Jared Callupe Lopez

GAM250

Team Luminosity

Game KnightLight

Playtest Report 2

Custom Engine: C# Implementation

Subject

C# scripting in the custom engine

Executive Summary

As our game team's custom engine keeps getting developed, more ways for the designers to work on it get added. Recently, C# has been implemented in the engine in order to allow us designers to write code and make it work in the engine. The designers were guided by one of the programmers through the current state of the custom engine to learn to create C# scripts for the engine using a template. They were also given a script for documentation of all the functions and an example script to see how they're used in a scene in the engine. After learning the basics, designers worked by themselves on their own scripts learning how to translate and implement their prototype scripts into the engine, while making observations of their experience with C# scripting in the engine. Based on these results, feedback from the designers was shared to the programmers, specifically the one in charge of C# implementation, to communicate their needs for functions in the engine. Moving forward, designers will share the specific functions they need with the programmers in order to keep them informed.

Purpose

“How does C# work in the custom engine?”

The purpose of this playtest was to start working in the custom engine using C# by writing code and testing it in the custom engine. So far, we’ve only been able to modify values in components and in .json files, but these changes weren’t significant or too relevant in the engine.

Due to the limitations of the engine, not all of the features in our prototypes can be accurately translated to it. Although, learning how to write C# code for the custom engine will help us implement our prototypes authentically and closer to our original vision.

Additional Playtesting

In addition to C# scripting, and due to the limited number of functions available, a list of functions that are needed to recreate our prototypes in the engine were made to communicate the programmer in charge of C# functions, what we’re currently in need of. This is how we’ll mostly be working for the rest of the semester communicating our needs to programmers.

Build

The playtest build was the custom engine itself. The Menu scene was used to showcase the way C# works on a scene, using a custom ‘kngiht’ entity with an example script. The example script contained examples of the multiple C# functions implemented. A copy of this scene was made to work freely on it while testing C# scripts. A script with all the available functions was added for documentation using comments and regions for better readability.

Custom Engine Menu

Method of testing

The way this playtest session was structured was by being guided by one of our team's programmers, the one in charge of C# implementation specifically, through the Visual Studio solution and showing us how to create C# files and how to make them work in the engine. Then, we'd work on our own scripts trying out the different functions that were added to try and implement our prototype ideas.

The playtest was performed in person in our team space with one of our team's programmers, Jack Love, guiding us through the custom engine solution and helping us create scripts and locate the functions we can utilize.

Custom Engine Guide

- Name: Jack Love (DigiPen RTIS Student)
- Team Role: Programmer (C# Implementation)
- Date: 02/02/24

Session length: 30 minutes approximately

Observations

Before starting to work with C# on the custom engine, a zipped template file was sent to help us designers create C# scripts in the solution. This file was put in our Visual Studio templates folder. Once in the custom engine, I was told to look at the 'kngiht' entity and the new script component and I noticed the Example.cs script. In the Visual Studio solution, I was told to look at the solution explorer and create my own script by right-clicking on KL_Scripts and adding a new item using the template previously imported. Once in the script, I immediately noticed the resemblance with C# in Unity as the template had the StartUp and Update functions. I was told that I had to add the Exit function manually. I asked how these functions worked and got a good idea of it comparing them to Unity C# functions:

- Startup(IntPtr parent) → Start()
- Update(float dt) → Update()
- Exit() → OnDestroy()

In Startup, the parent IntPtr parameter is used to get a reference of itself, which Unity doesn't do. This means we have to set the script's Parent to the Startup 'parent' parameter in the Startup function of every script (Parent = parent;). In Update, the dt float parameter is simply delta time which can be useful for timers or lerps.

The first thing I did before testing scripting was declaring an int variable just to see if it worked the same way. In the Visual Studio solution, I was told to check the KL_Script.cs file to see all functions available with documentation and the Example.cs script to see how they're used. After this, I copied and pasted the menu scene to have my own scene to work on. I made an empty game object and attached my script with the script component.

Once I learned how to create scripts and implement them in a scene, I finished my session with the programmer and started working by myself testing the different functions for 3.5 hours, approximately. After looking at the Example script, I noticed how there was a list of IntPtr and all entities created were added to this list and deleted all elements of the list in the Exit function, so I did the same. I tried out almost all of the functions available excluding the physics ones, and made a Main Menu Manager script.

```
/*****************************************************************
 * @file:   JaredTest.cs
 * author: jared.callupelopez (jared.callupelopez@digipen.edu)
 * date:   February 2, 2024
 * Copyright © 2023 DigiPen (USA) Corporation.
 *
 * @brief:  C# test script
 *****************************************************************/

internal class JaredTest : KL_Script
{
    public bool keyPressed = false;
    public bool startSequence = false;

    List<IntPtr> list = new List<IntPtr>();

    public override void Startup(IntPtr parent)
    {
        Parent = parent;

        //Knight sprite
        IntPtr KnightSprite = CreateEntity("Empty", "MainScreen");
        //Add sprite component
        //Add components
        //Set knight sprite color
        //Set knight sprite animation
        list.Add(KnightSprite);
    }

    public override void Update(float dt)
    {
        if (InputGetKeyDown(Keys.SPACE))
        {
            keyPressed = true;
        }

        if (keyPressed)
        {
            if (!startSequence)
            {
                //Start Sequence

                //Play audio feedback
                SoundPlay("OpenPauseMenu", "Example", false, 1.0f);

                //Play button entity
                IntPtr PlayButton = CreateEntity("MenuText", "MainScreen");
                TextSetText(PlayButton, "PLAY", 80);
                Vec3 PlayPos = new Vec3(0, -300, 0);
                TransformSetPosition(PlayButton, ref PlayPos);
                float PlayRot = 0f;
                TransformSetRotation(PlayButton, PlayRot);
                //Set text color
                //Set text layer
                list.Add(PlayButton);

                //New game button entity
                IntPtr NewGameButton = CreateEntity("MenuText", "MainScreen");
                TextSetText(NewGameButton, "NEW GAME", 80);
                Vec3 NewGamePos = new Vec3(0, -400, 0);
                TransformSetPosition(NewGameButton, ref NewGamePos);
```

```
                float NewGameRot = 0f;
                TransformSetRotation(NewGameButton, NewGameRot);
                //Set text color
                //Set text layer
                list.Add(NewGameButton);

                //New particle system
                IntPtr KnightParticles = CreateEntity("Empty", "MainScreen");
                //ParticleEmitterSetActive(KnightParticles, "", true);
                //ParticleEmitterEmitBurst(KnightParticles, "");
                //Set layer
                list.Add(KnightParticles);

                startSequence = true;
            }
        }
    }

    public override void Exit()
    {
        foreach (var child in list)
        {
            DeleteEntity(child);
            //list.Remove(child);
        }
        list.Clear();
    }
}
```

The script works similarly to my prototype in the way that it uses bools to play the start sequence, which I declare at the start along with the list of objects that will be deleted on exit. However, that's where the similarities end as most of the functions in my script were done using animations in Unity. One difference from Unity is that I cannot declare null entity variables and drag them from the hierarchy to the inspector, or even set their values in the script itself. Instead, I have to create the entity and a reference to it at the same time. This made it hard to work with an entity variable in multiple parts of a script until I found out you could simply not set a value instead of trying to set it to null.

After going through all of them, I took notes on all regions in the documentation:

- Entity: Creating an entity requires an archetype or type of entity which is similar to a prefab. You can also choose the layer and if it doesn't exist, it creates a new one on the scene.
- Transform: Basic functions like get/set position/rotation. One detail I noticed is how position requires a vec3 while rotation, a float. This is because entities only rotate in one axis in the engine.
- Sound: The only function played a sound and needs 4 parameters: the name of the sound which had to be in the Sound folder in the repository, the group name or name of the channel where the audio is playing similar to Unity audio mixer groups, a loop bool to determine whether the sound loops or not, and a volume float to set the volume scale.
- Input: Get key and get key down/up functions, another function that returned the mouse coordinates which I didn't know to apply but I can see how it is useful.
- Physics: Basic set/get velocity/acceleration functions getting the IntPtr parameter and a reference to a vec3.
- Particle Emitter: It had two functions: set active and emit burst.
- Loose functions: These are functions that were not in a region:
    - TextSetText, which required the IntPtr, the text input and a font size int.
    - ColliderGetTag, which returned the collider tag of a collider, not the object tag.
    - DynamicColliderGetCollisionInfo, which I'm not sure how it worked but I believe it's like OnCollisionEnter and gets various info from the collider like the side the collided or resolution which I'm still unsure of its meaning.

Conclusions

After learning how to work with C# on the custom engine and collecting these observations, we can recognize some of the main strengths and weaknesses of it.

- The engine lacks a way to add and/or remove components in an entity.
- The engine lacks a way to change the game's time scale which we're going to need later for the pause manager.
- There are no functions for all components like sprite (set color, sprite, layer, etc.), button (interactable, hover and nonhover color, etc.), particle system (various particle system variable adjustments), among others but these are the most important for my main menu prototype.
- There are no input functions for any key pressed or mouse buttons.
- Currently, there's no way to get a reference to singletons or tagged entities in the custom engine.

Recommendations

Based on the conclusions, the following solutions or features can be implemented to the custom engine.

- A function to add and remove components from entities should be implemented.
- A function to get and set the time scale should be implemented.
- More functions for specific components should be implemented, prioritizing the components the designers are currently using the most or plan to use to recreate their prototypes.
- Input options for mouse buttons should be implemented, as well as functions for any key pressed whether that be keyboard or mouse input.
- A function that gets a tag and looks for an entity in the scene with said tag should be added.