figure 3.6.1).

Modular Architecture

In a modular architecture, specific subsystems are clearly identified and separated into modules or libraries. Modules can vary in how they interface with the rest of the game. On one extreme, they can just be a group of related objects or functions that anybody can use, and on the other extreme, they can present a unified facade to the rest of the system.

Reuse and maintainability of code are greatly improved over an ad-hoc architecture. This approach also allows easier integration of middleware packages since modules can be more easily replaced or even wrapped to present a unified interface to the rest of the engine. However, dependencies between modules are not controlled, so over time, things often degenerate into a situation where every module communicates directly with almost every other module, leading to tighter coupling than we would ideally want (see Figure 3.6.2).

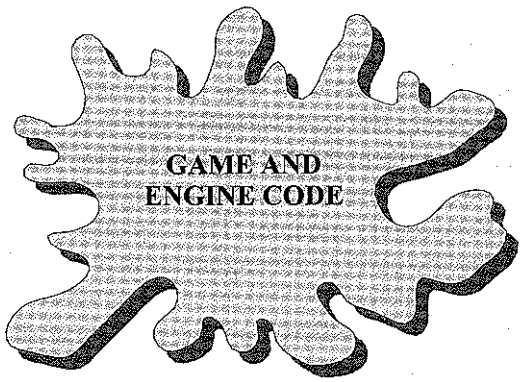

FIGURE 3.6.1 *Code base with an ad-hoc architecture.*

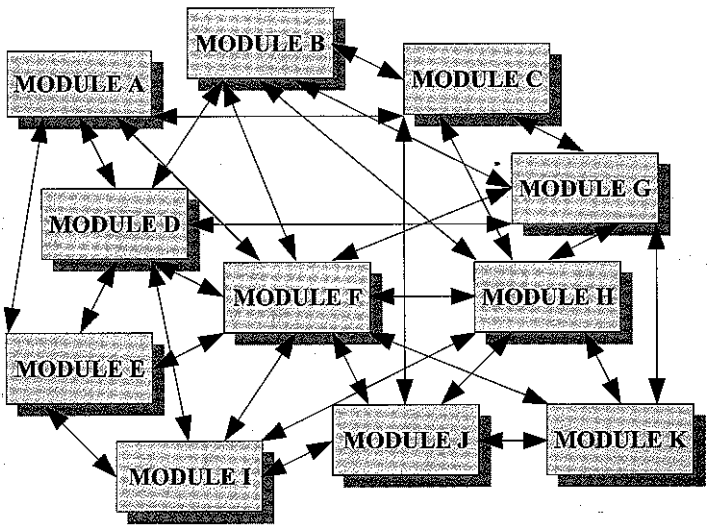

FIGURE 3.6.2 *Code base with a modular architecture.*

Directed Acyclic Graph (DAG) Architecture

A DAG architecture is a modular architecture in which the dependencies between modules are tightly controlled. Think of modules as nodes in a graph, and their dependencies as the edges. A DAG architecture requires that there be no loops in the dependencies. Therefore, if Module A depends on Module B, Module B cannot depend on any module that directly or indirectly depends on Module A.

This arrangement allows us to classify some modules as being higher or lower level than others, and has the advantage that it keeps modules ignorant of any modules that are higher level than they are, which reduces the overall coupling between modules. Typically, the game-specific code will be the highest level module since it depends on all of the high-level engine modules. It also makes it easy to decide to use only part of the code base, since by selecting any branch of the DAG we're guaranteed to get all the necessary modules. This is particularly useful when creating tools that only need specific parts of the engine, or that are completely game independent and don't need any game code.

Ideally, we'd like the DAG to be as wide (or as shallow) as possible, which means that the chain of dependency between modules is relatively short. That makes it easier to add, replace, or remove modules without affecting the rest of the program. The worst case is when all the modules are lined up, each of them depending on the previous one (see Figure 3.6.3).

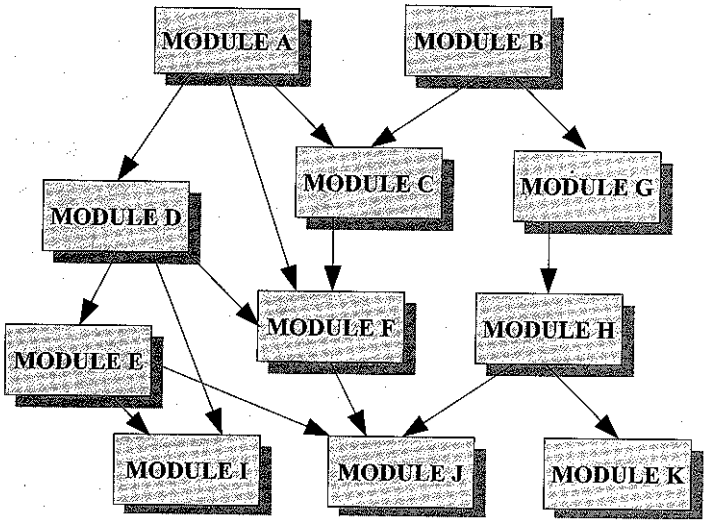

FIGURE 3.6.3 *Code base with a DAG architecture.*

Layered Architecture

A layered architecture is an arrangement like the DAG architecture in that there cannot be any cyclic dependencies between modules, but it takes a step further. While the DAG architecture allowed a module to access any other module underneath itself, here modules are arranged in rigid layers, and a module can only interact with modules in the layer directly below.

This type of architecture is more heavyweight than a simple DAG arrangement, and it can sometimes lead to duplication of code or interfaces in order to expose certain functionality available in a lower layer to a layer that is several levels above. Some domains are very well suited to a layered architecture, such as network communication, because, by their nature, they perform many serial operations from layer to layer. However, the code base of a game is not as rigidly structured, so it might not be a perfect match for a layered approach.

This type of architecture is more desirable when the number of modules grows to be quite large, and keeping track of individual dependencies between modules becomes too difficult. At that point, the layers can act as another level of organization on top of the DAG arrangement of the modules within each layer (see Figure 3.6.4).

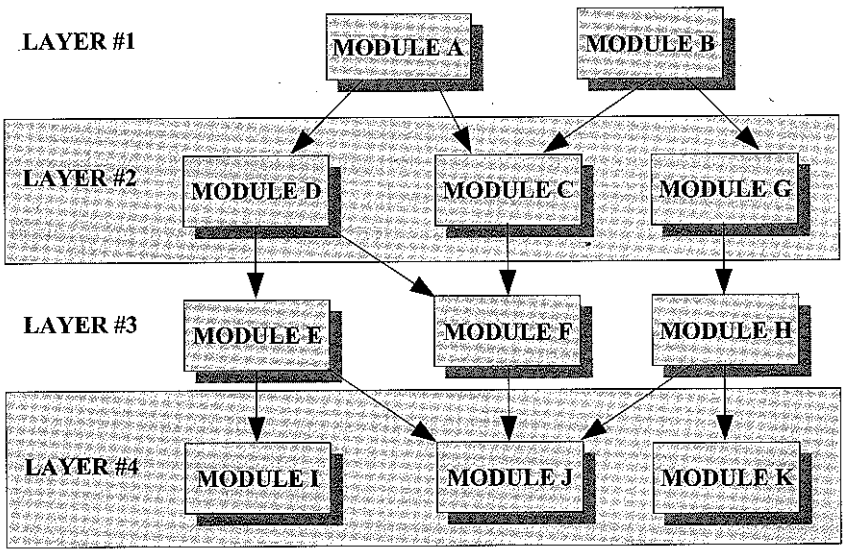

FIGURE 3.6.4 *Code base with a layered architecture.*

In-House Tools

The in-house development tools always warrant special consideration. These are the tools we create for the artists and developers so they can create awesome worlds and fill them with amazing content. How do these tools fit into the overall architecture?

One of the first things we need to decide is how much functionality will be in common between the game engine and the tools. Perhaps the only way the tools and the game interact is by creating a series of XML files that can be parsed in the game. In that case, they don't need to have any common code, and keeping them as two separate code bases would be a good idea.

Another possibility is that tools require some of the game functionality, but not all of it. For example, they might need to load resources and write files back out. Or, they might need to render some 3D graphics, but they have no need for any of the AI, collision, or networking code. In that case, ideally we should use only the parts of the game code that are necessary for the tool. If we have a DAG or layered architecture, it should be very easy to grab only the parts in which we're interested. After all, nobody wants to initialize the graphics renderer just to run a command-line tool.

Finally, our last option is full integration. If the tools need full access to all areas of the game (which is a suspicious situation that might indicate an ad-hoc or modular architecture), then we can make it so the tools use all of the game code. Or, take it even further, and make it so the tools are part of the game engine itself and you just have one executable for everything.

As with the overall architecture, thinking about this ahead of time and deciding which approach to use will make it easier to develop the tools and ensure they integrate well with the game engine itself.

Bird's-Eye View of a Game

Before we go any further and start getting into details, let's have a quick look at what the game does at the highest level.

Here is a sequence of events that a typical game will go through from the moment it starts until the moment it exits:

1. Game initialization
2. Main game loop
   a. Front-end initialization
   b. Front-end loop
      i. Gather input
      ii. Render screen
      iii. Update front-end state
      iv. Trigger any state changes
   c. Front-end shutdown
   d. Level initialization
   e. Level game loop
      i. Gather input
      ii. Run AI
      iii. Run physics simulations
      iv. Update game entities
      v. Send/receive network messages
      vi. Update time step
      vii. Update game state
   f. Level shutdown
3. Game shutdown

As you can see, there are some patterns in this sequence of events. We repeatedly see initialization and shutdown events for a variety of phases. These events are always paired together, and there's a matching shutdown for each initialization. We also see several game loops that run through a sequence of operations every frame. The next two sections cover each of these features in detail.

Initialization/Shutdown Steps

Initialization and shutdown of different systems and phases of the game is a very important step, yet it is often overlooked. Without a clean and robust way of initializing and shutting down different parts of the game, it becomes very difficult and error prone to switch between levels, to toggle back and forth between the game and the front end, or to even run the game for a few hours without crashes or slow downs.

Overview

The purpose of an initialization step is to prepare everything that is necessary to start a certain part of the game. For example, the front-end initialization step could take care of initializing the GUI system, loading some common art resources, and setting the correct state into the player profile.

```
void FrontEnd::Initialize() {
    GUI::Initialize();
    LoadCommonResources();
    PlayerProfile::Set(LoadPlayerProfile());
}
```

The shutdown step usually has a set of statements to undo everything the initialization step did, but in the exact reverse order they were listed in the initialization step. Every system that was initialized is shut down, every object that was created is destroyed, and every resource that was loaded is freed.

It is very important that the shutdown step undoes everything that was done in the initialization step. Failure to do so will result in memory leaks, subsystems that are never shut down, and plenty of other subtle bugs. In addition, to minimize the potential for bugs, it is recommended that it does everything in the opposite order than the initialization step. That way, we can be sure that we don't free a resource or shut down some system that is required by a later part of the program.

```
void FrontEnd::Shutdown {
    // Nothing to free for the player profile
    ReleaseCommonResources();
    GUI::Shutdown();
}
```

This is how the initialization and shutdown functions would be called from the game program:

```
{
    FrontEnd frontEnd;
    frontEnd.Initialize();
    frontEnd.Loop();
    frontEnd.Shutdown;
}
```

Notice the conspicuous absence of any error handling in the previous functions. In practice, you will want to check for errors in initialization and loading and deal with them following the game's policy for error handling: perhaps displaying the error, logging it and continuing as normal, trying to recover gracefully, or just by using the `assert()` function and letting a programmer fix it right away.

Resource Acquisition Is Initialization

A good rule to follow to minimize mismatch errors in initialization and shutdown steps is to use the *Resource Acquisition Is Initialization* philosophy (often abbreviated as RAII). That means that creating an object will acquire and initialize all the necessary resources, and destroying it will take care of destroying and shutting down all those resources. The advantage of this approach is that the initialization and shutdown calls are automatically called as the objects themselves are created and destroyed, so there's no potential to forget to call them, or to have mismatched calls.

Continuing the previous example, if we used RAII with the `FrontEnd` class, its initialization will be done in its constructor, and all its shutdown operations in its destructor. The functions `Initialize()` and `Shutdown()` shouldn't even be listed as public because nobody should call them directly. If an object of the class `FrontEnd` is around, we can be sure it has gone through the initialization phase successfully. Now we can use the `FrontEnd` class in the following way, and all the bookkeeping happens behind the scenes:

```
{
    FrontEnd frontEnd;
    frontEnd.Loop();
}
```

How do we check for errors now? Earlier we could just have the `Initialize()` function return a Boolean variable indicating whether it had succeeded or failed. Now we're doing the initialization in the constructor, which doesn't return any values. The shutdown step happens in the destructor and can't return any values.

The cleanest way of handling it is to use exception handling. Instead of cluttering all the source code with statements checking for errors, and returning out of a function as soon as they find a problem, we can just wrap the previous front-end creation into a try-catch statement:

```
try {
    FrontEnd frontEnd;
    frontEnd.Loop();
}
catch (...) {
    // Handle any problems here
}
```

It's a clean solution, but unfortunately not always practical. Dealing with exceptions can be a tricky issue, especially in C++. It requires writing exception-safe code, which requires that all resources are freed correctly whenever an exception is triggered, even if it happens in the middle of a constructor. At the very least, it will require the use of smart pointers (which is mostly a good thing) and that some programmers brush up on the consequences of dealing with exceptions.

Another issue is the support for exception handling in different languages and compilers. Unfortunately, in C++ not every compiler will deal with exceptions very well. Some of them will generate very inefficient code that might affect the performance of the game. Some console manufacturers even admit that their compiler doesn't really generate good exception-handling code and that you shouldn't count on it for your games.

Even if you can't use exception handling, it's still worth using the RAII approach whenever possible. You will have to check manually whether the initialization was successful, but you will still benefit from knowing that the object is always in a well-known state and not having to worry about calling the matching shutdown step by hand.

Optimizations

The following are some techniques sometimes used in games to make the shutdown phase faster or more reliable.

Fast Shutdown

Ideally, we want to be able to transition between levels or between the game and the front end as quickly as possible. During the initialization step we need to load some resources, so we try to do that as quickly as possible, but sometimes, you'll be surprised to see that you spend a second or two in the shutdown step for the game level before we start loading anything. Reducing that time to a minimum would definitely improve the user experience and keep players happier and more engaged in your game.

What exactly are we doing that takes so long? Usually, the shutdown step for the game is not trivial; after all, we need to destroy the entire world we had loaded in memory and bring everything back to its initial, pristine state. Games will usually iterate through all the game entities in the world and free them one at the time. Then they will do the same with each of the resources that were loaded (textures, geometry,

etc.). They'll reset or destroy complicated data structures with many nodes and edges, and do a lot of general cleanup, most of which involves freeing memory that was allocated dynamically.

If we're just going to wipe the entire level, do we really need to be so careful and meticulous by deleting each little piece one bit at the time? It's like taking down a house by carefully removing each brick and placing it neatly in a pile. An alternative is to make sure all dynamic memory allocations for the game level happen in a set of memory pools dedicated exclusively to that level. At the end, when it's time to shut down the level, all we have to do is to reset those memory pools to an empty state, which is a very fast operation. As long as no code ever tries to access any objects or memory that was allocated in those memory pools, we have safely reduced the shutdown time for a level from a second or two to virtually nothing.

Warm Reboot

Your game should be able to run for hours on end without crashing. That's more difficult than it sounds. Just because it runs fine for a few levels doesn't mean that it can last for three days of nonstop play (some publishers require that the game runs for that long without any problems before they approve a game). Even if the game doesn't crash after several days of play, it often develops annoying side effects, such as a choppy frame rate caused by memory leaks, memory allocation problems caused by memory fragmentation, or jittery animations produced by loss of precision in the game timer.

One drastic but very effective way to avoid these problems is to do a warm reboot of the machine after each level. Clearly, this is not something we can do on a PC, but it will work fine on a game console since it's fully dedicated to the game. As long as the warm reboot is fast enough and the game loads immediately and preserves all its state, the player won't notice the difference, but the machine will be reset to its initial state, without developing any long-term problems. To make this even more seamless for the player, some game consoles provide functions to display an image on the screen while the machine is rebooting instead of going blank or flickering.

Main Game Loop

At their heart, games are driven by a game loop that performs a series of tasks every frame. By doing those tasks every frame, we put together the illusion of an animated, living world. Sometimes, games will have separate loops for the front end and the game itself, since the front end usually involves a smaller subset of tasks than the game. Other times, games are organized around a unified main loop. This section describes in detail the different tasks done during a game loop and ways of implementing them.

Tasks

The tasks that happen during the game loop perform all the actions necessary to have a fully interactive game, such as gathering player input, rendering, updating the world, and so forth. It is important to realize that all of these tasks need to run in one

frame. In the case of a game that runs at 30 frames per second, that means all the tasks for a frame have to be done in less than 33.3 ms. If we choose to run the game at 60 frames per second, then we have half that amount of time: 16.6 ms. A task can't decide to take an unusually long time in one frame and run over its allotted time, because it would affect the overall frame rate and detract from the game experience. If a task really needs a long time to complete, it has to be broken down into multiple steps and executed across several frames. Even in games that allow for a variable frame rate, it is desirable to avoid sudden changes in frame rate to provide smooth gameplay.

The following are the main tasks just about every game needs to perform in its game loop at one point or another:

Time Step

Once upon a time, in the dark ages of game development, games didn't bother using clocks. They just did as many things as they could in one frame (which was determined either by how much the CPU could do, or until the next vertical synch signal of the monitor). That worked fine as long as the game only ran on that exact same hardware. As soon as somebody with a faster CPU tried to run it, the game became faster. Not only did the game run at a faster frame rate, but everything actually moved faster on the screen, causing the game to quickly become unplayable.

Today, things are different. Most contemporary games use some form of clock to drive the game and make it independent of the speed of the system on which they are run. Even console games that can always count on running on the same hardware usually benefit from using a clock to deal with updating the game at different video frequencies for PAL and NTSC video systems.

Most of the computations done during the game loop involve updating objects to reflect all the changes that happened since the last frame (or since the last time they were updated). Since we want to avoid our game changing speed depending on the actual frame rate, we use the duration of time since the last pass through the game loop as the amount of time to move our simulation forward for this frame.

The time step in the game loop updates the game clock to match the hardware clock and computes how many milliseconds elapsed since the last time step. These are the only two sources of time information we will use during this frame. If we went back and read the hardware clock every time we needed to know the time elapsed, we would get slightly different readings throughout the frame because the hardware clock never stops. Instead, we just read the time values once at the beginning of the frame, and use them throughout the entire simulation for this frame.

There are two ways of handling time in games: *variable frame duration* and *fixed frame duration*. Most games on PCs, as well as many console games, use variable frame duration. That means that frames can last any amount of time depending on what is displayed on the screen, what the player is doing, or any number of factors. Sometimes, frames will be blazingly fast and only take 10 ms (100 frames per second), and sometimes they can slow down and take 50 ms (20 frames per second). This approach

has the advantage that it scales very well to different hardware configurations, different content, and different game loads. It also lets players tweak the game settings to achieve a quality versus speed tradeoff.

Fixed-frame duration games are most commonly found on consoles, where the hardware is unchanging and frames can always be assumed to be of the exact same duration (which will usually coincide with a multiple of the vertical synch signal on the display). Fixed-frame duration has some nice properties, such as more predictable behavior, easier physics simulation, and more reliable network behavior. Even if you plan to have fixed-frame duration, it's still a good idea to make sure your game measures and uses the actual time duration of each frame for its simulation instead of assuming a fixed time per frame. That way, the game will be able to react correctly to a hitch in the frame rate (a much bigger explosion than you had anticipated, for example), and it will make it easier to run at a different fixed frame rate, such as on a PAL video system (which runs at 25 frames per second instead of 30 frames per second on an NTSC system).

Input

A game is an interactive experience, so one of the most important tasks is to gather the player input and react accordingly. We get input through a variety of input mechanisms: gamepad, mouse, keyboard, driving wheel, video camera, or any of a myriad of custom input devices. The important thing is that the input be consistent and as close as possible to instantaneous.

To provide the user with consistent input, it is best to sample the input device once per frame and save those values for the rest of the frame. Otherwise, if we sample the input device every time we need to know its state, we could end up with very inconsistent results within the same frame.

It is important to minimize the amount of time elapsed between the moment we sample the input from the device and the moment our game reacts to the input. This will give the player a better impression of responsiveness and control. To minimize that time, we typically get the input at the beginning of the game loop, right before the simulation step. Otherwise, if we left the input task for the end of the loop, we would be reacting to the player a frame behind, and everything would feel lagged by about 30 ms (or less in the case of higher frame rates).

Networking

Another aspect of input is the input we receive from the network. At some point in the game loop, we need to collect all the new messages we received from the network, and deal with them accordingly, by updating game entities or providing new input.

Simulation

This is a huge task that encompasses all sorts of subtasks, and it is where the world really comes alive. The simulation step takes care of running any AI behavior code so AI entities decide what to do next and where to go. It runs any game code or scripts that update the game state or trigger new events. It runs physics simulations to make

sure objects move correctly on the screen. It updates particle systems to make the misty waterfalls and the fiery explosions come alive. It moves the animations forward for all the visible characters. It updates the player's position and camera based on the input recorded in an earlier task.

Because of the sheer number of computations to do and the entities to update, this is often the most expensive of all the tasks we do in a game loop. So much, that for a game to run at a reasonable frame rate, it is important that we limit the number of entities and the type of updates we do every frame. We don't need to waste any time with an AI entity that isn't activated, and we don't need to continually update an enemy that is moving inside a building three blocks down the street. Deciding what to update and when to update it is one of the largest performance boosts we can do in some of today's games with large worlds filled with lots of game entities.

Collision

In the previous task, when we did the simulation step, we just moved everything to the position where it would ideally like to be. In this phase, we check for collisions between entities and deal with them accordingly. This is done in two separate phases: *collision detection* and *collision response.*

Collision detection is the simpler of the two phases. For each entity, we need to detect whether it's colliding with another entity. By *entity*, we really mean anything in the world: another character, an arrow, or even the ground. This is relatively straightforward, but it's not a cheap operation. We usually try to speed it up by providing simplified volumes to collide against, and we only check for collisions for the entities we care about (the ones that are directly in view or nearby).

The second phase is collision response. This can be much more complicated than the detection part. The response phase deals with correctly updating the entities that have collided (*correct* being defined by the consistent laws of the game, not necessarily by reality). If a character collides with an arrow, we need to assign damage and notify the character that it has been hit. If a car collides with a wall, we need to crumple the car, and change its position and velocity to account for the collision. Making sure that those things look realistic requires a solid physics simulation and some good game tuning.

Object Updates

Now that we have run the simulation and dealt with any collision issues, it's time to update the objects to their desired position. Here we're applying the correct transform to the objects, updating all their children, applying animations to a skeleton, and so forth.

If we have our world structured as a scene graph, this is when we propagate states up and down the tree such as render or game states.

Rendering

Finally, the moment we've been waiting for during this entire frame. Now that everything is in place, we finally get to display it on the screen.

Again, because of the large worlds in today's games, we can't just throw the entire world at the graphics hardware and expect to have decent performance. We first need to identify all the objects that could be potentially visible, and then pass down only those objects to the graphics hardware. To do that, we can use a variety of spatial partitioning techniques, such as portals or BSP trees, and perhaps combine them with a simple frustum cull against the frustum defined by the camera.

This operation needs to be repeated once for every camera we're displaying on the screen, as would be the case in a split-screen game, or for a rear-view mirror in a driving game. Sometimes we need to render the scene more times if we have any real-time reflections or environment maps.

Many techniques are applied at this time to achieve realistic shadows, complex lighting models, full-screen processing, and so forth. Rendering is an active area of research, and every year new techniques are being developed to create new effects and more realistic visuals. The graphics hardware also keeps advancing at a breakneck pace, which contributes to the improvement of game visuals every year.

Other

There are plenty of miscellaneous tasks that have to be done during each frame, so it's important that they are included in the game loop. For example, we might need to tend to the sound system and update it once per frame to keep all the sounds loading and mixing correctly, or perhaps we batched all the outgoing network packets created during the frame and we need to send them at once at the end.

Structure

We now know what a game loop does, but how exactly is it structured? The most straightforward approach is to simply have a `while` loop with all the steps included in the loop. The following code is a typical game loop:

```
while (!IsDone()) {
    UpdateTime();
    GetInput();
    GetNetworkMessages();
    SimulateWorld();
    CollisionStep();
    UpdateObjects();
    RenderWorld();
    MiscTasks();
}
```

Such a game loop has the virtue that it's simple, straightforward, and very clear. The steps are clearly spelled out, and it's very easy to add new steps or remove existing ones. However, the steps are hardwired in the loop itself, so what if we want to have different steps depending on the state of the game? For example, if we're not playing a game on the network, we don't need to have a network step. We can easily fix that by doing a check at the beginning of the `GetNetworkMessages()` function itself, but there might be other similar situations.

One of the most common places where we need a very different game loop is the front end. Conceptually, it is very similar: we want to loop, get player input, update the state of the menus, and render them, but unless we have some sort of 3D front end, we probably don't want to do collision detection, network updates, or many other game-specific tasks. The same applies for other game states such as the loading screen while we're loading level data, or special transitions between levels.

A possible solution is to have multiple game loops, one for each major game state. However, that solution involves duplicating a lot of code, and is error prone and hard to maintain. Every time we make a change to one of the game loops, we need to think about how it should affect all the other loops, or suffer subtle bugs that could be hard to track down.

A more flexible alternative is to consider each of the steps in a game loop as generic tasks, and have the game loop simply iterate through all the tasks and call the `Update()` function in each:

```
while (!IsDone()) {
    for (Tasks::iterator it=m_tasks.begin();
         it != m_tasks.end(); ++it) {
        Task * task = *it;
        it->Update();
    }
}
```

Now we can control exactly what steps we want to perform in the game loop from the game code itself. That means that we can have a single game loop for all the states of the game, including the front end itself. Whenever we transition from the front end to the game, we just add the correct tasks to the main loop and continue running as usual.

Coupling

So far, the game loop we have seen has been extremely simple. Every pass through the loop corresponds to a frame, and at every frame, we perform the same set of operations. Not all game loops are structured like this, however. A common technique is to decouple the rendering step from the expensive simulation and update steps. This technique allows a game to run the simulation at a fixed rate (e.g., 20 times per second), while still rendering as fast as possible. It combines the advantages of a fixed time step simulation with the scalability and improved frame rates of a variable time step game.

For a decoupled main loop to work effectively, we can't just run the simulation 20 times per second and render the graphics 100 times per second. If we did that, many of the frames we would render on the screen would be duplicates of the previous frame because the game state didn't change. To solve this problem, before rendering the screen, we interpolate any position and rotation values based on their previous position and known velocity. This arrangement can result in higher frame rates, smoother animation, and better overall responsiveness than a fully coupled game loop.

We can implement a decoupled game loop by using two threads: one for the simulation, and one for the rendering. It seems like a good idea on paper, but unfortunately, multithreading programming can be a tricky business, and it's much more prone to errors than single-threaded approaches are. The performance cost of context switches between the threads and synchronization to the same set of data can also add up and become a significant drain on performance.

Since we have a tight game loop that repeats every 30 ms or so, we can do the scheduling ourselves without much trouble. The main loop would look something like this:

```
while (!IsDone()) {
    if (TimeToRunSimulation())
        RunSimulation();

    InterpolateState();
    RenderWorld();
}
```

The function `RunSimulation()` could be implemented using the flexible task approach described in the previous section. `RunSimulation()` only gets executed at fixed intervals, and the rest of the loop runs as fast as possible.

Execution Order

When we talked about the different steps involved in a game loop, we didn't really address the order in which they were executed. For the most part, it doesn't matter too much, and the game will run fine whether we do network message gathering at the beginning or toward the end. We're going around the game loop constantly anyway. However, there are a few situations in which execution order of the different tasks is important.

In a game, the player is constantly interacting with the world. One of the goals we want to achieve is to keep that interaction as seamless as possible, which means reducing the delay between the time the player interacts with the world and the time the game is updated to reflect that interaction. For example, if the player moves the mouse, we want to move the camera as soon as possible, not 100 ms from now. If we waited that long, the game would feel sluggish even if it had a very high frame rate.

To achieve this, we want to minimize the time between the input gathering step and the time the rendering happens with those changes taken into account. A natural arrangement would be to gather input, perform simulations, and render the frame in that order. After rendering, we can take care of noncritical tasks such as updating the sound system.

We also want to reduce the time between receiving network messages and the time we process them in the game. If we were to delay them for a few frames, we would add another 30–60 ms delay to the messages, which is a significant percentage of the 100–200 ms that they spent traveling through the network already.

Another reason to be careful with the execution order of the different steps in the game loop is to maximize the parallelism we can achieve between the graphics hardware and the CPU. Most video cards in PCs nowadays, and all modern game consoles, have dedicated graphics hardware. This hardware can work in parallel without affecting the CPU, so we want to maximize this parallelism.

The best ordering of steps will depend on your specific hardware. Most graphics hardware have buffers to queue instructions, so they can be somewhat forgiving about when the data was sent to them, and they'll work at full efficiency as long as this buffer is always full. Ideally, we want to keep this buffer full while the CPU is busy working on the simulation for the next frame. This is represented in Figure 3.6.5a. Contrast that with the worst situation shown in Figure 3.6.5b, where the CPU sends a bunch of graphics data to the graphics hardware, and then waits until it's all done before continuing on to the next frame. In the first case, our game will run twice as fast and smoothly as in the second one.

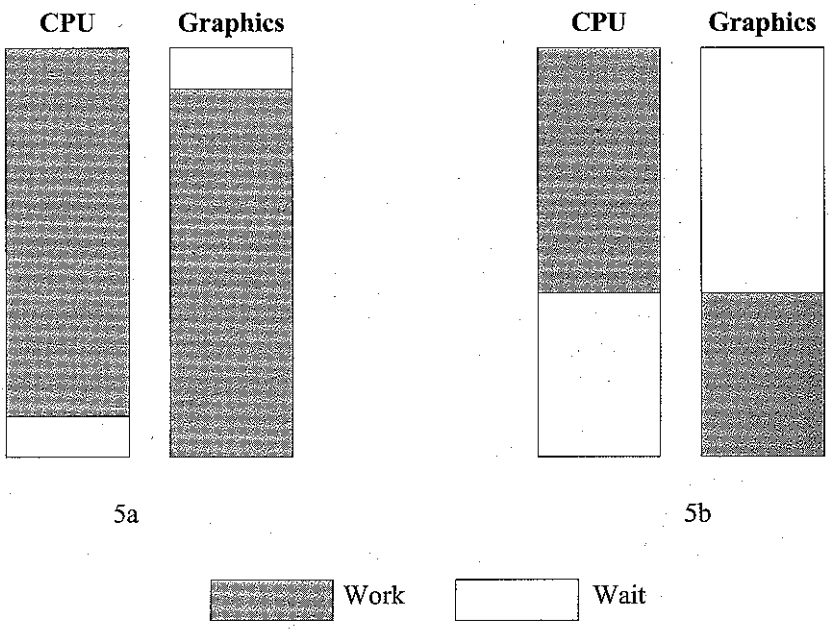

FIGURE 3.6.5 *a) A well parallelized game loop. b) A game loop not taking advantage of the parallelism between the graphics hardware and the CPU.*

As we move into the next generation of game consoles and modern PC hardware, the presence of multiple CPUs will become increasingly common. Games architected today will need to take advantage of the parallelism between the different CPUs to achieve the best performance. How exactly to achieve this while minimizing the

amount of communication between threads in different CPUs will be an active area of research for the next few years.

Game Entities

A game is all about interacting with the world, but it's the smelly orcs and telepathic aliens, the faster-than-light spaceships and the all-terrain vehicles, the rocket launcher and the magical sword that inhabit the world and make it an unforgettable experience for the player. Those are the game entities and those are what ultimately make a game. This section describes how game entities are handled in a game: how they are organized, how they interact with each other, and how they can be put together to create a full game.

Definition

Up until now, we've been talking about game entities without ever defining them rigorously. We have been relying on an intuitive understanding of what a game entity is. Frankly, that's because a game entity is a very fuzzy and slippery beast. It can really refer to anything in a game world that can be interacted with.

Some examples of game entities are obvious: an enemy unit is a game entity, a sword you can pick up is a game entity. Others are a bit fuzzier, because we don't think of interacting with them so much, but the sky dome is probably a game entity as well since it needs to be rendered, animated, and moved along with the player. In addition, possibly so are effect-rich objects like fires and waterfalls; even the level geometry or heightfield itself might be considered a game entity. Other game entities that people sometimes forget about include triggers that generate some action when the player enters an area, or even the camera that is controlled by the player; they don't have any physical representation in the world, but they're still an essential part of the game.

Looking back over those examples, a good definition of game entity might be "a self-contained piece of logical interactive content." It is broad enough to cover all the previous examples, but fuzzy enough to deal with just about any type of game entity. The most important part of that definition is the self-contained aspect. A piece of clothing might be a game entity if it can be picked up in the world and equipped, but it wouldn't be considered a game entity if it's just decoration on a player avatar and can't be changed.

Deciding what is and what is not a game entity is not an arbitrary decision. As we'll see in the rest of this section, a game entity has a certain memory footprint and performance cost associated with updating it and traversing it. Only those things that we really are going to interact with should become game entities.

Organization

Conceptually, all the game entities in the world are stored in a list. We then iterate through the list to update the entities every frame, to render them, or to perform any

other operations. It is important that entities be stored in a list, and not in an array or vector, because some game entities tend to be very volatile and new entities are constantly being created (bullets, spawned enemies, dropped items), while others are being destroyed (fading corpses, finished explosions, and projectiles after hitting a target).

In practice, keeping all of the entities in a list and traversing them linearly is probably too much of a naive and slow approach for anything but games with just a handful of entities in the world. We usually want some better organization that allows the game to perform its simulation and rendering steps as quickly as possible.

Here's where games vary a lot. A real-time strategy game will have a very different organization of game entities than an indoor first-person shooter or a fighting game. In general, entities will be stored in data structures that allow the game to do whatever operations it needs to do most efficiently. For example, a real-time strategy game could use a grid structure to quickly have access to all entities in a particular region of the world since the world can be easily projected on a 2D plane. However, a first-person shooter might use BSP trees or portals as a more efficient method given the type of environments it has to deal with.

However, we might want to perform different operations with conflicting requirements on the game entities. One interesting approach to solving this is not to limit ourselves to a single arrangement of game entities, but to use as many data structures as is necessary for the different operations we want to perform. That way, collision detection can use a type of data structure that is highly optimized to find contact between objects, but the rendering code can use one that can quickly cull objects that are outside of the camera frustum and occluded by other objects. To do that, we would go back to the original idea of keeping all entities in a single list, and then each type of data structure would keep a reference to an entity in the list.

If we're going to have the same entity referenced in many different data structures, it is crucial that all data structures remain in sync. If the entity is moved or deleted, all data structures need to be automatically updated. The observer design pattern is a very clean solution to this situation.

Updating

One of the operations we want to perform very frequently on game entities is to update them. Normally, we give each game entity a chance to do all the updates it needs to do once a frame during the simulation step. This involves running any AI, scripts, physics simulations, triggering events, or sending messages to other entities.

We usually want every entity to have a chance to run its update code once per frame, but unfortunately, that is often too expensive with large worlds and complex entities. Instead, we can try being smart about what entities we update, and only deal with those that are near the player or are important to the game in some way. The ones that are out of sight and not currently having a direct effect on the game can be left for later when we have a few CPU cycles to spare.

Game entities are sometimes organized hierarchically, so instead of storing them in a straight list, we store them in a tree structure. This allows us to impose some type of hierarchy on the entities: if the parent entity doesn't need to be updated, we can skip updating all the children. This can cut down the number of updates we need to do per frame if the tree is deep and well populated. Sometimes, this approach will be mixed with a logical spatial partitioning, by having all entities in a room or sector in one branch of the tree, so we can ignore them if we're nowhere near the room.

A more general technique involves using a priority queue of game entities to decide which ones to update every frame. The idea is that entities are added to the priority queue and sorted based on the time when they next need to be updated. Therefore, an entity that is right in front of the player would be sorted toward the front of the queue, while an entity that is very far away would remain at the back. The importance of the entity would also affect the time of update, so a very important entity would be updated frequently, even if it is far away.

The key concept is that now we can just start popping entities off the front of the queue, calling their update functions, and putting them back in the queue. Whenever the entity at the front of the priority queue doesn't need to be updated this frame, we can stop all the processing of entities. The biggest win is not so much reducing the number of entities that are updated, but not even having to traverse the full list of entities asking them one at the time whether they need to be updated or not. In modern hardware, traversing a list with many elements and accessing each can be painfully slow due to the constant cache misses. In contrast, the priority queue method only needs to access the exact number of entities we updated this frame and not one more.

Creation

There's more to creating game entities than meets the eye. At first, we might think it's a trivial operation: there is a class that corresponds to every game entity, and we simply do a `new` on that class to create an object of that time. That was easy.

It turns out that things aren't that simple. Depending on how you structure your game entities, you might not have one class per game entity (see the "Component Systems" section in Chapter 3.4), so you can't `new` an object whenever you want. In addition, we will need to create game entities by name (or ID) when we initially load a game level or a saved game, so using `new` directly is out of the question.

From the requirements we just listed, the creation of game entities is a perfect match for an object factory. An object factory will take care of creating the correct game entity on request. It is not limited to simply creating one specific object, but can create any other objects it needs if it uses a component approach.

In Chapter 3.4, we presented a simple object factory that created game entities. Here we will describe a more complex version, called an *extensible object factory*, which allows us to register new object types to be created at runtime. The advantages of an extensible object factory are many. One of the immediate consequences is that the factory itself doesn't have to know about every item it can create, so the coupling (and

physical dependencies) between the factory and the items it produces is greatly reduced.

A second consequence is that it is much easier to add new object types to the factory, just by registering them at runtime. This makes it easier to add new objects during development, and opens the door to extending our game after it ships by having DLLs or some other dynamically loaded code register any new object types with the factory.

Let's start with a simple implementation of an extensible object factory. Our factory needs a `Create()` call, and a way to have the program register (and unregister) new object types:

```
class ExtensibleGameFactory {
public:
    GameObject * Create(GameObjectType type);
    void Register (FactoryMaker * pMaker, GameObjectType type);
    void Unregister (GameObjectType type);
private:
    typedef std::map<GameObjectType,FactoryMaker*> TypeMap;
    TypeMap m_makers;
};
```

Every time we register a new maker type, we add it to the map structure that gives us very fast mapping between the type of an object and a pointer to its maker. Creating new objects is just a matter of looking up the map and calling the creation function on the maker if we find one:

```
GameObject * ExtensibleGameFactory::Create(GameObjectType type) {
    TypeMap it = m_makers.find(type);
    if (it == m_makers.end())
        return NULL;
    FactoryMaker * pMaker = (*it).second;
    return pMaker->Create();
}
```

As you can see, the game factory knows absolutely nothing about what object types it is creating this time around. That detail is totally left up to the maker itself.

What steps do we have to take to add a new object to this factory?

1. Define a new object type (which is just an enum at the moment).
2. Create a maker class that creates the object we want.
3. Register it with the factory at the beginning of the program.
4. Unregister it at the end of the program.

So, there's a fair amount of work involved. If we're going to have hundreds of such object types, it could get cumbersome taking all those steps every time we need to add a new object type. It turns out that we can automate quite a few things in that process.

First, the object type can be changed so it's not an enum, but a unique ID that every object type has. This avoids having to explicitly add an entry to an enum list, and also means we don't have to have a centralized list of all object types, which would make it harder to extend from other sections of the code or from a DLL.

If we have some sort of runtime type identification in our engine, we could use a unique ID from that system for every class. Otherwise, we could simply create a class static variable in the classes we're interested in, and use its memory address as the unique ID into the map since it would never change and would be guaranteed to be unique. The only drawback is that it might not be the same between different executions of the program, so we should make sure we use a type ID that will not change for saved games and level files.

Another approach is to simply use a unique string for every new object type. It won't be as efficient as a 32-bit unique ID, but it will be easy to debug and can be saved to disk without any problems because it will never change.

If all the objects are going to be created in more or less the same way, taking the same type and number of parameters in their constructors, and performing the same set of operations, we can wrap the `maker` class in a template so they're automatically generated.

Finally, since the game factory doesn't know anything specific about the type of objects it creates, other than their base type, we can also create a template for the factory and reuse it in other places of the game where we need to create objects by type. The companion CD-ROM includes the source code for a fully templated extensible factory you can start using right away in your projects.

This is how we would use the templated extensible factory in the game:

```
ExtensibleObjectFactory<GameObject> m_factory;
m_factory.Register(new FactoryMaker<GameClass_1>);
m_factory.Register(new FactoryMaker<GameClass_2>);
m_factory.Register(new FactoryMaker<GameClass_3>);
//... etc...
```

As you can see, it takes virtually no effort to register new objects with the factory.

One technique you might come across is the automatic registration of object types with factories. Automatic registration makes it unnecessary to register object types by hand, or even to have any sort of registration step. All that happens behind the scenes for you, taking advantage of constructors and global object creation. The idea is to create a global object that deals with the registration in its constructor when it's first created. Since it's a global object, its constructor will be called during the static initialization phase, which happens even before `main()` is called.

You could create a template to make creating registration objects as simple as possible:

```
template <class Factory, class Type>
class Registrar {
    Registrar(Factory & factory) {
```

```
            factory.Register(new FactoryMaker<Type>);
        }
    };
```

Now, all you need to do to register your object types automatically is create a global `Registrar` object:

```
Registrar<ExtensibleObjectFactory<GameObject>, GameClass_1> regis-
trarGameClass_1;
```

If that's too cumbersome to type every time, just wrap it up in a macro (or combine it in a class definition macro if you already have one), so you can just type:

```
FACTORY_REGISTER(GameClass_1)
```

However, as convenient as this technique might sound, it has its share of drawbacks. One of the most annoying ones you're bound to run into sooner or later is that the compiler could very well strip out the registration code when optimizations are turned on. We're creating a global object for the sole purpose of executing its constructor, but nothing in the rest of the program references it anywhere. Consequently, many compilers with aggressive dead-code reduction will remove it from the final executable, and none of our object types will be registered.

You might be able to trick the compiler into not stripping that code, but there are other drawbacks as well. One of the biggest problems with this approach, and with anything that relies on static initialization, is that you don't have much control over exactly when it happens. You can't control the order of initialization (unless the objects are in the same file), and you can't ensure that other subsystems have been initialized before. What if we wanted to initialize the factory before we register anything? We're out of luck, or we have to check in the `Register()` function whether it's already initialized, and do it if it isn't, which is just ugly and asking for trouble. What if we need to call a different function, or a totally different subsystem?

As if that weren't enough, automatic registration doesn't allow us to customize which objects get registered at runtime. For example, if a tool is running in "artist mode," we might want to only register and make available certain object types, but if it's running in "programmer mode," we might register a different set of objects. Even in the game, we might have different object types for a mod than for the regular game.

Once we consider what we gain by automatic registration of types and all the possible drawbacks, explicit registration sounds more and more appealing. Especially if you can get it down to one line as we did in the previous example, there's very little reason not to do it that way and save ourselves many headaches down the line.

Level Instantiation

Before we can play a level, we need to load it from disk. We aren't just loading the assets, but the actual game state. Whether this is our first time playing the level, or

we're loading a previously saved state, we need to create the game entities that are in the world at this point, and set the correct state for each.

The creation part is easy now that we have an extensible object factory that creates game entities. The level file will contain a list of entities in the world, listed by name or ID. All we have to do is read through that list and repeatedly call the object factory with each of the object types, and we'll end up with the correct amount and type of game entities.

What we haven't talked about is the state of the game entities themselves. Just creating a game entity of the class `Orc` isn't enough. We need to set its correct position, orientation, amount of health, what weapons it is carrying, and so forth. Otherwise, we would have many, completely identical orcs in the world, and that's probably not what we want.

This situation is even clearer if we have more general classes like `Enemy`. It's not enough to just create an enemy; we need to load the correct textures, geometry, animation, AI state, scripts, and so forth for each type of enemy that we want to create.

One straightforward way of doing this is to store all necessary state data with each entity in the level file, so it can be restored as we create each entity. This approach will work, but it has one major drawback: similar entities will have the same data loaded repeatedly, and, even worse, they will take the extra memory to store all that information. If we have 200 orcs in our level, all 200 orc objects are going to contain most of the same data (geometry, animations, etc.), and are only going to differ in a handful of values (current health, position, and state). So, not only are our load times slower, but we're wasting memory in the game. Figure 3.6.6 shows this situation with several similar entities that have a lot of data in common.

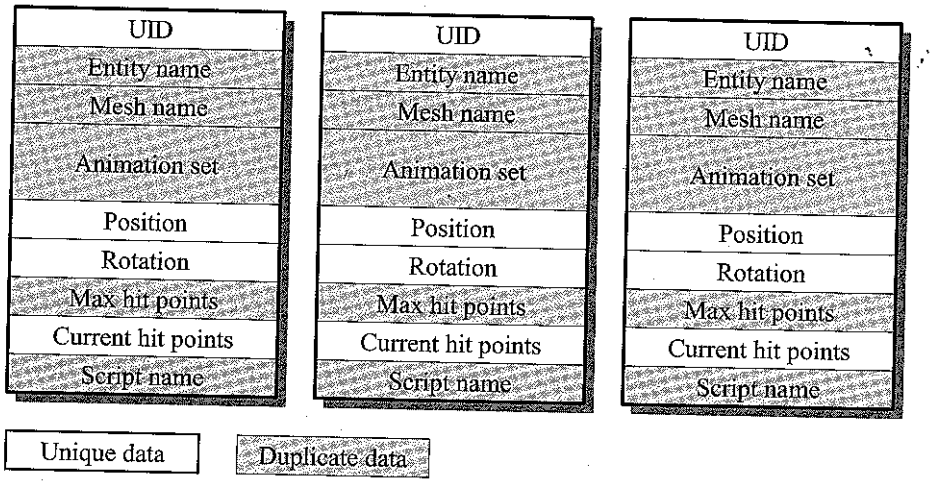

FIGURE 3.6.6 *Several similar entities with a lot of duplicate data.*

To solve this, we divide the data that is associated with each entity into instance data versus template data. Instance data are the values that differ from entity to entity: position, rotation, current AI state, current health, and so forth. Template data are the values that apply to all entities of that type: animation, textures, geometry, scripts, and so forth. Notice that we're using the word *template* in the general sense of a template to create a particular entity with some specific attributes, not in the C++ sense of code template. With this approach, we can load the template data just once, and have the entities themselves contain only the instance data, which should be much smaller. Figure 3.6.7 shows several entities sharing the same template data.

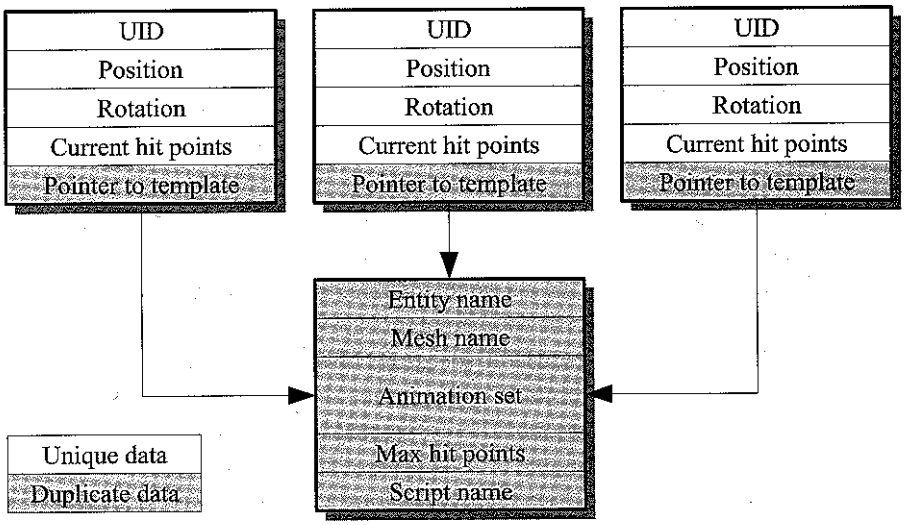

FIGURE 3.6.7 *Entities of the same template type share common data and only have per-instance data.*

This is a particularly powerful approach because it means we can have multiple entity templates that use the same C++ class, but have different sets of template data. Expose these values to your designers, and they'll be able to create a large variety of entities with minimal code changes. For example, we could have the regular "Orc" template, which loads the standard orc values. However, it would be trivial to add an "OrcChieftain" template that uses values very similar to the Orc, but has twice the health, better armor, a slightly different appearance, and perhaps a different weapon. All those values are part of the template, not of the instance, and a designer could create them just by editing a text file or through the game editor.

Identification

Before we can deliver a letter, we need to know the name and address of the person to whom we're sending it. It's the same thing with interaction between game entities. Before two entities can interact with each other, they need to know how to find each other and address each other by name.

A naive approach would be to use strings to identify game entities. We could make sure entities are created with a unique string based on their template or class name plus a timestamp or a sequential number. However, strings will take too much memory and be too slow to process at runtime unless you have a tiny level with just a handful of entities.

Another approach is to use pointers to the entities themselves. Unfortunately, pointers are error prone, and there's no way to know if the entity referred to by the pointer is still there. This is a particularly insidious problem because of how dynamic most modern games are. You can interact with many objects in the environment, move them, pick them up, and even destroy them. Things are constantly being created and destroyed, so there are no guarantees that just because you have a pointer to something that you saw a few frames ago, that object is still around now.

Consider the following example: Right after a turret fires a volley of projectiles in the air, it is destroyed by a nearby tank. A few seconds later, the projectiles land and destroy some other unit. Usually, when a projectile hits, we want to notify the entity that shot that projectile so it can gain experience, or get upgrades, or simply keep stats of hit ratios and accuracy. If the projectiles had a pointer to the turret that generated them, the game is about to crash as they try to access an invalid pointer since the turret doesn't exist anymore. Clearly, we need a better approach.

Most games use a system of unique IDs (often referred to as UIDs) or handles. This means that entities refer to each other not through pointers, but through some value that uniquely represents each entity in the world. When necessary, the game itself takes care of translating between handles and pointers. The advantage of this approach is that if somebody tries to communicate with an entity after it has been destroyed, the game will detect that the handle is no longer valid and ignore the message or even indicate that the entity is no longer around.

Many schemes allow us to map handles to game entities. The only requirement is that each handle maps uniquely to a game entity, and that the translation is as efficient as possible because it could be done hundreds or even thousands of times per frame. The most straightforward approach is to create a hash table in which the handles are the keys, and the buckets contain pointers to game entities.

As for generating unique handles, usually a globally increasing 32-bit integer will do the job nicely. Every time a new entity is created, the entity is given the next larger number. This approach will work fine for most games. However, if your game is expected to come close to generating 2^{32} entities in one run (as a persistent online world might do), then you need to use more bits, or come up with a way to reuse old, unused IDs.

Communication

We have created a bunch of entities to populate the level, we update them every frame, and they do their own thing. Even though it might be fun to wander aimlessly through the world for a bit, in order to have a full game we need some way to interact with those entities (and have them interact with each other). Communication between entities allows that interaction.

Communication between entities can be very straightforward through the use of function calls, or can be slightly more complicated by using a full messaging system.

In the simplest case, when an entity needs to communicate with another entity, it gets the pointer to the entity it wants to talk to (through the handle to pointer translation we discussed in the previous section), and calls a function directly on the receiving entity. This approach is simple and straightforward, but it has several drawbacks.

The first problem is that it might require that the sending entity know a fair amount of things about the receiving entity in order to know what functions to call. For example, if entity A attempts to pick up entity B, it first needs to find out if B can be picked up, and then it needs to know what sequence of functions to call in B to make sure its status is updated to reflect that it has been picked up. Much of this type of code leads to entities finding out what specific type of entity other entities are, casting them to their real type, and doing conditional operations on them based on their type (usually with a long `switch` statement). This leads to brittle code that is hard to maintain, with lots of potential for bugs with entity interactions.

The second problem is that if we call a function on an entity directly, the entity will deal with it right away. This is fine sometimes, but other times we want to make sure we update entities in a certain order, or that we update them all in lockstep (i.e., we first run the simulation in all of them, and only when they're all finished do we want them to deal with any messages they received this frame). To do that, we need to buffer the communication between them.

A common approach in many games is to use a messaging system. Entities, instead of calling functions directly, send messages to each other. An entity then creates a message, fills it with the information it wants to pass to another entity, and sends it to the messaging delivery system. The message is queued there until we want to deliver it, and only then do we pass it to the receiving entity (assuming it's still around).

This approach solves the problem of being able to buffer messages (since they can stay in the queue as long as we want), and minimizes the coupling between entities. Now an entity can send a message to another entity without knowing anything about it. If the other entity doesn't know how to handle that message, nothing happens. For example, entity A can go around sending "pick-up" messages to entities it encounters without knowing what they are. An enemy unit will ignore a pick-up message, but a weapon entity will react correctly and initiate a pick-up sequence.

The messages themselves can be implemented in a variety of ways. If you're planning on only having about half a dozen different messages, you might want to consider coming up with a single struct that can hold all the data you need, along with a

type ID that indicates what type of message this is. As long as you manage to keep your message size small (since it's going to have to be filled, passed around, and read), this has the advantage of making all your messages exactly the same type and exactly the same size, which means you can easily optimize how they get created and passed around.

If you're going to have many different types of messages, and modern games can easily have hundreds of different messages to model all the different ways entities and players can interact with each other, then a different approach might be beneficial. We could create an interface `Message` class that can be used to send and receive any type of message, and then implement specific messages as classes (or structs) that inherit from `Message` and add their own set of data. This allows us to have very different messages, requiring completely different sets of data, without bloating the size of the message or having to resort to reusing the same memory space with multiple variables (through unions or by casting directly). With this approach, you don't even need to have a type ID indicating what type of message it is. Instead, you can use your system's runtime type identification (either the standard C++ one or a custom-made one) to find out the type of a particular message and handle it accordingly.

Messages are being created and destroyed constantly. For every interaction between a pair of entities, there will be at least one message created and sent. It can easily be followed by a response message, or perhaps the message itself triggers a cascade of events that create more messages. In any case, we should be ready for potentially hundreds or even thousands of messages per frame.

What does all this mean to us? We should be very careful about how we pass these messages. The messages themselves usually aren't very large, but they aren't a plain 32-bit number either. We should avoid copying them as much as possible, and instead prefer to create them once and then pass pointers (or references) to them during their lifetime.

We also need to be careful about how we allocate these messages, especially in game consoles. If we simply did a new/delete every time we wanted to create or destroy a message, performance would be suboptimal (`new` is hardly a fast operation that you want to do thousands of time per frame), and it can potentially fragment memory in no time. To solve this, we should take care to allocate messages from a set of memory pools, which avoids fragmentation and makes allocation faster. By overriding operators `new` and `delete`, we make it so the fact that we're using memory pools is totally transparent to the rest of the program, which keeps the code cleaner and allows us to change how we implement it. However, if we take the approach of using several different message classes, message objects are going to be of all different sizes, which makes using a memory pool a bit less straightforward. See Chapter 3.7 for more details on memory pools.

One word of caution: just because we're passing messages between entities, it doesn't mean that we can pass those messages across the network and implement an online game that way. Usually, we only want to pass a subset of those messages across

the network, and they often need to contain different data (using timestamps, special network IDs, etc.) and be treated in different ways (they might need to be compressed, or their results might need to be interpolated or ignored). For these reasons, it is usually better to have a completely different set of messages for network communication.

Summary

As projects grow in size and complexity, carefully considering the architecture of a game code base is becoming increasingly important, especially if you plan to reuse some of the code in future projects.

At the highest level, games are usually a set of initialization/shutdown steps and one or more game loops. Each initialization step takes care of setting up any resources or systems needed by the game, and the corresponding shutdown step cleans up anything done by the initialization step. The game loop is executed once per frame, and it does all the tasks that have to be done each frame to make the game respond to the player: input gathering, simulation, collision, rendering, and so forth.

Every game has some form of game entity. These are self-contained units of gameplay logic. They can be enemy units, animated scenery, the player avatar, or even just a trigger. Creating, managing, and updating these entities efficiently is very important for the smooth functioning of the game.

Exercises

1. Why exactly does the shutdown phase need to mirror the order in which items were initialized? Write a program with an initialization and a shutdown phase that demonstrates why using a different shutdown order can cause problems.
2. Most game engines will have a memory manager that, among many other things, will report memory leaks. However, many C++ libraries provide special functions to display memory leaks. Learn what functions are available in your platform and write a quick program with memory leaks to demonstrate all the information you can gather about them. Are you able to display the number of memory leaks? Their address? What part of the code allocated them? Their content?
3. Implement a simple main game loop that calls each of the tasks described in this chapter. Each task function simply prints out its name to the screen and returns. Now, modify the game loop to decouple simulation and rendering. Run it again and compare the outputs.
4. Reimplement the game loop from the previous exercise by using tasks that are registered in a generic game loop. Set up two sets of tasks: one set for the front end, and one set for the main game loop. Add the ability to toggle between the two sets of tasks by pressing a key.

5. Choose a game you have played recently. Create a list with all the possible game entities you see and interact with in the first few minutes of play. Remember to look out for entities without physical representations such as triggers or timers.

References

[Alexandrescu01] Alexandrescu, Andrei, *Modern* C++ *Design*, Addison-Wesley, 2001.

[Bilas02] Bilas, Scott, "A Data-Driven Game Object System," *Game Developers Conference 2002,* available online at *www.drizzle.com/~scottb/gdc/game-objects.htm.*

[Duran03] Duran, Alex, "Building Object Systems: Features, Tradeoffs, and Pitfalls," *Game Developers Conference 2003*, available online at *www.gdconf.com/archives/2003/Duran_Alex.ppt.*

[Lakos96] Lakos, John, *Large-Scale* C++ *Software Design*, Addison-Wesley, 1996.

[Laramée01] Laramée, François, "A Game Entity Factory," *Game Programming Gems 2*, Charles River Media, 2001.

[Llopis04] Llopis, Noel, "The Clock: Keeping Your Fingers on the Pulse of the Game," *Game Programming Gems 4*, Charles River Media, 2004.

[Rabin00] Rabin, Steve, "The Magic of Data-Driven Design," *Game Programming Gems*, Charles River Media, 2000.

[Ranck00] Ranck, Steven, "Frame-Based Memory Allocation," *Game Programming Gems*, Charles River Media, 2000.

[Rollings03] Rollings, Andrew, and Morris, Dave, *Game Architecture and Design: A New Edition*, New Riders, 2003.

[Sutter00] Sutter, Herb, *Exceptional C++*, Addison-Wesley, 2000.