Jared Callupe Lopez

GAM200

Team Luminosity

Game KnightLight

Playtest Report 4

Custom Engine Introduction

Subject

How KnightLight's custom engine works.

Executive Summary

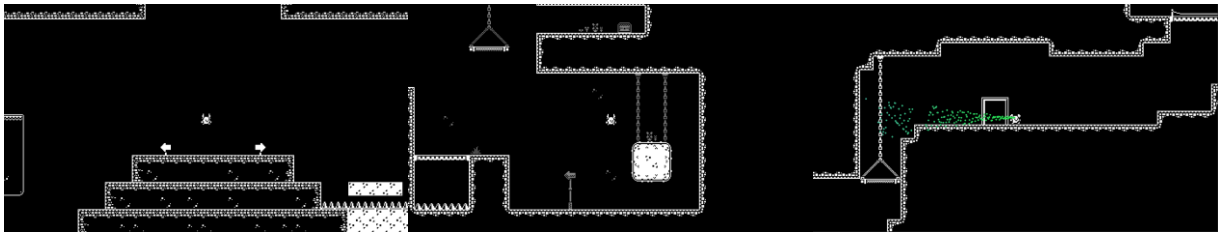
As our team game's custom engine keeps getting developed, the team's designers need to be introduced to it so they can learn how it works and how to work on it. The designers will be guided by one of the programmers through the current custom engine build files and see what different files do, to then go into the engine itself and look around it, making observations and findings. These observations will be analyzed to form conclusions and recommendations as feedback for the programmers to see. Based on these results, appropriate changes can be made to the custom engine to improve accessibility to the designers.

Build

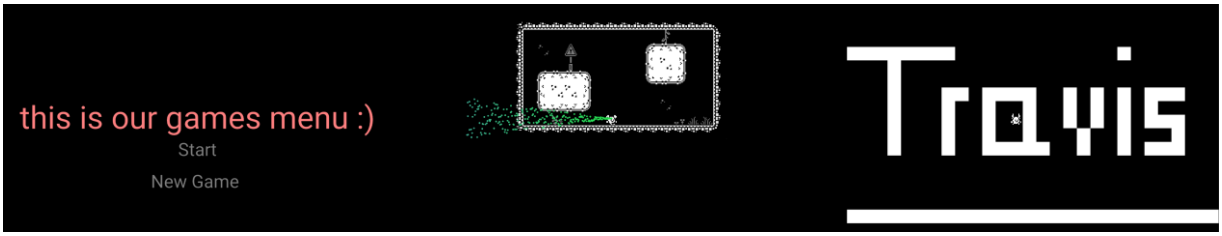
The playtest build was the custom engine itself, which included six different scenes, all of which were made in Tiled, except for the game's menu:

- JaredMap: The unfinished big map
- layertest: Layer test scene which features the decorations layer
- Level4_1: Our current Unity build's level scene
- Menu: Game menu
- testmap: Small room for testing
- travismap: Scene that says 'Travis'

These will be analyzed more in depth in the Observations section.



JaredMap, layertest, and Level4_1



Menu, testmap, travismap

Purpose

“How can we work on the custom engine?”

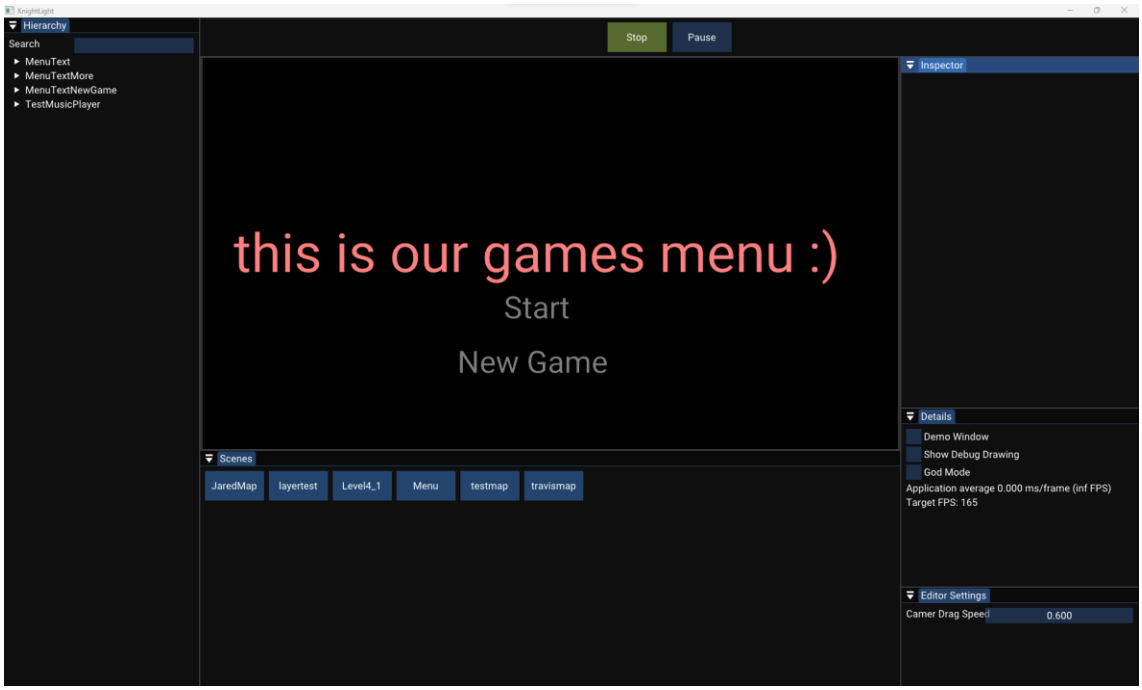
The purpose of this playtest was to learn how the custom engine works so we, the designers, can soon start working on it. So far, we’ve only been working on our prototypes in Unity, but we haven’t been making significant changes to the team’s repo, other than uploading our weekly Unity builds in a different folder away from the custom engine.

Learning how the custom engine works will also help us set a scope on what’s possible to implement or not when coming up with ideas and prototyping them. Due to the Unity engine being way more advanced than ours, it is important to recognize that not every feature we prototype in Unity can be accurately translated into the custom engine, or even implemented at all.

Additionally, learning to work on a primitive custom engine can help us familiarize ourselves with other more complex custom engines we’ll be using in the future in other game projects that require one.

Method of testing

The way this playtest session was structured was by being guided by one of our team's programmers through the team's repo and showing us how everything worked in it including .json files and how we can edit them to affect variables in the engine. Then we'd be shown how the custom engine itself works, including hierarchy, inspector, scenes, details, and editor settings. And finally, we'd be able to freely play around it and analyze the different features it includes.



Start of the engine

Extra playtesting

In addition, any unusual gameplay elements or bugs will be observed and written down to analyze them and let the respective programmer know about them for their prompt fix.

Observations

The playtest was performed in person in our team space with one of our team's programmers, Charles Winters, guiding us through the team's repo and custom engine.

Custom Engine Guide

- Name: Charles Winters (DigiPen RTIS Student)
- Team Role: Programmer (Physics)
- Date: 11/02/23
- Session length: 1 hour approximately

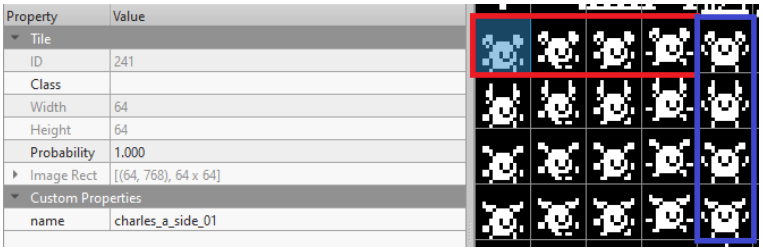
The first thing we were shown was where to find the .json files in /trunk/KnightLight/Engine/Assets/Data and there we could find three different folders: the first called 'Entities' which contained entity behavior, or what we in Unity know as 'game object'. We were told these are the ones we'll be mostly be working on as they contain variables of which values we can modify to change how they're shown in game. For example, Charles.json, the player controller (because it uses the tile with the 'Charles' ID), contains the following variables: gravity, movement speed, jump speed, speed boost, all of which we can modify to match our Unity prototype's movement.

```
{
  "Type": "Player",
  "Data": {
    "Gravity": -400.0,
    "Movement Speed": 500.0,
    "Jump Speed": 500,
    "Speed Boost": 2.5
  }
},
```

Charles.json movement variables snippet

Additionally, it contains two animations for both walking and jumping. One thing I noticed was how it used the tileset index to load the sprites for the animation. In the Tiled screenshot I highlighted in red the walking animation frames, while the jumping frames are highlighted in blue. The selected tile is the one with the 241 ID which is the same one that starts the walking animation.

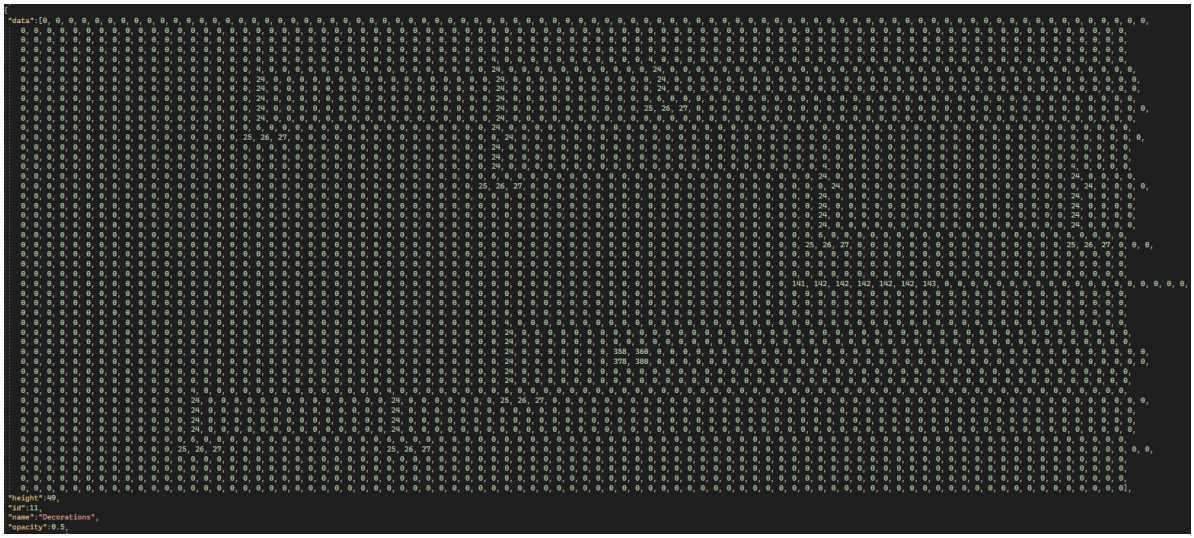

```
{
  "Type": "Animation",
  "Data": {
    "NumFrames": 8,

    "Frame1": {
      "Duration": 0.066,
      "Index": 241,
      "NextFrame": 2,
      "Name": "Anim1Start"
    },

    "Frame2": {
      "Duration": 0.066,
      "Index": 242,
      "NextFrame": 3
    },

    "Frame3": {
      "Duration": 0.066,
      "Index": 243,
      "NextFrame": 4
    },

    "Frame4": {
      "Duration": 0.066,
      "Index": 244,
      "NextFrame": 1,
      "Name":
"SequenceEnd"
    },
    "Frame5": {
      "Duration": 0.15,
      "Index": 245,
      "NextFrame": 6,
      "Name": "Anim2Start"
    },

    "Frame6": {
      "Duration": 0.15,
      "Index": 265,
      "NextFrame": 7
    },

    "Frame7": {
      "Duration": 0.15,
      "Index": 285,
      "NextFrame": 8
    },

    "Frame8": {
      "Duration": 0.15,
      "Index": 305,
      "NextFrame": -1,
      "Name":
"SequenceEnd"
    }
  },
},
```

Walking and jumping animation frames

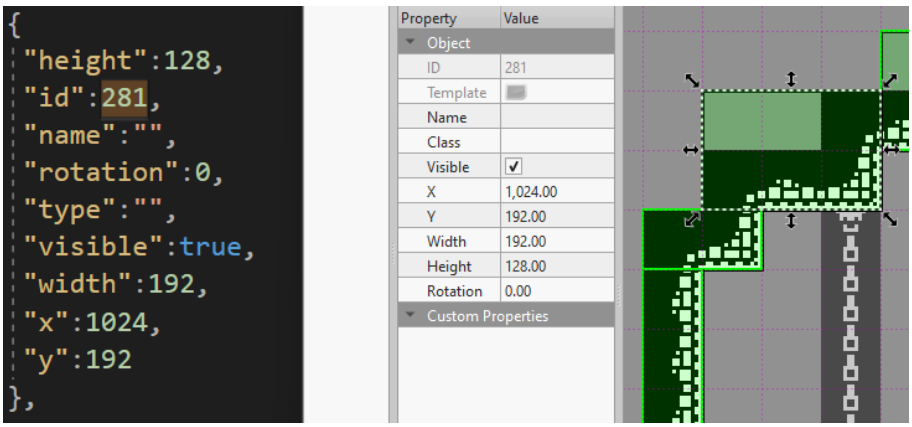
Finally in this script we could find the particles emitted from the player which contained multiple variables that can be modified in the engine itself. This folder also contained other .json files for other entities' behavior.

The second folder was called ‘Levels’ which included all the Tiled tilemaps exported as .json files, as well as another folder called ‘TiledData’ which include the Tile tileset exported as a .tsx file. I was already familiar with the level .json files and how they include data regarding layers and their components since I’ve been working in Tiled and exporting the tilemaps for the programmers to work with. For example, the ‘Decorations’ layer contains a copy of the whole tileset with a value different to 0 where it places a tile in the layer and sets its opacity to 0.5.



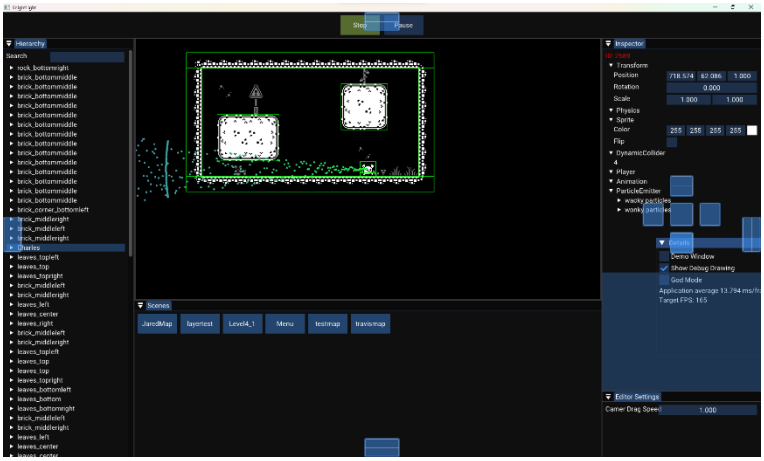
Tile data snippet in Level4_1.json

In addition, I found the Tiled object layers including the ‘Collisions’ object layer where I could find every collider I placed on the tilemap.



Collider object with the 281 ID in both Level4_1.json and Tiled

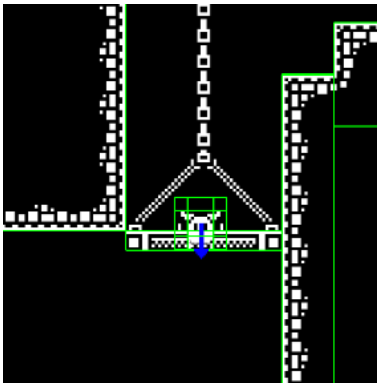
After this, we went to the custom engine itself. To load it we had to go to /trunk/KnightLight/ where we could find the Visual Studio solution. In this file, you can press F5 to load the debugger and open the custom engine. Since it was our first time opening it, the debugger had tons of logs and the windows were all minimized and in the same spot on the screen. These windows could be dragged to any part of the screen and arranged however you want on the interface. In my case, I made an interface similar to Unity's default layout. With the hierarchy on the left, the inspector on the right, the assets ,or in this case the scenes, at the bottom of the screen, the main game window in the middle, and miscellaneous windows like details and editor settings at the bottom of the inspector.



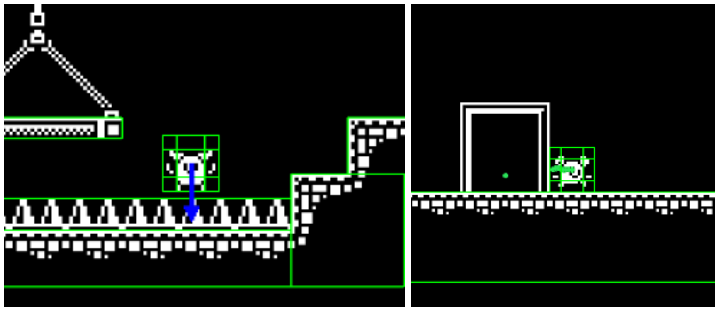
Custom engine layout

The first thing I notice was the six different scenes:

- JaredMap: This was the tilemap I was originally working on, which we ended up discarding due to its scale. It was simply used as a test when importing maps to the custom engine.
- layertest: This tilemap made in Tiled had an extra tile layer with a name component called decorations, which our programmer who specializes in serialization and deserialization told me to give, as he can give them a lower opacity in the custom engine.
- Level4_1: This is the current Unity build's level scene. It contains all colliders on the Colliders layer but is missing the JumpThrough layer as it's still being worked on, as well as the Exit layer which doesn't load another scene, yet. However, the hazards do work fine, and they restart the scene when the player collides with them.



Player cannot collider with jump through layer



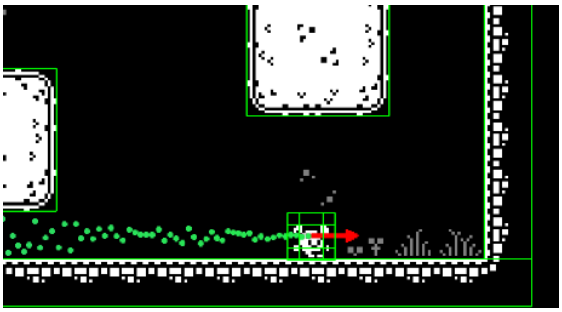
Hazards layer does work and restarts the scene

- Menu: The menu scene only contains four entities: three for text, and one for music. One thing I noticed was how the two buttons ‘MenuTextMore’ and ‘MenuTextNewGame’ had a button component, which was not editable in the engine, but the one that said ‘Start’ loaded the layertest scene.
- testmap: This was a smaller scale room with two blocks with 1 block gaps between them and the floor or ceiling. I believe this is mostly used for physics testing purposes as that’s what I used it for mostly personally.
- travismap: This is just a map that spells ‘Travis’ I think it servers the same purpose as JaredMap which is just to perform importing tests in the engine. Though, I believe at one point it was used to create an artificial floor by not letting the player fall past certain Y coordinate, which explains the white horizontal line at the bottom.

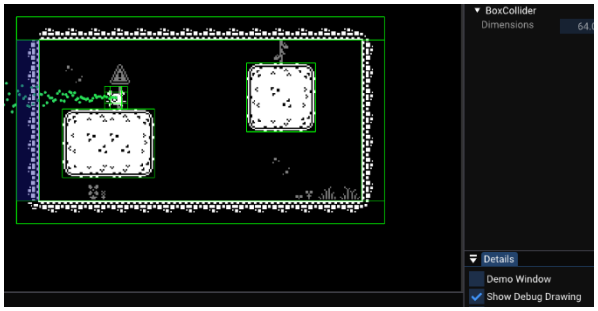
While going around these scenes, these were some of the things I noticed:

- First of all, the debug console shows when sound effects are played, but I also noticed it showed when the engine gained or lost focus when I changed applications and went back to the engine.
- One of the first things I tested was the particles emitted from the player. The ‘wonky’ particles were the only ones that showed up and could be edited in the engine. While the ‘wacky’ ones didn’t show up at all as they have to be edited through the script itself. While trying to enable them, I pressed the ‘F’ which I realized toggled between fullscreen and windowed views.
- One thing our guide pointed out was how you have to move a tile entity to the side if you want to access the collider behind it. Though, you can also use the hierarchy and look through all the other colliders. I think there could be a way to organize the hierarchy better as every entity is visible in it.
- Another thing our guide pointed out was how, in the editor settings, the camera drag was buggy and will drag you away from the scene focus if its value was set to 1 or greater. The current solution to this problem is starting and stopping the engine which will move the game view back to the player, and obviously not set the camera drag speed to 1 or higher. Our guide also realized the component’s name was misspelled as ‘Camer Drag Speed’.
- Another thing I noticed when messing with the particle system coming from the player was how the particles didn’t rotate or scale with the player even though they’re its component, unlike Unity. This is because their transform is set to world space and doesn’t follow its “parent’s” transform. I don’t think this should be a problem for the player, but for the enemies that we’re planning on adding it might be, if we decide to add particles to them getting “burned” by the light, as their scale slowly decreases as they do.
- Another thing I found different from Unity’s engine was how colliders push you out of them. In Unity, if you clip through a collider, it can let you stay inside it. However, in the custom engine, if you enter a collider in god mode, then disabled god mode, it will push you to the closest edge.
- One thing I really liked about the custom engine was how the movement direction was signified by a gizmo of which arrows appeared when force was applied in their respective

direction. Additionally, the debug drawing function was really helpful to visualize how colliders interacted in the scenes.

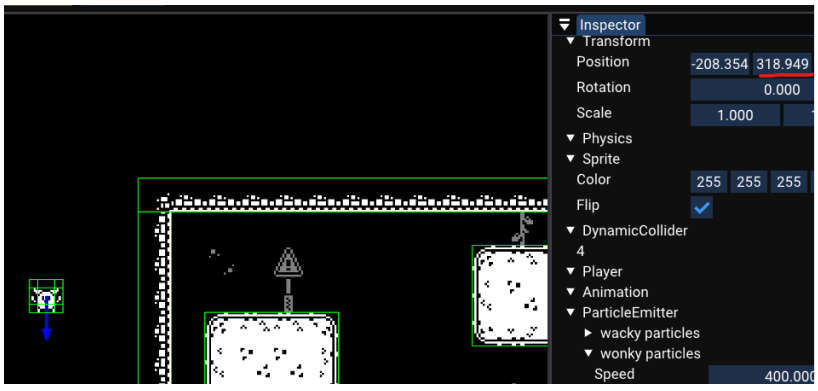


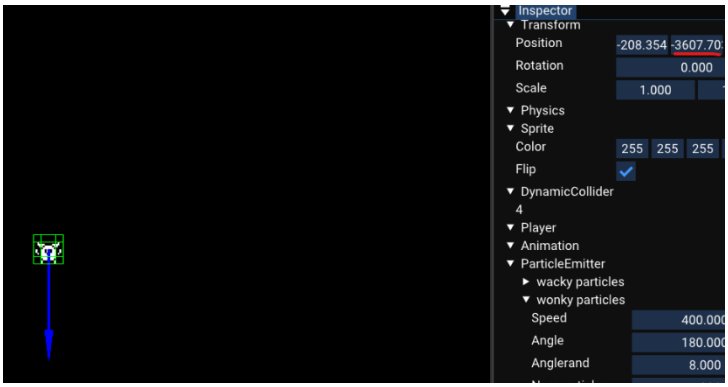
Gizmo signifier



Debug drawing

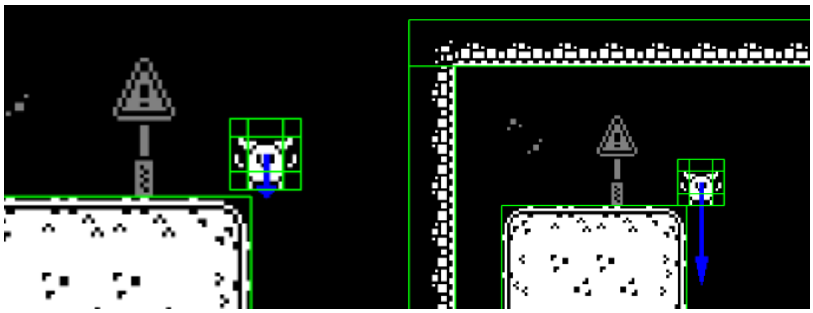
- This allowed me to see how gravity stacked as you fell, with the arrow increasing in length as the gravity increased. This also affected the jump force which got reduced as the player went up on the jump and increased as they went back down.





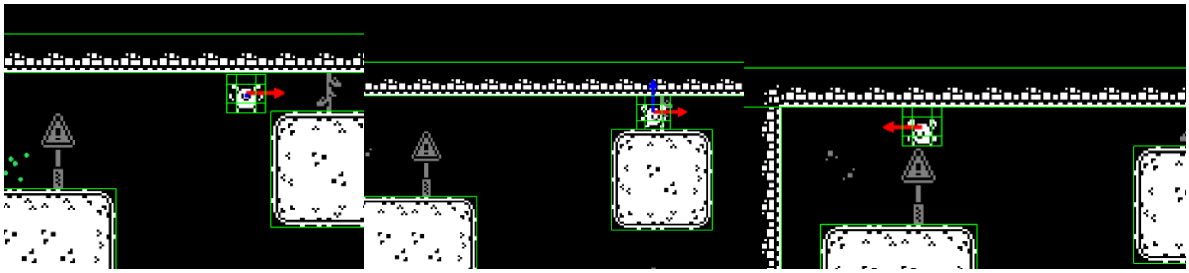
Gravity force increases as the player falls

- This made me realize how corners would push you out if the player's ground check was over the edge of its collider. This assumed the player was falling and it kept stacking the gravity force which ended up in the player abruptly falling down the corner. Since our guide was in charge of this behavior, he knew this was a current bug and is in the process of fixing it.



Gravity force increasing as the player "falls" on the collider's corner

- Another bug I found was how you can get stuck the ceiling if you jump in a tight space. In my case, I jumped on 1 tile tall space which made my player stick to the ceiling while I could still move left and right until I jumped again releasing the player. Though I'm not planning on adding any tight spaces in our level design, this is a bug that needs to be fixed.



Player gets stuck to the ceiling when jumping on a tight ceiling

- Another physics bug our guide was aware of was how you'd get launched if you paused the engine right after landing on the floor. I couldn't personally replicate this one as I couldn't get the timing right, but our design lead was able to replicate it and also found out it could be replicated if you hit a ceiling, too.

Conclusions

After learning how our game's custom engine works and collecting these observations, we can recognize some of the main strengths and weaknesses of it.

- A particle system's transform is set to world space by default.
- The hierarchy looks too messy with all tile entities displayed on it.
- Camera drag speed gets buggy if its value is set to 1 or higher.
- Colliders push you out of them.
- Gizmo signifier and debug draw are very useful tools.
- There's a bug where corners push you down.
- There's a bug where the player gets stuck to ceilings.
- There's a bug where the player gets launched when pausing right after landing or hitting a ceiling.

Recommendations

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

- A new setting in the particle component that sets the particles system's rotation and scale the same as its parent entity. Similar to how in Unity, a parent game object's transform will also affect all its children.
- It'd be a good idea to parent all tiles to an empty parent entity you can open and close to see all children. Or simply hiding them in the hierarchy could also work.
- Fix the camera drag speed bug or set its maximum value to be a number smaller than 1. Also fix the misspelling.
- Keep it so colliders push objects inside them as it can prevent more bugs that can happen in Unity's engine.
- Both the gizmo signifier to show the forces and debug draw to see all colliders have been very useful so far. Something I think should be added is being able to see the velocity values on the inspector, similar to how you're able to see when the player sprite is flipped or not.
- Make it so it doesn't recognize the player as falling when it's on the corner of a collider. Changing the ground check size might be an alternative, though I'm unsure how movement works.
- Make it so the player collider doesn't clip through other colliders the same way a regular collider pushes other colliders outside of it. Though this might not work as those colliders are static and the player's dynamic.
- Make it so pausing also freezes the player's and its collider's position.