

Rhythm Platforming Plugin

by

Chenkai Zhou

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Interactive Media and Game Development

April 2021

APPROVED:

Dean O'Donnell, Thesis Advisor

Charles Roberts, Committee

Abstract

We design and implement an accessible way for developers working with the Unreal Engine to build rhythm platformers. We built a plugin to predict the player location at the time of the audio. Unreal developers used the plugin to make their own level and then completed a survey to record their satisfaction. We published this plugin to the Unreal Marketplace.

Acknowledgments

Allow me to express my deepest gratitude to my thesis advisor Dean O'Donnell, for his professional guidance, advice, and moral support throughout this process.

In addition, I want to thank Dr. Charles Roberts for pointing me in direction on this project at the very beginning.

Last but not least, the person from the Unreal Marketplace Team who has been following my submission and guiding me through all the necessary steps for the plugin to be released.

Table of Contents

Abstract	1
Acknowledgments	2
Table of Contents	3
Executive Summary	5
Introduction	6
Overview	6
Terminologies	6
Events	6
Resolving	6
2D Platformer games	7
Endless Runner	8
Rhythm games	8
Rhythm Platformers	10
Building a rhythm platformer	10
The Tool Design	12
Design process	12
Unreal Module	12
Visual Design	14
Button Box	14
Bottom Tool Box	15
Waveform Canvas	15
Function Design	15
IO	17
Utility	18
Draw Process	18
Features	20
Workflow	20

The Game	21
Player	21
Blocks	22
The missing traditional platformer element	22
Jump?	22
Range action	23
Evaluation	24
Audio synchronization problem	25
Placement error	26
Perceived error	26
Other error	26
Postmortem	27
What went right	27
What went wrong	27
What We Would Do Differently	28
Conclusion	28
Bibliography	29
Appendices A - All evaluation data	30
Appendices B - Documentation	37
Introduction	37
Project and plugin files setup	37
Create 2d side scroll template and enable the plugin	37
Project Manager Setup	42
Setup Variables	44
Adaptive Placing:	48
Placing Event	50
Player Running Speed	51
Recommended Workflow/Reconstruct Example Level	54
ExampleLevel Gameplay	56

Executive Summary

We built an Unreal Engine plugin for developers to easily build rhythm platformers. The plugin is an overlay to the engine viewport. It aims to visualize the player location given an audio soundtrack and the player's moving speed. The plugin ships with a demonstrative scene with simple gameplay mechanics. However, we hope this plugin can ultimately serve as an interface for any designer to implement their own features. After the plugin was complete, we sent it out to a few WPI students that had experience developing in the Unreal engine, and asked them to build a level using the plugin. After collecting these levels, we asked them to finish the feedback survey. Upon processing these feedback surveys, we find that the plugin meets the goal with a few drawbacks. Such as the plugin is not presenting the information precise enough to the designer that these events they placed are synchronized to the music. This asynchronization is also caused by the obstacles that come with the plugin.

Introduction

Overview

This project is to create a rhythm platformer plugin for the Unreal Engine. The plugin has three major characteristics. First, it predicts the player location given an audio time and visualizes that information as an overlay to the engine viewport. Second, the tool should rearrange the obstacles given different players' running speed. Third, the tool expects the designer to inherit its based class and build on top of it.

This paper first explores the background and examples of this type of games. It then explains the building block and components of the tool. This section is followed by the evaluation. The evaluation exposed some of the drawbacks of the current plugin and we tried to address issues. Finally, the paper concludes with appendices.

Terminologies

Events

We consider everything happening within a level of an event. For example, in a **Mario** level, a player may encounter a goomba, or may jump over a gap. Here, the goomba and the gap are events.

Obstacles are events that prevent the player from reaching the end and they have to be dealt with. Obstacles can split into two categories: moveable and static. Enemies such as Goomba and Koopas move around and Mario must attack (by jumping on them), avoid (by jumping over them), or use the environment (lure them into a pit) to overcome them. Static obstacles can be as simple as a pit, or may be rotating fire walls that require the player to time their movement.

Resolving

There are many ways to resolve an event; two of them are evading and destroying. Evade is an action where the player leaves the state of the level untouched yet keeps proceeding, from jumping over a goomba to a gap, to using a tunnel or jumping onto the top of the level to “skip” a large section of the level. Other than evading, an obstacle can also be destroyed. Destroy means removing the obstacle from the level such as jumping on top of a goomba to “kill” it.

2D Platformer games

One type of game is 2D platformer games. In this genre, players move from the left to the right of the screen, but they have some freedom to explore the environment. They need to navigate to the goal, if there is one, and deal with all the events along the way.

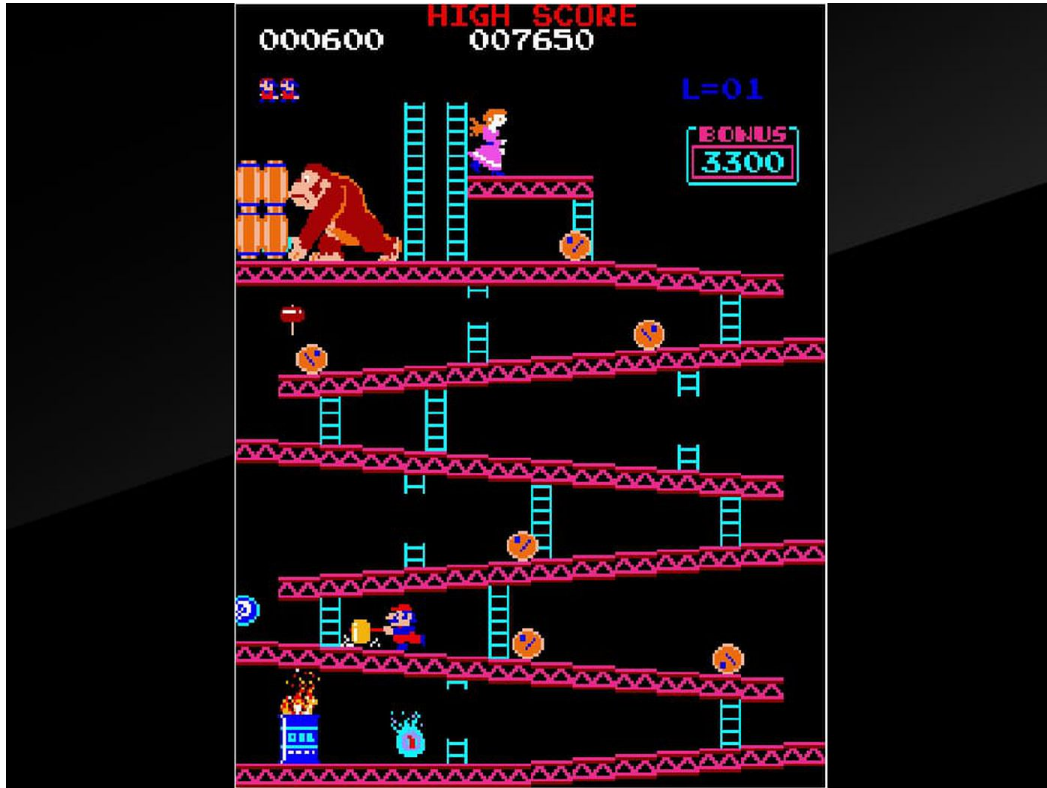


Fig. 1 Donkey Kong

1

The platformer is one of the famous and most distinguished genres in the video game media. **Donkey Kong**, published in 1981², is considered to be the first platformer that contains the defining characteristic for this genre³. In Donkey Kong, the player controls the avatar to jump over obstacles thrown by Donkey Kong to get to the end. This avatar later returns as Mario in **Mario Bros.** and that game becomes the typical example for the platformer genre. In a typical Super Mario Brothers level the player overcomes a series of obstacles in the level to reach the end. Obstacles are elements that must be attacked (monsters), or environmental blocks that must be overcome (walls and pits).

¹ <https://www.theverge.com/2018/6/14/17466778/donkey-kong-arcade-version-nintendo-switch-e3-2018>
accessed on 14 April 2021

² *Donkey Kong*. Nintendo, 1981

³ "Electronic Platform Game." *Encyclopædia Britannica*, Encyclopædia Britannica, Inc.,
www.britannica.com/topic/electronic-platform-game Accessed on 16 April 2021



Fig 2. Super Mario Bros.

4

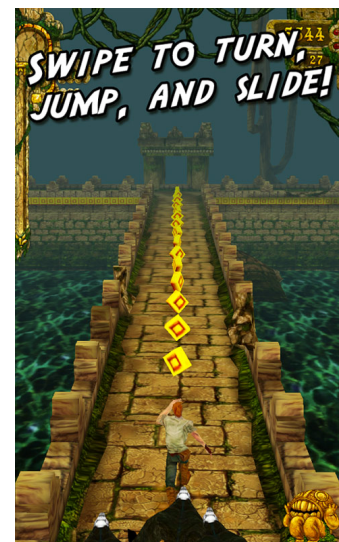
Endless Runner

One of the main differences between an endless runner, such as **Temple Run**⁵, and a 2d platformer is that in the endless runner, the game is always pushing the player forward instead of letting the player walk at their own pace. And unlike 2D platformers, endless runner does not have an end goal, but rather increases the player's running speed as the game proceeds and lets players strive to run longer than last time. In many ways, the endless runner requires the player to find a rhythm to their movements.

Rhythm games

Martin and Fares categorize rhythm games into 2 types: rhythm games and electronic instrument games. In rhythm games, events are usually arranged closely to audio cues.⁶ Taking **Taiko no Tatsujin**⁷ as an example, the player is given a series of events, to match the correct button, ahead and the system will rate each event when they are resolved. The closer these events are resolved to a set time, the higher the rating gets -- this is a rhythm game, where a level is based on a predefined soundtrack. As for electronic instrument games, such as **Crypt of the NecroDancer**⁸ and

Fig 3. Temple Run



⁴ <https://ultimateclassicrock.com/super-mario-bros-history/> access 16 April 2021

⁵ *Temple Run*. Imangi Studios, 2011

⁶ <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.190.2004&rep=rep1&type=pdf> access 4 May 2021

⁷ *Taiko no Tatsujin*. BANDAI NAMCO Studios, 2001

⁸ *Crypt of the NecroDancer*. Brace Yourself Games, 2015

Bullets Per Minute, are where players have to make actions according to the music. For example, In **Crypt of the NecroDancer**, the player controls a character that tries to get through a procedurally generated dungeon and every move and attack has to be performed to the rhythm of the soundtrack.



Fig 4. Taiko no Tatsujin

9

⁹ <https://damisanthrope.wordpress.com/2018/11/01/review-taiko-no-tatsujin-drum-n-fun/> accessed 16 April 2021

Rhythm Platformers



Fig 5. Rayman Legends Music Level

10

Rhythm platformer is a mix of rhythm games and 2D platformer. One good example is **Rayman Legends'** music level¹¹. In Rayman's music level, Rayman is constantly running forward and the game's events, mostly obstacles and some visual effects, are closely tied to audio cues. For example, rayman has to jump over the big green monster on a downbeat.

Unlike the traditional rhythm game where the resolution of an event is clear, events in rhythm platformers often offer more choices to resolve -- it is up to the player to decide whether to evade or destroy the enemy.

In short, the rhythm platformer asks the player to resolve 2d platformer events in rhythm of the audio cues.

Building a rhythm platformer

With the help of modern game engines, it is fairly easy to create a 2d platformer. In addition to designing the level by hand, Procedural Content Generation (PCG) is a new method to design 2D platformers. PCG

¹⁰ <https://switchplayer.net/2017/09/27/rayman-legends-definitive-edition-review/rayman-legends-music-level/>
accessed 18 April 2021

¹¹ *Rayman Legends*, Ubisoft, 2013

lets the designer frame a list of variables (such as jump, move) and the program will generate the level based on the fed parameters.

Designing rhythm games on the other hand, is less accessible to the public. **OSU!**¹² is a rhythm game that comes with an edit mode where the user can use their own music to make a level manually by themselves, the end product is another **OSU!** level. There are also other rhythm games, such as **Audiosurf 2**¹³, that lets the player feed their own soundtrack to the program and it generates a level based on that soundtrack.

Making a regular 2d platformer is easy enough via modern game engines such as Unreal -- there are a total of 455 platformers on itch that are made with Unreal. Since one of the major features in rhythm platformers is to line up platformer events with the audio cues, therefore, if the engine can predict the players location given the time of the music, the designer can easily design a rhythm platformer level without worrying about the audio syncing. Therefore, the goal of the project is to extend the engine functionality to support such a feature -- via engine plugin.

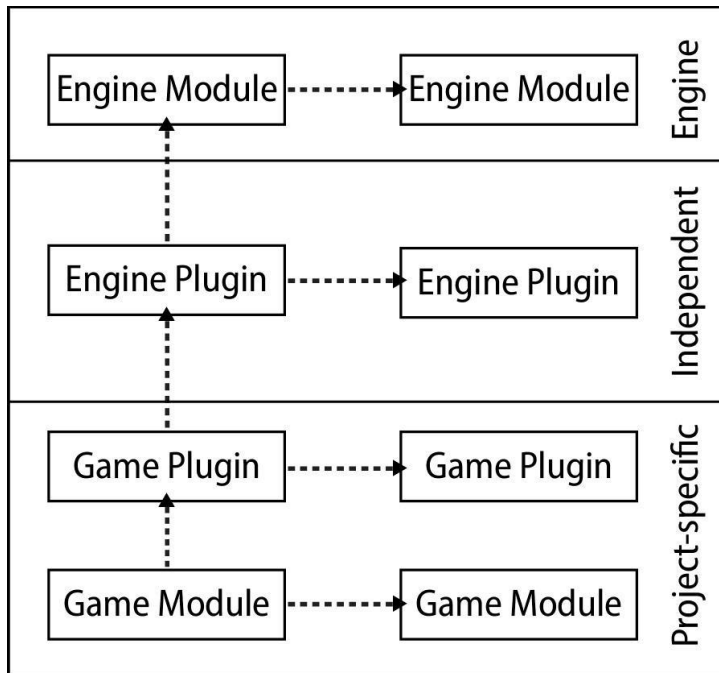
¹² *OSU!*, Dean Herbert, 2007

¹³ *AudioSurf 2*. Dylan Fitterer, 2015

The Tool Design

Design process

Since this is an Unreal plugin, there is crucial background knowledge to know beforehand, such as how Unreal is envisioned and how Epic expected the developer to extend the engine.



Unreal Module

According to Epic¹⁴, Unreal is built on an extensive collection of modules, and each module encapsulates a specific set of functionalities. They propose a hierarchy of modules and expect developers to follow it: Engine being the highest level, then independent, and below that project-specific. One important restriction here is that each plugin or module can only be dependent on same-level or higher-level ones. In other words, a higher-level module must be self-contained and can be built without a lower-level module.

Fig 6. Unreal dependency levels.

15

On the highest level is the Unreal Engine itself. The middle-level ones are plugins such as Chaos Editor, a plugin that makes destructible meshes. Or some other house made plugin that can be applied to other games. For example, most of the games need to load different scenes, and to present a loading bar between the scene loading is not uncommon. A developer can then extract this part of logic into a plugin. In the future, the developer can use this plugin to create loading scenes for different games.

And on the bottom levels are game-specific plugins that are usually not abstracted enough to make a stand-alone plugin but specific enough that the developer can use it again in their other projects. For example, a tool developer in a team may create a plugin that takes in variables and creates the enemy class and pass that plugin to the rest of the team to increase productivity. But this plugin will not be as useful in other games.

The Rhythm Platforming tool resides at the middle level, In order to be applicable to a more general set of games and projects.

¹⁴ <https://docs.unrealengine.com/en-US/ProductionPipelines/Plugins/index.html> accessed 10 Oct 2019

¹⁵ ibid

Another rule is the module property.¹⁶ Even though developers can quickly write their modules and extend the game engine without modifying any source code, each module has a property list indicating the module's design. One of the module properties is module type, and Unreal takes this property seriously. For example, if a module type is "editor," which means "loads only when the editor is starting up," this module cannot be included in the finished build because the plugin depends on some part of the editor and Unreal Engine will not include the whole editor in the shipping products.

RPP creates an overlay in the editor viewport. The tool should also come with some ready-to-use blueprint classes to put into scenes. This causes problems. For the former, the plugin must include the UnrealEd module because the tool needs a reference to the editor viewport, hence setting the module type to "editor." But setting such a property will result in the previously mentioned case.

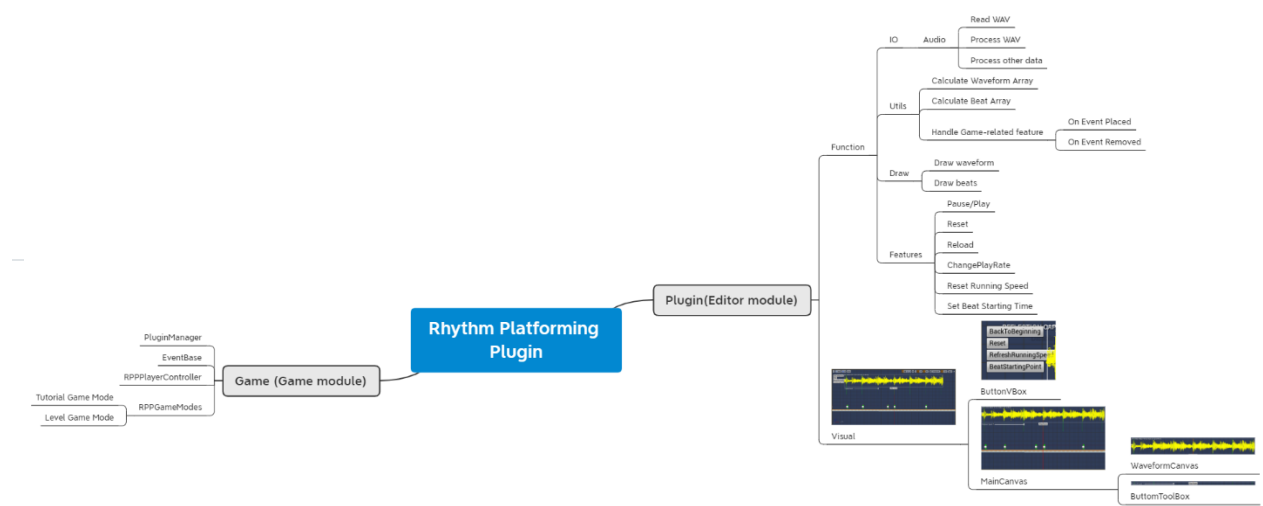


Fig 7. Plugin Architecture

Therefore, the tool is designed as not one but two modules: editor module and game module to resolve these restrictions. The editor part overlays the editor's viewport, providing a visual aid for the end-user to design the rhythm platformer more easily. The game module serves as an abstraction of rhythm platforming logics, containing base game objects, such as PluginManager and EventBase, designed to pass data to the editor module and provide basic game features to the end-user. The designers are also expected to inherit these game objects to suit their needs, and game features will be explained in the later section.

¹⁶ https://docs.unrealengine.com/en-US/API/Runtime/Projects/EHostType_Type/index.html. accessed 11 oct 2019

Visual Design

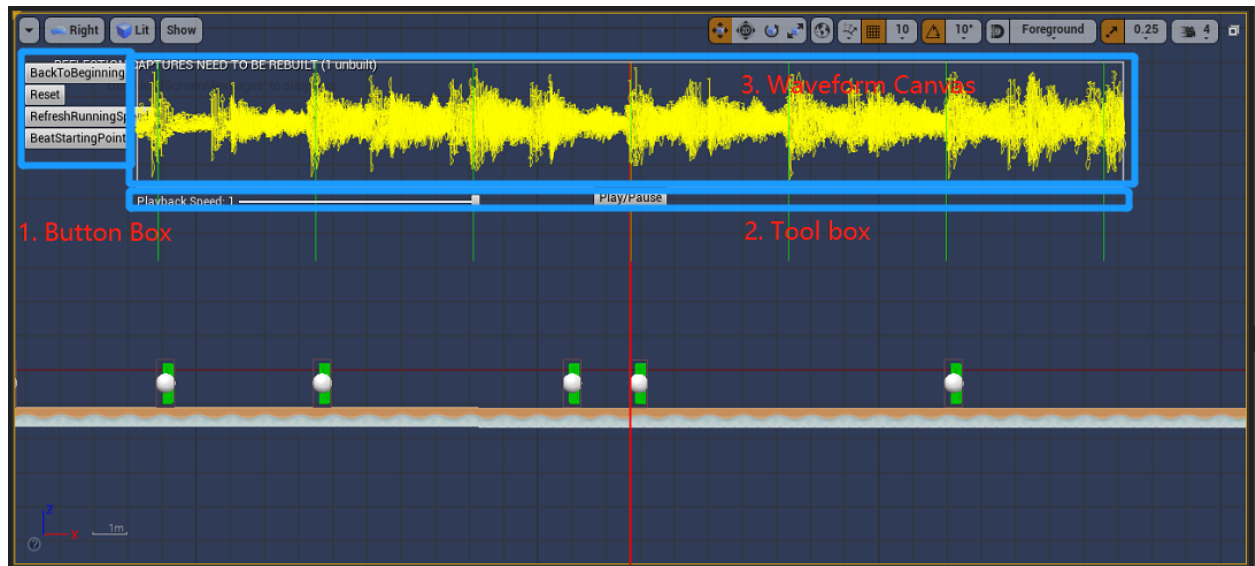


Fig 8. Main Interface

The visual block consists of three parts. Part 1 is the Button Box. Part 2 is the Bottom Tool Box. And part 3 is Waveform Canvas.

Button Box

The Button Box contains four buttons, and their function will be further explained in the following table.

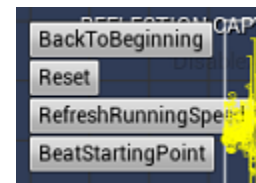


Fig 8. Button Box

Button	Short description
<i>BackToBeginning</i>	sets the level viewport back to the beginning of the soundtrack and the starting of the game
<i>Reset</i>	reloads whatever soundtrack detected in the PluginManager
<i>RefreshRunningSpeed</i>	gets all the EventBase class in a scene and recalculates their positions according to the running speed set in the PluginManager
<i>BeatStartingPoint</i>	lets the end-user set beat the starting offset.

Table 1

Bottom Tool Box



Fig 9. Tool Bar

The bottom toolbox has three components. A text block indicates the current playback speed, a slider to adjust the playback rate, and a play/pause button.

Waveform Canvas

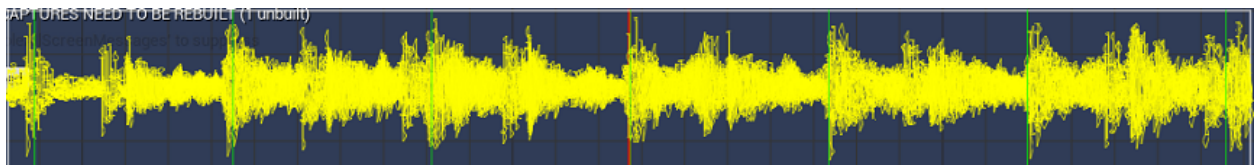


Fig 10. Main Canvas

This block paints the waveform of the soundtrack. The red snap line indicates the corresponding game world location to the current soundtrack time. And the green lines indicate the beats.

Function Design

Before we dive into each section, few crucial variables needed to be introduced first:

Variable	Short Description
PluginManagerObject	Serves as an interface for user-configurable settings, such as the Soundtrack, PlayerRunningSpeed, etc....
AudioCursor	Current audio location of the WAV file.
DataRawArray	Contains the raw data read from the WAV file.
DataDrawArray	Processed raw data from DataRawArray, including widget width and height.
DrawArray	Part copy of the DataDrawArray, determined by the AudioCursor.
BeatRawArray	Contains the beat information in seconds.
BeatDrawArray	Part copy of the BeatRawArray, determined by the AudioCursor.

Table 2

Let's use a typical 44,100hz, 10 second stereo WAV file as an example to illustrate the ideas:

- PluginManagerObject
 - Other being the setting interface of the program, this object also serves as the starting point of the game.
 - This must be placed the same vertical coordinates as the player starting point of the game.
- AudioCursor

- AudioCursor represents the current location of the soundtrack. This variable is changed on by only 2 means:
 - If the soundtrack is NOT currently playing in the editor, the program checks the current editor viewport location and compares it to the PluginManagerObject location, calculates the horizontal distance between 2 objects and divides it with PlayerRunningSpeed. This process runs every frame.
 - If the soundtrack IS playing, the value is set by the return calls of “PlaybackPercentageNative”, an Unreal editor API that reports how much of the current soundtrack has been played in percentage. Multiplying this percentage with the audio duration in total, the current audio location is set.
- For every frame, the program calculates the correct view location in the editor game world based on this variable.
- DataRawArray
 - This array of floats is the container of the raw data read from the WAV file.
 - For the example WAV file, there are 44,100 sample points every second for each channel. The program reads in these samples and calculates the average given the number of channels. 10 second wav file at 44,100hz, there are total of $44,100\text{hz} * 10\text{s} = 441,000$ elements in this array.
- DataDrawArray
 - This array of floats contains processed data from the DataRawArray.
 - The “process” procedure includes two aspects:
 - Scale: during the traversal of the DataDrawArray, the program gets the absolute max magnitude from DataDrawArray and divides all other elements with this number. So that the absolute max will be +/- 1.0 in the array
 - Compress: the process also selects an absolute max in a bucket. The bucket size is 5 at default (picking an absolute max in every 5 elements in the DataRawArray) and changed automatically when the end user zoomed in in the editor viewport.
 - After the process, if taking the previous example, will contain $441,000/5 = 88,200$ elements ranging from -1.0 to 1.0.
- DrawArray
 - This array contains data that needs to be shown on the editor viewport.
 - Given the AudioCursor, the program calculates the percentage this audio location is at and uses this percentage to get the index in the DataDrawArray.
 - For the example WAV file, 1 second of AudioCursor will index to $10\% * 88,200 = 8,820$. The DrawArray will then take data from this DataDrawArray at this index and copy the value into itself.
 - This array is referenced every frame in the paint process because AudioCursor potentially varies every frame, resulting in a different DrawArray every frame.
- BeatRawArray
 - This array stores the raw beat data. Data inside indicates a beat.
 - For example, if the BPM is 60, this array will be 0.0, 1.0, 2.0 ... etc..

- BeatDrawArray
 - Similar to the relationship between DataDrawArray and DrawArray, this array also serves as a selection of the BeatRawArray given different audio locations.
 - Like DrawArray, this array is also referenced every frame in the draw process.

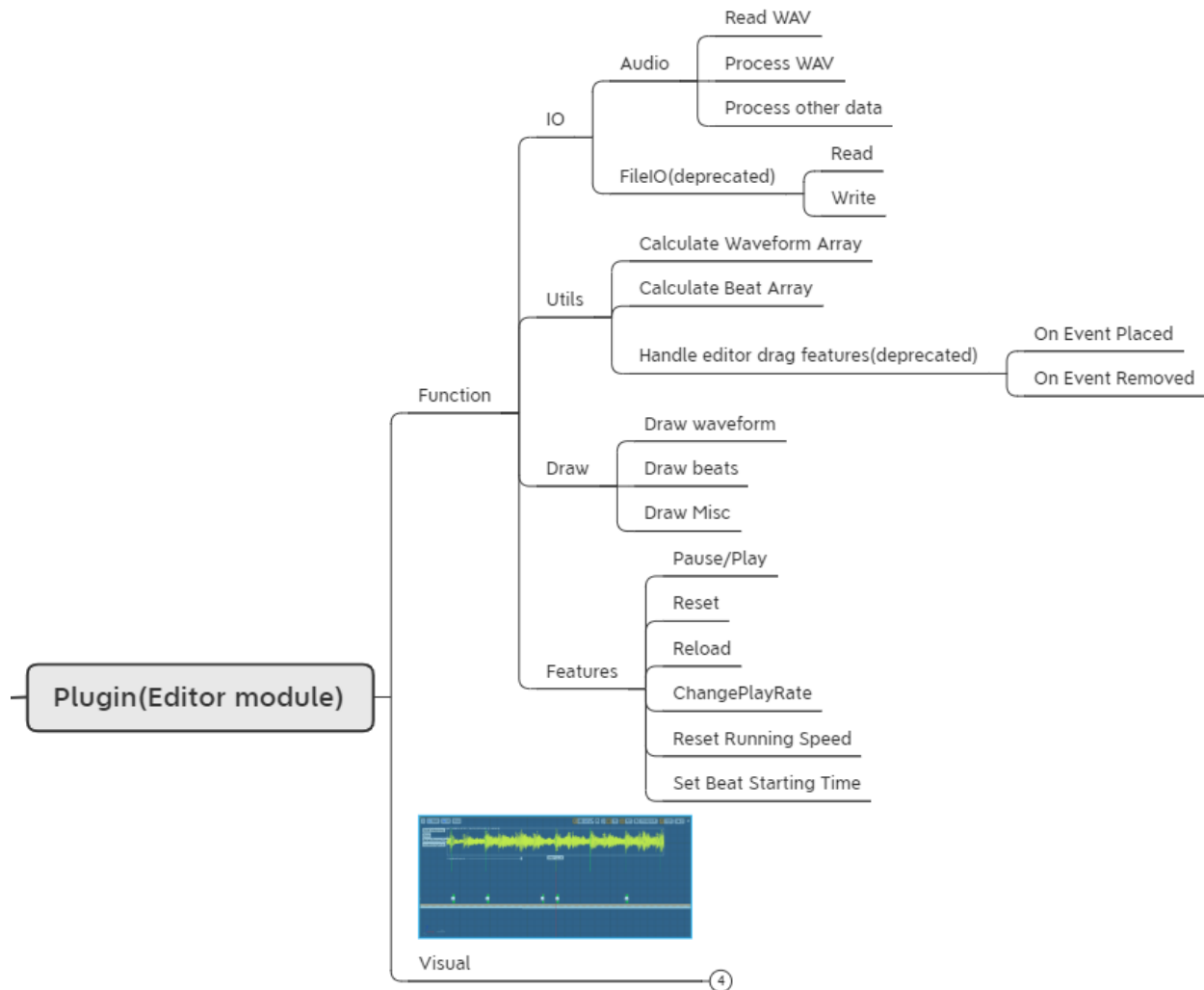


Fig 11. Class Design

The function block consists of 4 parts: IO, Utilities, Draw, and Features.

IO

As the name suggested, this part is mainly designed to handle reading the audio track. The first step is to read in the WAV file. In this step, the tool takes all the sample points and stores it into a float array.

Process WAV further "compresses" the RawDataArray into RawDrawArray. It has a variable *BucketSize* to serve as the zoom factor. For instance, if the BucketSize is 5, the program takes an absolute peak value in every five samples and puts that into the RawDrawArray.

Aside from processing the WAV file, IO also processes other data such as widget height and width (crucial to calculate window length).

Utility

The utility is a collection of functions that can't be categorized elsewhere. The most important functions here are GetDrawArray and GetBeatDrawArray.

GetDrawArray determines which part of the RawDrawArray shows up on screen given an audio location. In other words, after processing the RawDataArray, all the points in the RawDrawArray are ready to use, and the last step is to decide which part of this RawDrawArray gets to print on screen and this function does so. Receiving the audio location in seconds, the function then calculates the percentage that this audio location is at and uses this percentage as the starting index to copy data from RawDrawArray to DrawArray.

Similar to GetDrawArray, GetBeatDrawArray determines which part of the RawBeatArray shows up on screen given an audio location as well.

Draw Process

In the Slate framework (Unreal Editor's rendering system for UI), OnPaint is called every frame and programmers can override this function to define custom behavior, as long as it still executes the parent code. And therefore this is the suitable place to write our custom drawing code, if we want to draw on top of the editor viewport.

Besides OnPaint, Unreal also has a library, FSlateDrawElement, that contains building blocks API such as MakeLine. MakeLine takes an array of 2D vectors and draws lines on a given location on the widget. With these two functions, we can draw the waveforms and other info on the screen.

```

int32 SRPPWaveformCanvas::OnPaint(const FPaintArgs& Args, const FGeometry& AllottedGeometry, const FSlateRect& MyClippingRect, FSlateW
{
    FSlateDrawElement::MakeLines(OutDrawElements,    //render border
        LayerId,
        AllottedGeometry.ToPaintGeometry(),
        BoarderBox,
        ESlateDrawEffect::None,
        FLinearColor::White,
        true,
        1.f
    );

    FSlateDrawElement::MakeLines(OutDrawElements,    //render waveform
        LayerId,
        AllottedGeometry.ToPaintGeometry(),
        URPPUtility::DrawArray,
        ESlateDrawEffect::None,
        FLinearColor::Yellow,
        true,
        0.5f
    );

    SRPPWaveformCanvas::DrawBeatGrid(RPPMain->AudioCursor, AllottedGeometry, OutDrawElements, LayerId);

    return SCompoundWidget::OnPaint(Args, AllottedGeometry, MyClippingRect, OutDrawElements, LayerId, InWidgetStyle, bParentEnabled);
}

```

Fig 12. Code Snippet

DrawArray and BeatDrawArray are updated every frame so all this process needs to do is call the function MakeLine in the OnPaint function. The MakeLine function will draw a line on every element in the 2D vector passed in consecutively. This is fine for the DrawArray but not suitable for BeatDrawArray because it will connect the bottom of line a to the top of a line b. Unlike OpenGL where one of the draw modes is to draw a single line every two data points, in order to achieve this, multiple calls to MakeLine must be made.

```

/*
this function is called multiple times while on painting phase.
e.g., if theres 2 beat line in presence, BeatDrawArray will have 4 points [1top, 1bot, 2top, 2bot]
this function groups 1top and 1bot as the first line, 2top and 2 bot as the second line.
*/
void SRPPWaveformCanvas::DrawBeatGrid(float CurrentCursor, const FGeometry& AllottedGeometry, FSlate
{
    for (int32 i = 0; i < URPPUtility::BeatDrawArray.Num(); i += 2)
    {
        FSlateDrawElement::MakeLines(OutDrawElements,    //render beat grid one by one
            LayerId,
            AllottedGeometry.ToPaintGeometry(),
            { URPPUtility::BeatDrawArray[i], URPPUtility::BeatDrawArray[i + 1] },
            ESlateDrawEffect::None,
            FLinearColor::Green,
            true,
            0.5
        );
    }
}

```

Fig 13. Code Snippet

Besides audio waveform and beat grid, the border and the snapline are also drawn at this stage.

This process uses Unreal's MakeLine to display the underlying data in DrawArray and BeatDrawArray.

Features

BackToBeginning sets the level viewport back to the beginning of the soundtrack and the starting of the game. *Reset* reloads whatever soundtrack detected in the PluginManager. So, the end-user can use this button to reload the current soundtrack, acting as a soft reset or switch to a new one.

RefreshRunningSpeed gets all the EventBase class in a scene and recalculates their positions according to the running speed set in the PluginManager. For example, if the end-user wants to change the player's running speed from 600 units/s to 800 units/s, after the end-user sets it in the PluginManager, pressing this button will move the event 1 second from 600 units to 800 units. With this feature, the end-user can adjust the player's speed more efficiently and therefore iterate more times. *BeatStartingPoint* lets the end-user set the beat starting offset. For example, not every soundtrack's first beat begins as soon as the soundtrack plays. Most of the time, the window between the first beat and the start of the soundtrack is uncertain, and it needs a human to calibrate.

Workflow

In a typical workflow, the designer needs to drag RPPPluginManager to the scene, which is an actor packed into the plugin content. In the detail panel of the RPPPluginManager, the designer can set variables such as the soundtrack and the player running time. After setting the variables, the designer can then enable the plugin. The plugin will then show the interface and the designer can start putting actors into the scene.

In addition, the designer can also inherit from RPPEventBase to create their own events. In this way, whenever the designer adjusts the player's running speed, he or she can press the "ResetRunningSpeed" button to quickly adjust the location of all the events in the scene.

For a complete document for the plugin, see appendix - B.

The Game

To demonstrate the functionality of the RPP, the plugin comes with an example level. As mentioned above, rhythm platformers are a series of events that need to be resolved at the correct audio cue.

One of the most popular events in platformers is the obstacle and the resolution is to destroy it. When resolving a rhythm event, the player can be one of the three: too soon, on par, and too late.

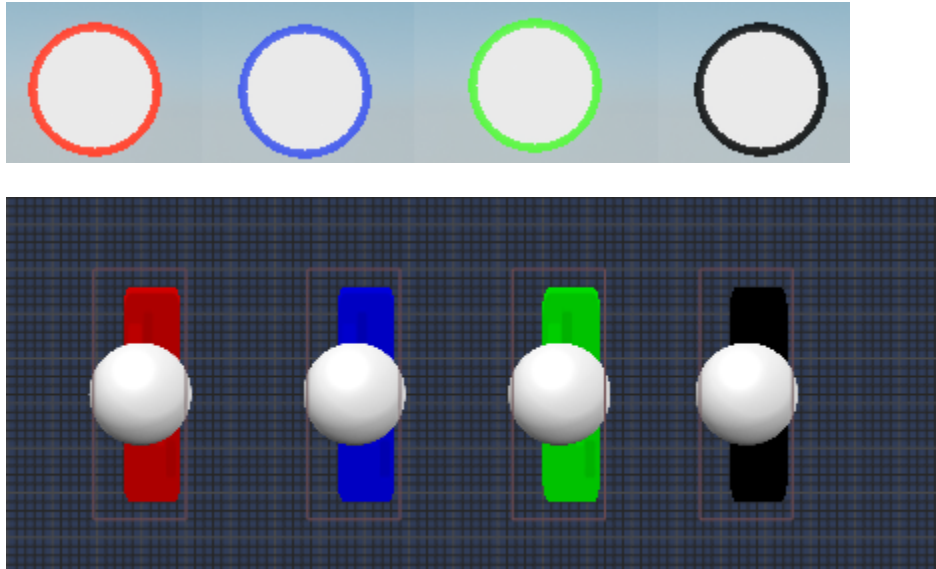


Fig 14. Four colors and blocks

The game asks the player to do two things: matching itself's color to the obstacle and performing the appropriate actions at the correct time. There are 4 types of color in total: red, blue, green and black. And two types of obstacles: simple blocks and hold blocks.

Player

The player has two actions to perform: change color and perform action. Players can perform these actions at any time. When performing the action, the player's color will turn to solid.

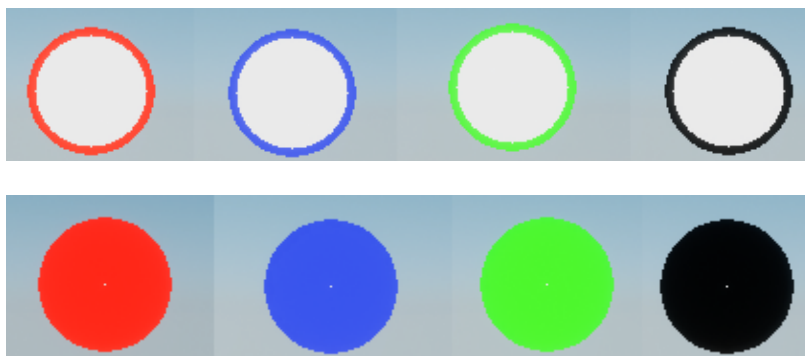


Fig 15. Four colors and action states

Blocks

Blocks are a basic component to the demo scene. They require the player to hit within the time frame to resolve. Note that the detection period (yellow collision box) starts before the beat (red line) to encompass when the player resolves the event too soon. The detection logic works like such: as soon as the player enters the detection period (overlapped with the collision box), the block enables a timer to constantly check the player's state. Within this period, if the obstacle detects the player's action, the block will first check the player's color to itself. If matched, this block is successfully resolved. There are two types of blocks in the level: simple and hold.

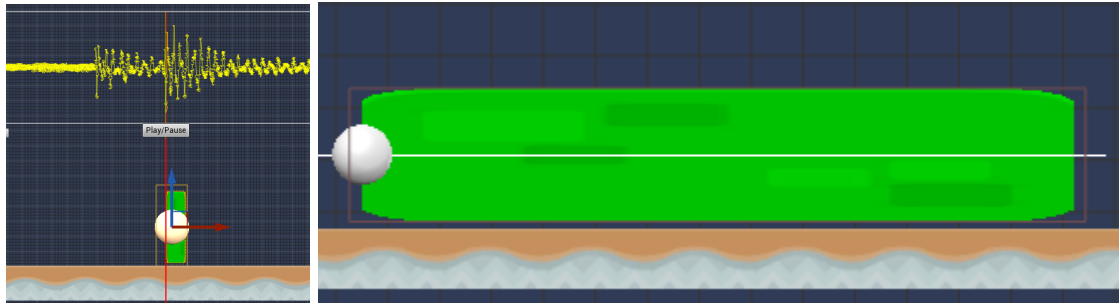


Fig 16. Two Types of Blocks

Hold blocks need the player to remain in the action state for the time being in the block.

The missing traditional platformer element

Jump?

In a traditional platforming level, jumping is crucial when navigating in the game. And during a previous version, jumping over a gap does exist in the level. But what should happen when the player fails to jump over the gap and fall into the ground? In a previous build, like other regular platformers, the player would respawn a couple of steps before the gap, rewind the soundtrack and have the player try again. In addition, jumping on the ground (vault over obstacles) has its problems too: the jump distance is different with the different player's running speed using Unreal's default jump method (vid Jump node). Therefore, all the jump locations have to be re-adjust if the designer wants a new running speed.

Considering all of these factors, jumping is removed from the demo game. We envisioned a jump pad to help the players to gain altitude instead. The jump pad works like a spline line. In Unreal, a spline line is a predefined path for positional data. Given the horizontal difference, it calculates the time between two endpoints and linearly interpolates the position along the line and assigns that location information to the player character. Unfortunately, due to time limitations, this jump pad is not available at the time of testing.

Range action

Similar to jumping, the flying distances for the projectiles are much more complicated to handle compared to melee action. We envisioned a flying projectile that travels in the measure of beats. For example, the designer can specify that a projectile travels a third, half or one full beat and so on. The projectile travels twice the speed as the player character. In this way, if a projectile's flying time is 1 beat, and the soundtrack's bpm is 60 and the player running speed is 600 units/second, the projectile will fly for 1 second at the 1200 units/second, covering 1200 units in distance. There are two approaches for the placement. If the designer wishes the player to throw the projectile at the first beat, the designer needs to place the obstacle at the third beat. Or, if the designer wants a point of contact at the forth beat, then he must signify the player to throw the projectile at the second beat.

However, due to time constraints, and playtest players having trouble managing switching colors alone, ranged action is not implemented for the game.

Evaluation

The goal of this evaluation should be simple -- to see whether the participants can make a rhythm platformer using the tool. We sent out invitations and asked for WPI students who have some level of Unreal experience to volunteer for the test and eventually four people finished and returned a level. Finding people with experience in Unreal and the time and interest in developing a rhythm platformer was quite difficult. We then asked the participants to take a survey.

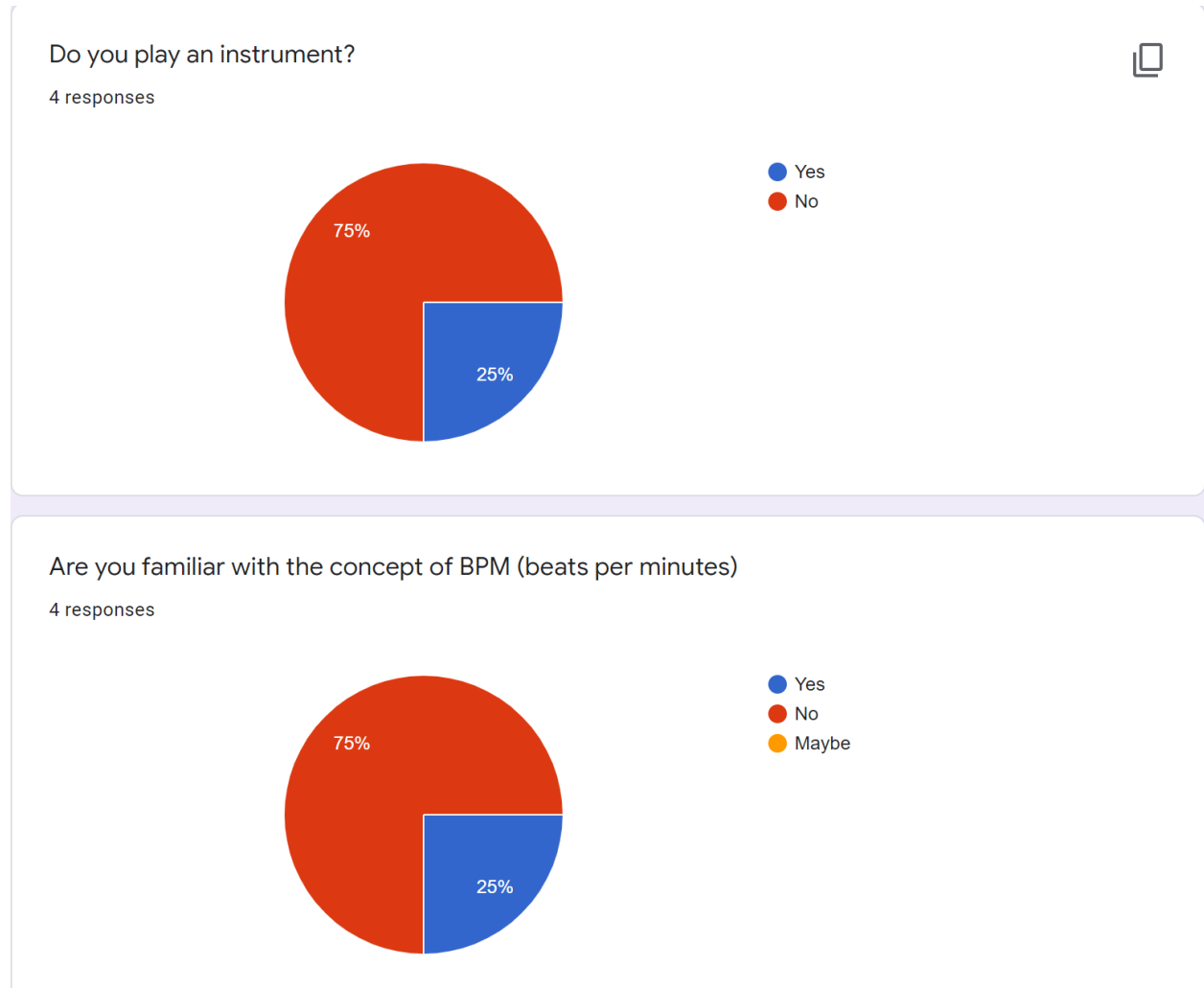


Fig 17. Evaluations

We begin by asking whether the participant plays an instrument themselves and are they familiar with the concepts of BPM (beats per minutes). Three out of four participants are not familiar with these concepts. Which makes us wonder whether the familiarity with music will affect the development process.

How much time did you spend with the plugin?

4 responses

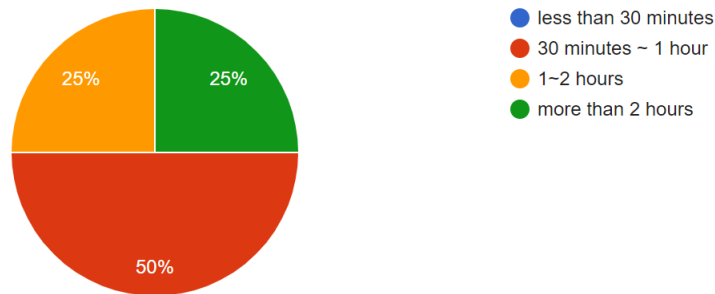


Fig 18. Evaluations

We then asked how much time they spend with the tool. Most of the participants spend less than 2 hours with the tool. And for the one who does spend more time, he implements his own jump mechanics and assets.

Audio synchronization problem

How well do you think the event is synced to the music?

4 responses

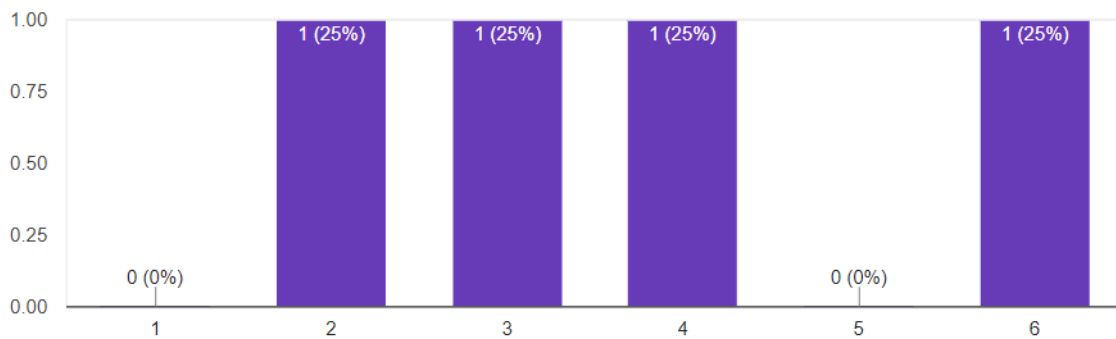


Fig 19. Evaluations

We asked how well do they think the obstacles they placed are synced to the music, which is a crucial aspect of the plugin. Unfortunately half of them don't consider the events are sync well. We suspect that there are two possible explanations: the placement error and perceived error.

Placement error

The placement error comes from the designer side, whether it is due to the designer themselves or the limitation of the plugin. The error of the placement is large enough to sense offbeat. This may be due to the carelessly placed from the designer, or the beat setting is simply wrong to begin with. Since three of 4 people did not know what BPM is, it is highly possible that they did not use the beat grid at all. So the work flow for them would be to listen to the sound track; and when encountering an interesting place; they would pause the sound track and place an event there. And there will be errors between the time of the cue and the pause. They could slow down the playback rate to minimize such errors but the process is not error proof.

Perceived error

The perceived error comes from the player. It is when the game tells the player that they are not resolving the event to the audio cues. And one big aspect of the game's feedback is audio. The demo obstacles are set up to play an audio effect as soon as the player hits action. Therefore, the event must be resolved within the window where the human ears perceives two sounds as one. To mitigate this effect, we believe in addition to the sound effect played by the player action, there should be another identical sound effect played by the obstacle itself, and this sound must be tied to the audio track cues where the player would perceive the sound effect and audio cues as one. In other words, each obstacle should have a variable converting its location information to audio time and a timer. In this case, the obstacle would start counting when the game starts. If the player resolved the events within the acceptance window, the action sound effect should be played accounting to the obstacle time instead of the action time.

Other error

On top of these two errors, all of these levels were playtested by the designer themselves and therefore we cannot ignore the fact that their perception on measuring the time sync is subjective. More experiments need to be conducted upon this matter and we intend to respond to feedback from developers who use its published form on the Unreal Marketplace.

Postmortem

Plugin development for Unreal Engine seems daunting at the first glance, especially when the official tutorial tells you nothing more than printing hello world in a message box. Even though the end result is not as polished as I expected from the beginning, the tool has the crucial features that I proposed.

What went right

I think the biggest factor in this development is that we have a good strategy to begin with. After the proposal, we quickly recognized three milestones for the project:

1. to take in the soundtrack and paint the waveform;
2. to paint on top of the viewport in the editor;
3. to combine two previous steps and to implement other functionalities.

We first prototyped the waveform painting logic using OpenGL with C++ in the visual studio. By doing so, we hoped we could figure out the core logic without the impediment of Unreal in the way. We achieved this goal by taking all the amplitudes in the WAV file sample points and plotting them in lines using `glDrawLine`. After this milestone, we then moved to the engine and started the plugin development.

The next goal was to draw on top of the editor viewport. This step was extremely difficult due to the lack of materials either from the official documents or from the internet in general. After wandering aimlessly in the Unreal documents, we learned that the editor's interface is largely built using Slate¹⁷. We started to dig into this topic and successfully drew a box in the editor viewport.

The last step is to bring the previous two products together and extend the functionalities. During this phase, we first “glue” the logics together and implement all other functionalities (such as BPM marking, adaptive placing) on top of this base; this is also the traditional development phase. Afterwards, we decided to re-architect the plugin so it fit more to the Unreal standard -- essentially re-writing the plugin given all the knowledge we gathered since the project started. Before this phase, every piece of logic was basically written in one class. But after this rewriting, functions were separated into different classes and modules and the plugin's interface is composed of multiple widgets instead of one giant canvas.

This divide and conquer strategy successfully decoupled features from a giant project into bite-sized chunks and allowed them to be prototyped individually.

What went wrong

Unfortunately, possibly due to the time constraint, some parts of the plugin's function were not used by testers. For example, in all of the playtest levels, the player character runs at 600 units/second which is the default speed from the demo scene. Which means they did not experiment with different running speeds for the player character.

Second, during the development phase, we spent too much time on implementing the tool's functionalities that we did not have time to polish the game. In the demo scene, all the gameplay logical elements were there. For example, it plays the soundtrack and allows the player to interact with the

¹⁷ <https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Slate/index.html> accessed 23 April 2021

obstacles. However, for a game, it lacks polished art, visual effects and a feedback system. It is much more a technical demonstration than a game. Coming from an engineering background, we underestimated the art aspect of the game and did not leave enough time to acquire better arts and visual effects. The end result is more a prototype than a game.

What We Would Do Differently

After the testing, we found out that much of the playtester did not utilize every aspect of the plugin, especially the adaptive placement function. The point of this function is to auto adjust the obstacles given different players' running speed. So that the obstacles are always tied to the audio location instead of level location and therefore giving designers more freedom to experiment with different speed settings without worrying to re-adjust all the events placed. Unfortunately no one uses this function and we suspect it is due to the time constraint.

During the testing, we told the tester to spend around 2 hours with the plugin and create a level with it. If we were to do that again, we would have to write up another testing procedure stating the functions we would like them to try.

Conclusion

With the survey and reviews from the UE market, we believe the plugin successfully delivers the function we proposed -- visualizing character location at a speed given a sound track. With this location information, the designer can plan every kind of event such as placing obstacles or triggering visual effects. This tool is also published to the Unreal Engine market for public access.

In addition to constantly maintaining this tool, we also plan to make another demo scene with improved mechanics and better art assets.

Bibliography

Aubrey, Dave. "Taiko No Tatsujin: Drum 'n' Fun! Review - Tubthumping." *Wccftech*, Wccftech, 2 Nov. 2018, wccftech.com/review/taiko-no-tatsujin-drum-n-fun/.

Byford, Sam. "Original Arcade Donkey Kong Comes to Nintendo Switch in First-Ever Re-Release." *The Verge*, The Verge, 15 June 2018, www.theverge.com/2018/6/14/17466778/donkey-kong-arcade-version-nintendo-switch-e3-2018.

"Electronic Platform Game." *Encyclopædia Britannica*, Encyclopædia Britannica, Inc., www.britannica.com/topic/electronic-platform-game.

Heyworth, Reece. "Rayman Legends Music Level." *Switch Player*, 26 Sept. 2017, switchplayer.net/2017/09/27/rayman-legends-definitive-edition-review/rayman-legends-music-level/.

Pichlmair, Martin, and Fares Kayali. "Levels of Sound: On the Principles of Interactivity in Music Video Games ." *Authors & Digital Games Research Association (DiGRA)*., 2007.

"Plugins." *Unreal Engine Documentation*, docs.unrealengine.com/en-US/ProductionPipelines/Plugins/index.html.

"Slate UI Framework." *Unreal Engine Documentation*, docs.unrealengine.com/en-US/ProgrammingAndScripting/Slate/index.html.

Smith, Gillian, et al. "Rhythm-Based Level Generation for 2D Platformers." *Proceedings of the 4th International Conference on Foundations of Digital Games - FDG '09*, 2009, doi:10.1145/1536513.1536548.

"Super Mario Bros." *Nintendo of Europe GmbH*, www.nintendo.co.uk/Games/NES/Super-Mario-Bros--803853.html.

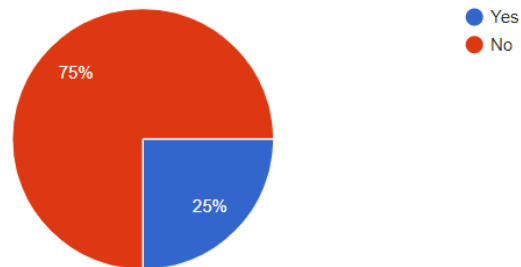
"Temple Run Play Over 1000 Games." *Lasopaan*, lasopaan213.weebly.com/temple-run-play-over-1000-games.html.

"Unreal Engine 4 Documentation." *Unreal Engine Documentation*, docs.unrealengine.com/en-US/index.html.

Appendices A - All evaluation data

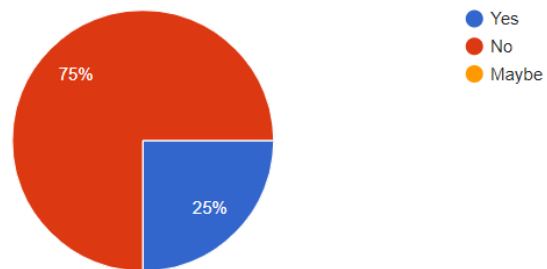
Do you play an instrument?

4 responses



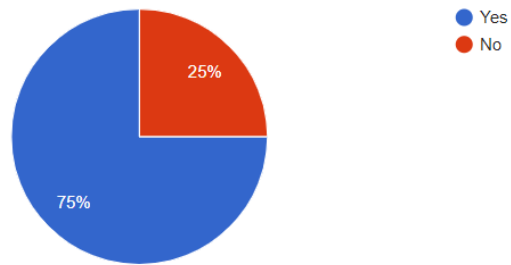
Are you familiar with the concept of BPM (beats per minutes)

4 responses



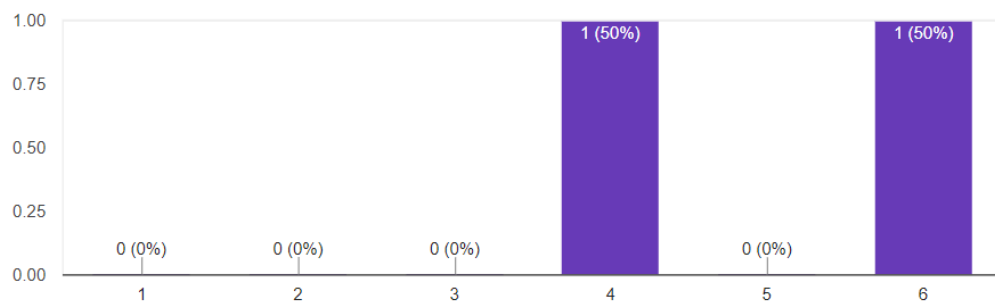
Did you recreate the example?

4 responses



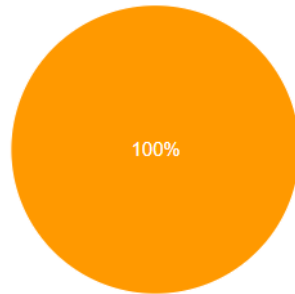
If no, where did you stop?

2 responses



If not, why did you stop?

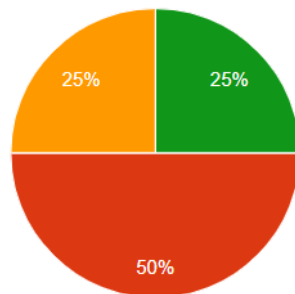
1 response



- Feeling ready half way through?
- Tutorial are too confused to follow.
- I played the example project and I wanted to try making a different kind of game

How much time did you spend with the plugin?

4 responses

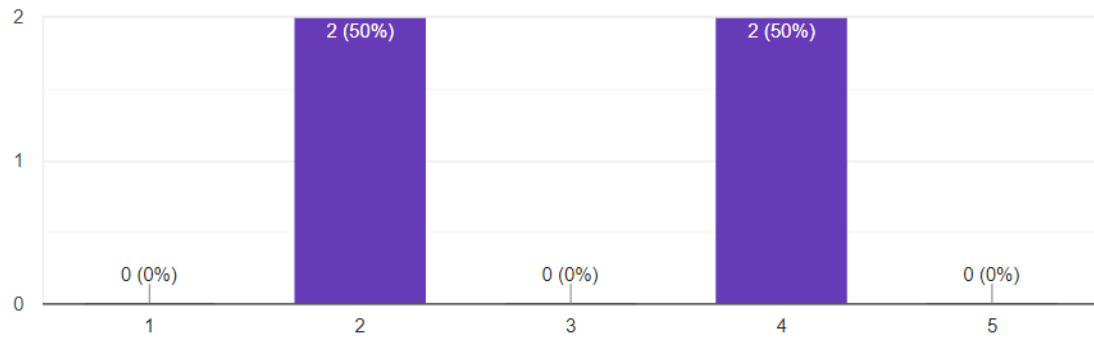


- less than 30 minutes
- 30 minutes ~ 1 hour
- 1~2 hours
- more than 2 hours

How well are you familiar to rhythm games? (RockBand, Guitar Hero, Thumper ...)

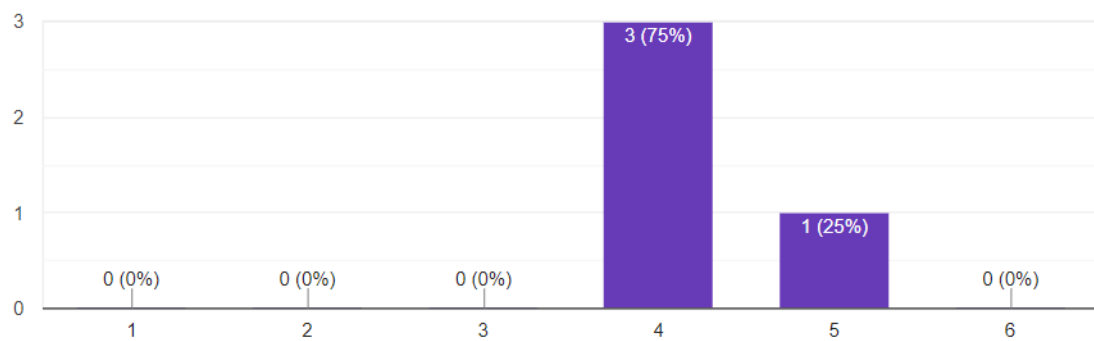


4 responses



How well are you satisfied with the game you made?

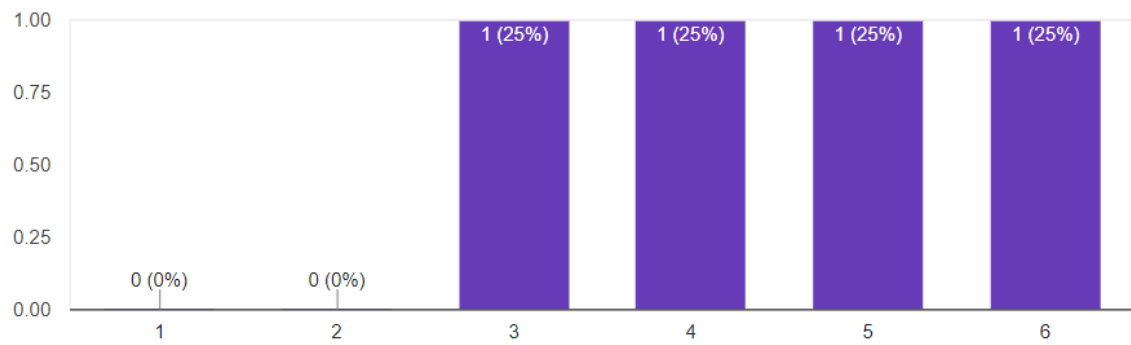
4 responses



How helpful is the waveform?

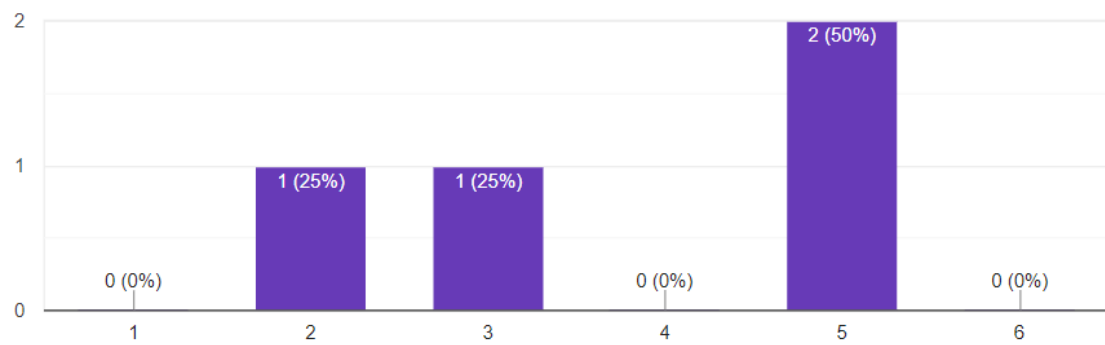


4 responses



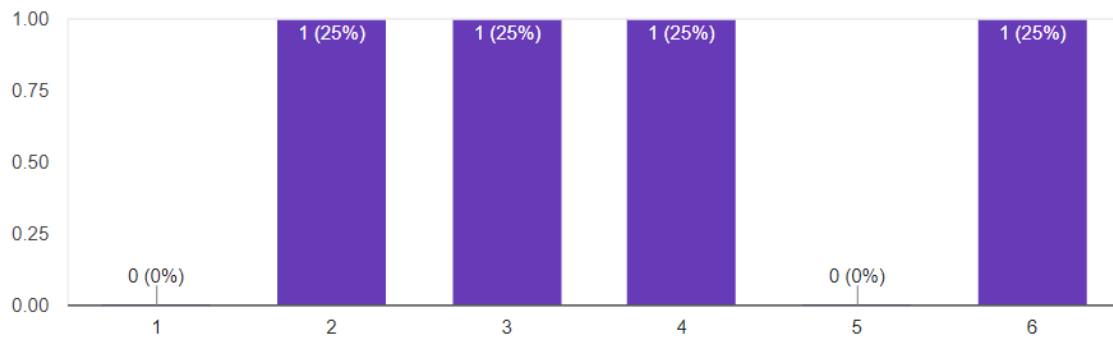
How helpful is the beat grid?

4 responses



How well do you think the event is synced to the music?

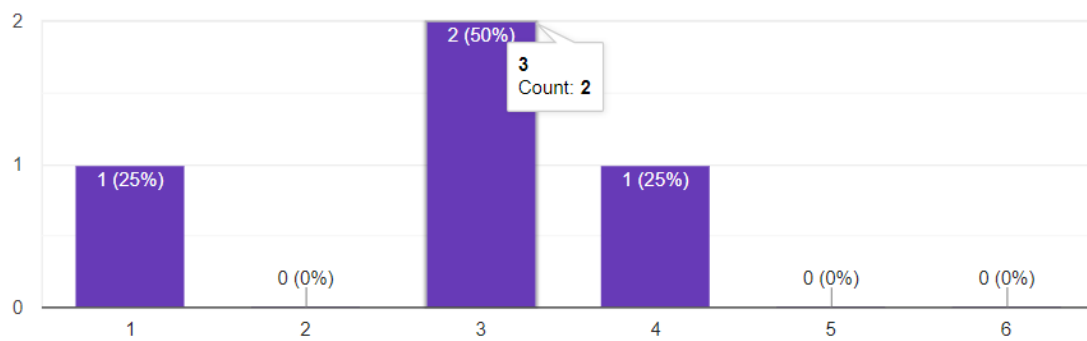
4 responses



How similar do you consider your product to the demo?



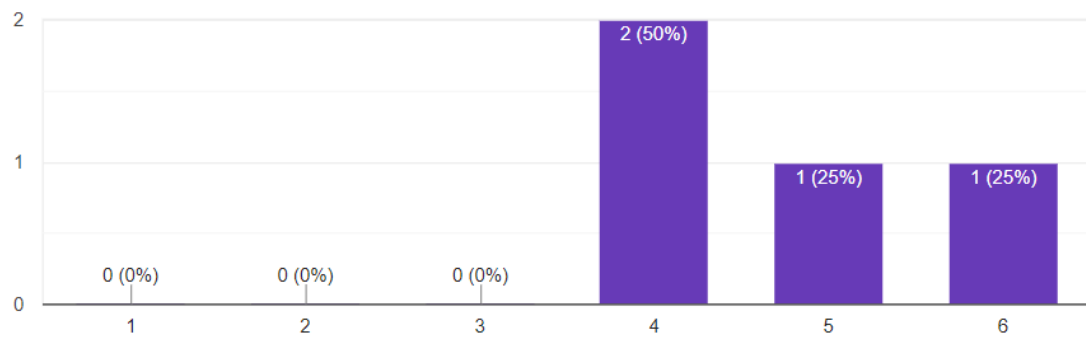
4 responses



How different do you think you can make a game with the plugin, were you given more time?



4 responses



Appendices B - Documentation

Introduction

RPP, Rhythm Platformer Plugin, lets you quickly sketch and produce a rhythm platformer, without the need of figuring out synchronization between audio and character placement. The user can use the waveform to quickly locate the audio location and start design at that point. The end result is a platformer based on a piece of music the designer has based on.

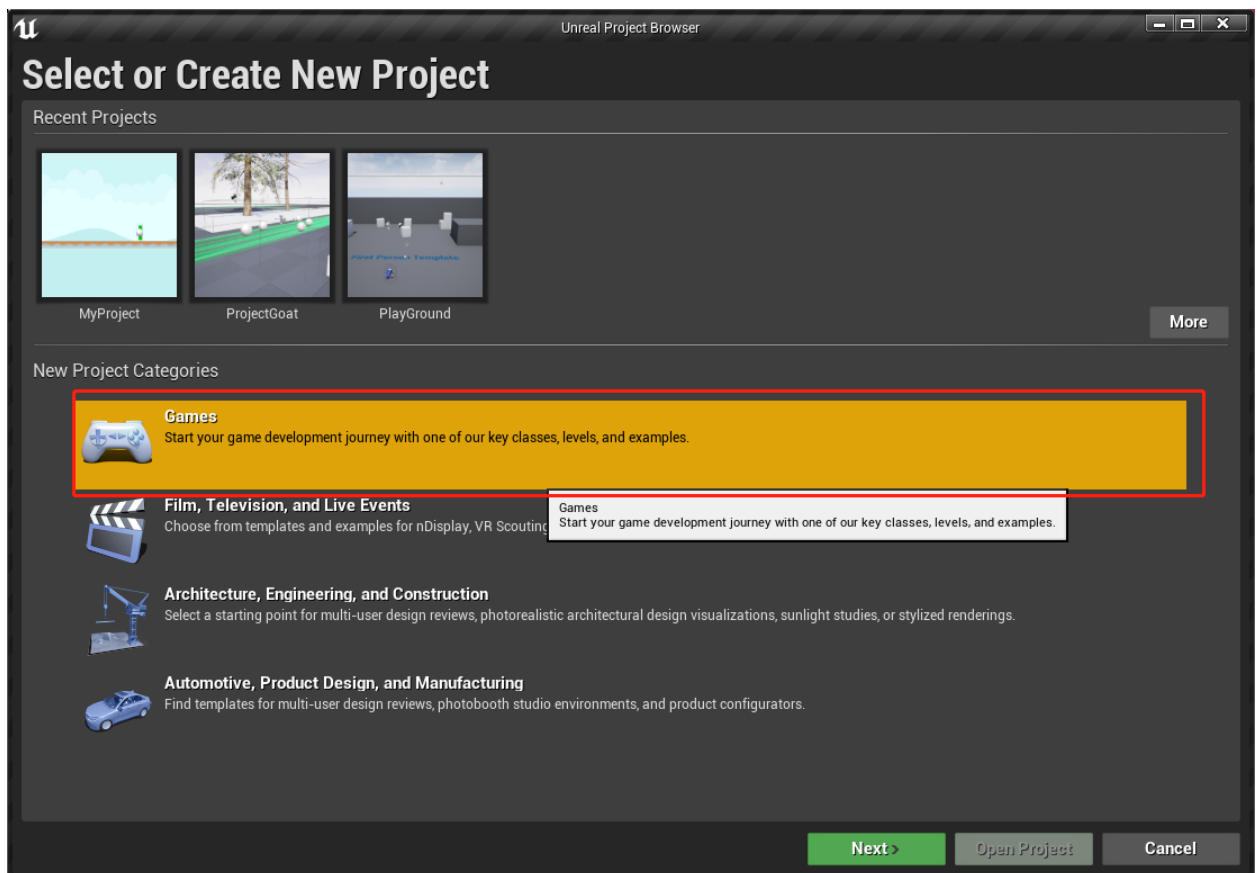
Project and plugin files setup

This is for those who do not have access to the example project or wish to start using the plugin with a new project. For people who wish to start with the example project, jump to the Variable Setup section.

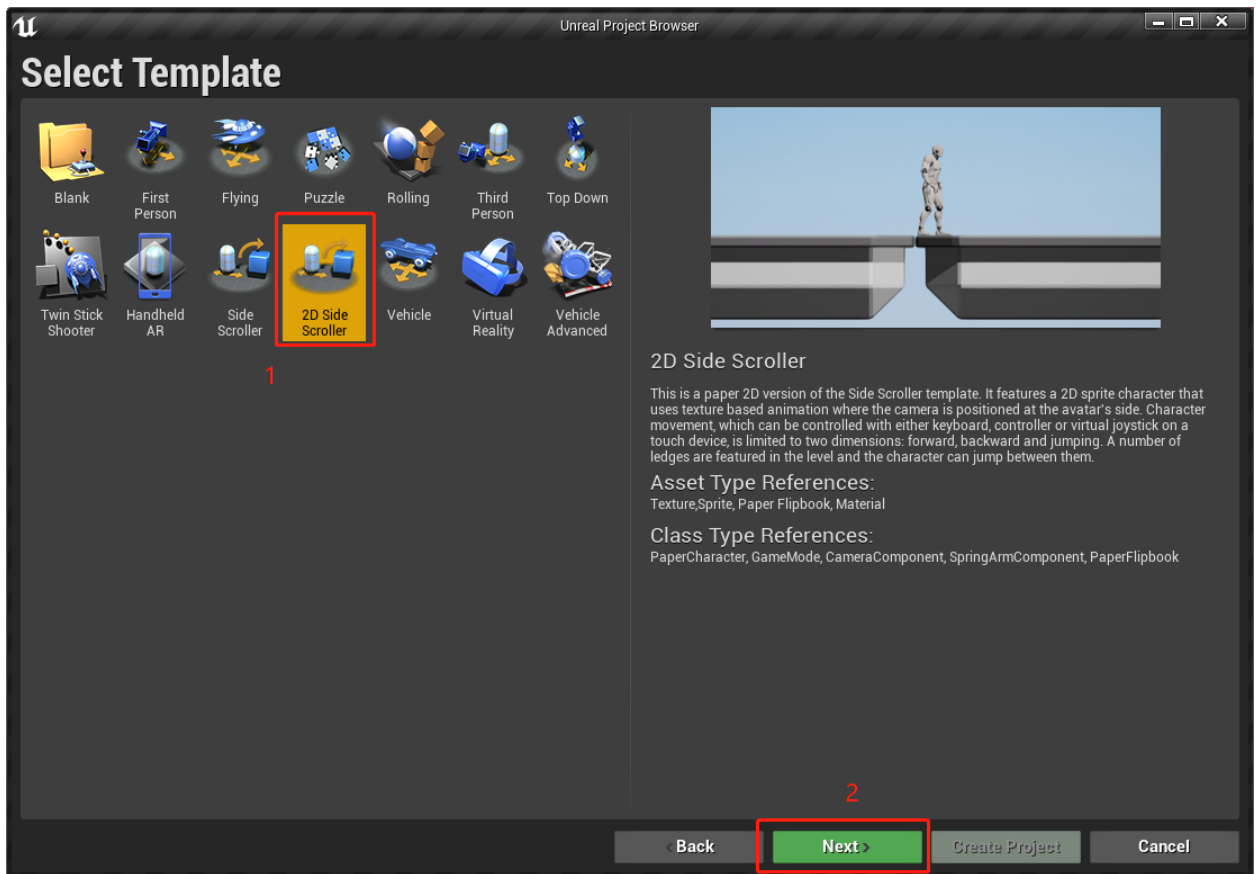
To better illustrate the idea of the plugin, we will use the 2d side scroll template.

Create 2d side scroll template and enable the plugin

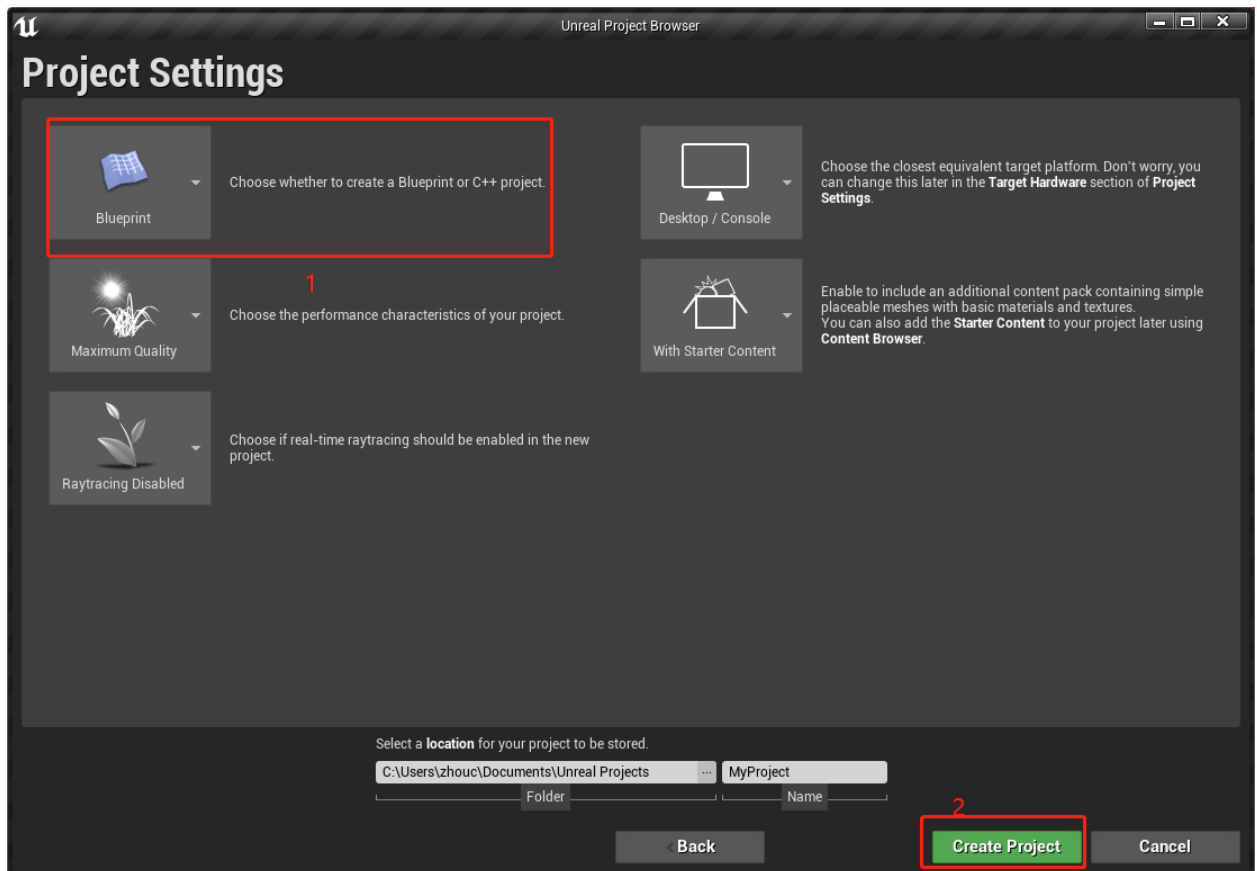
1. select Games.



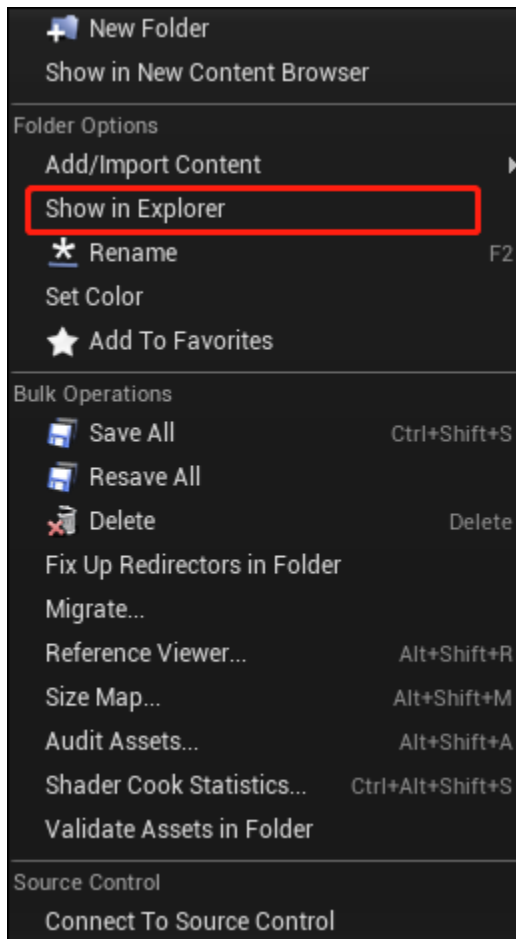
2. select 2D side scroller template and click next



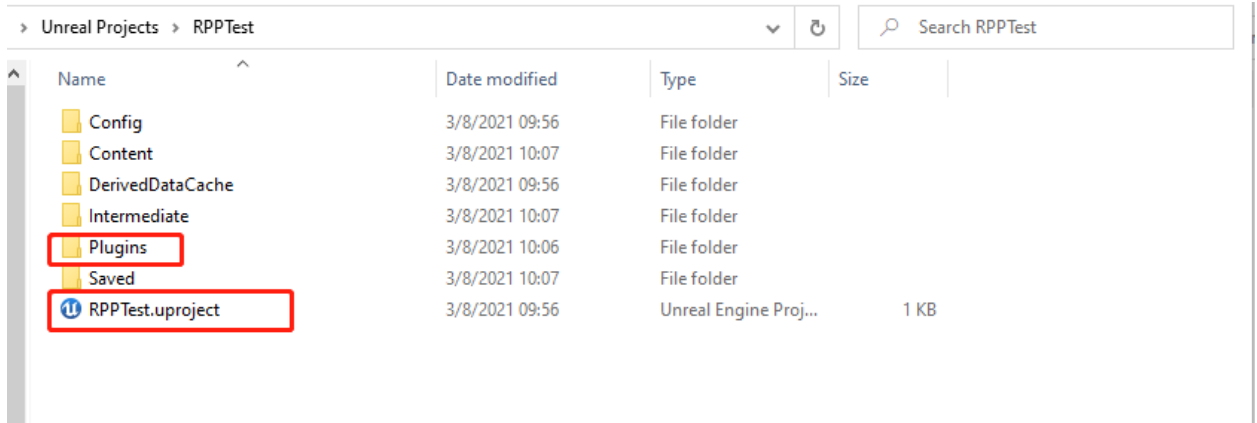
- select the type of project you desire (blueprint vs c++), rename the project and click create project. (if you don't know what the difference between blueprint and C++, i recommend select blueprint. Or you can visit this side to get better understanding of these two)



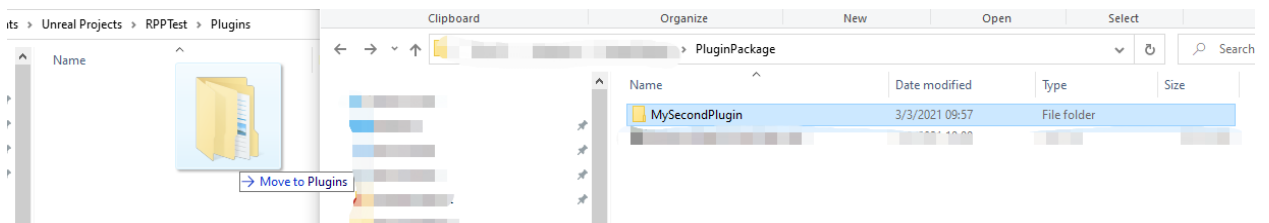
4. In order to open the project directory, you can right click any folder in the content browser and select "show in explorer". This will bring you to the project folder via default file explore.



5. locate the root directory of the project (it will be whatever you named the project), and right click, select New Folder, and name the new folder Plugins to create the Plugins folder if the plugins folder does not exist. (The root directory has the layout below and you can see projectName.uproject)



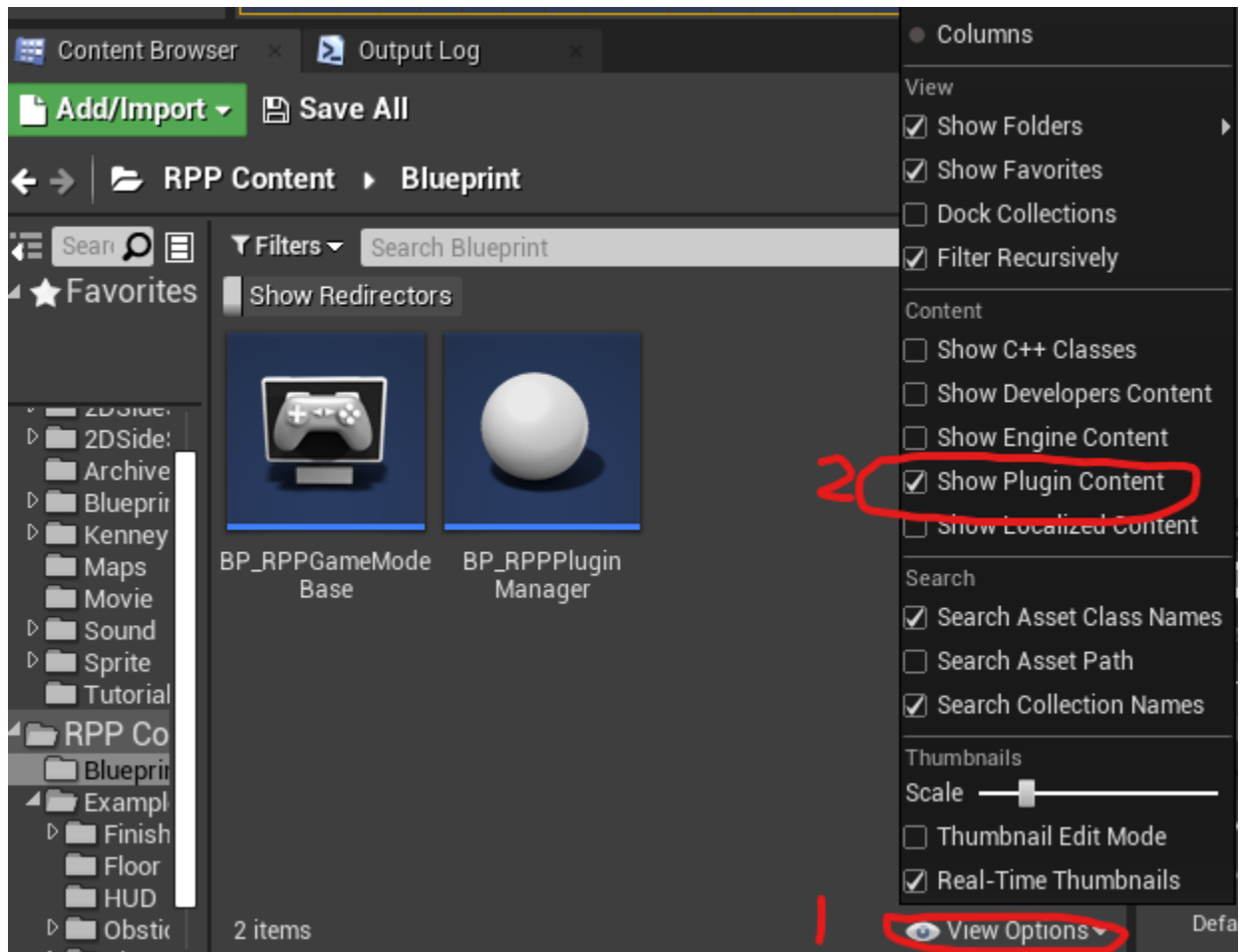
6. download the zip folder from this [link](#).
7. unzip, copy and paste the MySecondPlugin folder into ../Plugins/



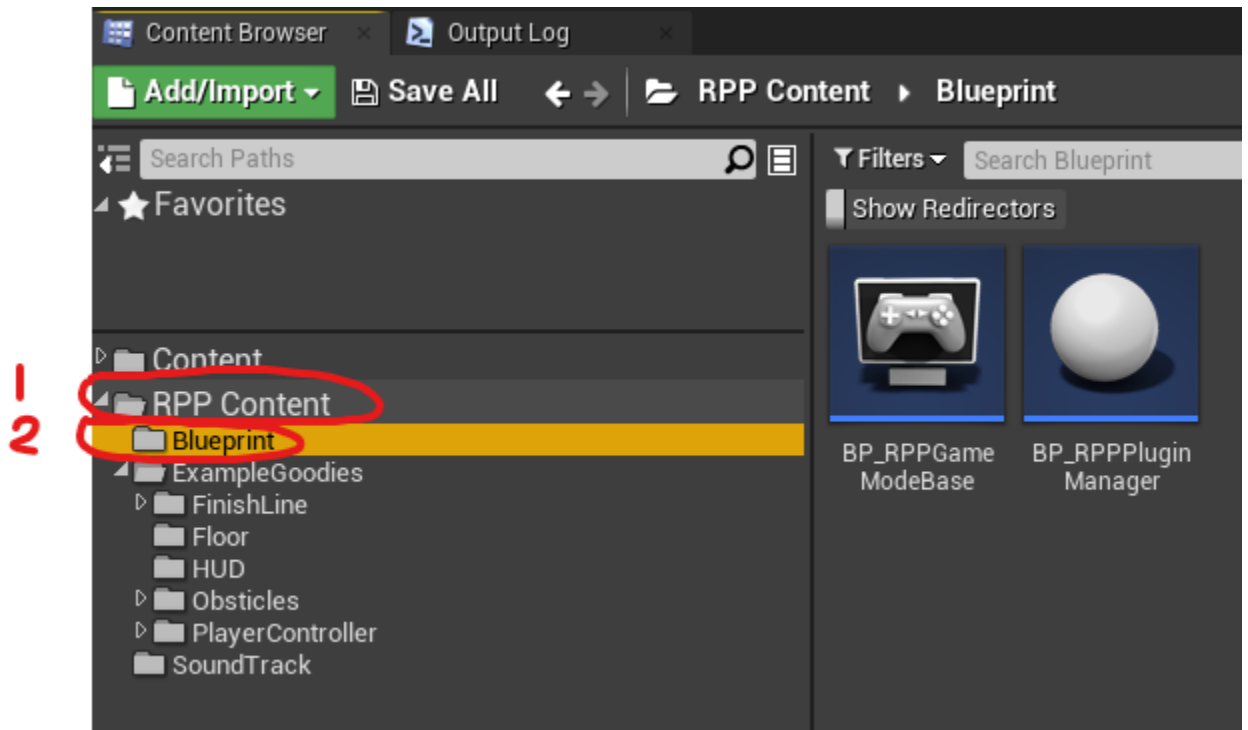
8. Close the project in Unreal Engine, or switch to another project.
9. Reopen the project.

Project Manager Setup

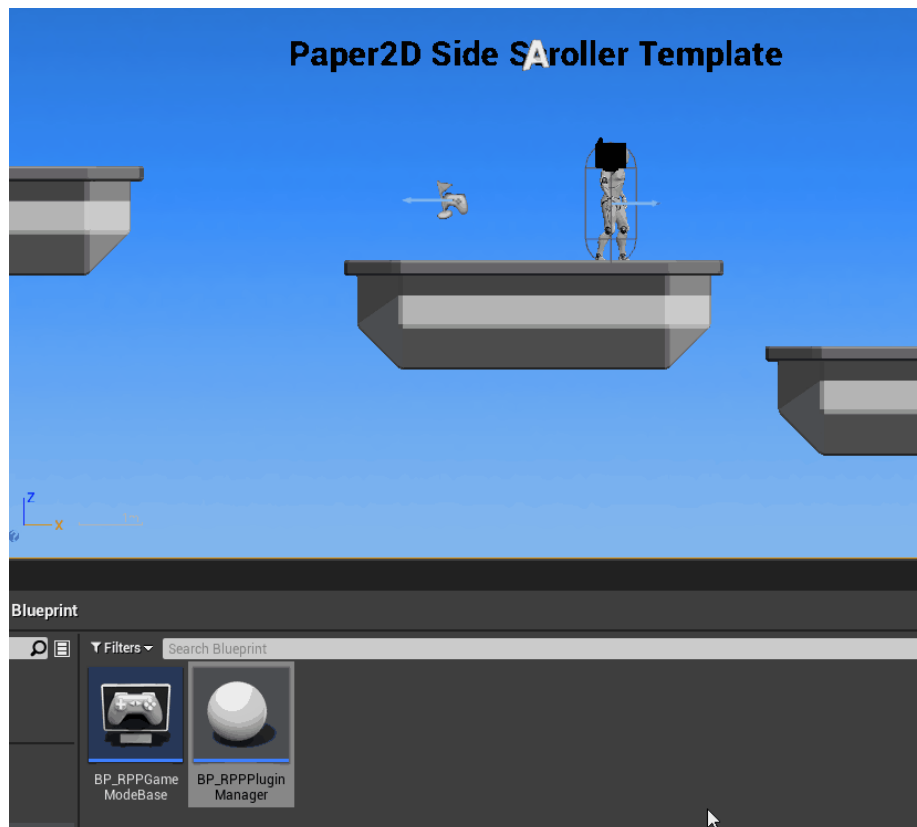
1. Make sure you are seeing Plugin Contents.
 - a. Select View Options under Content Browser.
 - b. Check Show Plugin Content



2. Locate BP_RPPPluginManager and draw that into the scene. Remember, the location of this actor will also serve as the starting point of the plugin.
 - a. Goto RPP Content - Blueprint

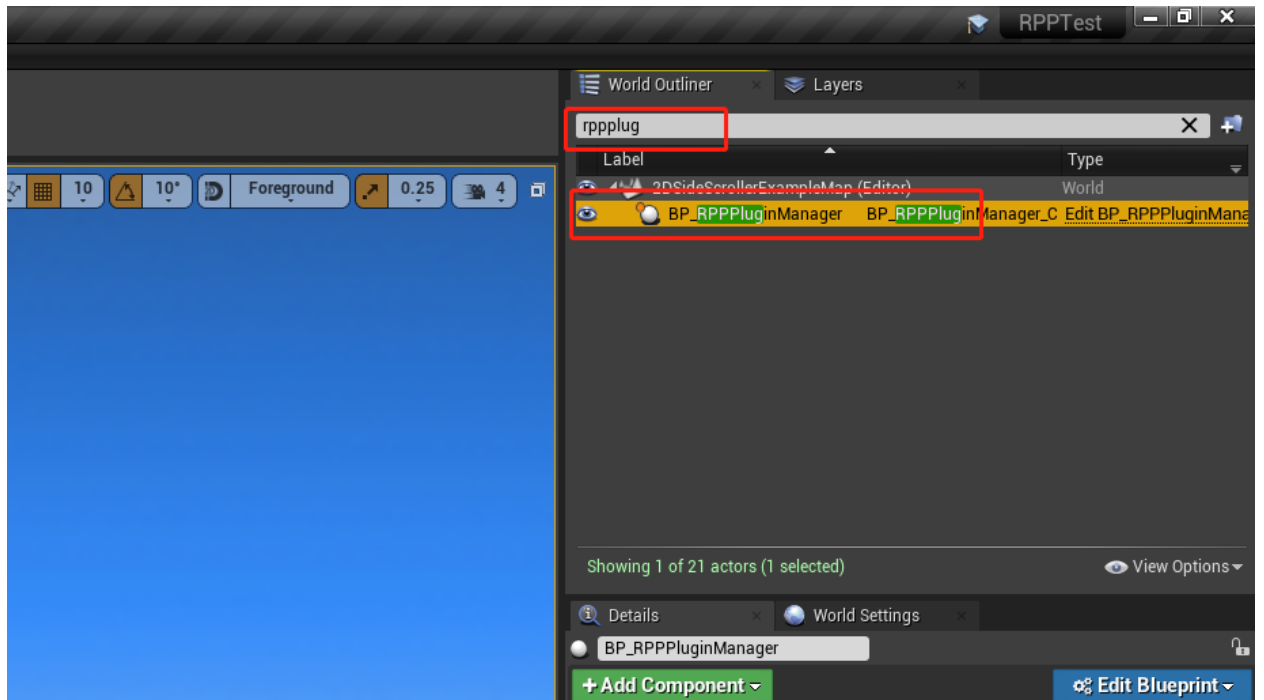


- b. Drag BP_RPPPluginManager to the scene.

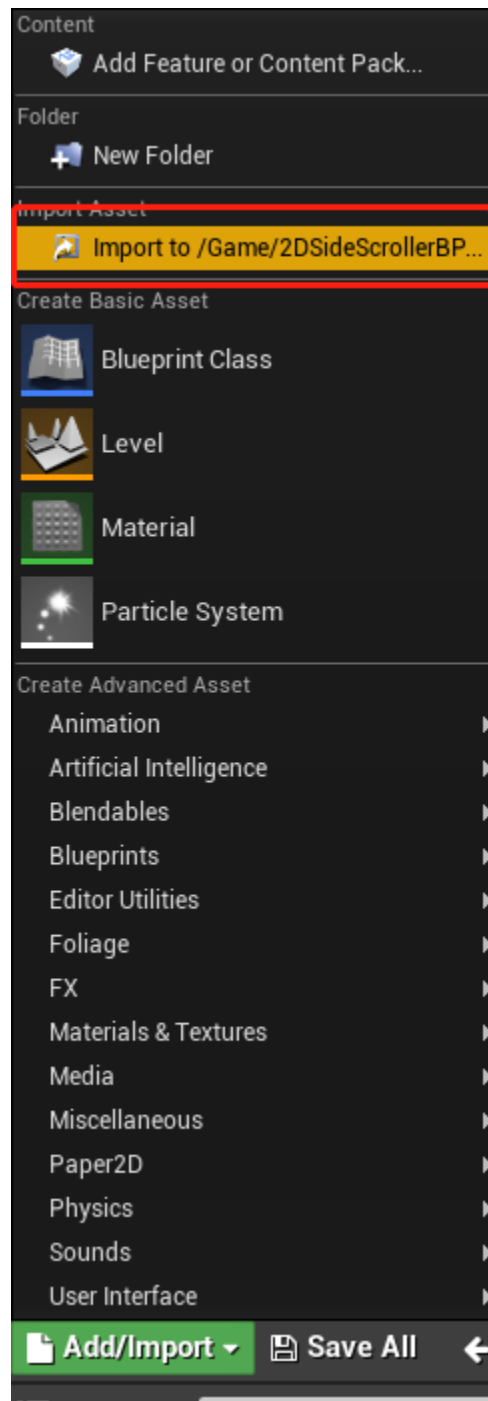


Setup Variables

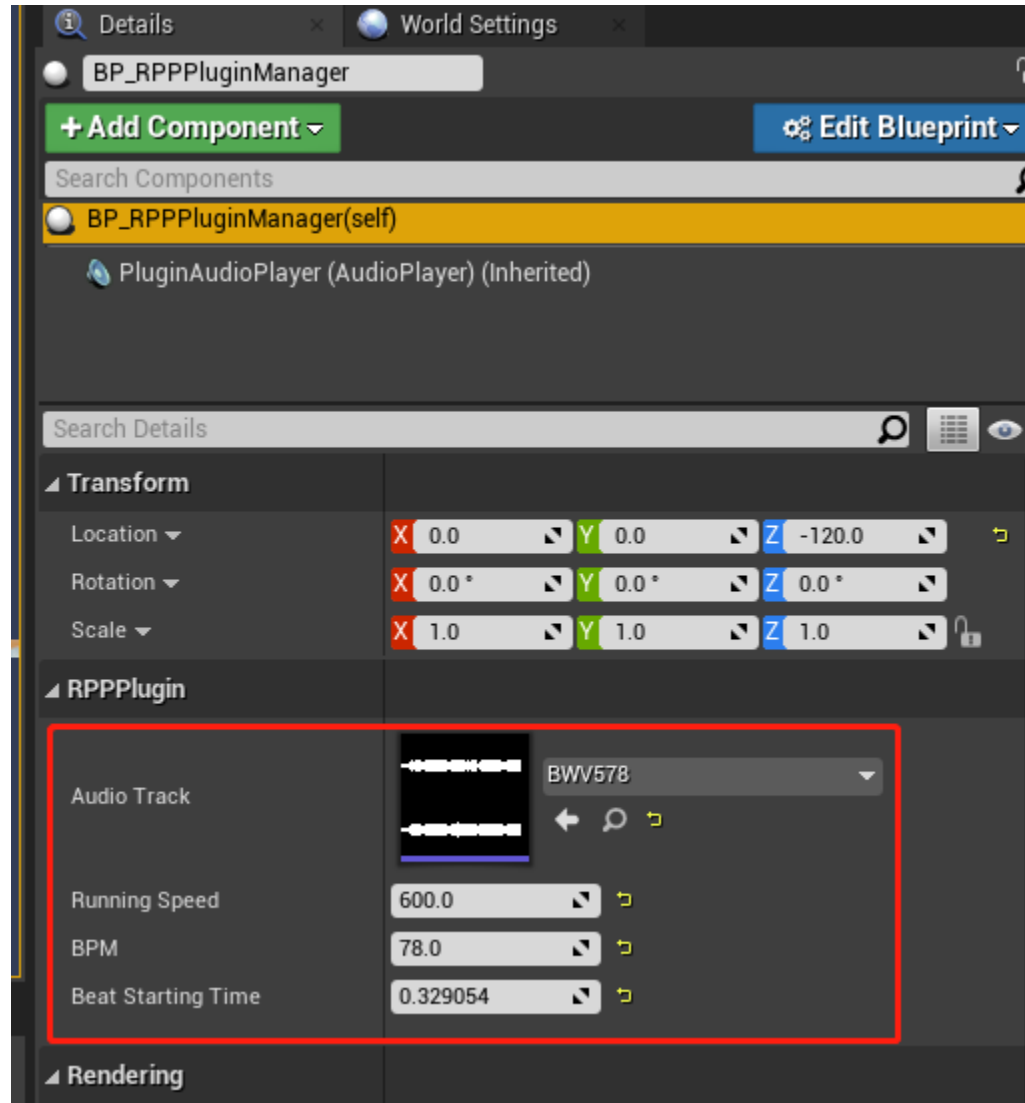
1. Locate the RPPPluginManager actor in the scene and select it (you can search in the world outliner)



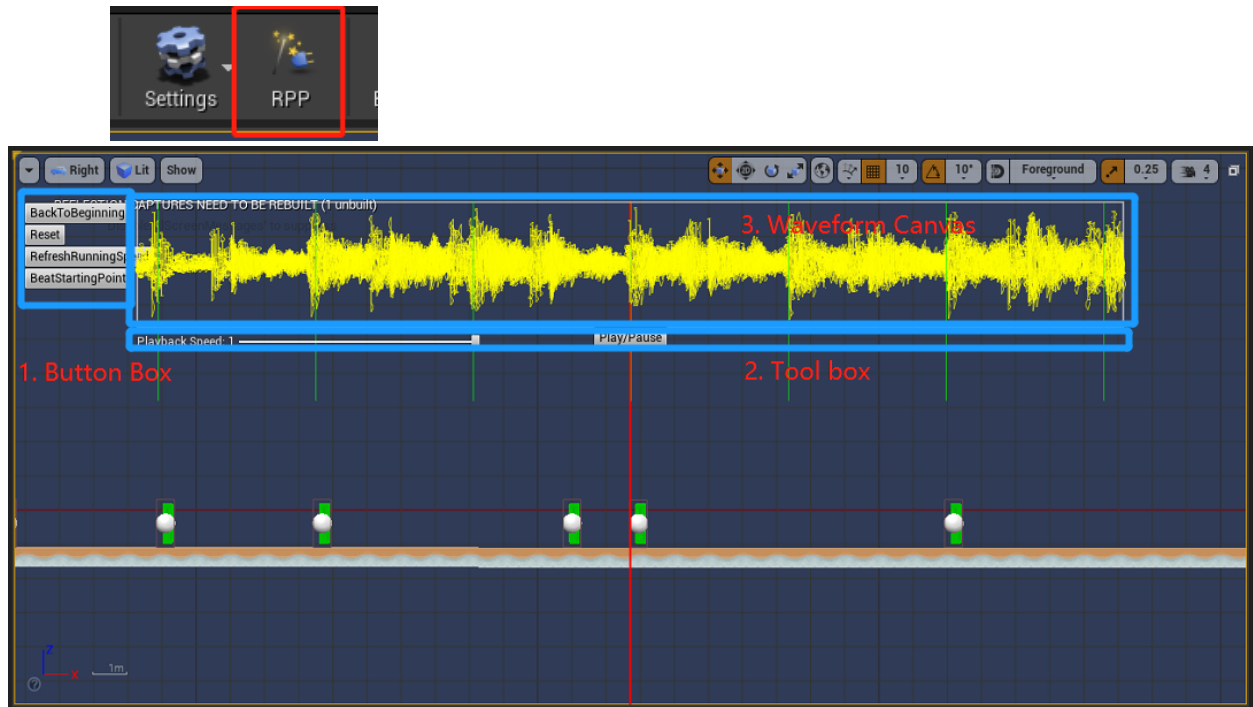
- import the WAV file you are going to use with the level and import it to the project.



3. In the details, make sure these variables are set:
 - a. Audio Track: plug in the audio you are going to use with the level. (WAV format)
 - b. Running Speed: set the desired player running speed. This number must match the player running speed you set to the player character. (Suggest 600 as the starting point, cannot be negative)
 - c. BPM: optional. Shows beat grid as green line on wave canvas (suggest to be 0 for new users) (press reset to apply this setting)
 - d. Beat Starting Time: optional. This variable sets an offset to tell the plugin where the first beat happens.



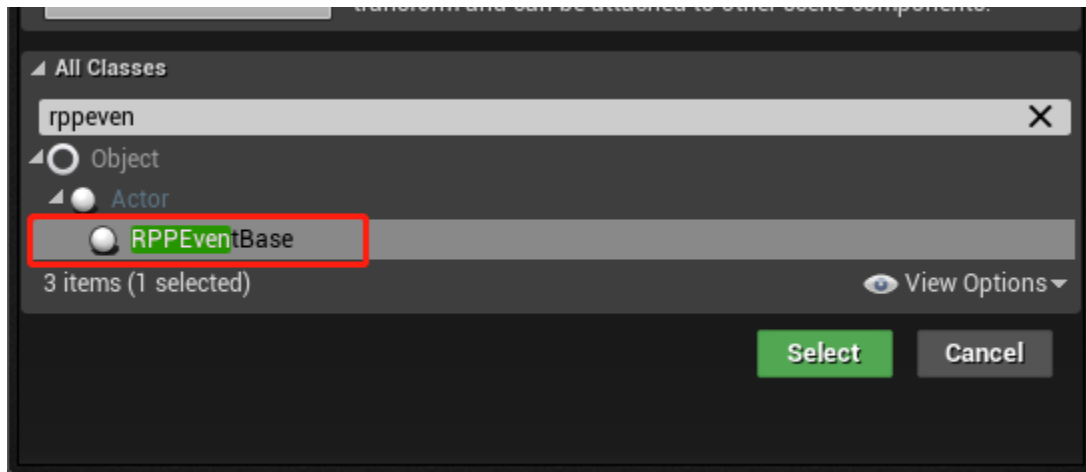
- In the menu bar above the viewport, press the RPP plugin next to the setting button in the menu bar, you should be able to see the waveform.



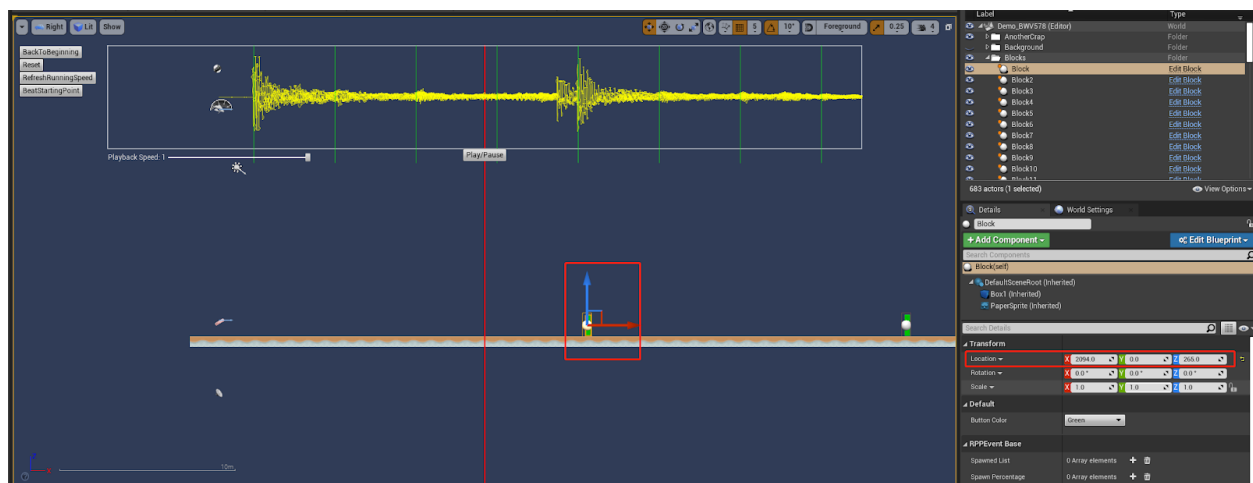
Button	Short description
<i>BackToBeginning</i>	sets the level viewport back to the beginning of the soundtrack and the starting of the game
<i>Reset</i>	reloads whatever soundtrack detected in the PluginManager
<i>RefreshRunningSpeed</i>	gets all the EventBase class in a scene and recalculates their positions according to the running speed set in the PluginManager
<i>BeatStartingPoint</i>	Let the user set beat starting offset to the current location.

Adaptive Placing:

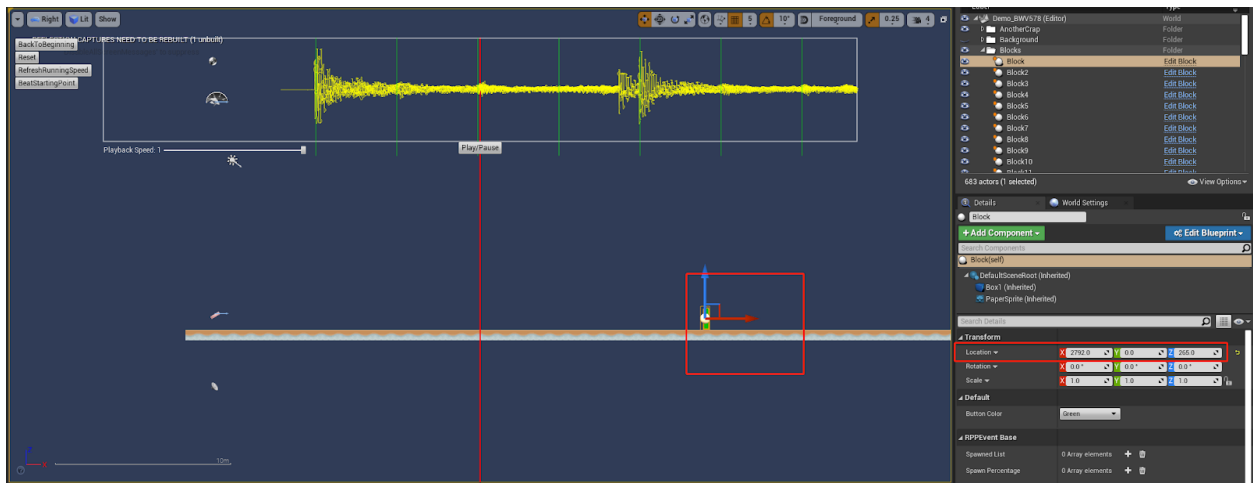
To utilize *RefreshRunningSpeed* function, we recommend making child class of *RPPEventBase* when developing your own obstacle class. In this way, the designer can adjust the player running speed without replacing all the obstacles.



For example, the first block is at 2094 @ 600 running speed.



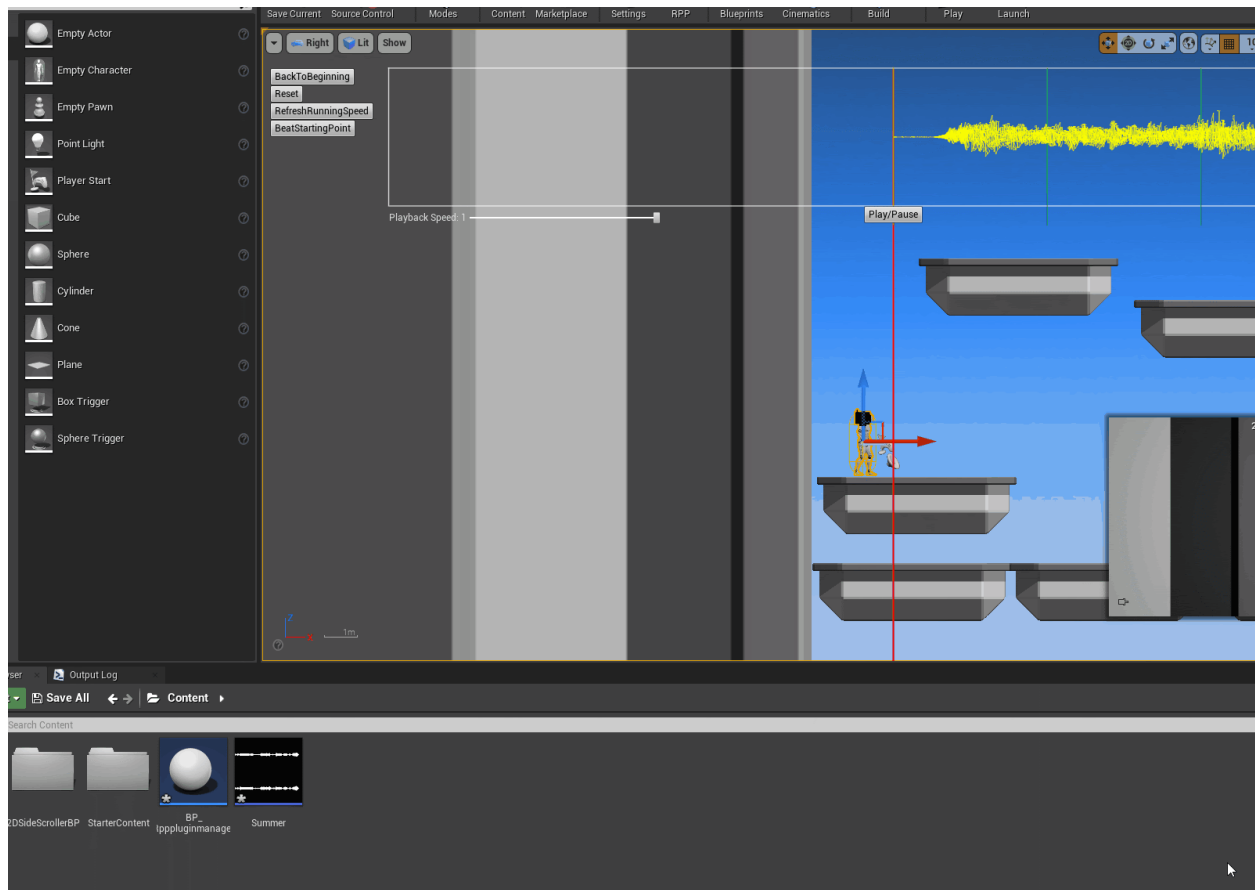
After adjusting the running speed to 800, the first block is at 2792.



Only actor that derives from RPPEventBase can be affected by this function.

Placing Event

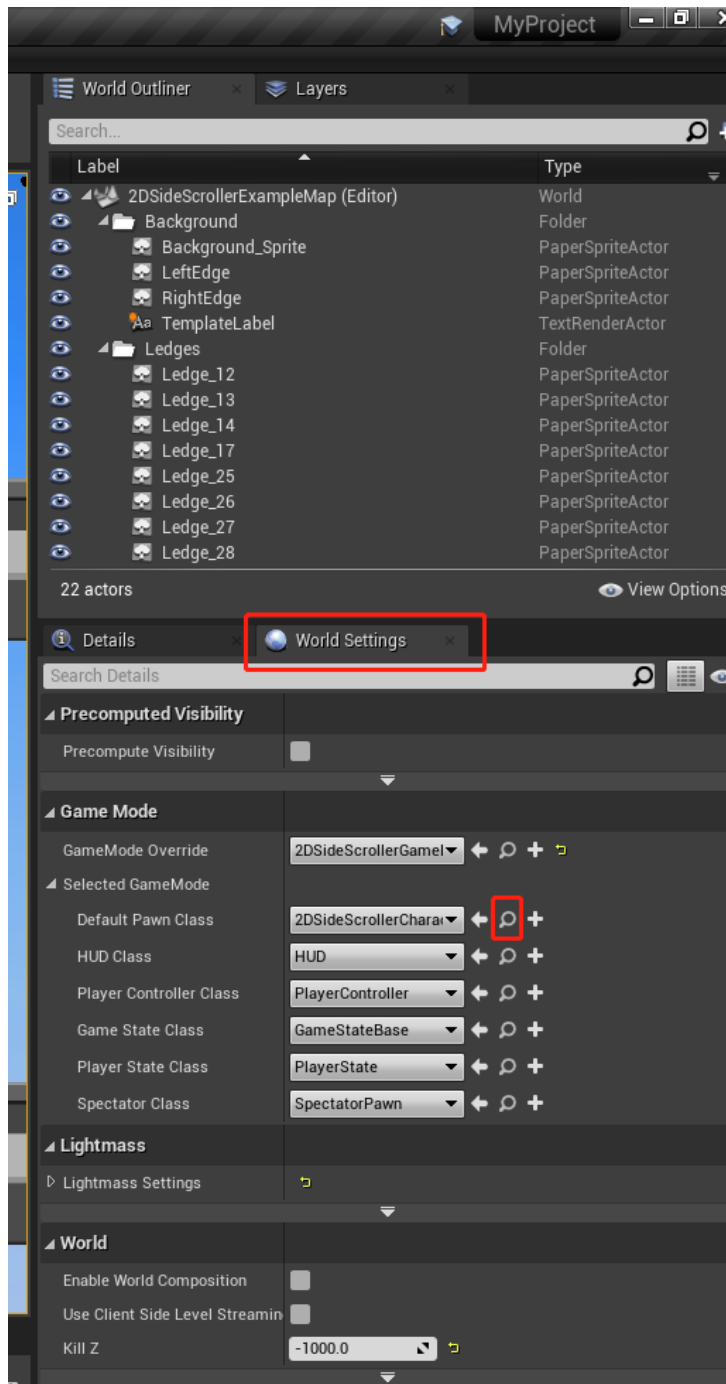
To place an event, the user only needs to drag the actor to an appropriate place. The waveform will help the user choose this place. The centered red line signifies the exact place of the sound that's playing in the waveform, and will again, help the user to place the actor.



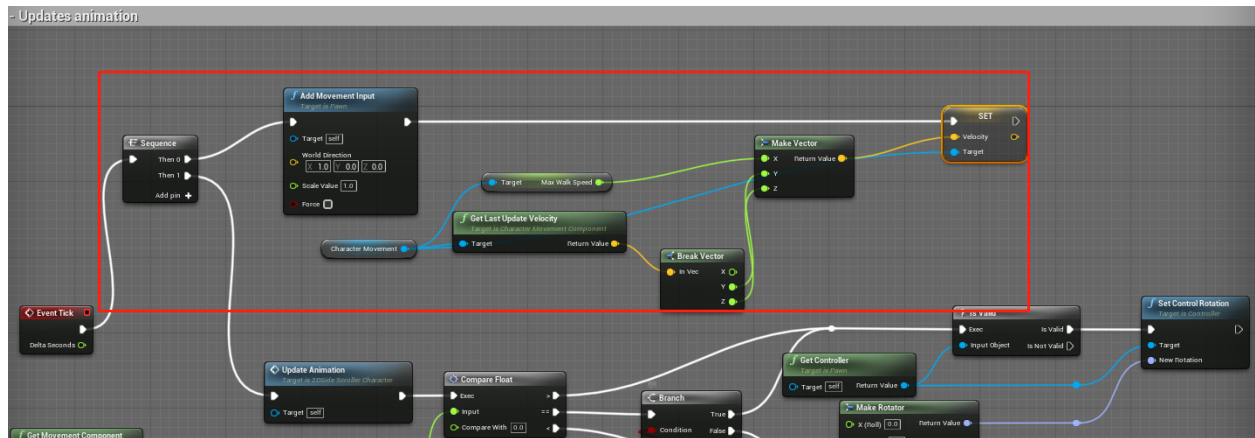
Player Running Speed

The waveform requires the player character runs/moves at a constant speed equivalent to the number that the user set at the “Player Running Speed” tab. There are multiple ways of achieving it.

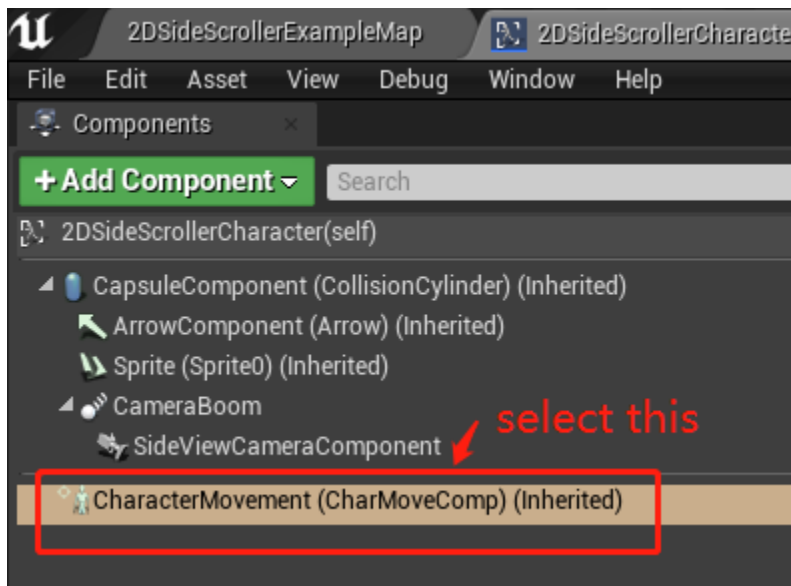
If you are starting with 2D side scroller template, you can edit the player character via World Setting - Default Pawn Class.

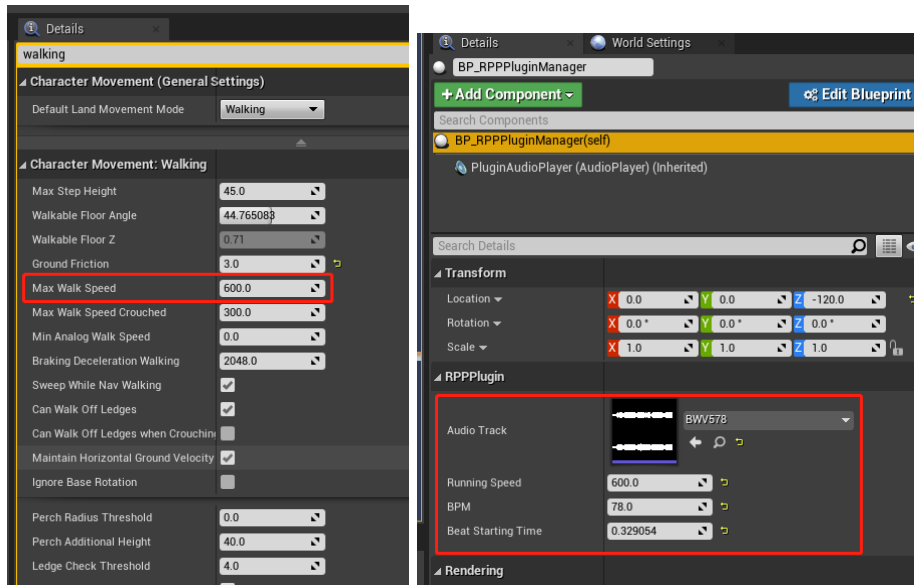


1. In the Tick event, add a sequence and add the following logic. (add movement input (the max walk speed) to the player character every tick)).



2. with the CharacterMovement selected, in the detail panel, search “walk” and change the “max walk speed” that equivalent to “running speed” in the manager



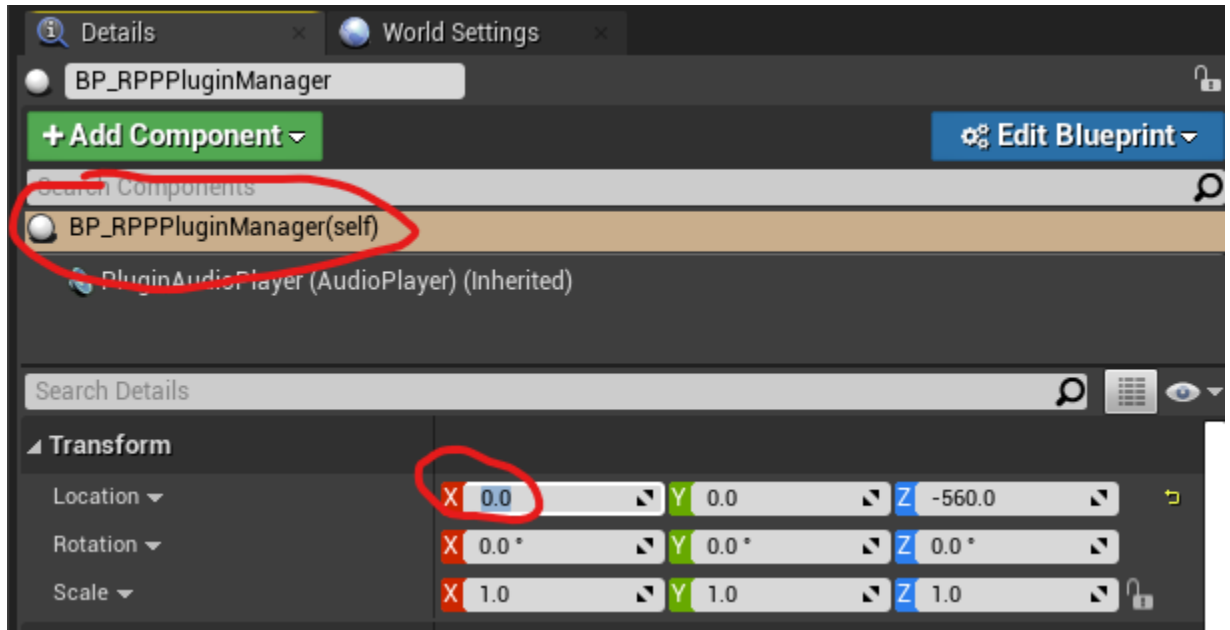


The Max Walk Speed from CharacterMovement Component **must** match up with Running speed in the RPPPluginManager's detail panel.

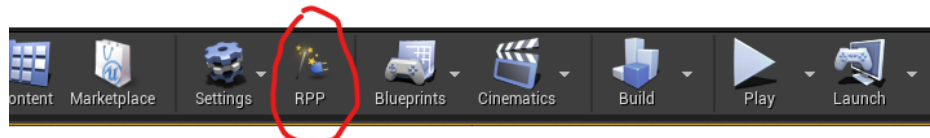
Recommended Work Flow/Reconstruct Example Level

These are the steps of how I constructed the example scene “Demo_BWV578”.

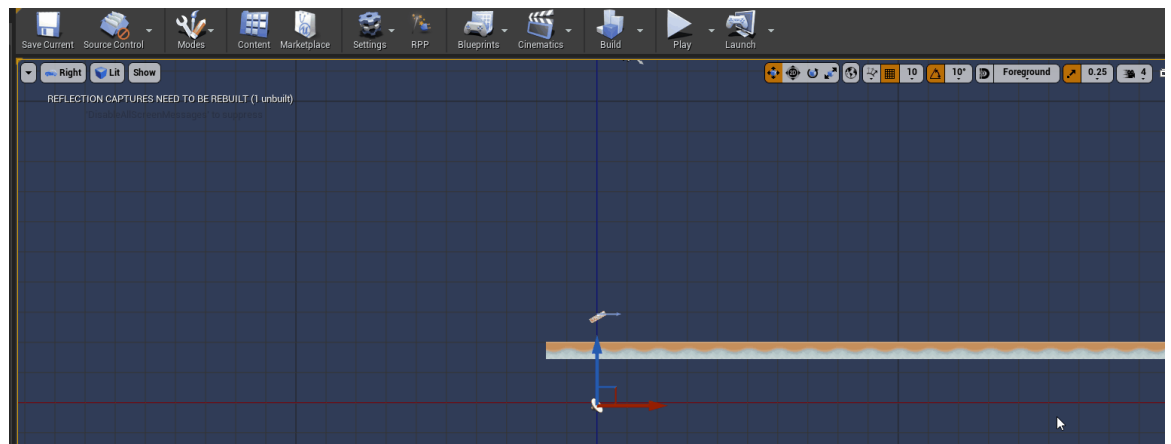
1. Follow steps in “Project Manager Setup”
 - a. Make sure the x-coordinate of the RPPPlugin is set to 0, and place “PlayerStart” actor to this location as well.



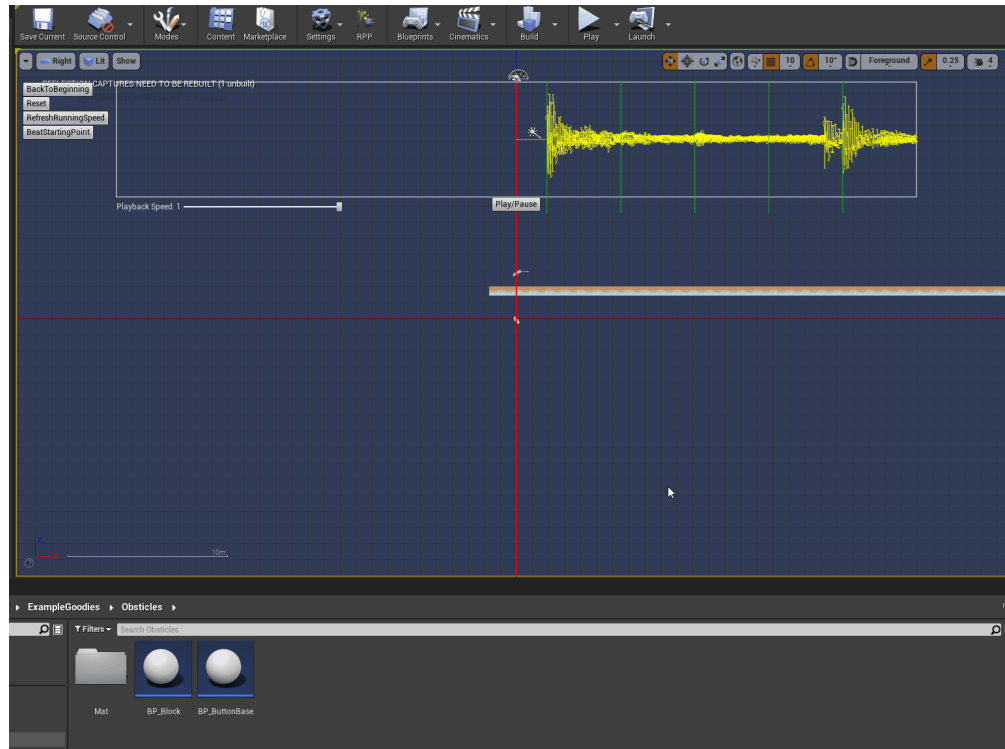
2. Follow steps in “Setup Variables”
 - a. Use “BWV578” as the sound track
 - b. Set Running Speed to 600
 - c. Set BPM to 78
 - d. Enable plugin by pressing



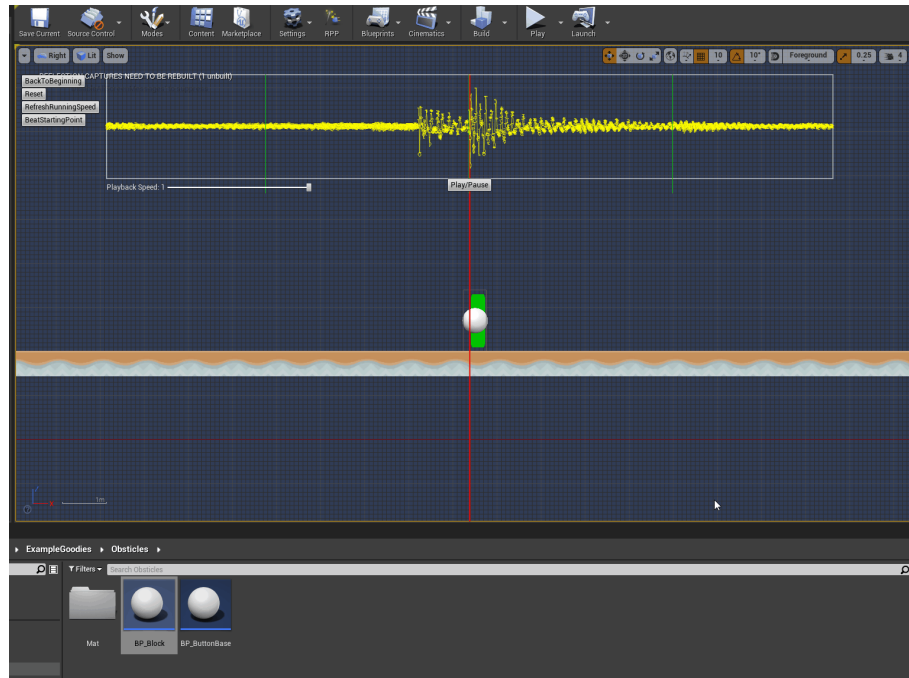
- e. To set Beat Starting Time, zoom in to appropriate level and move cursor to where the sound wave starts, then press BestStartingTime.



3. Now that the plugin is setup, we can start putting down obstacles.
 - a. Before placing obstacles, we need running ground that are long enough for the player to run at. In the example, i use DirtGround (RPPContent/ExampleGoodies/Floor)
 - b. Next, I would first listen to the music and find an appropriate place to place an obstacle. Say i want to put down an obstacle on the second drum beat.



- c. After, I decided to place another obstacle at the third beat



- d. And so on...
4. After designing the level, make sure the gamemode etc. are set
 - a. Set game mode to BP_RPPGameModeBase (RPP Content/Blueprint)
 - b. Set default pawn class to BP_RPPPlayerChatacter (RPP Content/ExampleGoodies/PlayerController)
 - c. Set HUD Class to BP_RPPHUD (RPP Content/ExampleGoodies/HUD/)
5. You can now press Play to test the level.

ExampleLevel Gameplay

In the example level, to break a block, the player has to match the color and hit Action (bind to right mouse button at default) at the correct time.