Exelius Engine - A Simple Multiplatform 2D Game Engine

Introduction

This document will attempt to help you understand everything you can do with the engine, both in the Editor and in Lua to be able to make some simple 2D games. Eventually this document will evolve into a Doxygen document, so the description of the entirety of the engine will be made available, but for now, this is a hand-written document. As such, some things may not be up to date.

Please note that the engine is being actively developed and it will have bugs and crashes. If you encounter any crashes, please feel free to contact me.

Getting to know the Editor

The editor is where you can create and edit scenes and prefabs. By default, a new empty scene will be loaded into the editor. The editor functions very similarly to the Unity 3D engine, so users familiar with Unity will find navigating this editor to be \*hopefully\* easy and intuitive.

The `File` button, located in the top left in the editor taskbar has a small menu that allows the following functionality:

- Start a new scene - This will clear/unload the current scene and any unsaved changes will be lost.

- Open a scene - This will open your systems file browser and allow you to load a `.excene` asset, which contains scene data.

- Save a scene - This will open your systems file browser and allow you to save the current scene with a name to your file system.

- Edit editor settings - This provides an additional menu of some settings that are changeable.

- Toggle Showing Physics Debug Colliders.

- Toggle the ImGui Demo window visibility.

- Toggle the engine Debug Stats window visibility.

- Toggle the Asset Browser window visibility.

- Toggle the Scene Hierarchy window visibility.

- Toggle the Inspector window visibility.

- Toggle Fullscreen mode.

- Edit the Editor Style/Theme.

\*Please note that changes to any editor settings are not currently saved after closing the editor.\*

There are two modes that the editor can run in: `Edit` and `Play`.

- `Edit` is the state in which you can actively make changes to and edit scenes and prefabs. In `Edit` mode, the scene view has a build in camera that is independent from any cameras within the scene, and this camera will not affect how the scene is viewed when in play mode. This camera can be moved by selecting the Scene View panel, holding the `alt` key, and holding either the `Left Mouse Button`, `Right Mouse Button`, or `Middle Mouse Button`.

- `Play` is where the engine will simulate any physics and script behaviors, and any changes made within the editor that effect a scene while this editor is in the `Play` state will not be saved. These states can be toggled by pressing the `Play` or `Stop` button in the editor taskbar to the right of the `File` button.

There are a few windows available to you when you first open the editor, and each of them serve a special purpose. These windows will be explained in detail on their use and functionality below. Any of these windows can be moved, undocked, redocked, or dragged into a totally separate window or monitor at any time. Any changes made to each window \*will persist after closing the editor\*.

\*\*The editor and engine are still being worked on, so there are likely a lot of features that are missing that you might like to see, or you might encounter a bug. If you do, please reach out to me so that I can add your suggested feature or squash your bug.\*\*

Scene Hierarchy

The scene hierarchy panel lists all the GameObjects in the current scene by name. By `right clicking` in the empty space within this panel, you can create a new GameObject that will be added to the current scene. Right clicking on a GameObject will allow you to delete it from a scene. Selecting a GameObject with the `Left Mouse Button` will display that GameObject in the Inspector panel. You can save GameObjects in this panel by dragging them into the Asset panel or loaded by performing the opposite.

Scene View

The scene view panel is your view of the current scene. When in `Edit` mode, it uses a built in Editor camera. In `Play` mode, it will use the views from any Camera Components that are in the currently simulated scene. For more information on how the `Edit` and `Play` states effect the editor, please see the Getting to know the Editor topic above. Selecting a GameObject in the scene view panel will display that GameObject in the Inspector Panel.

Inspector

The inspector panel is what is used to describe a GameObject. This panel will list any Components of the currently selected GameObject.

This panel is where you can make any changes to a GameObject. All GameObjects by default will always have two accessible Components, the Name Component, and the Transform Component. In this panel you can add/edit any other the components listed below.

Asset Browser

The asset panel is your view into the assets folder of your project. This will mirror your filesystem exactly. Supported filetypes will have a dedicated icon, and others will be marked with a '?' icon. These assets can be dragged into various parts of the editor to different effect:

- `.exobj` files are Prefabs and can be dragged into the Scene Hierarchy panel to instantiate a copy of that prefab. Changes made to a prefab in a scene or at runtime will NOT be saved.

- `.excene` files are Scenes and can be dragged into the Scene View panel to load a scene.

- `.png`, `.bmp`, or `.jpg` files are images and can be dragging into the Inspector panel of a selected GameObject to add a SpriteRenderer component or change a SpriteRenderer components texture.

- `.lua` files are scripts and can also be dragged into the Inspector panel to add a LuaScriptComponent or change the script of a current LuaScriptComponent.

\*Other asset types will be supported soon!\*

Debug

The Debug panel displays some basic debug information, such as FPS and rendering stats. This panel is disabled by default and can be made visible from the `File->Editor Settings` menu in the taskbar.

ImGui Demo Window

The ImGui Demo panel displays ImGui's demo window, this window is used purely to develop new features to the Exelius Editor. This panel is disabled by default and can be made visible from the `File->Editor Settings` menu in the taskbar. In the future, this will be stripped from the engine entirely.

Scene

A scene is a `Level` in your game. It contains everything within it to make the game run and be interactable. Currently only one scene can be loaded or run at a single time in the editor. This is expected to change in the future.

You can run a scene by pressing the `Play` button in the top left corner of the editor. While the scene is running, there are some features that are silently disabled in the editor, including adding components to an object from the inspector and saving/loading scenes.

Press the `Stop` button to stop a scene. The state of GameObjects may change during the scene's runtime, but once a scene is stopped, all changes are reverted. Currently there is no way to store changes made to an object after a scene finishes. However, GameObjects can still be made into prefabs while a scene is running. See Prefabs to see why this might be useful.

Prefab

A prefab is a saved version of a previously created GameObject. Think of it as a template or blueprint. It can be used to spawn a copy of that GameObject into the scene by dragging the asset into the Scene Hierarchy Panel, or by calling Instantiate in Lua. To save an object as a prefab, simply drag it from the Scene Hierarchy panel into the Asset Browser panel to the desired directory. This can even be done while the scene is running! This could be useful to save an object that has had its transform values changed at runtime, so you could save a prefab and it will instantiate at that position in the scene by default.

In the future there will be a dedicated prefab editor panel, but until then, the only way to save a prefab is to perform this drag operation.

Getting to know GameObjects & Components

- Camera: Creates a view into the game-world. NOTE: Multiple \*simultaneous\* cameras is not currently supported!

- Is Active - Whether this camera is active.

- Size - The viewport size of the camera.

- Fixed Aspect Ratio - Whether the aspect ratio is fixed. \*Not currently used\*

- Viewport Rect - Where the view is displayed in the screen space. \*Bugged\*

- Transform: The position, size, rotation of a GameObject and its components.

- Translation - Position. \*Z determines sorting order\*

- Rotation - \*X and Y rotations are not useful in 2D\*

- Scale - \*Z Scale is not useful in 2D\*

- Rigidbody

- Body Type - The type of body that the physics system uses.

- Static - Immoveable.

- Dynamic - Moveable.

- Is Fixed Rotation - Can this body rotate?

- Is Affected By Gravity - Is this body effected by gravity?

- Gravity Scale - How much is this object effected by gravity?

- Box Collider

- Offset - The collider offset from the center of the GameObject. \*Bugged\*

- Size - The collider size difference to the GameObject.

- Density - The "Heaviness" of the GameObject.

- Friction - The amount that any "sliding" effect is lost over time.

- Restitution - The "bounciness" of an GameObject.

- Restitution - The "bounciness loss" over time of a GameObject.

- Circle Collider

- Offset - The collider offset from the center of the GameObject. \*Bugged\*

- Radius - The collider size difference to the GameObject.

- Density - The "Heaviness" of the GameObject.

- Friction - The amount that any "sliding" effect is lost over time.

- Restitution - The "bounciness" of an GameObject.

- Restitution - The "bounciness loss" over time of a GameObject.

- Sprite Renderer

- Color - The color/transparency or tint of a GameObject.

- Texture - The path to the `.png`, `.bmp`, or `.jpg` texture on this GameObject.

- Tiling Multiplier - The tiling effect on the texture.

- Circle Renderer

- Color - The color/transparency or tint of a GameObject.

- Thickness - The thickness of the outline. 1 being solid, 0 being non-existent.

- Fade - The fade-out transparency of a circle.

- Lua Script

- Script - The path to the `.lua` script asset.

Getting to know the Lua Scripting Engine

Understanding Lua is core to understanding how to write a script for Exelius. The Engine has a few core entry points into a Lua script that \*must\* be adhered to for the engine to interface with it. They are the following methods:

- `OnInitialize()` - Called when the scene begins to play. Like Unity's `Awake` method.

- `OnUpdate()` - Called every frame.

- `OnDestroy()` - Called when the owner of this script gets destroyed.

- `OnCollisionEnter()` - Called when the owner of this script has a rigidbody & collider component and begins a collision.

- `OnCollisionExit()` - Called when the owner of this script has a rigidbody & collider component and stops a collision.

Important Notes on Scripts:

- They are currently being actively developed and are VERY likely to change and break.

- Always create the local table and return it to the engine.

- \*Bug\* Currently the OnDestroy function is only called when a scene ends on objects that are still alive.

Lua Script Functions

The following functions are accessible to Lua from the Exelius engine. These are C++ classes and functions.

- GameObject - A class representing a GameObject. The 'self' keyword has a member 'gameObject' that acts on the owning GameObject.

- `CreateGameObject("Name")` - \*This is a global function.\*

- `InstantiateGameObject("Name")` - \*This is a global function.\*

- `DestroyObject(gameObjectToDestroy)` - \*This is a global function.\*

- `Instance()`

- `Destroy()`

- `GetName()`

- `GetGUID()`

- `AddBoxColliderComponent()`

- `AddCircleColliderComponent()`

- `AddCameraComponent()`

- `AddCircleRendererComponent()`

- `AddSpriteRendererComponent()`

- `AddRigidbodyComponent()`

- `AddLuaScriptComponent()`

- `RemoveBoxColliderComponent()`

- `RemoveCircleColliderComponent()`

- `RemoveCameraComponent()`

- `RemoveCircleRendererComponent()`

- `RemoveSpriteRendererComponent()`

- `RemoveRigidbodyComponent()`

- `RemoveLuaScriptComponent()`

- `GetBoxColliderComponent()`

- `GetCircleColliderComponent()`

- `GetCameraComponent()`

- `GetCircleRendererComponent()`

- `GetSpriteRendererComponent()`

- `GetRigidbodyComponent()`

- `GetLuaScriptComponent()`

- Components - These match the ones available in the editor.

- `BoxColliderComponent`

- `offset`

- `x`

- `y`

- `size`

- `x`

- `y`

- `density`

- `friction`

- `restitution`

- `restitutionThreshold`

- `CircleColliderComponent`

- `offset`

- `x`

- `y`

- `radius`

- `density`

- `friction`

- `restitution`

- `restitutionThreshold`

- `CameraComponent`

- `isActive`

- `isFixedRotation`

- `viewportRect`

- `TransformComponent`

- `translation`

- `x`

- `y`

- `z`

- `rotation`

- `x`

- `y`

- `z`

- `scale`

- `x`

- `y`

- `z`

- `LuaScriptComponent`

- `script` - This is the important one here. You can call the scripts functions if they are available. You must know what those functions are currently as this system is under construction.

- `RigidbodyComponent`

- `bodytype`

- `gravityScale`

- `isFixedRotation`

- `SetPosition(myVec2)`

- `SpriteRendererComponent`

- `color`

- `textureResource`

- `textureTilingMultiplier`

- `CircleRendererComponent` \*Bugged\*

- `color`

- `thickness`

- `fade`

- Physics

- The `OnCollision` functions will pass in a `CollisionData` object. These are the data members within.

- `thisGameObject` - a reference to this game object. \*Not necessary?\*

- `otherGameObject` - a reference to the other Game Object involved in the collision.

- `isCollisionEnter` - was this collision just started? \*Not necessary?\*

- Utilities

- Color - a struct with the following members.

- `new.Color()` - White Color Constructor.

- `new.Color(color)` - Constructor setting r, g, b to the same values.

- `new.Color(r, g, b)` - Constructor setting the r, g, b of the struct.

- `new.Color(r, g, b, a)` - Constructor setting all the values of the struct.

- `r` - Red (Between 0 and 255)

- `g` - Green (Between 0 and 255)

- `b` - Blue (Between 0 and 255)

- `a` - Alpha/Transparency (Between 0 and 255)

- Time - a namespace containing a set of global objects/functions that return time related values.

- `Time.DeltaTime` - The time the last frame took in seconds with timescale factored into it. \*Not a function but behaves like one. No '()' needed.\*

- `Time.DeltaTimeUnscaled` - The time the last frame took in seconds. \*Not a function but behaves like one. No '()' needed.\*

- `ElapsedTime` - The total time the application (including editor) has been running.

- `SetTimeScale(scale)` - How fast time moves. 0 is paused, and 0.5 is half speed. 2.0 is doubletime.

- `Log` - A global namespace for calling logging functions

- `Trace(msg)` - Prints text to the console in white.

- `Warn(msg)` - Prints text to the console in yellow.

- `Info(msg)` - Prints text to the console in green.

- `Error(msg)` - Prints text to the console in red.

- `Fatal(msg)` - Prints text to the console in red background with white text.

- `print(msg)` - Prints text to the console in white.

- `warn(msg)` - Prints text to the console in yellow.

- Random Number Generation

- `Random` - XOR128+ Random object.

- `new.Random()` - Random Constructor that is seeded with 'time(nullptr)' in the high and low bounds.

- `new.Random(seed)` - Random Constructor that is seeded with 'seed' in the high and low bounds.

- `new.Random(lowSeed, highSeed)` - Random Constructor that is seeded with 'lowSeed' and 'highSeed' in the high/low bounds.

- `Rand()` - Get a random unsigned integer.

- `FRand()` - Get a random unsigned float.

- `IRandomRange(min, max)` - Get a random integer in the given range (Inclusive).

- `FRandomRange(fmin, fmax)` - Get a random float in the given range (Inclusive).

- `new.PerlinNoise()` - A noise generator.

- `GetSeed()` - Gets the seed from the generator. Has a reasonable default.

- `SetSeed(seed)` - Sets the seed for the generator.

- `GetNoise(x, y)` - Retrieve a noise value from the generator.

- `GetNoise(x, y, seedOverride)` - Retrieve a noise value from the generator.

- `GetNoise(x, y, minX, minY, noiseInputRange)` - Retrieve a noise value from the generator.

- `GetNoise(x, y, minX, minY, noiseInputRange, seedOverride)` - Retrieve a noise value from the generator.

- `GetNormalizedNoise(x, y)` - Retrieve a normalized noise value from the generator.

- `GetNormalizedNoise(x, y, seedOverride)` - Retrieve a normalized noise value from the generator.

- `GetAverageNoise(x, y)` - Retrieve an averaged noise value from the generator.

- `GetAverageNoise(x, y, seedOverride)` - Retrieve an averaged noise value from the generator.

- `SquirrelNoise` - Another noise generator.

- `GetSeed()` - Gets the seed from the generator. Has a reasonable default.

- `SetSeed(seed)` - Sets the seed for the generator.

- `Vec2` - A 2 float struct. Members: `x`, `y`

- `Vec3` - A 3 float struct. Members: `x`, `y`, `z`

- `Vec4` - A 4 float struct. Members: `x`, `y`, `z`, `w`

- `ExeliusMath` - A namespace containing Exelius specific math functions.

- `Lerp(min, max, weight)` - Linear interpolate between values.

- `SmootherStep(x)` - Smooth the passed in value.

- `Normalize(x, min, max)` - Normalize the value within the given range.

- `Texture` - A 2D image that can be rendered with a SpriteRendererComponent.

- `new.Texture(width, height)` - Construct a new Texture with a given height and width (unsigned int).

- `Bind` - Don't use this.

- `GetWidth()` - Get the width in pixels of the texture.

- `GetHeight()` - Get the height in pixels of the texture.

- `GetRendererID()` - Don't use this.

- `SetData(data, size)` - Set the texture from a data table. This tables contents and size \*must\* match the texture width \* height. This is a slow function, and \*SHOULD NOT\* be called every frame.

- Resources - Loading and managing resources from Lua.

- ResourceTypes - The types of resources available to Lua.

- `TextureResource` - The handled resource containing a texture.

- `GetTexture()` - The texture held by this resource.

- `SetTexture(texture)` - Sets the texture to this resource.

- `ResourceID` - A class containing a path to an asset.

- `new.ResourceID(filepath)` - Constructor for a new resourceID.

- `Get()` - Get the filepath associated with this resourceID as a string.

- `IsValid()` - Is this a valid resource id?

- `ResourceHandle` A reference counted handle to a resource. A resource get unloaded when there are no remaining references.

- `Release()` - Release the resource held by this handle. If no references remain, the resource will be unloaded.

- `SetResourceID(resourceID)` - Sets the resourceID \*if not already done.\*

- `IsReferenceHeld()` - Does this handle hold an active reference?

- `LoadNow()` - Load the reference resource while blocking the main thread.

- `LockResource()` - Prevent a resource from being unloaded until Unlock is explicitly called.

- `UnlockResource()` - Allows a locked resource to be managed again.

- `CreateResource(resourceID)` - Give the resource factory a new resource to create. This ResouceHandle will own this.

- `GetAsTextureResource()` - Get the resource held by this Handle.

- Input