**The Sanguine Engine**

**Author: Vlad Munteanu**

**General Description**

The herein document represents the documentation for the current phase of Sanguine 3D Engine. This project was initiated as a course project for the 3DGE (3D Graphics Engine course).

The Sanguine Engine is an academic demonstration of how to develop a modern graphics engine. My personal goal with this project is to see how far I can develop it along the way, experimenting with a number of interesting features, by the end having a somewhat fleshed out/somewhat rudimentary Game Engine that integrates several major logic modules, besides the Renderer, such as, a Physics Engine and a Client/Server Connectivity layer.

**Project Structure**

The underlying scheme of the engine consists in our Engine project being built as DLL which exposes the needed API and is injected into the Client project. It publishes Application and Layer interfaces for working with the Application window and render surfaces, Input and Camera controller classes, Renderer API Commands and Primitives, mathematics library and so on.

Consequently, the Engine Core is written with an Engine Abstraction Layer (EAL) -> Middleware -> Implementation model in mind.

Namely, the EAL should be the top, API agnostic, most generic layer of the API, the one which is exposed to the Client module, offering a suitable interface for render commands, primitives, routines and so on.

Then the Middleware is responsible for determining any relevant system information such that it may select the proper Implementation to handle the users request for the selected/available graphics API.

The solution is generated based on premake5 series of script files which specify configure different aspects of the project, such as: compiler version, type of application (.exe/.lib/.dll), system configurations (Debug, Release), include directories, linked libraries and so forth. Most of the external libraries are either generated and linked through their own premake5 script, or, added as include directories from the root script.

Now that we’ve touched the subject of external libraries, let’s enumerate the used 3rd party libraries:

* *SPD Log* is a logging library over which I implemented a series of defines, so that I may have global console logging access, with differing levels of severity
* *GLFW & GLad* are the required libraries to enable OpenGL rendering capabilities
* *GLM* is a mathematics library based on the GLSL standard
* *ImGui* is a 3rd party GUI library which offers access to a whole suite of standard and intuitive controls
* *STB\_image* is used to support loading of multiple texture file formats
* *ASSIMP* is used to support loading of multiple 3D object file formats
* *EnTT,* an ECS dedicated API, with highly optimized storage and retrieval methods for templated components
* *YAML,* is for human readable formatting XML formatting API for serializing/deserializing scenes
* *ImGuizmo,* a plugin for the *ImGui* library, intended for drawing 3D Widgets
* *ASIO,* which stands for Asynchronous Input Output, is intended as the networking API for socket connectivity and communication

**Project Modules**

* **Client/Application/Game Module**

This should be considered the environment/workspace/sandbox/user space of the developer/user. Ideally this is the only project where he would have access to work with the exposed API from the Engine DLL.

Thus, through a series of extensive GUI controls, scripting support, shaders creation and management, scene and camera controls, he could ideally build a simulated world, which he could then export as a proper standalone application.

The basic requirements of this module as of now is that the user should first of all, define the desired rendering API, which for now it’s exclusively OpenGL, then, write a Client Application class which implements the Core Application interface.

Then, push the root drawing surface to the Layer stack and define a static function which returns an Application pointer to the Client Application, this will be internally used by the Engine to create the Application instance.

All that is left now is to attach a Camera to the viewing surface, configure our controllers and scene, and that should be it, the sky is the proverbial limit from here on.

A last thing to note here is that in this project it is recommend to hold your data or resources, respectively, in the directory named assets. There already exist a series of resources such as: shaders (the primary ones being FlatColor.glsl and Lighting.glsl), textures, 3D object files, etc.

* **Sanguine Engine**

Here is where we create the universe and laws that bind it. We will go through each of the individual components and note off any particular features.

* + *The precompiled headers (sgpch),* pretty much standard nowadays for projects with a larger code base; basically, it saves some compile time and some disk space, and the nicest thing is that we have global access to all the libraries included in here
  + *Sanguine.h,* this is the header file through which the Engine exposes its API to the Client side. This is required to be included by the Client project to have access to any required functionalities.
* **Core Module**
  + *EntryPoint.h,* here is where the *main* function resides, and where the Application instance is created and ran. Also, here is defined the entry point which should be implemented by the Client Application.
  + *Core.h,* here are located multiple defines, templates, constant or static variables.
  + *FilesystemUtils.h/ FilesystemUtils.cpp,* a helper type class for opening/saving file dialogs
  + *KeyCode.h* contain the internal key codes defines.
  + *MouseButtonCodes.h* contain the internal mouse button codes defines.
  + *Timestep.h* is a simple wrapper class over the float data type, which is generally used to pass the delta time between updates.
  + *Log.h/Log.cpp,* in here we basically instantiate 2 separate logger references and define a series of macros to call each of them for every log level. The reason for two loggers is that the Core one should be used exclusively in the Engine project, while the other one exclusively in the Client project.
  + *Window.h/Window.cpp,* these represent the wrapper class over the Application window. Contains a reference to the GLFW window which it creates internally based on a properties object provided through the constructor method. This is where all the appropriate input and window callbacks are set and where the OpenGL graphics context is created and loaded.
  + *Layer.h/Layer.cpp,* a simple Layer interface with the appropriate overridable functions.
  + *LayerStack.h/LayerStack.cpp,* a stack data structure specialized for Layer instances.
  + *Application.h/Application.cpp,* this class acts as a Singleton and holds a reference to the Application window which it creates internally, and also the Layer Stack, thus, here is where events are propagated down the stack and handled by its appropriate layer and also here is where the Application update loop is started and controlled, basically iterating through all the layers and updating them.
  + *Input.h/Input.cpp,* this is where a Input interface and WindowInput implementation can be found. These classes expose relevant input polling methods.
  + *Instrumentation.h,* here is where the Instrumention classes are defined for function profiling, alongside a few macros that provide the actual interface for using the profiler.
  + *MathUtils.h/MathUtils.cpp,* a math helper namespace, currently it’s main use is for decomposing a 4x4 matrix into the TRS vectors
* **Events Module**
  + *Event.h*, here is where the Event and EventDispatcher classes are defined, along with a series of values for event types and categories, methods and defines.
  + *AppEvent.h,* here is where the actual Application events are implemented, such as, WindowResize, WindowClose events.
  + *KeyEvent.h,* here are the actual implementations of the keyboard input events, such as, KeyPressed, KeyHold, KeyReleased.
  + *MouseEvent.h,* here is where the actual implementations of the mouse input events, such as, MouseMoved, MouseScrolled, MouseButtonPressed.
* **ImGui Module**
  + Here is only the implementation of the ImGui Layer, where we configure our layer to initialize, update and destroy the ImGui data.
  + I expect this module to grow a bit once I start working on a more extensive GUI for the Engine.
* **Entity Component System**

This module is based on the EnTT 3rd party framework, due to internal, highly optimized template sparse tree structure, which handles storing and retrieving component groups with very efficient runtimes.

* *Components.h,* this file holds the storage definitions for the majority of the working components available currently, e.g.:
  + *TagComponent,* component storage for a string value representing the entity in the active scene hierarchy
  + *TransformComponent,* this component holds and handles the logic for the TRS of an entity in the scene space.
  + *MeshComponent,* this component contains an actual OpenGLMesh reference and has a higher-level logic for working with the internal mesh object.
  + *CameraComponent,* the storage class for a Scene Camera object type
  + *NativeScriptComponent,* is the storage class which holds a instance reference to a Entity Behavior object, and the respective callbacks to which to perform the static binding for instantiating and destroy the behavior object
* *Entity.h/Entity.cpp,* this is a storage class for the Entity handle and a set of component accessor function based on the internal handle.
* *Scene.h/Scene.cpp,* this is the storage class for a scene hierarchy of entities. It’s responsible with adding/deleting entities from the registry and exposes a series of callbacks for when certain components are added to the scene.
* *EntityBehaviour.h*, the virtual interface for adding native scripted behaviors to an entity.
* *SceneCamera.h/SceneCamera.cpp,* a dedicated scene camera component for creating multiple render targets into the scene.
* *SceneSerializer.h/SceneSerializer.cpp,* a class the handles serialization/deserialization of a given Scene.
* **Renderer Module**

The general idea of this module is to define a generic interface for any relevant resource for working with the rendering pipeline. Effectively, here are implemented the EAL and the Middleware for specific render resources.

* + *Camera.h*, defines the interface that the Orthographic and Perspective classes should implement. The virtual interface has been provided so that the camera can work as a object component.
  + *OrthographicCamera.h/OrthographicCamera.cpp,* here resides the implementation of the Orthographic camera class.
  + *PersepctiveCamera.h/PerspectiveCamera.cpp,* here resides the implementation of the Perspective camera class.
  + *PerspectiveCameraController.h/PerspectiveCameraController.cpp,* the controller class for a Perspective camera object. Updates the camera’s View/Projection based on the position and direction vectors.
  + *Framebuffer.h/Framebuffer.cpp,* the framebuffer definition class. Very useful for when wanting to display multiple view render targets, or for drawing complex geometry (e.g., particle systems)
  + *GraphicsContext.h/GraphicsContext.cpp,* is the generic interface that defines the API for dealing with render API specific pipeline logic, in our case being the Init and Swap Buffers methods for now. We’ll see the actual use cases in the OpenGL Implementation class.
  + *RendererAPI.h/RendererAPI.cpp,* these files contain the effective rendering interface which we expose to the client side. Here is where we have defined our set of agnostic methods useful for working with the rendering device. Methods such as, Draw calls, Clear calls, Set Viewport etc.
  + *RendererCommand.h/RendererCommand.cpp,* this is the actual Middleware for calling actual implementation of the rendering API methods.
  + *Buffer.h/Buffer.cpp,* hold the generic interface definitions for the VBO and IBO classes, along with a Factory method which creates references to the actual render API implementation of these buffers. Also, we have defined here a Buffer Layout and Buffer Element classes for helping us setting up and loading the VBO data topology.
  + *VertexArray.h/VertexArray.cpp,* these files define generic interface for the VAO object, as well, with a Factory method to create specific references. To note here is the interface itself defines the VAO containing VBO and IBO buffers.
  + *Shader.h/Shader.cpp,* same as with the previous cases, here we have a generic Shader interface with Factory methods for creation. To note is the fact that the Shader class is defined with a set of instructions for binding various uniform variables. Also, a Shader Library class which works as a manager for a collection of Shaders.
  + *Texture.h/Texture.cpp,* same as before, here we have a generic interface for Texture class along with factory methods which create the actual implementation objects.
  + *TextureManager.h/TextureManager.cpp,* a helper class for managing a collection of Texture objects.
  + *Renderer.h/Renderer.cpp,* in here we have a set of render commands for our actual 3D scene. In here we can register 3D objects to be rendered, Begin, Update and End the current frame scene, Reset the scene. Basically, it acts as the Scene Renderer. Also, another thing to note here is that the Physics engine is integrated in the Update loop of the Renderer, where it’s being polled at every subsequent Update call for the updated positions in space of the mesh vertices.
  + *Renderer2D.h/Renderer2D.cpp,* this class holds a set of commands for drawing 2D geometry, namely varying quads (scaled, colored, rotated, textured). The only notable thing here is that this Renderer has a Batch Rendering system implemented. Effectively, holding only the VBO data for each 2D object and creates only one VAO per frame, on the Flush call, where it batches together every VBO into a single buffer, thus resulting in a single Draw call for the entire geometry.
* **OpenGL Module**
  + *OpenGLContext.h/OpenGLContext.cpp,* as previously stated for the Graphics Context interface, this is the actual implementation of the Graphics Context which handles any render API specific pipeline logic, in our case, Init which calls GLad to load at runtime into memory the OpenGL calls available on the GPU. And Swap buffers which actually instructs OpenGL to send the data to the drawing device
  + *OpenGLRendererAPI.h/ OpenGLRendererAPI.cpp,* here are the actual OpenGL renderer API calls which are called from the RendererCommand class.
  + *OpenGLBuffer.h/OpenGLBuffer.cpp,* these hold the actual implementation of the VBO and IBO objects, specific for the OpenGL render API. The VBO implementation supports both Static and Dynamic drawing.
  + *OpenGLFramebufferr.h/OpenGLFramebuffer.cpp,* the actual OpenGL implementation of the framebuffer class. Essentially it creates a frame texture to which renders a color and depth buffers.
  + *OpenGLVertexArray.h/OpenGLVertexArray.cpp,* these contain the implementation overt the VAO object. As previously stated, this holds a reference for both a VBO and IBO object.
  + *OpenGLShader.h/OpenGLBuffer.cpp,* this the actual implementation of the OpenGL Shader class. Supports loading GLSL source code, both from string values and file location. Also, the file parses have capabilities to determine the shader type given a corresponding type token that needs to be specified in the shader file.
  + *OpenGLTexture.h/OpenGLTexture.cpp,* here are defined the OpenGL specific Texture implementations. Internally it uses STB\_image to load the file into memory.
  + *OpenGLTransform.h,* here is implemented a Transform3D class, with the GLM maths library. It essentially defines a Transform class that has the TRS (Translation/Rotation/Scale) operations available and returns a model matrix.
  + *OpenGLMesh.h/OpenGLMesh.cpp,* here we have defined a Mesh object, specifically for working with the OpenGL standard. The Mesh object can be created either from raw data, meaning indices and vertex data only, or from a 3D object file on disk. The mesh itself holds a reference to its Material type, which is composed of a Texture and lighting values. Also, for applying scene lighting, the Render method takes in light position, direction and camera position.
* **Networking Module**

The networking module has been implemented as an asynchronous, passive system, where the actual server does not contain any world or client rules that would enforce, yet, the actual local client is responsible with doing the actual game world physics boundaries and this Client/Server infrastructure just replicates it accordingly to the connected clients.

Due to some complications with circular includes and forward declarations, I’ve opted with making a single header include file for this module, which effectively holds the definition for all our Networking API.

Further looking into we can see the following:

* A series of network message types, e.g.:
  + Server\_GetStatus, Server\_Ping
  + Client\_Accept, Client\_AssignID, Client\_RegisterWithServer
  + Game\_AddPlayer, Game\_RemovePlayer, Game\_UpdatePlayer, Game\_ChatMessage
* A series of network payloads to be sent:
  + ChatMessage, holds the Peer ID into the network and the text message to be transmitted
  + ClientDescription, holds the Peer ID into the network and vectors for the respective’s client position and direction in the game space
* A templated message type which contains a header type object that holds only a network message type. Also, a vector of unsigned 32bit integers representing the actual data buffer. It has its stream operators overloaded to work with writing and reading POD (plain ol’data).
* Also, an owned message which is effectively a message but which has a connection associated with it.
* A thread safe queue for storing and retrieving async messages from the either the client or server socket. Internally it works with two mutexes and a condition variable, which helps when waiting for messages to fill up the queue.
* A connection class which deals with connecting to either a client/server socket and has writing and reading capabilities, which serialize/deserialize async network messages and adds them to its owner’s message queue.
* A client connection interface which should be implemented by an actual network client implementation class. Nevertheless, it allows for Server Connection/Disconnect and can Send messages through its connection object.
* A server connection interface which offers internal functionality for connecting incoming clients, effectively adding them to a deque structure and distributes received messages to the connected clients accordingly. Also, provides appropriate virtual callbacks for allowing higher level logic definition.

* **Physics Module**
  + *Collider.h,* defines the collider class which can be composed further into the actual Bounding Volumes classes, such as AABB, Sphere, Plane. Here is also defined the Collider State, state which the Collider recalculates at every update frame, depending on the allocated frame time through collisions and a series of simple Implicit Euler Integrations. Also, the collider has its own reference to the attached Mesh, so it can access the vertices data for calculating Triangles to Point collisions. Finally, here is defined a Collision Data structure, which define relevant data to pass on further when a collision is detected. The collider types into which it can be composed into are:
    - *Plane,* has only a normal vector denoting the direction of the plane and its distance.
    - *AABB,* this is the Axis Aligned Bounding Box which defines the Minimum and Maximum extents in 3D space.
    - *Bounding Sphere,* it uses a center of the sphere and radius of it to calculate collisions with other active colliders.
    - *Object,* this class doesn’t define any other special logic beyond the collider class, yet it exists to better determine the type of collider at runtime and thus select the appropriate collision detection method.
  + *PhysicsEngine.h,* this class orchestrates the entire physics simulation. Ideally it should run on a separate thread than the one that its responsible for the rendering, and polling the simulated vertex positions at every update loop. The general idea is that it registers Colliders which then iterates through them and Updates them in space, then iterates once again through them to check for collisions and handle them appropriately. The collision detection routine is handled as a scheduling system, where the collision check is done on demand on certain data polling events, for better accuracy. The collision response function is based on a Impulse Response Model, which ideally and optimistically it would calculate the appropriate forces for each of the objects and apply them to be integrated into the Collider’s state.
* **Sanguine Editor**

This application project serves as the main Editor application where the full capabilities of the engine are used. Essentially, it defines specialized logic for working with multiple scenes and the scene hierarchy, with features for adding, removing and updating scene objects.

A specialized feature of this application is the Scene Hierarchy UI Panel which:

* Sets the active scene context from where to poll the entity collection
* In its dedicated UI panel, it displays the entire entity collection for which:
  + It can display for each component controls for the appropriate data members
* It also works with displaying the appropriate TRS widgets for the selected entity
* **Game Client**

This application is specialized for demonstrating the local multiplayer capabilities. It inherently expects that a Game Server application is running.

To validate Client connections to the Server, a Handshake type system is implemented, where, after creating a new connection, the Server sends a request with a certain integer value scrambled and the Client is responsible with sending back unscrambled, before accepting its connection.

Otherwise, it just has a default game world where a physics simulation runs locally and where other players can connect to an interact with the other connected clients.

This application also holds a Chatbox feature, in which clients can write messages to other connected clients in the network.

* **Game Server**

This is a standalone application which contains the definition for a Server class which implements the server interface mentioned in the Network module. It works internally with a mapping between the connected client IDs and their respective client description (i.e., spatial information).

Moreover, it provides actual implementations to the virtual methods for handling certain events (when a client was connected, when a certain message was received, etc.)

To note, it is required to have this application started when trying to run the Game Client application.

* **Further Development**

For the future installments of this project there a few things on my to-do list:

* Implement a Batched Rendering System
* Improve/refactor the Physics Engine
* Scene Culling routines