**Resource Manager**

**Resource GUIDs**

Resources have a GUID - for now, that's their resource file's name, followed by a colon (“:”), followed by the path to that file. Folders are separated by forward slashes, and the entire GUID should be lowercased.

Given some file "resource.res" in a folder "Folder" in a resource file named "resFile.zip", the GUID would be "resfile.zip:folder/resource.res".

If a GUID doesn't have a resource file prefix, assume the path is a relative path in the filesystem and continue as normal.

Resource files are just ZIP files - ZIPs have a directory tree, so it's easy to query. Almost everything needs some form of processing - we'll support this via a set of loader modules.

**File Formats Supported:**

Formats that must be supported by September in red. Types that must be supported by September in BRIGHT RED!

* Meshes
  + Everything supported by Assimp. Three seem very compatible:
    - DAE (COLLADA)
    - 3DS
    - OBJ
    - Blender doesn't seem to be well supported...
  + *Maybe* FBX, after demo.
* Animation
  + ???
* Materials
  + Custom format, based on Assimp's material system (see Model Import Procedure). Details on format TODO.
* Textures
  + PNG (png++)
  + JPEG (JPEGReader)
* Sound
  + OGG (Vorbis for compressed)
  + WAV?
  + *Maybe* MP3 towards the end of the year.
* Settings, Physics Parameters, Maps...
  + XML - need to specify exact document layouts, however

The manager needs to know how big the file will be in memory, not just how big it is on disk. It also needs to know what can be unloaded to free up cache space (probably a Least Recently Used list), be able to load multiple resource packages, and be able to handle crossreferences between resources (models, for instance, *must* be composed of a mesh, material, and texture). Here, crossreferences will be represented by the GUID of the referenced object; in the case of a model, this means a Model object in memory would have Resource objects (or at least handles to Resource objects) of the constituent mesh, mat., and textures; on disk, it'd just have the GUID, possibly having all of the references stored in a subsection in the file or in a separate metadata file.

**Cache Initialization**

Caches must be a fixed size; they're usually a fairly small size given the data used, however. There might be a GB of data, but the cache might only be a hundred MB. Smaller than even that, maybe, depends on the situation. Whatever the case, it should also be capable of handling multiple resource files.

**Resource Buffers**

Individual resource buffers need to use reference-counted pointers to remove the need to inform the cache manager exactly when it's safe to delete resource data. It'd be possible to implement this locally for just Resources, but C++11 has a class for this, the shared\_ptr. Of course, this means using allocator hooks, but it should work anyway. A ResourceManager should keep a shared\_ptr to each resource in its cache; when it's okay to flush from the cache, just remove that shared\_ptr. Once no more references to the resource remain, the resource's memory will then be deleted.

**Resource Files**

At the moment, resource files will be ZIP files handled via zlib. zlib's free, so we can use the code directly in the engine's code, or build a static library. It only handles the direct compression/decompression of the data, however; ZIP files contain directory and information structures as well, and you will need to use a separate library for that or make your own code to handle it.

You can probably go with your own code; use a DataStream to read the ZIP's directory data, and to get a file in the archive you do a partial read at whatever position the directory mentions. Remember that file structures must have no compiler packing (set #pragma pack(1) at struct starts, #pragma pack() at ends).

Process is as follows:

* Open archive as binary stream.
* Go to end of file, and read a struct listing archive directory information. That has a file offset to the headers for each file, which are listed in a big ol' array, sorta. There's variable length data, and the header specifies its VLD length so you can get to the next header.
* Each file header has a file offset to the header for the file's data. To get the compressed data, seek to that offset plus the size of a data header.
* All of these headers have signature values; if they don't match, you know the file's corrupted or you read it wrong.

**Misc.**

Use extensionless GUIDs for shader loading? Or just use shader names, and assume shaders will be in a specific folder in an archive?