Lauryn Wade

GIMM 110

Prof. Ellertson

Nov 2nd 2023

E-Pluribus Contrarus Rhetorical Analysis

Creating this game has been an interesting test of what I want to implement versus what is possible to implement, and the struggle with technology and using what I have already done to start over. There were ideas I had that wouldn't work, some that got removed that I had to replace, and even sudden changes to the game's mechanics that had to be made in order to even make any progress. When first designing the visual elements of my game, I wanted to use designs and visual concepts I had already created prior to beginning the game. I first drew my idle animation to begin the entire process, to at least establish the resolution of the game and the size of the game. I opted to have all sprites then forward to be 16 pixel resolution. After I implemented my base sprites and minimal animation, I opted to use AI generation for one of my backgrounds and some of my bricks. The process of which I drew a simplified version of what I wanted, then had an image generator make it pixelated and larger.

In terms of the code that I put into my game, I went more traditional with the features I put in. First of which, is a sprint function:

```
void Move()
```

```
{
    float currentMoveSpeed = Input.GetKey(KeyCode.LeftShift) ? moveSpeed * 3 : moveSpeed;
    Vector3 movement = new Vector3(Input.GetAxis("Horizontal"), 0f, 0f);
    transform.position += movement * Time.deltaTime * currentMoveSpeed;
}
```

This was really easy to implement once I was told what in Unity's library can affect movement speed. I was inspired by other games that use a sprint function, and I did not want my game to feel sluggish. I used get keycode shift as the trigger for the method, then everything else came through.

The next feature I implemented was a bit more complicated, which was an attack. The primary version of my attack function featured a health system, which is one I also showed in my conference. Unfortunately, due to low disk space, a lot of my code was changed, removed, or corrupted concerning a majority of my health systems. This challenge allowed me to create a solution however that saved a lot of space on my memory, and a lot of space on the page of Visual Studio.

```
 public float attackRange = 2.0f;
 public LayerMask enemyLayer;
 int bossHits = 0;
 void Update()
 {
```

```
    if (Input.GetKeyDown(KeyCode.Q))
    {
        Collider2D[] hitColliders = Physics2D.OverlapBoxAll((Vector2)transform.position, new
Vector2(attackRange, attackRange), 0f, enemyLayer);

        foreach (Collider2D collider in hitColliders)
        {
            if (collider.CompareTag("enemy"))
            {
                Destroy(collider.gameObject);
            }
            else if (collider.CompareTag("Boss"))
            {
                bossHits++; // Increment the bossHits counter
                Debug.Log("Boss hit! Hit count: " + bossHits);

                if (bossHits >= 5)
                {
                    Destroy(collider.gameObject); // Destroy the boss if hit 5 times
                    Debug.Log("Boss defeated!");
                }
            }
        }
```

```
    }
```

I opted for the button for attack be Q. I based this button completely to be along the lines of games like Overwatch or League of Legends. This system was a bit of a last minute pivot, since the memory on my device is low, sometimes non-existent script errors occur or scripts are deleting random lines of code that affect multiple factors. To have the attack be in all one place, without having the code spread between three game objects, I had to make some sudden adjustments and I got rid of the health system, and implemented a number of hits system just like this one. The player has a similar function as the attack script, where as, after the player collides with an enemy three times, the game will restart at the beginning menu.

```
 public int maxHits = 3;
 private int hitCount = 0;
 private Camera mainCamera;
 private float flashDuration = 0.1f;
 private Color originalColor;


 private void Start()
 {
    mainCamera = Camera.main;
    originalColor = mainCamera.backgroundColor;
 }
 private void OnCollisionEnter2D(Collision2D collision)
 {
    if (collision.gameObject.CompareTag("enemy"))
```

```
    {
        FlashScreen();
        hitCount++;
        if (hitCount >= maxHits)
        {
            SceneManager.LoadScene(2);
        }
    }
}
```

I declared how many hits the player could take before the game starts (where it is underlined), then if the threshold of collision is met, I can successfully create a system that punishes the player if they get hit, while saving space and time only in just two total scripts establishing a health system. I used this again to also keep track of how many game objects with certain tags were destroyed, once all tag “enemy”s are destroyed, the game will return to the main menu as well. Tags have been the most useful in putting together my game and having components interact with each other. Most of the interactions in my game are from making sure everything is tagged, labeled, and layered correctly. The reason why I eventually landed on using Unity’s tag functionalities instead of declarations in my code is because it was a lot safer and secure for me. Dealing with the scripts not being completely safe on my hardware, it was really easy to see and pick up where I left off, or fill in blanks on the Unity engine itself, or to easily identify what is missing or disconnected from other components in my code.

Another big aspect of the game, while not the focus, was challenging, were my moving platforms at the bottom of the map. The “targets” I set for them to move simply did not work. My platforms ended up jittering back and forth but not following any set points. Eventually I did get two of my three platforms to move correctly at the bottom of the map. Once they were moving correctly, I was able to make the player’s position a child of the platform's movement.

```
private void OnCollisionEnter2D(Collision2D collision)
{
   if (collision.gameObject.name=="Angel")
   {
      collision.transform.SetParent(transform);
   }
}

private void OnCollisionExit2D(Collision2D collision)
{
   if (collision.gameObject.name=="Angel")
   {
      collision.transform.SetParent(null);
   }
}
```

What I have shown is while on enter- meaning when the player collides with and ground checks the platform, they will become a child of the platform and the platform’s movement. Then I have

another method that will allow the player to move independently again upon exiting the platform.

Despite the struggle and intense changes that needed to be made to each of the previous mechanics above, I was able to implement something simple yet fun into my game that ended up working out right from the start. That mechanic is the use of an enemy AI that follows the player. I wanted to make the enemies more than just objects or projectiles on a set path, I wanted them to follow the player and make avoiding collision with them a bit harder.

```
public float detectionRange = 5.0f;
public Transform player;
private bool isPlayerInRange = false;
private SpriteRenderer spriteRenderer;
void Start()
{
    spriteRenderer = GetComponent<SpriteRenderer>();
}

void Update()
{
    float distanceToPlayer = Vector2.Distance(transform.position, player.position);
    isPlayerInRange = distanceToPlayer <= detectionRange;
    if (isPlayerInRange)
    {
```

```
        Vector2 directionToPlayer = (player.position - transform.position).normalized;
        directionToPlayer.y = 0f;
        transform.Translate(directionToPlayer * moveSpeed * Time.deltaTime);
}
```

Essentially I declared a detection range that allowed there to then be a number of units the enemy will check before following the player. Since the player could be followed forever, even outside of range if the cycle starts once, there is also a bool that ensures the enemy will stop pursuit after the player has left its range. The float distance to player, further establishes where the player is in-game comparatively to the enemy (detection range) and allows it to move, by transforming its position. On its own, this code applies to not only the x axis, but also up and down. Since this code in particular defies gravity for the game object, simply setting the enemy game object to move along only the X axis immediately fixes the issue.

The reason why I chose these scripts to look at in particular, is because these scripts had a lot of components that I borrowed from them to put into smaller supplemental scripts. There was a lot that I copy-pasted from these that allowed other code to work and function in my game. I think these are the most important aspects of the game, the first to be noticed in game play, and also the first to make the game not functional if any of them were not working.