Can't Escape Games

TECHNICAL DESIGN DOCUMENT

DigiPen's Team Can't Escape Games

Team Members:
Albert Harley
Aleksey Perfileve
Arnold George
Jose Rosenbluth
Ramzi Mourtada

Gam 550
DigiPen Institute of Technology

November 12, 2019

Contents

Major Systems

System	.cpp folder	Description
EventManager	Managers	Main Event Bus singleton, creates all main managers, separates input and game into separate threads
CameraManager	Managers	Registers cameras and updates and screen changes
FrameManager	Managers	Controls and passes frame time data
InputManager	Managers	Runs main thread, injects input events into system
ResourceManager	Managers	Stores all pointers to resources loaded for each state
SystemManager	Managers	Updates all component systems (component behavior)
StateManager	Managers	Controls state stack and allows loading states on separate threads
ScriptingManager	Managers	Initializes and stores all lua table data. Contains all binds
AudioManager	Managers	Controls all audio channels and has access to audio resources
EventBus/Event/Delegate	Event	Low Level code for event pipeline and passing events to delegates bound to scripts
GameObject	GameObjects	GameObject Interface
RigidBodySystem	System	Updates all physics data (check Physics folder for more info)
CantMemory	CantMemory	Creates memory pools of 4K size and stores components, gameobjects, and certain resource types in those pools
Factory	Factory	Translates json information to data loaded by resource manager
AppRenderer	Graphics	Main Rendering Pipeline
DebugManager	CantDebug	Contains Main Level Editor Tool in DEVELOPER build

Libraries Used

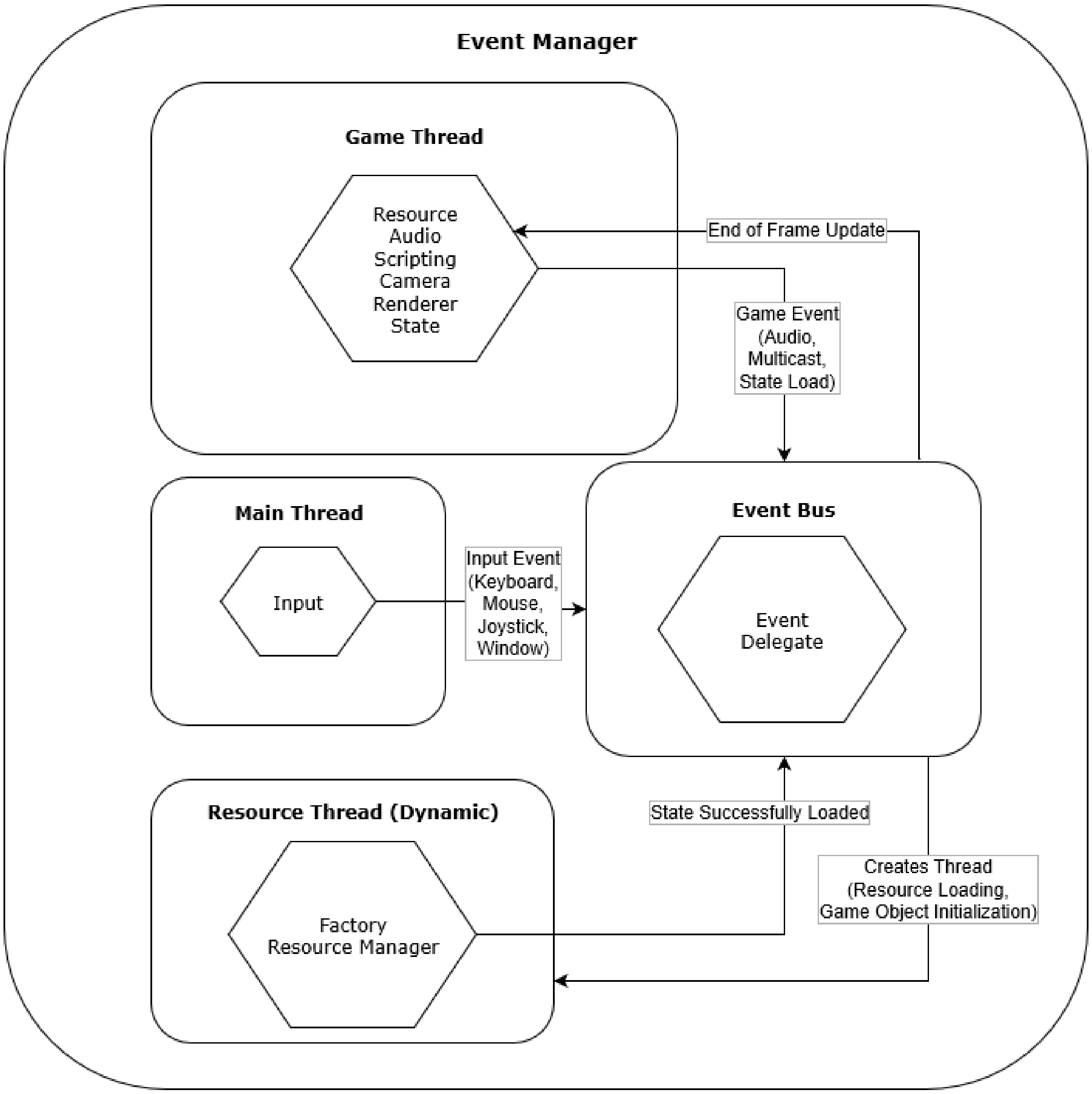
Library	Function	Description
DirectX	Graphics	Rendering/Particles/Shaders/Debug Drawing
Assimp	Graphics	Loading models and animatiosn
imgui	Debug	Level Editor/Profiler in DEVELOPER build
SimpleMath	Math	Base for all math functions
Lua (Sol)	Scripting	Scripting Player/AI
RapidJSON	.json Parsing	Parsing Object/Level/Material Files
SDL2	Window/Input/Frame Rate	Input Event Injection (controller/window…)
FMOD	Audio	Playing Audio on multiple channels
Custom 3D Physics	Physics	Dynamic Collisions using AABB Tree (GJK)
RTTR	Reflection	Used for Serialization and some engine tools

Components Used

Component	Contains
Transform	Render data such as position, rotation, scale
Rigid Body	Physics data such as position, velocity, acceleration, mass, bounciness…
Camera	Attaches a camera to the object's transform
HaloEffect	Lighting effect for halo
Mesh	Model data such as mesh arrays
Renderer	Texture data with material
Audio	Sound Effects Data such as collision audio`
Animation	Play key animation frames
Light	Customizable light
ParticleEmitter	Play customizable particle effects
SkyboxIrradiance	Skybox irradiance
UI	All UI information for scripted UI

Overall Project Architecture

Global Event Manager:


Event Manager
Game Thread
Resource
Audio
Scripting
Camera
Renderer
State
End of Frame Update
Game Event (Audio, Multicast, State Load)
Main Thread
Input
Input Event (Keyboard, Mouse, Joystick, Window)
Event Bus
Event Delegate
Resource Thread (Dynamic)
Factory
Resource Manager
State Successfully Loaded
Creates Thread (Resource Loading, Game Object Initialization)

Delegates and Multicasts

Aside from the event system, the engine supports the use of C++ delegates and Multicasts.

Delegates are a templatized wrapper class that is defined with a function or method signature and will hold all the information needed to call any method or function that shares the same signature. In the case of non-member functions, it works just like a normal function pointer. In the case of methods, it will also hold a reference to the caller Object, and when the delegate is called, it will call the member function through the object reference. It also supports functors.

Multicast are also templatized classes which allows us to hold variables that acts like "function containers" that share some signature. Example of how they are declared and called:

Declaration: Here, the first multicast will hold functions that receive a GameObject pointer and a float. The second one, functions that receive no parameters.

```
Multicast<void(GameObject*, float)> ExampleMulticast1;
```

Binding to the multicast. In this case, the binding is with a delegate, which allows for C++ to C++ communication. We also support binding Lua functions (which gives us the option to have out scripts reacting to engine multicasts).

```
ExampleMuticast2 += delegate<void(void)>::Create<ClassA, &ClassA::Method>(this);
```

In this example, *ClassA* is a C++ class, and *Method* is some member function in that class. "*this*" is a pointer to the instance of type *ClassA* that we want to call *Method* through when the multicast gets fired.

Calling the multicast. This will call every function that has been bound, in whatever order they were added. Every bound function, be it a Lua or C++ one, will receive the arguments *goInstance* and *4.34f*.

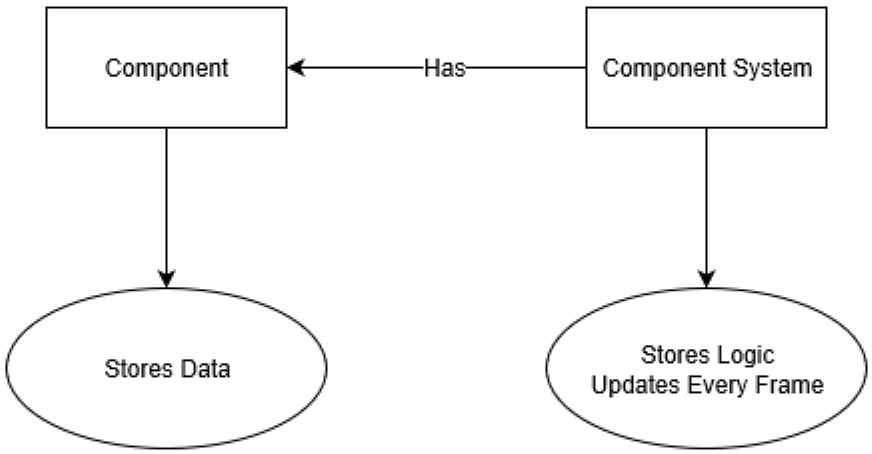
```
ExampleMulticast1(goInstance, 4.34f);
```

The main difference between **Multicasts** and the **Events** on the engine is that the Multicasts are more of a direct message towards a *particular instance*. When an object binds to a multicast, it doesn't do it towards a global event (like, button press).

For example, you can have an enemy bind a method call "*Teleport*" to a Multicast held by a Chest instance. Say Chest is programmed so this multicast gets fired when the player tries to open it. Then, as a result of this action, the enemy would teleport and attack. The important part to note is that the binding was with this Chest instance. If the player opens another chest, this enemy will not teleport.

As stated, Multicasts can be between C++ and C++, or between C++ to Lua (as in, when a collision Multicast between to objects gets fired in a C++ class, a Lua file can have one of its methods fired). There is a different kind of Multicast which we implemented in Lua which works for the pure Lua to Lua bindings. These are Lua Multicasts.

ECS (entity-component-system):



CEG engine is built under the ECS architecture. This means the game consists on entities without any behavior or distinction other than some basic data (like tags, or Id), which we call **GameObjects**. The way to make these entities distinct from one another is by adding behaviors to them in the form of **Components**. Components will mostly deal with data and have only a few lines of logic, mostly for internal dealing with this data (checking if some fields are valid, getters, setters, etc). Finally, **Systems** will be the translation units in charge of dealing with handling the logic and the communication between different components. This way, data and logic are effectively separated, which helps in making the engine handle memory in a more efficient way, while also aiming at having less cache misses (since components are smaller, it's easier to have all the data on a single memory page and achieve locality or references).

Game Objects

In CEG engine, GameObjects have a unique ID, which cannot be changed and is determined on creation, a tag which can be optionally set when instantiating (and will be used for hashing the entity for easy lookup in the scripts), and a list of all the components held by the GameObject. The components are owned by the object, so it has to clean them up when being destroyed. Because CEG engine uses a ECS architecture, GameObject is not in charge of updating the components.

The GameObject Manager handles the instantiation, and this can happen through one of the following channels: Either via the Factory creating GameObjects when the State is first created, or via queueing. This last method can also be used from the scripted components, which is very useful since it allows us to create GameObject instances from Lua and hold their references on the script. We can either create a new, empty GameObject and add the components we wish it to have on the script (we can also get this components to override whatever data we want), or create it based on a prefab (in which case it will have a list of already defined components and overrides).

When a Entity is created this way, it won't be part of the game until the next frame (when the GameObject manager handles the de-queueing of both the instantiation and the destruction containers). At this point, the manager will be in charge of calling Init and Begin (explained in the **Components** section) and registering the newly created GameObject using the System Manager.

Components

In CEG we have two kind of components. Engine and scripted components (both inheriting from the same BaseComponent class).

Engine components are C++ only classes with behavior that cannot be overwritten by the user of the engine (the parameters of the component can be defined of course). They are usually important components which either talk directly to fundamental units of the engine (like graphics or physics), are very expensive in term of computations, or both. Examples are:

- Transform component: Fundamental, as almost every entity needs one, and it is used by almost every other component for their calculations.
- Rigidbody component: Handled directly by the physics engine. Very important as it has to do with the collision calculations.
- Rendering component: Talks directly to the renderer (graphics), and thus is also important to keep it in the group of engine components.

Scripted Components are Lua-defined components which are 100% written by the end user and can do whatever they want. On the backend, they work as an engine component, since they have a C++ CustomComponent class that defines their Lua reference and a CustomSystem class which handles the update calls. They all can access their **Owner** GameObject reference, and each Entity can have as many Scripted Components as the user wants. On the Lua side of things, they have a simple structure as follows:

- Table of **local variables**, which can be overwritten from json so the same script can be assigned to different Entities with different settings. The values on the script will be used as the default values if no overwrite occurs.
- They all have an **Init** method, called when the Entity is instantiated, and a **Begin** method, which is called for every instantiated GameObject once the instantiation process of each frame ends. The idea is that Init is used for when needing information from other components within the object, while Begin can be used to look for other GameObjects and accessing their data.
- They all have an **Update** method, called every frame from their CustomSystem class. It receives a delta time parameter.
- The user is free to add as many custom methods and functions as they want in the scripted component. From within, the user can also access any other component (both scripted and engine) belonging to the Owner Entity or any other (as long as the user holds a reference to this other GameObject).
- Since you can access Engine Components, CEG engine also allows scripts to bind their methods and functions to Multicasts so they can respond to engine events (like a collision between two Entities), and to engine messages (like a button press).

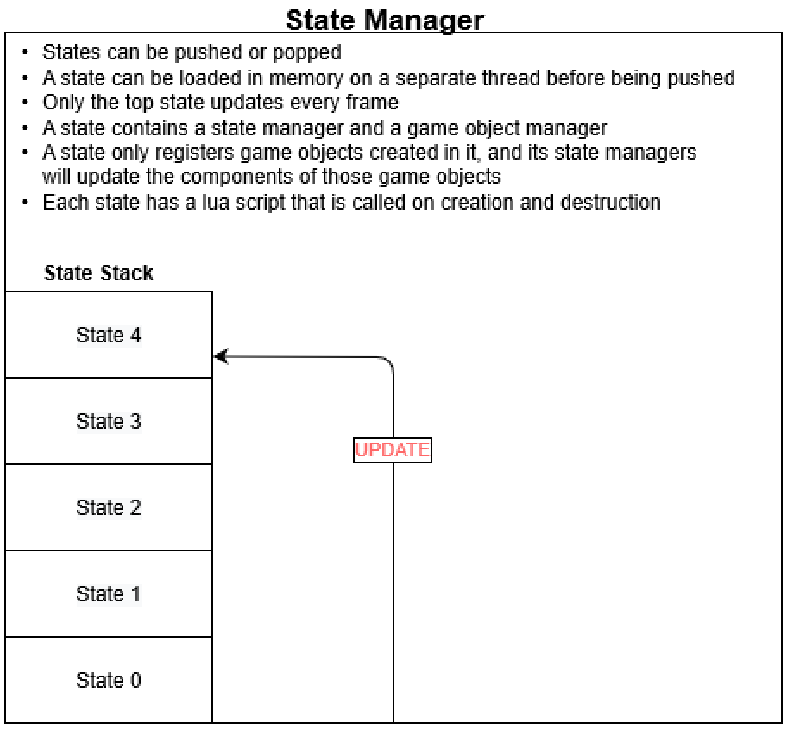
Systems

Systems are the units in charge of updating the components and handling the communication between them. The idea of this is to take logic away from components, and keep them as containers of data, and also avoid components knowing about one another (because, for example, the RigidBody component needs to talk to the Transform when being updated). This last bit means that a system is required for every time one or more components need updating. So, for instance, the transform component needs to check if it needs to recalculate the model matrix every frame, and thus it has a system. But there is also a need to update both the Transform and the Rigidbody component simultaneously (since Rigidbody will affect position and rotation), and thus there is also a system that will handle that interaction. The fact that one or more systems can touch upon the same component (Transform in this example would be affected by two systems) means we also need to be careful when deciding the order in which systems are updated, since it does affect the resulting behavior.

This results on systems being classes created as they are needed. Contrary to components, we don't create systems per Entity. Instead, we have one per **required components** (so, if one system requires both Transform and Rigidbody, there will only be one such instance in the state). And once a GameObject is instantiated, it will be sent to the **System Manager,** so it is subscribed to the right amount of systems, based on the components it owns.

The **System manager** will hold and handle the systems in the current state, so once per frame, when the state updates, the manager will call Update on all the system on a predefined order, and each system will iterate through the Entities registered to them.

State Stack


State Manager
• States can be pushed or popped
• A state can be loaded in memory on a separate thread before being pushed
• Only the top state updates every frame
• A state contains a state manager and a game object manager
• A state only registers game objects created in it, and its state managers will update the components of those game objects
• Each state has a lua script that is called on creation and destruction
State Stack
State 4
State 3
State 2
State 1
State 0
UPDATE

In CEG engine, a state is a fundamental logical unit which represents the current context for the game. In other words, a game will always be running a state, and so, with just one state you could build a whole game (albeit, limited in many ways). A state will have their own set of managers, which will hold all the data and logic that makes this state different from others. A game level is going to be a state, but a pause menu, a title screen, and even an inventory window, all those are states too.

In every moment, a state will be running (which we call the current state), on top of a stack (the current stack). Every frame, states need to be **updated** and **rendered**. The way this works with the stack is that only the top state will be updated (thus preserving the context of the states that are currently sitting below the current state in the stack), but all states are going to be drawn from bottom to top (so if a pause state is placed on top of the game level state, we can still see stuff from the game drawn under the pause state's UI).

Whenever we need to switch to a new state, but without exiting the old one (because we may want to come back to it), a new state is pushed on top of the stack, and it becomes the new current state.

When we are done with this state, it can be popped from the stack, and then the state below it becomes the current state.

Finally, when we need to switch game levels, thus not requiring preserving the current state in any way, we switch the current stack for a new stack, on which we push the newer state. This is usually done when switching from one game level to the next. Information can be passed from one state to the other, since they all have a script component that facilitates reacting to the switches and adding custom behavior on top.

State

Each state will have data and local managers that will distinguish them from others. The managers a state is responsible for creating and destroying are as follow.

Factory

This is a static class used by the state to create all the instances it needs before it starts running.

Game Object Manager

This manager is the one tasked with the creation, management, and destruction of all the entities of the state. At the start, it will be filled with the GameObjects created by the Factory. After this, it will handle creation and destruction of GameObjects at the start of each frame (both operations are handled via queue of commands). Every frame, the state will call Update on its GameObject Manager, which in turn will call update on all its underlying GameObjects. Apart from this, it also handles the hashing of the tagged entities, allowing for a constant time lookup of a GameObject in the scripts.

System Manager

Finally, there is the system manager, which will hold all the relevant systems, and will once per frame update them. As stated in the Systems section, this manager will iterate through the systems on a predefined order. The CustomSystem, which handles the update for the scripted components, will be called first, so this means all user defined updates will happen before the engine components updates (so the data they affect will be the data from last frame).

Also, as stated briefly in the same section, there will be only one instance of any given system per state, and that instance will hold a list of every GameObject registered to it. This means that, if a certain Entity has the components required by the system, it will register to it on creation, and when the system updates it will apply its changes to the components held by that entity.

When the state is destroyed, the System Manager and all its systems are destroyed as well.

Scripting

Scripting is done in Lua version 5.3, and we are using SOL version 3.0 as the library for handling the bindings. The Scripting works by defining one Lua state, which will hold all the scripted components. All the setup happens on the Scripting Manager, as follows:

First, a Lua State reference is created, and all the external libraries that we may need to call from the scripts are loaded into the state. Then, some global functions are added into the state (since all components share a state, these will be callable from all components). Finally, we handle the binding of every class that we may want to call from the scripts. This means that, if we want to call a function from a class A that has arguments of type B, we need to bind both A and B on the Scripting Manager, in order for Lua to be able to interpret those two fields correctly.

Both the scripted components and the scripts defined for each Game State consist on Lua tables, which are added into the main Lua State, and this tables reference is held by a C++ class (either a CustomComponent or a State). In the case of the Components, in order to allow for different GameObjects to point to the same script but have their own data reflected on the variables, we call a Lua global function we defined early called deepcopy, which will make a deep copy of the script, thus allowing the previous behavior.

Also, we defined a **scripted multicast** as a global type in Lua. This allows the user of the scripting to do stuff like the following: If you are inside a custom component of GameObject A, and have a reference to GameObject B, you can access one of its scripts, and bind a function of yours to a Multicast variable belonging to that scripted component. This means that, whenever B calls that multicast (with the operator parenthesis), the function binded from A will be fired. This is not to be confused with the **C++ defined Multicasts**. This is a Lua to Lua interaction, while the other allows for the same behavior, but between C++ classes (or between C++ and Lua).

All this allows us to have a self contained system for scripting, in which you can add as many scripts as you want for an Entity, and also create and reuse the same script for multiple instances, each with their own different behavior (since they can all have different starting data for the script's variables). The **scripting Manager** is a global manager that is created and destroyed with the game process itself.

Function calls

To enforce a structure of the states, there are some functions that will be called from the different actors that influence the process. For the scripted components, some of these calls are Init, Begin, and Update. Init and Begin are called from the GameObject Manager on instantiation of a new Entity. This call is performed on the CustomComponent C++ class, which holds the reference to the Lua table, and will make the actual script call. Every Lua call is made on a try-catch block, in order to make it so any scripting error does not crash the engine, and only generates errors on the logging system. Likewise, Update is called from the CustomSystem which has a reference to the CustomComponent of each registered Entity, and thus can also get the table reference and make the call to update.

For states, the calls in question are OnCreateState and OnExitState, and are called from the State C++ class, which holds the Lua table reference. Again, using a try-catch block. The only instance in which scripting should crash the engine is when the scripts have errors that make it so they cannot even be loaded into the engine (since the script will crash before reaching the point in which it can return the reference to its table).

Graphics Overview

Background

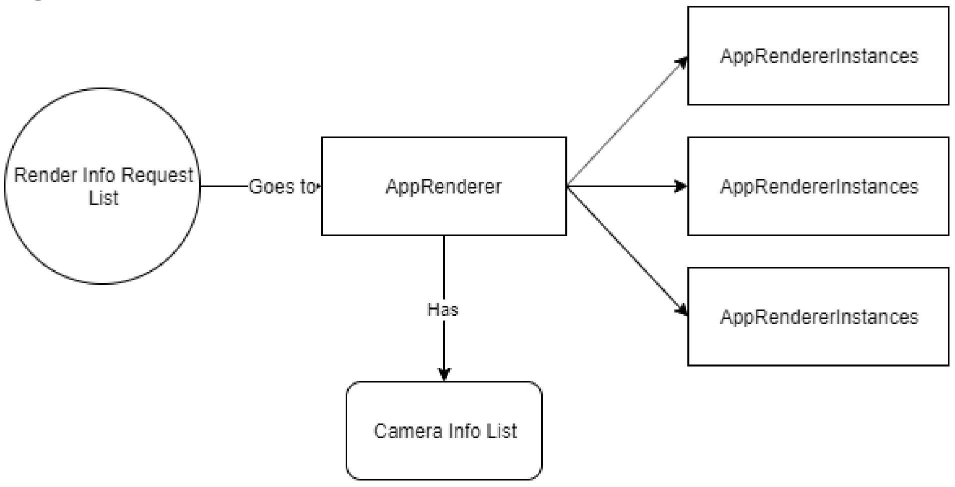
Can’t Escape Games’ (abbreviated as CEG for this section) graphics engine Can’t Escape Games’ (abbreviated as CEG for this section) graphics engine use DirectX 11 API for communicating with the lower level GPU kernel. In CEG’s graphics engine, the lower level DirectX 11 api call is abstracted away and not used anywhere around the codebase except for selected header/source files. What this mean is that the higher-level render logic uses user-defined API function & class. The purpose of doing this is so that the graphics engine is ready for multiple lower level graphics API implementation rather than being constrained to just DirectX 11 API. This also allows for faster feature implementation since the user-defined function/class can automatically abstract away DirectX 11 API function/member variable that is usually never changes throughout the codebase yet it still a necessity to be called or initialized to utilize DirectX 11 API properly. The example is **D3D11_SAMPLER_DESC**, a description struct to initialize a new sampler state. DirectX 11 API defined **MinLod** & **MaxLod** variable as a limitation on the range of mipmap level that a sampler state can sample from, however in most cases you always want the lowest mip level to be sampled (0 in this case) and have access to the highest mip level as well. Being able to abstract some of this code helps make the CEG’s graphic engine’s code base a lot cleaner.

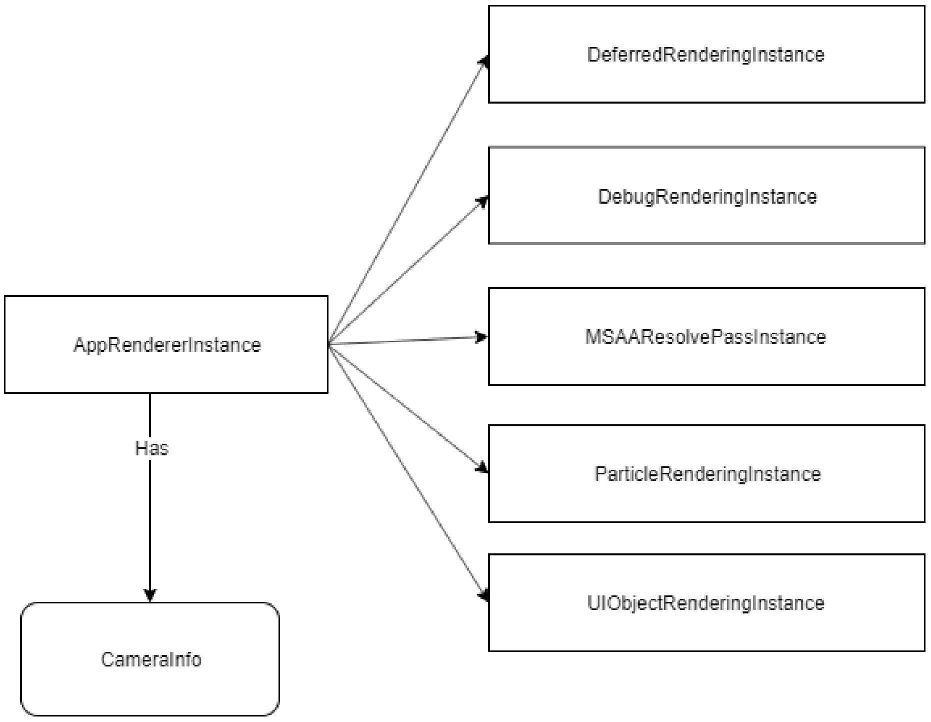
The other major abstraction is the **GraphicsPipeline** class defined in the CEG’s lower level graphics code base. The **GraphicsPipeline** class contains a shader pointer, blending state, depth state, rasterizer state, primitive topology type, and vertex input layout. When we bind the pipeline with a single function call **cmd_bind_pipeline**, the inner DirectX 11 renderer will bind the defined shader, blending state, depth state, rasterizer state, primitive topology type and vertex input layout all together. Once again, this is in order to make the main render loop code logic cleaner.

Asset Loading

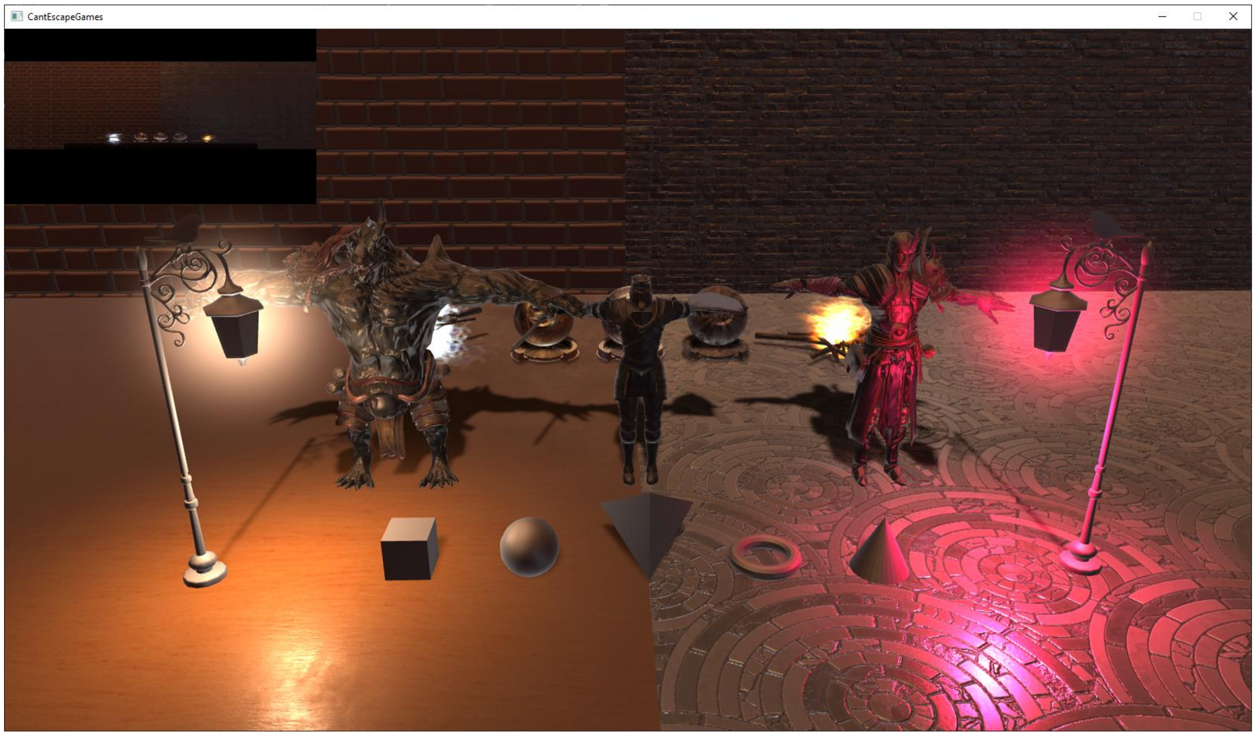
There are two ways for loading assets in CEG’s graphic engine. The first is the usage of **DirectXTK** api function that Microsoft provided as a helper function for Directx11 API. **DirectXTK** provides a function that loads WIC texture from hard disk. It also automatically generates mipmap and appropriate shader resource view. However, **DirectXTK** API does not provide method/function to load HDR image and since CEG’s graphic engine has a feature that utilizes HDR asset (for Image Based Lighting feature, for example) it used **stbi_loadf** function from **stb** open source 3rd party library that loads raw float* data and call the necessary DirectX 11 API to initialize mipmaps and shader resource view.

High Level Overview





AppRenderer is the graphics manager, it holds all the render info request list and will appropriately process all the requests based on the render loop logic. **AppRenderer** also holds a list of render camera list and the reason for this is to provides way to render multiple scene from different projection & look settings from different camera and display it all at once in the window screen (or in D3D11 terms, swap chain present). Here is an example:



Above shows the main camera rendered in perspective projection matrix while the top left corner shows the scene rendered in orthographic projection. This flexibility will allow some game design genre immediately prototype-able, such as local multiplayer game genre, or it can be used as a “radar” or “mini map” in games like first person shooter or real time strategy.

The rendering of this multiple render targets is handled by **AppRendererInstance**. This class contains all the necessary rendering class features in the CEG’s graphic engine and it also holds a pointer/reference to a camera info. **AppRendererInstance** will have its own “main” render target and will pass through all the render pass technique in the set order. Each of this render pass technique relies on unique camera info, hence they share the same class syntax name “instance”. For render pass that doesn’t rely on unique camera info, **AppRenderer** will execute those render pass.

Main Graphics Features

Lighting and Shading

CEG's graphics engine features two lighting types: point lights and directional light. Directional Light is the light that is responsible for casting orthographic shadow on the render scene. For efficient point light shading calculation the engine utilizes deferred shading. Deferred shading lighting model allows the graphics engine to render hundreds of point lights while keeping reasonable frame rate. Deferred Shading contains three passes: The first being the geometry pass, where we render the depth, 3D normal direction, albedo color, specular, roughness, metallic value into three separate textures/render targets. Second is the ambient shading pass, where we apply the directional light's lighting calculation together with the shadow calculation. The third is the point light pass, where we render multiple spheres geometry (position & scale of these spheres are determined by the transformation data of each point light) around the scene and those spheres geometry will determine the new shading color of the scene.

Both lights types utilize PBS (Physically Based Shading) and Image Based Lighting algorithms. The PBS part includes Cook Torrance BRDF lighting model that allows the rendering of material with different level of roughness and metallic value. The Image Based Lighting implementation allows the scene to be lit globally by the environment's irradiance and specular factor in the form of skybox.

CEG's graphic engine supports normal mapping, parallax mapping, and some other PBR textures material (such as roughness texture and metallic texture). Normal Mapping allows surfaces to have better detailed lighting to them while parallax mapping elevate this even further by producing realistic bump to the surface based on the mapped gray-scale height map.

Shadows

The graphics engine also has filtered soft moment shadow map. Moment shadow map stores 4 variables in the shadow map and it used compute shader to blur all four variables to produces 4 stochastic moments. It also helps reduces shadow acne and peter panning, both being the common problems for shadow maps and ultimately it provides soft shadow when rendering 3D meshes.

Custom Artist Defined Model Rendering

CEG's graphic engine supports the loading of artist skinned model by utilizing assimp, a 3rd party source library that can read many popular 3D models file format. It can render models with textures/art assets defined by artist.

Post Processing

The graphics engine has MSAA implemented as its main anti-aliasing solution and it resolves the MSAA samples with its own custom shader. Instead on using the standard D3D11 box filtering resolve, the graphics engine employed a cubic function filtering based on additional variable called "filter size". This implementation allows higher anti-aliasing quality with small amount of MSAA sample count than simply using the default D3D11 MSAA box filter resolve.

Particle Rendering
CEG's graphic engine has billboard particle rendering implemented. Game designers can edit the particle emitter properties as however they like, for example its cone angle emission for yaw or pitch orientation. The particle system is implemented on the GPU side, specifically its lifetime is managed on the geometry shader. D3D11 supports stream-out vertex buffer. Stream-out vertex buffer is a vertex buffer that can be edited immediately in the runtime in geometry shader. This allows for a reasonably fast particle system. It is also cheap in memory since the vertex buffer only stores one primitive point for each particle quad and in another geometry shader each primitive point will produces 4 additional point to produces a quad based on the camera's view matrix.

Miscellaneous
The other extra graphic feature is the light halo effect. The light halo effect is a spherical volumetric effect that is used to simulate light sources scattering around the medium when an eye is looking directly towards it. Light halo effect can also be used when a game wants to specifically make certain object glow more than usual for game design purpose. The light halo effect is the part of the deferred shading algorithm, as it also uses drawing of 3D sphere model. The difference is that it uses some form of ray casting to get two time variable, t1 and t2. T1 and t2 will determine which intersected positions of the ray when it was casted from the camera to light halo effect. It will then integrate the density function to get the final brightness color of the halo effect.

CEG's graphic engine support debug rendering. It can draw bounding boxes, lines, and spheres, all in wireframe primitive.

Animations
CEG engine handles Keyframe animation via the Animation Component. On it, you can load animations, and they will be stored with information such as a name, duration, ticks per second and if it loops. All loading of animations is done using Assimp, and our animations are all in FBX format.

Keyframe Animation
For the keyframe animation, we have a set of structures that store all the information needed in order to animate the model. This includes a Bone struct, which holds information such as the bone's name, the children's of this joint, and also the matrices we need in order to transform the vertices. There is also a Animation Struct, which will read all the animation information, such as the list of keyframes information for each bone that is affected by the animation.

Animation Data
Other relevant data for the animation is a vector of **animation events**. This is an array of Multicasts which has length equal to the duration parameter. The idea is that, from the script, different functions can be bound to this Multicast, and then when that time is reached on the animation, the Multicast will fire. We use this, for example, to put sounds on the animations (it can also be used to activate a hit collider just during a part of the punch animation).

This **Animation Component** will hold an **Animator Controller**, which will be a State Machine that allows us to define fully closed animation cycles which we can easily influence over from scripting. The way this works is that, from the script, you can access the Animation Component and create **AnimStates**, which will hold one animation each. Then, the user can define **Transitions** per state, which will need a target state as parameter, and a transition duration. These transitions will have a set of **conditions** and whenever one of these conditions may be reached, the state will loop through its transitions, and jump to the first one that returns true on all the conditions.

By default, the Animator Controller will set the first AnimState as the entry point for the state Machine, and if no other state exists, it will remain in that one for its lifetime. Note that animations can only be ran through a state machine.

There are two ways of exiting an AnimState.

One is if the animation ends and does not loop. In this case, the state will be notified of the animation end, and will check to see if there are any transitions defined. If there are, it will check if any returns true on all conditions and jump to the first that does. If it doesn't have any transition, or no transition returns true on all conditions, it will stay on the current state (but will not replay the animation).

If the animation loops, then the state won't check its transition unless forced to by some external notice. This external notice is fired from script.

Conditions are handled via a concept of Triggers. These work by defining the trigger as a parameter which one can activate via scripts (calling *SetTrigger(triggerName)*). Every time a script activates one, that trigger gets marked as true on an unordered map on the Animator Controller. At the end of that same frame, the Animation Controller will check through its **current AnimState** list of transition´s conditions to see if any returns true. Independent of whether this happens, at the end of the frame all triggers in the map are set to false. This way, triggers act like a switch that once turned on, goes back to the off position on its own.

Blending

Finally, transitions blend the animation from the old state with the animation of the new state. This makes it so attacks and different animations can be interrupted in the middle to perform different actions, and it won't like there was a weird cut on the animation.

Quaternions

To handle the interpolation of the animations, we defined a Quaternion class, and use the spherical linear interpolation (SLERP).

Physics

Frame control

- Is paused: steps one frame forward
- If not pause update normally

Broad phase (Dynamic AABB tree query)

- Update bounding volume for each object
  - Rotate AABB defined in a object space for the mesh
  - Construct an AABB of that rotated AABB
- Reinsert into the tree this AABB
  - Remove from the tree old AABB with preinitialized ID corresponding to this object
  - Add to the tree newly constructed AABB
- Self-query the tree to detect any possible collision between any bounding volumes (AABBs constructed earlier)
  - Query left side of the tree to the right side
  - Query left side of the tree
  - Query right side of the tree
- Delete duplicate query results
- Check with the collision table
  - If the objects not mapping to each other in the table then remove query result
- Validate last frame contacts if they still relevant, for each contact keep contacts if:
  - The two objects in contact still penetrate
  - Coordinate of contact for object A did not change much
  - Coordinate of contact for object B dd not change much

Narrow phase (for each query result from broad phase)

- Construct support shape (OBB collider)
  - Keep reference to transform, global scale, rotation, local scale (if model is not from -0.5 to +0.5 space)
- Run GJK algorithm to check collision between convex shapes (OBB in our case)
  - Calculate search direction
    - Global position of object A I position of object B
    - If this direction is zero vector then initialize to arbitrary (1,0,0)
  - Compute supports for the objects (furthest point in the support shape defined earlier in the given direction)
    - Object A in the direction
    - Object B in the – direction
  - Initialize simplex with this CSO (Configuration Space Obstacle) point
  - Run 100 times loop below
    - Identify in which Voronoi region is the origin in in respect to simplex
      - Calculate closest point using barycentric coordinates
      - Calculate new search direction from closest point to the origin
    - If the closest point that was found Is close enough to origin (within 0.001 distance)
      - Fill the simplex to tetrahedron

            - If the simplex is one point look for another point in principal axis directions and add it to simplex, continue filling simplex with 2 points
            - If simplex is two points expend simplex by searching for new point in principal axis directions and adding it to simplex, continue filling simplex with 3 points
            - If simplex is three points expend simplex by searching for a new point in the direction of the normal of a triangle that is formed by these 3 points and adding it to the simplex
            - Fix tetrahedron to be in CCW ordering
            - Terminate the GJK with the true result that the objects penetrate
        - Search for new support point for the object in the new search direction
            - If this new point is no further then the closet point then terminate GJK with false, there is no collision
        - Add the new point to simplex
- Run EPA if GJK returns confirmation that OBBs penetrate
    - Start with the tetrahedron that GJK terminated
        - Init the 4 triangles (faces of the tetrahedron)
    - Loop 50 times:
        - Find closest triangle to the origin
        - Calculate for next support point in front of the triangle away from the origin
        - If new point is now further away from the origin then this triangle
            - Store this point as point of contact
            - Store distance from origin to this new support point as depth
            - Store normal of the triangle as a normal of the collision
            - Terminate EPA
        - For each triangle if it is visible from the currently found support point
            - Add 3 edges from the support point to each point of the triangle
            - Remove the triangle
            - Initialize and add the triangles defined by new edges
- Process new contact that EPA returns
    - Check if collision manifold already exist for these 2 objects (if they were already colliding last last frame) and if so
        - Loop through all contacts in manifold and see if this new contact is far enough away from the each contact, and if so
            - Build constraints for new contact
            - Normal constraint, 2 friction constraints
            - Add contact to manifold
            - Keep only up to contacts in the manifold
            - Find deepest penetrating contact
            - Find furthest away contact point from deepest penetrating
            - Find furthest contact away from the line formed by the 2 contact points found earlier
            - Find fourth point away from the triangle formed by previously found points

    - If this is new collision manifold
        - Add new collision manifold to the list of the collision manifold
        - Build constraints for new contact
        - Add contact to this new collision manifold

Update velocity

- Calculate gravity force
- Calculate air drag force
- Sum forces, divide them by mass, multiply by dt and add to last frame velocity

Solve constraints

- For each manifold (50 times, Projecy Gauss-Seidel)
    - Solve for normal constraint
    - Solve for both friction constraints
    - Update velocity with each constraint

Update position

- Position = position + velocity * dt

UI

UI is divided into 2 part

C++

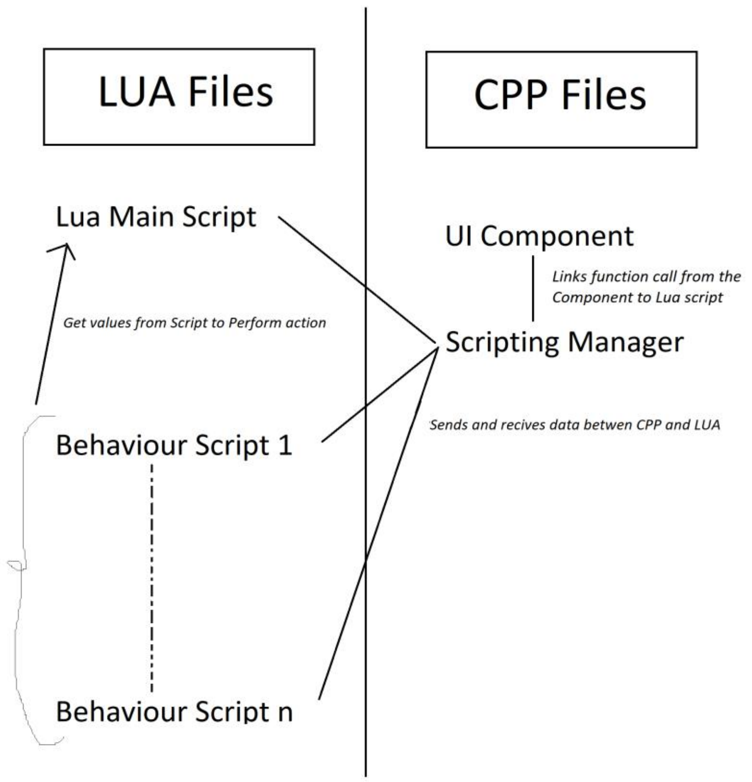
The UI Component in C++ will have the value read from Json files (Json files store each component values for a particular level). LUA files can directly read the values from the respective components and perform a common expected behavior. Examples:

UI Object script which is serialized and stores values related to (Affine animation, Button, Slider, etc..)

UI system script loops through the UI objects and sets an ordering sequence for rendering them

Lua

There is a Main LUA file for every state in the game stack, and this LUA file acts as a parent and perform action such as decisions of which states to load next, or change values for components of other objects. In addition, each script has a base UI component script that reads values from its C++ counterpart, and performs a certain behavior. We currently support behavior for UI Buttons, Slider (volume), and Drop Down (window resolution selection


LUA Files
CPP Files
Lua Main Script
UI Component
Links function call from the Component to Lua script
Get values from Script to Perform action
Scripting Manager
Sends and recives data betwen CPP and LUA
Behaviour Script 1
Behaviour Script n

Audio

All audio components will communicate with the audio manager and trigger certain SFX. The audio manager uses resources that are pre-loaded into the resource manager. We currently have two channels we play audio on, which are music and sound effects. This way we can manipulate effects on every channel. As our engine contains 3D graphics, we plan on rolling out 3D audio features into our audio pipeline. FMOD initializes the audio playback to run on a separate thread. We only call any audio playback events at the end of the frame using the same event system explained previously.

Debugging Tools and Editor

Our debugging tools are created in the developer and debug builds. The debug build follows the typical visual studio debug build and is used mainly for breaking within code and observing local symbols.

The developer build runs in release mode but adds certain macros in engine code that allow for the debugging tools to show up.

Current Debug Tools include:

- Memory Profiler: Show addresses of all memory pools and contained data
- Graphics and Physics flags for enable/disable certain features
- Material Generator: Generates a material .json fie based on the engine’s expected format.
- Level Editor:
    - Allows for pausing the game and frame stepping
    - Create, select, delete, and modify game objects and their components
    - Load/unload specific resources
    - Generate a level file containing the current level in it’s current state

Memory Management

We perform pool allocations by type. Any class type can have its own memory pool, and allocation is performed following a free list method. Alignment for each class type is also ensured in those allocations. At the current stage, we use the pool allocation for the following:

1- All components (each component type has its own pool)
2- Game objects
3- Textures
4- Models
5- Materials
6- Script Tables

Localization

We are using csv files per language. Since we only care about text rendering data, each language csv file stores an English mapping of a certain text to its respective language translation. Text rendering is still under progress, and we will be able to showcase different language localization once this feature update is completed.

Coding Methods

File Naming

Class Definitions: camel case i.e. GraphicsManager
Function Definitions: camel case i.e. **PlayMusic(const char* name)**
Class member variables: Hungarian notation as follows:

- members start with "m_" **m_position**
- member pointer starts with "m_p" **m_pTexture**
- camel case for followup name: **m_pTextureFrame**

File Directories

All parts of this project were compiled into one visual studio solution containing three projects

1. The main engine directory is CantEngine, compiled into a dll for game. Subfolders within the engine have names that should be self-explanatory.
2. Debug Library is CantDebug, only compiled in Debug/Developer builds. Allows for level editing and profiling. Contains all imgui calls.
3. The game directory is Albot (named after last semester's game). This contains all the asset folders and script files, organized in a fashion fit for the engine with a few rules (including certain shader files and script files). Contains a .cpp for initializing the engine with screen resolution, a flag for fullscreen, and the startup level that is loaded.

Code Documentation

DOxygen style commenting in main interfaces of commonly used header files. A documentation folder is provided with the submission. It will contain a documentation link for

Source Control

We are using Digipen's git source control on our cant_escape_games server. Each member develops software on their own branches and push it to master. Everything is rebased onto master, giving us a clear backtracking for all code development by the team.

UPDATE: After the engine milestone, we moved our repository to GitHub for better Code Reviewing. We used the Pull Request feature for performing these code reviews. All commits to master can be seen on the Team's GitHub link:

https://github.com/ramzimort/CantEscapeGames

Bug/Milestone Tracking

We keep our weekly objectives on a Trello board, and update it weekly during our live meetings. All bugs encountered during development have their own board and are added on Trello as well.

Communication

All team communication is performed on Slack, with different members and topic pages. We believe this helps each member focus on their own tasks without being sidetracked with other member tasks. We have at least two people working/discussing every aspect of the engine, as having an extra hand on any task usually helps in finding better/faster solutions.