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.

- Maintains *referential integrity*. This includes *internal* referential integrity (cross-references within a single resource) and *external* referential integrity (cross-references between resources). For example, a model refers to its mesh and skeleton; a mesh refers to its materials, which in turn refer to texture resources; animations refer to a skeleton, which ultimately ties them to one or more models. When loading a composite resource, the resource manager must ensure that all necessary subresources are loaded, and it must patch in all of the cross-references properly.
- Manages the *memory usage* of loaded resources and ensures that resources are stored in the appropriate place(s) in memory.
- Permits *custom processing* to be performed on a resource after it has been loaded, on a per-resource-type basis. This process is sometimes known as *logging in* or *load-initializing* the resource.
- Usually (but not always) provides a single *unified interface* through which a wide variety of resource types can be managed. Ideally a resource manager is also easily extensible, so that it can handle new types of resources as they are needed by the game development team.
- Handles *streaming* (i.e., asynchronous resource loading), if the engine supports this feature.

6.2.2.2 Resource File and Directory Organization

In some game engines (typically PC engines), each individual resource is managed in a separate "loose" file on-disk. These files are typically contained within a tree of directories whose internal organization is designed primarily for the convenience of the people creating the assets; the engine typically doesn't care where resource files are located within the resource tree. Here's a typical resource directory tree for a hypothetical game called *Space Evaders:*

```
SpaceEvaders      Root directory for entire game.
  Resources       Root of all resources.
    NPC           Non-player character models and animations.
      Pirate      Models and animations for pirates.
      Marine      Models and animations for marines.
      ...

    Player        Player character models and animations.
```

`Weapons`	Models and animations for weapons.
`Pistol`	Models and animations for the pistol.
`Rifle`	Models and animations for the rifle.
`BFG`	Models and animations for the big... uh... gun.
`...`	
`Levels`	Background geometry and level layouts.
`Level1`	First level's resources.
`Level2`	Second level's resources.
`...`	
`Objects`	Miscellaneous 3D objects.
`Crate`	The ubiquitous breakable crate.
`Barrel`	The ubiquitous exploding barrel.

Other engines package multiple resources together in a single file, such as a ZIP archive, or some other composite file (perhaps of a proprietary format). The primary benefit of this approach is improved load times. When loading data from files, the three biggest costs are *seek times* (i.e., moving the read head to the correct place on the physical media), the time required to open each individual file, and the time to read the data from the file into memory. Of these, the seek times and file-open times can be nontrivial on many operating systems. When a single large file is used, all of these costs are minimized. A single file can be organized sequentially on the disk, reducing seek times to a minimum. And with only one file to open, the cost of opening individual resource files is eliminated.

Solid-state drives (SSD) do not suffer from the seek time problems that plague spinning media like DVDs, Blu-ray discs and hard disc drives (HDD). However, no game console to date includes a solid-state drive as the primary fixed storage device (not even the PS4 and Xbox One). So designing your game's I/O patterns in order to minimize seek times is likely to be a necessity for some time to come.

The OGRE rendering engine's resource manager permits resources to exist as loose files on disk, or as virtual files within a large ZIP archive. The primary benefits of the ZIP format are the following:

1. *It is an open format.* The `zlib` and `zziplib` libraries used to read and write ZIP archives are freely available. The zlib SDK is totally free (see http://www.zlib.net), while the zziplib SDK falls under the Lesser Gnu Public License (LGPL) (see http://zziplib.sourceforge.net).

2. *The virtual files within a ZIP archive "remember" their relative paths.* This means that a ZIP archive "looks like" a raw file system for most in-

tents and purposes. The OGRE resource manager identifies all resources uniquely via strings that appear to be file system paths. However, these paths sometimes identify virtual files within a ZIP archive instead of loose files on disk, and a game programmer needn't be aware of the difference in most situations.

3. *ZIP archives may be compressed.* This reduces the amount of disk space occupied by resources. But, more importantly, it again speeds up load times, as less data need be loaded into memory from the fixed disk. This is especially helpful when reading data from a DVD-ROM or Blu-ray disk, as the data transfer rates of these devices are much slower than a hard disk drive. Hence the cost of decompressing the data after it has been loaded into memory is often more than offset by the time saved in loading less data from the device.

4. *ZIP archives are modular.* Resources can be grouped together into a ZIP file and managed as a unit. One particularly elegant application of this idea is in product localization. All of the assets that need to be localized (such as audio clips containing dialogue and textures that contain words or region-specific symbols) can be placed in a single ZIP file, and then different versions of this ZIP file can be generated, one for each language or region. To run the game for a particular region, the engine simply loads the corresponding version of the ZIP archive.

Unreal Engine 3 takes a similar approach, with a few important differences. In Unreal, all resources must be contained within large composite files known as *packages* (a.k.a. "pak files"). No loose disk files are permitted. The format of a package file is proprietary. The Unreal Engine's game editor, UnrealEd, allows developers to create and manage packages and the resources they contain.

6.2.2.3 Resource File Formats

Each type of resource file potentially has a different format. For example, a mesh file is always stored in a different format than that of a texture bitmap. Some kinds of assets are stored in standardized, open formats. For example, textures are typically stored as Targa files (TGA), Portable Network Graphics files (PNG), Tagged Image File Format files (TIFF), Joint Photographic Experts Group files (JPEG) or Windows Bitmap files (BMP)—or in a standardized compressed format such as DirectX's S3 Texture Compression family of formats (S3TC, also known as DXT*n* or DXTC). Likewise, 3D mesh data is often exported out of a modeling tool like Maya or Lightwave into a standardized format such as OBJ or COLLADA for consumption by the game engine.

Sometimes a single file format can be used to house many different types of assets. For example, the *Granny* SDK by Rad Game Tools (http://www.radgametools.com) implements a flexible open file format that can be used to store 3D mesh data, skeletal hierarchies and skeletal animation data. (In fact the Granny file format can be easily repurposed to store virtually any kind of data imaginable.)

Many game engine programmers roll their own file formats for various reasons. This might be necessary if no standardized format provides all of the information needed by the engine. Also, many game engines endeavor to do as much offline processing as possible in order to minimize the amount of time needed to load and process resource data at runtime. If the data needs to conform to a particular layout in memory, for example, a raw binary format might be chosen so that the data can be laid out by an offline tool (rather than attempting to format it at runtime after the resource has been loaded).

6.2.2.4 Resource GUIDs

Every resource in a game must have some kind of *globally unique identifier* (GUID). The most common choice of GUID is the resource's file system path (stored either as a string or a 32-bit hash). This kind of GUID is intuitive, because it clearly maps each resource to a physical file on-disk. And it's guaranteed to be unique across the entire game, because the operating system already guarantees that no two files will have the same path.

However, a file system path is by no means the only choice for a resource GUID. Some engines use a less-intuitive type of GUID, such as a 128-bit hash code, perhaps assigned by a tool that guarantees uniqueness. In other engines, using a file system path as a resource identifier is infeasible. For example, Unreal Engine 3 stores many resources in a single large file known as a *package*, so the path to the package file does not uniquely identify any one resource. To overcome this problem, an Unreal package file is organized into a folder hierarchy containing individual resources. Unreal gives each individual resource within a package a unique name, which looks much like a file system path. So in Unreal, a resource GUID is formed by concatenating the (unique) name of the package file with the in-package path of the resource in question. For example, the *Gears of War* resource GUID `Locust_Boomer.PhysicalMaterials.LocustBoomerLeather` identifies a material called `LocustBoomerLeather` within the `PhysicalMaterials` folder of the `Locust_Boomer` package file.

6.2.2.5 The Resource Registry

In order to ensure that only one copy of each unique resource is loaded into memory at any given time, most resource managers maintain some kind of

registry of loaded resources. The simplest implementation is a *dictionary*—i.e., a collection of *key-value pairs*. The keys contain the unique ids of the resources, while the values are typically pointers to the resources in memory.

Whenever a resource is loaded into memory, an entry for it is added to the resource registry dictionary, using its GUID as the key. Whenever a resource is unloaded, its registry entry is removed. When a resource is requested by the game, the resource manager looks up the resource by its GUID within the resource registry. If the resource can be found, a pointer to it is simply returned. If the resource cannot be found, it can either be loaded automatically or a failure code can be returned.

At first blush, it might seem most intuitive to automatically load a requested resource if it cannot be found in the resource registry. And in fact, some game engines do this. However, there are some serious problems with this approach. Loading a resource is a slow operation, because it involves locating and opening a file on disk, reading a potentially large amount of data into memory (from a potentially slow device like a DVD-ROM drive), and also possibly performing post-load initialization of the resource data once it has been loaded. If the request comes during active gameplay, the time it takes to load the resource might cause a very noticeable hitch in the game's frame rate, or even a multi-second freeze. For this reason, engines tend to take one of two alternative approaches:

1. Resource loading might be disallowed completely during active gameplay. In this model, all of the resources for a game level are loaded *en masse* just prior to gameplay, usually while the player watches a loading screen or progress bar of some kind.

2. Resource loading might be done *asynchronously* (i.e., the data might be *streamed*). In this model, while the player is engaged in level X, the resources for level Y are being loaded in the background. This approach is preferable because it provides the player with a load-screen-free play experience. However, it is considerably more difficult to implement.

6.2.2.6 Resource Lifetime

The *lifetime* of a resource is defined as the time period between when it is first loaded into memory and when its memory is reclaimed for other purposes. One of the resource manager's jobs is to manage resource lifetimes—either automatically or by providing the necessary API functions to the game, so it can manage resource lifetimes manually.

Each resource has its own lifetime requirements:

- Some resources must be loaded when the game first starts up and must stay resident in memory for the entire duration of the game. That is, their lifetimes are effectively infinite. These are sometimes called *global resources* or *global assets.* Typical examples include the player character's mesh, materials, textures and core animations, textures and fonts used on the heads-up display, and the resources for all of the standard-issue weapons used throughout the game. Any resource that is visible or audible to the player throughout the entire game (and cannot be loaded on the fly when needed) should be treated as a global resource.
- Other resources have a lifetime that matches that of a particular game level. These resources must be in memory by the time the level is first seen by the player and can be dumped once the player has permanently left the level.
- Some resources might have a lifetime that is shorter than the duration of the level in which they are found. For example, the animations and audio clips that make up an *in-game cinematic* (a mini-movie that advances the story or provides the player with important information) might be loaded in advance of the player seeing the cinematic and then dumped once the cinematic has played.
- Some resources like background music, ambient sound effects or full-screen movies are streamed "live" as they play. The lifetime of this kind of resource is difficult to define, because each byte only persists in memory for a tiny fraction of a second, but the entire piece of music sounds like it lasts for a long period of time. Such assets are typically loaded in chunks of a size that matches the underlying hardware's requirements. For example, a music track might be read in 4 KiB chunks, because that might be the buffer size used by the low-level sound system. Only two chunks are ever present in memory at any given moment—the chunk that is currently playing and the chunk immediately following it that is being loaded into memory.

The question of when to load a resource is usually answered quite easily, based on knowledge of when the asset is first seen by the player. However, the question of when to unload a resource and reclaim its memory is not so easily answered. The problem is that many resources are shared across multiple levels. We don't want to unload a resource when level X is done, only to immediately reload it because level Y needs the same resource.

One solution to this problem is to reference-count the resources. Whenever a new game level needs to be loaded, the list of all resources used by that

Event	A	B	C	D	E
Initial state	0	0	0	0	0
Level X counts incremented	1	1	1	0	0
Level X loads	(1)	(1)	(1)	0	0
Level X plays	1	1	1	0	0
Level Y counts incremented	1	2	2	1	1
Level X counts decremented	0	1	1	1	1
Level X unloads, level Y loads	(0)	1	1	(1)	(1)
Level Y plays	0	1	1	1	1

Table 6.2. Resource usage as two levels load and unload.

level is traversed, and the reference count for each resource is incremented by one (but they are not loaded yet). Next, we traverse the resources of any unneeded levels and decrement their reference counts by one; any resource whose reference count drops to zero is unloaded. Finally, we run through the list of all resources whose reference count just went from zero to one and load those assets into memory.

For example, imagine that level X uses resources A, B and C, and that level Y uses resources B, C, D and E. (B and C are shared between both levels.) Table 6.2 shows the reference counts of these five resources as the player plays through levels X and Y. In this table, reference counts are shown in boldface type to indicate that the corresponding resource actually exists in memory, while a grey background indicates that the resource is not in memory. A reference count in parentheses indicates that the corresponding resource data is being loaded or unloaded.

6.2.2.7 Memory Management for Resources

Resource management is closely related to memory management, because we must inevitably decide *where* the resources should end up in memory once they have been loaded. The destination of every resource is not always the same. For one thing, certain types of resources must reside in video RAM (or, on the PlayStation 4, in a memory block that has been mapped for access via the high-speed "garlic" bus). Typical examples include textures, vertex buffers, index buffers and shader code. Most other resources can reside in main RAM, but different kinds of resources might need to reside within different address ranges. For example, a resource that is loaded and stays resident for the entire game (global resources) might be loaded into one region of memory, while resources that are loaded and unloaded frequently might go somewhere else.

The design of a game engine's memory allocation subsystem is usually closely tied to that of its resource manager. Sometimes we will design the resource manager to take best advantage of the types of memory allocators we have available, or vice versa—we may design our memory allocators to suit the needs of the resource manager.

As we saw in Section 5.2.1.4, one of the primary problems facing any resource management system is the need to avoid fragmenting memory as resources are loaded and unloaded. We'll discuss a few of the more-common solutions to this problem below.

Heap-Based Resource Allocation

One approach is to simply ignore memory fragmentation issues and use a general-purpose heap allocator to allocate your resources (like the one implemented by `malloc()` in C, or the global `new` operator in C++). This works best if your game is only intended to run on personal computers, on operating systems that support virtual memory allocation. On such a system, physical memory will become fragmented, but the operating system's ability to map noncontiguous pages of physical RAM into a contiguous virtual memory space helps to mitigate some of the effects of fragmentation.

If your game is running on a console with limited physical RAM and only a rudimentary virtual memory manager (or none whatsoever), then fragmentation will become a problem. In this case, one alternative is to defragment your memory periodically. We saw how to do this in Section 5.2.2.2.

Stack-Based Resource Allocation

A stack allocator does not suffer from fragmentation problems, because memory is allocated contiguously and freed in an order opposite to that in which it was allocated. A stack allocator can be used to load resources if the following two conditions are met:

- The game is linear and level-centric (i.e., the player watches a loading screen, then plays a level, then watches another loading screen, then plays another level).
- Each level fits into memory in its entirety.

Presuming that these requirements are satisfied, we can use a stack allocator to load resources as follows: When the game first starts up, the global resources are allocated first. The top of the stack is then marked, so that we can free back to this position later. To load a level, we simply allocate its resources on the

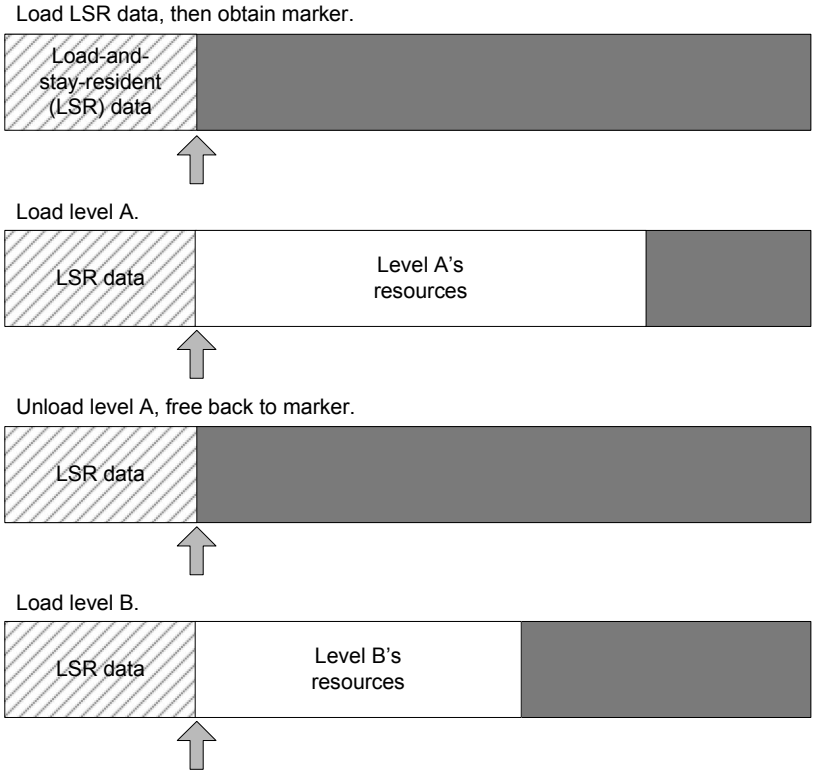

Figure 6.3. Loading resources using a stack allocator.

top of the stack. When the level is complete, we can simply set the stack top back to the marker we took earlier, thereby freeing all of the level's resources in one fell swoop without disturbing the global resources. This process can be repeated for any number of levels, without ever fragmenting memory. Figure 6.3 illustrates how this is accomplished.

A double-ended stack allocator can be used to augment this approach. Two stacks are defined within a single large memory block. One grows up from the bottom of the memory area, while the other grows down from the top. As long as the two stacks never overlap, the stacks can trade memory resources back and forth naturally—something that wouldn't be possible if each stack resided in its own fixed size block.

On *Hydro Thunder*, Midway used a double-ended stack allocator. The lower stack was used for persistent data loads, while the upper was used for temporary allocations that were freed every frame. Another way a double-

ended stack allocator can be used is to ping-pong level loads. Such an approach was used at Bionic Games, Inc. for one of their projects. The basic idea is to load a compressed version of level B into the upper stack, while the currently active level A resides (in uncompressed form) in the lower stack. To switch from level A to level B, we simply free level A's resources (by clearing the lower stack) and then decompress level B from the upper stack into the lower stack. Decompression is generally much faster than loading data from disk, so this approach effectively eliminates the load time that would otherwise be experienced by the player between levels.

Pool-Based Resource Allocation

Another resource allocation technique that is common in game engines that support streaming is to load resource data in equally sized chunks. Because the chunks are all the same size, they can be allocated using a *pool allocator* (see Section 5.2.1.2). When resources are later unloaded, the chunks can be freed without causing fragmentation.

Of course, a chunk-based allocation approach requires that all resource data be laid out in a manner that permits division into equally sized chunks. We cannot simply load an arbitrary resource file in chunks, because the file might contain a contiguous data structure like an array or a very large `struct` that is larger than a single chunk. For example, if the chunks that contain an array are not arranged sequentially in RAM, the continuity of the array will be lost, and array indexing will cease to function properly. This means that all resource data must be designed with "chunkiness" in mind. Large contiguous data structures must be avoided in favor of data structures that are either small enough to fit within a single chunk or do not require contiguous RAM to function properly (e.g., linked lists).

Each chunk in the pool is typically associated with a particular game level. (One simple way to do this is to give each level a linked list of its chunks.) This allows the engine to manage the lifetimes of each chunk appropriately, even when multiple levels with different life spans are in memory concurrently. For example, when level X is loaded, it might allocate and make use of N chunks. Later, level Y might allocate an additional M chunks. When level X is eventually unloaded, its N chunks are returned to the free pool. If level Y is still active, its M chunks need to remain in memory. By associating each chunk with a specific level, the lifetimes of the chunks can be managed easily and efficiently. This is illustrated in Figure 6.4.

One big trade-off inherent in a "chunky" resource allocation scheme is wasted space. Unless a resource file's size is an exact multiple of the chunk

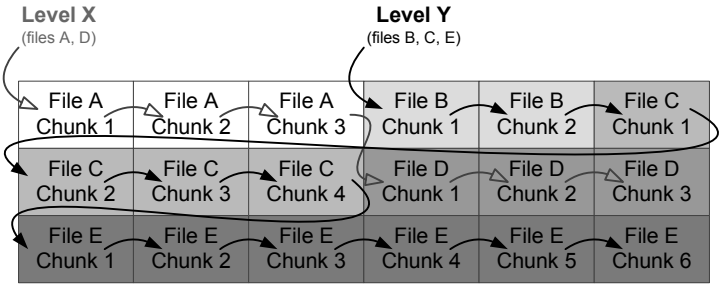

Figure 6.4. Chunky allocation of resources for levels X and Y.

size, the last chunk in a file will not be fully utilized (see Figure 6.5). Choosing a smaller chunk size can help to mitigate this problem, but the smaller the chunks, the more onerous the restrictions on the layout of the resource data. (As an extreme example, if a chunk size of one byte were selected, then no data structure could be larger than a single byte—clearly an untenable situation.) A typical chunk size is on the order of a few kibibytes. For example at Naughty Dog, we use a chunky resource allocator as part of our resource streaming system, and our chunks are 512 KiB in size on the PS3 and 1 MiB on the PS4. You may also want to consider selecting a chunk size that is a multiple of the operating system's I/O buffer size to maximize efficiency when loading individual chunks.

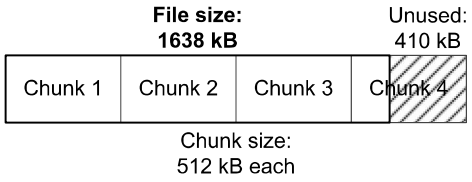

Figure 6.5. The last chunk of a resource file is often not fully utilized.

Resource Chunk Allocators

One way to limit the effects of wasted chunk memory is to set up a special memory allocator that can utilize the unused portions of chunks. As far as I'm aware, there is no standardized name for this kind of allocator, but we will call it a *resource chunk allocator* for lack of a better name.

A resource chunk allocator is not particularly difficult to implement. We need only maintain a linked list of all chunks that contain unused memory,

along with the locations and sizes of each free block. We can then allocate from these free blocks in any way we see fit. For example, we might manage the linked list of free blocks using a general-purpose heap allocator. Or we might map a small stack allocator onto each free block; whenever a request for memory comes in, we could then scan the free blocks for one whose stack has enough free RAM and then use that stack to satisfy the request.

Unfortunately, there's a rather grotesque-looking fly in our ointment here. If we allocate memory in the unused regions of our resource chunks, what happens when those chunks are freed? We cannot free part of a chunk—it's an all or nothing proposition. So any memory we allocate within an unused portion of a resource chunk will magically disappear when that resource is unloaded.

A simple solution to this problem is to only use our free-chunk allocator for memory requests whose lifetimes match the lifetime of the level with which a particular chunk is associated. In other words, we should only allocate memory out of level A's chunks for data that is associated exclusively with level A and only allocate from B's chunks memory that is used exclusively by level B. This requires our resource chunk allocator to manage each level's chunks separately. And it requires the users of the chunk allocator to specify which level they are allocating for, so that the correct linked list of free blocks can be used to satisfy the request.

Thankfully, most game engines need to allocate memory dynamically when loading resources, over and above the memory required for the resource files themselves. So a resource chunk allocator can be a fruitful way to reclaim chunk memory that would otherwise have been wasted.

Sectioned Resource Files

Another useful idea that is related to "chunky" resource files is the concept of *file sections.* A typical resource file might contain between one and four sections, each of which is divided into one or more chunks for the purposes of pool allocation as described above. One section might contain data that is destined for main RAM, while another section might contain video RAM data. Another section could contain temporary data that is needed during the loading process but is discarded once the resource has been completely loaded. Yet another section might contain debugging information. This debug data could be loaded when running the game in debug mode, but not loaded at all in the final production build of the game. The Granny SDK's file system (http://www.radgametools.com) is an excellent example of how to implement file sectioning in a simple and flexible manner.

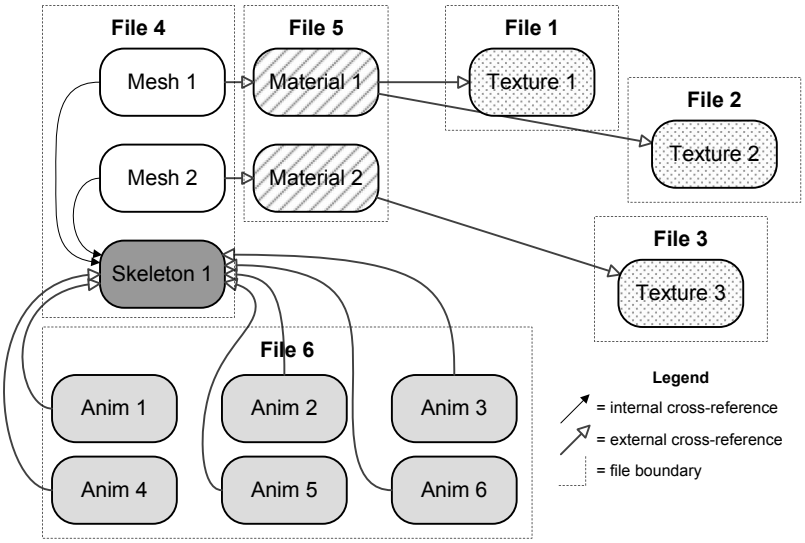

Figure 6.6. Example of a resource database dependency graph.

6.2.2.8 Composite Resources and Referential Integrity

Usually a game's resource database consists of multiple *resource files*, each file containing one or more *data objects*. These data objects can refer to and depend upon one another in arbitrary ways. For example, a mesh data structure might contain a reference to its material, which in turn contains a list of references to textures. Usually cross-references imply dependency (i.e., if resource A refers to resource B, then both A and B must be in memory in order for the resources to be functional in the game.) In general, a game's resource database can be represented by a *directed graph* of interdependent data objects.

Cross-references between data objects can be *internal* (a reference between two objects within a single file) or *external* (a reference to an object in a different file). This distinction is important because internal and external cross-references are often implemented differently. When visualizing a game's resource database, we can draw dotted lines surrounding individual resource files to make the internal/external distinction clear—any edge of the graph that crosses a dotted line file boundary is an external reference, while edges that do not cross file boundaries are internal. This is illustrated in Figure 6.6.

We sometimes use the term *composite resource* to describe a self-sufficient cluster of interdependent resources. For example, a *model* is a composite resource consisting of one or more *triangle meshes*, an optional *skeleton* and an