# Report for Game Engine Design and Implementation

Game Engine Programming Unit 2019

By: Gabriel Mateus Camargo Lima

Date: 17/01/2020

Student number s5006074

## Table of Contents

Introduction and Overview of Project……………………………………3

Program Design and Implementation……………………………………3

- Application Class
- Components and Entity system
- Mesh renderer class
- Transform and Camera Class
- Material and Mesh class
- Resources Class and Resource class
- Keyboard and PlayerControl
- Box Collider Class and Environment Class
- GUI Class and Exception class

Analysis and Conclusion

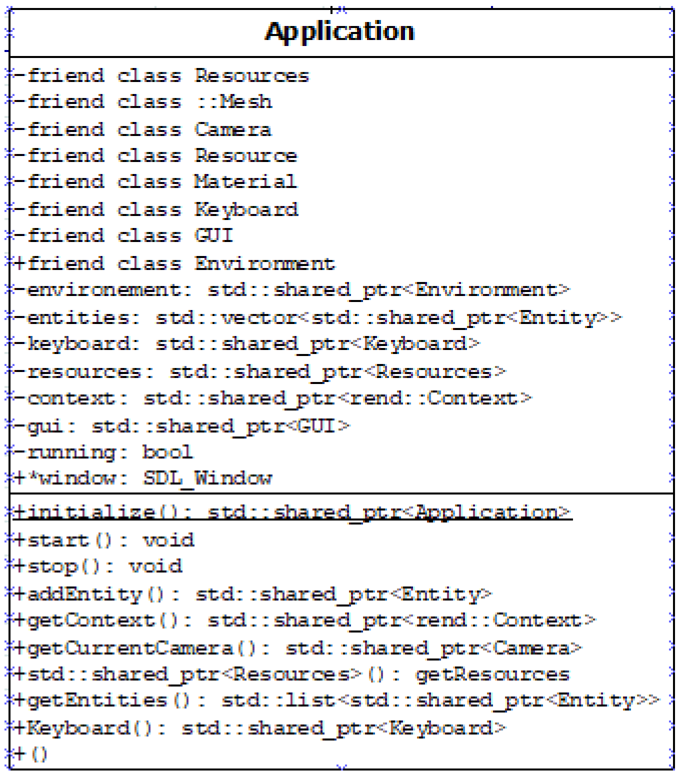
## Introduction and Overview of Project

The assignment goal is to develop a 3D game engine to showcase the architecture of a game engine and create a simple game or tech demo to show case the engines features. The engine that has been created will provide a structure to help new users create games and enjoy making them. The engine will involve a resources class which will handle the loading aspect of the engine, a components class which can be added to entities in the scene, a shader class to create shaders and display texture and meshes to screen, a entities class to add game object to scene, a input class for keyboard, a GUI class and Application class which will be the core of the engine.

## Program Design and Implementation

### Application Class

Figure 1:



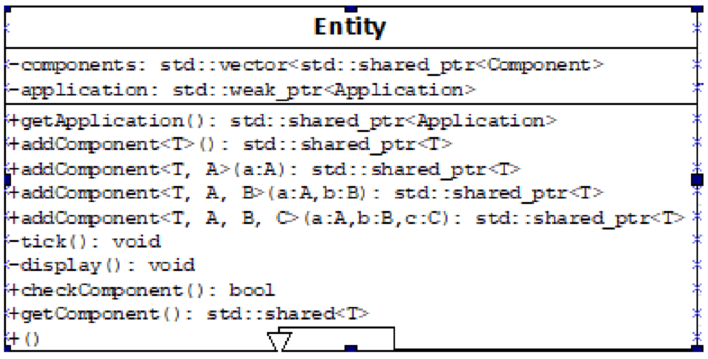
```
Application
-friend class Resources
-friend class ::Mesh
-friend class Camera
-friend class Resource
-friend class Material
-friend class Keyboard
-friend class GUI
+friend class Environment
-environement: std::shared_ptr<Environment>
-entities: std::vector<std::shared_ptr<Entity>>
-keyboard: std::shared_ptr<Keyboard>
-resources: std::shared_ptr<Resources>
-context: std::shared_ptr<rend::Context>
-gui: std::shared_ptr<GUI>
-running: bool
+*window: SDL_Window
+initialize(): std::shared_ptr<Application>
+start(): void
+stop(): void
+addEntity(): std::shared_ptr<Entity>
+getContext(): std::shared_ptr<rend::Context>
+getCurrentCamera(): std::shared_ptr<Camera>
+std::shared_ptr<Resources>(): getResources
+getEntities(): std::list<std::shared_ptr<Entity>>
+Keyboard(): std::shared_ptr<Keyboard>
+()
```

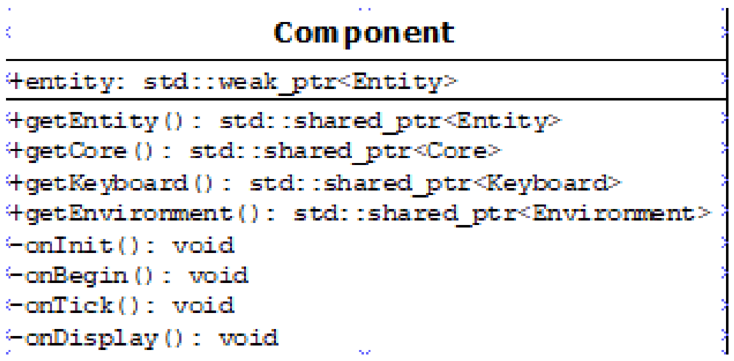
The game engine needs a core, and the application class will suffice the role. The application class will be initializing all functionalities of the engine such as the engine itself, entities, components, and SDL window. Figure 1 shows the diagram for the application class, the initialize function is what the user must call to start the engine, and within this function it will create references to other classes and their components, and having access to them all.

The application class is also responsible for creating the SDL window, a context for OpenGL and creating references to other classes. In the start function a while loop is created which will contain the events that occur in every frame, in this loop we will be checking for key inputs by users and storing them into a vector later explain in keyboard class. The OnUpdate and OnDisplay function are also called in here, so other entities created are able to access them, the application class creates a list of entities and adds entities into the scene and returns pointers to other classes.

## Components and Entity system

Figure 2 and 3:


Entity
-components: std::vector<std::shared_ptr<Component>
-application: std::weak_ptr<Application>
+getApplication(): std::shared_ptr<Application>
+addComponent<T>(): std::shared_ptr<T>
+addComponent<T, A>(a:A): std::shared_ptr<T>
+addComponent<T, A, B>(a:A,b:B): std::shared_ptr<T>
+addComponent<T, A, B, C>(a:A,b:B,c:C): std::shared_ptr<T>
-tick(): void
-display(): void
+checkComponent(): bool
+getComponent(): std::shared<T>
+()


Component
+entity: std::weak_ptr<Entity>
+getEntity(): std::shared_ptr<Entity>
+getCore(): std::shared_ptr<Core>
+getKeyboard(): std::shared_ptr<Keyboard>
+getEnvironment(): std::shared_ptr<Environment>
-onInit(): void
-onBegin(): void
-onTick(): void
-onDisplay(): void

The entity class is contains all the different functionalities in the engine, storing all the components using a list within this class. The entity class contains a reference to application class and a reference to itself, which enables components to find themselves when being called.

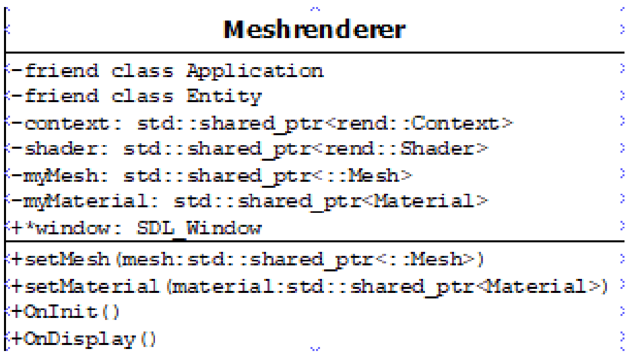
A template was used to add components to enable users to create more components freely with different parameters, as before, the process would be very inefficient and more complex as users would have to create various functions with different parameters over and over again. The template allow users to add components with no parameters, and as many as 3 different types of parameters, as it can extinguish which data type is being passed, making the progress of updating and managing the engine more efficient.

A getComponent function was created to check for components in the scene by iterating through the list of components and returning the component if found, if not return an exception errors telling users component isn't found. A virtual Display() and OnUpdate() function has been created so other classes that are inherited from components can override them.

The component class is a base class where each new component will inherit from, this allows components to access up the hierarchy, through getters and references. An example is finding other components and accessing the variables needed inside the component or locating in which entity it is attached to.

## Mesh renderer Class

Figure 4:

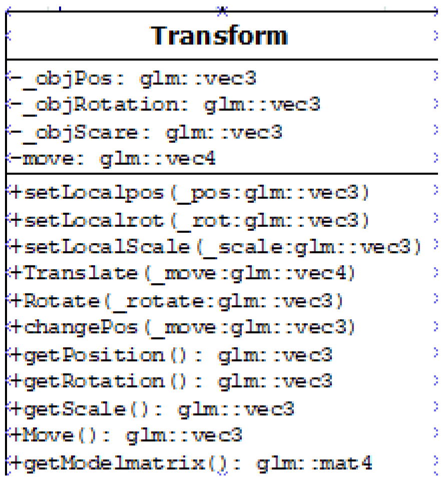


```
Meshrenderer
-friend class Application
-friend class Entity
-context: std::shared_ptr<rend::Context>
-shader: std::shared_ptr<rend::Shader>
-myMesh: std::shared_ptr<::Mesh>
-myMaterial: std::shared_ptr<Material>
+*window: SDL_Window
+setMesh(mesh:std::shared_ptr<::Mesh>)
+setMaterial(material:std::shared_ptr<Material>)
+OnInit()
+OnDisplay()
```

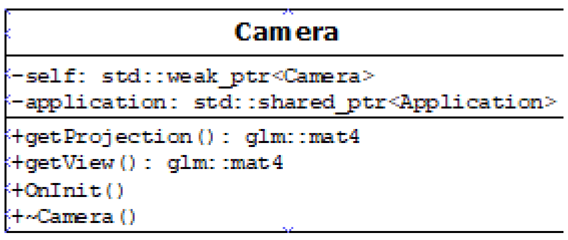
The Mesh renderer class handles the display of the engine, it will access the context initialized inside application, and create a shader. Inside this class all the set Uniform are done for the shader, and from this class we are able to set Meshes for objects and Materials and display them. The mesh and material used are copies of the original one created and read in the mesh and material class using the rend library.

## Transform and Camera Class

Figure 5 and 6:



```
Transform
-_objPos: glm::vec3
-_objRotation: glm::vec3
-_objScare: glm::vec3
-move: glm::vec4
+setLocalpos(_pos:glm::vec3)
+setLocalrot(_rot:glm::vec3)
+setLocalScale(_scale:glm::vec3)
+Translate(_move:glm::vec4)
+Rotate(_rotate:glm::vec3)
+changePos(_move:glm::vec3)
+getPosition(): glm::vec3
+getRotation(): glm::vec3
+getScale(): glm::vec3
+Move(): glm::vec3
+getModelmatrix(): glm::mat4
```



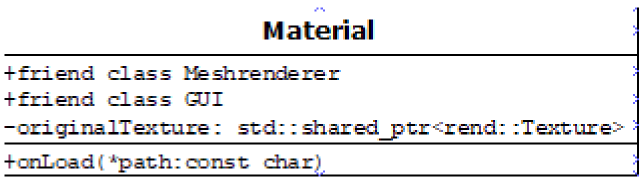
```
Camera
-self: std::weak_ptr<Camera>
-application: std::shared_ptr<Application>
+getProjection(): glm::mat4
+getView(): glm::mat4
+OnInit()
+~Camera()
```

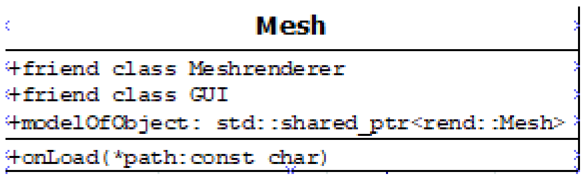
The Transform Class is used to store and set the local position of the characters in the scene, this class will also allow us to access the objects model matrix and change its rotation and position. The transform class will be used by the camera class and player control struct, in the camera class we will be using the getModelMatrix in order to create our view matrix and our projection matrix for the camera.

The camera class is where we create our view and projection matrix using the transform class getModelMatrix function where we create a matrix that can be used, we then initialize our camera by getting the current one from a list of cameras stored in Application class and return the projection and view matrix calculations we created.

## Material and Mesh class

Figure 7 and 8:


Material
+friend class Meshrenderer
+friend class GUI
-originalTexture: std::shared_ptr<rend::Texture>
+onLoad(*path:const char)

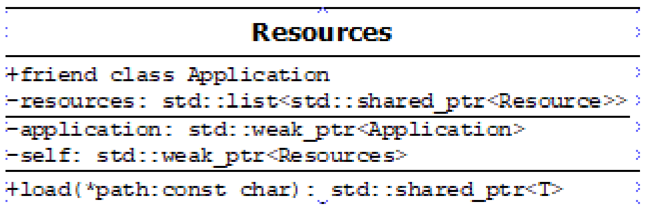

Mesh
+friend class Meshrenderer
+friend class GUI
+modelOfObject: std::shared_ptr<rend::Mesh>
+onLoad(*path:const char)

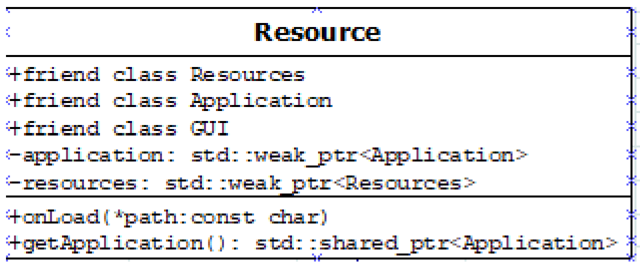
The material class will be used to read and create a texture for the models the user wants. It will take in a path from resources that user has entered and read it, then create a texture using context, which Meshrenderer class then uses to display.

Likewise, the mesh class will be used to read and create meshes for the models the user wants. It will take in a path from resources that user has entered and read it, then create a mesh using context, which Meshrenderer class then uses to display.

## Resources Class and Resource class

Figure 9 and 10:

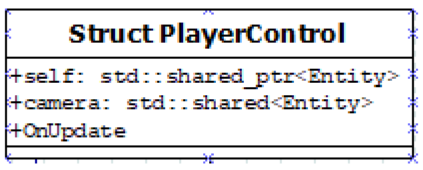

Resources
+friend class Application
-resources: std::list<std::shared_ptr<Resource>>
-application: std::weak_ptr<Application>
-self: std::weak_ptr<Resources>
+load(*path:const char): std::shared_ptr<T>

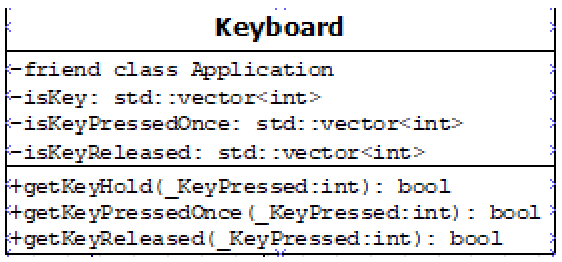

Resource
+friend class Resources
+friend class Application
+friend class GUI
-application: std::weak_ptr<Application>
-resources: std::weak_ptr<Resources>
+onLoad(*path:const char)
+getApplication(): std::shared_ptr<Application>

The resource class is used to distribute the path given from resources to mesh and material. Resources was designed to store a path and deleted other copies of it, and check in the list of resources if the path exist and reusing it instead of reloading, saving loading resources time and increasing efficiency.

## Keyboard and Player Control

Figure 11 and 12:



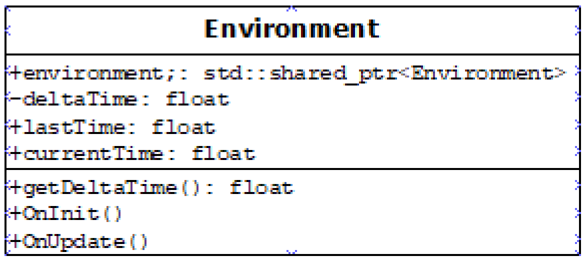

Keyboard
-friend class Application
-isKey: std::vector<int>
-isKeyPressedOnce: std::vector<int>
-isKeyReleased: std::vector<int>
+getKeyHold(_KeyPressed:int): bool
+getKeyPressedOnce(_KeyPressed:int): bool
+getKeyReleased(_KeyPressed:int): bool

Keyboard class checks for inputs of users when pressing a key and storing it in a vector for different types of key press, then checking if they are pressed using iterator.

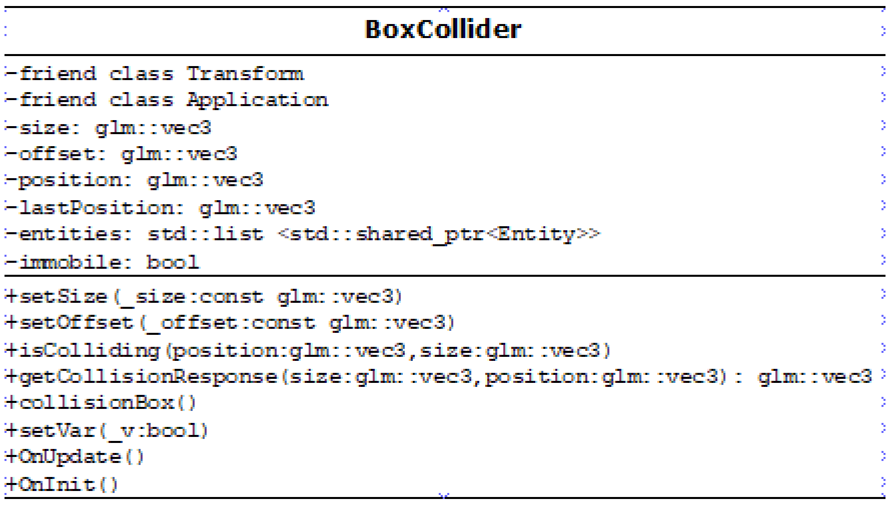
The player Control class will handle player movement by taking keyboard input and changing the position and rotation of the player using the transform class and updating it every frame.

Box Collider Class and Environment Class

Figure 13 and 14:



```
Environment
+environment;: std::shared_ptr<Environment>
-deltaTime: float
+lastTime: float
+currentTime: float
+getDeltaTime(): float
+OnInit()
+OnUpdate()
```



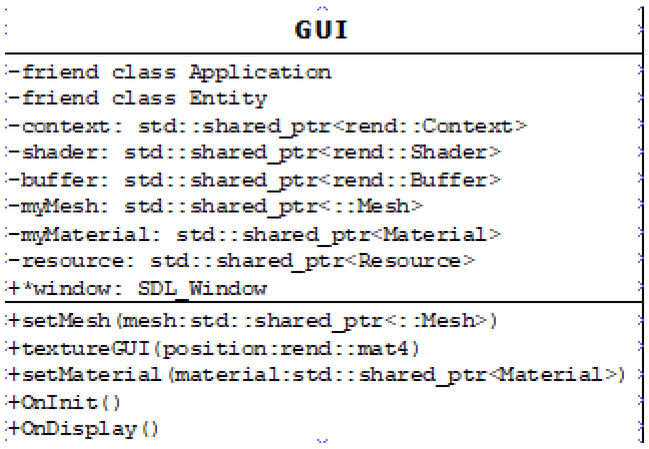
```
BoxCollider
-friend class Transform
-friend class Application
-size: glm::vec3
-offset: glm::vec3
-position: glm::vec3
-lastPosition: glm::vec3
-entities: std::list <std::shared_ptr<Entity>>
-immobile: bool
+setSize(_size:const glm::vec3)
+setOffset(_offset:const glm::vec3)
+isColliding(position:glm::vec3,size:glm::vec3)
+getCollisionResponse(size:glm::vec3,position:glm::vec3): glm::vec3
+collisionBox()
+setVar(_v:bool)
+OnUpdate()
+OnInit()
```

Box Collider class is responsible for collision detection between objects in Scene, using the position of objects around the scene.

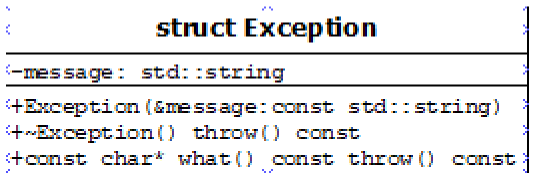
Environment class is used to create delta Time for even performance independent of platform.

GUI Class and Exception class

Figure 15 and 16:



```
GUI
-friend class Application
-friend class Entity
-context: std::shared_ptr<rend::Context>
-shader: std::shared_ptr<rend::Shader>
-buffer: std::shared_ptr<rend::Buffer>
-myMesh: std::shared_ptr<::Mesh>
-myMaterial: std::shared_ptr<Material>
-resource: std::shared_ptr<Resource>
+*window: SDL_Window
+setMesh(mesh:std::shared_ptr<::Mesh>)
+textureGUI(position:rend::mat4)
+setMaterial(material:std::shared_ptr<Material>)
+OnInit()
+OnDisplay()
```



```
struct Exception
-message: std::string
+Exception(&message:const std::string)
+~Exception() throw() const
+const char* what() const throw() const
```

GUI class created to allow users to create graphical user interface such as buttons, logo, player health, etc.

Struct Exception Handles errors and display them to users so they can easily identify where errors are occurring in the game engine, making debugging easier for new users.

## Analysis and Conclusion

Overall, I enjoyed working on this assignment and was fun creating a game engine and learning the architecture of them. The strength of my engine is that it was coded in a readable manner and has a well-designed architecture. For future improvements I would have liked to add more components such as sound, lighting, mesh collision and animation and finish making my GUI component as I Implemented it but it wasn’t fully working unfortunately due to using orthographic projection and pointer errors. I would have also liked to make my resources class more efficient by adding the check for existing paths and reuse them to speed up resource loading.