Spring 2024

Unity Two Dimensional C# Game

Nick Zajac
naz14@uakron.edu

Follow this and additional works at: https://ideaexchange.uakron.edu/honors_research_projects

Part of the Computer and Systems Architecture Commons

Please take a moment to share how this work helps you through this survey. Your feedback will be important as we plan further development of our repository.

Recommended Citation
Zajac, Nick, "Unity Two Dimensional C# Game" (2024). Williams Honors College, Honors Research Projects. 1787.
https://ideaexchange.uakron.edu/honors_research_projects/1787

This Dissertation/Thesis is brought to you for free and open access by The Dr. Gary B. and Pamela S. Williams Honors College at IdeaExchange@UAkron, the institutional repository of The University of Akron in Akron, Ohio, USA. It has been accepted for inclusion in Williams Honors College, Honors Research Projects by an authorized administrator of IdeaExchange@UAkron. For more information, please contact mjon@uakron.edu, uapress@uakron.edu.
Honors Research Project: Unity C# Game

Computer Information Systems – Programming

Nick Zajac
**Introduction**

Entertainment is something that is always changing, and that change is parallel to the change in technology. Eventually, technology had advanced to a point where video games could enter the homes of millions of Americans. The goal of this project was to develop a game that could be played on windows systems and requires nothing besides a mouse and a keyboard. Through multiple sprints over the semester, I was able to develop the game in my senior design project class. The product is a game called “Mini Souls”. It features multiple levels, gameplay that adds more concepts onto itself with each level, and a challenge that will keep players returning to the game.

**Business Problem**

With entertainment always changing, people are looking for new forms of such entertainment. Video games have been around for a long time, and their evolution has brought about a huge industry and market for video game consumption. Inside this market, people are looking for video games for a multitude of reasons. Some reasons are for personal use, a gift for someone who would play the game, collection, or to be used in secondary markets such as streaming. Overall, video games not only have left a huge impact on the entertainment industry, but also the technology industry as well.

**Technology Stack**

This project required a few tools to be used. The first one is the Unity 2D Environment. This engine handles game objects, scenes, components for the objects, and running the game in general. This environment’s utility comes from being able to design the game in all aspects that can be seen or heard by the player. The second main tool I used was Visual Studio 2022. This
tool is used for many programming projects, mainly due to its high versatility. It can create databases, programs in multiple languages, and much more. For this project, it is mainly used to create scripts that are to be used by the Unity Engine. It is also used to alter, create, and remove lines of code from said scripts that are going to be used in the game as well.

Features Implemented

Sprint 1 – Basic Setup and Player Movement

Importing Assets

A few things were needed before I started development on the game. The first is assets that can be put in the game. To find open-source assets that are free for everyone to use, I used itch.io and the unity store. They provide free assets that anyone can use as well as paid assets that must be purchased before having the right to use them. Once sprites were acquired, it was time to convert them into a sprite sheet and format them correctly. In unity, this can be done through the inspector shown below.
This is where sprite’s properties can be altered. The important things to change are the sprite mode, filter mode, and the pixels per unit. If an image has multiple sprites within it, the sprite mode must be set to multiple. The pixels per unit is how many pixels are meant to fit within one square unit that is inside Unity’s grid feature. This differs between sprites, but it is important to set this properly to allow the sprites to be their proper size. The filter mode is how the sprites are to be rendered. Normally, Unity applies a smoothing filter to make certain objects look neat. However, for the sprite style that is the opposite of idea, so setting the filter to “Point” really allows the images to look sharper and cleaner. After changing these settings, it is then important to open the sprite editor tab to set up the sprite sheet to play animations later on.

Below is what happens when the sprite editor is opened.
In the top left is the sprite editor tab, which is used to “slice” the sheet into individual sprites. It is best to use the “automatic” feature so that Unity can make the slices uniform. After this, simply drag the image the sprite sheet uses on the scene to have it appear.

Creating Tiles and an Interactable Environment

To create an environment the player can interact in, a grid and tilemap must be created. After importing a tilemap object, it is important to add components to the object. Unity has built in components called “Tilemap Collider 2D” and “Composite Collider 2D” that must be added to the object.
After that, simply import a sprite sheet that has the ideal environment sprites and perform the same steps as done with the player object. However, it must be inserted into the Tile Palette tool to be considered a tile. From here, the environment can be created.
**Player Movement**

After creating the tiles, putting them in the scene, and adding the player object to the scene, this is the scene manager will look like:

![Scene Manager](image)

Before starting to code the player movement, the player object requires components as well. The important ones are the “Script”, “Rigidbody 2D”, “Box Collider 2D”, and the “Audio Source” components. The “Script” component allows the object to use C# scripts. The “Rigidbody 2D” component allows the object to experience Unity’s built-in gravity engine that can be changed in here. The “Box Collider 2D” component allows the object to collide with other objects such as the environment or other objects. The “Audio Source” component allows the object to play sounds that the user will hear. The picture below is what the result will look like:

![Player Movement](image)
After importing the proper components, scripts can be created, and code can be written.

In Unity, scripts can be created that are in the C# language, and automatically import Unity’s
base libraries. An example of a blank script is shown below:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class blank : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

There are two main functions that are also automatically generated. One is the start function which runs at the start of the scene being loaded, and the other is the update function. This function runs every frame update and is very similar to main functions in other coding formats. Inside the update function is where movement is written. The code I wrote for the
movement is shown below:

```csharp
// Update is called once per frame
void Update()
{
    /*----movement and animation----*/
    float horizontal = Input.GetAxisRaw("Horizontal");

    movement = new Vector3(horizontal, 0f, 0f);

    //setting float for animation
    ani.SetFloat("Speed", Mathf.Abs(horizontal));
    transform.Translate(movement * speed * Time.deltaTime);
}
```

This code is very simple yet effective. The float horizontal takes the direction that the player is going to move, and the movement variable, defined earlier in the script, creates it as a vector3 variable. A vector3 holds directions in both the x, y, and z directions. The line starting with “ani” is for animation which will be explained later. The line “transform.translate” is what actually moves the player object, and calculates the direction, speed, and accepts a time variable. The “speed” variable is a variable that is initialized earlier in the script and “Time.deltaTime” is a line that ensures it is consistent through the process.

Another important aspect of movement is flipping the sprite in the direction the player is moving, so that the player will always be running towards the direction that is being pressed. To do that, we are using this function below:
These functions serve an important purpose. While the variable “isRight” may seem redundant, it is actually going to be used for other functions later and is a convenient way to store the direction that the player is moving.

**Player jumping**

Jumping is another form of movement that I implemented during this sprint. Having the player jump was coded in the picture below:

```csharp
//flipping character based on horizontal
if (horizontal < 0)
{
    facingRight = false;
}
else if (horizontal > 0)
{
    facingRight = true;
}

if (facingRight == false)
{
    sr.flipX = true;
}
else if (facingRight == true)
{
    sr.flipX = false;
}
```

However, this has some issues. Conditions must be set to make sure that the player cannot jump whenever they want. To do this, the variable “isGrounded” is created. It is used in
These functions, “OnCollisionEnter2D” and “OnCollisionExit2D” are called by Unity whenever the collider 2D component collides with something and when it stops colliding with something. This prevents the player from jumping while they are in the air. It compares itself to the parameter (“other”) which is anything the player interacts with. If it has the tag “ground” then it will change the variable.

**Animation**

Animation is important to game development, as it allows game objects to look like they are doing the action that the player is trying to perform. For this to work, the object requires the “animation” component. This allows for the storage of animations for an object and decides which animations should be played. Before we set animations, we actually need to create them in the “animation” tab. To perform this, first create an empty animation. Then select sprites and drag them into the animation tab. From there, space the frames out and save the animation. Here
is what the animation tab should look like:

After creating the animations open the “animator” tab. From here you can create parameters for the transitions and set the transitions. Parameters are similar to variables, but they can be the reason a certain animation is played. The arrows represent a transition that can happen. The blocks represent an animation. The orange block represents the default animation, which is the idle animation. The blue block is the “any state” block. This block can represent any animation that can transition from any existing animation. Below is a picture of the completed
For the parameters to be changed, they must be changed in scripts as well. Here is a function that demonstrates how animations can be changed in a script:

```csharp
//object variables
public Animator ani;
public Transform AttackPoint;
private Rigidbody2D rb;
private SpriteRenderer sr;

void Start()
{
    currentHealth = maxHealth;
    Debug.Log(currentHealth);
    rb = GetComponent<Rigidbody2D>();
    sr = GetComponent<SpriteRenderer>();
    ani = GetComponent<Animator>();
    audiosource = GetComponent<AudioSource>();
}
```
First, the animator object must be declared and public in the beginning of the script.

Then in the start function, the “GetComponent” command must be used to acquire the information sent by the Unity Engine. Later on, the animator object, is called “ani” in this file, is called. It is very similar to calling and changing the value of a variable normally, however it is for a parameter inside the animator.

Using this information, it is possible to make it so that the player can perform a running animation, a jumping animation, and a falling animation based on certain conditions. Also, the player can perform these movements as key inputs are pressed.

**Sprint 2 – Ground Enemy Object and Basic Combat**

After creating a player object, movement, and basic animations, it was time to start combat. I started by finding an open-source enemy sprite sheet and performing the above steps to
create animations for the object. From there, the object is put into the scene. It is important to give the ground enemy object the same components so it can interact with the environment the same way that the player can. However, it is important to give it different scripts so that the object is not controlled at the same time that the player object is.

**Attacking – The Basics: The Player**

Creating combat was not an easy task for this game. It involves the use of functions, layers, and running delays. Similar to the jump function, it can be called in the update function, by checking if a key is pressed. Below is the function that is being called by the player:

```csharp
/*----- Attacking Enemies -----*/
void Attack()
{
    audioSource.clip = attack;
    audioSource.Play();
    //play attack animation
    ani.SetTrigger("Attack");

    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(AttackPoint.position, attackRange, enemyLayers);

    //damage the enemies
    foreach (Collider2D enemy in hitEnemies)
    {
        //calls the hurt function from the health manager script
        enemy.GetComponent<DamageScript>().TakeDamage(attackDamage, enemy);
        Debug.Log("We hit " + enemy.name);
    }
}
```

The beginning of the function uses audio sources, which will be explained later. The function calls a trigger for the animator to start the attack animation. Then, an array is created that stores every object that is in a certain layer, inside a certain range, starting from a certain point. This point is set in unity as an empty object, and the range is set at the start of the function. In unity, the empty object and range can be drawn to look like the picture below:
After storing the objects that are being hurt, they call a function in another script that manages health. This was developed in a later sprint and will be covered later. Originally, the health value was managed in each individual script but this created issues in later sprints. The variable “attackDamage” is also set at the beginning of the script.

**Attacking – The Basics: Ground Enemy**

Originally, the ground enemy ran the same function as the player but checked specifically for the player layer. It has a slightly smaller attack range, but it is made in the same way that the player range is programmed. It was also based on a key press for testing purposes. This led to the function seeming complete later on when automating the attack process. Below is Unity’s drawing of the range for the enemy object’s attack point:
At this point, major issues were about to arise. These functions were simply tested based on two things: there was only one enemy, and that the enemy would only run through the function once. However, for what was created at the time, it was functional.

**Taking Damage**

Taking damage is a process that can be troublesome if not done correctly. The first thing that should be done is set the animation parameter to play the specific animation. Then, after playing the sound, change the health amount to the new health amount. Below is the picture of this function:
This function, while being in the enemy script, is almost the exact same for the player script. At this point, the enemy object requires a little more to keep the enemies balanced, and not overbearing the player. The function also has a call to a “Die” function if the health is less than zero. The enemy needs a delay, so that way time passes and allows the animation to play and to give the player time to react to the enemy taking damage. To run a delay, the following line is written. This is called in the update function so that the object does not perform other actions during the delay.

```
StartCoroutine(hurtDelay());
```

This line calls the delay “hurtDelay”. Delays in C# work much like functions, however they can implement delays. The “hurtDelay” function is shown below:

```
/*-------Delays-------*/
IEnumerator hurtDelay()
{
    rb.velocity = Vector2.zero;
    ani.SetFloat("Speed", 0);
    yield return new WaitForSeconds(delayTime);
    ishurt = false;
}
```

This delay function sets the movement to zero, set the animator parameter for movement off, runs the delay for the stored amount in a variable, and set the Boolean “ishurt” to false. This variable is meant as a way to keep the object from performing other actions while they are running the hurt function.
**Death – The Player and The Ground Enemy**

It is common for video games that when an enemy or a player’s health reaches 0, they are defeated. This requires functions and delays for both the player and the enemy objects. The function is called from the hurt function when the specific object takes enough damage. Earlier, the hurt function for the enemy object was shown. Below is the death function that is called in said hurt function:

```csharp
/*-----Death function-----*/
1 reference
void Die()
{
    EKBody.velocity = Vector2.zero;
    Debug.Log("Enemy Died");
    //die animation
    ani.SetBool("isDead", true);
    isAlive = false;

    GetComponent<Collider2D>().enabled = false;
    StartCoroutine(deathDelay());
}
```

It is setting the enemy’s movement to zero in case the enemy would have started chasing the player, setting an animation parameter to play the specific animation, making sure the object will not chase after the player by setting a Boolean called “isAlive” to false. It disables the
collider 2D component and then calls a delay so the object will remain on the ground for a period of time. The “deathDelay” delay function is shown below:

```
IEnumerator deathDelay()
{
    yield return new WaitForSeconds(deathTime);
    Destroy(objectToDisable);
}
```

This delay is very simple. After running the delay time, the object is disabled and removed from the scene. The “objectToDisable” object is set in unity, and it is set to itself.

This is also very similar to the player function, with only minor changes. While both objects run functions that are almost identical, it is important to run them in separate scripts dedicated to the specific object. This makes sure that one object does not accidentally use information such as variables, functions, or object addresses that it should not have access to.

**Sprint 3 – Flying Enemy and Following the Player**

In order to provide a challenge, video games require variety. Not only was another enemy type added, but code was written for both enemy types to chase after the player as well. During this time some issues showed their faces when trying to program combat for multiple enemies and allowing them to take damage from the player object.
**Flying Enemy – Movement**

To create the flying enemy the same steps for the player object and the ground enemy object were performed. The flying enemy has its own unique script, since the movement and attack ranges vary when compared to the ground enemy. Since the enemy is flying, it requires only an idle, attack, hurt, and death functions. Below is the flying enemy’s appearance on the testing environment for the game:

![Flying Enemy Image](image)

Inside the flying enemy’s unique script, movement is based on the fact that it can freely move against gravity. It requires certain conditions to move, as well as certain conditions to attack the player. Below is the function that is used to determine the direction the player if
facing, as well as moving towards the player in order to attack them:

```csharp
void Update()
{
    // calculating the direction towards the player
    Vector2 direction = (Vector2)player.position - (Vector2)transform.position;
    // storing direction's x value into a temp variable to use in the facing right function
    batDirection = direction.x;
    direction.Normalize();

    distance = Vector2.Distance(player.position, transform.position);

    if (distance > 1 && isAlive == true && distance < viewRange)
    {
        // face the enemy towards the player
        if (isHurt == false) //
        {
            StartCoroutine(HurtDelay());
        }
        //Debug.Log("Moving!");
    }
    if (distance < 1 && isAlive == true && isAttack == false)
    {
        isAttack = true;
        Attack();
        //Debug.Log("Attacking!");
    }
}
```

This function has many variables. It uses a vector2 variable to determine the direction that the enemy must face in order to look at the player. It stores the x coordinate from that vector in a variable before normalizing it to use for movement later on. The variable “batDirection” allows the sprite to flip directions similar to the player function displayed earlier. The function uses the built-in distance calculator from Unity to determine the distance between the enemy object and the player object, and then is used to determine if the object should move towards the player or if the object should attack the player. However, before moving it must check if a Boolean is false, to make sure it is not currently taking damage. It also checks to make sure that the object is alive, not currently attacking already, and it is within a certain distance before attacking the player.
**Flying Enemy – Death function**

The death function is worth noting here as well. The only slight variation is that instead of disabling the collider and destroying the object after a certain amount of time, it is movement based. Below is a part of the death function:

```csharp
/**--Death function----*/
1 reference
void Die()
{
    isAlive = false;
    Debug.Log("Enemy Died");
    //die animation
    ani.SetBool("isDead", true);

    GetComponent<Collider2D>().enabled = false;
    batBody.velocity = Vector2.down * fallspeed;
}
```

This is only part of the complete death function. Note how the current vector is set to 0, the Boolean that the previous function is based on is set to false, and a new vector is set in place to make the object fall downwards at a certain speed “fallspeed”. This would mean that the enemy falls infinitely, however in the update function there is this statement:

```csharp
/**--checks to see if the object has gone out of bounds---*/
//check to see if past out of bounds range, then destroys and disables this script
if (transform.position.y < 0.08)
{
    Destroy(objToDie);
    this.enabled = false;
}
```
This function, which is called every frame, checks the y coordinate of the object to make sure it is above a certain value, which is defined in the variable “OOB” or “out-of-bounds”. Once again, the destroy line is used, with “objToDie” set to itself within Unity. While these differences might not seem like much, they greatly affect how the flying enemy’s movement differs from the ground enemy’s movement.

**Ground Enemy – Movement**

The ground enemy’s movement function will look very similar to the flying enemy’s, except there is are slight differences. The function is shown below:

```csharp
//calculating the direction towards the player
Vector2 direction = (Vector2)player.position - (Vector2)transform.position;
//storing direction's x value into a temp variable to use in the facing left function
ENDirection = direction.x;
direction.Normalize();

distance = Vector2.Distance(player.position, transform.position);

/*/-----Moving and Attacking--------*/
if (distance > 1 && isAlive == true && distance < viewRange) {
    //move the enemy towards the player
    if (ishurt == false) {
        //if not hurt, move towards the player
        ani.SetFloat("Speed", 1);
        ENBody.velocity = direction * speed;
    } else {
        StartCoroutine(hurtDelay());
    }
} else {
    //otherwise do not move
    rb.velocity = Vector2.zero;
    ani.SetFloat("Speed", 0);
}

//attack if the player is within range
if (distance < 1.2 && isAlive == true && isAtk == false) {
    isAtk = true;
    Attack();
```
The basics of the function are still here: it gets the direction the object needs to face, it gets the distance between the player and the object and moves based on how close the player is to the object. However, the main difference lies with where the attacking check is placed. It is placed outside of the movement function in order to attack the player even if it does not need to move. The flying enemy was planned on being placed in positions where it would have to chase down the player, while the ground enemy might not need to since they are required to be on platforms like the player. This is the complete version of the movement that includes the “hurtDelay” line that was shown earlier.

**Health Manager – A Major Rework**

As previously stated, functions involving receiving damage from the player had worked when only 1 enemy existed. The player simply called a function inside the enemy script which was made public. The player script could not access any variables, it could only give a parameter which was the amount of damage the function was to calculate with. The same goes for the ground enemy, except for the player’s unique function. Once this issue emerged, a serious rework was required. A brand-new script, called “damage_script” was created and added to all existing objects. It was meant to count as a health manager and would change its calculations based on which object called the function. The main function is shown below:
After reading this function, one thing becomes clear: what about the player’s health?

When the enemies only have to attack one player, adding a whole section to the health manager felt unnecessary, since the player’s personal health manager worked just fine. This creates a distinct change in the enemy’s and player’s attack functions. The first picture below is the

```csharp
// Start is called before the first frame update

private int batHealth = 0;
private int EKnightHealth = 0;
private int BossHealth = 0;

void Start()
{
    // allows for use with other scripts
    Bat_Script batSc = GetComponent<Bat_Script>();
    Enemy_Script enSc = GetComponent<Enemy_Script>();
    Boss_Script bossSc = GetComponent<Boss_Script>();
}

/*---- new takeDamage, replaced individual functions----*/

public void TakeDamage(int damage, Collider2D caller)
{
    if (caller.GetComponent<Bat_Script>() != null)
    {
        batHealth -= damage;
        caller.GetComponent<Bat_Script>().hurt(batHealth);
    }
    else if (caller.GetComponent<Enemy_Script>() != null)
    {
        EKnightHealth -= damage;
        caller.GetComponent<Enemy_Script>().hurt(EKnightHealth);
    }
    else if (caller.GetComponent<Boss_Script>() != null)
    {
        BossHealth -= damage;
        caller.GetComponent<Boss_Script>().hurt(BossHealth);
    }
```
player’s function, and the second is the flying enemy’s:

```csharp
void Attack()
{
    // play attack animation
    ani.SetTrigger("Attack");

    // detecting if enemies are in range of the attack, and stores them all in a variable
    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(AttackPoint.position, attackRange, enemyLayers);

    // damage the enemies
    foreach (Collider2D enemy in hitEnemies)
    {
        // calls the hurt function from the health manager script
        enemy.GetComponent<Damage_script>().TakeDamage(attackDamage, enemy);
    }
}
```

```csharp
void Attack()
{
    // play attack animation
    ani.SetTrigger("attack");
    audiosource.clip = attack;
    audiosource.Play();

    // detecting if the player is in range of the attack, and stores them all in a variable
    Collider2D[] hitPlayer = Physics2D.OverlapCircleAll(AttackPoint.position, attackRange, playerLayer);

    // damage the enemies
    foreach (Collider2D enemy in hitPlayer)
    {
        enemy.GetComponent<Player_Script>().TakeDamage(attackDamage);
        Debug.Log("Enemy hit ", + enemy.name);
    }
    StartCoroutine(athDelay());
}
```
Two more things were required for the health manager to function: keeping calculations up to date with the actual values and being able to send them back to the proper object. In the previous image showing the health manager’s function based on what object called the function, a line is shown where it returns the new health back to the specific object through a “hurt” function. On the object’s end, it is merely for playing animations, checking if the object is alive, and updating to its new health value. In order to make sure the health values the health manager stores are correct, it stores a reset health function for each enemy type that is called by each of the enemies when they first load into the scene. The pictures below show the flying enemy object calling the reset function, as well as the manager’s function of resetting it’s internal values:

```csharp
void Start()
{
    //getting the component for the rigidbody
    batBody = GetComponent<Rigidbody2D>();

    //getting animator object
    ani = GetComponent<Animator>();
    //sprite component
    sr = GetComponent<SpriteRenderer>();
    //getting audio compoenenet
    audiosource = GetComponent<AudioSource>();
    //creating reference to the health manager
    damage_script damageScript = GetComponent<damage_script>();
    //restting the health in the health manager
    damageScript.ResetBathealth(maxHealth);
}
```
Sprint 4 – The Dash and The Boss

The Dash

Having only offensive options for the player seemed like it might be a bit unfair, especially with the enemies being able to chase down the player. This and a personal enjoyment of movement-based abilities inspired the idea for the player to be able to dash. This dash is a quick burst of speed that can allow for avoiding attacks and moving to far off platforms. The function below is the dash function that is called when a key is pressed, and no other actions are being performed:

```java
/*---resetting health functions, called by each object----*/
1 reference
public void ResetBathealth(int batInput)
{
    batHealth = batInput;
}
1 reference
public void ResetEkhealth(int EKInput)
{
    EKnightsHealth = EKInput;
}
1 reference
public void ResetBosshealth(int bossInput)
{
    BossHealth = bossInput;
}
```
After being called, the function sets animation parameters that allow it to player and determine when to stop playing the animation, freezes the player’s y coordinate, ignores collision with enemy attacks, and moves at a fast speed depending on the direction the player is facing.
Afterwards, a delay function is called. This function is shown below:

```csharp
IEnumerator DashDelay()
{
    yield return new WaitForSeconds(dashDelayTime);

    //reactivates collision with enemy attacks and player layer
    Physics2D.IgnoreLayerCollision(playerLayerNum, rangeLayerNum, false);
    rb.constraints = RigidbodyConstraints2D.FreezePositionY;
    ani.SetBool("isDash", false);
    rb.velocity = Vector2.zero;
    isDash = false;
}
```

This function allows the speed boost to only be active for a certain time, before resetting the animation parameters, y coordinate freezing, speed, and the variable that prevents other actions from being performed. Overall, this ended up not only being a very useful tool to give players, but a very fun tool that makes movement interesting and engaging.

**The Final Boss – Part 1**

Typically, at the end of video games a powerful enemy challenges the player’s skills known as “the final boss”. I wanted such a challenge that forced players not just to learn the mechanics of the game but master them. To test this foe, a separate test room was created similarly to the previous test room, but only contained the boss and the player. The test room is shown below:
After being integrated to the health manager (seen in the health manager section of the previous sprint) and being given the same components as the other objects for basic functioning, the boss was ready to be programmed. When it came down to designing the boss’ attack patterns I wanted actions that it would randomly select between and have a period after a certain number of actions where it would stay still and be open for player attacks. During this sprint, only one attack was finished, mainly due to time constraints. The main function is shown below:
void Update()
{
    if (isAttck == false)
    {
        // calculating the direction towards the player
        direction = (Vector2)player.position - (Vector2)transform.position;
        playerDirection = direction.x;
    }
    // Vector2 b_location = new Vector2 rb.GetCenterPosition;

    // storing location before teleport in order to return to this position after the attack
    if (isTeleport == false)
    {
        b_location = transform.position;
        Vector2 p_location = (Vector2)player.position;
    }

    if (playerDirection < 0)
    {
        facingLeft = true;
        sr.flipX = true;
    }
    else if (playerDirection > 0)
    {
        facingLeft = false;
        sr.flipX = false;
    }
    if (isAlive == true & actionCount < 3 & & isDash == false & & isTeleport == false)
    {
        // Debug.Log(actionCount);
        StartCoroutine(actionDelay());

        // if action less than 3, the boss will teleport otherwise he will dash
        if (actionChoice < 3)
        {
            Teleport(transform.position, player.position);
        }
        if (actionChoice > 3)
        {
            dash(movement);
            actionCount++;
        }
    }

    if (isDash == true)
    {
        attackCollide(col);
    }
}
Note that the function is based on a few parameters. While there are multiple parameters, the one to take note of is how “actionCount” must be less than 3. The function requires it to generate a random number from 1 to 6, and this is calculated in the function shown below, which is outside of the update function:

```c
int generateRandomAction()
{
    // generating a random number from 1 to 6 so it has more chances to switch between the actions
    int randomNumber = UnityEngine.Random.Range(1, 6);
    // Debug.Log("Random Number: "+ randomNumber);
    return randomNumber;
}
```

After this is generated, the function calls a certain attack function that is either the teleport or the dash attack. Since the dash attack was only completed in this sprint it will be covered here. This function is very similar to the player’s dash, however it’s dash is lethal and can damage the player while the boss is moving. To perform this, the dash function is called which is shown below, but also the Boolean variable “isDash” is used.
“isDash” is used in the update function to call the function “attackCollide”. This function acts very similarly to the normal attack function that enemies have. However, one important thing is different. To make this a constant check, and not one single one, it must be called in the update function repeatedly. “col” is the name set to the variable storing the reference to the collider 2D object that is set in Unity. This is done in the update function with this code:
At the end of the dash function, a delay is called, which is shown below:

```csharp
IEnumerator DashDelay()
{
    yield return new WaitForSeconds(dashDelayTime);
    rb.velocity = Vector2.zero;
    isDash = false;
    GetComponent<Collider2D>().enabled = true;
    rb.constraints = RigidbodyConstraints2D.FreezePositionY;

    actionCount++;

    if (actionCount >= 3)
    {
        Debug.Log("on cooldown!");
        StartCoroutine(CDDelay());
    }
}
```
This function has an important addition to the end of it. After resetting all the restrictions set during the function after the delay, it also checks to see if the number of actions being performed has exceeded the maximum amount. If it does, then it runs a separate delay inside of itself known as “CDDelay” which is the main cooldown delay. This delay features a delay for an amount of time set in the private variable “coolDownDelay” and resets the action count, allowing for the pattern to reset.

```csharp
//delay for action cool down
private IEnumerator CDDelay()
{
    yield return new WaitForSeconds(coolDownDelay);
    actionCount = 0;
}
```

Overall, the idea that the final challenge is a mix of previous enemies is not only shown in how it attacks the player but can be seen in the code as well. While functions can be reused, they often require changes that are needed to fit the specific situation.

**Sprint 5 – Boss Second Attack, Level Design, and Sound Design**

**Final Boss – Second Attack**

In this sprint, the boss’ attack pattern is completed with the function that allows it to teleport. In short, the boss will play teleport animations, move towards the player, attack, and then return to its original position. In code, this was broken down in three major sections. These are moving towards the player, attacking the player, and returning to their original position.
Final Boss Attack - Teleport

Before being called, the script stores the boss’ initial location. In the if statement, as long as “isTeleport” is false, it will not store the current location. This allows the location to be updated until that information is needed, and it will not change.

```csharp
//storing location before teleport in order to return to this position after the attack
if (isTeleport == false)
{
    b_location = transform.position;
    Vector2 p_location = (Vector2)player.position;
}
```

After the teleport function is called, this is the function that is ran:

```csharp
private void Teleport(Vector2 bossPosition, Vector2 playerPosition)
{
    isTeleport = true;
    isAtk = true;

    //creates new xy coordinate of the target location
    Vector2 bossTemp = new Vector2(playerPosition.x - 1.5f, bossPosition.y);

    StartCoroutine(TPtoPlayerDelay(bossTemp));
    // Debug.Log("done teleporting");
}
```

This function takes two parameters, the boss’s position, and the player’s position. It starts by setting two variables that prevent other actions from being performed, and then creates a new vector2 variable that stores the player’s location but slightly to the left. This allows the boss to
move close to the player, but not right on top of them. It then calls a delay function called “TPtoPlayerDelay” while taking the target location as a parameter. The delay runs as followed:

```csharp
IEnumerator TPtoPlayerDelay(Vector2 tpLocation)
{
    audiosource.Stop();
    audiosource.clip = tpSound;
    audiosource.Play();

    ani.SetTrigger("disappear");
    yield return new WaitForSeconds(preTPDelay);
    objToDie.GetComponent<Renderer>().enabled = false;
    transform.position = tpLocation;
    objToDie.GetComponent<Renderer>().enabled = true;
    ani.SetTrigger("reappear");
    yield return new WaitForSeconds(preTPDelay);
    Attack();
}
```

The delay plays the sounds necessary, plays animations with delays in between them, and then moves towards the target position stored in the parameter “tpLocation”. After moving, it calls the attack function. The attack function then runs as seen below:

```csharp
/*--------Attacking player---*/

void Attack()
{
    isAtk = true;
    //play attack animation
    ani.SetTrigger("attack");
    StartCoroutine(atkDelay());

    //attack is ran in delay
}
```
After running this attack function, it starts an attack delay. From here, a delay function is called inside an already running delay function. This is due to how Unity handles delay functions. If two delay functions are called, it will call them at the same time. Therefore, in order to have them run sequentially one must be called at the end of the original delay function. This looks as followed:

```
IEnumerator atkDelay()
{
    yield return new WaitForSeconds(atkDelayTime);
    audiosource.clip = attackSound;
    audiosource.Play();
    if (facingLeft == true && direction.x < 1 || facingLeft == false && direction.x > 1)
    {
        Collider2D[] hitplayer = Physics2D.OverlapCircleAll(AttackPoint.position, attackRange, playerLayer);
        //damage the player
        foreach (Collider2D enemy in hitplayer)
        {
            //run player hurt function
            enemy.GetComponent<Player_Script>().TakeDamage(attackDamage);
        }
    }
    //wait a bit after attack
    yield return new WaitForSeconds(atkDelayTime);
    isAtk = false;
    //runs final delay for teleport with og location as parameter
    StartCoroutine(ReturnDelay(b_location));
}
```
The return teleport function looks very similar to the first teleport function, however it has a few new additions. After returning to the original location, stored in the parameter “location”, it checks to see if the number of actions that it has taken has exceeded the maximum amount allowed without a cooldown just like at the end of the dash delay. This also helps maintain this attack pattern that will keep the game going.
**Level Design**

The game has been broken down into 3 levels and a start screen. This first level is relatively easy which is intended to help the player adjust to the controls and learn as they move. The second level provides more of a movement challenge and some tricky enemy placements, and the third is the boss arena where the showdown with the boss happens. Thanks to the environment setup from Sprint 1, the only new additions are the background assets. Below are the screenshots of every major scene that will be encountered by the player:

![Title Screen Screenshot](image)

The title screen features 2 clickable buttons and the logo of the game. It also provides a short list of the controls and information the player will want to know before starting the game. At the top is the logo I have designed by myself. Both buttons have the following script attached to them:
The reason both of these functions can be stored in the same script is due to the buttons only being able to call on function at a time. The function that the button calls is set in unity, so having them in the same script is not harmful at all. Here is what this looks like for the start button in Unity.
Above is Unity’s editor view of the first level. The white box on the left is the camera, and for this game it does not move vertically, so everything below it is not seen by the player. On the right side, there is a blue portal object that exists to act as a transition between levels. The portal has a script attached to it that takes a public variable from unity that allows it to change to a new level. This is shown below:
The “sceneBuildIndex” variable is set by Unity, but how is that number determined? This can be done but altering the scenes that can be loaded in the build settings. The index is shown on the right in the window below:

To change to the next scene, the script simply takes the value of the next index after being set in Unity. Below is the picture for the second level as well:
This is very similar to the first level, with slightly more challenging platforming. The portal in this level takes the index of the final level, and colliding with it takes the player to the final level:

**Level Design – Winning, Losing, and Quitting**

At any point in the game, the player can lose by being defeated, or falling out of bounds by falling off of a platform. When the player object runs the death function after losing health, it makes the retry button and exit button appear, which function the exact same as the start button
and exit button from the title screen, although they look slightly different. The object calls a function from a script that has two major functions and is called “tryAgainMovementScript”. This script allows the camera to follow the player if it is bound to the camera, and stores functions to make the retry and exit button appear. This script is shown below:
The “button” public variable stores the current button the script is attached to, if it is attached to a button. It then disables the button making it unusable until it is called back from the player script calling the enable functions. The try again button and exit button then have this script attached to it to allow it to be re-enabled. The buttons still have the functions stored from the script that starts the game over or exits the game, thanks to being able to store more than one script on an object.

When the player defeats the boss, it performs a death function similar to how the player calls a death function. However, it is a special version. There is a third button called the win button that is hidden in the final level where the boss resides. If the boss is defeated, it calls a version of the script that summons the win button as well as the try again and exit buttons. This script is shown below:

```csharp
IEnumerator deathDelay()
{
    yield return new WaitForSeconds(deathDelayTime);
    winButton.GetComponent<winShowScript>().winShow(winButton);
    Destroy(objToDie);
}
```
These buttons can seem repetitive, but it is important to let the player choose to replay or quit the game as much as possible without interfering with core gameplay. This also reduces the chance of players using other means to close the game like task manager that could cause unexpected errors. From here, adding sound to the game is required and then the game is complete.
Game Design – Audio

Sound is something that can really help keep the player engaged with a game. Whether it is music or sound effects it is a level of game design that cannot be ignored. Thankfully, adding audio in Unity is quite simple. Just like adding a script component, make sure to add the “Audio Source” component to every object that will be playing a sound. The main camera by default has the “Audio Listener” component attached already, which will listen to sounds from game objects. In order to have music and sound effects, another listener can be added as an empty object to the level. Since the game is two dimensional, the audio will not weaken as the player moves farther from the object. From here, in each script create public “AudioSource” variables that can store each audio file that is intended to be played.

```csharp
//sound variables
private AudioSource audiosource;
public AudioClip jump;
public AudioClip attack;
public AudioClip dash;
public AudioClip hurt;
```

In order to play a sound effect in C#, simply write the following lines:

```csharp
audiosource.Stop();
audiosource.clip = jump;
audiosource.Play();
GetComponent<Rigidbody2D>().velocity = new Vector2(0f, jumpForce);
```

For the audio listener that is playing music, open it in Unity and have it specifically play the audio file that it needs to play. This way it will “listen” to only that audio file and will not interfere with the code.
With this, the game is complete. It is able to be ran from build settings in the Unity Editor, or from the application location if it has been previously built. While a majority of my ideas were implemented, there were some that I had intended to implement but could not for various reasons.

**Backlog**

Some ideas I had were based around level design and combat. I originally wanted to give the player multiple ways to attack enemies, but due to my limited knowledge on C# and Unity I felt that it would take more time to program than was given for the project. The game play would have come down to a rock-paper-scissors style of combat where certain abilities might be better than others depending on the situation. Another idea that never worked out was for the boss to have another attack option. This was not implemented mainly due to lack of time as well. In the future, I would like to implement these concepts and add additional levels that could bring new concepts that would further challenge the player and bring new concepts.

**Conclusion**

This project was a big challenge, it used one environment I was vaguely familiar with and one I was completely new to using. Thankfully, a majority of the issues I had ran into were due to my lack of knowledge with Unity, not programming as a whole. It was important to keep in
mind that each time a section of code was not working it was an opportunity to learn and understand a brand-new environment that I had not touched before. It taught me the importance of keeping information secure and making it so data cannot be altered by every program it touches. If a variable was public and was not supposed to be, Unity can mess up what data is stored in that variable and further break the code. I also learned the importance of writing comments to keep track of functions, variables, and objects. In classes, when projects are around one hundred lines or less it is very easy to still keep track of things. However, when coding a project that averages one hundred – two hundred lines per script, keeping notes and keeping track of everything is especially important. Overall, while this project was a big challenge and required me to learn a lot, I am immensely proud of how it turned out.

Sources used:
https://docs.unity3d.com/Manual/Unity2D.html

https://docs.unity3d.com/Manual/Sprites.html

https://docs.unity3d.com/Manual/Tilemap.html

https://docs.unity3d.com/Manual/Audio.html

https://docs.unity3d.com/Manual/ScriptingSection.html

https://www.youtube.com/watch?v=XtQMytORBmM&pp=ygUZY29tcGxlGgYmVnaW5uZXIgdW5pdHkyZA%3D%3D