

# KnightLight

## Engine Design Guide

by Taylor Cadwallader and Jared Callupe Lopez

## Sophomore Game Project Fall 2023 - Spring 2024

[KnightLight on Steam](#)

[KnightLight on DigiPen Game Gallery](#)

## Team Luminosity

Evan Gray - Producer

Travis Gronvold - Tech Lead

Adam Lonstein - Programmer

Charles Winters - Programmer

Jack Love - Programmer

Taylor Cadwallader - Design Lead

Jared Callupe Lopez - Designer

# Disclaimer

This document was **collaboratively** written by both designers at Luminosity. It primarily covers the design processes used to create *KnightLight* within the custom engine. For a deeper technical breakdown of specific functionalities, please refer to the [Technical Guide](#) provided by the Luminosity programming team.

## Executive Summary

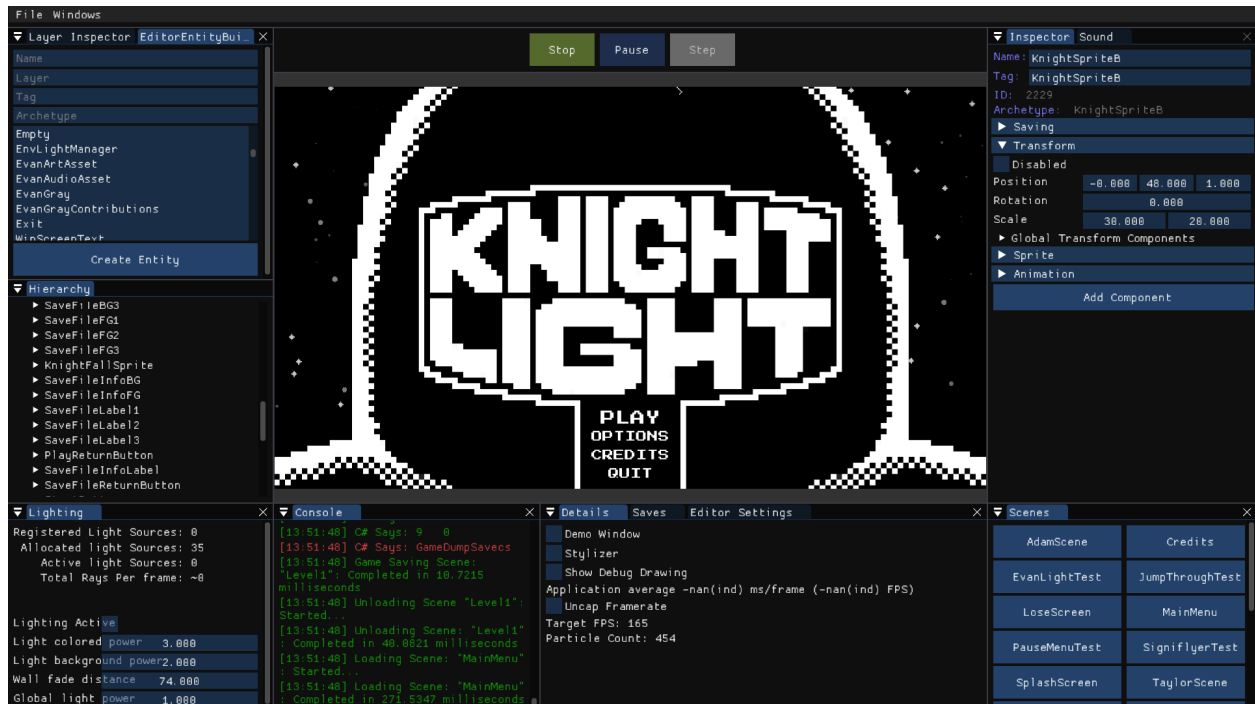
The purpose of this document is to outline how the designers worked with the engine to develop *KnightLight*, detailing the initial state of the engine and workflow, and how it evolved into the final version at the end of the GAM250 semester. This document serves as a guide based on the designers' experiences, providing insights into the creation of *KnightLight*. Both designers thoroughly reviewed the engine, explaining how its editor tools, scripting, scenes, and components functioned, with specific examples of their implementation in *KnightLight*. Key engine aspects, such as tool availability, C# scripting integration, level importing from *Tiled*, and component diversity for entities, were also evaluated. A central takeaway from this engine's development is the critical role of features that streamline workflow for both programmers and designers.

# Table of Contents

<b>Disclaimer.....</b>	<b>2</b>
<b>Executive Summary.....</b>	<b>2</b>
<b>Engine Overview.....</b>	<b>4</b>
<b>Editor.....</b>	<b>5</b>
Hierarchy:.....	5
Inspector:.....	6
Editor Entity Builder:.....	7
Scenes:.....	8
Editor Settings:.....	8
Details:.....	9
Console:.....	9
Lighting:.....	10
Sound:.....	10
<b>Scripting.....</b>	<b>11</b>
JSON:.....	11
C#:.....	14
<b>Scenes.....</b>	<b>17</b>
Scene Management:.....	17
Tiled:.....	19
<b>Components.....</b>	<b>22</b>
Managing Components:.....	22
Transform:.....	23
Colliders:.....	24
Physics:.....	26
Sprites:.....	27
Particle Emitters:.....	28
Light Sources:.....	30
Text:.....	31
Buttons:.....	32
Sliders:.....	32
Animations:.....	33
Script (C#):.....	34

# Engine Overview

In alignment with the requirements of the GAM200 and GAM250 courses, the team of programmers developed a custom engine in C++ using *Dear ImGui* for the editor's structure. To enhance usability, the programmer responsible for the *Dear ImGui* integration modeled the editor interface closely to *Unity*'s, aiming to improve familiarity and efficiency for both designers and the broader team. Additionally, the editor allowed the team to save their custom window and tab layouts, providing various options for personalization.



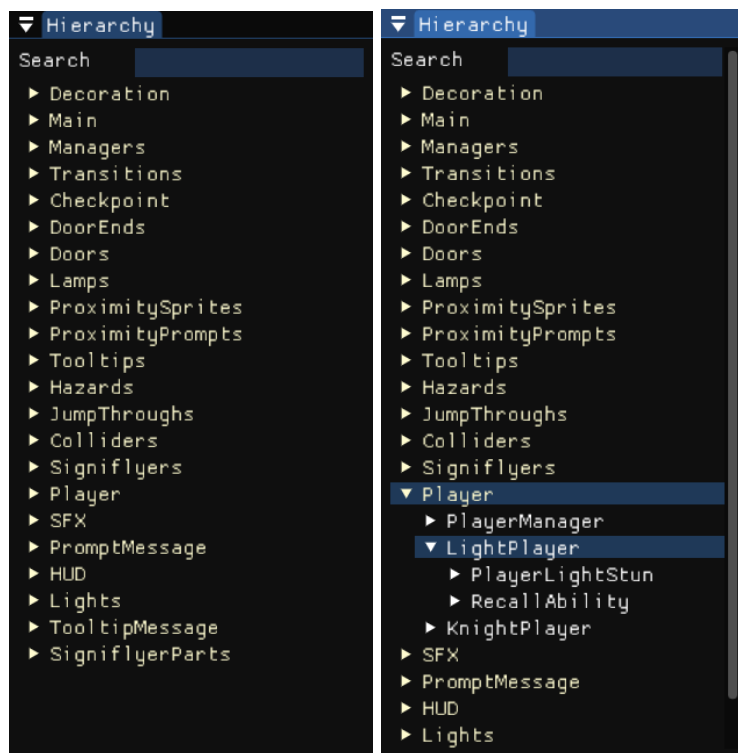
Screenshot of the Design Lead's editor layout.

# Editor

All of the following editor windows can be toggled within the *Windows* tab next to *File* tab on the top left corner of the custom engine's editor.

## Hierarchy:

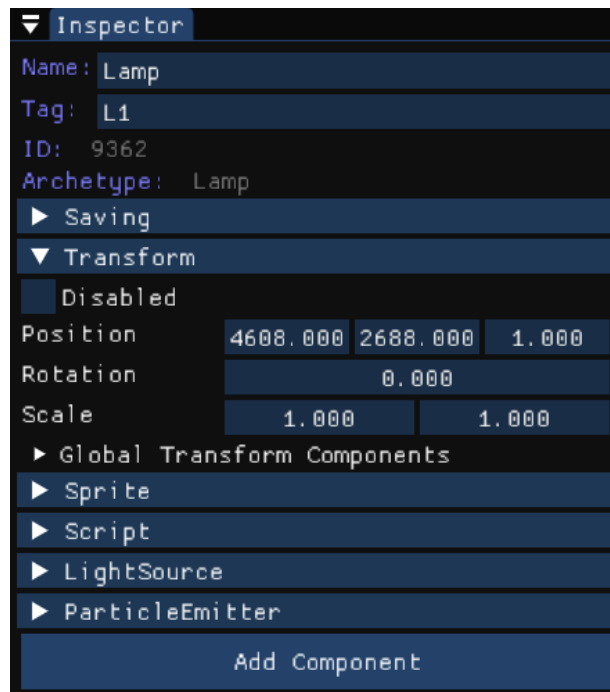
Like most engines, the *KnightLight* engine includes a *Hierarchy* that displays all entities within an open scene; in the example below, the hierarchy shows the entities in Level 1. A search bar at the top allows users to quickly locate specific entities by name. Notably, the *KnightLight* engine has a unique approach to entity grouping: all entities in the *Hierarchy* must be assigned to a layer, which functions as a parent entity or folder but does not impact gameplay. These layers, indicated in yellow text, can be expanded to reveal the entities within each group. Within layers, entity parenting is possible, as demonstrated by the “LightPlayer” entity in the screenshot below.



*Hierarchy in Level 1 with layer grouping, LightPlayer parent entity.*

## Inspector:

Similar to the *Hierarchy*, the editor includes an *Inspector* panel that displays detailed information about the selected entity. At the top of the *Inspector*, users can view the entity's "Name", "Tag", and "Archetype" (which functions similarly to a *Unity* prefab). Below this, all components attached to the entity are listed, each expandable as a dropdown for easy access. Each entity is also assigned a unique ID when the scene loads, primarily used by programmers during the initial development of entities in the custom engine. However, designers are encouraged to rely on tags or archetypes to reference specific entities within a scene for scripting purposes.



*Inspector with a Lamp entity selected.*

## Editor Entity Builder:

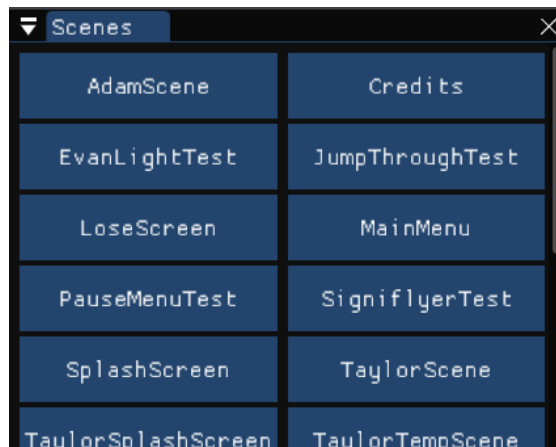
In the *KnightLight* engine, creating entities is managed through a dedicated window, rather than a single create button as seen in commercial engines like *Unity* or *Unreal*. The “Editor Entity Builder” allows users to define an entity’s name, layer, and archetype, with the tag being optional. Both names and tags can be modified at any time, while layers can be renamed or deleted in the Hierarchy via right-click. The “Archetype” field functions similarly to a search bar; users can type in the name of the desired archetype, but to ensure proper creation, the archetype must be selected from the list displayed below.



*Editor Entity Builder, hovering over the Arrow archetype.*

## Scenes:

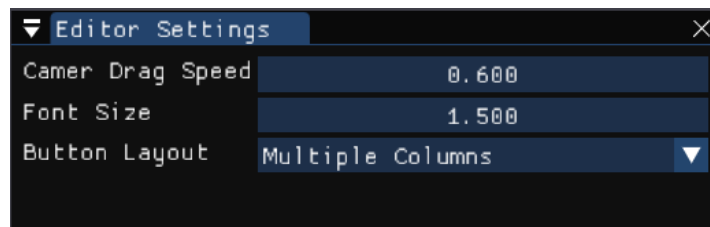
This window allows you to navigate through the list of scenes in the engine. All scenes are ordered alphabetically and stored in a 'Levels' folder. Clicking on one of them will load the selected scene on the next frame. You can also set the current scene or get the previous one through code or by using buttons.



*Scenes window*

## Editor Settings:

The *Editor Settings* window provides various customization options for users of the engine. "Camera Drag Speed" controls the rate at which the camera moves when dragged with the mouse, while the "Font Size" property adjusts the scale of text in the editor interface (not to be confused with in-game text size). Additionally, the "Button Layout" setting allows users to configure the arrangement of buttons in interfaces containing multiple options, such as scenes or saves, with choices between a single-column layout or a more compact multi-column layout.

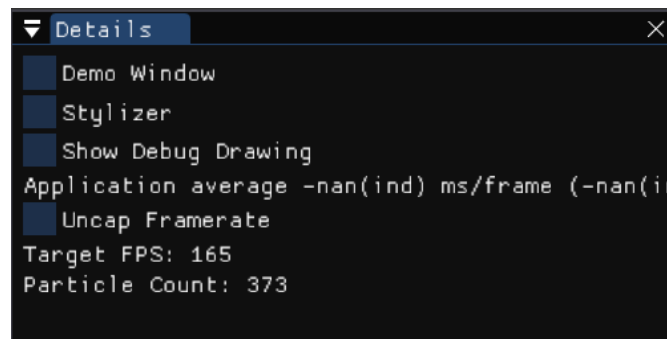


*Screenshot of the Design Lead's Editor Settings.*



## Details:

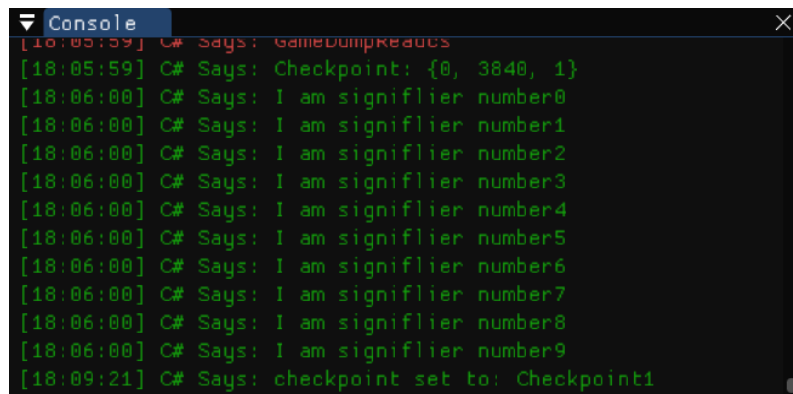
The *Details* window is dedicated to debugging. The “Show Debug Drawing” toggle displays all colliders, assisting both the physics programmer and designers during playtesting as they evaluated the limits of the engine. “Uncap Framerate” removes the FPS cap (set at 165) and allows for unlimited frames, useful primarily during particle and lighting experiments. Additionally, two other options, “Demo Window”, and “Stylizer”, open interface setup windows for *Dear ImGui*. These were only accessed by the programmer who initially configured *Dear ImGui* for the engine.



*Details window*

## Console:

The engine includes a *Console* window built-in to the editor. It's also color-coded to display the type of message, green meaning it was intentionally called from code, red meaning it's an error, and yellow meaning it's a warning. To differentiate between an engine-specific C++ log message and a C#-made message from a script, the log message states “C# Says: ” followed by the content of the message, as displayed in the image example below.

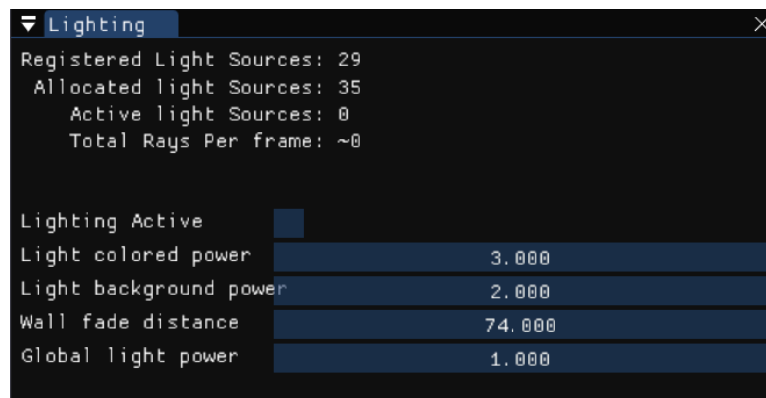


*Console window display with C# debugging for Signiflyers.*

## Lighting:

The engine includes a *Lighting* window, essential for a game like *KnightLight*, which emphasizes light-based mechanics. This window displays key debug information primarily monitored by the lighting programmer, including registers, allocated and active light sources, and ray count.

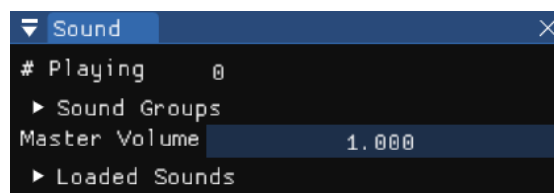
The “Lighting Active” toggle, while intended for debugging, serves as a tool for team members to view the environment during testing. Other settings in this window are global variables affecting lighting across all scenes, except for “Wall Fade Distance”, which adjusts visibility through tiles, ensuring players can see surrounding platforms and walls more clearly.



*Lighting data for Level 1.*

## Sound:

The final relevant window is the *Sound* window. This window displays the number of sounds currently playing for debugging purposes, as well as the “Sound Groups” or tracks, which list all groups/tracks utilized throughout the game. The “Master Volume” variable can be adjusted via sliders in the options menu, while the “Loaded Sounds” dropdown contains all sounds located within the engine's ‘Sound’ folder. This window was primarily used by the programmer responsible for implementing most, if not all, of the sounds, while designers focused on other tasks.



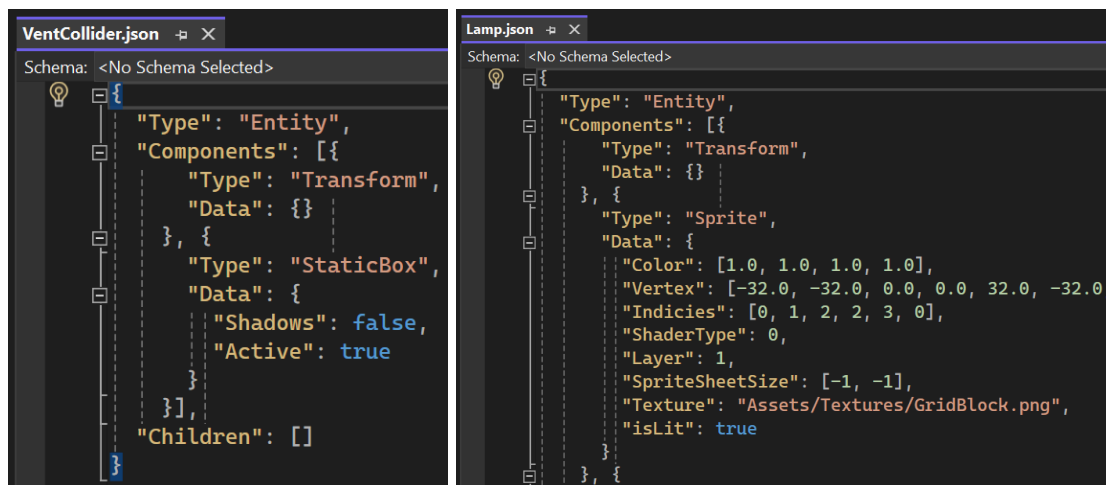
*Sound window*

# Scripting

## JSON:

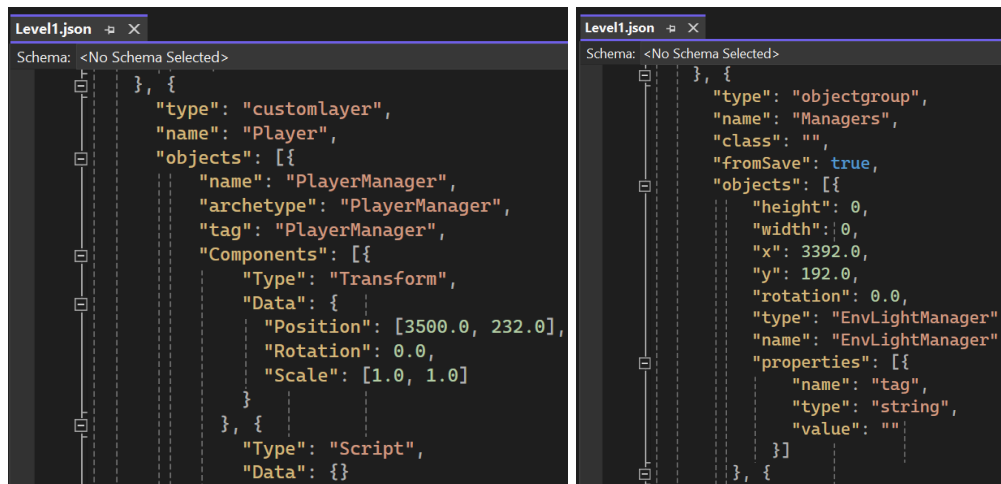
The engine utilizes JSON files to efficiently store large amounts of data that can be easily read by the engine. This approach allows for an optimized workflow, facilitating the importation of files from other tools, primarily from *Tiled* for level content. Four different types of data are saved as JSON files:

- 1) **Entities:** Each archetype in the engine is stored as a JSON file that includes all its components and individual properties. This allows for access to an archetype's JSON file outside the engine, enabling modifications to property values. For example, by accessing the *Collider.json* file, it was possible to duplicate it and create a new archetype with the same components and properties, changing only the “Shadows” value to false for the vent colliders.



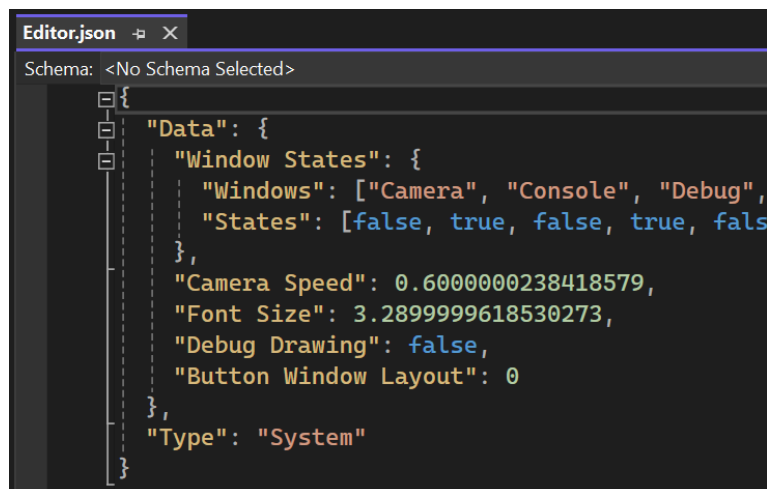
*VentCollider and Lamp JSON files.*

- 2) **Levels:** Scenes in the engine are also saved as JSON files, structured in layers containing individual objects. The engine detects whether a scene JSON file includes a tile layer and builds the level using the tileset data within the JSON file. Custom layers can be added to any scene; however, modifying *Tiled* layers is not permitted, as this would interfere with level construction. This allows for recurring data, such as managers, to be directly created in *Tiled*, avoiding the need for manual creation.



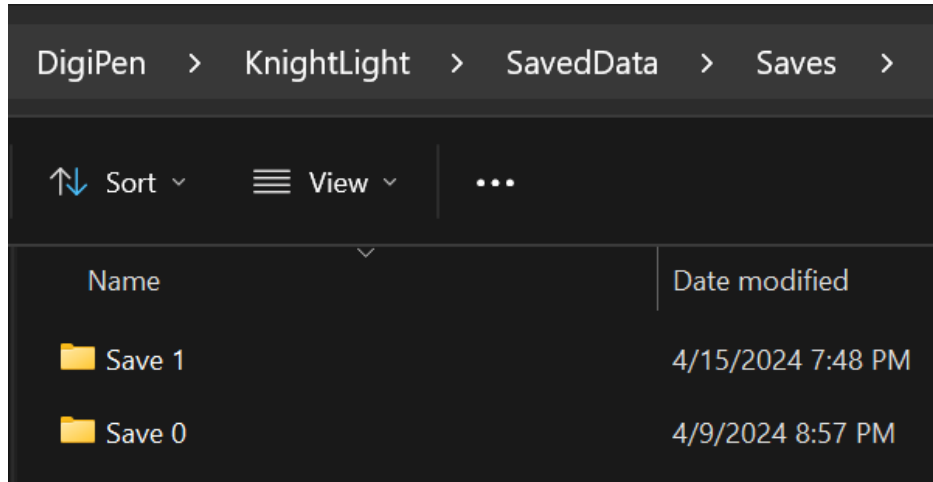
*Player and Manager layers in Level 1 JSON.*

- 3) **Systems:** Settings for various tools within the engine are also stored in JSON files. This encompasses individual engine windows, such as the camera, particle manager, and entity builder, along with editor layout settings like font size, camera speed, and open windows.



*Editor layout stored as a JSON.*

- 4) **Save Files:** Stored in the 'AppData' folder, these files are used for saving and loading player save data. Each save file consists of instances of different level JSONs included in the release build. Players are able to create up to three save files within the game, although the engine can support multiple save files.



*Save files in 'AppData' folder.*

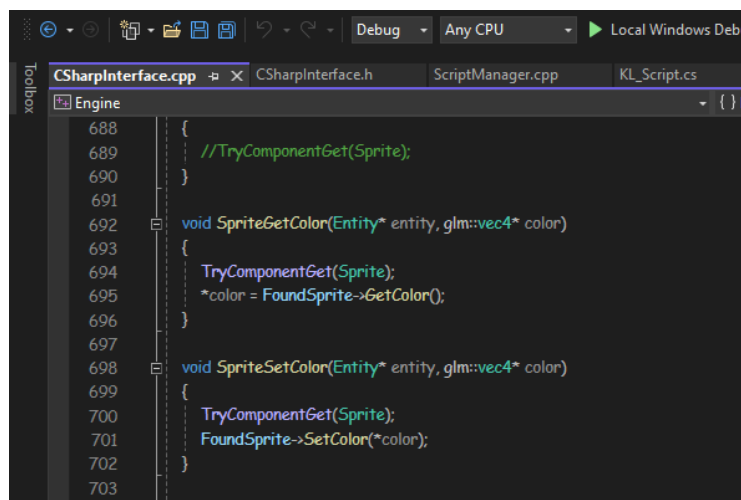
Overall, the readability of JSON files has been essential for both the game's development within the engine and as a feature in the game itself. Throughout the engine's development, various features were added and incorporated into JSON files. For instance, the inclusion of child entities within parent entities and support for multiple tile layers with different properties directly imported from *Tiled* are notable enhancements.

## C#:

The engine uses *Mono*, enabling the incorporation of C# for designers to utilize, alongside C++ from our programmers. In the first half of the semester, a programmer dedicated time to implement *Mono*, with the consensus that this investment in time would benefit designers in the long run, even if it temporarily diverted attention from other engine development tasks. As a result, the team believes that this decision has proven worthwhile given the final status of the game.

Due to the custom nature of the engine, where C++ serves as the foundation, all scripting functionalities desired by designers require cross-referencing between C++ and C#. The programmer responsible for the C# implementation noted that while the process is generally quick and straightforward, there is still a learning curve to ensure all necessary features are accessible in C#. To streamline this, the Design Lead and several programmers have offered to learn the step-by-step process of creating the C# functionalities, though the primary responsibility for this work remains with the original implementer. How it works:

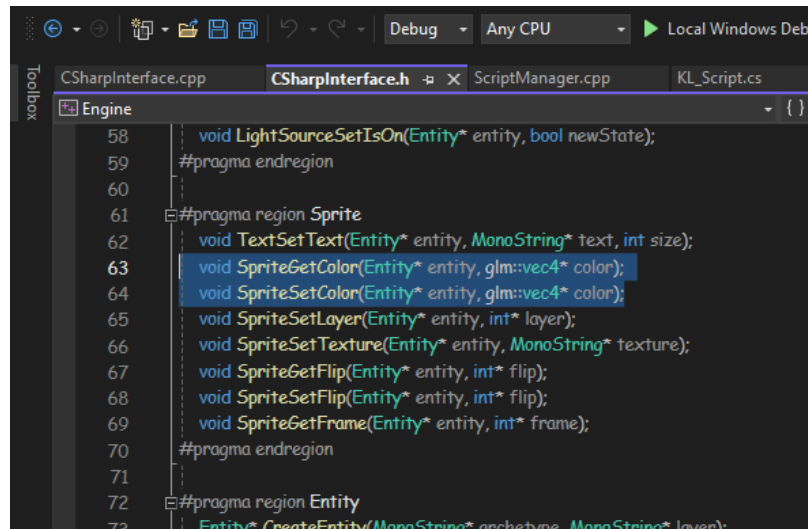
- 1) **CSharpInterface.cpp:** In *CSharpInterface.cpp*, locate the relevant section where functions need to be added. For example, if implementing functionality to get and set a sprite's color, find the section for sprites and add the functions there. The structure of these functions is fairly standardized, often using *TryComponentGet* along with *FoundSprite*, similar to *FoundTransform* for other components.



```
688 {
689     //TryComponentGet(Sprite);
690 }
691
692 void SpriteGetColor(Entity* entity, glm::vec4* color)
693 {
694     TryComponentGet(Sprite);
695     *color = FoundSprite->GetColor();
696 }
697
698 void SpriteSetColor(Entity* entity, glm::vec4* color)
699 {
700     TryComponentGet(Sprite);
701     FoundSprite->SetColor(*color);
702 }
703
```

*CSharpInterface.cpp example on Sprite Get and Set Color.*

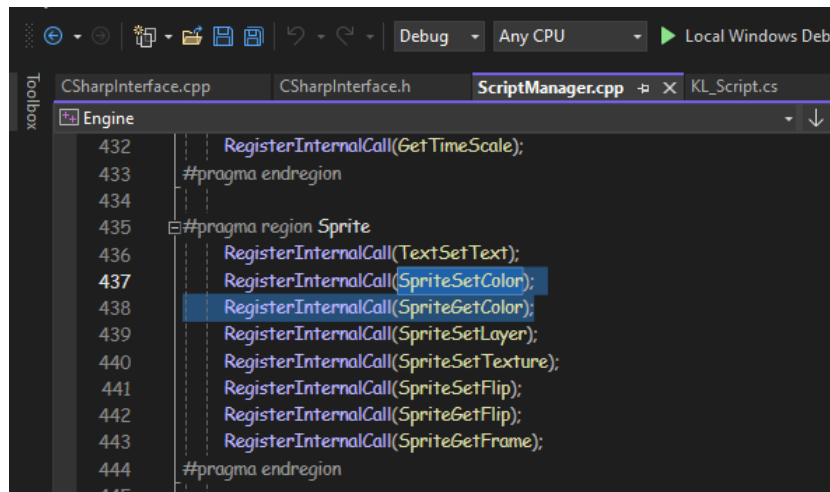
- 2) **CSharpInterface.h:** Next, open *CSharpInterface.h* and add the function there as well; skipping this step will prevent the function from working.



```
58 void LightSourceSetIsOn(Entity* entity, bool newState);
59 #pragma endregion
60
61 #pragma region Sprite
62 void TextSetText(Entity* entity, MonoString* text, int size);
63 void SpriteGetColor(Entity* entity, glm::vec4* color);
64 void SpriteSetColor(Entity* entity, glm::vec4* color);
65 void SpriteSetLayer(Entity* entity, int* layer);
66 void SpriteSetTexture(Entity* entity, MonoString* texture);
67 void SpriteGetFlip(Entity* entity, int* flip);
68 void SpriteSetFlip(Entity* entity, int* flip);
69 void SpriteGetFrame(Entity* entity, int* frame);
70 #pragma endregion
71
72 #pragma region Entity
73 Entity* CreateEntity(MonoString* archetype, MonoString* layer);
```

*CSharpInterface.h*, continuing with Sprite Get and Set Color.

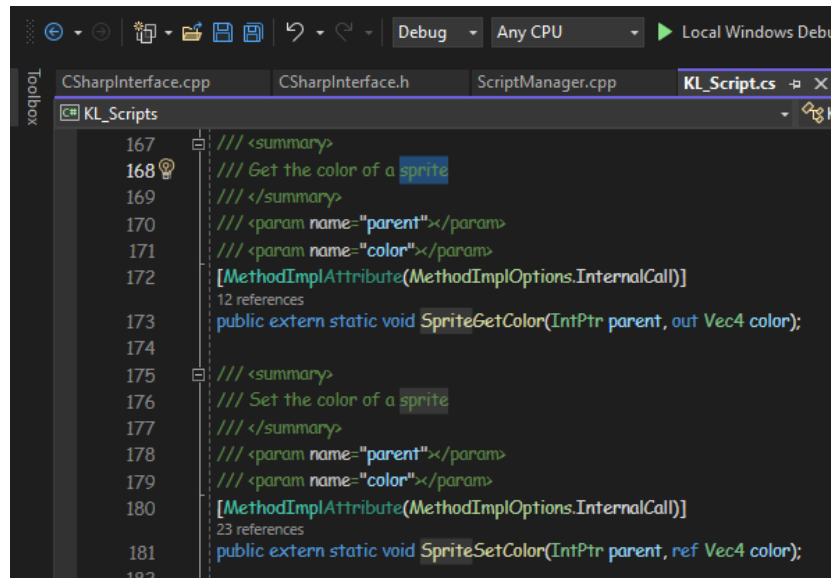
- 3) **ScriptManager.cpp:** Then, in *ScriptManager.cpp*, register the function to initialize cross-functionality with C#.



```
432 RegisterInternalCall(GetTimeScale);
433 #pragma endregion
434
435 #pragma region Sprite
436 RegisterInternalCall(TextSetText);
437 RegisterInternalCall(SpriteSetColor);
438 RegisterInternalCall(SpriteGetColor);
439 RegisterInternalCall(SpriteSetLayer);
440 RegisterInternalCall(SpriteSetTexture);
441 RegisterInternalCall(SpriteSetFlip);
442 RegisterInternalCall(SpriteGetFlip);
443 RegisterInternalCall(SpriteGetFrame);
444 #pragma endregion
445
```

*ScriptManager.cpp*, registering for C#.

- 4) **KL\_Script.cs:** Finally, in *KL\_Script.cs*, ensure that each function has the appropriate line above it, along with a summary that displays when hovering over it in C#.



```
167  /// <summary>
168  /// Get the color of a sprite
169  /// </summary>
170  /// <param name="parent"></param>
171  /// <param name="color"></param>
172  [MethodImplAttribute(MethodImplOptions.InternalCall)]
173  12 references
174  public extern static void SpriteGetColor(IntPtr parent, out Vec4 color);
175  /// <summary>
176  /// Set the color of a sprite
177  /// </summary>
178  /// <param name="parent"></param>
179  /// <param name="color"></param>
180  [MethodImplAttribute(MethodImplOptions.InternalCall)]
181  23 references
182  public extern static void SpriteSetColor(IntPtr parent, ref Vec4 color);
```

*KL\_Script.cs, final step where the description is also added to Sprite Get and Set Color.*

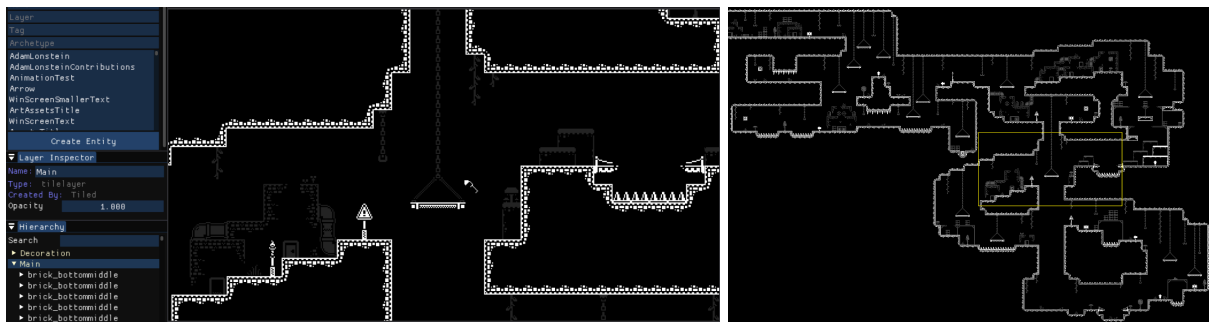
Before C# integration, designers primarily focused on prototyping and playtesting in Unity, with some engine testing and JSON adjustments. Most programmers also ended up scripting with C# to address game-specific needs.



# Scenes

## Scene Management:

Designers work in scenes using *Tiled*, an open-source level editor recommended during the initial milestone meetings. *Tiled* allows the creation of various types of layers but primarily focuses on two types commonly used in 2D games: object layers and tile layers. “Object Layers” are utilized for drawing polygons, such as boxes, which serve as collision or trigger areas, while “Tile Layers” are used to construct the environment using a tileset that contains all the available tiles.

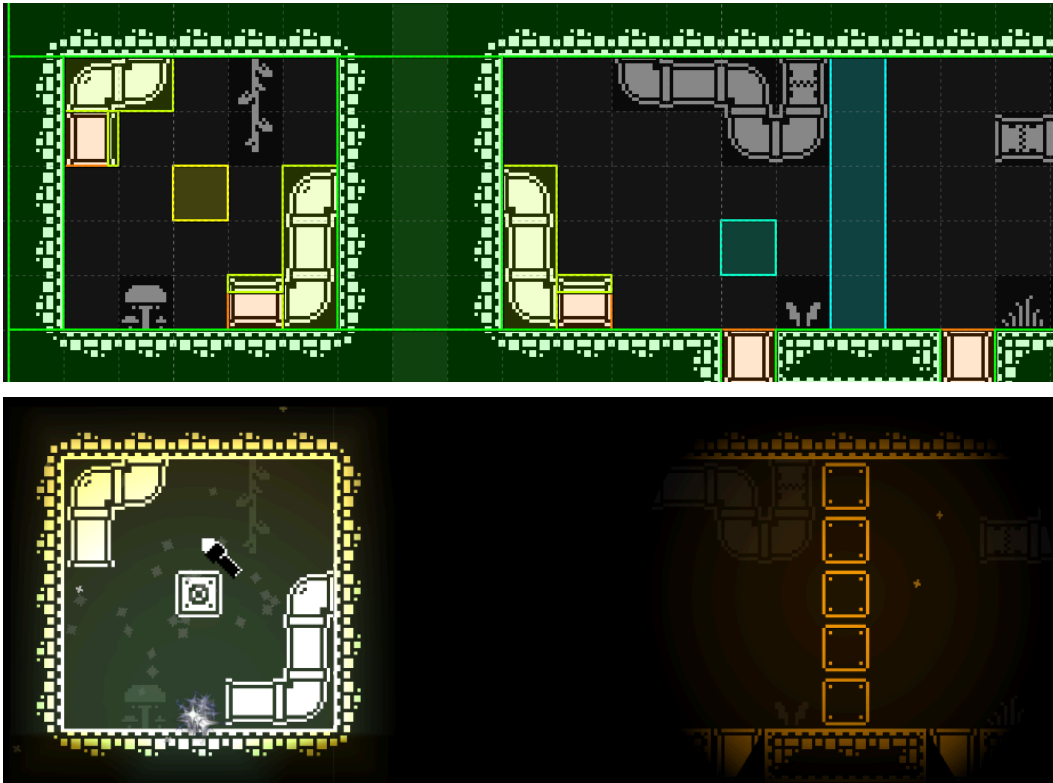


*Tilemap in engine and render.*

For non-gameplay scenes, like the Main Menu, Win Screen and Credits, these are directly created by the engine since they do not require a tilemap. Such scenes can be edited within the engine's editor and they typically require a manager script to control their behavior.

Initially, designers were able to access the level JSON files folder easily, allowing them to create new scenes by duplicating existing levels, renaming them, and deleting layers to customize their work. However, in these early stages, when it came to levels with tilemaps, designers were limited to testing platforming mechanics, as only the three main types of collider archetypes (colliders, jump-throughs, and hazards) were available for import from JSON.

This changed when the programmer responsible for level construction updated the engine's tilemap handling, allowing designers to build levels using various archetypes created in the engine, such as lamps and doors. Since then, the tilemap system has undergone multiple updates to better accommodate the needs of the level designer. Enhancements include support for multiple tile layers within the same scene, the ability to adjust the opacity of specific tile layers, and the capacity to tag entities directly from *Tiled* using properties.



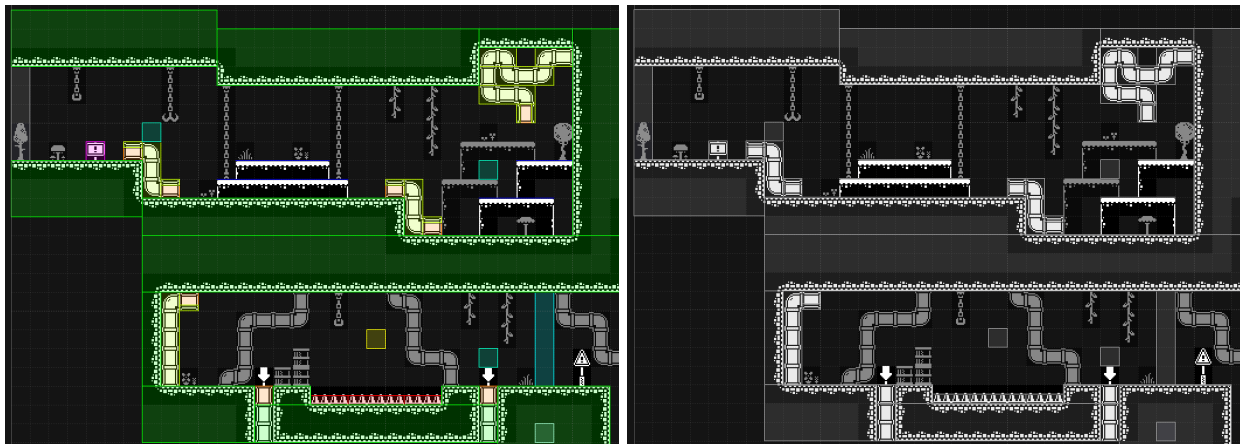
*Archetypes in Tiled and in engine.*

## Tiled:

Working in *Tiled* has significantly optimized the workflow for designers by automating many aspects of the level editing process. Previously, tasks like manually adding doors and lamps were time-consuming, especially when changes needed to be made. Designers had to re-import the JSON file for the level, locate the previously added engine layers, copy them, paste them into the new file, and reposition them if the map size changed.

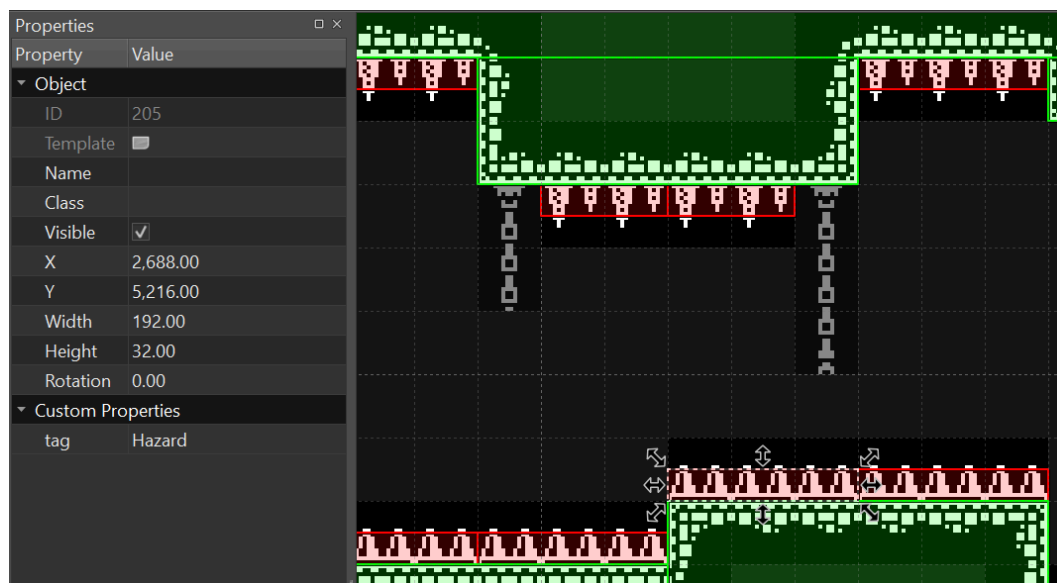
With updates to level construction, placing and tagging archetypes has become much easier, along with re-importing levels when changes are made. While the overall workflow for level design has improved, some processes still require manual input. For instance, placing player layer archetypes in *Tiled* is not feasible due to the player manager's requirements for a custom layer not created in *Tiled*. As a result, the player layer must be pasted back in whenever a level is updated.

These enhancements have enabled the level designer to create a [Tiled Design Guide](#) document, which details how to create levels in *Tiled* for import into the engine. This guide aims to maintain a consistent level structure when importing into the custom engine. Although the document originated during the *Unity* prototyping phase and contains some outdated information regarding custom *SuperTiled2Unity* properties, it provides valuable insights on level creation. One key guideline is color-coding layers to help recognize the various archetypes present in a tilemap.



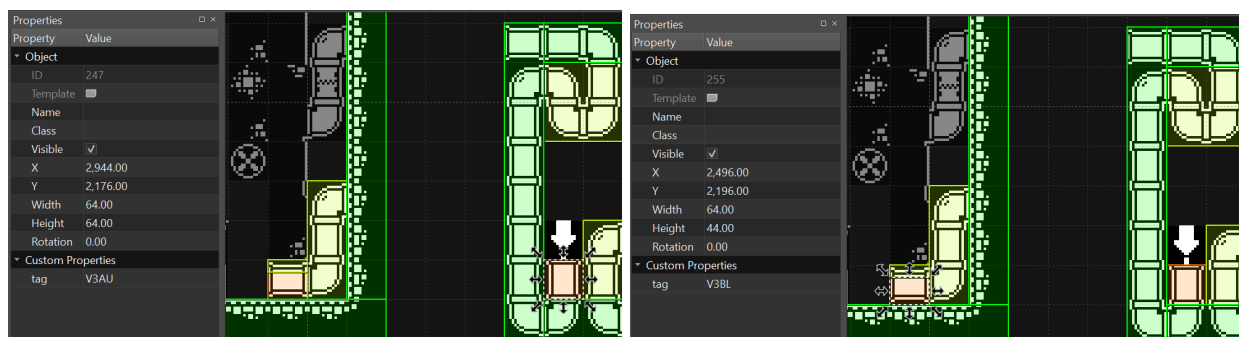
*Tilemap section with and without color-coding.*

Another important guideline is that every object in a tilemap must have a class that determines its archetype in the scene. This class string can be assigned to entire layers or individual objects, with individual object assignments overwriting the layer's class. This feature proves particularly useful for the manager layer. Additionally, using properties like “tag” allows designers to tag objects directly in *Tiled*, facilitating player collision detection for objects such as hazards and Signiflyers.



*Hazard tag*

“Archetypes” like tooltips, vents, lamps, and doors are highly customizable, enabling optimized scripting of their behavior through tagging. For example, a lamp with the ‘L1’ tag can be programmed to open a door with the ‘D1’ tag. Tags can also define the behavior of the archetype itself. In one case, tags like ‘V3BL’ and ‘V3AU’ indicate vent types, with ‘V’ standing for vent, ‘3’ representing the index of the vent in the current level, and ‘A’ and ‘B’ denoting the ends of the vent, while ‘L’ and ‘U’ indicate the exit direction for the player.



*Level 2 Vent 3 tags*

Overall, utilizing *Tiled* for level creation is highly effective for engines supporting 2D game development. The integration of features such as direct archetype creation from *Tiled* and object tagging enhances designers' workflows and productivity.

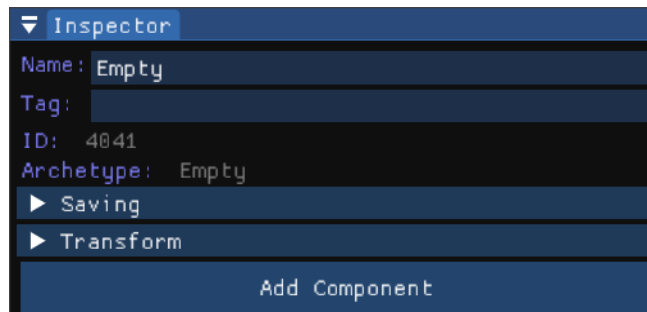


*Level 1 Tilemap with and without archetypes.*

# Components

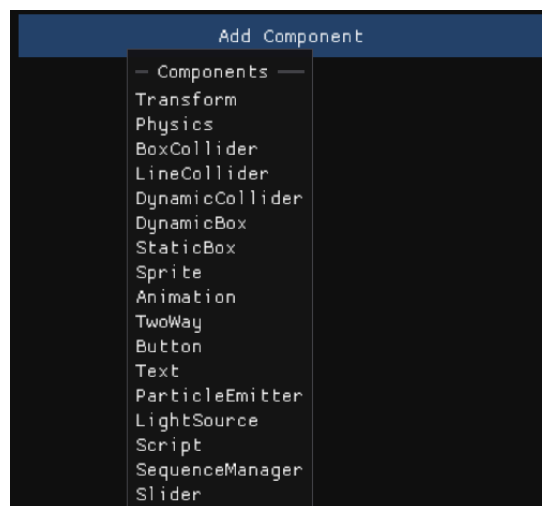
## Managing Components:

Similar to other game engines, all components are viewed within the *Inspector* when an entity is selected. The list of all applied components for entities is shown below, along with all the main information about an entity, such as its “Name”, “Tag”, “ID”, and “Archetype”.



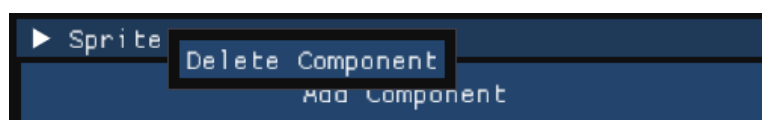
*Inspector window with an Empty entity selected.*

When clicking the **Add Component** button that is displayed at the bottom of an entity’s listed components, a list pops up of all the different components that can be added to the entity.



*All components in the engine.*

If a component has to be deleted, it can be done so by right-clicking on any of the components assigned to the selected entity, and then, selecting **Delete Component**.

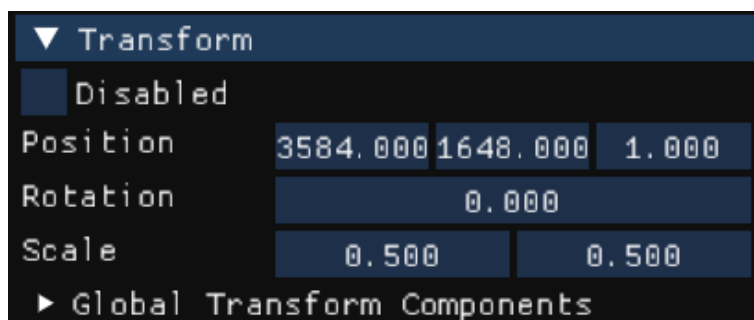


*Delete Component button displayed next to the mouse origin.*

## Transform:

Every entity in the engine has a *Transform* by default, which controls the position, rotation, and scale of the selected entity. Unlike most engines, the “Rotation” is represented as a float, due to the engine's design for *KnightLight* as a 2D side-view game. The “Disabled” checkbox in this component is non-functional, as the transform is essential for the existence of any entities.

“Global Transform Components” are designed for parenting entities and creating children. This feature was added later in the project and was not utilized by designers. However, the programmer responsible for the new Player Movement script implemented these components, as they were necessary for the Player Manager’s functionality.

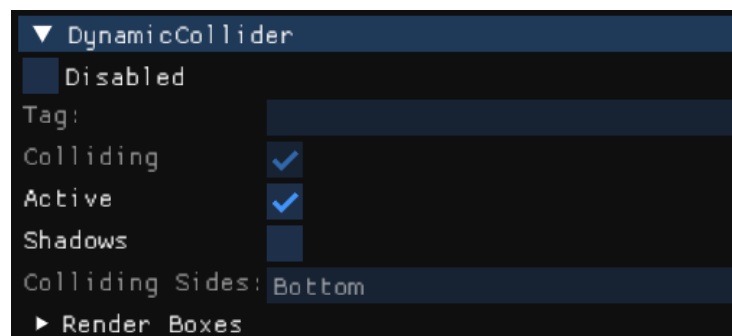


*Transform of the LightPlayer entity.*

## Colliders:

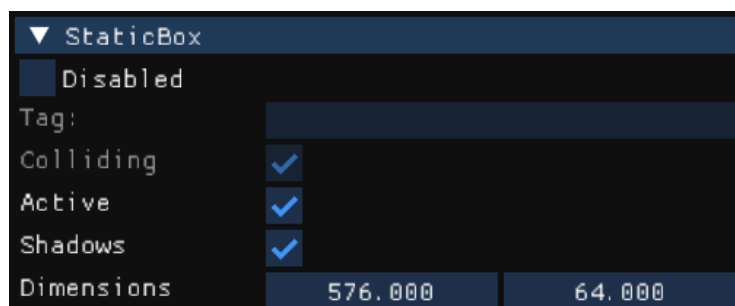
The engine features three types of colliders used in the game: *Dynamic*, *Static*, and *Line*. Each collider component includes an “Active” checkbox, which determines whether it can collide with other colliders, and a “Shadows” checkbox, which specifies whether the collider should cast shadows from light sources. The “Tag” and “Colliding” variables are displayed but not editable; they indicate whether the collider is currently colliding with anything and the tag of the other collider involved in the collision, if applicable.

The *DynamicCollider* includes a non-interactable variable called “Colliding Sides”, which shows which side(s) the entity is colliding with. This collider is used on the player to implement gravity, which is why shadows are disabled, to ensure the player remains visible and illuminated.



*DynamicCollider on the LightPlayer entity.*

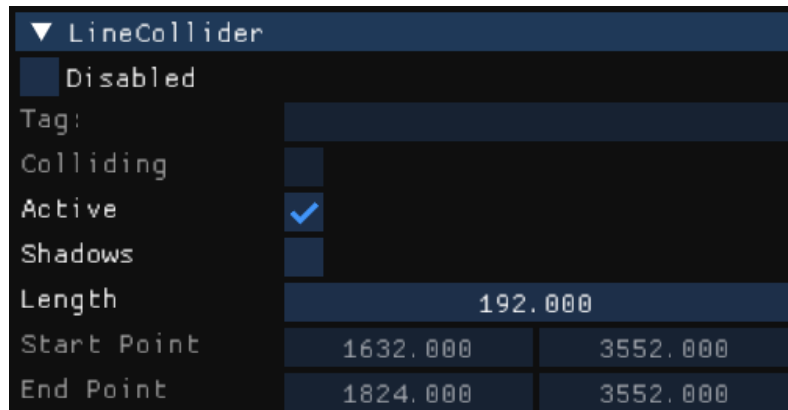
For most environmental elements, the *StaticBox* collider is utilized. These colliders have the “Active” and “Shadows” toggles enabled, ensuring solid collisions and shadow casting from light sources. The *StaticBox* collider also features adjustable dimensions in the inspector, which are set according to the specifications defined by the level designer in *Tiled* when creating new entities.



*StaticBox on some environment ground.*



The *LineCollider* component is specifically used for the two-way platforms. “Shadows” are typically disabled for these colliders, allowing players to see both above and below them with their light. The *LineCollider* includes a “Length” variable that can be edited, unlike the *StaticBox*, and features non-interactable “Start Point” and “End Point” variables, which are based on the positioning of the *Transform* component.

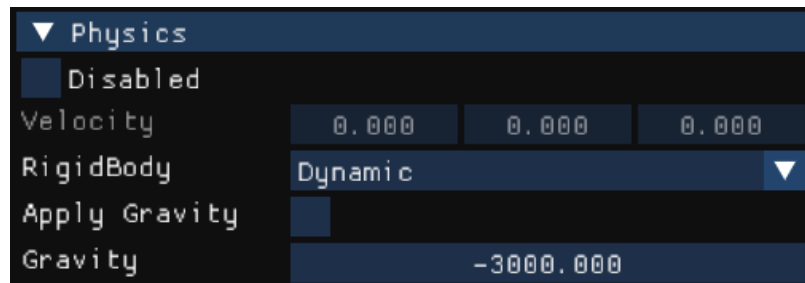


*LineCollider used on a two-way platform.*

## Physics:

The *Physics* component is currently utilized exclusively on the player entities (Knight and Light), using a *DynamicCollider*, as these are the only entities that utilize gravity. This component includes a “RigidBody” type that can be switched between “Dynamic” and “Static”.

An “Apply Gravity” toggle can be modified through the *Inspector*, although it is controlled and activated by the *Player Movement* script when the player is airborne. Additionally, there is a “Gravity” variable that dictates the speed at which the gravitational force is applied to the player.



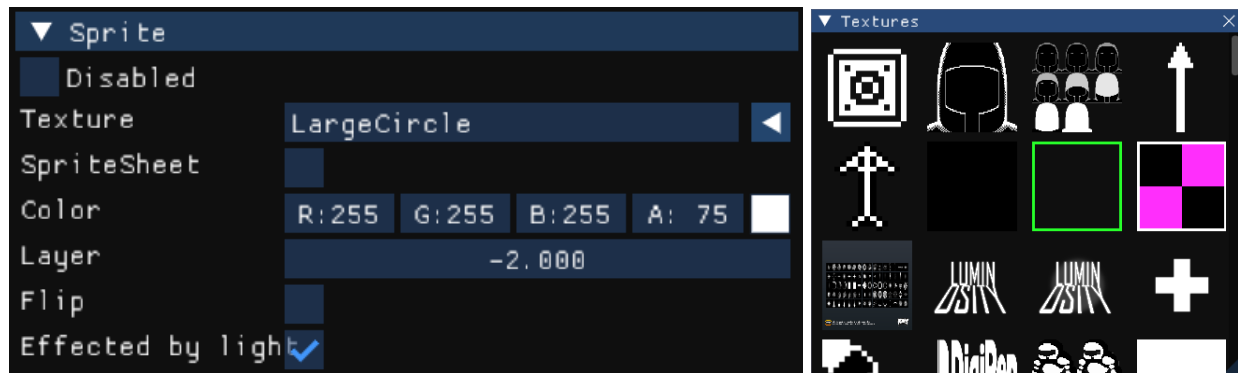
*Physics on the LightPlayer entity.*

## Sprites:

*Sprite* components are among the primary components used in the game, as they serve as the main method for visualizing entities within the engine. This makes it easy to test and debug various aspects of gameplay.

Each *Sprite* component features a “Texture” dropdown, allowing users to scroll through and select the desired texture to assign to an entity. The “SpriteSheet” toggle is activated when an *Animation* component is present alongside the *Sprite* component on the same entity.

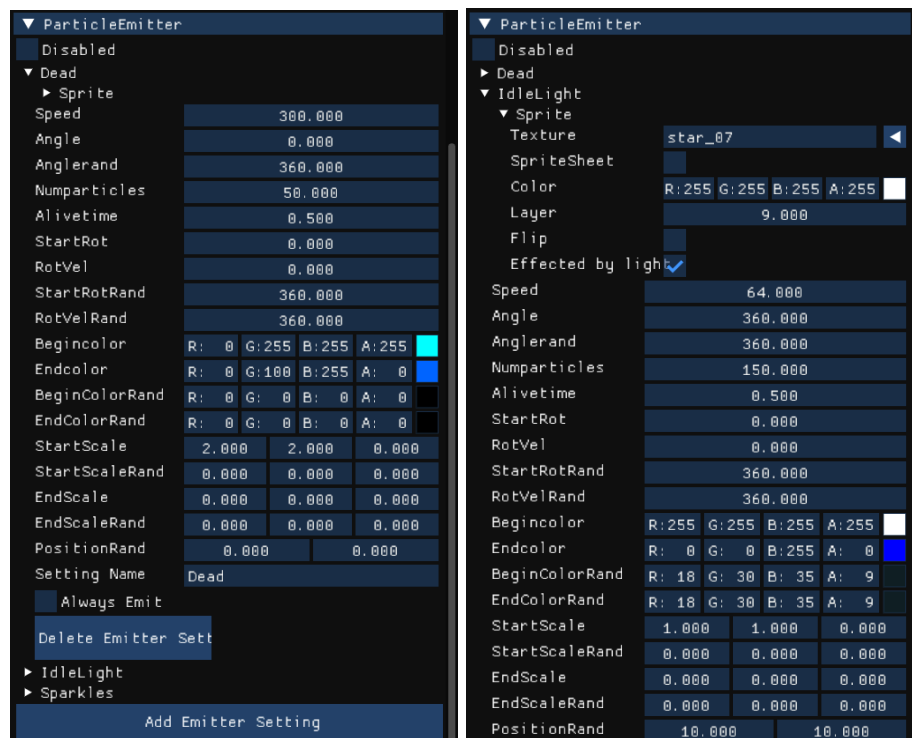
A “Color” field allows for inputting individual color values or selecting a color through the color picker for ease of use. The “Layer” variable sets the visual order of sprites when they are displayed next to one another. Additionally, there are “Flip” and “Effected by light” toggles: the “Flip” toggle horizontally flips the sprites, while the “Effected by light” toggle determines whether a sprite is visible in the dark without any light source. In gameplay, the Signiflyers are the only sprites that do not have this toggle activated.



*Sprite component on the LightPlayer entity, display of all Textures.*

## Particle Emitters:

The *ParticleEmitter* component in the engine offers a range of adjustable variables, allowing for fine-tuning to achieve the desired particle effects. Multiple emitter settings can be configured within the same *ParticleEmitter* component. For instance, the player's death effect is titled "Dead", and other player emitter settings located underneath the **Delete Emitter Setting** button, such as "IdleLight" and "Sparkles," are also configured for the light player in the game. The "IdleLight" particles are visible on the light player, while the "Sparkles" particles float in the background during gameplay, enhancing the visual experience.



*ParticleEmitter* attached to the *LightPlayer* entity. "Dead" setting on left, "IdleLight" on right.

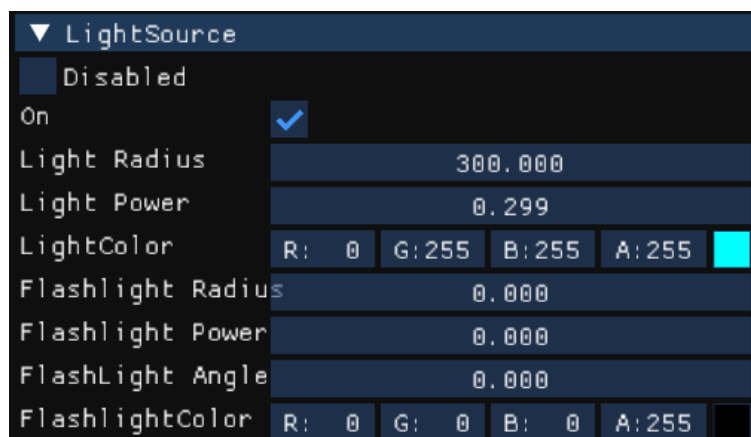
An emitter setting can be added or removed at any time via the *Inspector* at the bottom of the component, or even through scripting. Like other components, it includes a “Disabled” toggle. Most variables are intuitively named based on standard particle system terminology, though certain settings, particularly for angle and rotation, function uniquely in this engine.

- **Angle:** Determines the direction (in degrees) that particles will move when spawned, creating a linear emission.
- **Anglerand:** Acts like a pie chart, emitting particles in random directions within a specified angle. For example, setting it to 360 emits particles in a full circle around the origin point.
- **StartRot** and **StartRotRand:** Set the starting rotation of particles.
- **RotVel** and **RotVelRand:** Control the rotation speed of particles. “RotVel” sets a default rotational velocity, while “RotVelRand” introduces a randomized range. This works by taking “RotVel” and generating a random velocity within a calculated minimum ( $\text{RotVel} - \text{RotVelRand}$ ) and maximum ( $\text{RotVel} + \text{RotVelRand}$ ) range for each particle upon spawning.

## Light Sources:

The *LightSource* component in the engine equips entities with a built-in “flashlight”, essentially a view cone that emits light in a directed beam. While most entities with this component have their flashlight variables set to zero, the Knight character retains its “flashlight” active for gameplay.

The component includes an “On” toggle to enable or disable raycasting from the *Light Source*, critical for managing light behavior in active entities. For instance, spikes with proximity lights, Signiflyers, and the player’s flashbang effect all use this toggle to control when their *Light Sources* are active without deactivating the entity itself.



*LightSource on the LightPlayer entity.*

Radial light variables manage the primary light properties for most entities:

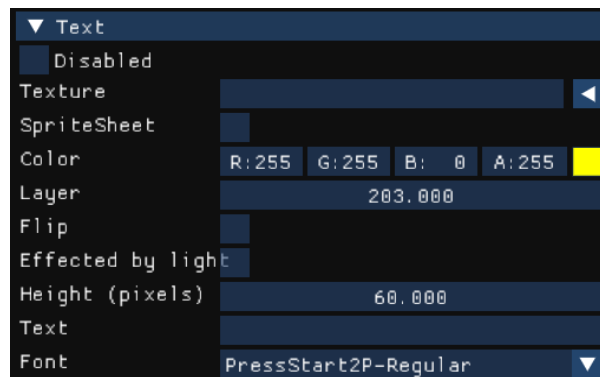
- **Light Radius:** Controls the light’s reach.
- **Light Power:** Sets the intensity of the light, with recommended values between 0 and 1 to avoid unintended visual effects like a “donut” shape.
- **Light Color:** Determines the color of the emitted light.

The “Flashlight” variables function similarly to these radial settings but add “FlashLight Angle”, which adjusts the width of the light cone, defining how narrow or broad the flashlight beam appears. This setup allows for dynamic and customizable lighting across various gameplay elements.

## Text:

The *Text* component includes various properties, some of which—like texture and the sprite sheet toggle—have no functionality. This is due to its foundation on the *Sprite* component, as both operate similarly. Shared properties like “Color”, “Layer”, “Flip”, and “Effect by light” function the same as in the *Sprite* component, allowing adjustments to color, render layer, orientation, and light influence.

Text-exclusive properties include “Height (pixels)” (or font size), the “Text” input field (for displaying specific text), and a “Font” property that supports multiple fonts, though only one is used in this project. The only limitation of this component is that it may not render text correctly if its height is not set to a multiple of 20, which can cause text distortion.

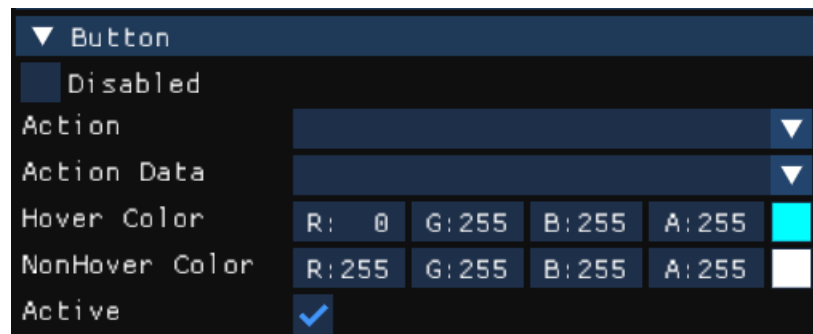


*Text example from the PAUSED text in the pause menu.*

## Buttons:

*Button* components require a *Sprite* component within the same entity to enable hover and non-hover color states, as the *Button* component overrides the entity's color with these states. Additionally, preset actions allow buttons to start a new game, load the previous scene, or load a specific scene, with action data specifying the target scene. An exclusive toggle, "Active", can prevent the button from triggering its assigned function.

Most buttons in the game share consistent settings to maintain uniformity. For greater flexibility beyond preset actions, custom C# functions can be assigned to buttons by declaring a listener delegate and adding it to a button. When triggered, the button then performs the custom function, which can vary widely in purpose throughout the game.

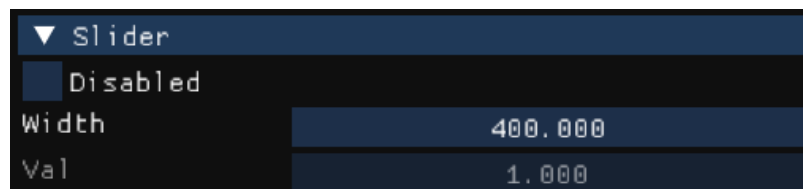


*Button values that are used throughout the game's UI.*

## Sliders:

The *Slider* component operates independently of a *Sprite* component, but using a *Sprite* provides visual feedback to display the slider's current value. It includes a "Width" property, which extends the slider's path to the right from its starting point, and an uneditable "Val" property, indicating the slider's current position within its maximum width.

To enable functionality, this component requires C# code to assign functions. In this project, sliders control volume settings for different sound groups, such as the master volume, background music (BGM), and sound effects (SFX) groups.

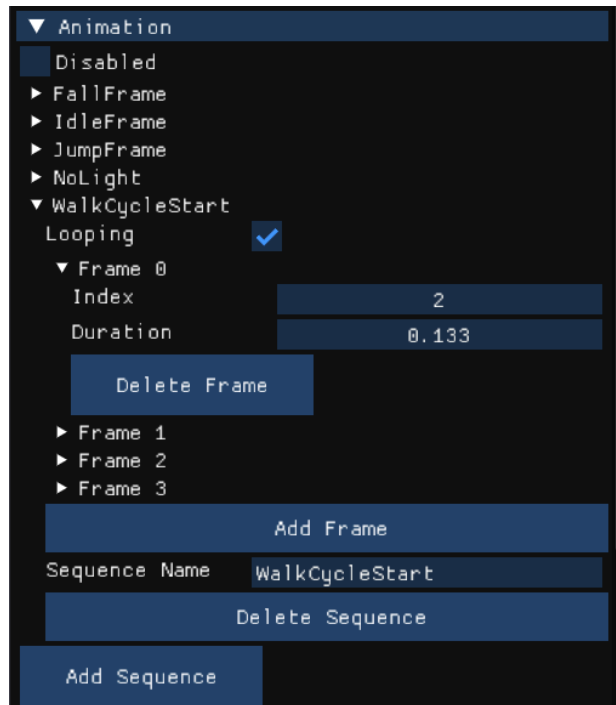


*Slider example taken from the volume options.*



## Animations:

The *Animation* component allows for the addition of sequences to an entity's animation list. Similar to *Particle Emitter* settings, this component allows multiple sequences within a single instance, and any sequence can be called through code. Each sequence includes a “Looping” toggle and an **Add Frame** button, which lets users add frames to the sequence and specify the duration for each frame. As with *Particle Emitters*, settings for each sequence can be saved under a unique name.

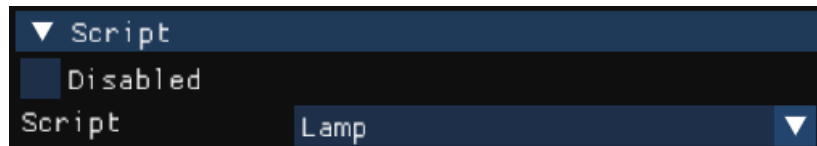


*Animation example from the KnightPlayer entity's walk cycle.*

A minor issue with this component is that it functions only with sprite sheets that have an equal number of frames horizontally and vertically, requiring sprite sheets to be formatted accordingly. Additionally, the component automatically plays the first animation in the list by default, even if it hasn't been explicitly called. To address this, the programmer who developed the component suggested adding a 1-frame animation as the first entry, using the initial sprite index intended for display.

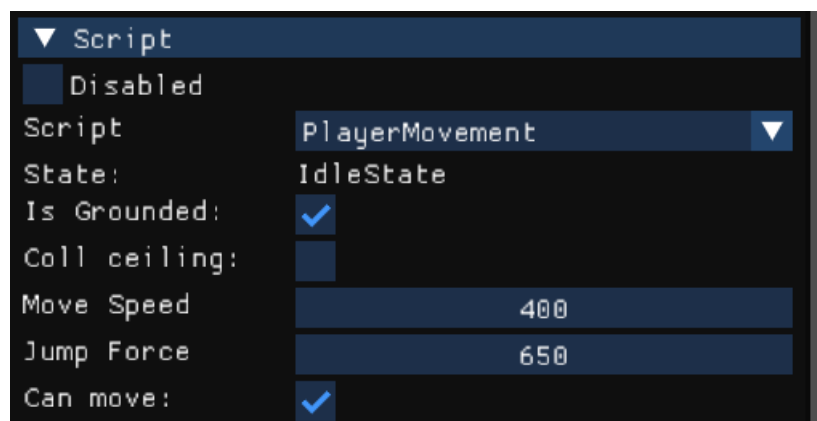
## Script (C#):

The *Script* component allows the user to attach a C# script to a specific entity, allowing the creation of various behaviors for different entities. This functionality has supported the development of singletons or manager scripts handling specific scene elements, such as the *Collectible Manager*, *Environment and Light Manager*, *Pause Manager*, and *Music Manager* in gameplay scenes. Non-gameplay scenes, like the Main Menu and Win Screen, typically use only one singleton.



*Script component for the Lamp archetype.*

Recently, a feature was introduced to display public variables in the *Inspector*. Although it doesn't permit real-time variable modification, this addition is valuable for debugging, helping prevent console clutter when tracking multiple variables simultaneously. Currently, the only scripts utilizing this feature are the new player movement-related scripts, including the *Player Manager*, *Player Light Source Manager*, *Light Player*, *Knight Player*, and *Recall*. Earlier C# scripts by designers do not use this feature due to its recent implementation. Had this feature been available earlier, it would likely see broader use in the game's current version.



*Script component with public variables displayed.*

The structure of C# scripts in the engine mirrors that of *Unity's* C#, making it familiar for designers.

The base functions in the engine's C#:

- **Startup**: This function acts like *Unity's* *Awake*, called when the script is loaded. The “Parent” variable must be set to this function’s parent parameter for the script to reference the attached entity.
- **LateInit**: This optional base function is not required for a script to function. It resembles *Unity's* *Start*, called immediately after the engine’s *Startup* function.
- **Update**: The *Update* function is invoked every frame and includes a dt (delta time) parameter for use within the function.
- **Exit**: This required function is called when a scene is unloaded, ensuring proper cleanup.

In addition to these base functions, custom functions can be created, which are essential for maintaining order, avoiding code duplication, and saving time in the long run. This implementation significantly optimizes the workflow for both designers and programmers.

```
1  using System;
2
3  0 references
4  internal class DesignGuideExample : KL_Script
5  {
6      1 reference
7      public override void Startup(IntPtr parent)
8      {
9          Parent = parent;
10     }
11     2 references
12     public override void LateInit()
13     {
14     }
15     2 references
16     public override void Update(float dt)
17     {
18     }
19     }
20     1 reference
21     public override void Exit()
22     {
23     }
24     }
25 }
```

*C# Script template with required functions.*

```

1 reference
public Vec3 LerpPosition(Vec3 startPos, Vec3 endPos, float t)
{
    float x = Lerp(startPos.x, endPos.x, t);
    float y = Lerp(startPos.y, endPos.y, t);
    float z = Lerp(startPos.z, endPos.z, t);
    return new Vec3(x, y, z);
}

3 references
public float Lerp(float a, float b, float t)
{
    return a + (b - a) * t;
}

1 reference
public float GetDistanceToPoint(IntPtr objectA, Vec3 point)
{
    TransformGetPosition(objectA, out Vec3 a);

    float x = a.x - point.x;
    float y = a.y - point.y;
    return (float)Math.Sqrt(x * x + y * y);
}

```

*Snippet of some functions used throughout different scripts.*