Some asynchronous I/O libraries allow the programmer to ask for an estimate of how long a particular asynchronous operation will take to complete. Some APIs also allow you to set *deadlines* on a request (which effectively prioritizes the request relative to other pending requests), and to specify what happens when a request misses its deadline (e.g., cancel the request, notify the program and keep trying, etc.)

6.1.3.1 Priorities

It's important to remember that file I/O is a real-time system, subject to deadlines just like the rest of the game. Therefore, asynchronous I/O operations often have varying priorities. For example, if we are streaming audio from the hard disk or Blu-ray and playing it on the fly, loading the next buffer-full of audio data is clearly higher priority than, say, loading a texture or a chunk of a game level. Asynchronous I/O systems must be capable of suspending lower-priority requests, so that higher-priority I/O requests have a chance to complete within their deadlines.

6.1.3.2 How Asynchronous File I/O Works

Asynchronous file I/O works by handling I/O requests in a separate thread. The main thread calls functions that simply place requests on a queue and then return immediately. Meanwhile, the I/O thread picks up requests from the queue and handles them sequentially using blocking I/O routines like `read()` or `fread()`. When a request is completed, a callback provided by the main thread is called, thereby notifying it that the operation is done. If the main thread chooses to wait for an I/O request to complete, this is handled via a *semaphore*. (Each request has an associated semaphore, and the main thread can put itself to sleep waiting for that semaphore to be signaled by the I/O thread upon completion of the request.)

Virtually *any* synchronous operation you can imagine can be transformed into an asynchronous operation by moving the code into a separate thread—or by running it on a physically separate processor, such as on one of the CPU cores on the PlayStation 4. See Section 7.6 for more details.

6.2 The Resource Manager

Every game is constructed from a wide variety of *resources* (sometimes called *assets* or *media*). Examples include meshes, materials, textures, shader programs, animations, audio clips, level layouts, collision primitives, physics parameters, and the list goes on. A game's resources must be managed, both in

terms of the offline tools used to create them, and in terms of loading, unloading and manipulating them at runtime. Therefore, every game engine has a *resource manager* of some kind.

Every resource manager is comprised of two distinct but integrated components. One component manages the chain of offline tools used to create the assets and transform them into their engine-ready form. The other component manages the resources at runtime, ensuring that they are loaded into memory in advance of being needed by the game and making sure they are unloaded from memory when no longer needed.

In some engines, the resource manager is a cleanly designed, unified, centralized subsystem that manages all types of resources used by the game. In other engines, the resource manager doesn't exist as a single subsystem per se, but rather is spread across a disparate collection of subsystems, perhaps written by different individuals at various times over the engine's long and sometimes colorful history. But no matter how it is implemented, a resource manager invariably takes on certain responsibilities and solves a well-understood set of problems. In this section, we'll explore the functionality and some of the implementation details of a typical game engine resource manager.

6.2.1 Offline Resource Management and the Tool Chain

6.2.1.1 Revision Control for Assets

On a small game project, the game's assets can be managed by keeping loose files sitting around on a shared network drive with an ad hoc directory structure. This approach is not feasible for a modern commercial 3D game, comprised of a massive number and variety of assets. For such a project, the team requires a more formalized way to track and manage its assets.

Some game teams use a source code revision control system to manage their resources. Art source files (Maya scenes, Photoshop PSD files, Illustrator files, etc.) are checked in to Perforce or a similar package by the artists. This approach works reasonably well, although some game teams build custom asset management tools to help flatten the learning curve for their artists. Such tools may be simple wrappers around a commercial revision control system, or they might be entirely custom.

Dealing with Data Size

One of the biggest problems in the revision control of art assets is the sheer amount of data. Whereas C++ and script source code files are small, relative to their impact on the project, art files tend to be much, much larger. Because

many source control systems work by copying files from the central repository down to the user's local machine, the sheer size of the asset files can render these packages almost entirely useless.

I've seen a number of different solutions to this problem employed at various studios. Some studios turn to commercial revision control systems like Alienbrain that have been specifically designed to handle very large data sizes. Some teams simply "take their lumps" and allow their revision control tool to copy assets locally. This can work, as long as your disks are big enough and your network bandwidth sufficient, but it can also be inefficient and slow the team down. Some teams build elaborate systems on top of their revision control tool to ensure that a particular end user only gets local copies of the files he or she actually needs. In this model, the user either has no access to the rest of the repository or can access it on a shared network drive when needed.

At Naughty Dog we use a proprietary tool that makes use of UNIX symbolic links to virtually eliminate data copying, while permitting each user to have a complete local view of the asset repository. As long as a file is not checked out for editing, it is a symlink to a master file on a shared network drive. A symbolic link occupies very little space on the local disk, because it is nothing more than a directory entry. When the user checks out a file for editing, the symlink is removed, and a local copy of the file replaces it. When the user is done editing and checks the file in, the local copy becomes the new master copy, its revision history is updated in a master database, and the local file turns back into a symlink. This system works very well, but it requires the team to build their own revision control system from scratch; I am unaware of any commercial tool that works like this. Also, symbolic links are a UNIX feature—such a tool could probably be built with Windows junctions (the Windows equivalent of a symbolic link), but I haven't seen anyone try it as yet.

6.2.1.2 The Resource Database

As we'll explore in depth in the next section, most assets are not used in their original format by the game engine. They need to pass through some kind of asset conditioning pipeline, whose job it is to convert the assets into the binary format needed by the engine. For every resource that passes through the asset conditioning pipeline, there is some amount of *metadata* that describes *how* that resource should be processed. When compressing a texture bitmap, we need to know what *type* of compression best suits that particular image. When exporting an animation, we need to know what range of frames in Maya should be exported. When exporting character meshes out of

a Maya scene containing multiple characters, we need to know which mesh corresponds to which character in the game.

To manage all of this metadata, we need some kind of database. If we are making a very small game, this database might be housed in the brains of the developers themselves. I can hear them now: "Remember: the player's animations need to have the 'flip X' flag set, but the other characters must *not* have it set... or... rats... is it the other way around?"

Clearly for any game of respectable size, we simply cannot rely on the memories of our developers in this manner. For one thing, the sheer volume of assets becomes overwhelming quite quickly. Processing individual resource files by hand is also far too time-consuming to be practical on a full-fledged commercial game production. Therefore, every professional game team has some kind of semiautomated resource pipeline, and the data that drive the pipeline is stored in some kind of *resource database*.

The resource database takes on vastly different forms in different game engines. In one engine, the metadata describing how a resource should be built might be embedded into the source assets themselves (e.g., it might be stored as so-called blind data within a Maya file). In another engine, each source resource file might be accompanied by a small text file that describes how it should be processed. Still other engines encode their resource building metadata in a set of XML files, perhaps wrapped in some kind of custom graphical user interface. Some engines employ a true relational database, such as Microsoft Access, MySQL or conceivably even a heavyweight database like Oracle.

Whatever its form, a resource database must provide the following basic functionality:

- The ability to deal with multiple *types* of resources, ideally (but certainly not necessarily) in a somewhat consistent manner.
- The ability to create new resources.
- The ability to delete resources.
- The ability to inspect and modify existing resources.
- The ability to move a resource's source file(s) from one location to another on-disk. (This is very helpful because artists and game designers often need to rearrange assets to reflect changing project goals, rethinking of game designs, feature additions and cuts, etc.)
- The ability of a resource to cross-reference other resources (e.g., the material used by a mesh, or the collection of animations needed by level 17). These cross-references typically drive both the resource building process and the loading process at runtime.

- The ability to maintain *referential integrity* of all cross-references within the database and to do so in the face of all common operations such as deleting or moving resources around.
- The ability to maintain a revision history, complete with a log of who made each change and why.
- It is also very helpful if the resource database supports searching or querying in various ways. For example, a developer might want to know in which levels a particular animation is used or which textures are referenced by a set of materials. Or they might simply be trying to find a resource whose name momentarily escapes them.

It should be pretty obvious from looking at the above list that creating a reliable and robust resource database is no small task. When designed well and implemented properly, the resource database can quite literally make the difference between a team that ships a hit game and a team that spins its wheels for 18 months before being forced by management to abandon the project (or worse). I know this to be true, because I've personally experienced both.

6.2.1.3 Some Successful Resource Database Designs

Every game team will have different requirements and make different decisions when designing their resource database. However, for what it's worth, here are some designs that have worked well in my own experience.

Unreal Engine 4

Unreal's resource database is managed by their über-tool, UnrealEd. UnrealEd is responsible for literally everything, from resource metadata management to asset creation to level layout and more. UnrealEd has its drawbacks, but its single biggest benefit is that UnrealEd is a part of the game engine itself. This permits assets to be created and then immediately viewed in their full glory, exactly as they will appear in-game. The game can even be run from within UnrealEd in order to visualize the assets in their natural surroundings and see if and how they work in-game.

Another big benefit of UnrealEd is what I would call *one-stop shopping*. UnrealEd's Generic Browser (depicted in Figure 6.1) allows a developer to access literally every resource that is consumed by the engine. Having a single, unified and reasonably consistent interface for creating and managing all types of resources is a big win. This is especially true considering that the resource data in most other game engines is fragmented across countless inconsistent and often cryptic tools. Just being able to *find* any resource easily in UnrealEd is a big plus.

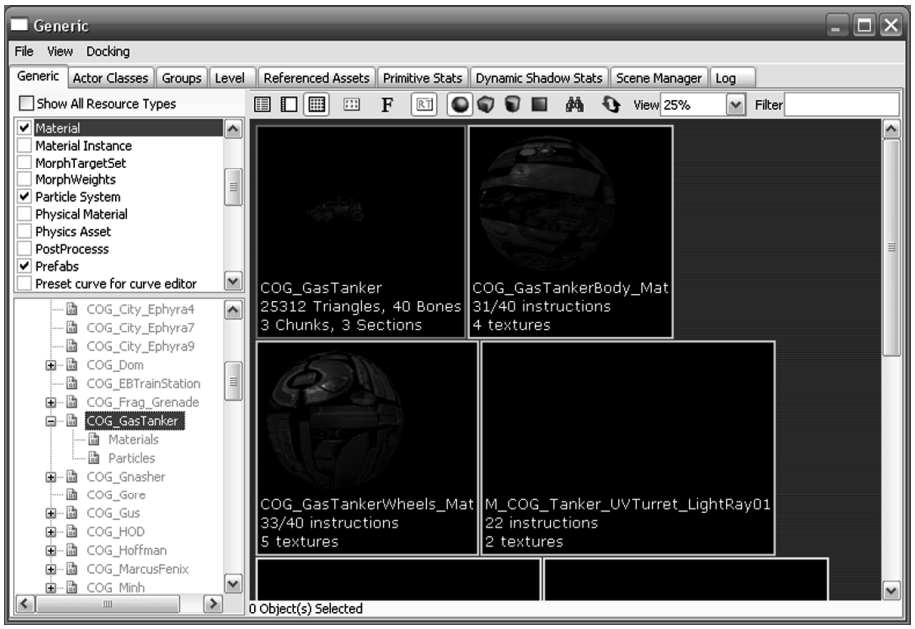

Figure 6.1. UnrealEd's Generic Browser.

Unreal can be less error-prone than many other engines, because assets must be explicitly imported into Unreal's resource database. This allows resources to be checked for validity very early in the production process. In most game engines, any old data can be thrown into the resource database, and you only know whether or not that data is valid when it is eventually built—or sometimes not until it is actually loaded into the game at runtime. But with Unreal, assets can be validated as soon as they are imported into UnrealEd. This means that the person who created the asset gets immediate feedback as to whether his or her asset is configured properly.

Of course, Unreal's approach has some serious drawbacks. For one thing, all resource data is stored in a small number of large package files. These files are binary, so they are not easily merged by a revision control package like CVS, Subversion or Perforce. This presents some major problems when more than one user wants to modify resources that reside in a single package. Even if the users are trying to modify *different* resources, only one user can lock the package at a time, so the other has to wait. The severity of this problem can be reduced by dividing resources into relatively small, granular packages, but it cannot practically be eliminated.

Referential integrity is quite good in UnrealEd, but there are still some problems. When a resource is renamed or moved around, all references to it are maintained automatically using a dummy object that remaps the old resource to its new name/location. The problem with these dummy remapping objects is that they hang around and accumulate and sometimes cause problems, especially if a resource is deleted. Overall, Unreal's referential integrity is quite good, but it is not perfect.

Despite its problems, UnrealEd is by far the most user-friendly, well-integrated and streamlined asset creation toolkit, resource database and asset conditioning pipeline that I have ever worked with.

Naughty Dog's Uncharted / The Last of Us Engine

For *Uncharted: Drake's Fortune*, Naughty Dog stored its resource metadata in a MySQL database. A custom graphical user interface was written to manage the contents of the database. This tool allowed artists, game designers and programmers alike to create new resources, delete existing resources and inspect and modify resources as well. This GUI was a crucial component of the system, because it allowed users to avoid having to learn the intricacies of interacting with a relational database via SQL.

The original MySQL database used on *Uncharted* did not provide a useful history of the changes made to the database, nor did it provide a good way to roll back "bad" changes. It also did not support multiple users editing the same resource, and it was difficult to administer. Naughty Dog has since moved away from MySQL in favor of an XML file-based asset database, managed under Perforce.

Builder, Naughty Dog's resource database GUI, is depicted in Figure 6.2. The window is broken into two main sections: a tree view showing all resources in the game on the left and a properties window on the right, allowing the resource(s) that are selected in the tree view to be viewed and edited. The resource tree contains folders for organizational purposes, so that the artists and game designers can organize their resources in any way they see fit. Various types of resources can be created and managed within any folder, including actors and levels, and the various subresources that comprise them (primarily meshes, skeletons and animations). Animations can also be grouped into pseudo-folders known as bundles. This allows large groups of animations to be created and then managed as a unit, and prevents a lot of wasted time dragging individual animations around in the tree view.

The asset conditioning pipeline employed on *Uncharted* and *The Last of Us* consists of a set of resource exporters, compilers and linkers that are run from the command line. The engine is capable of dealing with a wide variety of

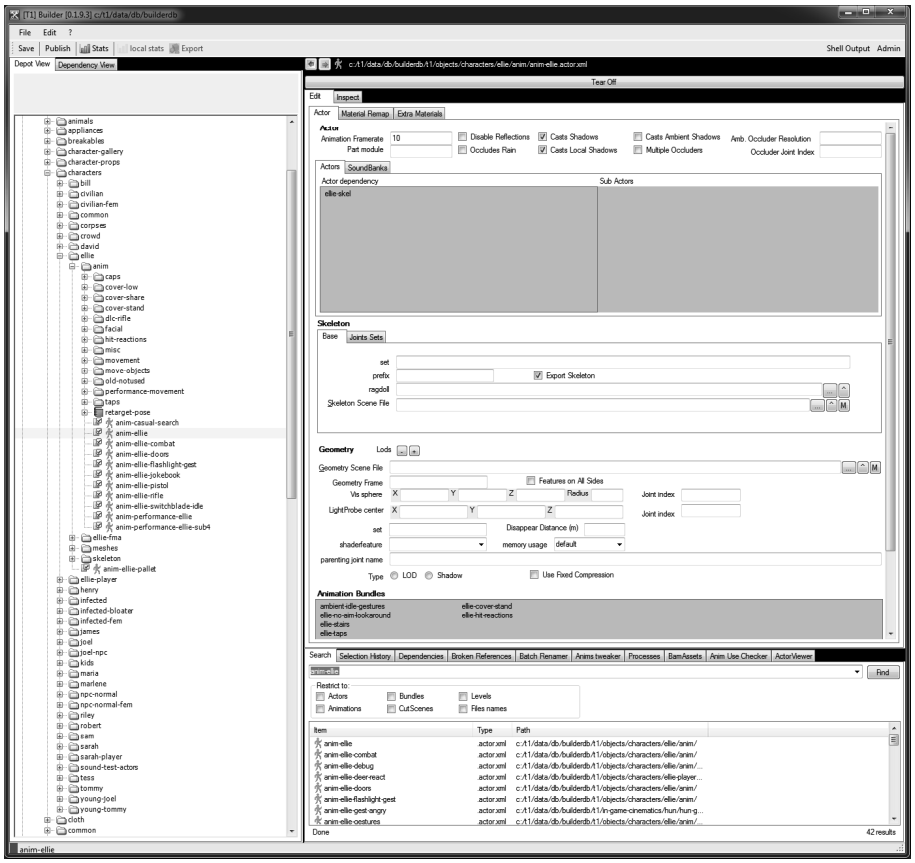

Figure 6.2. The front-end GUI for Naughty Dog's offline resource database, Builder.

different kinds of data objects, but these are packaged into one of two types of resource files: actors and levels. An actor can contain skeletons, meshes, materials, textures and/or animations. A level contains static background meshes, materials and textures, and also level-layout information. To build an actor, one simply types `ba` *name-of-actor* on the command line; to build a level, one types `bl` *name-of-level*. These command-line tools query the database to determine exactly *how* to build the actor or level in question. This includes information on how to export the assets from DCC tools like Maya and Photoshop, how to process the data, and how to package it into binary `.pak` files that can be loaded by the game engine. This is much simpler than in many engines, where resources have to be *exported manually* by the artists—a time-consuming, tedious and error-prone task.

The benefits of the resource pipeline design used by Naughty Dog include:

- *Granular resources.* Resources can be manipulated in terms of logical entities in the game—meshes, materials, skeletons and animations. These resource types are granular enough that the team almost never has conflicts in which two users want to edit the same resource simultaneously.
- *The necessary features (and no more).* The Builder tool provides a powerful set of features that meet the needs of the team, but Naughty Dog didn't waste any resources creating features they didn't need.
- *Obvious mapping to source files.* A user can very quickly determine which source assets (native DCC files, like Maya .ma files or photoshop .psd files) make up a particular resource.
- *Easy to change how DCC data is exported and processed*. Just click on the resource in question and twiddle its processing properties within the resource database GUI.
- *Easy to build assets*. Just type `ba` or `bl` followed by the resource name on the command line. The dependency system takes care of the rest.

Of course, Naughty Dog's tool chain does have some drawbacks as well, including:

- *Lack of visualization tools.* The only way to preview an asset is to load it into the game or the model/animation viewer (which is really just a special mode of the game itself).
- *The tools aren't fully integrated.* Naughty Dog uses one tool to lay out levels, another to manage the majority of resources in the resource database, and a third to set up materials and shaders (this is not part of the resource database front end). Building the assets is done on the command line. It might be a bit more convenient if all of these functions were to be integrated into a single tool. However, Naughty Dog has no plans to do this, because the benefit would probably not outweigh the costs involved.

OGRE's Resource Manager System

OGRE is a rendering engine, not a full-fledged game engine. That said, OGRE does boast a reasonably complete and very well-designed runtime resource manager. A simple, consistent interface is used to load virtually any kind of resource. And the system has been designed with extensibility in mind. Any programmer can quite easily implement a resource manager for a brand new kind of asset and integrate it easily into OGRE's resource framework.

One of the drawbacks of OGRE's resource manager is that it is a runtime-only solution. OGRE lacks any kind of offline resource database. OGRE does provide some exporters that are capable of converting a Maya file into a mesh that can be used by OGRE (complete with materials, shaders, a skeleton and optional animations). However, the exporter must be run manually from within Maya itself. Worse, all of the metadata describing how a particular Maya file should be exported and processed must be entered by the user doing the export.

In summary, OGRE's runtime resource manager is powerful and well-designed. But, OGRE would benefit a great deal from an equally powerful and modern resource database and asset conditioning pipeline on the tools side.

Microsoft's XNA

XNA is a game development toolkit by Microsoft, targeted at the PC and Xbox 360 platforms. XNA's resource management system is unique, in that it leverages the project management and build systems of the Visual Studio IDE to manage and build the assets in the game as well. XNA's game development tool, Game Studio Express, is just a plug-in to Visual Studio Express. You can read more about Game Studio Express at http://msdn.microsoft.com/en-us/library/bb203894.aspx.

6.2.1.4 The Asset Conditioning Pipeline

In Section 1.7, we learned that resource data is typically created using advanced digital content creation (DCC) tools like Maya, ZBrush, Photoshop or Houdini. However, the data formats used by these tools are usually not suitable for direct consumption by a game engine. So the majority of resource data is passed through an *asset conditioning pipeline* (ACP) on its way to the game engine. The ACP is sometimes referred to as the *resource conditioning pipeline* (RCP), or simply the *tool chain*.

Every resource pipeline starts with a collection of *source assets* in native DCC formats (e.g., Maya .ma or .mb files, Photoshop .psd files, etc.). These assets are typically passed through three processing stages on their way to the game engine:

1. *Exporters.* We need some way of getting the data out of the DCC's native format and into a format that we can manipulate. This is usually accomplished by writing a custom plug-in for the DCC in question. It is the plug-in's job to export the data into some kind of intermediate file format that can be passed to later stages in the pipeline. Most DCC applications provide a reasonably convenient mechanism for doing this.

Maya actually provides three: a C++ SDK, a scripting language called MEL and most recently a Python interface as well.

In cases where a DCC application provides no customization hooks, we can always save the data in one of the DCC tool's native formats. With any luck, one of these will be an open format, a reasonably intuitive text format, or some other format that we can reverse engineer. Presuming this is the case, we can pass the file directly to the next stage of the pipeline.

2. *Resource compilers.* We often have to "massage" the raw data exported from a DCC application in various ways in order to make them game-ready. For example, we might need to rearrange a mesh's triangles into strips, or compress a texture bitmap, or calculate the arc lengths of the segments of a Catmull-Rom spline. Not all types of resources need to be compiled—some might be game-ready immediately upon being exported.

3. *Resource linkers.* Multiple resource files sometimes need to be combined into a single useful package prior to being loaded by the game engine. This mimics the process of linking together the object files of a compiled C++ program into an executable file, and so this process is sometimes called *resource linking*. For example, when building a complex composite resource like a 3D model, we might need to combine the data from multiple exported mesh files, multiple material files, a skeleton file and multiple animation files into a single resource. Not all types of resources need to be linked—some assets are game-ready after the export or compile steps.

Resource Dependencies and Build Rules

Much like compiling the source files in a C or C++ project and then linking them into an executable, the asset conditioning pipeline processes source assets (in the form of Maya geometry and animation files, Photoshop PSD files, raw audio clips, text files, etc.), converts them into game-ready form and then links them together into a cohesive whole for use by the engine. And just like the source files in a computer program, game assets often have interdependencies. (For example, a mesh refers to one or more materials, which in turn refer to various textures.) These interdependencies typically have an impact on the order in which assets must be processed by the pipeline. (For example, we might need to build a character's skeleton before we can process any of that character's animations.) In addition, the dependencies between assets tell us which assets need to be rebuilt when a particular source asset changes.

Build dependencies revolve not only around changes to the assets themselves, but also around changes to data formats. If the format of the files used to store triangle meshes changes, for instance, all meshes in the entire game may need to be reexported and/or rebuilt. Some game engines employ data formats that are robust to version changes. For example, an asset may contain a version number, and the game engine may include code that "knows" how to load and make use of legacy assets. The downside of such a policy is that asset files and engine code tend to become bulky. When data format changes are relatively rare, it may be better to just bite the bullet and reprocess all the files when format changes do occur.

Every asset conditioning pipeline requires a set of rules that describe the interdependencies between the assets, and some kind of build tool that can use this information to ensure that the proper assets are built, in the proper order, when a source asset is modified. Some game teams roll their own build system. Others use an established tool, such as `make`. Whatever solution is selected, teams should treat their build dependency system with utmost care. If you don't, changes to sources assets may not trigger the proper assets to be rebuilt. The result can be inconsistent game assets, which may lead to visual anomalies or even engine crashes. In my personal experience, I've witnessed countness hours wasted in tracking down problems that could have been avoided had the asset interdependencies been properly specified and the build system implemented to use them reliably.

6.2.2 Runtime Resource Management

Let us turn our attention now to how the assets in our resource database are loaded, managed and unloaded within the engine at runtime.

6.2.2.1 Responsibilities of the Runtime Resource Manager

A game engine's runtime resource manager takes on a wide range of responsibilities, all related to its primary mandate of loading resources into memory:

- Ensures that only *one copy* of each unique resource exists in memory at any given time.
- Manages the *lifetime* of each resource.
- *Loads* needed resources and *unloads* resources that are no longer needed.
- Handles loading of *composite resources*. A composite resource is a resource comprised of other resources. For example, a *3D model* is a composite resource that consists of a mesh, one or more materials, one or more textures and optionally a skeleton and multiple skeletal animations.