**Game engine architecture document**

OGLEngine, Leo Asplund, 12 Dec 2024.

***Introduction:***

Entities in OGLEngine are called GameObjects. GameObjects can have Components as well. I’m trying to make a system similar to Unity’s GameObject and Component system.

My basic managers right now are as follows:

* **GameObjectManager**, singleton, keeps track of game objects allocated. Has functions for CreateGameObect (creates a new game object) and DeleteGameObject (deletes specified game object).
* **MeshManager**, singleton, has a list of meshes that have been loaded and cached. Has functions for loading a new mesh as well as messaging functions like QueueMessage and ProcessMessage. Since the MeshManager processes messages on its own threaded context, it can for example call LoadMesh() there as well if it receives a message telling it to do so, meaning that the mesh will be loaded in that same thread!
* **GameObjects** manage the creation and deletion (not implemented yet, oddly enough) of their components.
* **Engine class** handles the editor’s flying camera right now. I should make this a singleton in the future.
* **Graphics class** calls Update() on all GameObjects (maybe I should move this to Engine). Graphics does the necessary OpenGL stuff to render, like clearing buffers and whatnot. I’ve also put the function EscapeToCloseWindow() here, which means pressing Escape will make the while-loop most of these managers operate in terminate and exit the program. I should probably also make this a singleton in the future.
* **EditorGui** class is where most of the base code for rendering the ImGui interface for the editor is located. Where applicable, I’ve made specific functions for rendering GameObject and Component specific details in those specific classes, so the EditorGui class doesn’t have to do everything. I should also probably make this a singleton in the future.
* **++ ShaderManager,** singleton, currently houses the regular shader and the shadowmap shader. Call this manager to access the shaders.
* **++ MemoryCheckManager,** singleton, threaded, communicate only with messages of type MemoryMessage, has functions for checking if available memory is enough and for if a specified file is too big. Can send back a MemoryMessage with the results of the check-functions.

What do you need to know to use my engine? Besides learning some C++, I’ll say read my future additions to this document.

***Architectural choices:***

Right now the MeshManager is threaded. I primarily wanted it to be able to parse, load, serialize and deserialize meshes separately from the rendering part of the program. So it doesn’t freeze the program whenever we want to load in a new mesh. Messages used to communicate with it are MeshMessages, these hold information on what mesh to load.

How are mesh parsing/loading meshes optimized? Whenever a new mesh is parsed, the parser class (ObjReader in this case) will write the results of the parse to a binary file. We can then read the contents of that binary file when we want to load the same mesh again. And since we already have the results of the parse, we don’t need to parse again and can just load it in directly from disk. Much faster than having to parse it every time. And when we’ve loaded the mesh into memory, it’s saved into a list of meshes by the MeshManager. MeshComponents can then access these meshes, which are allocated to memory, even faster. Many MeshComponents can thus also use the same cached mesh, which means we don’t have to allocate memory for duplicate meshes!

***Reflections and future design choices:***

***Learning:***

I’ve learnt a lot so far. It’s taken me this long (dec 2024) to finally grasp the usefulness of singletons. I wanna use them more and more the more managers I create. Better understanding of forward declaration when two classes have class variables of one another. Better understanding of when and when not to use global variables. Better understanding of the headaches that can come from trying to debug and step through threaded code. Writing and reading from binary files is so cool!

***Future:***

For the coming 3 months, I want to get scene/level saving and loading in, that’s an integral feature. Lighting and physics of course, which we’ll have lectures on. I want to find time to do a proper refactoring and cleanup of ***everything***. This includes scripts, execution flow, managers, threading, messaging, and project solution structure. The lot. Docking ImGui instead of the floating ImGui windows I have right now. Resizable rendering window, and Resizable Windows window. Gizmos is important to get in as well, for the user experience. Later maybe a play-mode as well, when basic gameplay can be made with the engine.

*What managers mentioned would be relevant to my game engine? (see lecture or powerpoint from lecture for inspiration)*

**Singleton managers:**

(order of priority):

+ Mesh manager,

*-for caching loaded meshes and using cached meshes instead of loading new ones every time.*

- Texture manager,

*-load textures,*

*-list of textures,*

- Camera manager,

*-multiple cameras, especially like in Unity where you can have both the Play-in-Editor window and regular editor window viewing at the same time.*

*-camera settings.*

- Graphics manager,

*-keep track of which entities are registered for rendering (e.g. a bool in the hierarchy window to enable or disable rendering of a specific object.)*

++ Shader manager,

*-keep track of loaded shaders, assist in loading shaders, recompiling shaders if needed, OS/platform specific shader selection.*

- Maybe even a material manager / editor, so we can have specific materials for meshes whenever we implement PBR.

- Audio manager,

*-queue sounds,*

*-play sounds,*

*-queue music or ambience,*

*-play music or ambience,*

*-FMOD or OpenAL middleware communication and integration. Audio is important* *for games.*

- Animation manager,

*-load and manager 3d animations for models or particles (requires better asset importing than just .obj, right? Since .obj doesn't support animations).*

**Lower level singleton managers:**

- Input manager:

*-keep track of inputs,*

*-keep track of states of keys and buttons for both keyboard, mouse and gamepad.*

- Window manager:

*-Create and handle OS-windows, like in case I want to move an ImGui window to my second screen.*

*-Handle auxiliary windows (basically just an ImGui window manager for any tools I create for my editor in my case.).*

*-Process OS-level events (e.g. wndproc).*

- Application State manager:

*-handle changing states if my application has several modes (e.g. edit mode, play mode and whatnot).*