Don’t Stop Moving

(Developing a game in UNITY)

by

Ishan Prakash

Submitted to the Department of Mathematics and Computer Science
School of Natural and Social Sciences
in partial fulfillment of the requirements
for the degree of Bachelor of Arts

Purchase College
State University of New York

May 2022

Sponsor: Irina Shablinsky

Second Reader: Athar Abdul-Quader

Contents

Abstract

This paper describes the process of creating a high-speed 3D platform game that features a physics-based movement and custom physics for various elements in the game. The technical aspects of the game creation can also be used as a mini guide for people who want to create similar types of games or want to explore the usage of physics in the UNITY game engine along with the programming language C#.

I chose to focus on this topic because of the lack of tutorials and articles on a physics-based movement system. Even UNITY's proprietary character controller doesn't handle physics very well. My goal is to use this movement mechanic as a tool in other games I create.

Chapter 1: Introduction

1.1: Inspiration

I went into the field of Computer Science because I always had a sense that I could use programming to create a work that expressed my thought process and also expose others to my thought process. I explored many topics in the field but during a class in game development, I found exactly the mode of expression I wanted to use. Game development has allowed me to create something I really love and to create it however I wish and lastly allow other people to enjoy my creation. After extensive reflection on a possible project, I decided to lift some of what we might generally call the "various mechanics" from different games I like to play and combine them into one game.

The main mechanical features I have decided to put into my game are rapid movements and the idea of retaining stable aerial suspension to complete the level of the game. The player can become airborne by swinging from ropes in a Spider-man style.

1.2: Concept

This game is a high-speed platformer that mixes in the concept of verticality (staying off the ground as much as possible). I encourage this style of playing by limiting the speed the player can gain when starting from the ground and only allowing the player to gain speed once airborne.

The main way the player will stay off the ground is by using the grapple system granted to the player. The grapple system consists of a rope that can be shot from the player to some surfaces which will allow the player to swing on the rope and gain momentum.

The levels are designed to encourage the player to stay off the ground as much as possible to gain speed by taking advantage of the grapple system.

1.3: Tools

To create this game, I am using the game engine **UNITY** along with the programming language **C#.**

I am using UNITY because I have previous experience with it, and it is a very simple tool to use. I am drawn to UNITY because it has a very user-friendly environment for creating games. Most of its components are highly visual and greatly simplify the game development process. You can create many components in your game without any code by using the built-in drag and drop applications that come with UNITY.

UNITY also features an Asset Store which contains a library of free ready-made assets to download. Since my game focuses on the mechanics rather than the looks, I have used some ready-made assets to beautify the game.

I used the programming language C# since it is the default language used by UNITY. Getting used to C# was not hard since I already had some experience with C++ and JavaScript.

1.4: Literature Review

Catlike Coding[1] is a website that gives in-depth tutorials for all the basics of Unity. All these tutorials are in text with a few images which are very detailed. All concepts in this website build on one another, introducing new programming concepts, math, algorithms, and Unity features. This website contains a tutorial on a very simplistic model of a “Physics character-controller” which is built with the help of pure mathematical formulas which helped me to gain a basic understanding of how the physics of character movement should be represented.

When reading articles/tutorials from the Catlike Coding website I often would encounter functions or methods that were built-in to the UNITY engine and whose capabilities I could not readily or fully understand. In those cases, I would consult the official UNITY documentation.

The official UNITY documentation[2] is a collection of all function descriptions and methods used in the game engine and the scripts. I mainly used this documentation as a way of understanding more about a particular function I wasn’t sure how to use. The documentation also has mini-tutorials on how to use the various functions in the engine.

[1] (Catlike Coding, 2022)
[2] (Unity user manual 2020.3 (LTS), 2022)

Chapter 2: Player Movement Overview

2.1: Setup

2.1.1: Terminology

In this section I will describe terms that I reference repeatedly throughout the paper.

A **First-person** game is a game where the perspective is rendered from the point of view of the player's GameObject.

A **GameObject** is an object in the game world that can represent anything. A GameObject can represent shapes (cubes, spheres, etc.), characters, light sources, and many other things you want to be a part of the game world. A GameObject on its own cannot do much so we can attach **components** to GameObjects to add functionality. Components can also be created/destroyed and referenced through scripts the user adds. Scripts allow the user to define specific behaviors for GameObjects.

Every GameObject in the game comes with the **Transform** component attached to it. The Transform component keeps track of the position, rotation, and scale of the GameObject.

The Transform component also helps in linking **Parent** GameObjects with **Child** GameObjects. A child's GameObject transform is always relative to the parent's GameObject transform. This means that if the parent's position changes the child's position also changes but if the child's position changes the parent's position does not change. The same principle applies for everything the transform keeps track of. We can use this Parent and Child linking to keep track of complex GameObject hierarchies.

2.1.2: Scene

The player GameObject will be a simple capsule GameObject with a **capsule collider**, an added component that helps our player have physical interactions within the game world. This feature is important because this prevents the player's GameObject from going through other objects and helps the player determine when touching the ground or not.

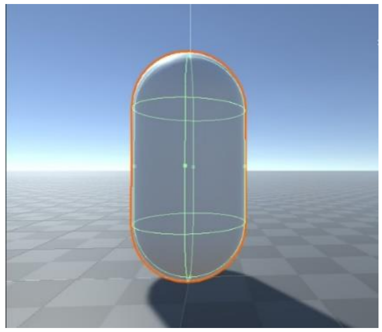

Fig. 2.1 Capsule Collider

Fig. 2.1 shows the capsule shape of the player's GameObject I have chosen, and the green outline shows our capsule shaped collider. (The red outline is insignificant; it just helps in identifying selected objects).

2.1.3: Rigidbody component

Since I want to create a movement system that utilizes real world physics, I am taking advantage of UNITY's built-in **Rigidbody** component. This component allows GameObjects to act under the influence of physical phenomena like translational and rotational forces.

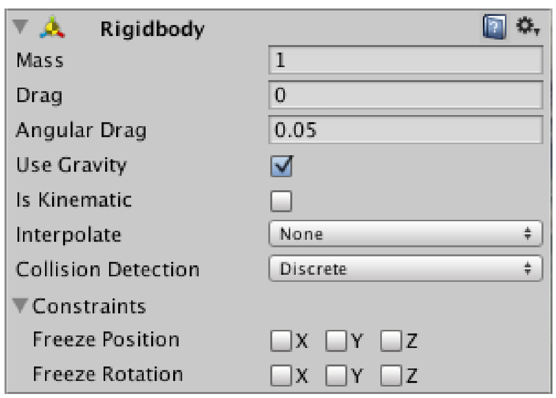

Fig. 2.2 Rigidbody component details

2.1.4: Scripts and Input Details

Once the engine knows what to do with our player GameObject, we need a way to tell the engine how to interact with the player object. We can do this by using scripts. In our case the script will contain code that describes the conditions for applying different forces to the player object and how to read input from the keyboard and mouse.

2.1.5: Vectors and Input Reading

The direction the player should move is decided by the direction the player is currently facing and the current input provided by the user. To easily compute all the required movements I am using **vectors**.

```
horizontalMovement = Input.GetAxisRaw("Horizontal");
verticalMovement = Input.GetAxisRaw("Vertical");

moveDirection = transform.forward * verticalMovement + transform.right * horizontalMovement;
```

Fig. 2.3 Movement Direction Code

Fig. 2.3 shows how the move direction is being calculated by reading the user input and the direction the player is facing. The input is being read by the first two variables “horizontalMovement” and “verticalMovement”. So, if you press the left or right arrow key “horizontalMovement” will be equal to 1 or -1 respectively and similarly for if you press the up or down arrow key “verticalMovement” will be equal to 1 or -1 respectively. If either pair of keys isn’t pressed, then the respective pair’s variable will be equal to 0.

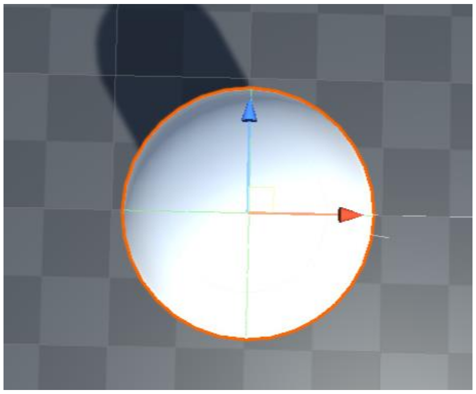

Fig. 2.4 Axes of the player

The direction the player is looking is calculated by the “transform.forward” and “transform.right” functions which return a 3-dimensional vector. The “transform.forward” vector describes the forward position (Blue axis on Fig. 2.4) and the “transform.right” vector describes the right position of the player (Red axis on Fig. 2.4). These vectors will always tell me the forward and right directions of the player even if the player rotates.

The” moveDirection” variable is the combination of the inputs and the direction of the player such that the output of that combination will be a 3-dimensional vector representing the direction the player should move relative to the current location and the direction the player GameObject is facing.

One important step to include is to normalize the “moveDirection” variable otherwise we would be able to move faster if the direction of the “moveDirection” vector is diagonal to the front and right vector of the player as shown in Fig. 2.5. This is a problem as the GameObject’s speed will increase faster if you were moving diagonally than opposed to moving front, back, left, or right. This is not realistic as you want to gain the same speed regardless of where the object is facing.

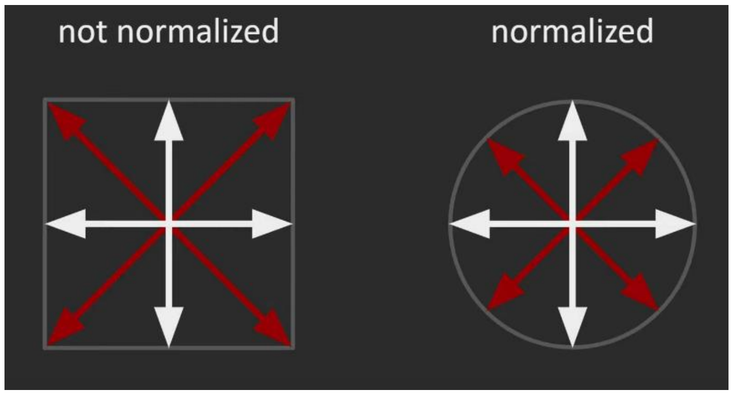

Fig. 2.5 Normalized vs. Not Normalized Vector

2.1.6: Main Camera

The main camera must follow the player object around when it moves. To accomplish that I make the main camera GameObject the child of the player GameObject.

Now that we know how to handle our player GameObject movement based on inputs and player GameObject direction, we describe how to handle our viewing direction. Since this is a First-Person game, the mouse will be directly connected to how the player GameObject will look around. We took input from our keyboard arrows above and now UNITY provides a convenient built-in function for the mouse.

```
mouseX = Input.GetAxisRaw("Mouse X");
mouseY = Input.GetAxisRaw("Mouse Y");

camRotY += mouseX
camRotX -= mouseY

camRotX = Mathf.Clamp(camRotX, -90f, 90f);
```

Fig. 2.6 Code Handling Camera

In Fig. 2.6, the “mouseX” and “mouseY” variables capture the vertical and horizontal movement of the mouse.

“camRotY” is tracking the mouse and determines how much the camera is rotated around the y-axis which will be the left and right movement of the camera, so we add the horizontal movement of the mouse to it. “camRotX” contains the value the camera is rotated around the x-axis which will be the up and down movement of the camera, so we subtract the vertical movement of the mouse to it (In this case we are subtracting because if you don't then if you move your mouse up the player will look down. This is due to how the UNITY game engine handles rotation on different axes).

One important thing to keep in mind is that we must keep the rotation capabilities for looking up and down of the camera limited because your neck can bend only so much while looking up and down. We achieve this my using "Mathf.Clamp" which limits a variables value between two given values. In this case it limits the x rotation of the camera to between -90 degrees and 90 degrees.

After calculating the appropriate rotational values, we are now ready to apply these values to the camera and player.

```
cam.transform.localRotation = Quaternion.Euler(xRotation, 0, 0);
transform.rotation = Quaternion.Euler(0, yRotation, 0);
```

Fig. 2.7 Code handling look direction

Looking at Fig. 2.7, The "xRotation" is only applied to the camera because we only want the camera to look up and down and we don't want the entire player GameObject to angle forwards and backwards (Angling your neck to look up and down vs. Angling your entire body to look up and down). This is accomplished by the first line of code in the above picture.

The "yRotation" is only applied to the player because we want the player to rotate left and right and the camera would also rotate accordingly because it it's the child of the player. This is accomplished by the second line of code in the above picture.

Now we have a system to determine where and how the player looks around and how to move around relative to the direction. The next step is to determine by how much we should make the player GameObject move and under what conditions we should slow down or speed up?

Chapter 3: Ground Movement

3.1: Dynamic Velocity Cap

The simplest way to add ground movement is to add a Force to the player GameObject in the appropriate direction (in our case it would be in the same direction as the "moveDirection" variable described in section 2.1.5).

```
rb.AddForce(moveDirection * speed);
```

Fig. 3.1 AddForce function

Fig 3.1 represents how to implement a function that applies force to the player's GameObject. The "rb" represents the RigidBody component (Section 2.1.3) the force is going to be applied to. The "AddForce" function takes in a velocity vector and applies force on the RigidBody in the direction of the velocity vector. In our case, the velocity vector will be calculated as "moveDirection * speed" (This is related to real world physics where velocity is equivalent to a specification of an object's speed and direction of motion).

The problem with this method is that the force applied is unbounded and the players gets to very fast speeds very quickly. This makes the game unplayable and not enjoyable. A simple fix for this is to cap the player's GameObject velocity, but this only temporarily fixes the issue. The player GameObject still can reach that limit very fast.

To fix both the issues and achieve smoother movement I implemented a "Dynamic Velocity Cap". This method still caps the player's GameObject velocity, but the cap dynamically increases or decreases depending on the player's GameObject current velocity. This lets me achieve a very smooth movement without affecting the force that is being applied to the player's GameObject.

The main idea behind this implementation is that if the player's GameObject current velocity is above around 90% of the max velocity, then increase the cap and vice versa.

I chose 90% as my cap after testing my game with various percentages and 90% seemed to have the smoothest movement.

```
if (currentVelocity >= maxVelocity - 0.5f)
{
    maxVelocity += Time.deltaTime * 2;
}
else if(currentVelocity <= maxVelocity * 0.9f)
{
    maxVelocity -= Time.deltaTime * 5;
}
```

Fig. 3.2 Dynamic velocity cap code

Fig. 3.2 describes how I have implemented the Dynamic Velocity Cap. The “currentVelocity” variable describes the player’s GameObject current velocity. This is the variable that will be used to determine whether we should increase/decrease the velocity cap. The “maxVelocity” variable describes the current velocity cap. Moving onto the if statements, the first if statement handles whether the current velocity is close to the current cap, and if it is we increase the cap by a small amount. If the player is not close to cap the second if statement checks if the current velocity is lower that 90% of the current cap, we decrease the cap.

With this I have accomplished my goal for smoother ground movement but since my game revolves around staying off the ground, I need to decide on what penalty to give the player if they are on the ground.

3.2: Penalty System

The design of the “Dynamic Velocity cap” allows me to easily implement this Penalty System. The basis of the penalty system is to encourage them to stay in the air by penalizing their movement when on ground. This was done by adding a few conditions to the Dynamic Velocity cap code.

```
if(!isGrounded && currentVelocity >= maxVelocity - 0.5f)
{
    maxVelocity += Time.deltaTime * 2;
}
else if(currentVelocity <= maxVelocity * 0.9f)
{
    maxVelocity -= Time.deltaTime * 5;
}
```

Fig. 3.3 Penalty system integrated code

The condition I added to the Dynamic velocity cap is that I only gain speed if I am not on the ground. I accomplish this by creating a variable called “isGrounded” which checks whether the player’s GameObject is on the ground or not. If this variable is false and the player’s GameObject velocity is close to the current velocity cap the cap will be allowed to increase.

Chapter 4: In-Air Movement

4.1: Jumping

```
if (Input.GetAxis("Jump") == 1 && isGrounded){
    rb.AddForce(Vector3.up * jumpForce);
}
```

Fig. 3.1 Jumping Code

There are three main components required to make the player GameObject jump. I am going to refer to Fig. 4.1 for the explanation. First, we need to make sure if the “Spacebar” has been pressed. This will be the input from the player. I achieve by that by using “Input.GetAxis(“Jump”) == 1”.

Then alongside of checking whether the “Spacebar” has been pressed we need to make sure if the player’s GameObject is on the ground or not. I want to make sure the player’s GameObject is on the ground because it will be impractical if you can jump midair. This is achieved by using the “isGrounded” variable (described in Section 3.2).

If these two conditions are met at the same time the player’s GameObject is allowed to jump. The way the player’s GameObject jumps is by adding an upward force on the GameObject. The “jumpForce” variable allows me to fine tune how strong the upward force should be.

4.2: Grapple Movement

4.2.1: Overview

Fig. 4.2 and Fig.4.3 show a basic overview on what Grapple Movement consists of. The player GameObject shoots a rope towards a surface and the rope anchors on the surface if hit. Then the player GameObject can swing using the rope to gain momentum and achieve higher speeds (Section 3.2).

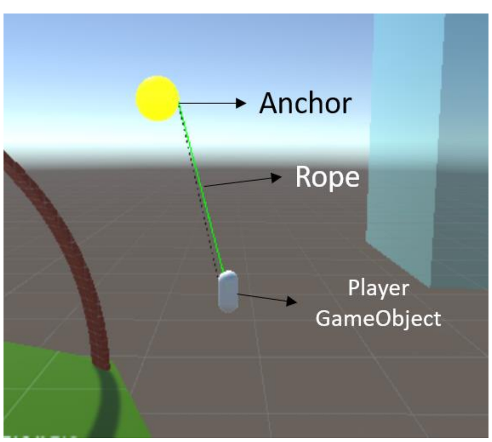

Fig. 4.2 In-Game Screenshot of Grappling

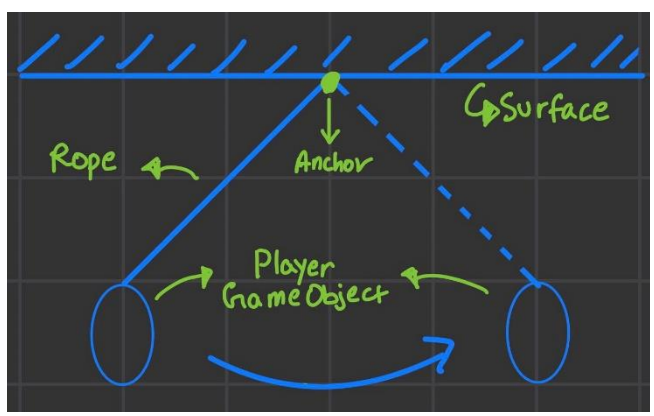

Fig. 4.3 Grappling Overview

4.2.2: RayCast

To check if the player GameObject is looking at a valid surface I use a built-in function called "**Physics.RayCast**" which takes in parameters ***"(origin, direction, hitInfo, maxDistance, layerMask)***". This function casts a ray, from the point ***origin***, in the direction ***direction***, of length ***maxDistance***, against all colliders in the Scene (Section 2.1.2). If this ray hits a valid surface of a GameObject which is determined by ***layermask*** it stores that GameObject's information in ***hitInfo***. This function returns a boolean (Returns true when the ray hits a valid surface of a GameObject, otherwise false).

4.2.3: Spring Joint Component

To simulate connecting the player GameObject with a rope to another valid GameObject in the scene, I use the SpringJoint component.

The **SpringJoint** component joins two GameObjects together as though they were connected by a spring. The spring acts like a piece of elastic that tries to pull the two anchor points (The player and the other GameObject) together.

The strength of the pull is proportional to the current distance between the points with the distance set by the ***spring*** property. This determines the strength of the force.

To prevent the spring from oscillating endlessly you can set a ***damper*** value that reduces the spring force in proportion to the relative speed between the two objects. This describes the amount that the spring force is reduced when active. The higher the value, the more quickly the swinging stops.

The ***minDistance*** and ***maxDistance*** values allow you to set a distance range over which the spring will not apply any force. This will allow the player GameObject a small amount of independent movement but then pull the player GameObject towards the other GameObject when the distance between them gets too large.

The ***autoConfigureConnectedAnchor*** property decides if UNITY should calculate the position of the connected anchor point automatically. I set this property to false since I want the player to be able to decide where to begin the swing.

Since I don't let UNITY calculate the position of the anchor point, I must specify where to place the anchor. I do this by using the ***connectedAnchor*** property. This property stores the position of the GameObject that the player wants to begin the swing from and creates the second anchor on that GameObject.

These are all the properties I used to create a basic simulation of an object swinging from another object using a rope.

4.2.4: Implementation

```
void GrappleManager()
{
    if (Input.GetKey(KeyCode.Mouse0) && !isGrappling){
        if(Physics.Raycast(transform.position, cam.transform.forward, out grappleHitInfo, grappleRange, swingMask))
        {
            grappleHitPos = grappleHitInfo.point;
            StartGrapple();
        }
    }
    else
    {
        if(Input.GetKeyUp(KeyCode.Mouse0) && isGrappling)
        {
            StopGrapple();
        }
    }
}
```

Fig. 4.2 Managing the grappling system

To control how the Grappling System works I created a function called "GrappleManager" which puts together all the working parts of the Grappling System. I did this to create an organized system where I can look at each part of the system independently and accurately determine if there are any issues in those parts.

To explain how the "GrappleManager" function works I will be referring to Fig 2.4. There are two main parts of the Grappling System, "StartGrapple()" and "StopGrapple()". These functions describe "what happens when the player GameObject starts grappling" and "what happens when the player GameObject stops grappling" respectively. But to Start/Stop grappling there are a few conditions that need to be fulfilled. These conditions are contained in the if-else statement in Fig. 2.4.

To **Start Grappling** the player must not be currently grappling, this check is handled by the variable "isGrappling" and the player must click and hold the left mouse button, this button check is handled by "Input.GetKey(KeyCode.Mouse0)" which returns true while the player holds down the left mouse button.

But since we want the player to grapple on specific GameObjects within a specified range of the player there needs to be a condition that checks that. I do this with the help of "Physics.RayCast" function (Section 4.2.2). This RayCast sends a ray out with the origin transform.position (the player GameObject position) and the direction "cam.tranform.forward" (the direction the player is looking) and with length "grappleRange". If this ray hits a GameObject with layerMask "swingMask" it stores that objects information in "grappleHitInfo". If the RayCast function returns true, I store the valid swingable GameObjects position in a variable called "grappleHitPos" and I proceed with the "StartGrapple()" function otherwise I don't do anything.

To **Stop Grappling** the player should have let go of the left mouse button and should have been already grappling. If both these conditions are true then I proceed with the "StopGrapple()" function.

```
void StartGrapple()
{
    grappleJoint = gameObject.AddComponent<SpringJoint>();
    grappleJoint.autoConfigureConnectedAnchor = false;
    grappleJoint.connectedAnchor = grappleHitPos;

    float distanceFromAnchor = Vector3.Distance(transform.position, grappleHitPos);

    grappleJoint.maxDistance = distanceFromAnchor * 0.8f;
    grappleJoint.minDistance = distanceFromAnchor * 0.25f;

    grappleJoint.spring = springiness;
    grappleJoint.damper = damper;

    isGrappling = true;
}
```

Fig. 4.3 StartGrapple() Implementation

The “StartGrapple()” function basically creates the SpringJoint component on the player GameObject and fills in the values of the properties I mentioned in Section 4.2.3.

Looking at Fig. 4.5, I start by adding a “SpringJoint” component to the player GameObject using “gameObject.AddComponent<SpringJoint>()”.

Next I set the “autoConfigureAnchor” property to false since I want to specify where the anchor should be located. Using the “connectedAnchor” property I set the anchor position to the “grappleHitPos” variable which I described earlier. This position is going to be the anchor the player GameObject is going to swing from.

To set the “maxDistance” and “minDistance” properties I first calculate the distance between the player GameObject and the anchor. This is stored in the “distanceFromAnchor” variable. Using this distance, I set the “maxDistance” value to be around 80% of the distance from the anchor and the “minDistance” value to be around 25% of the distance from the anchor.

The “spring” and the “damper” values are set values that I found by trial and error.

Lastly, I set the variable “isGrappling” to true so that I can use that in the “GrappleManager” function.

```
void StopGrapple()
{
    isGrappling = false;
    Destroy(grappleJoint);
}
```

Fig. 4.4 StopGrapple() implementation

The “StopGrapple()” function is a simple function that sets “isGrappling” to false and Destroys the “SpringJoint” component on the player GameObject.

With this approach I have fully implemented a “Grappling System” in the game.

Chapter 5: Movement System Results

5.1: Surprising Benefit

A benefit to this system is that it prevents the player's GameObject getting stuck in other objects. In other words, this system behaves very well against colliders.

This is because the player's GameObject's position is not being affected by any part of my code, all I am changing is how much force can be applied on the player's GameObject and how fast can the GameObject go.

If the code was affecting the position, the player's GameObject's position may get set to a value such that the value is close to or equal to the position of another GameObject like a wall essentially getting the player's GameObject stuck.

Since, my code does not affect the position, if the GameObject hits a wall it will just bounce off it.

5.2: Results

By combining various methods of movement, I have created a physics-based movement system in UNITY which relies on forces.

Since this movement system relies on forces, I don't have to worry about other physical aspects in the game (hitting a wall, getting hit by an object, etc.) breaking or stopping the players momentum suddenly.

I plan to create Third person action hack and slash type game (refers to a type of gameplay that emphasizes combat with melee-based weapons such as swords or blades.) and I plan to integrate this movement system in that game.

Chapter 6: Discussions

Even though this paper is primarily focused on the movement system aspect of the game I wanted to discuss some things I have been thinking about implementing in other games I create.

6.1: Level Design

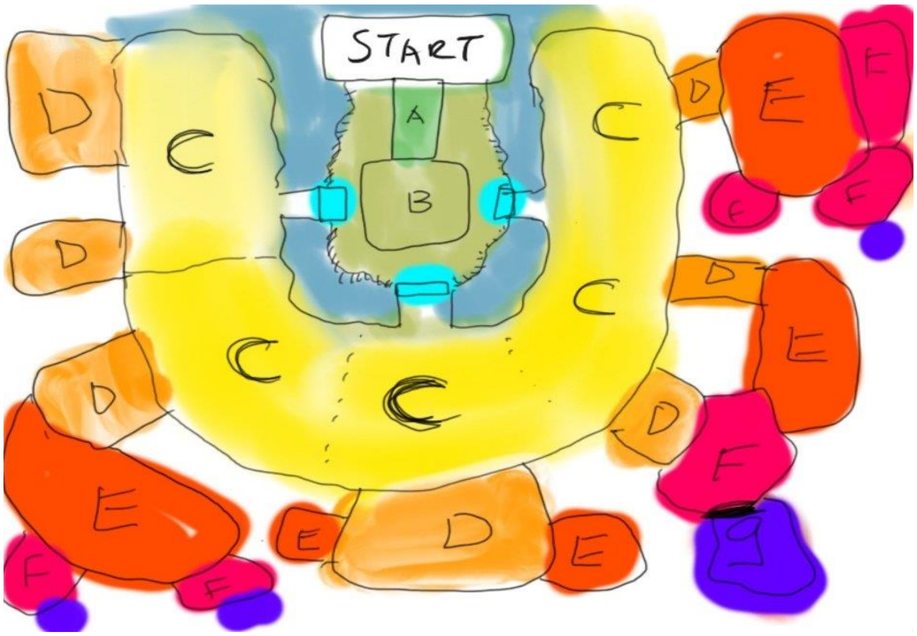

Fig. 6.1 Difficulty Scaling

Fig. 6.1 represents an idea on how difficulty scaling would work in a game. The difficulty is the lowest at section “A” and the highest at section “G”. This map design still has a progression system from A to G but does not force players to be stuck in a particular difficulty section.

If a Player want’s a challenge, then they can progress all the way to zone G but if they want to have fun or relax a bit and enjoy the game, they can go back to an easier zone like zone C.

Chapter 7: Download and Play

7.1: Link on Itch.io

Itch.io is a website for users to host, sell and download indie games. I created a page for my game which can be accessed using the following link:

https://ishanpqr.itch.io/dont-stop-moving

The instructions on how to download the game are also present on that page.

References

Catlike Coding. (2022, 03 26). Retrieved from https://catlikecoding.com/

Espíndola, F. (2022). *The Unity Shader Bible.* Jettelly.

Unity user manual 2020.3 (LTS). (2022, 03 26). Retrieved from Unity: https://docs.unity3d.com/Manual/index.html