Steel Engine 1.1 Documentation

Contents

About Steel Engine 2
About the Documentation 2
Components 3
GameObjects 3
Scenes 5
SceneManager 5
RenderShader 6
CollisionManager 7
Collider 8
BoxCollider 9
SteelRay 10
GUIElement 11
GUIButton 12
GUIText 13
GUIImage 14
GUIManager 15
InfoManager 16
InputManager 17
Mesh 18
SteelTriangle 19
SteelVertex 20
SteelEdge 21
Time 22
ObjImporter 23
Rigidbody 24
LightObject 25
LightManager 26
How to Use Steel Engine 27
How to Build Your Game 29

About Steel Engine

The engine is designed to be similar to Unity to provide an easy transition from Unity to Steel Engine for those who used to use Unity. Steel Engine supports 3D and 2D development. Steel Engine currently only works on Windows. It might run under wine on Linux but it has not been tested so this is not guaranteed. This engine is currently very primitive so certain features may be underdeveloped and can have bugs. If any bugs are found, they should be reported by opening an issue on the **Github Page**.

About the Documentation

Anything highlighted green is a feature that has been added in this version.

If there is anything that is unclear or missing from this documentation, please open an issue on the **Github page**.

Components

A component is a script containing a class that inherits from the ‘component’ class or a ‘component’ subclass. These classes can be instantiated and applied to any **GameObject**.

Functions:

Every component has a ‘**Tick**’ function that must be implemented. This function provides one float parameter. The value of this parameter is always the **DeltaTime** (time between the last and current frame).

Every component also has a ‘**Init**’ function. This function is protected and virtual. This means that only classes that are a subclass of the component class can access this function and that it *can* be overridden but it isn’t required. This function gets called when the component is **initialised**. Do not get this confused with the component being **instantiated**.

GameObjects

A GameObject in Steel Engine is pretty much the same thing as a GameObject in Unity. It is an object in the scene and can have the following properties:

- Position (vec3)
- eRotation (vec3)
- qRotation (quaternion)
- Scale (vec3)
- RenderShader (RenderShader)
- ID (int)
- Name (string)
- Components (List<Component>)
- Parent (GameObject)

The **eRotation** is the rotation of the object in Euler angles.

The **qRotation** is the rotation of the object as a quaternion. Of course, the eRotation is prone to gimble lock so qRotation should be used when possible.

The **RenderShader** is a variable of type RenderShader specifying the type of shader to use when rendering the GameObject.

The **Parent** of the GameObject is not implemented as of this version but it will in future make the **Position**, **Rotation** and **Scale** of the GameObject relative to its Parent.

Functions:

- Void AddComponent(Component component)
- T GetComponent<T>() where T : class
- Void Tick(float DeltaTime)
- Static GameObject Instantiate(GameObject Original)
- Static GameObject QuickCopy(GameObject Original)
- Matrix4 GetModelMatrix()
- Void Rotate(Vector3 rotation)
- Void Rotate(Float x, Float y, Float z)
- Void SetRotation(Vector3 rotation)

- Void SetRotation(Quaternion rotation)
- Void LoadTexture(String name, String extension)
- Void LoadTexture(String path)
- Void LoadTexture(Texture texture)
- Void Load()
- Void Render()

The **AddComponent** function appends the given component instance to the GameObjects component list and calls the **component's Init** function. This should be used only in runtime otherwise the component maybe initialised twice.

The **GetComponent** function is a generic function that takes a component type and finds the first component instance of the specified type on the GameObject. If no instances are found, null is returned.

The **Tick** function is called every frame by the **SceneManager** if the **InfoManager's GameRunning** variable is true. It calls every **Component's Tick**

The **Instantiate** function takes a GameObject as an original and creates a new GameObject instance with the same properties. This function is very slow so it should not be used often.

The **QuickCopy** function is a sort of faster version of the instantiate function with one drawback. This being that it's simply creating a new reference type and not value type variable. This means that it's pretty much just a pointer to the original GameObject provided, rather than a fresh space in memory with the same values. But it is much **faster** than the **instantiate** function.

The **GetModelMatrix** function returns a Matrix4 representing the transformation of the GameObject (the model matrix).

The **Rotate** function has 1 overload. The base function takes a Vector3 value representing the desired rotation in Euler angles and applies it to **eRotation** and **qRotation** of the GameObject. The overload takes 3 floating point values representing the Euler angles X, Y and Z values and applies them to the eRotation and qRotation of the GameObject.

The **SetRotation** function has 1 overload. Both functions set the eRotation and qRotation values of the GameObject, but the base function takes the rotation as a Vector3 (Euler angles) rotation, and the overload takes the rotation as a quaternion rotation.

The **LoadTexture** function has 2 overloads. They all apply a texture to the GameObject. The base function takes the desired textures name and file extension, and assumes it exists in the Resources/Textures folder. The first overload takes a path to the texture including the extension. The second overload takes a **Texture**.

The **Load** function sets up the GameObjects shaders, vertices, and indices along with calling every attached component's Init function.

The **Render** function renders the GameObject to the viewport.

Scenes

A **scene** contains information about the **GameObjects**, **LightObjects**, and **Cameras** in the scene. These are all stored in three separate lists that are public.

A scene also has two more properties being the **Name** and the **Starting Camera ID**. These properties are self explanatory.

Functions:

- Void Load()
- Void AddLight(Vector3 position, Vector3 colour, float intensity)

The **Load** function is used to replace all GameObjects, LightObjects, and Cameras in the **SceneManager** with the GameObjects, LightObjects, and Cameras in the scene.

The **AddLight** function is used to add a LightObject to the scene. It takes all of the parameters used to create a LightObject and is not the most useful function, but it exists, nevertheless.

SceneManager

The SceneManager is a static class that stores and handles all information around scenes.

The SceneManager public properties:

- Scenes (List<Scene>)
- GameObjects (List<GameObject>)
- Cameras (List<Camera>)
- GameRunning (Boolean)
- GameTick (event – void GameTick(float DeltaTime))

Functions:

- Scene GetActiveScene()
- void LoadScene(int BuildIndex)
- GameObject GetGameObjectByID(int ID)
- Scene ConstructScene(string[] Lines)
- void Init()
- void Tick(double DeltaTime)
- SteelRay CalculateRay(Vector2 MousePosition)
- void ChangeClearColour(Vector3 Colour)

GetActiveScene returns the currently loaded scene.

LoadScene uses the BuildIndex to load a scene.

GetGameObjectByID returns a loaded GameObject which matches the given **ID**.

ConstructScene creates a scene by using the lines of a scene file (.SES).

Init creates all the scenes by using the scene files in the Resources/Scenes folder.

Tick invokes the **GameTick** event when the **InfoManager's GameRunning** property is true.

CalculateRay creates a **SteelRay** from the mouse's screen position forwards from the camera into the world.

RenderShader

The RenderShader type is an Enum containing the following shader types:

- ShadeFlat
- ShadeLighting
- ShadeTextureUnit

The ShadeFlat shader is a simple shader that renders the object using just the colour of each vertex. This means that it wont react to lighting and doesn't support textures.

The ShadeLighting shader is in the current version slightly broken, but it will be improved as time goes on. This shader attempts to change the brightness of each vertex based off of the surrounding lights in the scene. It also only supports one light source in this version.

The ShadeTextureUnit shader is also a simple shader that uses the **Mesh's** UVs to determine how to display a texture onto the GameObject. This doesn't react to lighting and doesn't support vertex colours.

CollisionManager

Contains a list of all of the colliders in the scene that automatically updates itself if a collider is made so the user doesn’t have to append the colliders manually.

Collider

A general class that all forms of colliders inherit. This class inherits the component class

Functions:

- Virtual Vector3 CalculateCollisionNormal(Collider Other)
- Virtual Bool CheckCollision(Collider other)
- Virtual Bool CheckCollision(Collider other, Vector3 TargetPos)

The **CalculateCollisionNormal** function takes another collider and attempts to calculate a Vector3 that describes the motion required to resolve the collision. If no collision is present it will return invalid values so it should only be used if a collision has been previously detected.

The **CheckCollision** function takes another collider and attempts to check for a collision returning true of false stating if a collision is present. This function also has 1 overload that takes an extra Vector3 value. This value describes a position of the GameObject that the collider component it attached to. This can be used to check if a collision would be present between two colliders **if** the current one **was** at the specified position.

BoxCollider

A class that inherits the **Collider** class. The BoxCollider class describes the a cuboid not just a cube. Other than that, it has no new properties or functions than the Collider class.

SteelRay

A class that describes a theoretical ray.

Properties:

- Vector3 WorldPosition
- Vector3 WorldDirection
- Float StepSize
- Float Distance
- Float Threshold

The SteelRay Class has no functions.

GUIElement

A general class that all GUI elements inherit. All UI only works on 1920x1080 resolution.

Properties:

- Int RenderOrder
- Int LocalRenderOrder
- String Name
- Float ZRotation
- Vector3 AddedPosition
- Vector2 Anchor
- GUIElement ParentGUI
- Bool Visible
- Int TextureID
- Bitmap Texture
- GameObject RenderObject
- List<String> Textures

The **RenderOrder** property is a global render order for the element. This should only be set if the object has no parent. If the object has a parent, this property will become that of the parent plus one plus the **LocalRenderOrder**.

The **AddedPosition** property is a 3D position that can be added to the element. This should almost never be used unless in the strangest of cases. This is property only exists for special cases.

The **Anchor** property specifies where on the screen the UI object should be anchored to, using values -1 to 1.

Functions:

- Void SetZRotation(Float Rotation)
- Void ApplyTexture(Bitmap bmp)
- Virtual Void Render()
- Virtual Void CleanUp()

The **SetZRotation** function should be used to set the **ZRotation** variable of the GUIElement because it also applies this rotation to the **RenderObject**. This function is also slow and so should not be used very often. The reason for this is that the RenderObject is reloaded every time the function is called.

The **CleanUp** function removes all of the used textures from the temp folder.

GUIButton

A class that inherits from the GUIElement class, and functions like a button.

Properties:

- Event ButtonDown
- Event ButtonHold
- Event ButtonUp
- Vector2 scale

The **ButtonDown** event is invoked only on the tick that the button is initially pressed.

The **ButtonHold** event is invoked every tick that the button is pressed for.

The **ButtonUp** event is invoked only on the tick that the button is unpressed.

Functions:

- Void SetPressedImage(String Name, String Extension)
- Void SetPressedImage(String Path)
- Bool CheckBounds(Vector2 MousePosition)

The **SetPressedImage** function sets the image that is displayed on the button object when the button is held. The base function takes the name and extension of the image file and assumes it is stored in 'Resources\Textures\'. This function has 1 overload that takes just the full file path of the image including the extension as a string.

The **CheckBounds** function takes a mouse position and returns a true or false value determining if the specified mouse position is within the bounds of the button.

GUIText

A class that inherits the GUIElement class and displays some text.

Properties:

- String Text
- String Font
- Float Size
- Float Scale

The **Size** property refers to the font size, while the **Scale** refers to the size of the GUIElement.

Functions:

- Void PreloadText(String Text)
- Void SetText()

The **PreloadText** function is a slow function. This is used to create a text texture beforehand, so it is meant to be used on startup / initialisation instead of every tick. The speed of the function is greatly effected by the parameters given when creating a new GUIText object.

The **SetText** function is approximately 2x faster than the **PreloadText** function, but it must be considered that this function can also be incredibly slow depending on the parameters given when create a new GUIText object. If this function is too slow to use while the game is running, the **PreloadText** function should be used on load before, and then this function will run super-fast while the game is running.

GUIImage

A class that inherits the GUIElement class and displays an image.

Properties:

- Texture Image

Functions:

- Void SetColour(Vector4 Colour)

Since the **GUIImage** class has two uses the **SetColour** function is used to set the colour of the GUIImage. **This is easy to get confused** as the GUIImage could either be used to render an existing image, or to render just a single colour. This function only works on the latter. **It will not change / tint the colour of the loaded texture**, but rather set the GUIImage to one colour. The colour is specified by a Vector4. The X,Y,Z,W components correspond to R,G,B,A 0 - 255.

GUIManager

The GUIManager stores and handles most GUI related objects.

Properties:

- List<GUIElements> GUIElements

The **GUIElements** property is a list of all the **GUIElements** that are currently being rendered to the screen.

Functions:

- GUIElement GetElementByID(Int ID)
- GUIElement GetElementByName(String Name)
- Void AddGUIElement(GUIElement Element)

The **GetElementByID** function returns the GUIElement that has a matching id to the one specified.

The **GetElementByName** function returns the GUIElement that has a matching name to the one specified.

The **AddGUIElement** function adds the element to the currently rendered GUIElements list.

InfoManager

The InfoManager stores general information about the engine or game. It is a static class so that it can be easily accessed from all places.

Properties:

- String CurrentDevPath
- String DevDataPath
- String CurrentDir
- String DataPath
- Float GravityStrength
- Camera EngineCamera
- Vector2 WindowSize
- GameObject TestSphere
- Bool IsBuild

The **CurrentDevPath** is a file path that the user must change to work correctly for them. The path specifies a path to the second 'Steel Engine' folder. An example of a valid **CurrentDevPath** is:

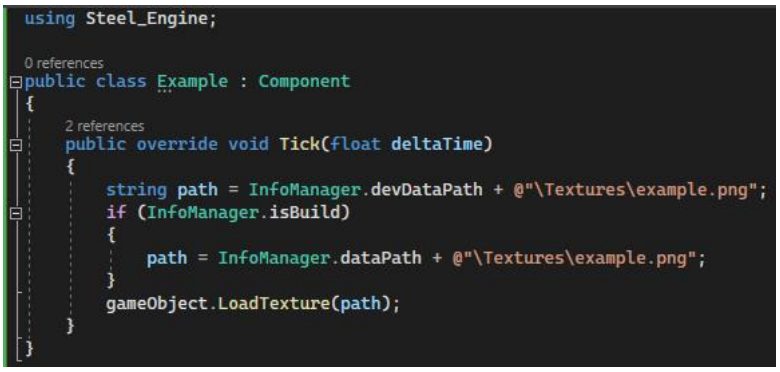
'`C:\Users\XUSER\source\repos\Steel Engine\Steel Engine\`'.

The **DevDataPath** is also a file path that the user must change to work correctly for them. The path specifies a path to the 'Resources' folder inside the second 'Steel Engine' folder. An example of a valid **DevDataPath** is:

'`C:\Users\XUSER\source\repos\Steel Engine\Steel Engine\Resources\`'

The **CurrentDir** is a file path that is set automatically when the game is built. The path specifies a path to the second 'Steel Engine' folder.

The **DataPath** is also a file path that is set automatically when the game is built. The path specifies a path to the 'Resource' folder. An example of how these file paths should be used is:



```
using Steel_Engine;

0 references
public class Example : Component
{
    2 references
    public override void Tick(float deltaTime)
    {
        string path = InfoManager.devDataPath + @"\Textures\example.png";
        if (InfoManager.isBuild)
        {
            path = InfoManager.dataPath + @"\Textures\example.png";
        }
        gameObject.LoadTexture(path);
    }
}
```

The **DevDataPath** is only used when developing the game, and the **DataPath** is used when the game is built to avoid errors that are further explained in the How to Build Your Game section.

This should also be done if using the **CurrentDir** or **CurrentDevPath** paths.

The **EngineCamera** is the scene camera.

The **TestSphere** is a GameObject loaded before all others and can be used for testing. Although loaded, the **TestSphere** does not get added to the scene via the SceneManager. It is up to the user to QuickCopy and append the copy to the scene.

InputManager

The InputManager is modelled to be as similar to the Unity 'Input' class. It has all the information about the mouse and keyboard.

Properties:

- Vector2 MousePosition;

Functions:

- CursorState GetCursorState()
- Bool GetKeyDown(Keys key)
- Bool GetKey(Keys key)
- Bool GetKeyUp(Keys key)
- Bool GetMouseButtonDown(MouseButton button)
- Bool GetMouseButton(MouseButton button)
- Bool GetMouseButtonUp(MouseButton button)

The **CursorState** is whether the cursor is normal, hidden, or grabbed.

The **GetXDown** functions return true or false only on the tick that the key / button has been initially pressed.

The **GetX** functions return true or false every tick that the key / button is pressed.

The **GetXUp** functions return true or false only on the tick that the key / button has been released.

Mesh

The mesh is a class that describes a model / shape.

Properties:

- String LoadedModel
- Bool Optimised
- List<SteelVertex> Vertices
- List<SteelEdge> Edges
- List<SteelTriangle> Triangles

The lists **should not** be ever added to by the user directly, instead the functions below should be used.

Functions:

- Void AddTriangle(SteelTriangle Triangle)
- Void AddTriangleQuickly(SteelTriangle Triangle)
- Void AddTriangleRapid(SteelTriangle Triangle)
- Void AddTriangleRapid(Vector3 p1, Vector3 p2, Vector3 p3, Mesh MeshParent)
- Void RefreshTriangles()
- Int[] GetIndices()
- Void MergeDuplicates()
- Void SetColour(Vector3 Colour)

All of the **AddTriangle** functions add a triangle to the mesh. The base function adds the triangle, then it refreshes all of the triangles, and attempts to remove all duplicate vertices (vertices that have the same position, colour, and texture coordinates).

The **Quick** version of the **AddTriangle** function skips the removing of duplicate vertices and so can produce unoptimized or strange-looking results.

The **Rapid** version of the **AddTriangle** function works slightly differently than the **Quick** version of the **AddTriangle** function but uses a faster method. This will still create the same results as the **Quick** version of the **AddTriangle** function. This function has 1 ov*erload that takes three points and a mesh-parent, instead of a **SteelTriangle**. This means that the added triangle will have no texture coordinates or colour.

The **RefreshTriangles** function is a slower but not super slow function. This function simply reloads all of the triangles.

The **GetIndices** function returns the indices of the mesh.

The **MergeDuplicates** function attempts to find and remove the vertices that are at the same location, share the same colour and have the same texture coordinates.

The **SetColour** function takes a colour as a Vector3 representing R,G,B with the X,Y,Z components in that order. It then sets the colour of **all** of the vertices to that colour and refreshes the triangles of the mesh and removes the duplicates.

SteelTriangle

The SteelTriangle is a class representing a triangle.

Properties:

- SteelEdge[3] Edges

The **Edges** property contains all 3 **SteelEdges** that make up the triangle.

Functions:

- Void SetColour(Vector3 Colour)
- Void SetVertexColour(Int Index, Vector3 Colour)
- SteelVertex GetVertex(Int Index)

The **GetVertex** function returns a **SteelVertex** that contains all of the vertex data of the vertex with the given index.

SteelVertex

The SteelVertex is a class that represents a vertex and its colour and texture coordinates (UVs).

Properties:

- Vector3 Position
- Vector3 Colour
- Vector2 TexCoord

Functions:

- Void AssignUV(Vector2 RefPoint)
- Float[] GetVertexData()

The **AssignUV** function assigns the **TexCoord** of the vertex to the given **RefPoint**.

The **GetVertexData** returns all of the vertex data as a float array in the layout:

Index 0 : X position of the vertex

Index 1 : Y position of the vertex

Index 2 : Z position of the vertex

Index 3 : Red colour of the vertex

Index 4 : Green colour of the vertex

Index 5 : Blue colour of the vertex

Index 6 : X texture coordinate of the vertex

Index 7 : Y texture coordinate of the vertex

SteelEdge

The SteelEdge is a class that represents an edge between two **SteelVertices**.

Properties:

- Int StartVertexIndex
- Int EndVertexIndex

Functions:

- SteelVertex GetVertexData(Int index)

The **GetVertexData** function returns a **SteelVertex** that contains all of the vertex data of the vertex with the given index.

Time

The Time class contains information about the time used in the game.

Properties:

- Double UpTime
- Float TimeScale

The **UpTime** property is how long the application has been open for stored as a double.

The **TimeScale** is fairly similar to Unity's TimeScale. The default value is 1, and it can be changed to any positive value.

The reason why the Time class doesn't contain a property for DeltaTime is because this is already passed to all Tick functions.

ObjImporter

The ObjImporter imports .Obj files and creates **Meshes** out of them. It also creates .SEO (Steel Engine Object) files out of the .Obj files for faster loading times.

Functions:

- Mesh LoadSEO(String Name, Bool Optimised)
- Mesh LoadOBJ(String Name, Bool Optimised)
- Mesh LoadOBJFromPath(String Path, Bool Optimised)

The **LoadSEO** function returns a **Mesh** from a .SEO file.

The **LoadOBJ** function returns a **Mesh** from a .Obj file, but it will attempt to load from a .SEO beforehand if it exists.

The **LoadOBJFromPath** works the same way that the **LoadOBJ** function, but takes a whole path including extension instead of just the name.

Rigidbody

The Rigidbody class is a basic physics **component** that gives very basic rigid body physics to the attached **GameObject**.

Properties:

- Vector3 Velocity

The Rigidbody really doesn't have much at the moment, not even any functions. But in future it will updates it will be expanded upon and improved. The current version has a very primitive physics system so if a more complex one is needed then it will be up to the user to make any more advanced physics. Although primitive, the physics can still be useful so it is likely that for more basic games the user will not have to create their own physics.

LightObject

The LightObject class simply describes a position, colour and an intensity value of a light. All lights are currently point lights, and it must be noted that lighting is currently very primitive.

LightManager

The LightManager class handles all of the **lights** in the scene. It must also be noted that the scene can only have 1 light at the moment, this **will** be improved later in development. It also must be noted that light seems to act strangely in cases where the camera rotates a lot and changing the FOV of the camera seems to change the intensity of the active light. Although only 1 light can be active in the scene, multiple can be placed into the LightManager's **Lights** list. Overall I wouldn't rely on any good lighting from Steel Engine in the current version, but in future versions it will be much better.

Properties:

- List<LightObject> Lights

The **Lights** list stores all of the lights created but only the 0th index gets activated and placed into the scene. Unlike the **CollisionManager's colliders list** the **Lights** list doesn't automatically detect created lights and add them to the list, so the user must manually add any lights they create to the list.

Functions:

- LightObject AddLight(Vector3 Position, Vector3 Colour)
- LightObject AddLight(Vector3 Position, Vector3 Colour, Float Intensity)
- Vector3 ColourFromRGB255(Vector3 RGB255)
- Vector3 ColourFromRGB255(Float RGB255X, Float RGB255Y, Float RGB255Z)

The **AddLight** function creates and adds a new **LightObject** to the **Lights** list. The base function assumes the intensity to be 1 (max), while the overload allows the user to select the light's intensity from 0 – 1.

The **ColourFromRGB255** function takes an RGB255 Vector3 colour value and returns a RGB1 Vector3 colour value. The overload just allows the user to not have to have a Vector3 to pass and instead takes the red, green and blue values separately.

How to Use Steel Engine

Using Steel Engine should be fairly similar to using Unity, to make it as easy as possible to transition from one to another. Although, it should be noted that the engine is very early in its development and so lacks a lot of features, but these will be implemented in future versions.

The Files

All files that will be used in the game should be organised into the correct folders. For example, if you have a texture you would like to add to your game it should be placed in the 'Resources/Textures' folder for it to be usable in your game.

The Scene File

The scene file is of type .SES and contains all information about the scene. The formatting of the scene file is very important. The formatting of the scene file is shown on the next page.

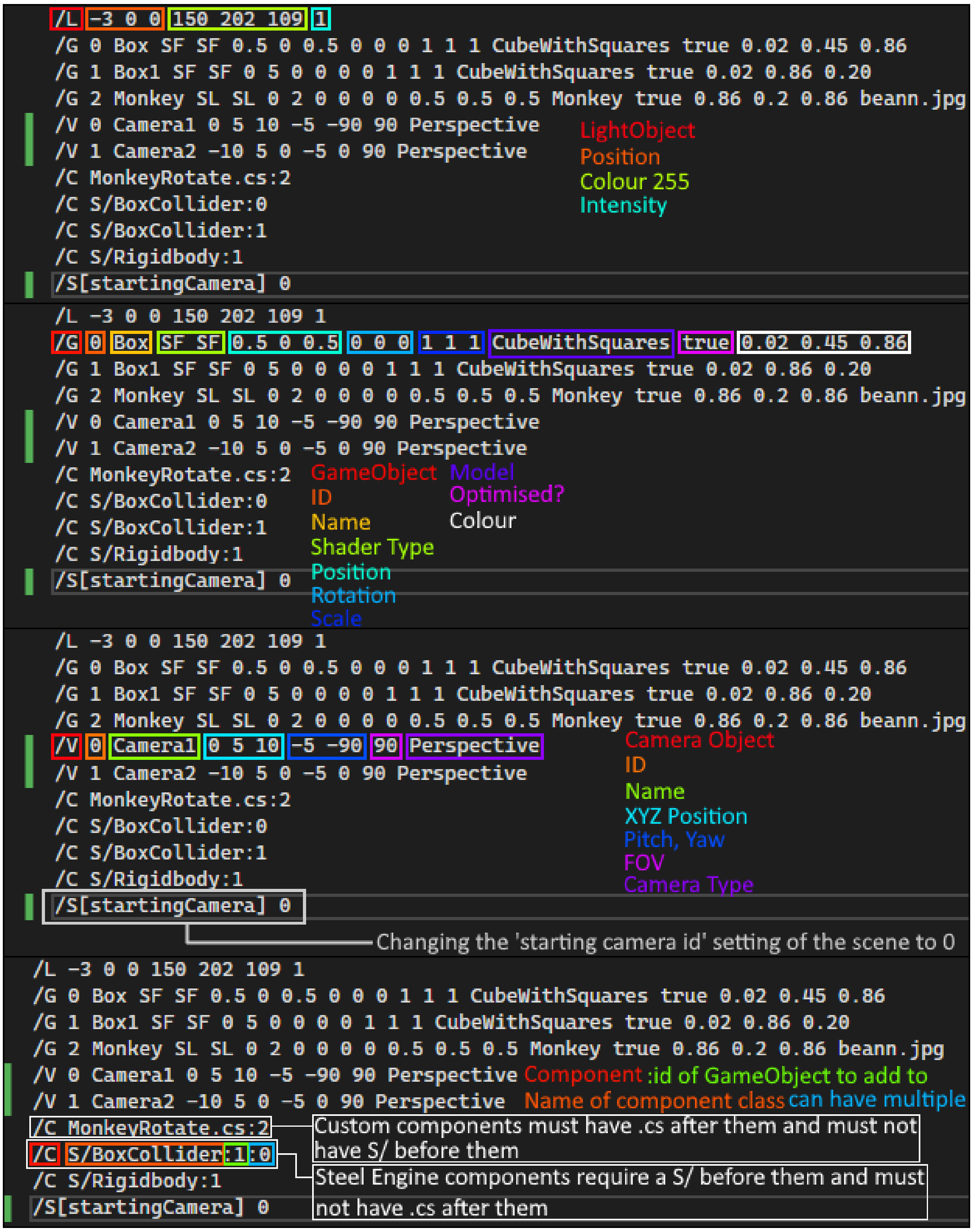
The Editor

The editor is very simple at the moment. It has a hierarchy of the **GameObjects** in the scene, and a very basic move tool. If you have made changes to the scene from within the editor, you can press Ctrl + S to save your changes to the scene file. At the moment the saving system is a little odd, and so you must first hold Ctrl and then press S to save, the other way around doesn't work.

You can move the engine camera with WASD + E and Q to move up and down respectively, along with the mouse to look around.

To toggle unlock / lock your mouse to the screen, press escape.

Like all of the Steel Engine UI the editors UI also only works properly when in 1920x1080 resolution. More supported resolutions might come in future.



```
/L -3 0 0 150 202 109 1
/G 0 Box SF SF 0.5 0 0.5 0 0 0 1 1 1 CubeWithSquares true 0.02 0.45 0.86
/G 1 Box1 SF SF 0 5 0 0 0 0 1 1 1 CubeWithSquares true 0.02 0.86 0.20
/G 2 Monkey SL SL 0 2 0 0 0 0 0.5 0.5 0.5 Monkey true 0.86 0.2 0.86 beann.jpg
/V 0 Camera1 0 5 10 -5 -90 90 Perspective
/V 1 Camera2 -10 5 0 -5 0 90 Perspective
/C MonkeyRotate.cs:2
/C S/BoxCollider:0
/C S/BoxCollider:1
/C S/Rigidbody:1
/S[startingCamera] 0
```

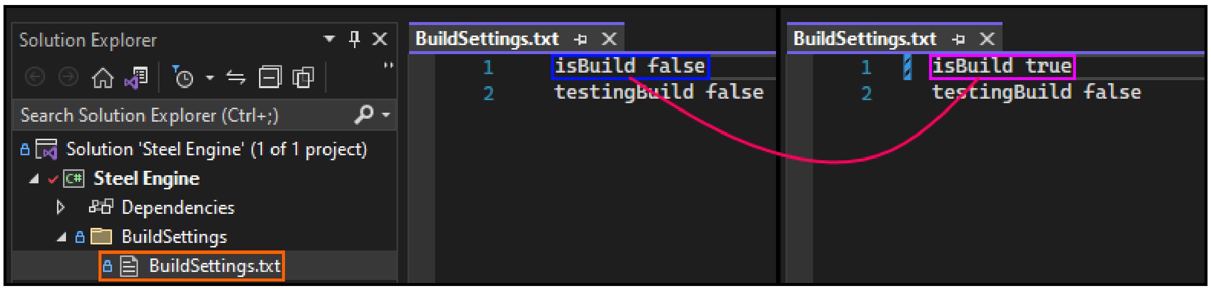
Changing the 'starting camera id' setting of the scene to 0

Custom components must have .cs after them and must not have S/ before them

Steel Engine components require a S/ before them and must not have .cs after them

How to Build Your Game

Once you have a game that you are happy with and want to share with others, you can change the 'isBuild' value in the 'BuildSettings.txt' file to true.

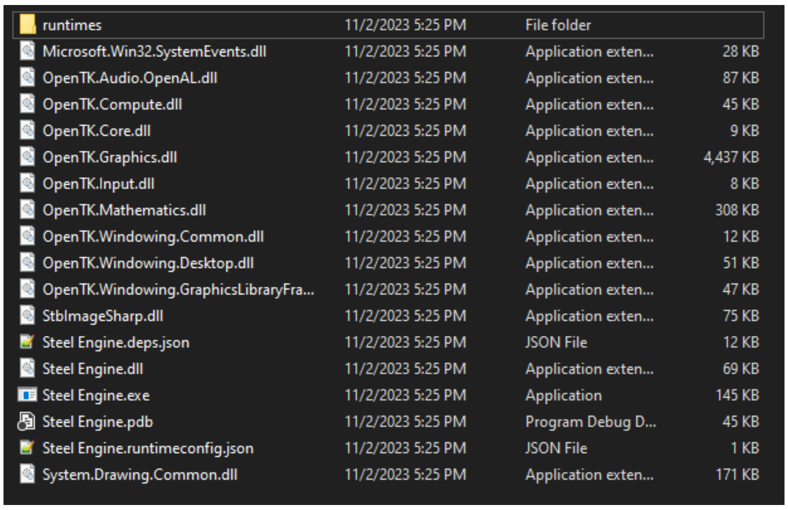


```
isBuild false
testingBuild false
```

```
isBuild true
testingBuild false
```

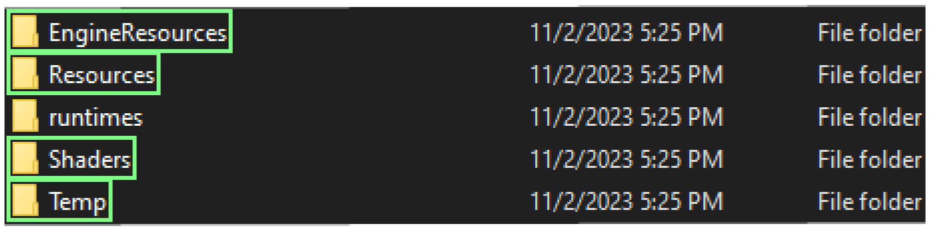
Keep in mind that if you would like to see what the game would look like while still making changes to the game, you should enable 'testingBuild'. This would allow you to make changes to your game, build it, and continue to iterate and change the game. If you don't enable this and attempt to change the game after having built it, it won't see changes to the game files. So only have testingBuild on false if this is the very last build you make before sending out the game files. If you forget this on accident there is a way to correct it that is explained later.

Once these settings have been setup, you can build your game. This will be different depending on what IDE you use, but in general you should use what ever build feature your IDE provides, and that should create some files that look like these:



runtimes	11/2/2023 5:25 PM	File folder	
Microsoft.Win32.SystemEvents.dll	11/2/2023 5:25 PM	Application exten...	28 KB
OpenTK.Audio.OpenAL.dll	11/2/2023 5:25 PM	Application exten...	87 KB
OpenTK.Compute.dll	11/2/2023 5:25 PM	Application exten...	45 KB
OpenTK.Core.dll	11/2/2023 5:25 PM	Application exten...	9 KB
OpenTK.Graphics.dll	11/2/2023 5:25 PM	Application exten...	4,437 KB
OpenTK.Input.dll	11/2/2023 5:25 PM	Application exten...	8 KB
OpenTK.Mathematics.dll	11/2/2023 5:25 PM	Application exten...	308 KB
OpenTK.Windowing.Common.dll	11/2/2023 5:25 PM	Application exten...	12 KB
OpenTK.Windowing.Desktop.dll	11/2/2023 5:25 PM	Application exten...	51 KB
OpenTK.Windowing.GraphicsLibraryFra...	11/2/2023 5:25 PM	Application exten...	47 KB
StbImageSharp.dll	11/2/2023 5:25 PM	Application exten...	75 KB
Steel Engine.deps.json	11/2/2023 5:25 PM	JSON File	12 KB
Steel Engine.dll	11/2/2023 5:25 PM	Application exten...	69 KB
Steel Engine.exe	11/2/2023 5:25 PM	Application	145 KB
Steel Engine.pdb	11/2/2023 5:25 PM	Program Debug D...	45 KB
Steel Engine.runtimeconfig.json	11/2/2023 5:25 PM	JSON File	1 KB
System.Drawing.Common.dll	11/2/2023 5:25 PM	Application exten...	171 KB

Next you **must** run the 'Steel Engine.exe' file and it should finish the build process. If the game opens and runs and you see the following new folders, you have successfully built your game. The new folders you should see:



EngineResources	11/2/2023 5:25 PM	File folder
Resources	11/2/2023 5:25 PM	File folder
runtimes	11/2/2023 5:25 PM	File folder
Shaders	11/2/2023 5:25 PM	File folder
Temp	11/2/2023 5:25 PM	File folder

If your game doesn't open it could either be an error that you have created or you could have setup / used the file paths in the InfoManager incorrectly.