# **JUSTIFICATION.**

The entire application has been rendered using a DirectX11 based engine or framework, called “Wolfy3D”. I started developing it as a good way to learn more about the graphic library of Microsoft, as before starting studying in Sheffield Hallam I just had used OpenGL.

The solution will have 3 projects. The main one will be used to make the static library of the engine, that will be linked to the other 2 projects, one for testing and the other for the assignment. This will be made using genie, with some useful “.bat” files that can be executed to clean or generate the Visual Studio solution.

I will start describing how the engine project (“Wolfy3D”) works and enumerating its main classes and characteristics:

* Wnd: Wnd is the core class to manage the window application, opening and closing it, and setting it resolution.
* Directx: Will manage all the DirectX11 rendering base, such as the frame buffering, device context.
* Cam: Designed to manage the camera of the scene and all its transformation, calculating the view and projection matrix that will be sent to the HLSL developed shader to execute all the render components.
* Entity: The base of the hierarchy node system. As the engine is oriented to components, this class will own a transform and render component. The node system will be the classical tree node with pointers to its children.
* Input: Class to manage all the keyboard and mouse inputs. Will give us all the information about if a button has been just pressed, or if it has been held down or up.
* Geo: Geometries, where the user can export from external “.x” files using the simple parser that I developed. All the vertex data will be allocated in this class, such as its normal vectors, positions, texture coordinates, colours and buffers for the vertex and the elements. The topology can be set manually to have different kinds of rendering.  
  Other predefined geometries that I offer to the user are cube, skybox, pyramid, terrain (based on a heightmap external file), triangle, quad and regular polygons extruded.
* Super Material: When the engine starts, one instance of a material will be loaded to the graphic card and used to render each entity. This is made to optimize, as the graphic card will have just one shader loaded, that will manage all the different material parameters that we need. Actually, the shader is using multiple textures, a custom constant buffer, and a basic directional light.
* Texture: Class designed to manage the different textures that we want to use in the shader. Loads all the info that we need to render external texture files.

As I mentioned before, the framework is based on components:

* Render: Will use an existing material and geometry to render an object in the scene. The camera will be the one who will call the render function recursively, to draw all the objects selected and its children. All the information of the transform, material and geometry will be given to the shader.
* Transform: Complete “API” with all the transformations of an entity, will calculate the model matrix used in the shader, and will have many useful getters and setters such as world and local positions, scales, rotations, direction vectors as forward, up and right. The rotation transformations are internally made using quaternions, but for the user there will be useful methods that will simulate that the engine is working with Euler angles, that are more user friendly. I made this decision because I liked the way that Unreal Engine manage the transformations, as you don’t know how they are programmed but they offer you the chance of rotate objects using quaternions or axis. This transform will also have information about the hierarchy, so will be recursively updated anytime that there is a change in any object. For example, if I change the position of an entity, automatically its transform will be changed, and all its children will be updated recursively too.

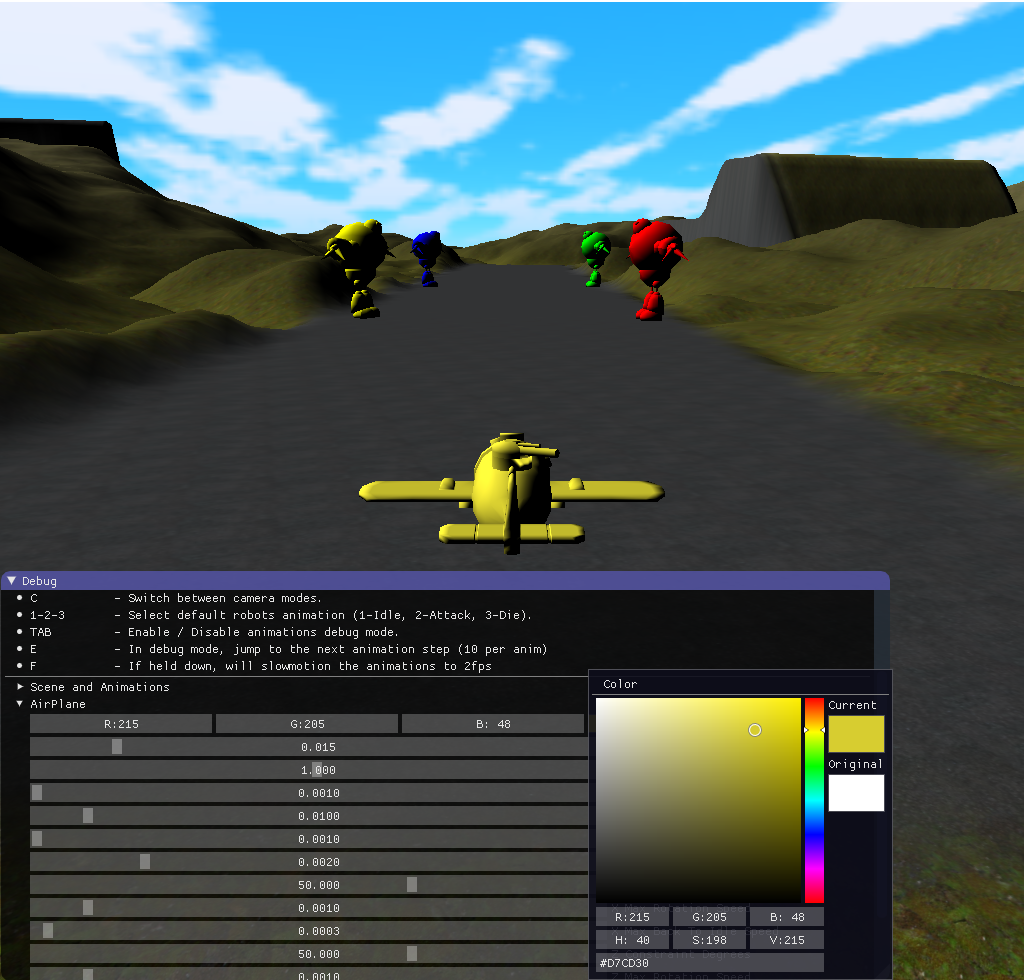
These classes are in the core, but they also have an abstraction layer, so the user can program simple applications without needing to know anything about how the core is working. The main abstractions are:

* Window: Simplified “API” to open and close a window with a custom resolution and size, and to start and to end a frame.
* Globals: Header that will contain all the enumeration and structures declarations that we may need to be accessible from everywhere in the core. To make sure that the most frequent variable types will have the same size, in any arquitecture, I have created my own data types, such as float32, uint64, or char8.
* Input: Namespace with all the getters and setters related with the mouse and keyboard input. An enumeration with all the available keys will be given to the user.
* Time: Information about the current time of the application in different formats, and a chronometer structure that we could use to know average execution times.
* Material: User abstraction of the super material, will consist in just a simple structure with information about how we want to render the object. These different materials offered to the users are the diffuse, textured, terrain and normals.
* Math: Simple “API” with mathematical methods of tweening, quaternions and Euler conversions. The engine will use “DirectXMath“ making the calculations using “XMVECTOR” and “XMMATRIX”, but allocating the memory in “XMFLOAT”, etc.

To optimize and testing the engine I have developed a simple structure called Chrome Debugger, that will generate a “.json” file that can be used in the Google Chrome Debug tool.

There are also some external dependencies integrated in the core of the engine:

* Tinyxml2: Used for parsing the xml files which contain the animations information.
* ImGui: Rendering editor which is very useful to make all the testing. In the next image we can appreciate how the assignment can be customized in real time.



The assignment will have a very simple class organization. There will be a scene which will manage all the entities, and an individual class to have all the functionality of the airplane, the robots, the bullets, the animations and the skybox.

* Airplane: Able navigate after its engine is turned on. Using the transform simulated Euler rotations, and getting the global forward vector of its gun child to take the next bullet available to be fired. The navigation parameters are customizable using ImGui, and the movements are relative to the rotations of the plane, to make them look natural and smooth.
* Bullet: There will be a pull of bullets that we will use to render the shoots.
* Skybox: Will be attached to the plane root, so we aren’t able to get out from it.
* Robots: Animated group of entities. Will attack when the plane is near, and will stay idle when it goes far from them.
* Animation Controller: The hierarchy to manage them is: The robots will contain one animation controller each, that will contain the three animations (idle, die and attack). These animations have information of each bone which have all the data of the transformation (translation and rotation) of each step or keyframe.
* Bones: The bones will have the same hierarchy system than the entities. Each bone has its own steps info. These steps will be made by using the engine math library interpolation functions, so the transition between keyframes and the blending between animations will be smooth and will look natural. There will be a debug mode and these animations animation parameters can be set to customize in each moment the speed, to be easily tested. The animations will be played in loop automatically.