Version 1.0

2024/25

University of Staffordshire

FlowState

Technical Design Document

Rajat Rajendra Prabhu

22018309

Contents

Project Introduction

FlowState is a fast-paced parkour focused game that emphasises on the player's freedom to continue onto the level. The game would challenge the player to approach each obstacle dynamically, allowing them to traverse through multiple choices. With a focus on seamless controls and responsive mechanics, FlowState delivers a high-energy traversal experience.

Project Goals

- *Dynamic Parkour System for Fluid Movement*
- *Mechanics such as wall running, wall jumping, ledge climbing and mantling.*
- *Hacking system for the player to manipulate the level where required.*
- *Visual and Audio Feedback*
- *Minute environmental design to support the systems*
- *Complete Game Loop*

Challenges and Risks

Since Motion Matching is a newly introduced system in Unreal Engine 5.4, there is limited documentation on how it is functioned. Integrating it would require extra testing to achieve a smooth and responsive character behaviour. Making sure that the animations match with the player's state and their surroundings properly without jittering or feeling out of place is crucial for maintaining immersion.

Hardware Requirements

MINIMUM

Operating System: Windows 10 64-bit
Processor: Quad-core Intel or AMD 2.5GHz or superior
Memory: 8GB RAM
Graphics Card: Any DirectX 11 or 12 compatible card

RECOMMENDED

Operating System: Windows 10 64-bit (Version 20H2)
Processor: Six-Core Xeon E5-2643 @ 3.4GHz
Memory: 16GB RAM
Graphics Card: NVIDIA GeForce RTX 2080 SUPER

Platforms

Target Platform

The Project is specifically targeted for Consoles and PC.

Engine Specific Specifications and Limitations

The game should run on a smooth 60 frames per second which would require the textures to not be so highly detailed and cause a drop in frames. The file size of textures should not exceed 1024 x 1024 unless absolutely necessary for UI or close-up assets. Level streaming should be used to manage assets efficiently and prevent memory overload. Complex collisions will be only used where necessary, whose alternative would be to use simplified collision meshes. As the project is also targeted to consoles(particularly ones with a gamepad/controller), the number of inputs used will be significantly less due to a smaller number of input binds present in a controller.

Engine Summary

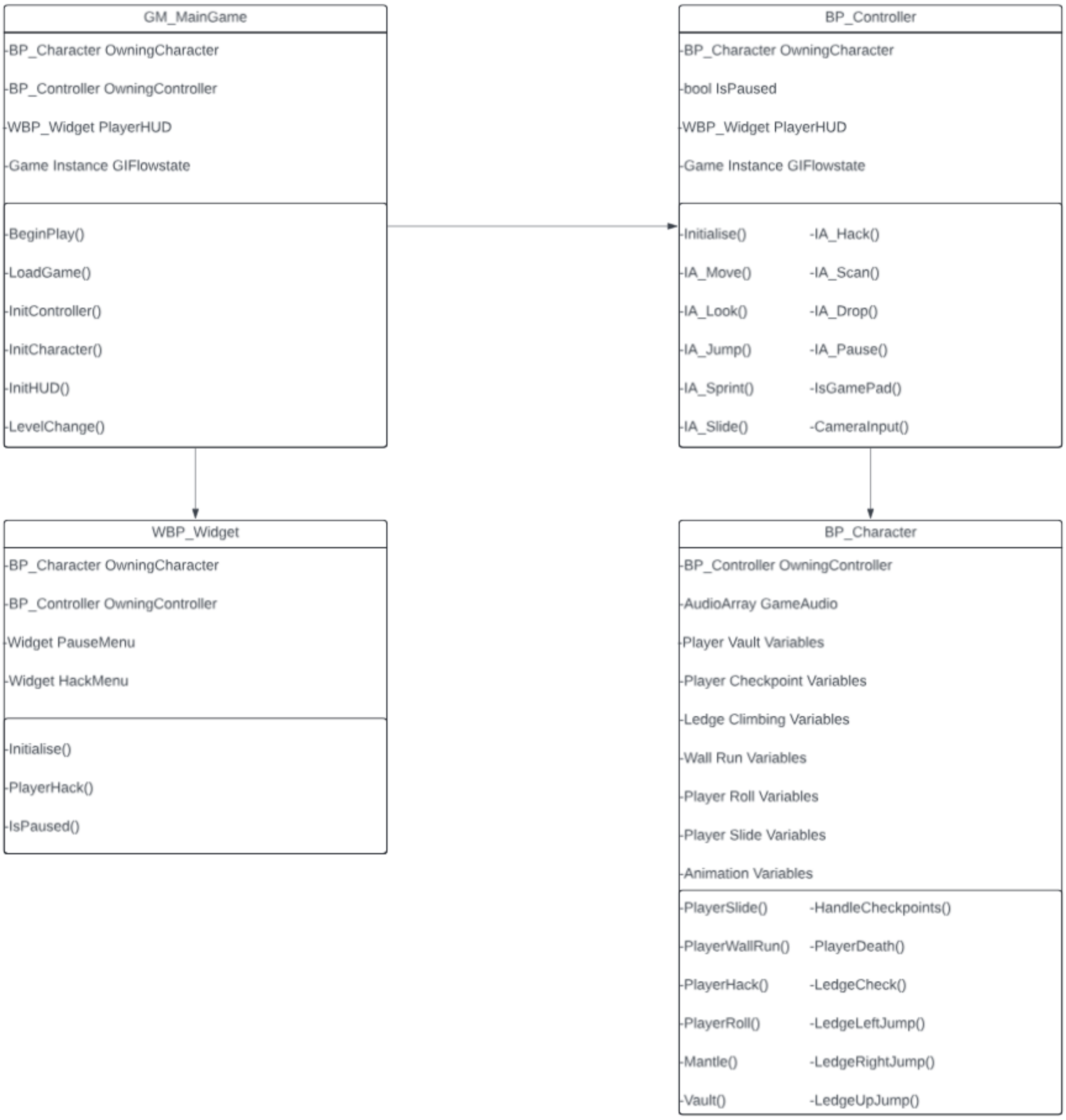
The project is built using Unreal Engine 5.4. Due to the use of the new Motion Matching system, the following plugins were enabled:

- Pose Search
- Chooser
- Animation Insights
- Animation Warping
- Motion Trajectory
- ImageResizUEr

Systems and Diagrams

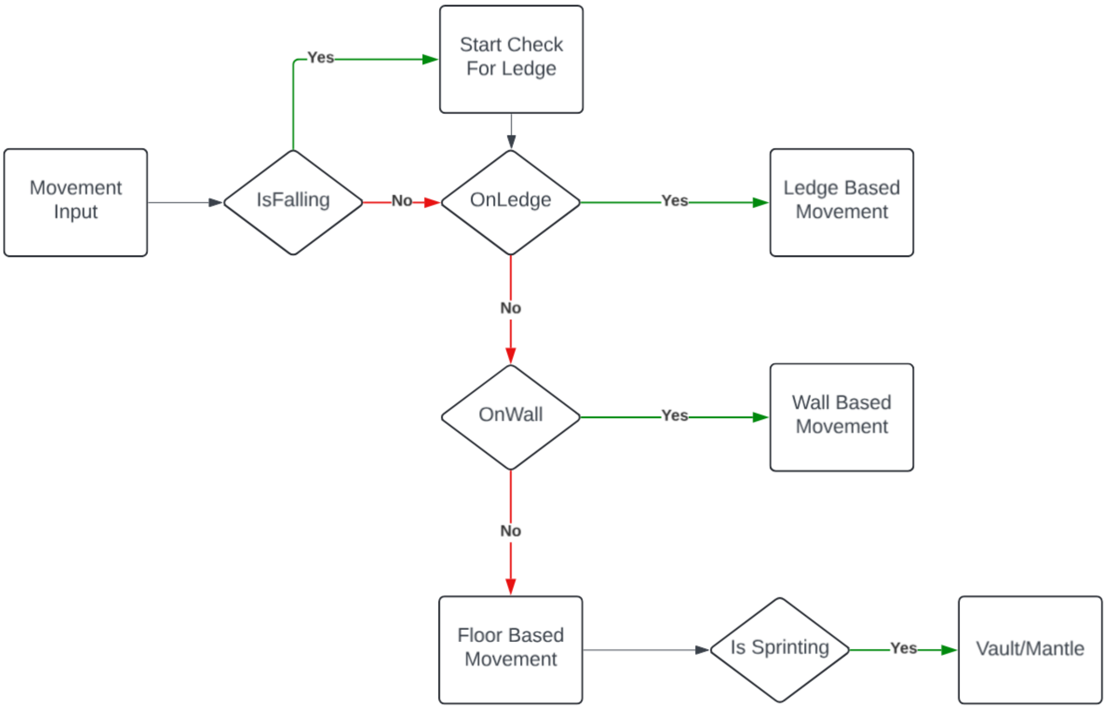
Base Character

Class Diagrams



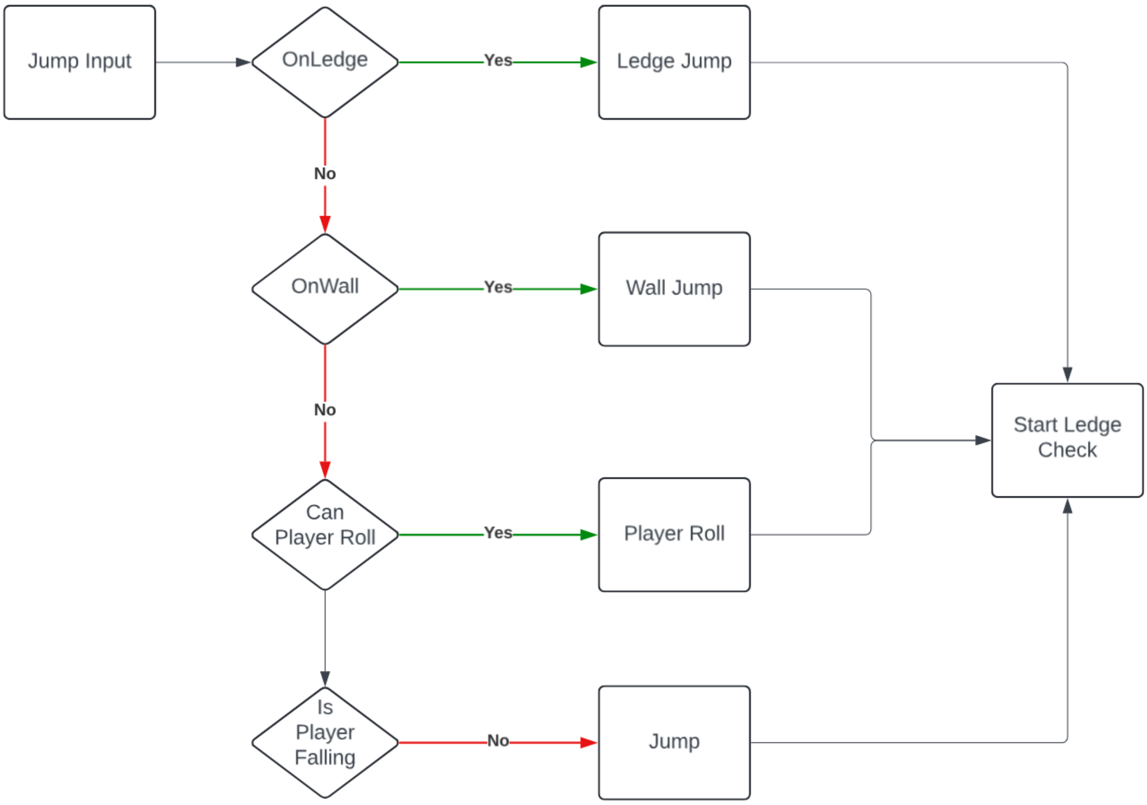
Player Movement

Talking about the different states the player may be in when trying to move:



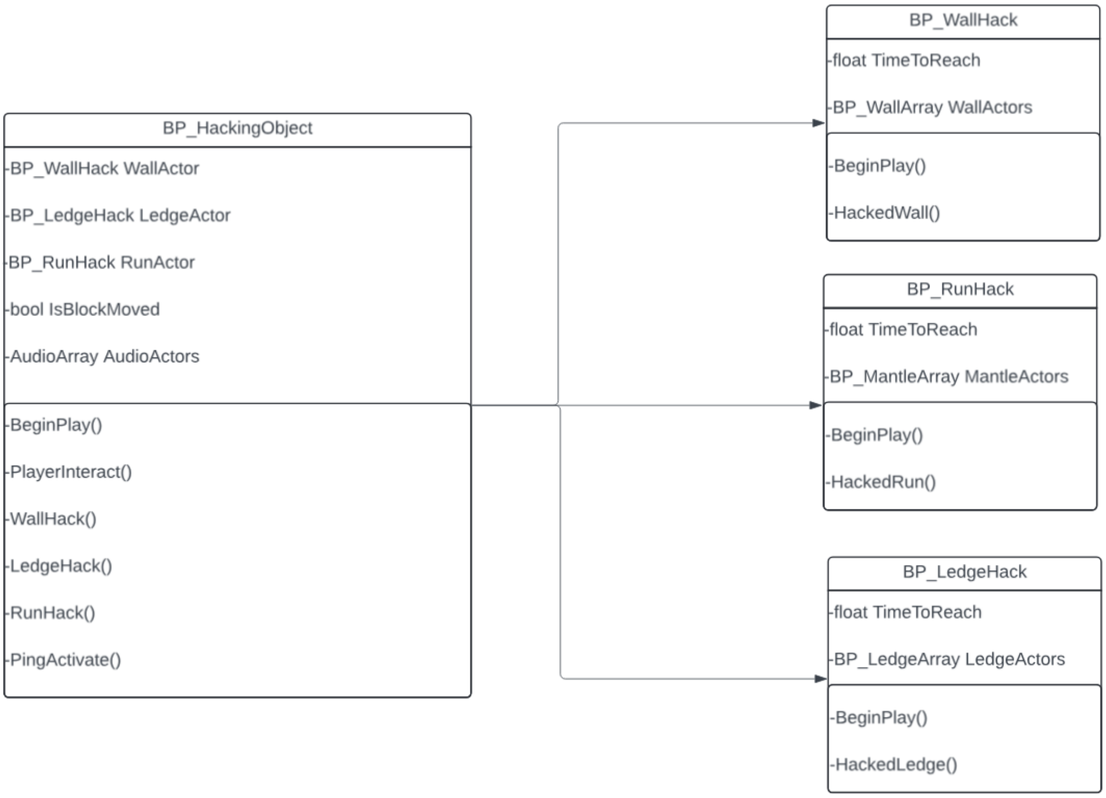
Player Jump

Talking about the different states the player may be in when trying to jump:



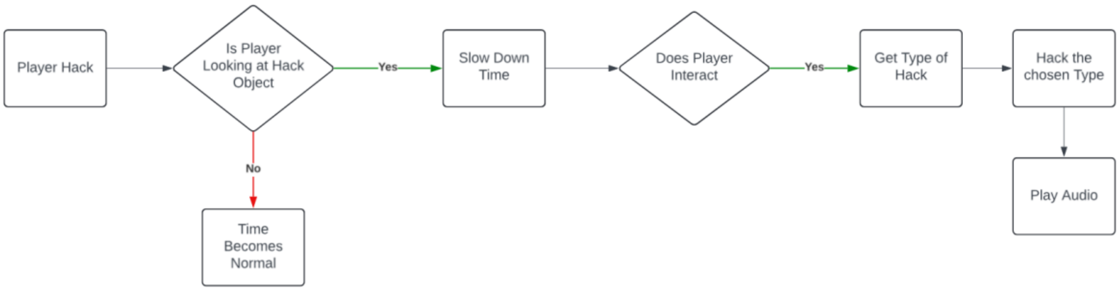
Player Hack

Class Diagram



Hack Input

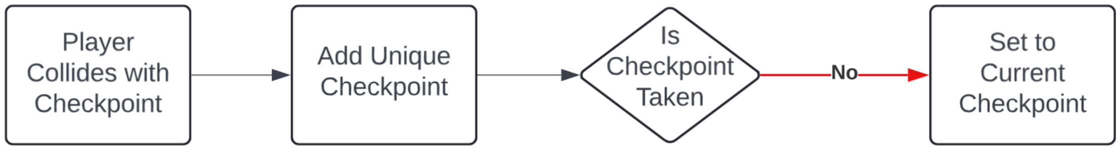
How the player will start interacting with the hacking actor to determine the path:



Player Respawning

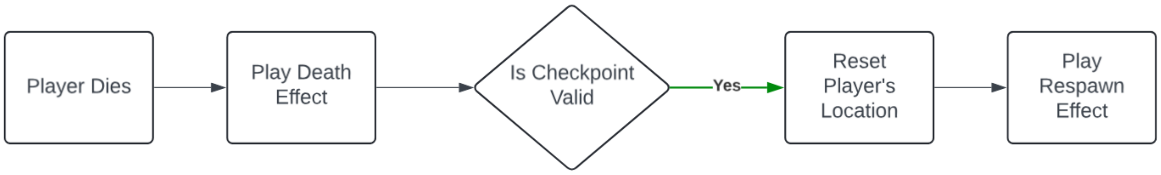
Checkpoint Handling

Logic whenever player collides with a checkpoint:



Player Death

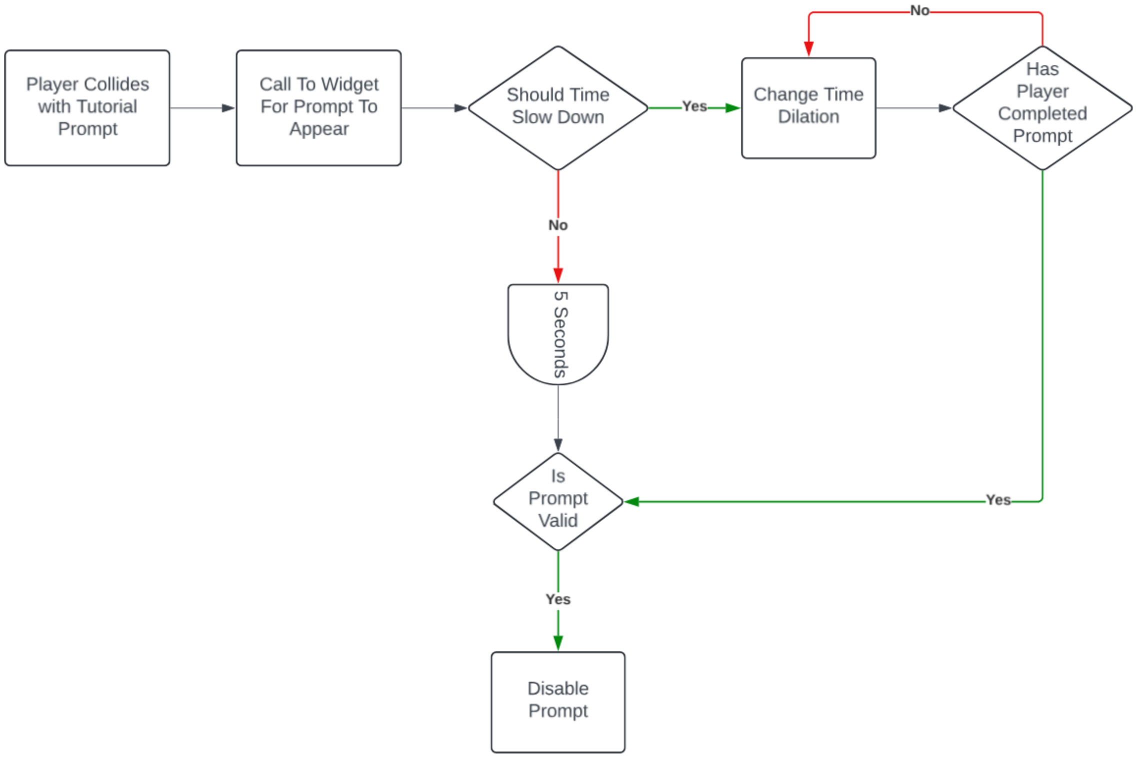
The player will have a death sequence and will spawn at the checkpoint:



Tutorial Level

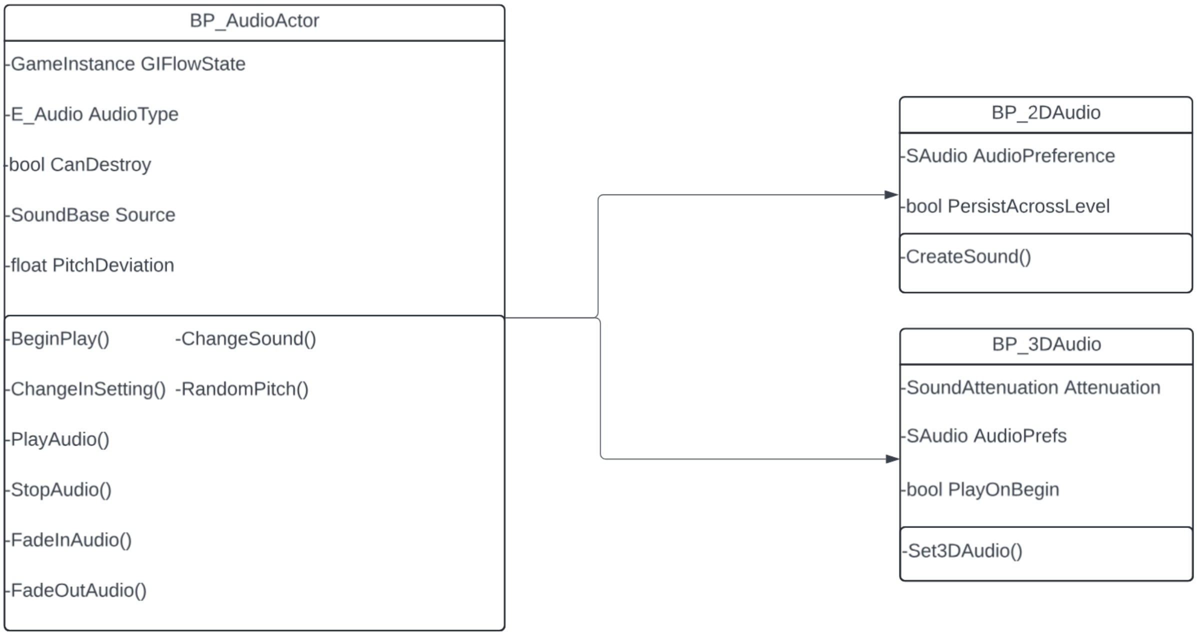
Prompt Hit

How the prompt will work once the player interacts/collides with it:


Player Collides with Tutorial Prompt
Call To Widget For Prompt To Appear
Should Time Slow Down
Yes
Change Time Dilation
Has Player Completed Prompt
No
No
5 Seconds
Yes
Is Prompt Valid
Yes
Disable Prompt

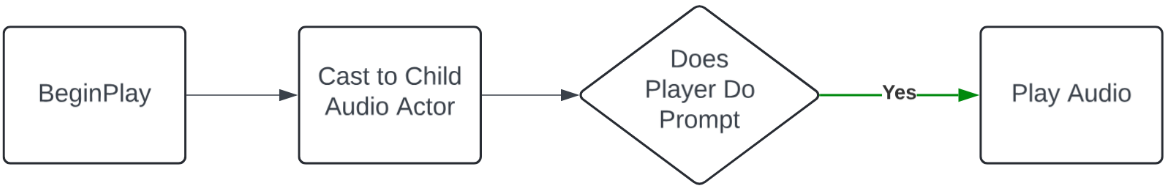
Audio Actor

Class Diagrams



Play Audio

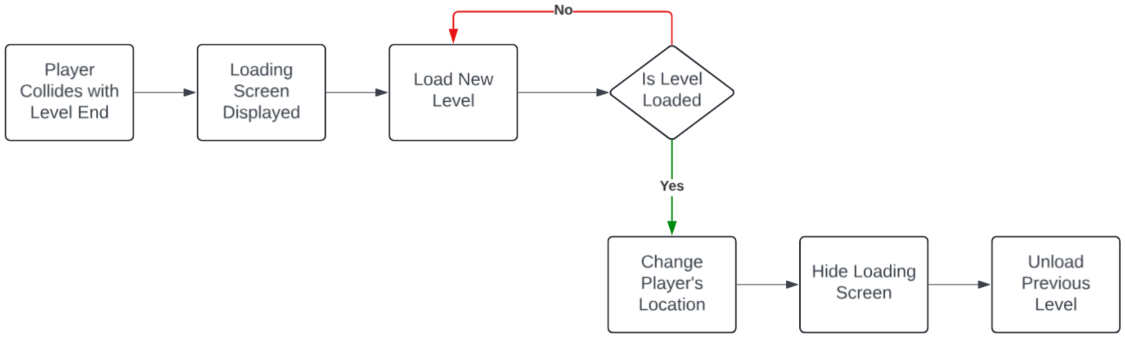
How Audio will be played in the game when needed:



Level Streaming

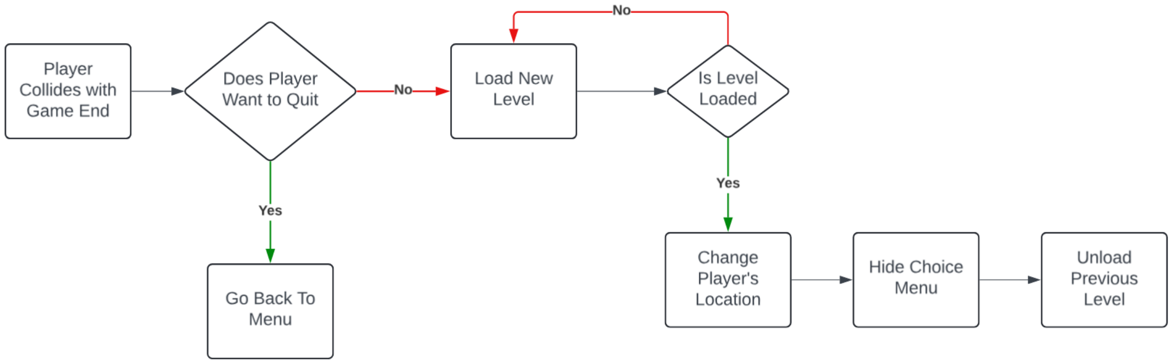
Loading Levels

To minimise loading times, levels will be loaded beforehand to keep the game's flow consistent:



Ending Game

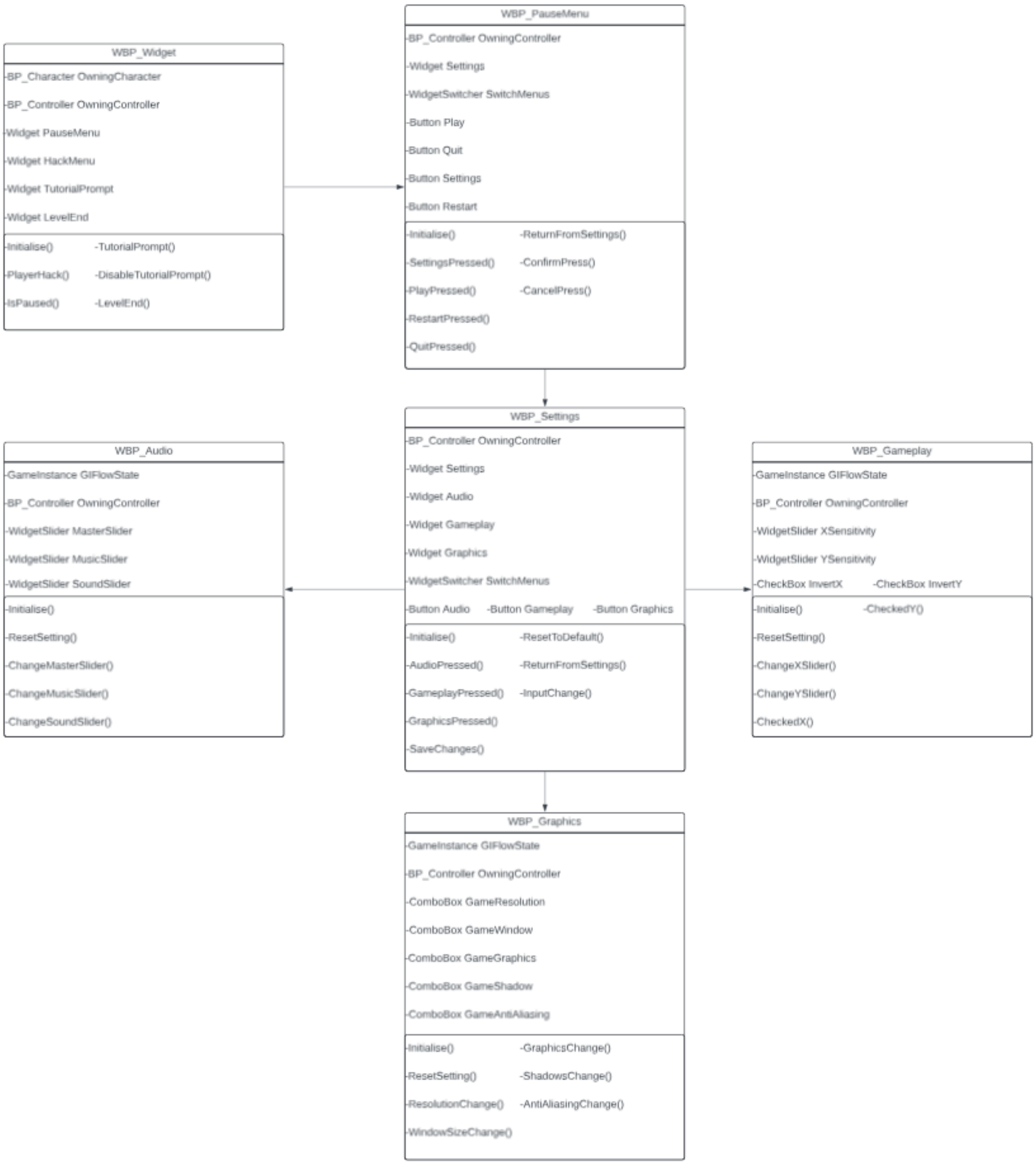
Once the player collides with the game end box, they might be given a choice to either restart the game or go back to the menu:



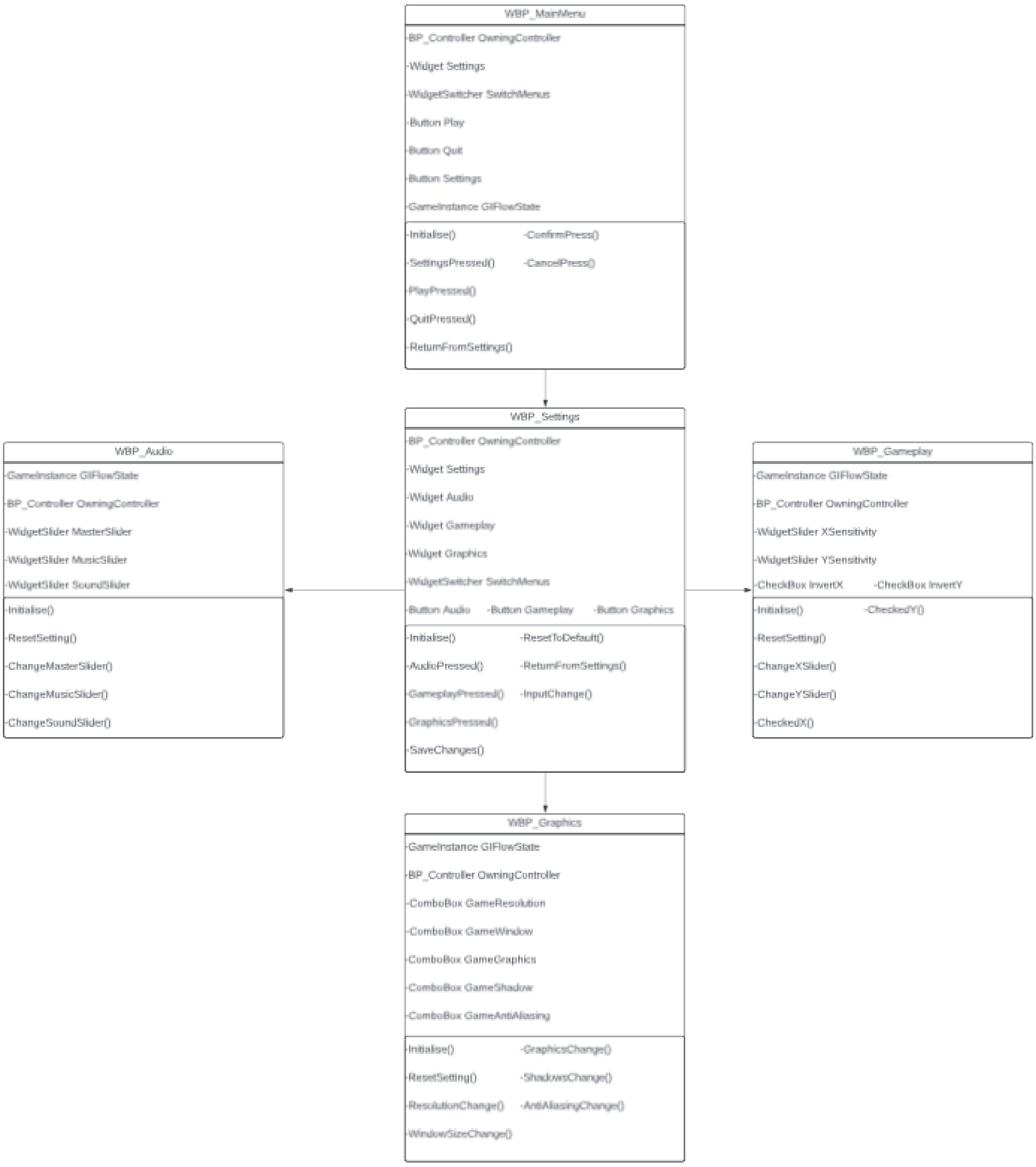
Widgets

Class Diagrams

Main HUD



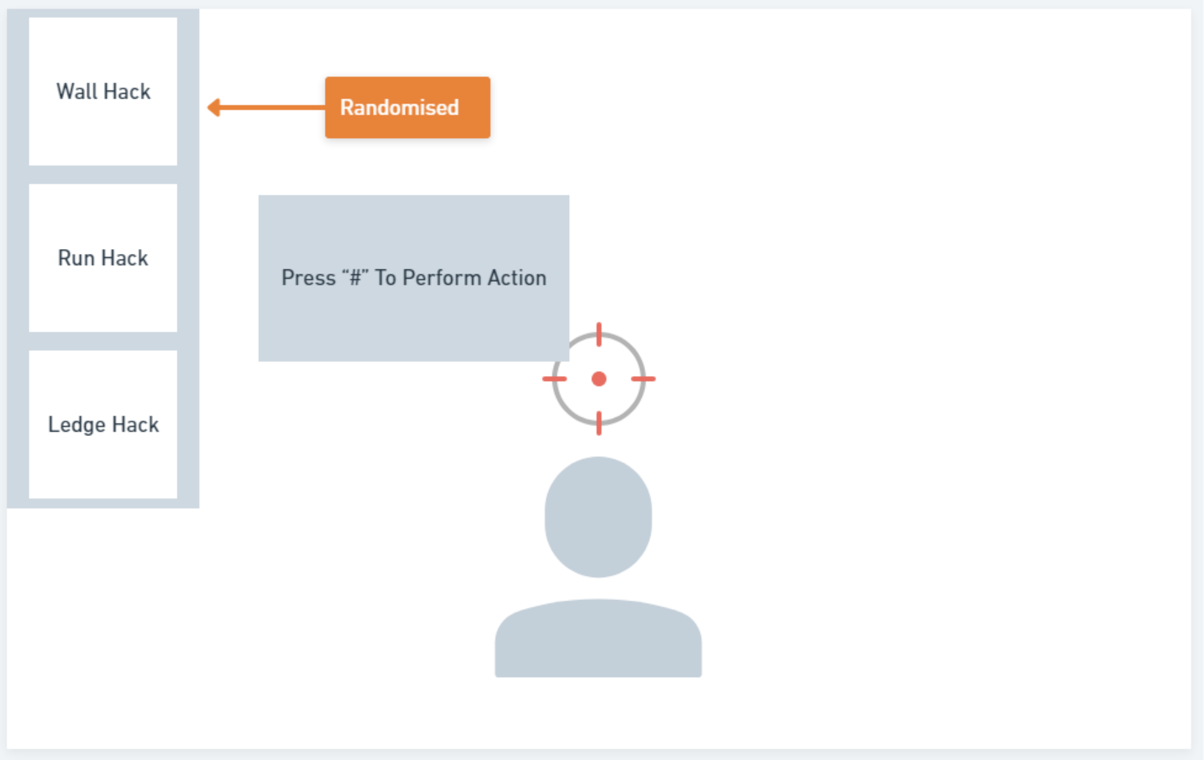
MainMenu


WBP_MainMenu
-BP_Controller OwningController
-Widget Settings
-WidgetSwitcher SwitchMenus
-Button Play
-Button Quit
-Button Settings
-GameInstance GIFlowState
-Initialise()
-ConfirmPress()
-SettingsPressed()
-CancelPress()
-PlayPressed()
-QuitPressed()
-ReturnFromSettings()
WBP_Audio
-GameInstance GIFlowState
-BP_Controller OwningController
-WidgetSlider MasterSlider
-WidgetSlider MusicSlider
-WidgetSlider SoundSlider
-Initialise()
-ResetSetting()
-ChangeMasterSlider()
-ChangeMusicSlider()
-ChangeSoundSlider()
WBP_Settings
-BP_Controller OwningController
-Widget Settings
-Widget Audio
-Widget Gameplay
-Widget Graphics
-WidgetSwitcher SwitchMenus
-Button Audio
-Button Gameplay
-Button Graphics
-Initialise()
-ResetToDefault()
-AudioPressed()
-ReturnFromSettings()
-GameplayPressed()
-InputChange()
-GraphicsPressed()
-SaveChanges()
WBP_Gameplay
-GameInstance GIFlowState
-BP_Controller OwningController
-WidgetSlider XSensitivity
-WidgetSlider YSensitivity
-CheckBox InvertX
-CheckBox InvertY
-Initialise()
-CheckedY()
-ResetSetting()
-ChangeXSlider()
-ChangeYSlider()
-CheckedX()
WBP_Graphics
-GameInstance GIFlowState
-BP_Controller OwningController
-ComboBox GameResolution
-ComboBox GameWindow
-ComboBox GameGraphics
-ComboBox GameShadow
-ComboBox GameAntiAliasing
-Initialise()
-GraphicsChange()
-ResetSetting()
-ShadowsChange()
-ResolutionChange()
-AntiAliasingChange()
-WindowSizeChange()

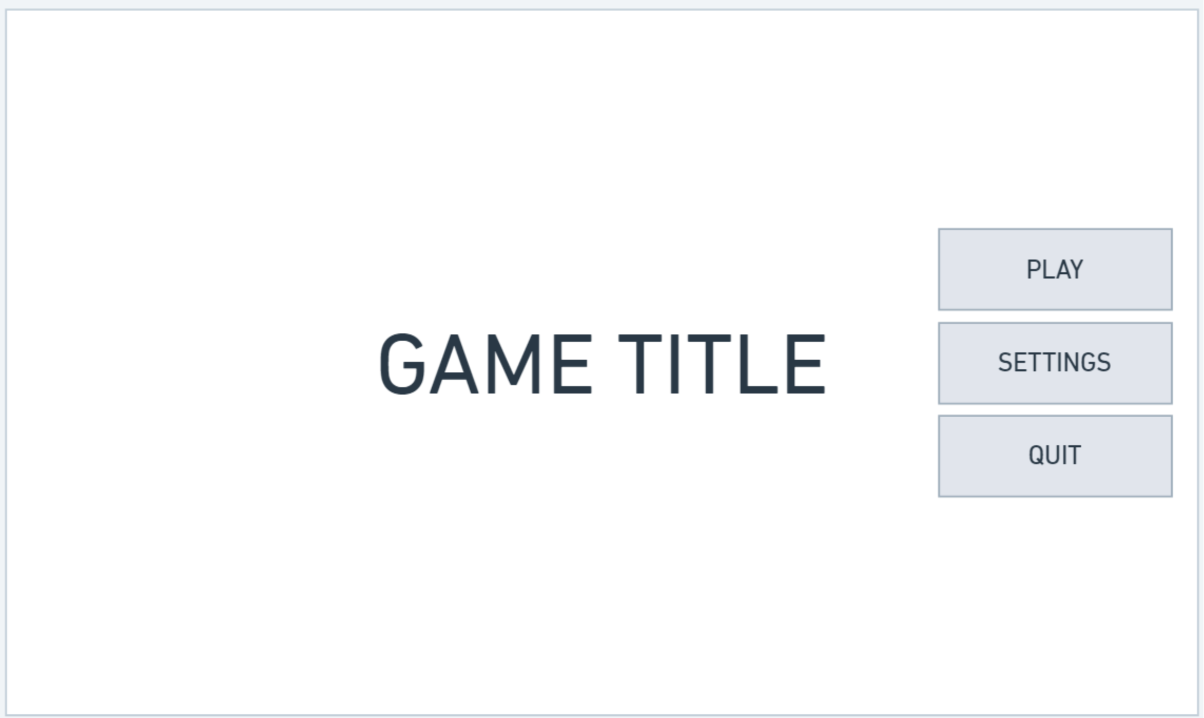
UI WireFrames

Here are some of the potential wireframes that will be present in the game:

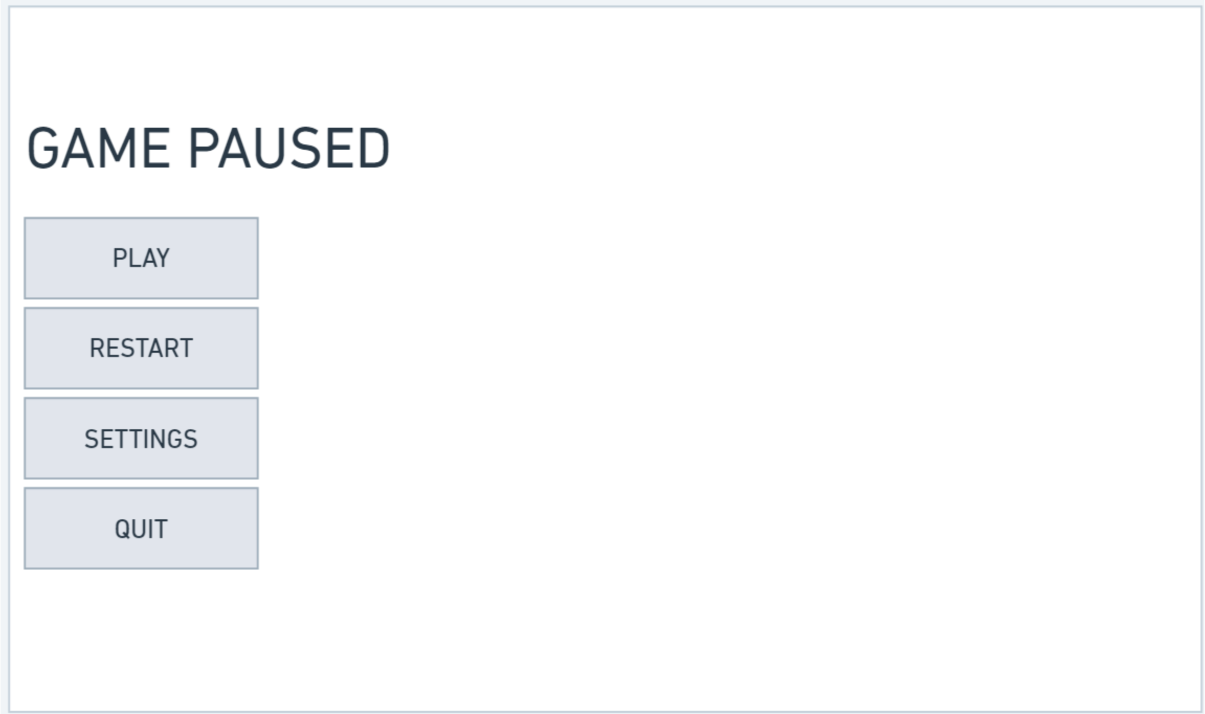
Main HUD



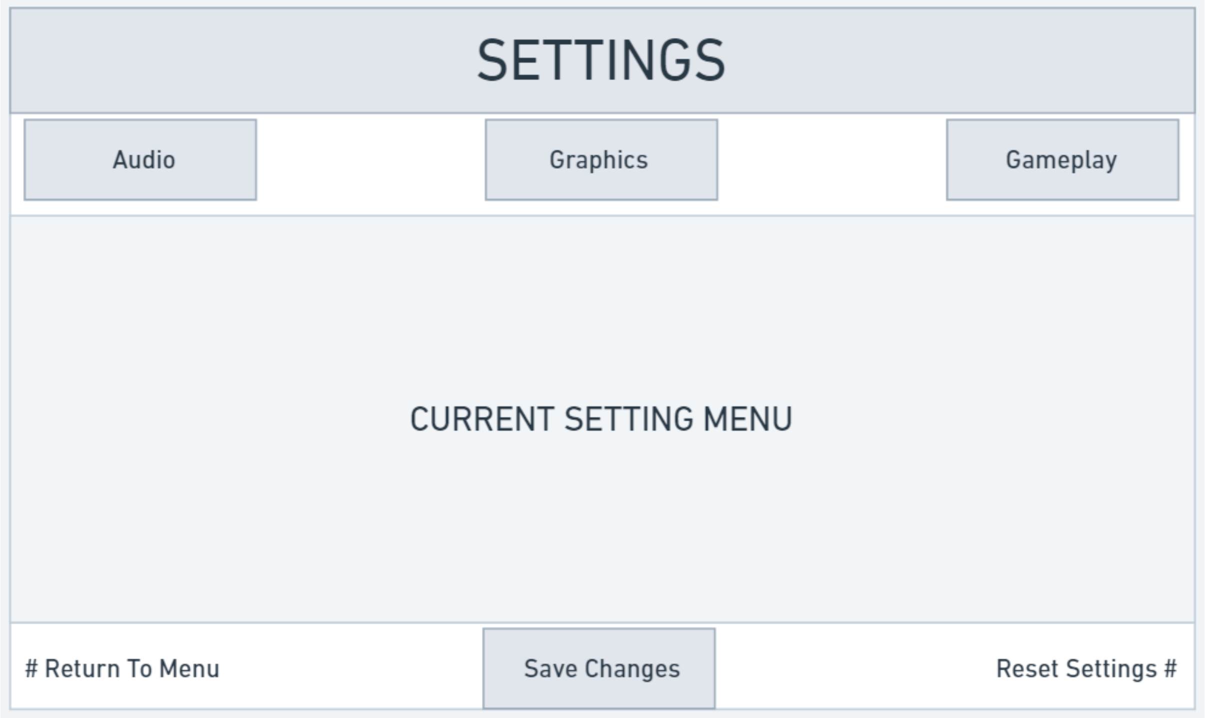
Main Menu



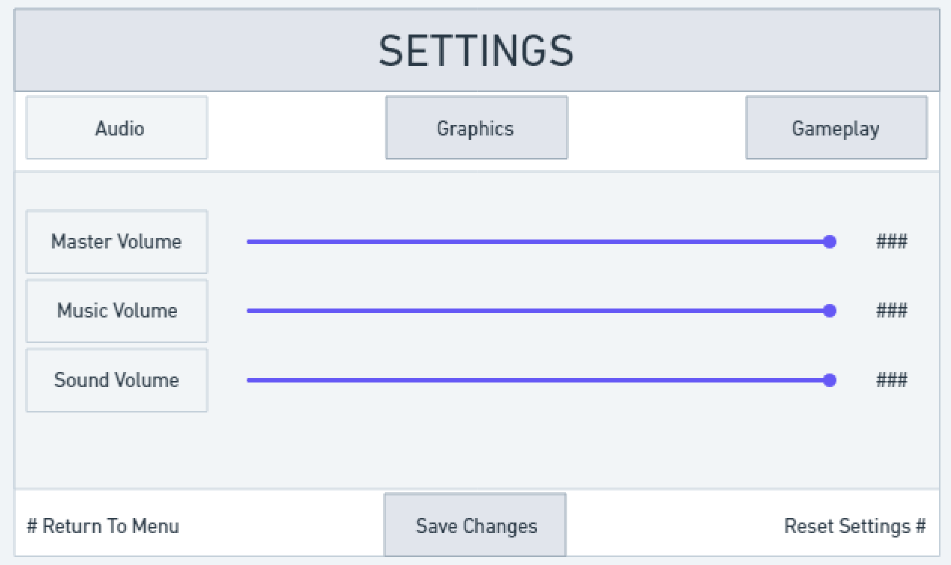
Pause Menu



Settings Menu



Audio Settings

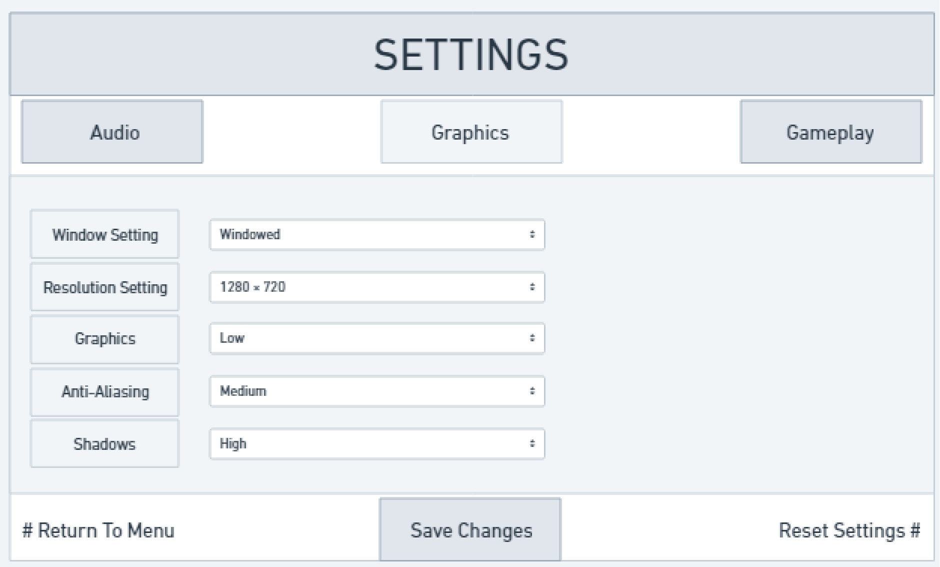


Master Volume ###

Music Volume ###

Sound Volume ###

Graphics Settings



Window Setting: Windowed

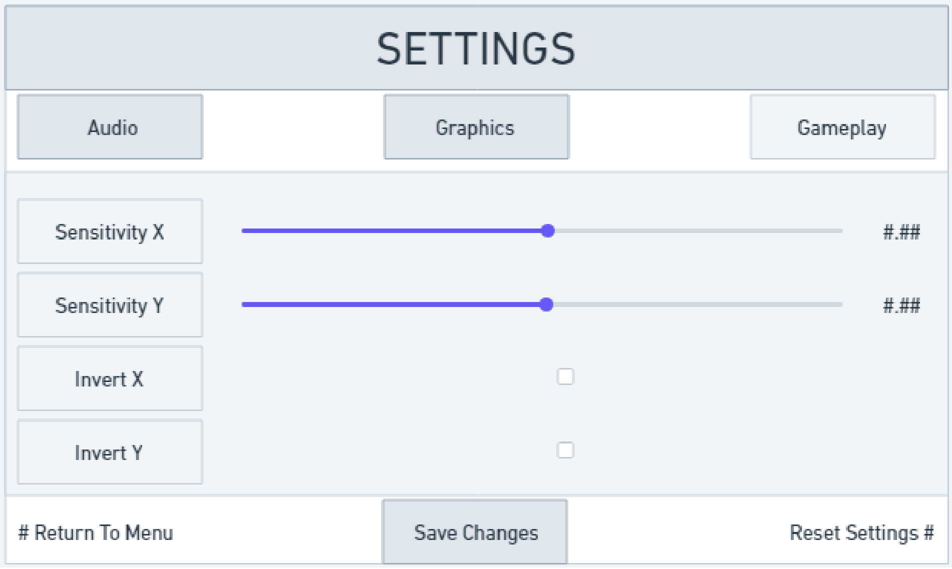
Resolution Setting: 1280 × 720

Graphics: Low

Anti-Aliasing: Medium

Shadows: High

Gameplay Settings



Sensitivity X #.##

Sensitivity Y #.##

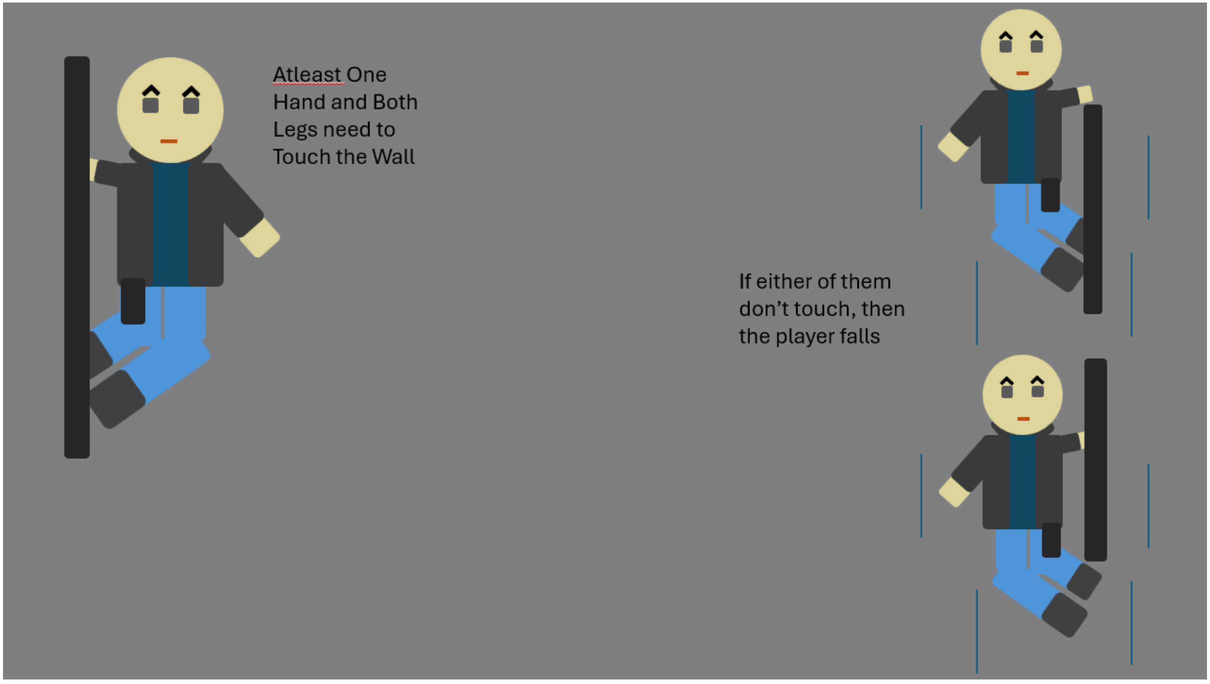
Invert X □

Invert Y □

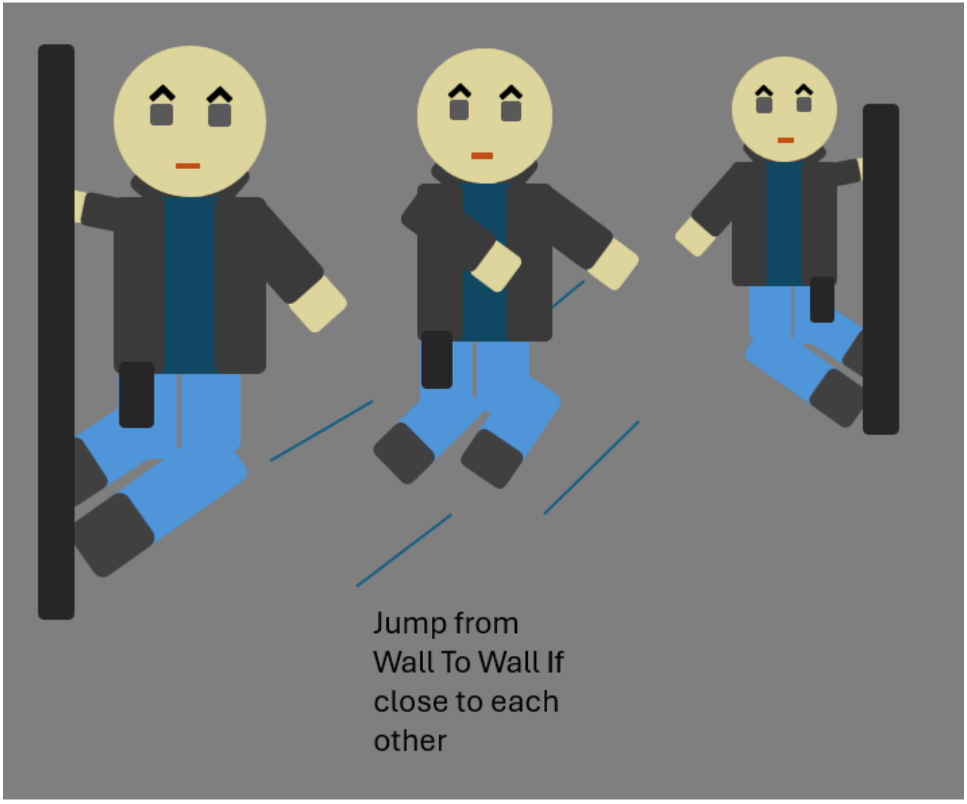
Wall Running

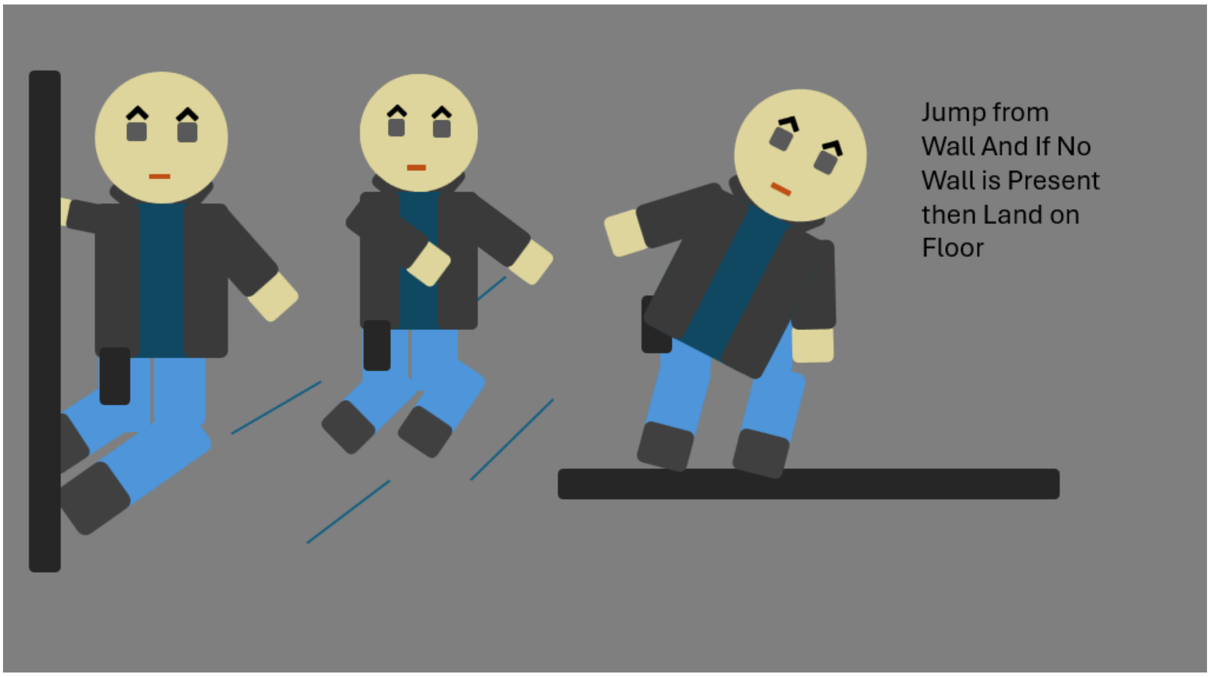
Mechanic Diagrams

This should be an accurate representation of how the wall running mechanic may work:


Atleast One Hand and Both Legs need to Touch the Wall
If either of them don’t touch, then the player falls

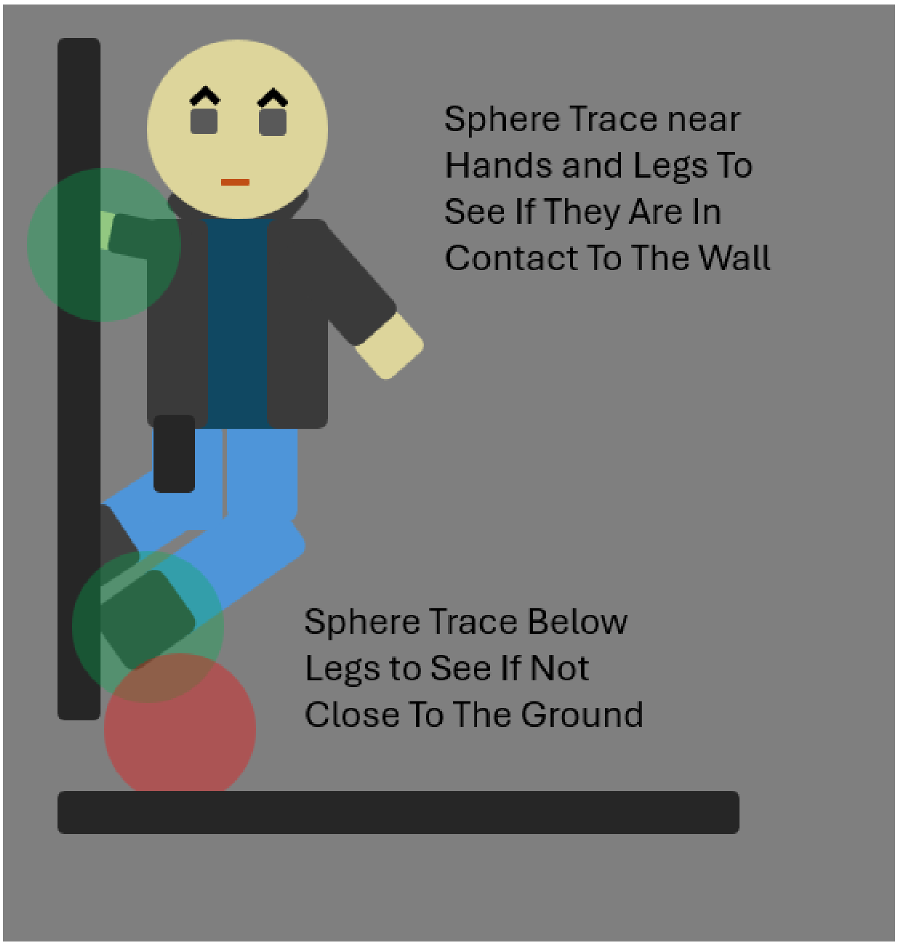
This is what the wall jump mechanic should look like:





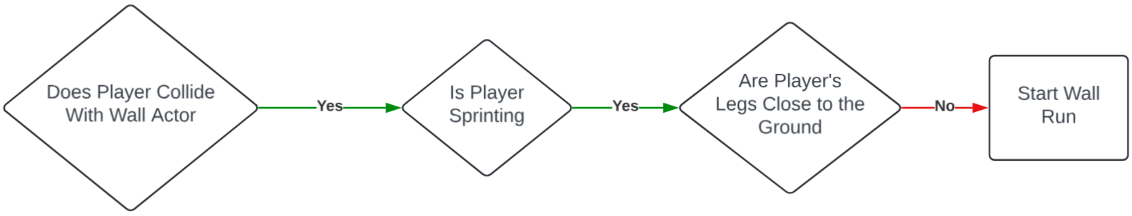
Logic Diagrams

How the player would interact with the wall once in contact with it and how the logic will be used to determine that:

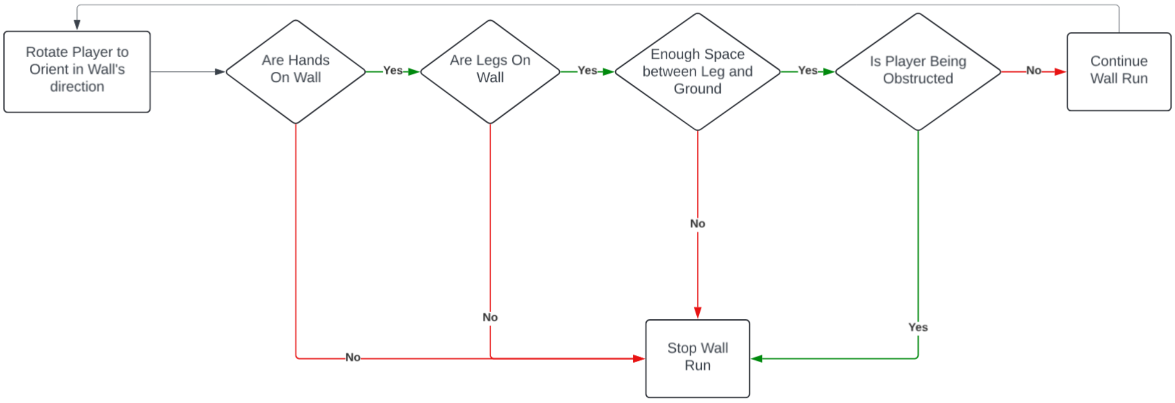


Flowcharts

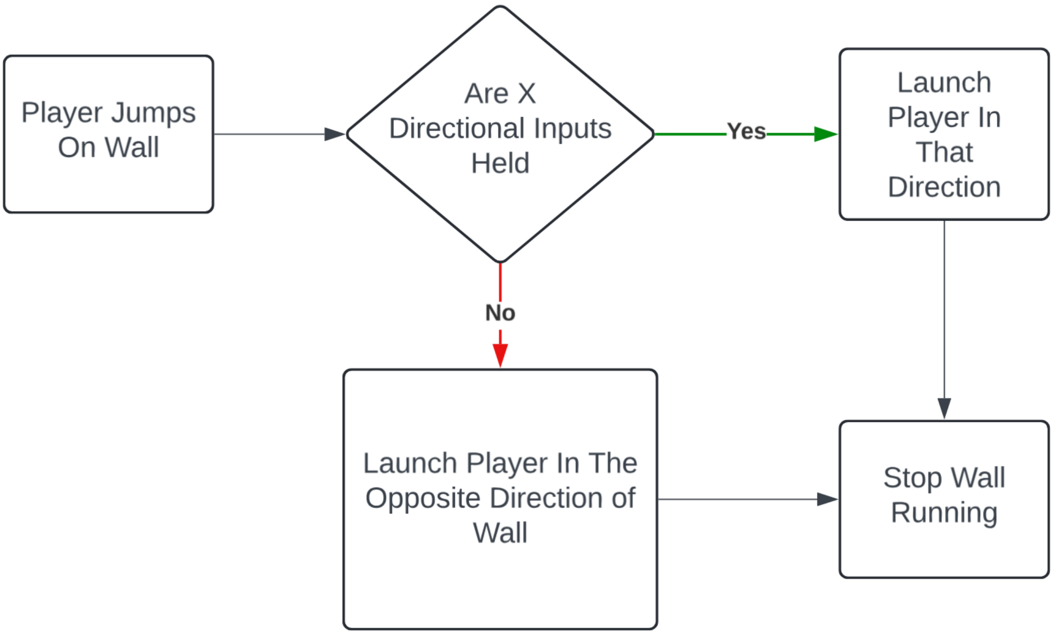
How will the wall run be detected when player comes in contact with one:


Does Player Collide With Wall Actor
Yes
Is Player Sprinting
Yes
Are Player's Legs Close to the Ground
No
Start Wall Run

How will the logic continue when the player is on the wall and going forwards:


Rotate Player to Orient in Wall's direction
Are Hands On Wall
Yes
Are Legs On Wall
Yes
Enough Space between Leg and Ground
Yes
Is Player Being Obstructed
No
Continue Wall Run
No
No
No
Yes
Stop Wall Run

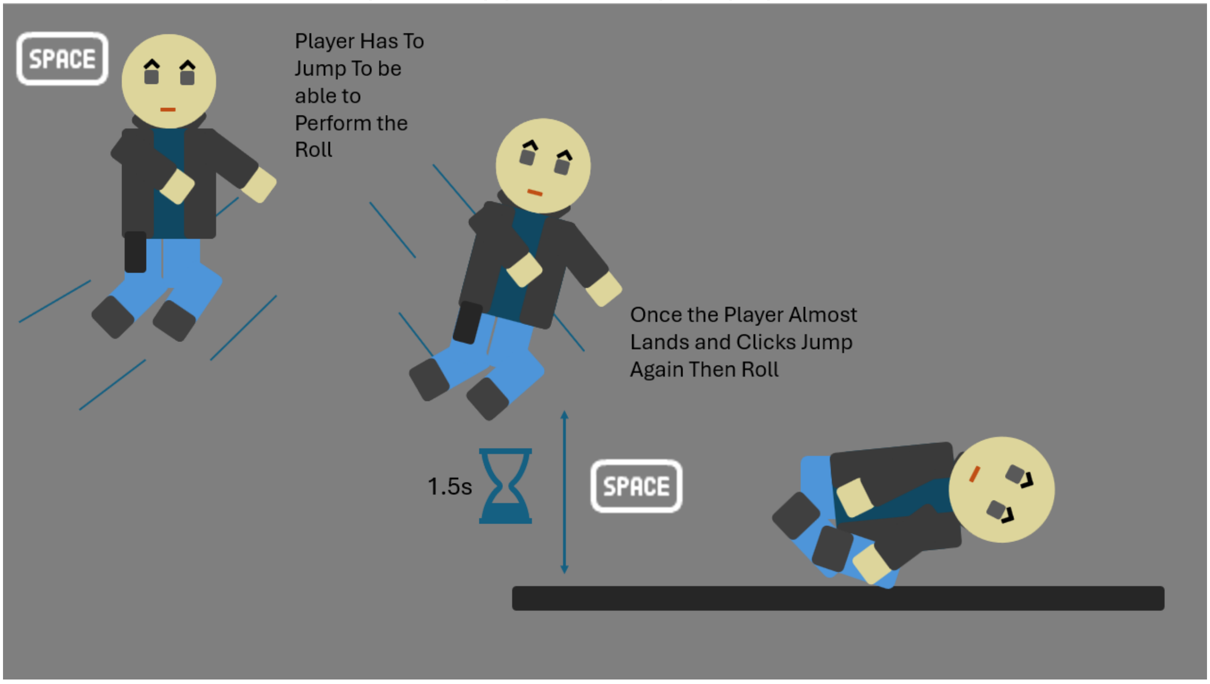
What will the logic be when the player tries to jump when on the wall:


Player Jumps On Wall
Are X Directional Inputs Held
Yes
Launch Player In That Direction
No
Launch Player In The Opposite Direction of Wall
Stop Wall Running

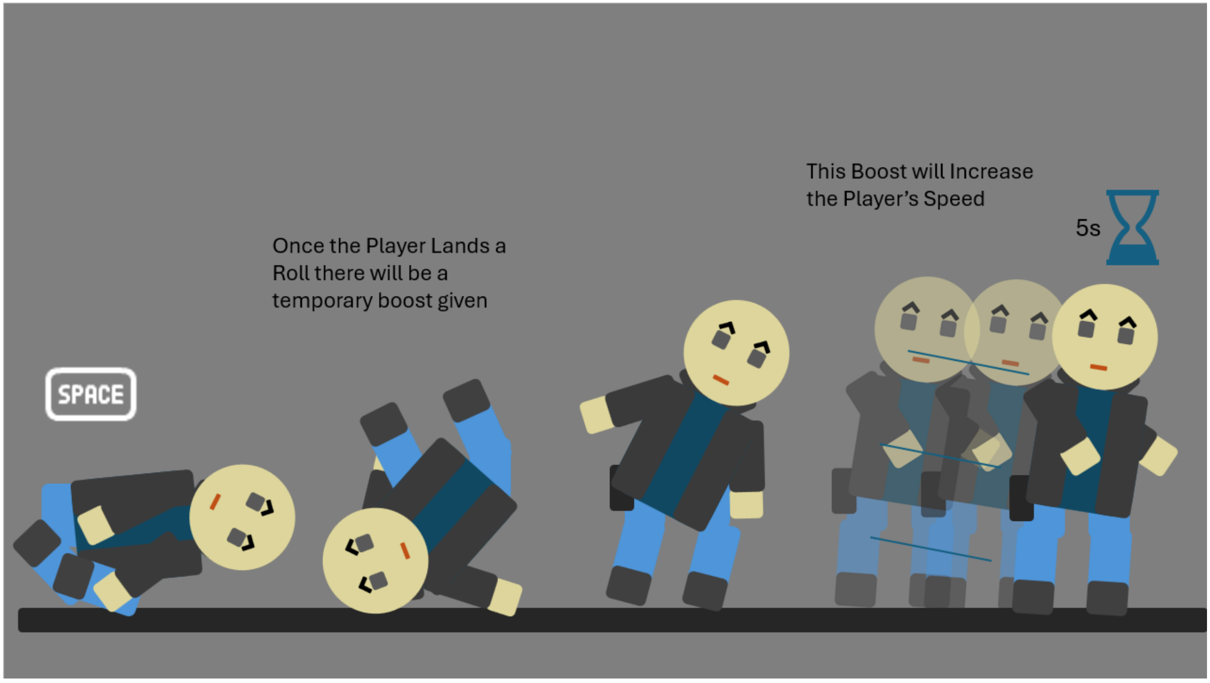
Player Roll

Mechanic Diagrams

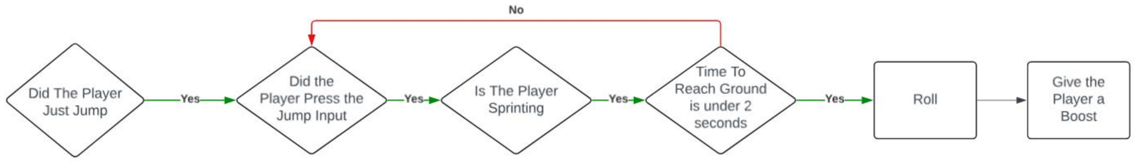
This is how the roll would be potentially performed by the player


SPACE
Player Has To Jump To be able to Perform the Roll
Once the Player Almost Lands and Clicks Jump Again Then Roll
1.5s
SPACE

The logic that will follow once the player lands a roll is as follows:


This Boost will Increase the Player's Speed
5s
Once the Player Lands a Roll there will be a temporary boost given
SPACE

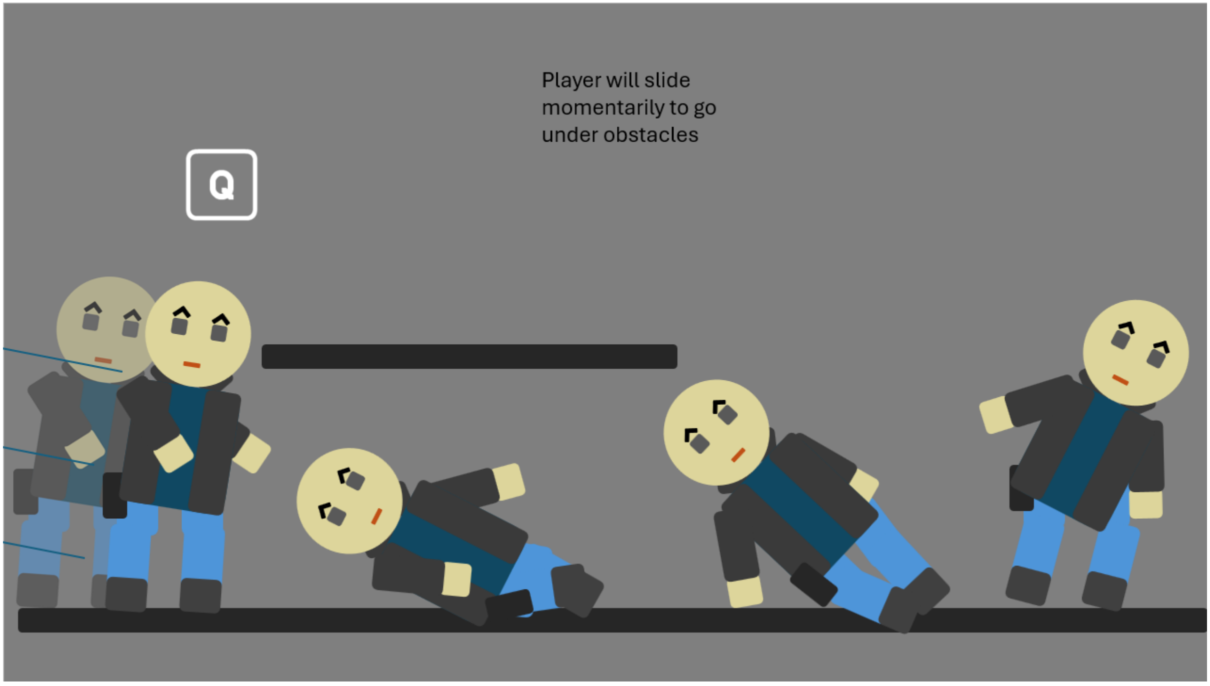
Flowcharts



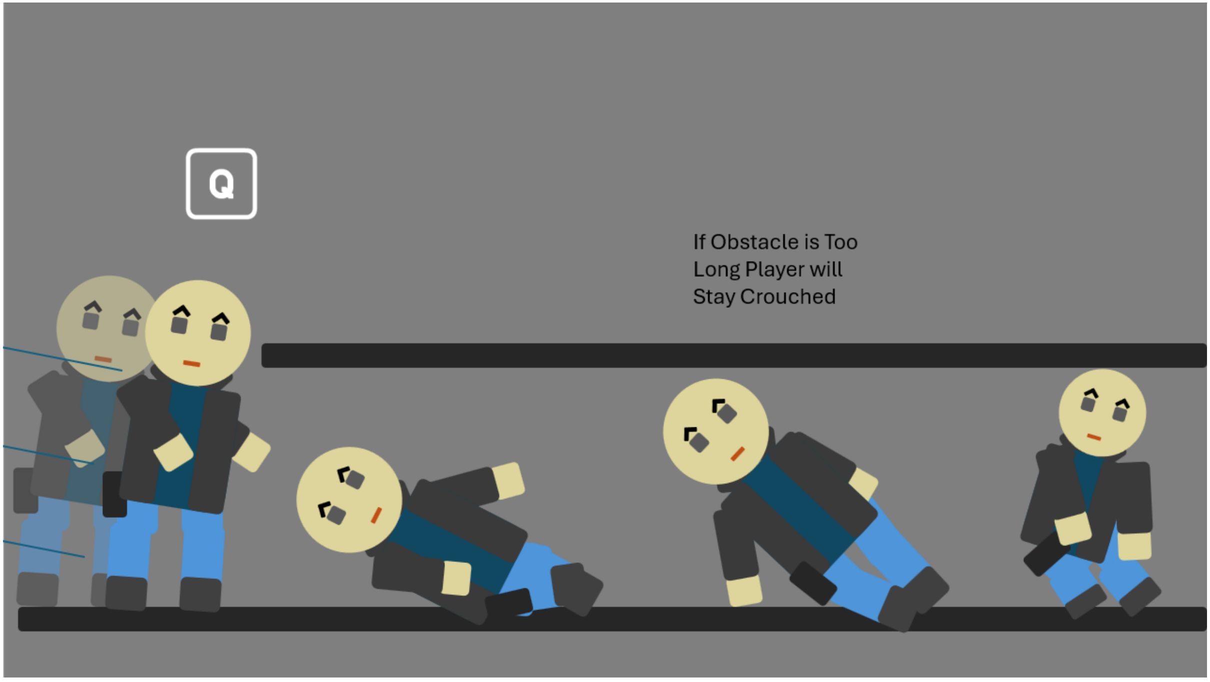
Player Slide

Mechanic Diagrams

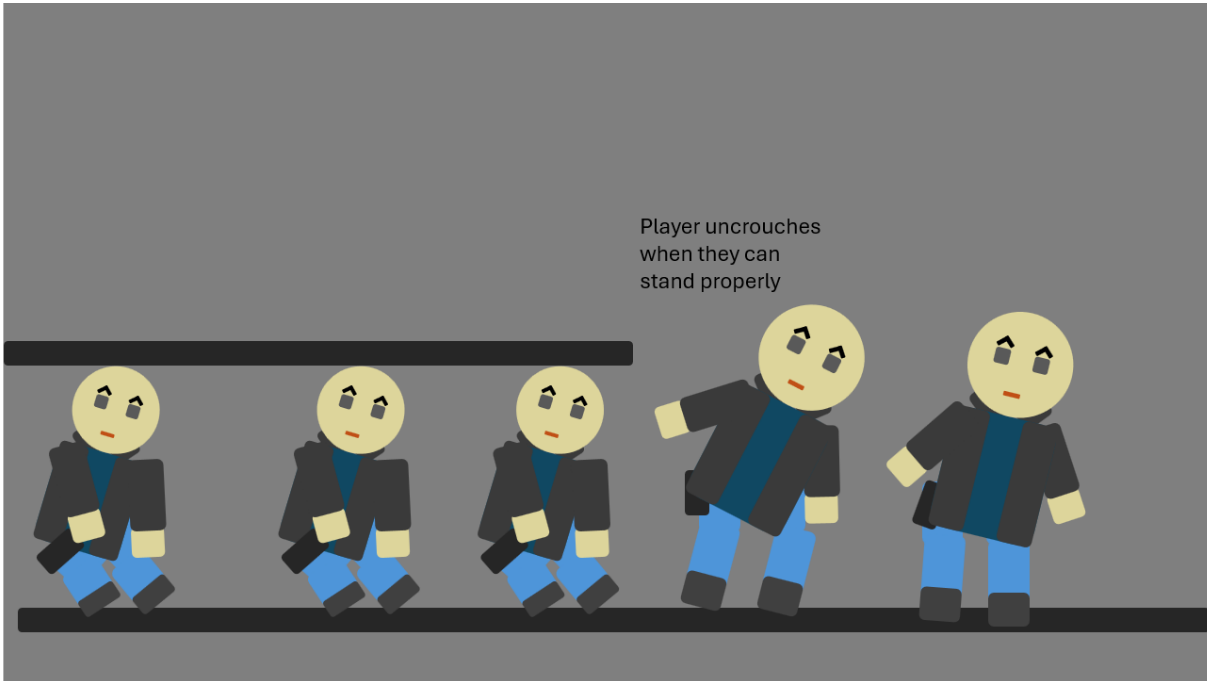
How will the Slide Logic work in game:



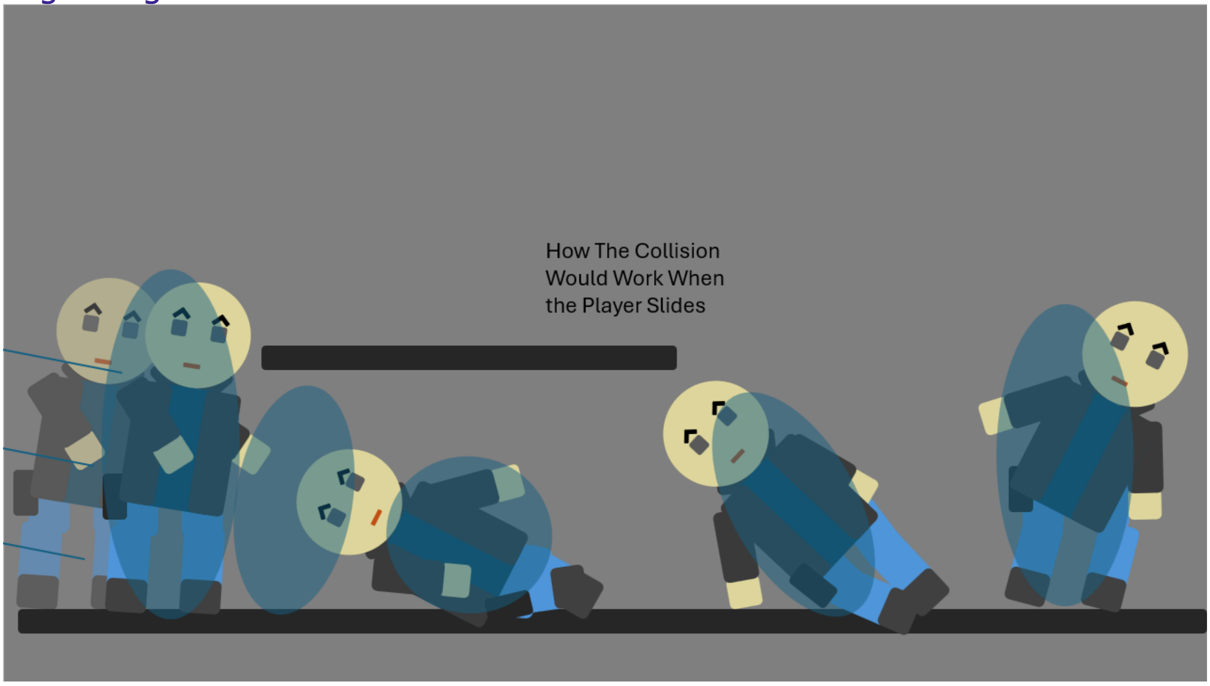
What will happen if obstacle is longer:

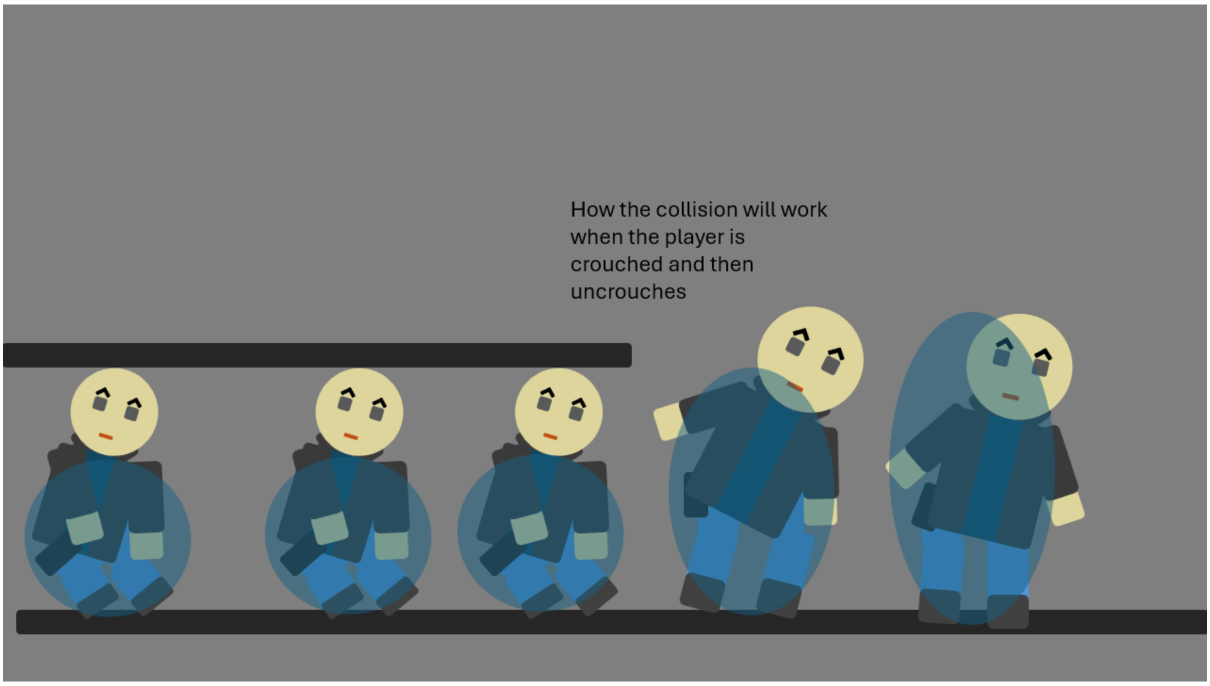


What will happen once the player has space to stand up after crouching:



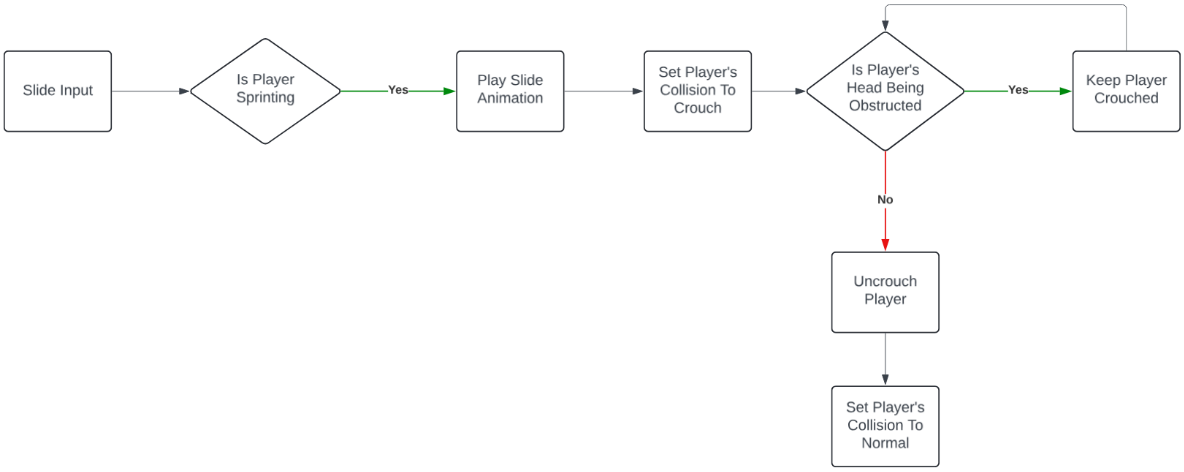
Logic Diagrams




How the collision will work when the player is crouched and then uncrouches

Flowcharts

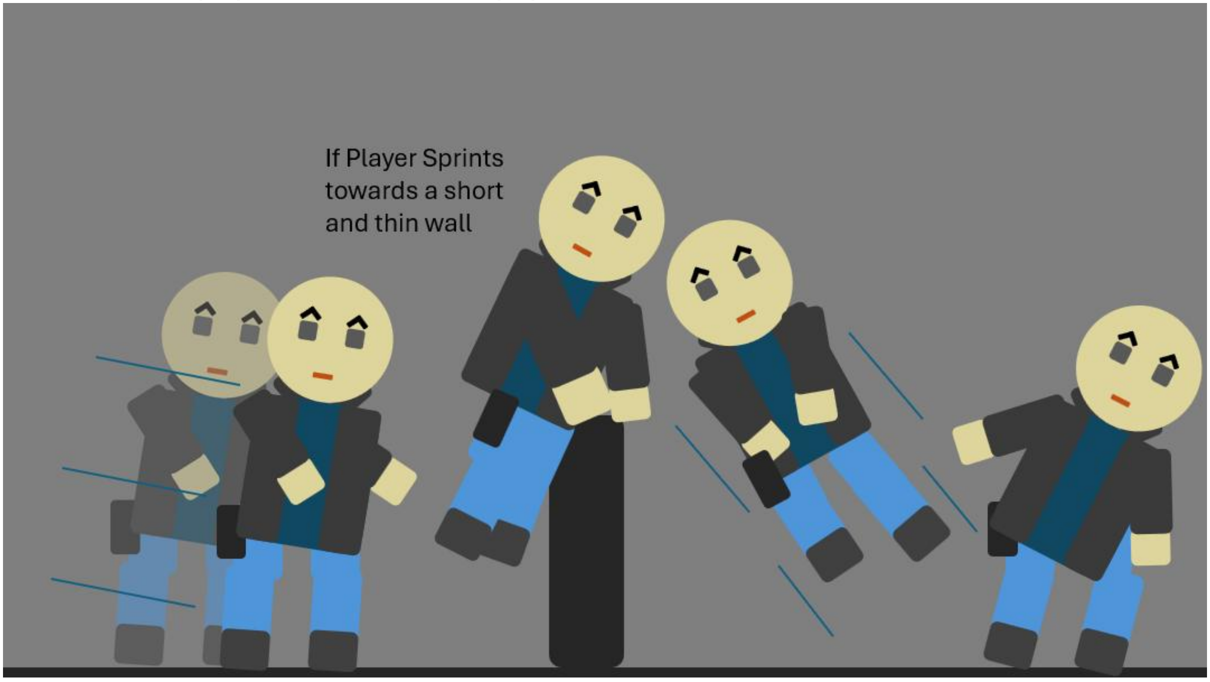
How would the logic work when the player tries to slide:


Slide Input
Is Player Sprinting
Yes
Play Slide Animation
Set Player's Collision To Crouch
Is Player's Head Being Obstructed
Yes
Keep Player Crouched
No
Uncrouch Player
Set Player's Collision To Normal

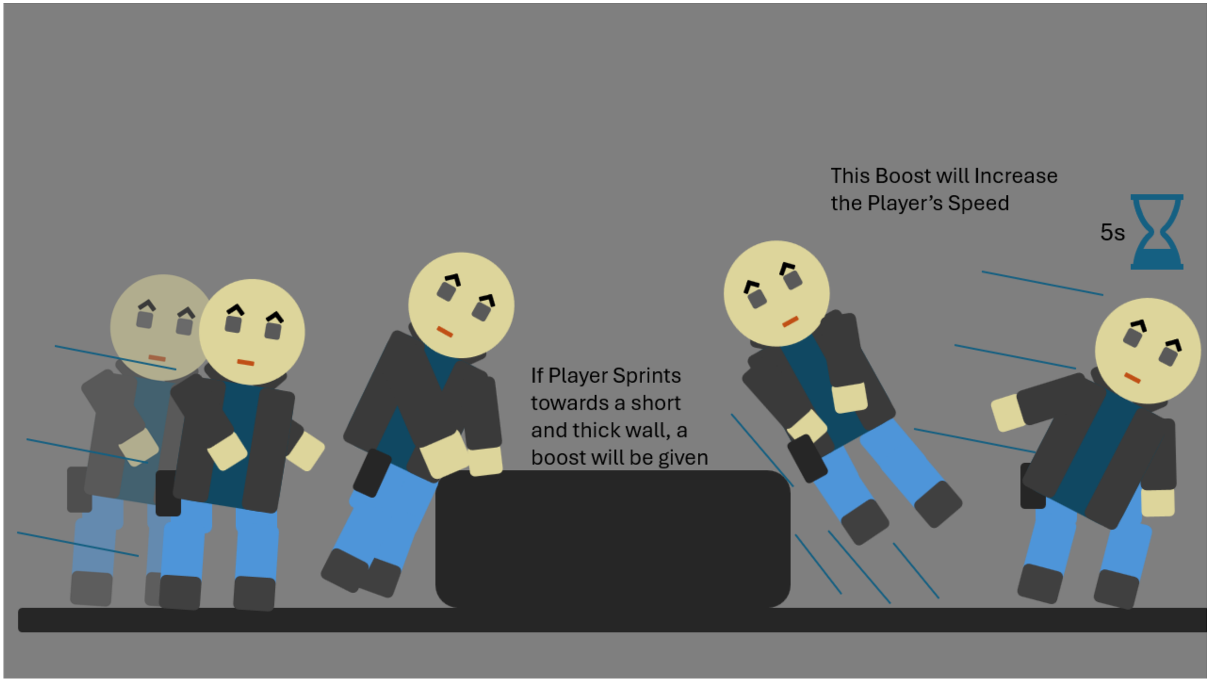
Player Vaulting

Mechanic Diagrams

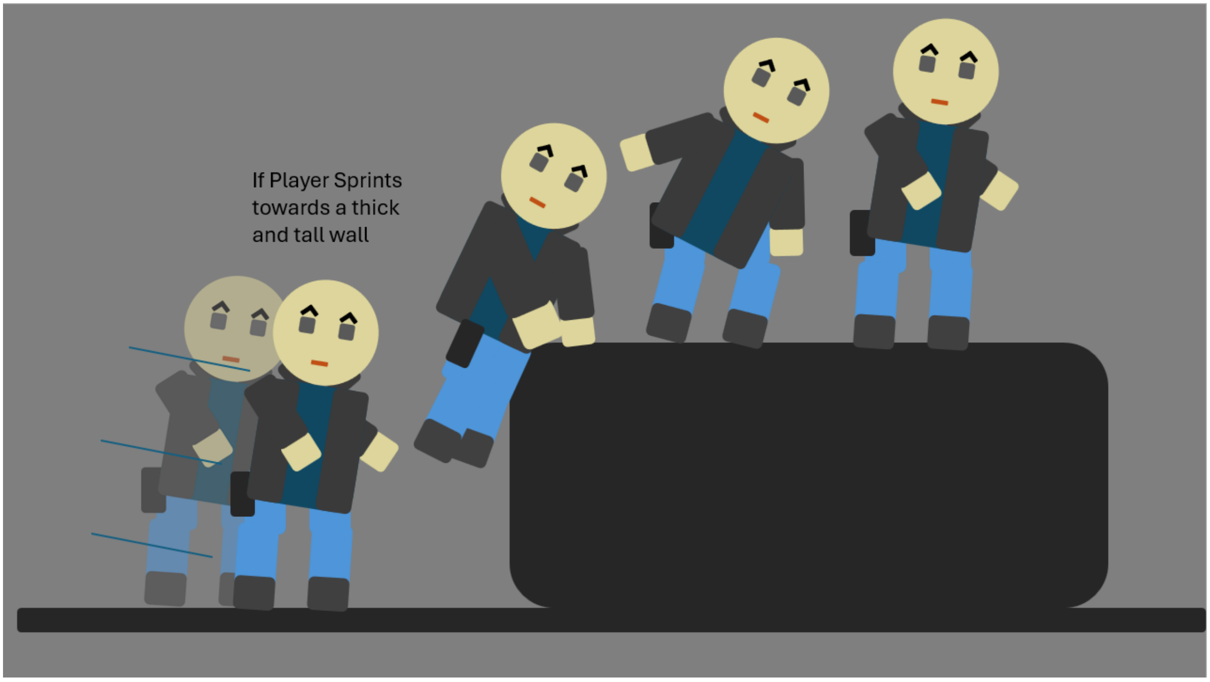
How would the player behave when they sprint towards a wall that isn't too tall or too thick:



How would the player behave when they sprint towards a thick but short wall:

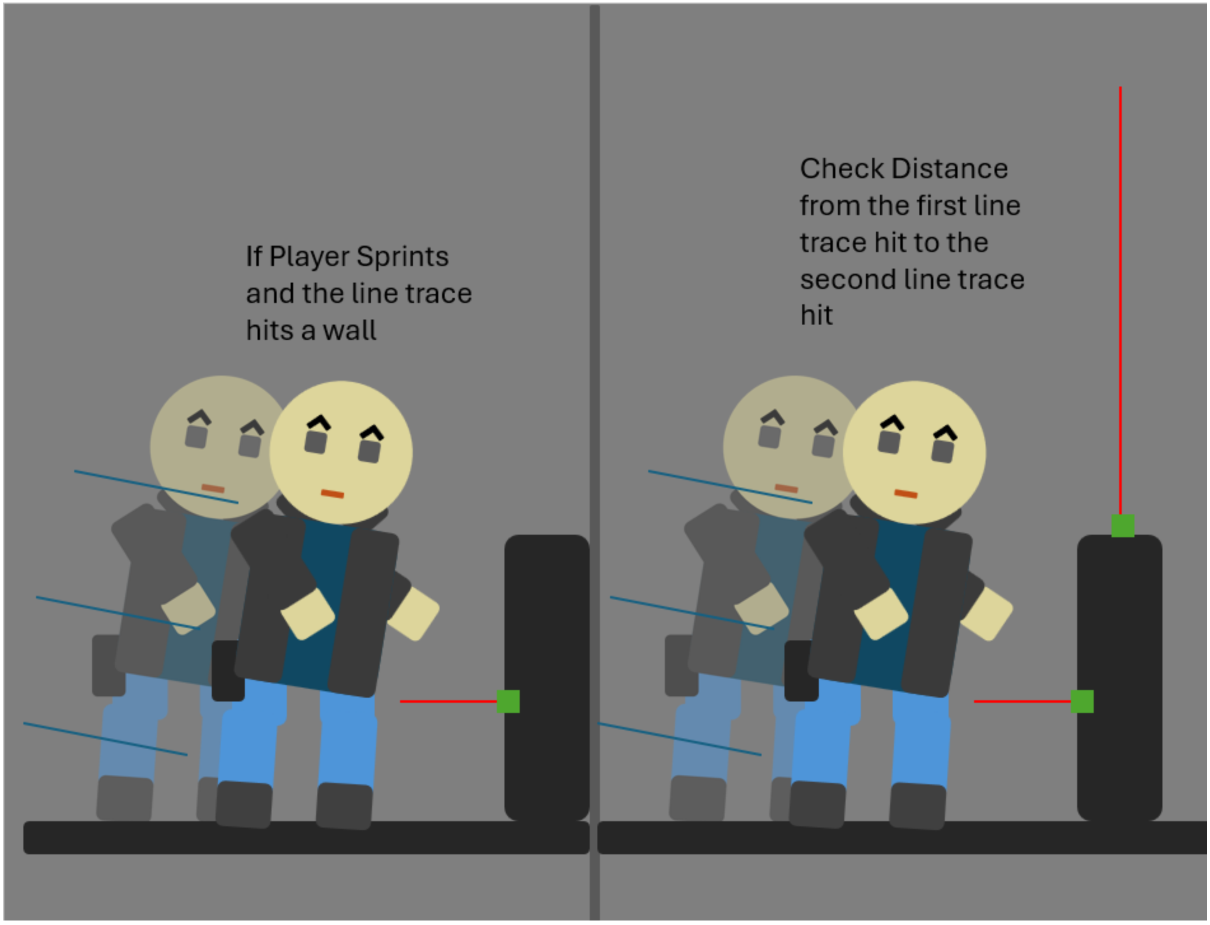


How would the player behave if the wall is thick and tall to vault over:

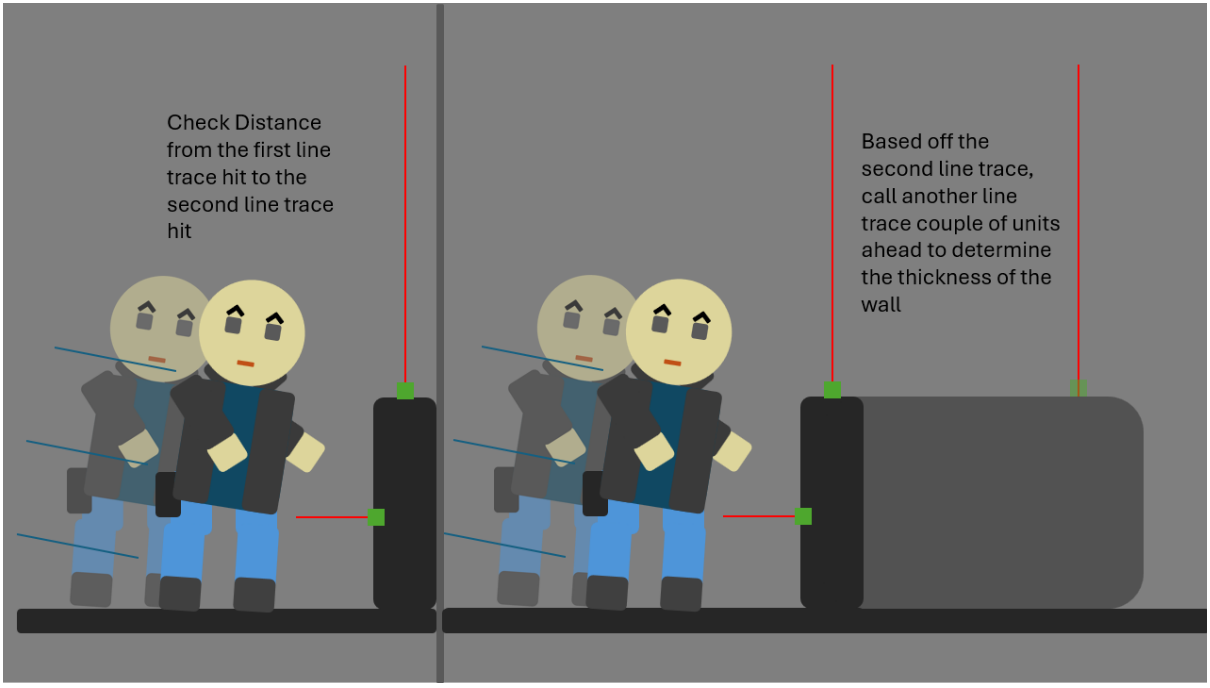


Logic Diagrams

How the height of the wall will be calculated for the player to determine type of vault/mantle:



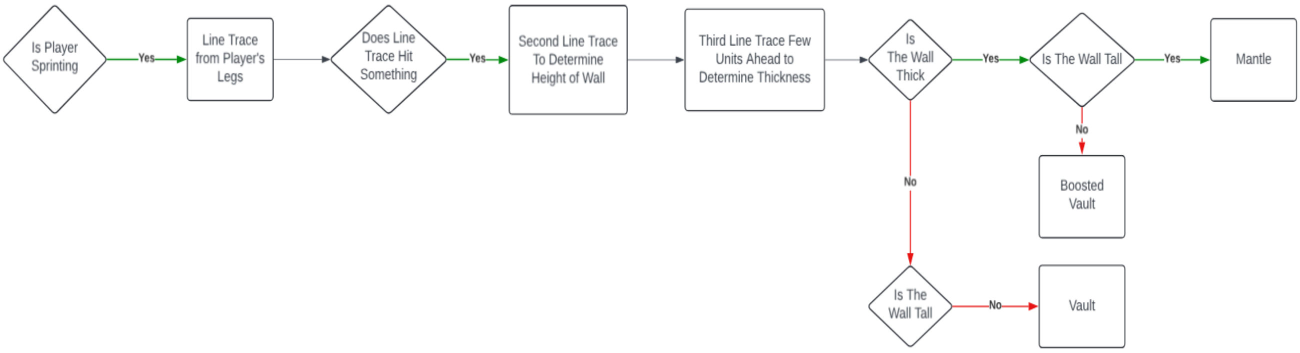
How the thickness of the wall is calculated to see if player can vault over it:



Check Distance from the first line trace hit to the second line trace hit

Based off the second line trace, call another line trace couple of units ahead to determine the thickness of the wall

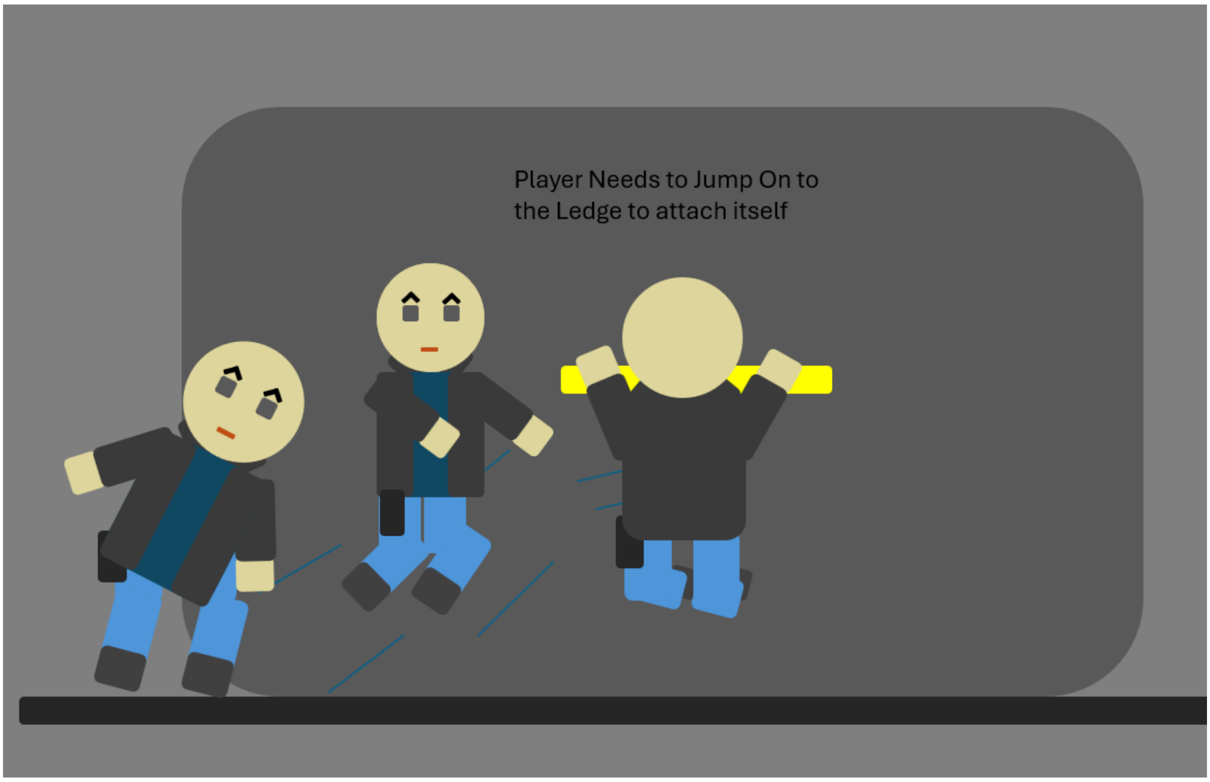
Flowcharts


Is Player Sprinting
Yes
Line Trace from Player's Legs
Does Line Trace Hit Something
Yes
Second Line Trace To Determine Height of Wall
Third Line Trace Few Units Ahead to Determine Thickness
Is The Wall Thick
Yes
Is The Wall Tall
Yes
Mantle
No
Boosted Vault
No
Is The Wall Tall
No
Vault

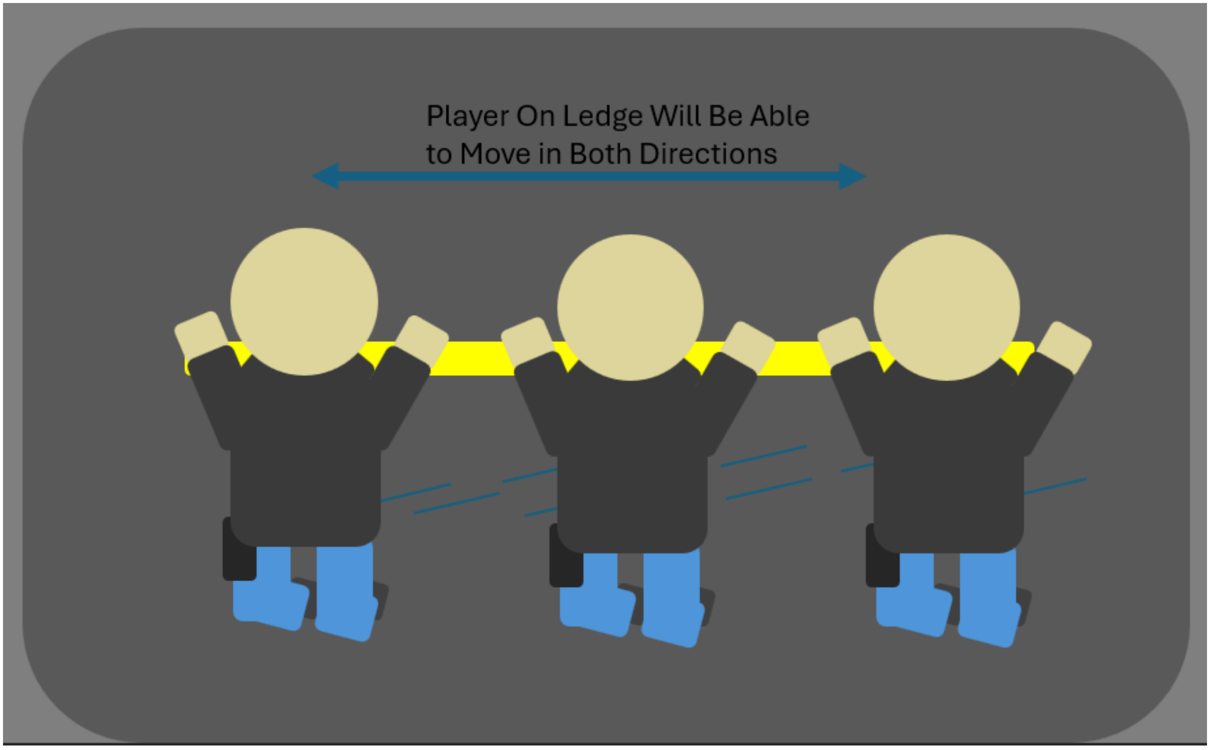
Ledge Climbing

Mechanic Diagrams

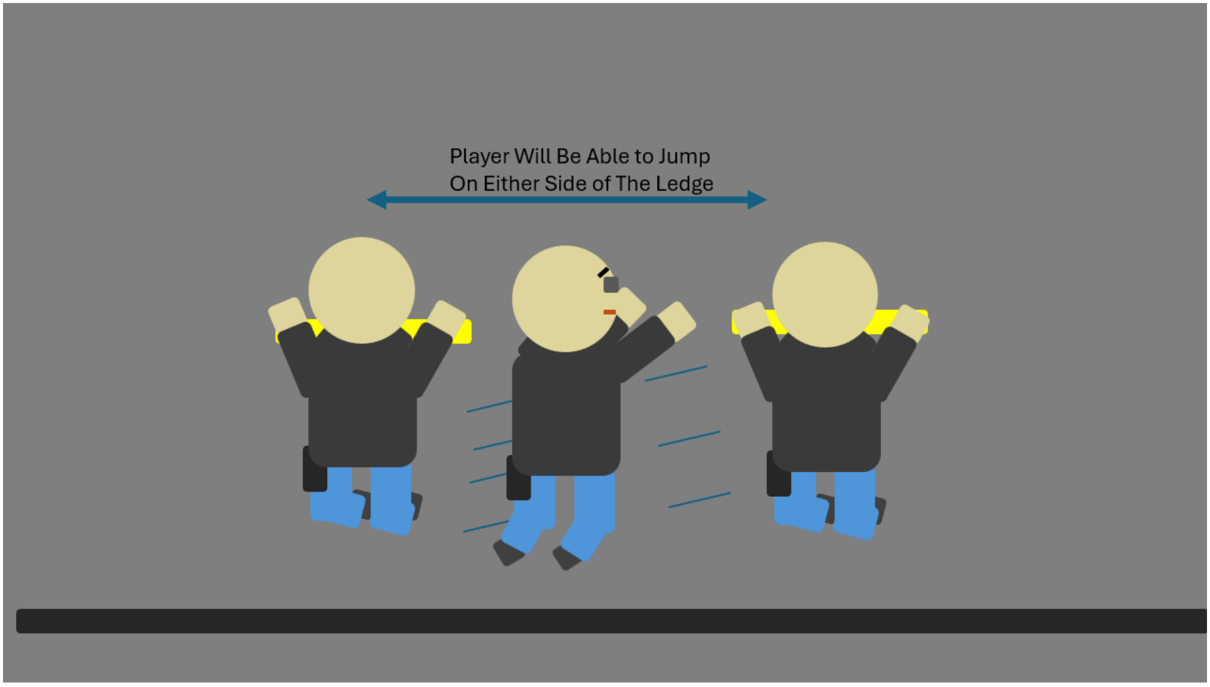
How will the player be able to interact with the ledge and climb onto it:



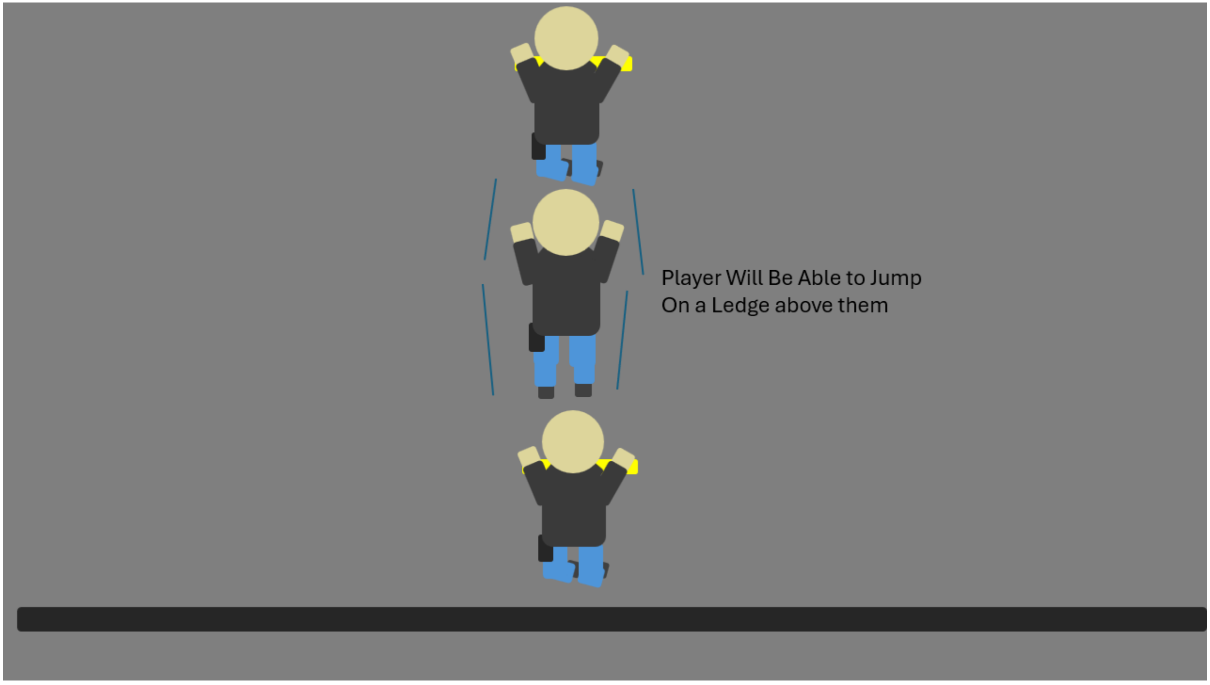
What will the player's movement be while on the ledge:



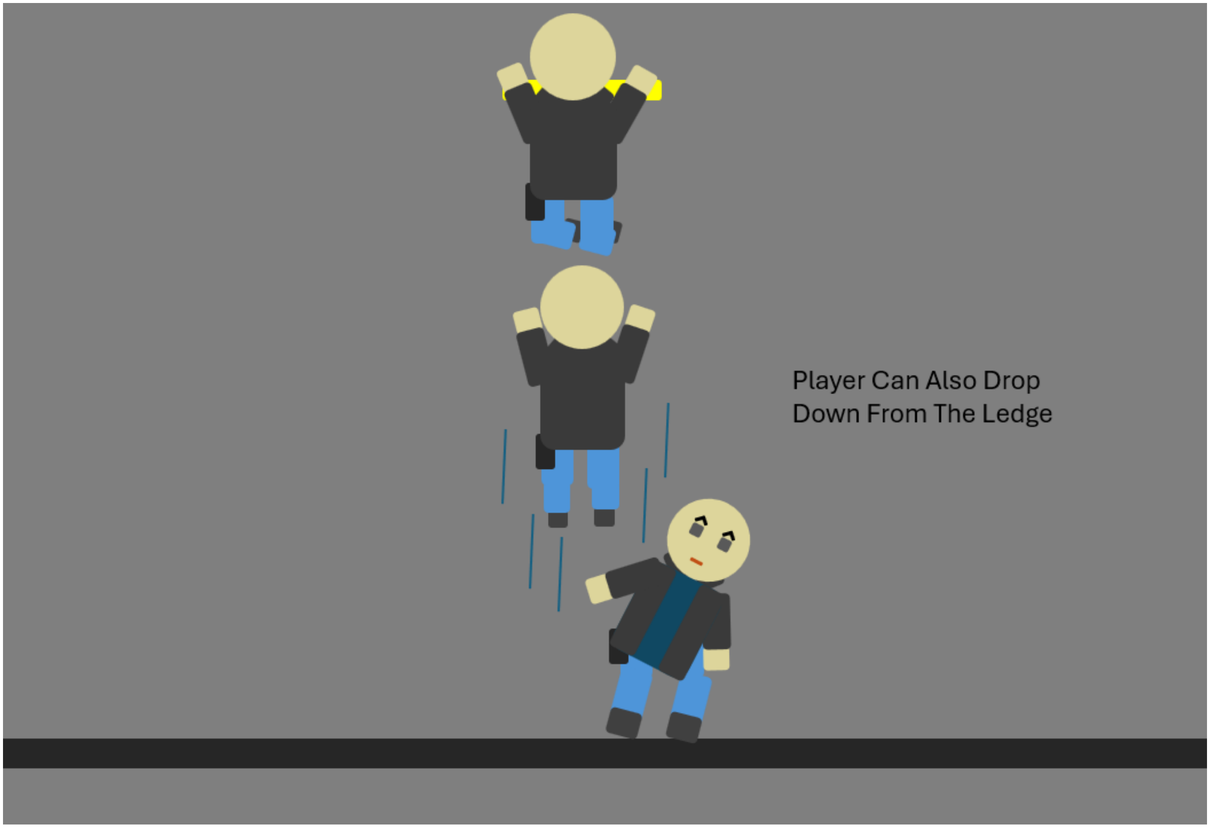
How will the player be able to traverse between ledges:



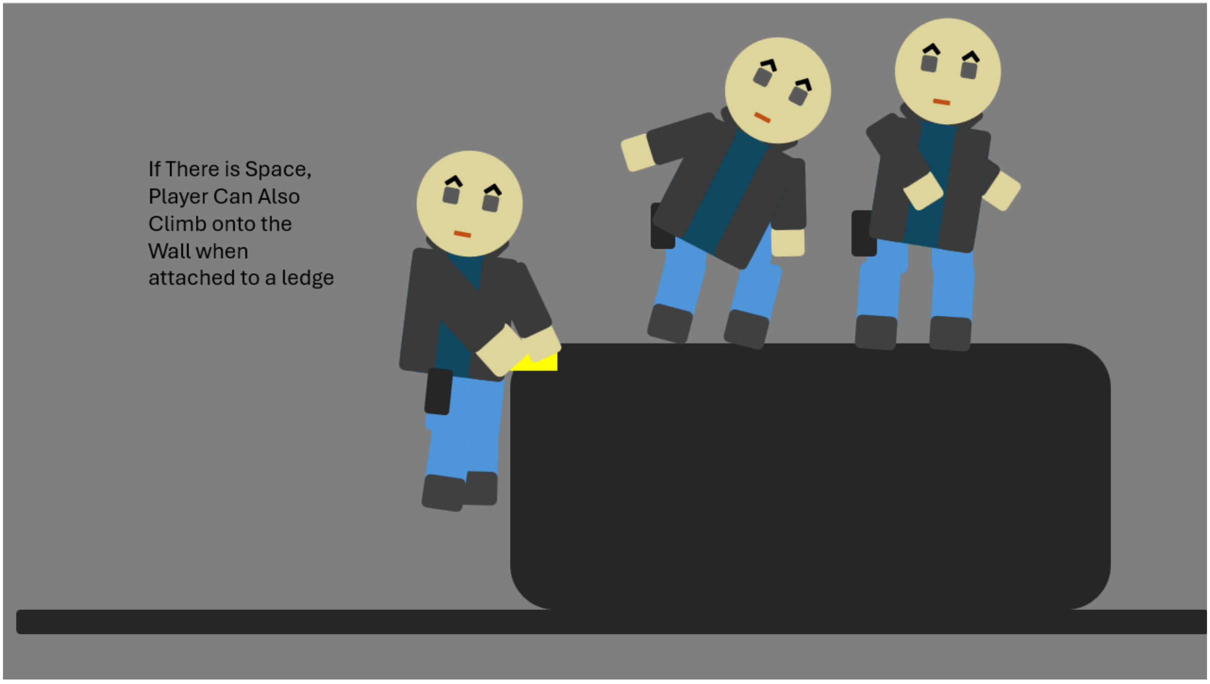
The player should also be able to go to a ledge above them:



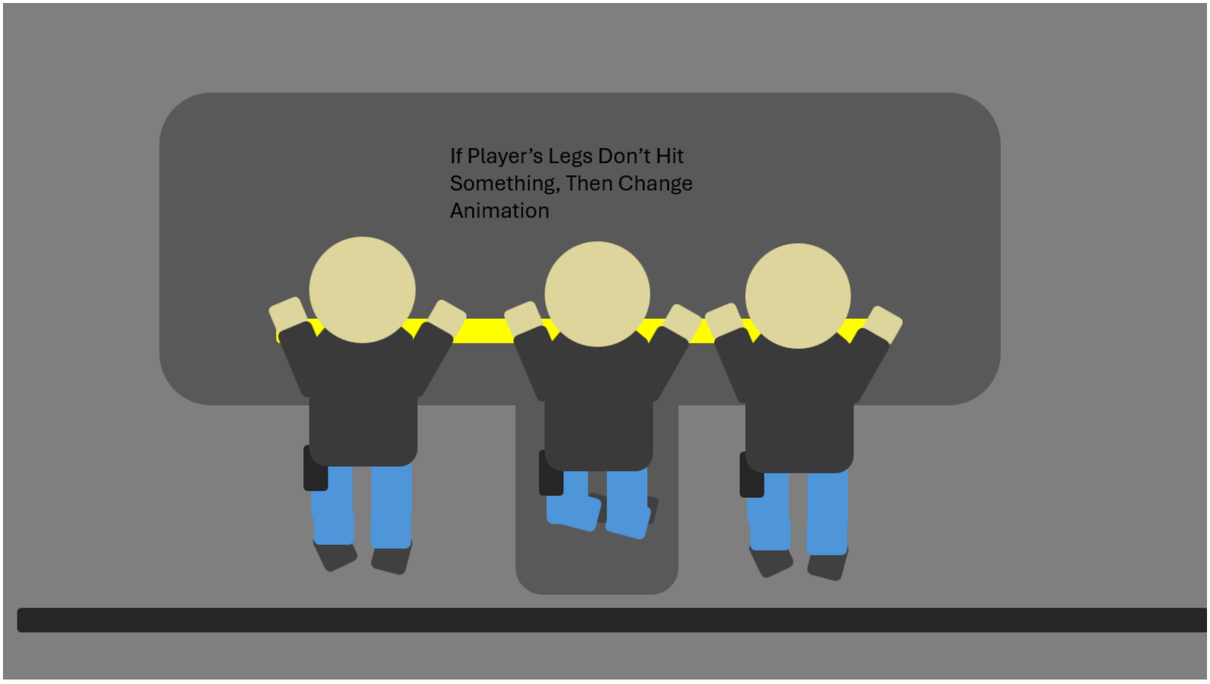
The Player should also be able to drop from ledges:



When possible, player can also climb onto the edge of the wall where there are no obstructions:

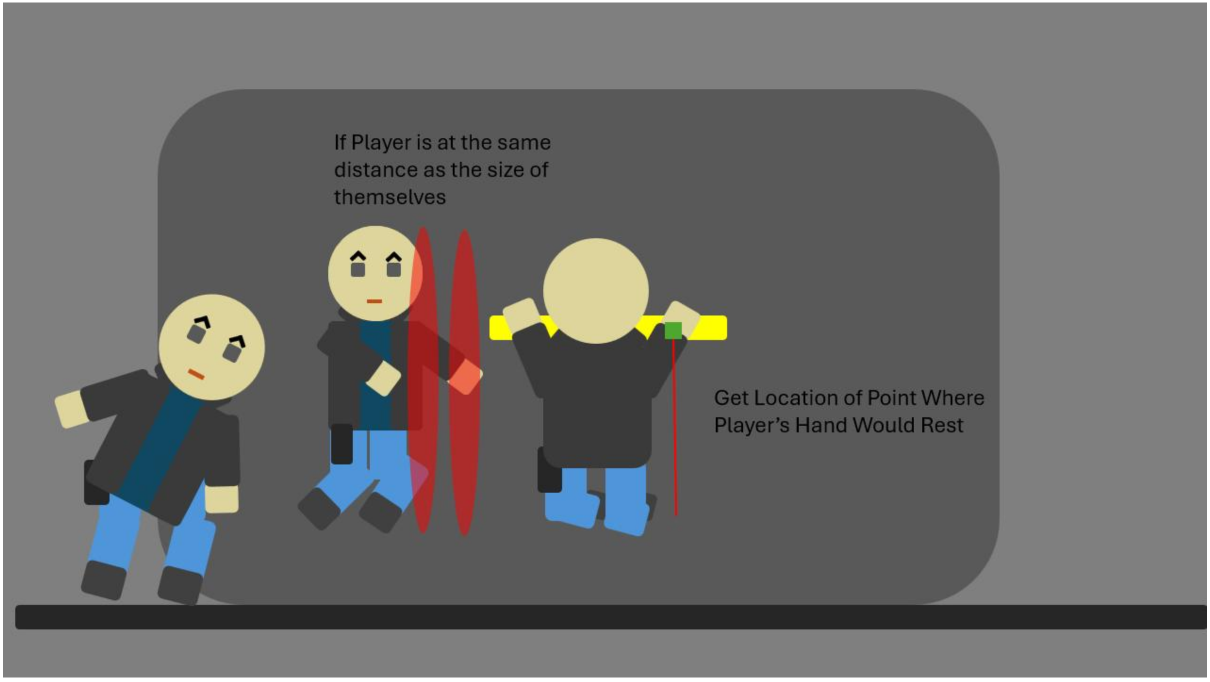


When the player's legs are not touching the wall, there needs to be a change in player's posture:

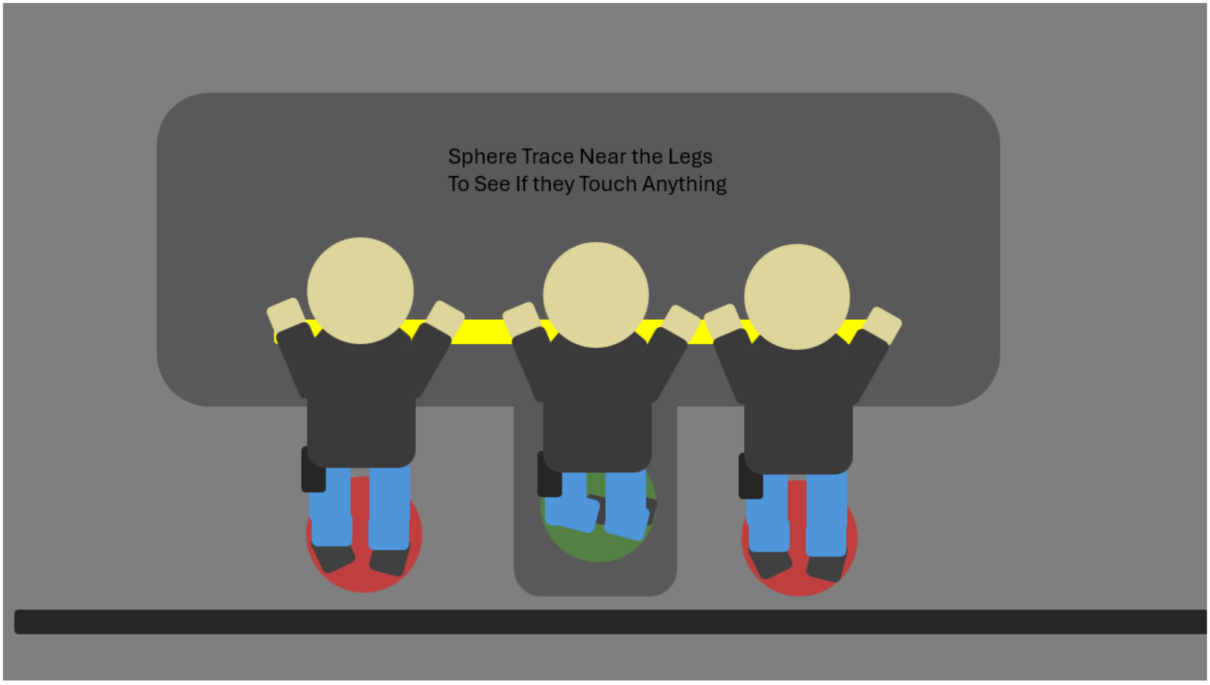


Logic Diagrams

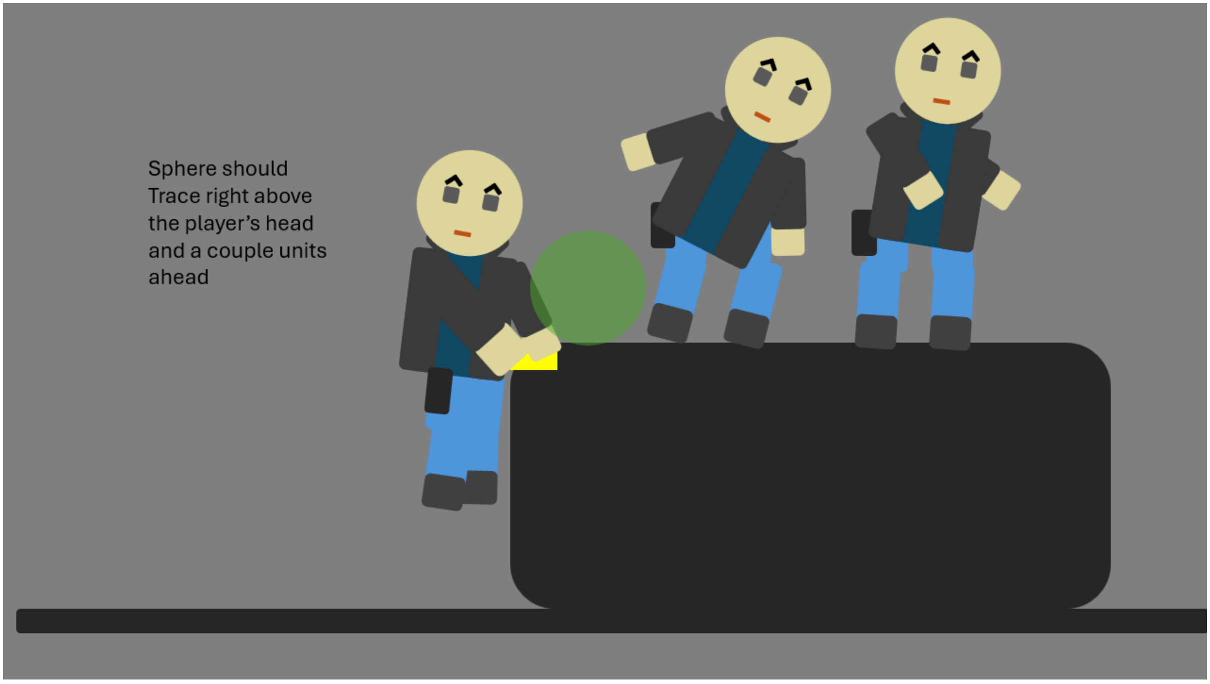
How a capsule trace and then a line trace may be used to determine player's accurate location on the ledge:


If Player is at the same distance as the size of themselves
Get Location of Point Where Player's Hand Would Rest

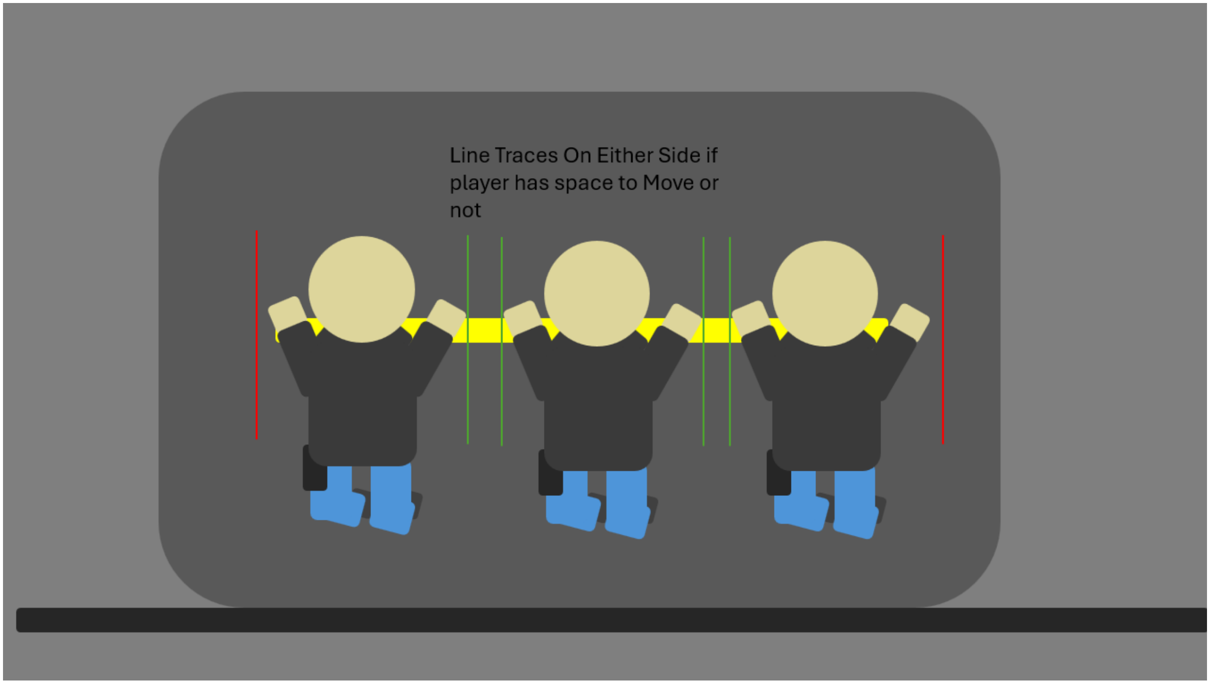
How the Check for If Legs Are Touching the wall may be done:



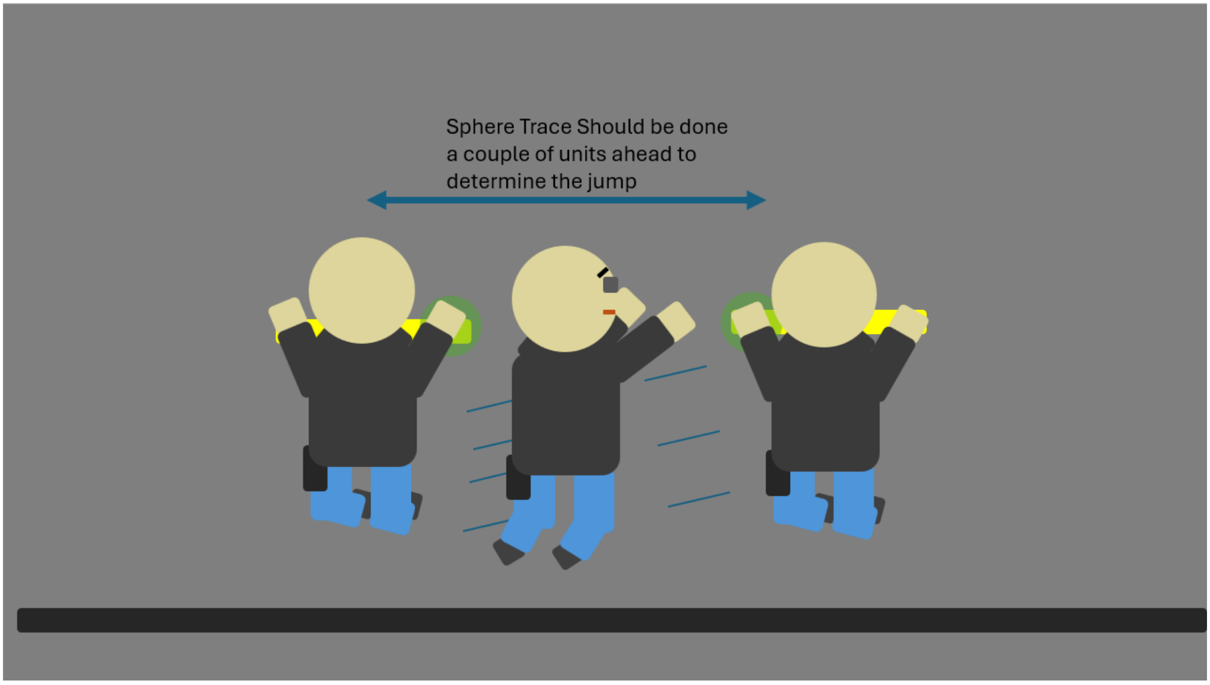
How to determine if there is space to climb on top of the ledge:



This is how each side of player will be checked to make sure they can move either side:

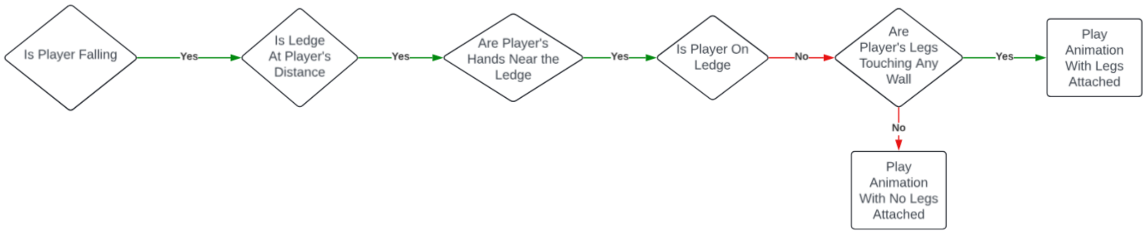

Line Traces On Either Side if player has space to Move or not

This is how a sphere trace may be used to calculate if a jump can be done on either side:

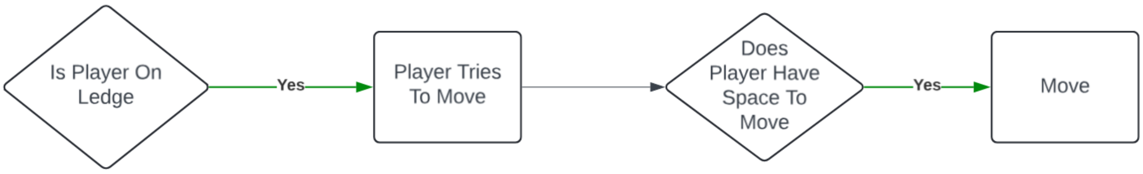

Sphere Trace Should be done a couple of units ahead to determine the jump

Flowcharts

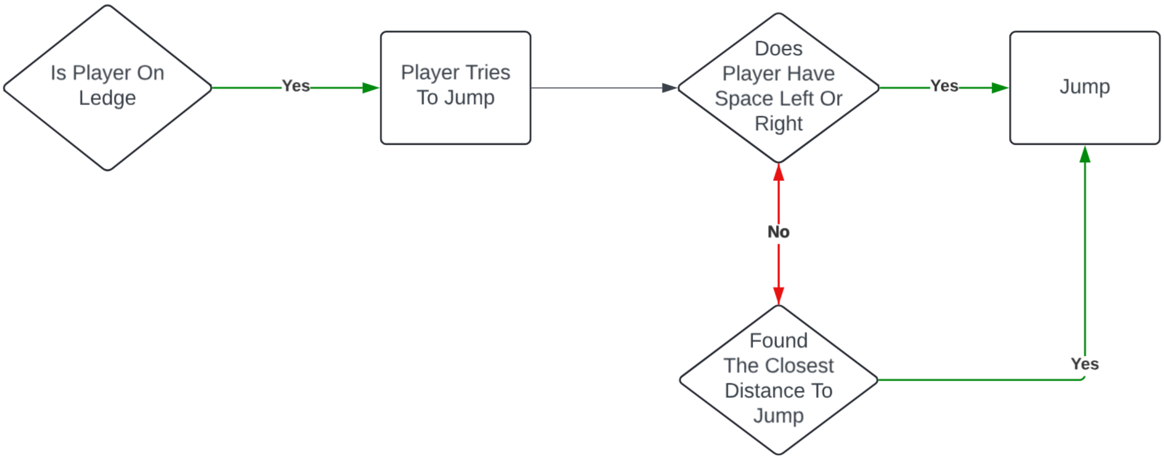
How the Player would be able to jump onto a ledge:


Is Player Falling
Yes
Is Ledge At Player's Distance
Yes
Are Player's Hands Near the Ledge
Yes
Is Player On Ledge
No
Are Player's Legs Touching Any Wall
Yes
Play Animation With Legs Attached
No
Play Animation With No Legs Attached

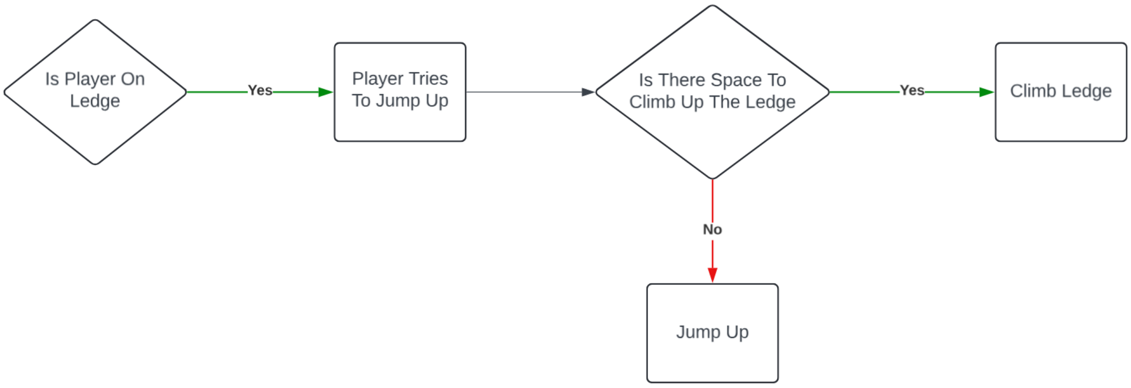
This is how the movement of the player while on a ledge should be checked:



How the jump should work when trying to do it either side:


Is Player On Ledge
Yes
Player Tries To Jump
Does Player Have Space Left Or Right
Yes
Jump
No
Found The Closest Distance To Jump
Yes

How the jump should work when trying to go up:



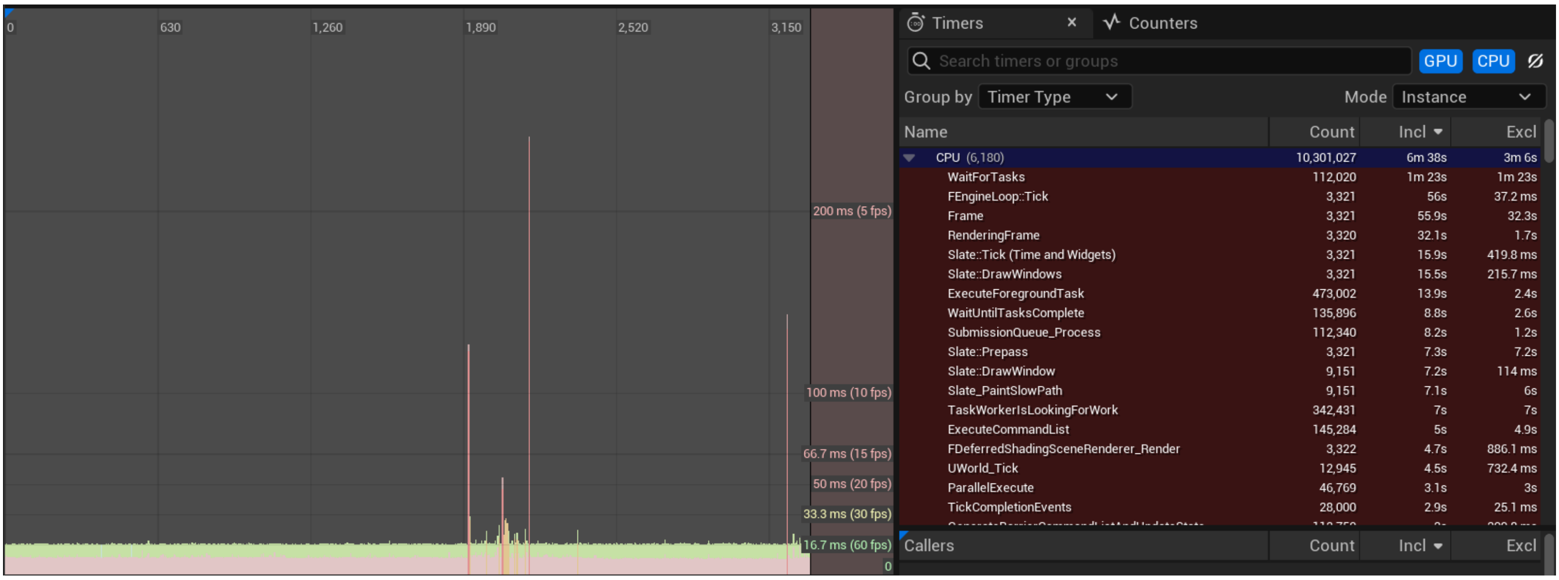
Optimisation and Profiling

Profiling Systems

Each movement mechanic will be tested in isolation to make sure that it works as intended. If needed two separate mechanics will be combined to verify smooth transitions. Systems which may require the animations to be realistic may have an in-depth test to eliminate unwanted behaviours like clipping into objects. All the tests will be made in a separate map called "PrototypeMap" to test the prototype of each system.

Profiling Graphics

Usage of Unreal Insights to track frame times and ensure a constant 60 frames per second performance. Identifying GPU bottlenecks such as high shader complexity and unoptimized post processing effects. Using the GPU Profiler to identify drops in frames and the use of distance field shadows to find the most optimal value. Here is an example of what Unreal Insights does when tracking on BeginPlay until the trace timings stop.



Name	Count	Incl	Excl
CPU (6,180)	10,301,027	6m 38s	3m 6s
WaitForTasks	112,020	1m 23s	1m 23s
FEngineLoop::Tick	3,321	56s	37.2 ms
Frame	3,321	55.9s	32.3s
RenderingFrame	3,320	32.1s	1.7s
Slate::Tick (Time and Widgets)	3,321	15.9s	419.8 ms
Slate::DrawWindows	3,321	15.5s	215.7 ms
ExecuteForegroundTask	473,002	13.9s	2.4s
WaitUntilTasksComplete	135,896	8.8s	2.6s
SubmissionQueue_Process	112,340	8.2s	1.2s
Slate::Prepass	3,321	7.3s	7.2s
Slate::DrawWindow	9,151	7.2s	114 ms
Slate_PaintSlowPath	9,151	7.1s	6s
TaskWorkerIsLookingForWork	342,431	7s	7s
ExecuteCommandList	145,284	5s	4.9s
FDeferredShadingSceneRenderer_Render	3,322	4.7s	886.1 ms
UWorld_Tick	12,945	4.5s	732.4 ms
ParallelExecute	46,769	3.1s	3s
TickCompletionEvents	28,000	2.9s	25.1 ms

Coding Standards

Programming Standards

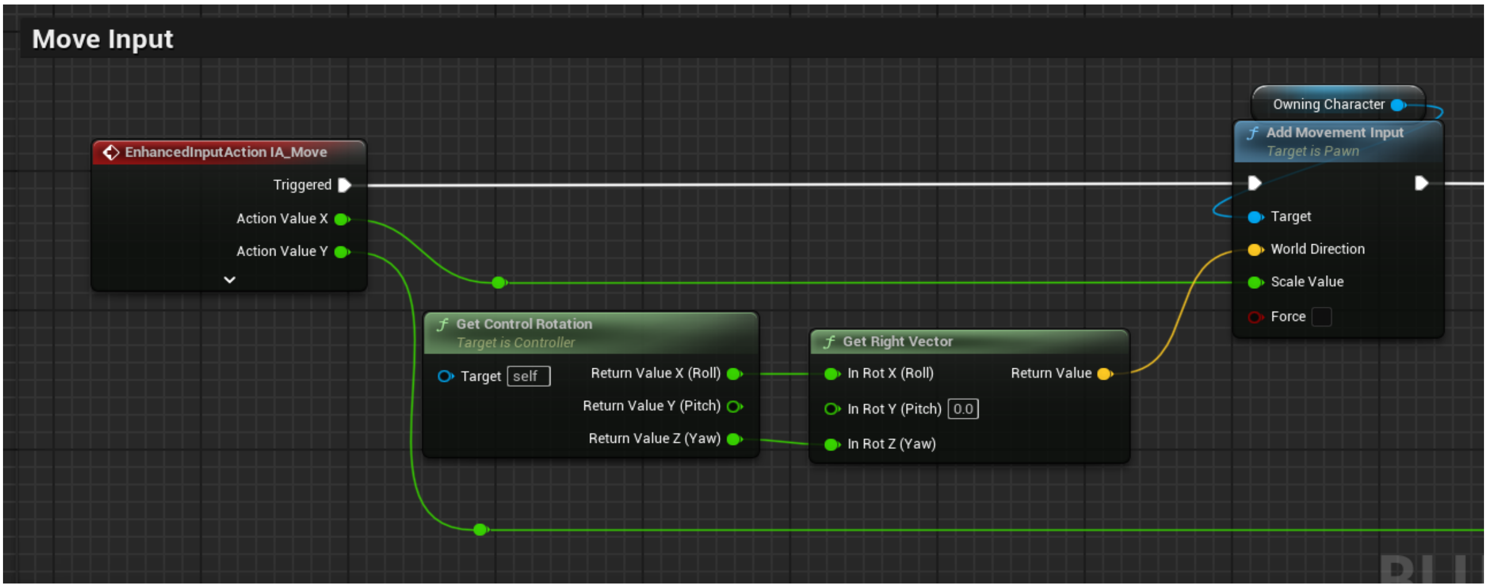
- All blueprints will be prefixed based on type:
    - BP_ for purely made Blueprint classes.
    - BPI_ for Blueprint Interfaces.
    - BPL_ for Blueprint Libraries.
    - WBP_ for User Interface Widget class.
    - ABP_ for Animation Blueprints.
    - E_ for Enumerations.
    - GM_ for Game Mode classes.
    - GI_ for Game Instance classes.
    - S_ for Structures(Structs).
- The same prefixes will be used for Animation object types like:
    - M_ for Animation Sequences.
    - AM_ for Animation Montages.
    - PSD_ for Pose Search Databases.
    - PSN_ for Pose Search Normalisation Sets
    - PSS_ for Pose Search Schemas.
    - CT_ for Chooser Tables.
- Other actors will use this similar trend to set them apart based on which folder they're placed in:
    - M_ for Materials.
    - MI_ for Material Instances.
    - MF_ for Material Functions.
    - NS_ for Niagara Systems.
    - LS_ for Level Sequences.
    - IMC_ for Input Mapping Contexts.
    - IA_ for Input Actions.
- Since all the variables are already colour coded, no naming conventions will be used for that, although they will be split into different categories based on their functionality and will be named logically based on the use of the variable.
- To make the game efficient, the use of event tick is minimised whose alternative will be the use of timers.
- Use of casting will be limited and if needed will be only used to store casted references at the beginning.
- Blueprint Interfaces and Event dispatchers will be used for communicating between actors.
- Blueprint Libraries will be used to call functions within multiple actors for limiting the creation of identical functions in different actors.
- Functions will be used to handle repetitive logic to avoid overcrowding in the event graph.

Style Guide

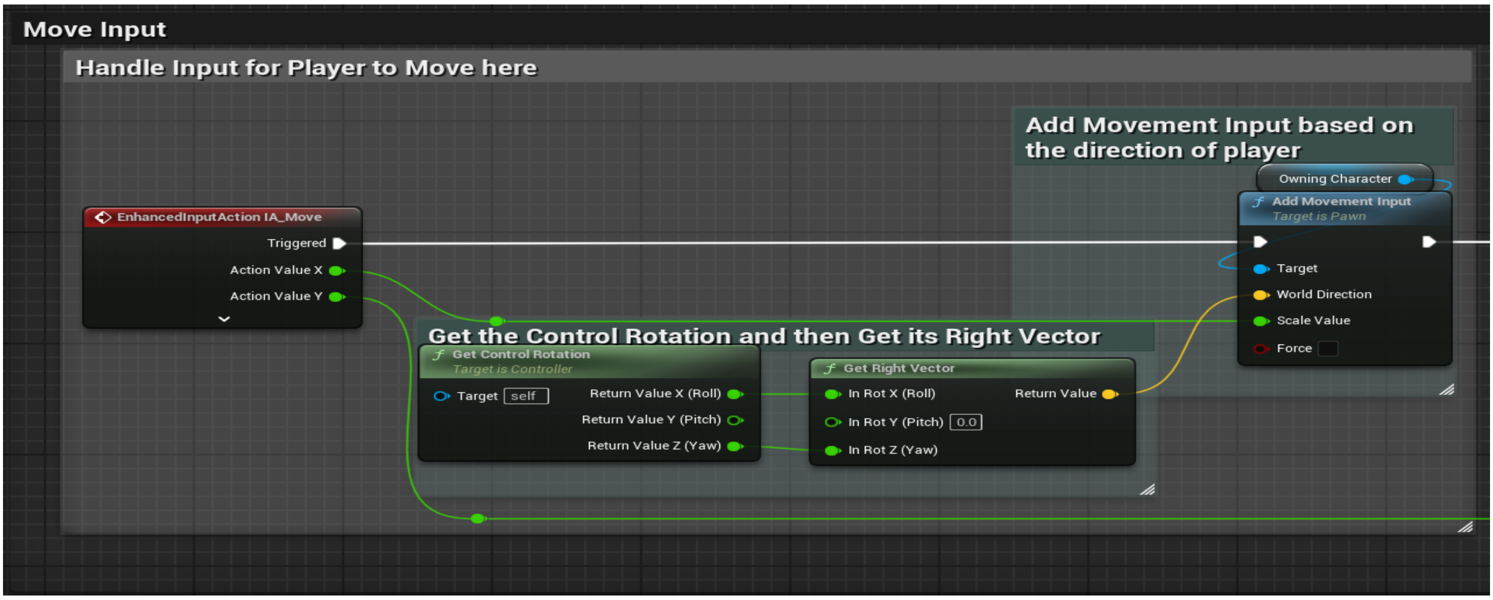
- *All blueprint nodes will start from left and end at right to keep the code readable and the flow consistent.*
- *Nodes will be rerouted to make sure none of them entangle and cause potential spaghetti code.*
- *Actors with multiple variables will be classified into their own category to keep them relevant in their own functionality.*
- If the nodes start to expand further, some will be broken down into individual custom functions and events(if using time-based nodes) to keep the code readable and efficient.
- Each function will also have their own category to make it easier to find due to its relevance to the category.
- All nodes will be regularly aligned to keep the code steady and not make the code hard to navigate through.
- All UASSETS will be sorted into colour coded folders within the engine for the ease of being able to find any asset.

Commenting Rules

- Each set of a main function/system will be commented to make them stand out from the rest. All the comments in the outermost layer will have a darker tint with a bubble text over it to make it distinguishable.
- Comments within the outer comment layer will be of grey colour whereas a comment inside that will have a shade of cyan. Layered comments won't have the bubble text to make the comments look more breathable.
- There will be a maximum of only three-layered comments after which will be the case of using a function to not make the event graph look crowded.
- Inner Comments will only be placed for non-obvious logic used within blueprints, or to separate the functionality.
- Logics which are self-explanatory will not be commented as that will make the comments lose its purpose.
- Here is an example of bad commenting:



Move Input



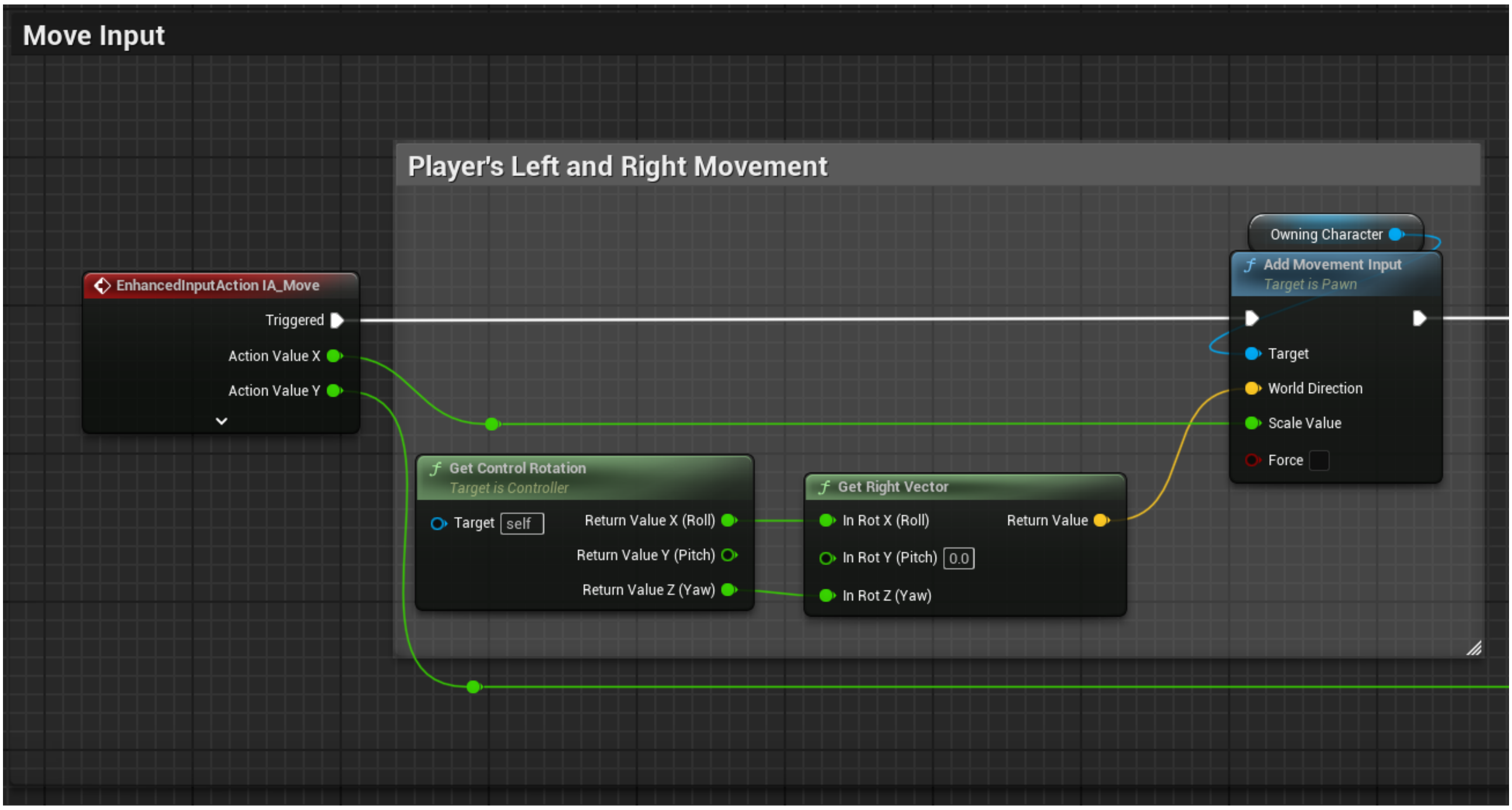
Move Input

Handle Input for Player to Move here

Add Movement Input based on the direction of player

Get the Control Rotation and then Get its Right Vector

- This is an example of a good comment:



Code Review Procedures

There will be a check done each week to see if coding standards are maintained. Code will be assessed for readability and adherence to good coding practices. Each system will be tested weekly to ensure its functionality and compatibility between systems will be checked to confirm smooth interaction and prevent conflicts.

Production Overview

Moscow

Must	Should	Could	Won't
Wall Running	Controller Support	Wall Sliding	Cover System
Wall Jumping	Sounds	Dropping to Ledge	Enemies
Ledge Climbing	Visual Effects	Free Running	Stamina System
Vaulting	Level Design	Player Swap	Combat System
Mantling		Dynamic Menu	Multiplayer Support
Sliding			Cutscenes
Fluid Animations			
Hacking System			
Tutorial Level			
Main Menus			

Timeline

GanttChart_FlowState.
xlsx

Budgeting

The project will take 8 weeks to complete with around 20 hours spent each week. Approximately 150-160 hours to complete the project.