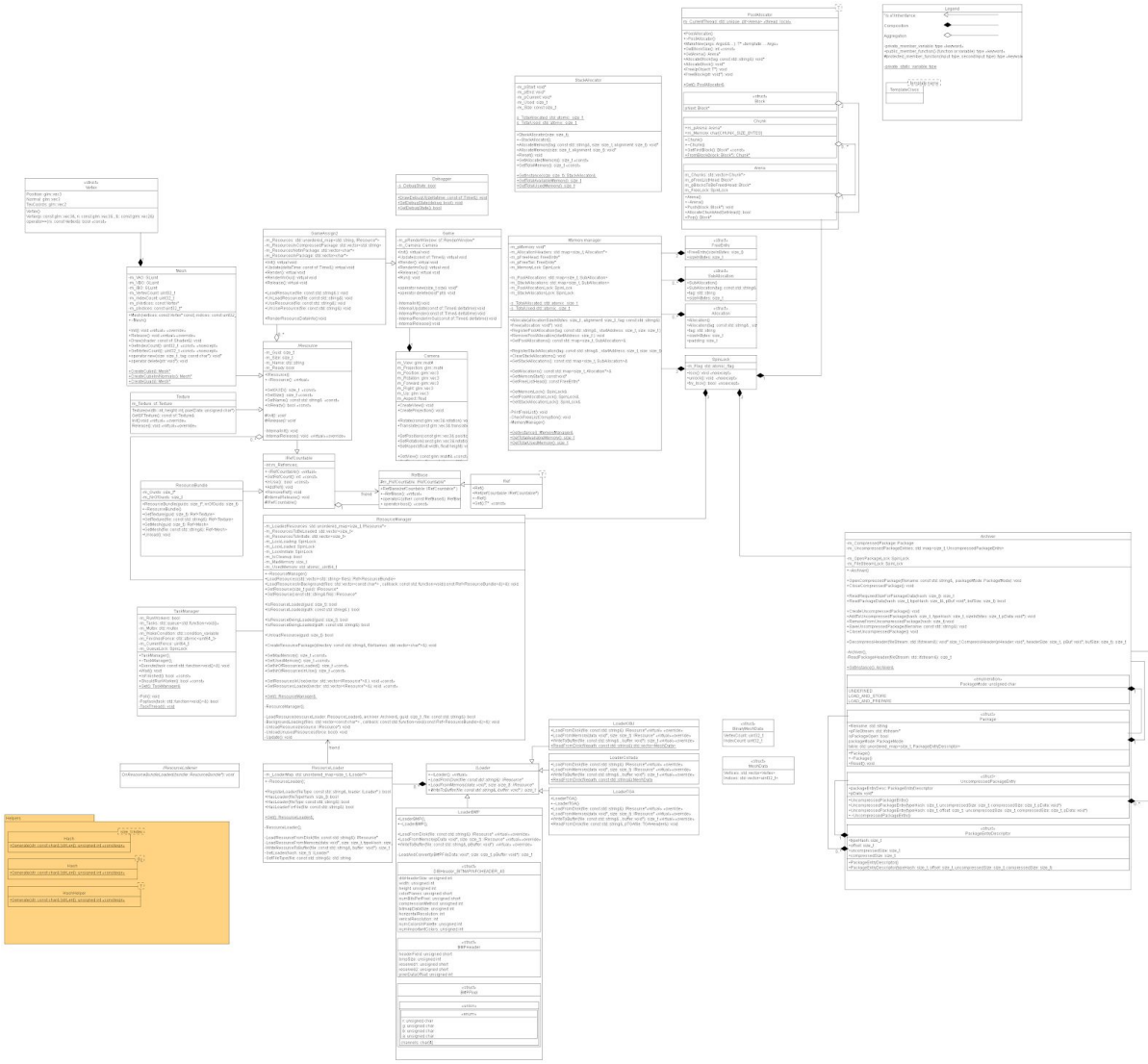
Resource Manager - Assignment 3

Design



Implementation

The implementation was done modularly where team members could work on some module of the whole system without getting in the way of some other team member. The Resource Manager itself is what connects all of these different modules.

Task Manager

The task manager is a pretty simple manager that makes use of the std::thread::hardware_concurrency() function to find out how many logical processors the system has. It then creates this many worker threads that immediately start polling a queue (Made thread-safe with a spinlock) for tasks to perform. A task is simply a function pointer that the worker calls. This means that tasks that are meant to be run for a long time should not be utilizing the task manager since if all the threads is busy all the time the tasks in the queue will never be finished. It is made for making simple function calls asynchronously. But it works perfectly for loading resources since all the resources that should be loaded can be queued up and executed on another thread when a thread becomes available, since loading a single resource can often be seen as a simple task.

The worker thread communicates with the manager with the help of a condition_variable, and the thread is not spinning all the time waiting for a task, it is put to sleep and awoken again with help of the condition_variable, so the manager wakes up a thread when a task is put in the queue.

Archiver

The archiver uses a simple archiving technique that we designed in conjunction with simple usage of the zlib compression library. It allows for both the creation of new resource packages and the reading of existing resource packages. The package is designed in such a way that every resource added to the package is compressed on its own, not together with other resources. This allows for single instances of resources to be read, decompressed and then instantiated without reading other instances of resources, therefore saving system memory. To find a given resource in the package, the package file stores a compressed header at the top of the file. The header contains information about where in the package a given resource is stored and other information needed about that resource.

When a package is opened by an application, the application can tell the archiver how to open the file, either LOAD_AND_STORE or LOAD_AND_PREPARE. LOAD_AND_STORE is very inefficient in system memory usage but is the faster of the two options, this is because when using this option, the archiver will open the package, read and decompress the header and store that in memory and then read the entire compressed block of data that contains all of the resources and store that in memory, and finally close the package. When a resource is later requested from the archiver it doesn't have to open the package on secondary storage again but can just use the data already in memory to fetch and decompress the requested resource. This is the implementation that was done first for the archiver and is typically not recommended. Instead, we recommend using the

LOAD_AND_PREPARE option instead which is what the resource manager uses by default when accessing a package.

When an application opens a package with the LOAD_AND_PREPARE option the archiver will open the package, read and decompress the header and store that in memory and then return. It doesn't close the package unless explicitly told to in order to improve performance. When a resource is later requested the archiver will use the header stored in memory to find the compressed resource in the package file, read and decompress it and return the data to the caller. This makes LOAD_AND_PREPARE more memory efficient than LOAD_AND_STORE but makes it a bit slower.

A package made by our archiver has the following structure:

- (Integer) Header Compressed Size
- (Integer) Header Uncompressed Size
- Compressed Header
- Compressed Resource Data

The header has the following structure when uncompressed:

- (Integer) Number of Resources in Package
- (Integer) Size of Compressed Resource Data
- Table of Resources, where every entry has the following structure:
    - (Integer) GUID
    - (Integer) Type Hash
    - (Integer) Offset in Compressed Resource Data
    - (Integer) Uncompressed Data Size
    - (Integer) Compressed Data Size (0 if not compressed)

One problem with compressing each resource by itself is that the compression algorithm becomes much less efficient especially for small resources. If a resource is very small, typically less than 32 bytes the compression algorithm will actually make the compressed data bigger than the uncompressed data. This is because the compression algorithm has to store a header in the compressed data as well. We worked around this problem by allowing non-compressed resources to be written to the package if they are detected to become larger after compression. A better solution, however, would be to allow the choice to bundle up all the small resources and compress them together. This would allow the total file size of the package to become smaller but degrade the memory usage and speed when reading the package. The file size reduction could, however, be worth it if the user has a large number of small resources. An even better solution would be to allow the user to select groups of resources to be compressed together and then allow the user to read all of those resources at the same time. This would improve the file size and could improve the speed if resources that are used at the same time by the application are bundled together. It was however decided that these two solutions would take too much time to implement and that our current implementation was sufficient.

Resource Loader

The resource loader (RL) is a singleton class responsible for mapping file the correct loader. This is done by registering a loader for each file type, for example (.tga). In the initialization of our program, all loaders are registered and put in a map with the filetype hash as the key. When the RL gets a request to load a resource, it will first try to find the correct loader by removing unnecessary parts of the filename, and then use the hash in the map. If a loader was found, then one of its member functions will be invoked, depending on if the resources should be loaded from the package or from disk. Each loader inherits the *ILoader* interface and implements its mandatory functions *LoadFromMemory* and *WriteToBuffer*. The purpose of the function *WriteToBuffer* is that the resource should be read from disk and then converted into a byte array. The byte array is then used to store the resource in a package. The second function is used to retrieve the data from a given buffer to be able to reconstruct the resource. Since each file type has its own loader we can be sure that the resource can be serialized both ways and in only one specific place. Currently we have four loaders, (.tga, .obj, .dae, .bmp)

OBJ

The OBJ loader has support for most OBJ files that contains a single mesh. OBJ files that contain multiple meshes is unsupported for the moment. It also supports triangulation of meshes that has faces with more than three vertices.

Since OBJ files are simple text-files the loader starts by reading the whole file into a buffer. Then the loader iterates over each character to see if it can find v, or f. If it finds any of these it starts to parse the element, otherwise, it skips all characters until it founds a new line.

An OBJ file has several types that the loader is interested in. these are v, vt, vn and f. These stands for vertex-position, vertex-texcoord, vertex-normal and face. The vertex-attributes can be two- or three-component vectors, and if more data is found on the line the loader returns an error. This means that it only supports files that have the “correct” layout. Data parsed from the vertex-attributes are put into separate arrays (std::vector), one for each attribute. The final vertex is then combined when the loader parses the faces.

The loader makes sure that there only is one copy of each unique vertex. This is done by having two arrays (std::vector) and one hashmap (std::unordered_map). When constructing a vertex it is hashed into the hashmap that holds the vertex as key and the value is the index inside the array of vertices. If it does not exist in the hashmap it is inserted, and finally, before the vertex construction is complete the vertex is also inserted into the array of indices. This approach makes it use a lot of memory because it stores the vertices and indices both in an array and in a hashmap, but it speeds up the loading tremendously.

Collada

The collada loader uses tinyxml2 to load the collada file and take care of the XML-parsing. When the loader starts to actually load data from the XML-nodes it starts by looking for the

<COLLADA> tag. If this one is not found it returns immediately. It then searches for the <library_geometries> tag to find all the geometry in the file. If it finds multiple meshes it will combine these into one single mesh.

Collada stores geometry in different kinds of XML-tags. This loader has support for the <mesh> tag. A mesh consists of many subtags and the loader supports <triangles> and <polylist> tags. They are very similar, they both specify a certain amount of <input> tags that specify an attribute for a vertex in the faces that builds up each primitive. The biggest difference between triangles and a polylist is that a polylist can contain polygons that have more than three sides. The same algorithm for triangulating these faces as in the OBJ-loader are used (So the collada meshes also supports non-triangle meshes).

A mesh also contains a number of <source> tags. These tags stores the actual data for a vertex-attribute. This means that there is a source-tag for each attribute, and when parsing the triangles or a polylist the loader has two go through the sources for each attribute and find the right element for that vertex. This is very similar to how the OBJ-loader operates that it stores positions, normals and texcoords in different arrays (These gets constructed when the source-tags are being parsed) and the actual vertices are constructed when parsing the faces.

The same problem arises here as with OBJ, we will have multiple copies of a certain vertex. So we use the same technique as with the OBJ loader were we store the vertices, indices in arrays and then store unique vertices in a hashmap.

The biggest difference between the OBJ and COLLADA loader is that the COLLADA loader makes heavy use of the standard library and a lot of memory is consumed, especially when loading big files. And could probably be improved.

BMP

The BMP Loader has support for BMP textures that use the BITMAPINFOHEADER_40 DIB header which is the common Windows format header. It also only supports uncompressed 24 bits per pixel BMP images, which is the most common type of BMP images. When reading from file it will read the file header and the DIB header contained in the file which stores information about the image that is needed to load it. If the settings read are supported the loader will create a buffer to store the pixel data, this buffer will be in RGBA format. It will then go through the pixel data stored in the BMP file, which is stored in a BGR format and copy the appropriate information into the allocated RGBA buffer. Then, depending on which function this is it will either instantiate a new texture resource with the size of the image and the pixel data as parameters and return that to the caller. Or it will create a new buffer which it memcpy's the BMP width, height and RGBA pixel data into, in that order. This buffer can then be used to save the texture data to a package so that when the application requests a BMP texture resource from the package the loader doesn't have to parse any BMP file header or DIB Header and check for supported formats and settings. Instead it can just assume that is is a supported format and it will instantiate a texture resource and return it to the caller.

TGA

The TGA loader has support for uncompressed True-color image which means that each pixel in the TGA texture is 32 bits small, each colour being one byte small and an alpha value which is also one byte small. When loading from disk we bytewise read the data and save the needed information into our own defined TGAHeader. If correct image type is read a data buffer of the image size is created with help from our memory manager. After the pixel data is read we need to swap the colour values from BGR to RGB. The pixel order of which the pixels should be read is given from the image descriptor. The two choices are either top-to-bottom order or left-to-right order.
When we load from memory we have already stored the height & width of the image and the pixel data in memory, these three components are then used to create the texture.
When we want to write the TGA image into our memory we memcpy's the TGA width, height and RGBA pixel data into a buffer newly created buffer.

Resource Manager

The resource manager (RM) is a singleton class that handles loading and unloading of resources. It does also handle the creation of a resource package. Each resource has a GUID and it is calculated using a hash of the filename. To create a resource package, we first have to call the *CreateResourcePackage* function and specify the name of each file that should be included. The RM will then create a large buffer and go through every file. For each file, the *WriteResourceToBuffer* function is called in the RL. If successful the resource is now properly serialized in the buffer. The buffer is then handed over to the archiver. When all files are serialized the *SaveUncompressedPackage* in the archiver is called.

To load resources from a package the *LoadResources* function or the *LoadResourcesInBackground* function can be used. The *LoadResources* function takes a list of filenames as argument. Each file is then checked against a map to see if the resources are already loaded. If the resource is not loaded the *LoadResource* function is called. The resource is now checked against the archiver to see if there exist resources with the given name. If it does, the size of the resource is obtained. Now the size is checked against the memory limit of the RM to see if it will fit or if we need to unload other resources. If we need to unload resources the RM will unload resources that have no references i.e. they are not being used. When the returned size is checked once again to see if the resources fit this time. If not the resources loading is aborted and an error message is displayed in the console. If the resource fit then the resource data is read from the archiver and if successful the data is handed over to the RL to find an appropriate loader. If successful a *IResource* is returned and saved in the loaded resources map. When all resources are loaded the function returns a ResourceBundle that holds all the GUIDs loaded from the current *LoadResources* call. From the resource bundle the user can simply get each resource casted to the correct type. The purpose of using resource bundles is so that each state of a game, for example, a level can easily be loaded together. When a level is over the resource bundle can simply be unloaded.

As for the other function *LoadResourcesInBackground,* the process is very similar. But it require a callback function in the argument. The purpose of the callback function is so that the caller can get notified when the resources has been loaded and ready to use. It all begins by getting a thread from the task manager and then we do some checks for each file to see if it is already loaded or if it is currently being loaded by another thread. If not the file is put in a list of files to be loaded by the current thread and it is marked as being loaded. When done each resource is loaded exactly like in the direct loader, using the function *LoadResource.* After each resource has been loaded the thread will check if any required resource not loaded by the current thread, has finished loading by another. If this is false the thread will wait until all required resources are loaded. When done, the callback function is invoked and the caller will get notified that the resources have been loaded.

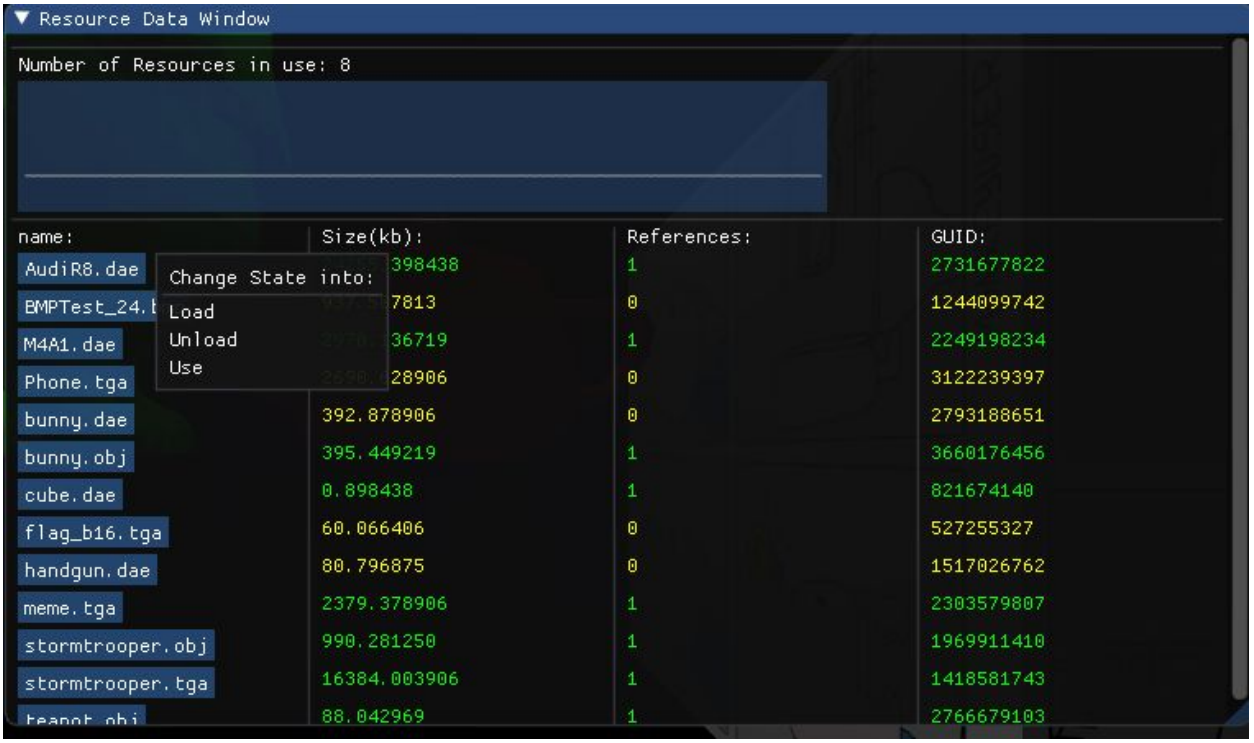
Packaging Tool

The packaging tool uses ImGui to create a simple UI window that can be used to create packages from files stored on disk. These files need to be one of the supported formats described previously. The layout of the UI of the packaging tool can be seen below.

As can be seen the UI contains two lists, the Available Resources list and the Resources in the Package list. When the user first starts up the Packaging Tool the Resources in Package list will be empty and the Create Package button will not be visible. The user can then select a resource in the left list, press the right-facing arrow and the resource will be added to the right list. When the user is satisfied with which resources are in the package he/she can press the Create Resource button which will tell the archiver to compress all the resources and create a new package file in secondary storage. It will also create a PackageHeader file, that stores the name of every resource added to the package, this can be used by the debug application to know what resources are stored in the package.

Resource Data Display

In order to keep track of how many resources are being used and in what current state each resource had a display was created that should try to capture this.



Each resource can be in 3 different states, either not loaded(Red colour), loaded(Yellow colour), loaded & used(Green colour).
The user is also given the opportunity to change the state of the chosen resource by clicking on a resource and assigning it a new state. As also can be seen the GUID that represents each resource as well as the size of the resource in kilobytes.

Performance

For the performance tests we tried loading all the files in the project with single threaded and multi threaded loading directly from files and from the package.

- Single threaded, 1.4476 seconds vs 0.1743 seconds from the package.
- Multi threaded, 1.3955 seconds vs 0.1265 seconds from the package.
- Compression Ratio: the package is about ~30% of original resources size.

The differences observed here can be mostly explained by the parsing of the different file types being done when the package is created. In the package all resources are stored in a standard format that we designed so it is much more easily loaded.

Work Distribution

Herman Hansson Söderlund

- Archiver
- Packaging Tool
- BMP Loader
- (Memory Manager Bugfixes), not related to the assignment directly

Christoffer Andersson

- ResourceManager
- ResourceLoader
- Background Loading
- Resource Limits
- Reference Counting

Alexander Dahlin

- OBJ Loader
- COLLADA Loader
- TaskManager
- Renderer and other SFML setup (Not really related to the assignment)

Tim Mellander

- TGA Loader
- UI utilizing the implemented functions

Tim Johansson

- UML Diagram