**Genesis Engine**

**Developer : Ivan Vukov**

**Student Number : s5071537**

**Github :** [**https://github.com/ivanchotom**](https://github.com/ivanchotom)

**Email : ivanvukov18@gmail.com**

* **Content**

1. **References / Learning resources**
2. **Build Info**
3. **3rd Party software**
4. **Design**
5. **Features**
6. **Debug Tools / Profiling**
7. **Known bugs**
8. **Future improvements**

* **References / Learning resources**

The development was largely inspired by the main learning resource used that was The Cherno youtube channel and discord community maintained and represented by Yan Chernikov.

<https://github.com/TheCherno/Hazel>

<https://github.com/TheCherno/Sparky>

His game engines Hazel and Sparky were a source for learning how to build a game engine and a great motivation to strive to. The devlogs on Hazel were the biggest gem I could find during the planning of my Game Engine.

The book that was with me during the development of Genesis was “Game Engine Architecture 3rd Edition” by Jason Gregory.

Gregory, J. (2019). *Game engine architecture*. Boca Raton ; London ; New York: Crc Press, Taylor & Francis Group.

* **Build info**

The build software used for Genesis was Premake due to the idea that me as a developer knowing a secondary build software than CMake would be beneficial and a scripting language named Lua that Premake uses.

**Premake :** [**https://premake.github.io/**](https://premake.github.io/)

**Version used: v4.4**

Text

Description automatically generated

The Lua scripts used in some of the 3rdParty software and the main SolutionDir are quite easy to read and just define the basic structure of the solution and its dependencies and how it links to the Sandbox and Editor projects. The lua script is ran by a batch file that generates the necessary project files required to run the program. Currently the Engine only runs under a Windows OS and Visual Studio compiler.

**Setup location: SolutionDir/Setup/GenerateGenesisWindows.bat**

**Currently supported platform: Windows**

**Currently supported compiler : Visual Studio**

**The program was tested on both Visual studio 2019 and 2017**

**System requirements :**

**OS: Windows XP or later**

**RAM: 4GB of RAM or higher**

**Graphics Card: compatible with openGL 4.5; Dedicated or integrated graphics card**

**CPU: 1 gigahertz (GHz) or faster 32-bit (x86) or 64-bit (x64) processor**

**Input : Mouse and keyboard**

* **3rd Party Software**

**3rd Party Software location :**

**Genesis\GenesisEngine\3rdParty**

Notable 3rd party software that is used in Genesis that might be new from previous projects submitted for University assignments are :

* **ENTT**

EnTT is a header-only, tiny and easy to use library for game programming and much more written in **modern C++**.  
[Among others](https://github.com/skypjack/entt/wiki/EnTT-in-Action), it's used in [**Minecraft**](https://minecraft.net/en-us/attribution/) by Mojang, the [**ArcGIS Runtime SDKs**](https://developers.arcgis.com/arcgis-runtime/) by Esri and the amazing [**Ragdoll**](https://ragdolldynamics.com/) Autodesk Maya plugin.

**Github :** [**https://github.com/skypjack/entt**](https://github.com/skypjack/entt)

The decision to use ENTT as opposed to writing a personal ECS was mainly time management and scalability of the engine after the submission for the assignment. I stuck ENTT and built classes to abuse its capabilities and found it easy to get into, along with Cherno’s explanations.

* **IMGUI**

Dear ImGui is a bloat-free graphical user interface library for C++. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline enabled application. It is fast, portable, renderer agnostic and self-contained (no external dependencies).

Dear ImGui is designed to enable fast iterations and to empower programmers to create content creation tools and visualization / debug tools (as opposed to UI for the average end-user). It favors simplicity and productivity toward this goal, and lacks certain features normally found in more high-level libraries.

Dear ImGui is particularly suited to integration in games engine (for tooling), real-time 3D applications, fullscreen applications, embedded applications, or any applications on consoles platforms where operating system features are non-standard.

**Github :** [**https://github.com/ocornut/imgui**](https://github.com/ocornut/imgui)

* **IMGUIZMO**

ImGizmo is a small (.h and .cpp) library built ontop of Dear ImGui that allow you to manipulate(Rotate & translate at the moment) 4x4 float matrices. No other dependancies. Coded with Immediate Mode (IM) philosophy in mind.

**Github :** [**https://github.com/CedricGuillemet/ImGuizmo**](https://github.com/CedricGuillemet/ImGuizmo)

* **SPDLOG**

Very fast, header-only/compiled, C++ logging library.

**Github :** [**https://github.com/gabime/spdlog**](https://github.com/gabime/spdlog)

* **YAML-CPP**

yaml-cpp is a [YAML](http://www.yaml.org/) parser and emitter in C++ matching the [YAML 1.2 spec](http://www.yaml.org/spec/1.2/spec.html).

**Github :** [**https://github.com/jbeder/yaml-cpp**](https://github.com/jbeder/yaml-cpp)

**Lua Scripts for 3rd Party software**

* **Genesis\GenesisEngine\3rdParty\assimp\premake5.lua**
* **Genesis\GenesisEngine\3rdParty\Glad\ premake5.lua**
* **Genesis\GenesisEngine\3rdParty\GLFW\premake5.lua**
* **Genesis\GenesisEngine\3rdParty\ImGui\premake5.lua**
* **Genesis\GenesisEngine\3rdParty\yaml-cpp\premake5.lua**

**Text

Description automatically generated**

* **Design**

The design for Genesis had one specific goal that I wanted to achieve before I submit this assignment and that was a level editor with some UI and Guizmo support for primitive objects rendered on the screen. Whitebox level design is a powerful tool used for creating a simple draft of how a level in a game will look like and that to me seemed like a reasonable goal for my experience and time given for this assignment. That goal is almost achieved with the only things missing for it to be perfect is the different primitives that the engine could render instead of just quads.

Logo, company name

Description automatically generated

This was because I decided to implement mesh/model loading in order to get more points, yet saw myself struggling to implement that with all the necessary features I wanted to have for it to be usable. In a way I did not just want a mesh to load without the user having any control over it (submeshes, guizmos etc.).

The scope of the project always was to be able to showcase something that can potentially be a nice portfolio piece for potential employers that showcases my ability to build a complicated system by myself. For that reason the project needed to be scalable in a way that if I decide to ever add support for another rendering API, that would happen without the need of rewriting a large portion of the engine in order to accommodate. For this reason the classes written to be in charge of the Rendering side of things needed to be made virtual and not tied to any specific rendering API, so that they can be overrided by whichever API the user of the engine would like to use or whichever rendering API is supported on the system that is running the engine (windows = DX, macOS = Metal).

For this reason most of the classes inside the “GameEngine /src/Renderer” directory are using virtual functions.

A picture containing text, indoor, screenshot, black

Description automatically generated

And are overridden in the OpenGL specific classes for them, that can be found in “GameEngine/src/Platform/OpenGL”.

Text

Description automatically generated

Doing it this way will hopefully abstract the RenderingAPI specific part that is specific to OpenGL,Vulkan or DirectX and the part that is independent and abstract and can be used for any sort of RenderingAPI the user wants to use.

Speaking of platform specific things the Engine needed to also have platform specific Window creator. As of at the moment the only window creation api that is used is GLFW, but that might be subject to change if the engine ever supports more platforms (Linux, macOS, Android). Aside from window creation different platforms might have different input methods and for this reason there is Windows specific definition for those things found under “Platform/Windows”.

Graphical user interface, text

Description automatically generated

The Utility folder under Platform is simply a class for open and safe file functions.

Another very powerful feature that is the layer system adopted by Hazel that lets the engine have a contiguous list of layers that determine in what order everything is drawn. This would be a very useful tool for not only the Editor project, but also for future UI tools and Rendering effects that need to be in front of everything else drawn behind them. For this the Layer and Layerstack classes come in that can be found under “GameEngine/src/main”. The layer class is also closely tied to both the ImGui which is the current UI library used and the Event system inside Genesis to dispatch events in accordance to which layer is the currently active one.

Graphical user interface, text

Description automatically generated

As with other parts of the engine the layering system is also made virtual so that applications like the Editor created with the engine can override and define their own specific layers. In the source files of the Editor we can see a Layers folder in which the editor defines its own unique layer.

Text

Description automatically generated

This system will greatly evolve and help enormously in future endeavours like the final year project I will be doing that will implement a form of “Eye Picking” for the UI of the game I will try to make using a Tobii eye tracking hardware. Where the recognition on which UI element the eye is hovering over should only be done in the UI specific layer, this will fix the problem of having separate UI id’s and entity id’s and will make it so the mouse is only viable for one layer and the eye tracking/picking only viable for the other layer.

Genesis is built as a static library with which other applications that want to use Genesis’s capabilities link to, yet even though it builds as a static lib it still needs to have a main function residing in it, that defines where the beginning of all the code is. The EntryPoint can be found under “src/main/EntryPoint.h” and is very simple.

The Core.h found also in the main folder is the place where Platform detection (Gives error if a platform other than Windows is running Genesis), Asserts and templated pointers are defined.

In the Event folder is where all the classes for event handling can be found. The notable parts of the Event system in Genesis is the separation of events and how they are handled by the “EventDispatcher”, in order for Genesis to scale and be an adequate engine that can support a complex Editor application and Games created with it, events needed a system that is brocken down into small parts that can be debugged and manipulated for specific use cases.

Text

Description automatically generated

* **Features**

1. **Log system**

The log system inside Genesis lets the developer log certain events that happen inside the Core (Engine) or inside the App (Editor or Game). A logging system is a nice addition for debugging as well as a industry standard thing to have in these sorts of applications that eases visibility on things that are getting tracked inside the application. QA workers inside IT (something I have previously worked in) use consoles inside games for special commands that make them test the application and a logging system that supports colors, although simple is used quite a bit.

1. **Pre-compiled headers**

Using pre-compiled headers helps with not including large standard library or other kinds of includes that they themselves include other large dependencies. The compilation process becomes a lot faster if the compilation of those types of includes only happens once or at least less than it would without pre-compiled headers. This is vital for any large project.

1. **Camera system**

Genesis currently supports different camera systems. The editor uses an Editor camera that can browse freely around the scene, and inside the editor camera entities can be created via the hierarchy panel that have UI for manipulating their properties like projection, fov or transform.

Graphical user interface

Description automatically generated

1. **Panels**

Genesis supports panels that are created via ImGui and hold valiable information -> how many entities are in the scene, what rendering stats are being processed in this current scene, what entity is the mouse currently hovering over, what properties does the currently selected entity have.

Text

Description automatically generated

1. **Mouse picking**

Mouse picking is something that is very intuitive and pretty much an industry standard that is in all game engines at this point. Having it greatly hastens the progress of work for anyone that is using the Editor.

1. **UI system**

The UI is one (maybe the largest) part of a game engine that makes it usable by other people that are going to be USERS not developers of the engine. It is one of the things that makes Genesis a game engine as opposed to a project filled with classes that only deal with rendering stuff on the screen, but no real user in mind that will potentially be using this game engine.

1. **Guizmos**

The game engine supports guizmos that combined with mouse picking make the editor even more user friendly and intuitive for manipulating the transform of objects.

A picture containing text, businesscard, sign

Description automatically generatedLogo

Description automatically generated with medium confidenceIcon

Description automatically generated

1. **Save, Load, New Scene**

Genesis currently supports the saving, loading and new scene features under the File button at the top right of the window. The legacy option is as is called legacy way I used for saving and loading a scene from file. This is a simple way of managing work done inside the editor that can be resumed later on.

Graphical user interface, text, application

Description automatically generatedGraphical user interface

Description automatically generated with medium confidence

1. **Scripting**

Genesis currently supports Native scripting that is just a simple way to test certain entity behaviour in a way that does not involve the integration of any scripting language (C# or Lua) which for the moment is unneeded.

1. **ECS**

The entity component system used inside Genesis is using a 3rd party header and made into classes inside the engine that can take advantage of EnTT’s capabilities of an ECS. Templated functions as well as definitions of what a components genesis currently supports and genesis’s definition of what an entity inside the scene is can be found under the “Scene” folder

**Currently supported components :**

**TransformComponent**

**SpriteRenderComponent**

**Tag/EntityID Component**

**CameraComponent**

**NativeScriptComponent**

* **Debug/Profiling**

Anything to do with rendering should have some sort of debugging system designed for it to help in the development process and in Genesis there are 3 main components designed just for that. Logging is once of them discussed in Features, the other two are the Asserts and Profiling macros that can be found all around the source code.

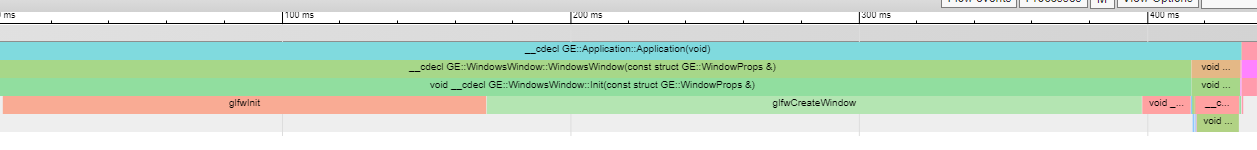
Asserts are designed to initiate a debug break whenever something in the code is not acting how it is supposed to. They are a form of “checks” to see if for example something has loaded properly, if a variable has changed etc. When combined with the logging system they provide valuable information on what exactly went wrong in the code.

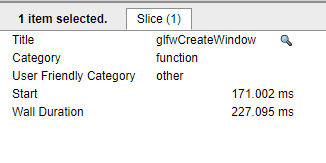
A screenshot of a computer

Description automatically generated with low confidence

Profiling is a way to visualize the overall runtime of the program, and the Instrumentor header is designed just for that.

URL : chrome://tracing





* **Known bugs**

There are a majority of bugs still in Genesis. One of them would be that Guizmos as well as the properties panel sometimes are unresponsive and need a couple of additional clicks for them to start working.

Switching from one entity to the other in the selection context and them trying to adjust properties in the properties panel sometimes makes the properties panel disappear as if there is no selected entity.

Mouse picking the guizmos sometimes gives odd results.

The current implementation of mouse picking sometimes triggers a crash when the mouse is hovering over a invalid entity. For instance the middle Panel which is the viewport has tabs that can be hidden or shown when clicked at the top left corner of the panel. At the moment that tab is hidden due to crashes whenever the mouse would hover over it.

Loading and saving scenes sometimes does not wipe the data from previous scene from the viewport. So when a user loads a separate scene the stuff rendered from the previous one still reside on the screen.

* **Future improvements**

The renderer at the moment is not multithreaded and that is something that will greatly increase performance.

Model/Mesh rendering and the corresponding components that need to be in place for manipulating them should be the next step for Genesis.

Scripting using C# would make the entities have easy to implement behaviours.

Compute shaders

A second(Game view) Viewport that takes in a camera created and adjusted in the Scene Viewport that represents the players perspective. Unity and Unreal have this already where the game being created inside the editor can be tested in the Game viewport after pressing a Play button.