MyGE - Game Engine by Adam & Sivert

Github

https://github.com/adamaske/MyGE/tree/ecs2

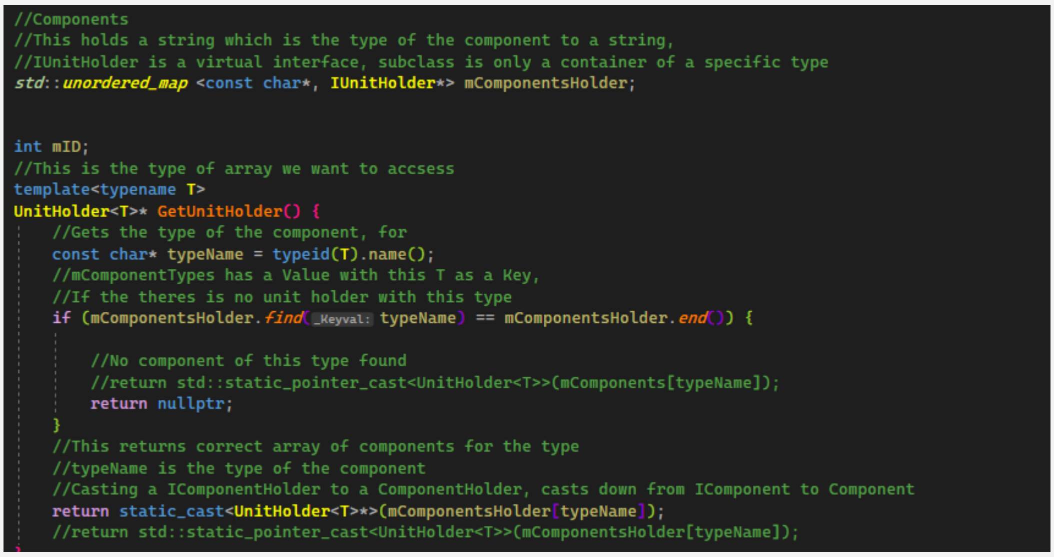
Overview

We have made a game engine with an Entity Component System, Lua integration and Audio integration.
The main function is in main.cpp, and the main game loop is in the while loop inside MyGE::Run. The main parts to look at is Registry.h, Scene.cpp, Systems.h, AudioSystem.cpp. Almost everything is created from a blank C++ project, some files are reused from our previous courses, however almost everything copy and pasted from earlier is now deeply altered to fit the new system.

Entity Component System

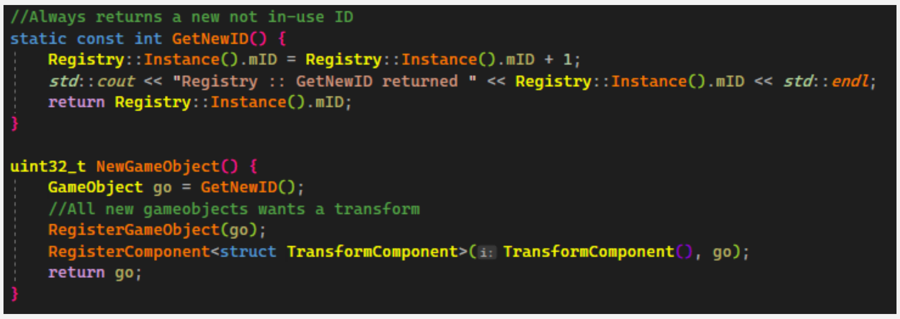
Registry
The registry is the main part of the ECS. This uses templates and templated classes to store ararys of components.
The Registry uses the name of the type which is sent to it for example “struct CameraComponent” to store each UnitHolder. A UnitHolder interface and IUnitHolder parent class allows for storing an array of templated classes and initialising new ones at runtime.

The registry has a singleton instance which will be removed later. The registry uses shared pointers to the templated objects, I started using this to solve many problems in dealing with pointers and references to templates. I store a unorered map of IUnitHolder. Once the Registry wants to register a new component, for example a RenderComponent, the mComponentsHolder gets a UnitHolder<RenderComponent>, the subclass of IUnitHolder. The GetUnitHolder<T> is the function which casts between these two classes. The key is the name of the type.



```cpp
//Components
//This holds a string which is the type of the component to a string,
//IUnitHolder is a virtual interface, subclass is only a container of a specific type
std::unordered_map <const char*, IUnitHolder*> mComponentsHolder;


int mID;
//This is the type of array we want to accsess
template<typename T>
UnitHolder<T>* GetUnitHolder() {
    //Gets the type of the component, for
    const char* typeName = typeid(T).name();
    //mComponentTypes has a Value with this T as a Key,
    //If the theres is no unit holder with this type
    if (mComponentsHolder.find(_Keyval: typeName) == mComponentsHolder.end()) {

        //No component of this type found
        //return std::static_pointer_cast<UnitHolder<T>>(mComponents[typeName]);
        return nullptr;
    }
    //This returns correct array of components for the type
    //typeName is the type of the component
    //Casting a IComponentHolder to a ComponentHolder, casts down from IComponent to Component
    return static_cast<UnitHolder<T>*>(mComponentsHolder[typeName]);
    //return std::static_pointer_cast<UnitHolder<T>>(mComponentsHolder[typeName]);
```

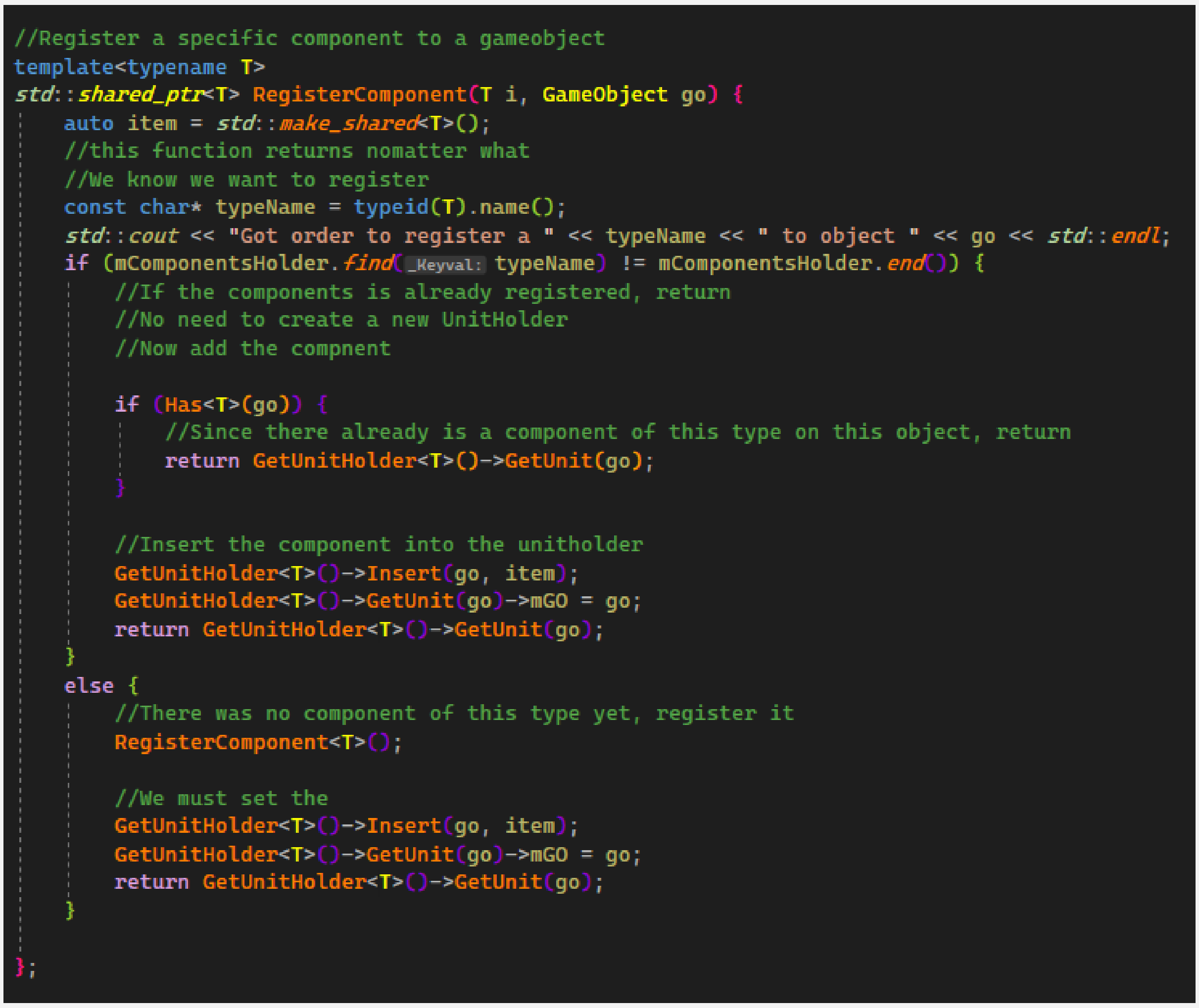
The registry uses an ID system, each gameobject gets an unique ID from the registry. In orderer to create a new GameObject we just call NewGameObject which returns the ID it got.



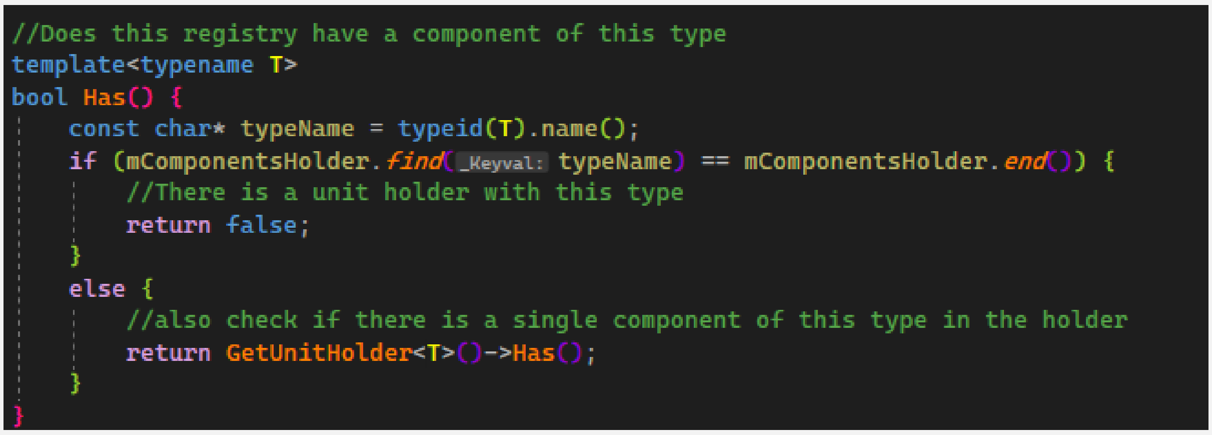
```cpp
//Always returns a new not in-use ID
static const int GetNewID() {
    Registry::Instance().mID = Registry::Instance().mID + 1;
    std::cout << "Registry :: GetNewID returned " << Registry::Instance().mID << std::endl;
    return Registry::Instance().mID;
}

uint32_t NewGameObject() {
    GameObject go = GetNewID();
    //All new gameobjects wants a transform
    RegisterGameObject(go);
    RegisterComponent<struct TransformComponent>(i: TransformComponent(), go);
    return go;
}
```

Entity
An entity in this game engine is just a uint32_t, we only want a entity to be a number which connects components.

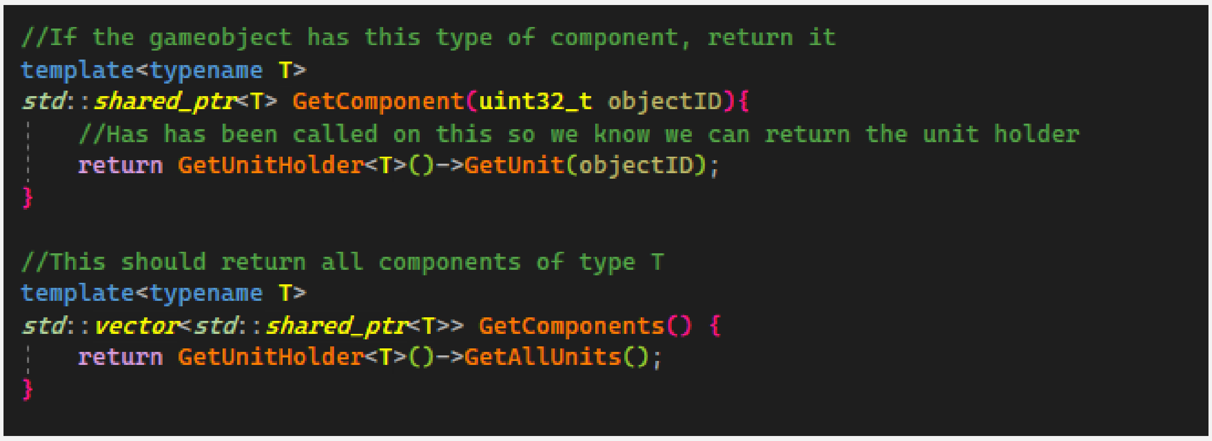
Components
Each component is a struct, only containing information. The main functions for the components are RegisterComponent, GetComponent(goID) and GetComponents(). RegisterComponent(T* item, uint32_t) is used for telling the registry “This component is now attached to this GameObject”. GetComponent<T>(goID), if the gameobject has a component of type T, it returns it. GetComponents<T>() gets every component of this type in the registry(in the whole scene). Both GetComponent and GetComponents should be checked with Has<T>(go) and Has<T>() before using them to avoid nullptrr, this must be dealt with later.



```cpp
//Register a specific component to a gameobject
template<typename T>
std::shared_ptr<T> RegisterComponent(T i, GameObject go) {
    auto item = std::make_shared<T>();
    //this function returns nomatter what
    //We know we want to register
    const char* typeName = typeid(T).name();
    std::cout << "Got order to register a " << typeName << " to object " << go << std::endl;
    if (mComponentsHolder.find(_Keyval: typeName) != mComponentsHolder.end()) {
        //If the components is already registered, return
        //No need to create a new UnitHolder
        //Now add the compnent

        if (Has<T>(go)) {
            //Since there already is a component of this type on this object, return
            return GetUnitHolder<T>()->GetUnit(go);
        }

        //Insert the component into the unitholder
        GetUnitHolder<T>()->Insert(go, item);
        GetUnitHolder<T>()->GetUnit(go)->mGO = go;
        return GetUnitHolder<T>()->GetUnit(go);
    }
    else {
        //There was no component of this type yet, register it
        RegisterComponent<T>();

        //We must set the
        GetUnitHolder<T>()->Insert(go, item);
        GetUnitHolder<T>()->GetUnit(go)->mGO = go;
        return GetUnitHolder<T>()->GetUnit(go);
    }

};
```



```cpp
//Does this registry have a component of this type
template<typename T>
bool Has() {
    const char* typeName = typeid(T).name();
    if (mComponentsHolder.find(_Keyval: typeName) == mComponentsHolder.end()) {
        //There is a unit holder with this type
        return false;
    }
    else {
        //also check if there is a single component of this type in the holder
        return GetUnitHolder<T>()->Has();
    }
}
```



```cpp
//If the gameobject has this type of component, return it
template<typename T>
std::shared_ptr<T> GetComponent(uint32_t objectID){
    //Has has been called on this so we know we can return the unit holder
    return GetUnitHolder<T>()->GetUnit(objectID);
}

//This should return all components of type T
template<typename T>
std::vector<std::shared_ptr<T>> GetComponents() {
    return GetUnitHolder<T>()->GetAllUnits();
}
```

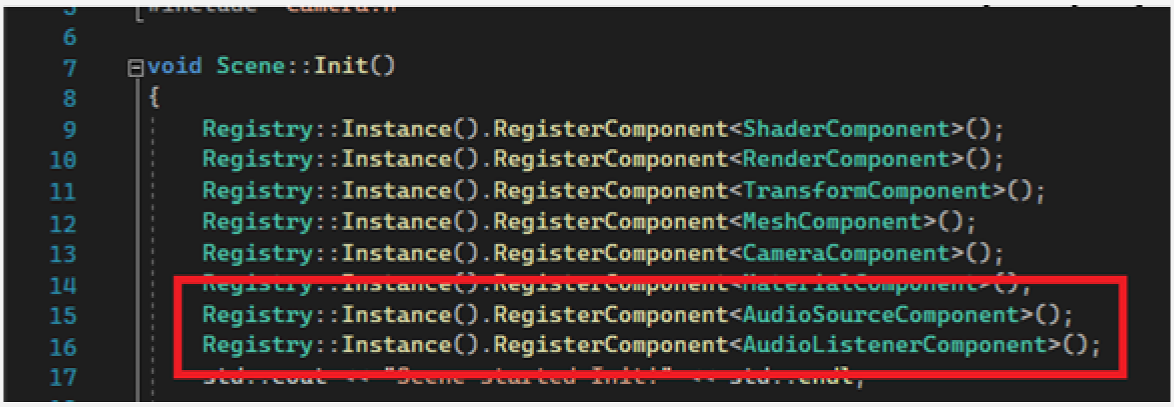
Systems
Each system is responsible for different parts of the engine. The AudioSystem handles everything audio and ObjMeshSystem handles creating and rendering meshes to the screen. Each system has a Init and OnUpdate function, these will be called when the scene calls its Init and OnUpdate.

OpenAL

Ive decided to do audio by using openAL in this project. Since it's built around a ECS(entity component system) the system is quite compressed compared to the audio system in the last compulsory. The system is now ready to let us easily create and play audio sources, listeners and no matter how many we add the AudioSystem will take care of all their functionality.

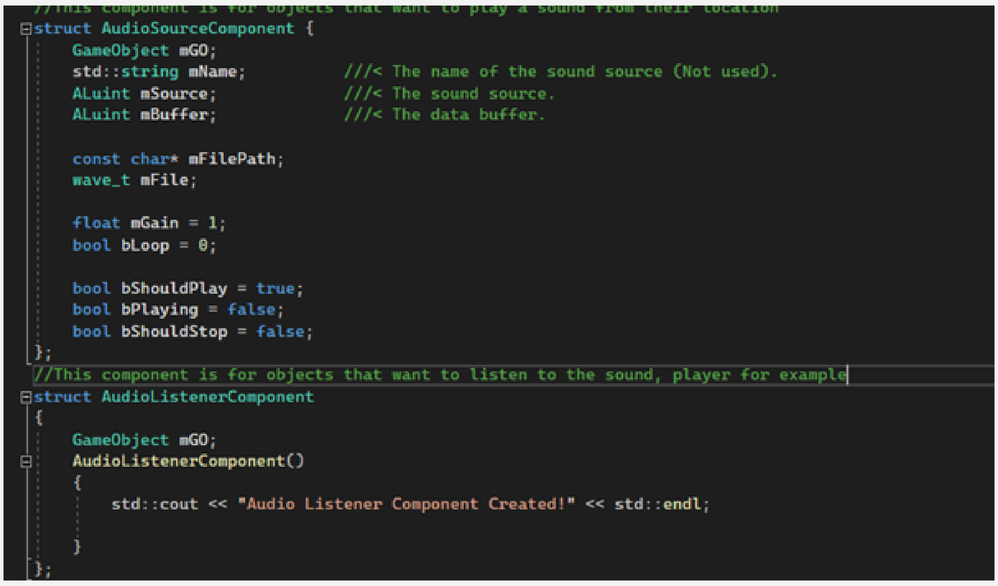
1 – registering an audio component to the registry

At the start the system needs to put the audio components in a registry, this is done in the scene.cpp



```cpp
#include "Camera.h"

void Scene::Init()
{
    Registry::Instance().RegisterComponent<ShaderComponent>();
    Registry::Instance().RegisterComponent<RenderComponent>();
    Registry::Instance().RegisterComponent<TransformComponent>();
    Registry::Instance().RegisterComponent<MeshComponent>();
    Registry::Instance().RegisterComponent<CameraComponent>();
    Registry::Instance().RegisterComponent<MaterialComponent>();
    Registry::Instance().RegisterComponent<AudioSourceComponent>();
    Registry::Instance().RegisterComponent<AudioListenerComponent>();
    std::cout << "Scene started Init!" << std::endl;
```

These components exists in forms of structs that are located in components.h

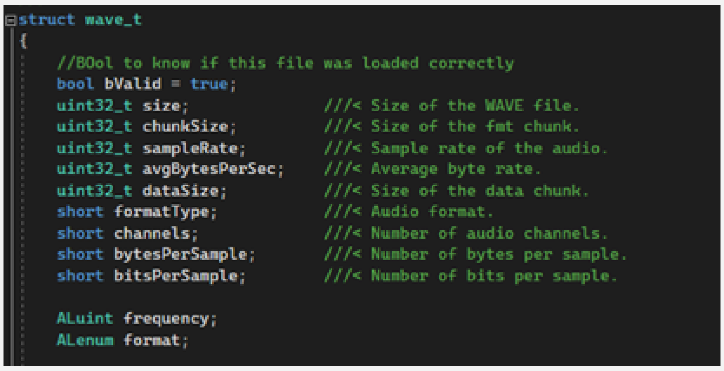


```
//This component is for objects that want to play a sound from their location
struct AudioSourceComponent {
    GameObject mGO;
    std::string mName;          ///< The name of the sound source (Not used).
    ALuint mSource;             ///< The sound source.
    ALuint mBuffer;             ///< The data buffer.

    const char* mFilePath;
    wave_t mFile;

    float mGain = 1;
    bool bLoop = 0;

    bool bShouldPlay = true;
    bool bPlaying = false;
    bool bShouldStop = false;
};
//This component is for objects that want to listen to the sound, player for example
struct AudioListenerComponent
{
    GameObject mGO;
    AudioListenerComponent()
    {
        std::cout << "Audio Listener Component Created!" << std::endl;

    }
};
```

Inside the structs are all the declarations that's necessary for audio to play.

AudioSourceComponent

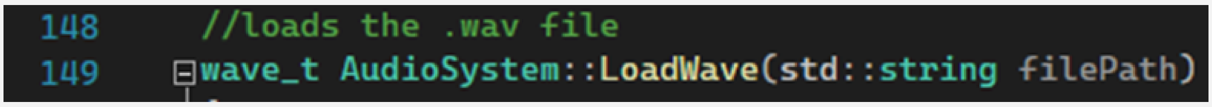
This one does everything from retrieving the .wav file to playing it. It starts by reading the .wav file. It does this with the wavefilereader.h and cpp.



```
struct wave_t
{
    //BOol to know if this file was loaded correctly
    bool bValid = true;
    uint32_t size;              ///< Size of the WAVE file.
    uint32_t chunkSize;         ///< Size of the fmt chunk.
    uint32_t sampleRate;        ///< Sample rate of the audio.
    uint32_t avgBytesPerSec;    ///< Average byte rate.
    uint32_t dataSize;          ///< Size of the data chunk.
    short formatType;           ///< Audio format.
    short channels;             ///< Number of audio channels.
    short bytesPerSample;       ///< Number of bytes per sample.
    short bitsPerSample;        ///< Number of bits per sample.

    ALuint frequency;
    ALenum format;
```

Here is all the information that the script needs to read the file. After this it retrieves and reads the file.

The file can now be loaded and used, which is done in the AudioSystem Class

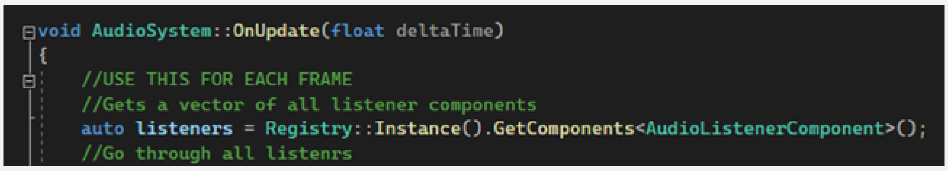


```
    //loads the .wav file
wave_t AudioSystem::LoadWave(std::string filePath)
```

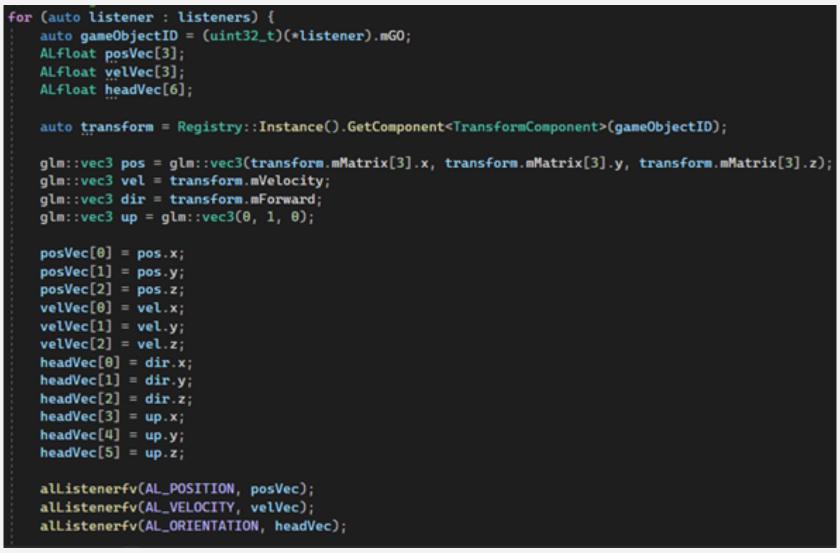
On init the AudioSystem will add all of the read and loaded sounds to a registry so that it can be indexed and relevant information can be stored. (like position, velocity, format. Etc).

AudioListenerComponent

This component makes sure the player can hear the audio that playing, this is the listener component that receives the audio. It does this by creating a listener tag and creating another registry for it



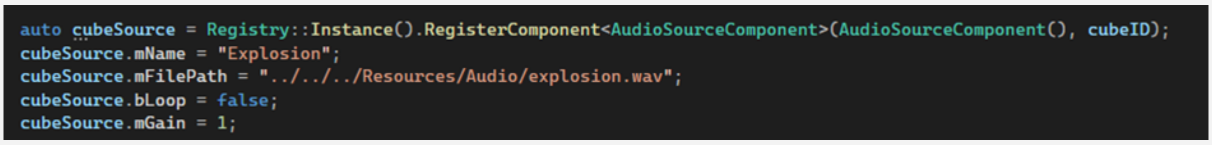
```cpp
void AudioSystem::OnUpdate(float deltaTime)
{
    //USE THIS FOR EACH FRAME
    //Gets a vector of all listener components
    auto listeners = Registry::Instance().GetComponents<AudioListenerComponent>();
    //Go through all listenrs
```



```cpp
for (auto listener : listeners) {
    auto gameObjectID = (uint32_t)(*listener).mGO;
    ALfloat posVec[3];
    ALfloat velVec[3];
    ALfloat headVec[6];

    auto transform = Registry::Instance().GetComponent<TransformComponent>(gameObjectID);

    glm::vec3 pos = glm::vec3(transform.mMatrix[3].x, transform.mMatrix[3].y, transform.mMatrix[3].z);
    glm::vec3 vel = transform.mVelocity;
    glm::vec3 dir = transform.mForward;
    glm::vec3 up = glm::vec3(0, 1, 0);

    posVec[0] = pos.x;
    posVec[1] = pos.y;
    posVec[2] = pos.z;
    velVec[0] = vel.x;
    velVec[1] = vel.y;
    velVec[2] = vel.z;
    headVec[0] = dir.x;
    headVec[1] = dir.y;
    headVec[2] = dir.z;
    headVec[3] = up.x;
    headVec[4] = up.y;
    headVec[5] = up.z;

    alListenerfv(AL_POSITION, posVec);
    alListenerfv(AL_VELOCITY, velVec);
    alListenerfv(AL_ORIENTATION, headVec);
```

Above we see that the listener gets a gets a position, direction and velocity, which are then converted to openAL so that the correct info is sent out.

Lastly the audio functionality is added in as an example below. It first finds the name of the file “Explosion”, its file path, if its looping or not and its gain.



```cpp
auto cubeSource = Registry::Instance().RegisterComponent<AudioSourceComponent>(AudioSourceComponent(), cubeID);
cubeSource.mName = "Explosion";
cubeSource.mFilePath = "../../../Resources/Audio/explosion.wav";
cubeSource.bLoop = false;
cubeSource.mGain = 1;
```

Lua

Lua has not been further developed since the last compulsory.

Native Scripting

Native scripting has not been started, but I know relatively well how I am going to create it. The native scripting will basically serve as an abstraction from the engine code, so the users dont have to deal with engine specific code. I will have a class called something like NativeScriptingEntity which will contain the functions we want the user to access, stuff like AddComponent, GetPosition, SetPosition, much like Unreal Engine.
The plan is to separate out the editor and runtime parts of the engine into different projects in the visual studio solution. That way we can use the engine with our own scripts without recompiling the entire engine.

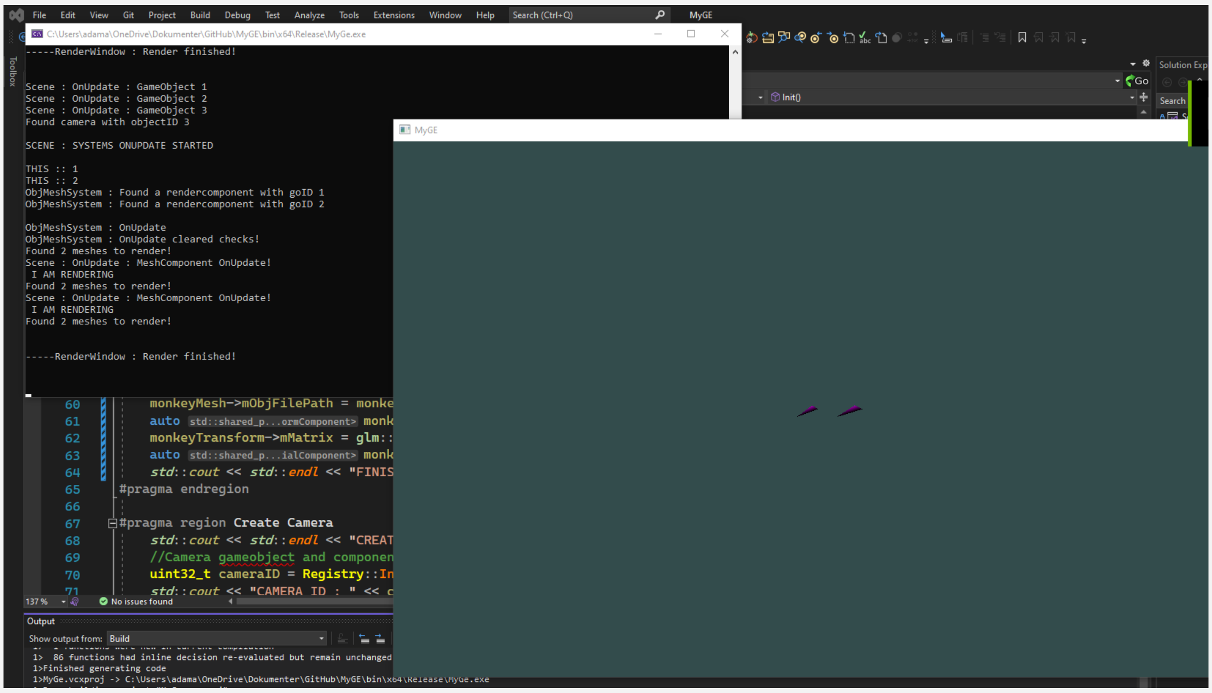
Removing OPENGL specific code

I will spend abstracting the code relating to opengl. This should make it easier to deal with, having generic code instead of opengl code makes it easier to work with. Creating classes for VAO, vbo's, ibo and such.

I completed this by creating the Buffer files and VertexArray class, in the ObjMeshSystem there is now no opengl specific code. These classes can now be slightly altered to make it easy to implement another rendering api without rewriting any engine code.

Result

Current state. The engine is currently only rendering what looks like two triangles, this is a bug caused by the buffer and vertex array class not using a correct mRendererID. We have the engine rendering our custom meshes to reading obj files in the ObjMeshSystem. Sound can also be deployed easily through creating AudioSourceComponents and giving the wav files. This will be easier to use once the native scripting is done.



```
-----RenderWindow : Render finished!


Scene : OnUpdate : GameObject 1
Scene : OnUpdate : GameObject 2
Scene : OnUpdate : GameObject 3
Found camera with objectID 3

SCENE : SYSTEMS ONUPDATE STARTED

THIS :: 1
THIS :: 2
ObjMeshSystem : Found a rendercomponent with goID 1
ObjMeshSystem : Found a rendercomponent with goID 2

ObjMeshSystem : OnUpdate
ObjMeshSystem : OnUpdate cleared checks!
Found 2 meshes to render!
Scene : OnUpdate : MeshComponent OnUpdate!
 I AM RENDERING
Found 2 meshes to render!
Scene : OnUpdate : MeshComponent OnUpdate!
 I AM RENDERING
Found 2 meshes to render!


-----RenderWindow : Render finished!
```

```
        monkeyMesh->mObjFilePath = monke
        auto std::shared_p...ormComponent> monk
        monkeyTransform->mMatrix = glm::
        auto std::shared_p...ialComponent> monk
        std::cout << std::endl << "FINIS
#pragma endregion

#pragma region Create Camera
        std::cout << std::endl << "CREAT
        //Camera gameobject and componen
        uint32_t cameraID = Registry::In
        std::cout << "CAMERA ID : " << c
```

```
1>  86 functions had inline decision re-evaluated but remain unchanged
1>Finished generating code
1>MyGe.vcxproj -> C:\Users\adama\OneDrive\Dokumenter\GitHub\MyGE\bin\x64\Release\MyGe.exe
```