Sam Lummus

GIMM 110

Dr. Ellertson

October 31, 2022

Individual Game Rhetorical Essay

This individual game project was initially stressful, being thrown into the deep end with no prior experience in the production of such stuff. I stuck by my personal work ethic and principles to see this through. I did not want to quit before I really even attempted it. I was worrying about the potential future the project held for me, but I just needed to sit down and start chipping away at the work. Small goals make up the larger whole, and that is what I sought to do. My biggest focus for this game was level design, and making it difficult to beat. My biggest inspiration was Jump King, which was the basis for one of my four physic solves, the charge jump.

The charge jump was chosen as my core game mechanic because whereas I wanted to make a 2D platformer, I wanted it to be difficult to beat. Choosing a more peculiar jumping mechanic would give my player something to master while playing my game. The charge jump works off the idea that for as long as the jump key (in this case W) is held, a jump value will increase. The hard cap of the jump value is 10f, as to limit how high my player can jump. If it reaches that value without the jump key being released, two local variables are created to temporarily lock the player's movement, the y movement is just the jump value of 10f, whereas the x movement is virtually zero as there is no move input or move speed occurring. When the player is forced to jump, the if-then statement invokes a ResetJump method, to ensure that the jump value is reset once my player lands back on the ground.

```
if (jumpValue >= 10f && isGrounded)
          {
              float tempx = moveInput * moveSpeed;
              float tempy = jumpValue;
              rb.velocity = new Vector2(tempx, tempy);
              Invoke("ResetJump", 0.2f);
          }
```
ResetJump consists of a switching a “canJump” variable to false to ensure no air jumping is possible, and resetting the jump value back to zero. If the player holds down a movement key (in this case A for left, D for right) and releases the jump key, they player will jump in that direction. However, the player’s movement is locked until they land back on ground again. This requires the player to make accurate jumps, with no chance to strafe in midair to correct themselves in case they go sailing into a hazard or enemy.

```
if (Input.GetButtonUp("Jump"))
        {
            if (isGrounded)
            {
                if (Input.GetAxis("Horizontal") > 0)
                {
                    rb.velocity = new Vector2(4f, jumpValue);
                }
                else if (Input.GetAxis("Horizontal") < 0)
                {
                    rb.velocity = new Vector2(-4f, jumpValue);
                }
                jumpValue = 0.0f;
                Vector3 movement = new Vector3(Input.GetAxis("Horizontal"), 0f, 0f);
                transform.position += movement * Time.deltaTime * moveSpeed;
            }
            canJump = true;
        }
```

Another physics solve I included in my game to give it some more challenge was an enemy follow script, which was quite simple actually. It consists of creating a transform variable named target that upon start would find my player object with a similar tag name, “Player”. In the update method, the enemy to which the script is attached too will start to go towards my player by transforming the enemy's position, to my player's position, based on the public speed variable I created times Time.deltaTime.

```
void Start()
  {
    target = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>();
  }

  // Update is called once per frame
  void Update()
  {
    transform.position = Vector2.MoveTowards(transform.position, target.position, speed *
Time.deltaTime);
  }
```

My moving platforms and enemies follow the same script for scripted path movements. To start, I created an array to hold my waypoints (or other similar gameObjects) as well as a speed to which my platform or enemy would move. These are both serialized fields so I could edit their speed and waypoints that they would go to and from in the inspector of unity editor. An if statement then says if the distance between the current waypoint index and the next waypoint element is virtually zero, the current waypoint index increases by one. If the current waypoint index exceeds the length of the array, it goes back to the beginning of the array, zero. For a simple right to left movement or up and down movement of platforms or enemies, this is ideal for switching the movement between index zero and one.

```
{
    if (Vector2.Distance(waypoints[currentWaypointIndex].transform.position,
transform.position) < .1f)
    {
      currentWaypointIndex++;
      if (currentWaypointIndex >= waypoints.Length)
      {
        currentWaypointIndex = 0;
      }
    }
    transform.position = Vector2.MoveTowards(transform.position,
waypoints[currentWaypointIndex].transform.position, Time.deltaTime * speed);
```

```
    }
```

My last physics solve was falling platforms, to challenge my players even more by forcing them to be even more precise with their jumps when the ground on which they stand falls away. A problem I ran into is that if a platform fell and was destroyed, it would cut off my player from advancing if they couldn't make the jump initially. This led me to discover Instantiating, which is essentially making a clone of a prefab I have created, retaining the position and other elements of that prefab.

```
Instantiate(FallingPlatform, new Vector2(-0.56f, 1.76f), FallingPlatform.transform.rotation);
```

A CoRoutine called SpawnPlatform is created that will recreate the platform that has fallen after the player has collided with its 2D Box Collider. When the platform falls, it does so after the value of variable falldelay passes, and destroys the gameObject after the value of variable destroydelay passes.

```
private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            PlatformManager.Instance.StartCoroutine("SpawnPlatform", new
Vector2(transform.position.x, transform.position.y));
            StartCoroutine(Fall());
        }
    }


private IEnumerator Fall()
    {
        yield return new WaitForSeconds(fallDelay);
        rb.bodyType = RigidbodyType2D.Dynamic;
        Destroy(gameObject, destroyDelay);
    }
```

For the scoring of my game, multiple collectibles would be gathered by my player, leading to the score counter increasing by two or even three. An easy fix for this would be creating a simple collected boolean variable (initially fales) that would switch to true when my player triggered the collectible box collider. Once that bool statement switched to true, it would destroy the collectible so my player could no longer interact with it on the next updated frame.

```
private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Coin") && collected == false)
        {
            ScoreCounter.coinAmount += 1;
            collected = true;
            Destroy(collision.gameObject);
        }
    }
```

With my collectibles (and for the sake of my audience's sanity) increasing my score counter, I opted for a checkpoint system to save myself and my audience the grief of having to restart the scene when they died. If I restarted the scene, my collectibles would also respawn, leading to theoretically infinite score. To do so, I created a transform variable called current checkpoint, so I could store a respawn point for my player once they collided with the in-game object's box collider. Once collided with, the checkpoint's box collider is disabled, as to not cause problems in case my player somehow made it back to a previous checkpoint and essentially progressed backwards.

```
private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.transform.tag == "Checkpoint")
        {
            currentCheckpoint = collision.transform;
            collision.GetComponent<Collider2D>().enabled = false;
            Debug.Log("checkpoint reached");
        }
    }
```

Upon my character’s death, the end of the death animation will run my respawn method, which essentially reverts my player character back to their original state, while also moving them to the last checkpoint they interacted with.

```
public void Respawn()
   {
      transform.position = currentCheckpoint.position;
      rb.bodyType = RigidbodyType2D.Dynamic;
      GetComponent<Collider2D>().enabled = true;
      anim.Play("Joe_Idle");
   }
```

For being my first proper experience coding, designing levels, and doing artwork for a game, it was a real challenge for me, juggling my classes alongside it. I am satisfied with the fruits of my labor, and glad that I stuck with this project, and the GIMM major as a whole. I feel more prepared than I was a month a half ago when this project was dropped on my lap. I can’t wait to see what other things this major is going to have me do, and I hope they help me see a path for my future career and endeavors.