Grant Olson

Professor Ellertson

GIMM 110

2 November 2022

Chasing Shadows,

Or, How I designed The Cave

In The Cave, you play as a little creature named Cosmo as he solves platforming puzzles to reach the "light at the end of the tunnel". The idea for this game came from the ancient philosopher Plato's *Allegory of the Cave,* an excerpt from his manifesto, Republic. My biggest goal for this game was to provide the player with this "lightbulb moment" where the things that may have confused them at the beginning of the game start to make more sense. For the game to be most effective, the player should reach an understanding that the character they play as is simply a shadow of the actual Cosmo, and all of the obstacles they faced, they could have simply walked past if they understood this truth at the beginning of the game.

This is where I started running into issues, as adding this moment to the game as a feature didn't seem to work, no matter how I twisted the rules of this simulation I created. As such, in the final product, I decided to end the show in the moment before this truth is fully discovered, leaving the player to interpret the ending how they so choose.

With the story now mostly explained in writing, I'd like to explain how I approached the game chronologically, because this shows the value I placed on each feature. My greatest takeaway from this project was that a game can always be improved, but deadlines determine how in depth the project goes.

At the very beginning of my journey, I began to develop art that supports the aesthetic of the world in my head. The only art in this project that I did not personally create was the death animation. I got this from a free asset pack I found on Unity's asset store. Besides that, every asset in this game was made in Aseprite, a pixel art production tool that I learned about from a peer in my GIMM 100 class. The most challenging aspect of asset creation was creating a tiled background that seamlessly tesselated. I did this by using Aseprites tile feature which extends the borders of the asset in the x and y directions so you can see what the asset will look like when repeated in these directions.

In tandem with working on my assets, I created the CavePlayerMovement.cs script, a large endaeavour that brought comfortable physics to Cosmo's world and determined how he would move. This Script mainly defines how cosmo jumps, runs, and falls with famous tricks from the past utilised. For instance, rather than setting one Jump velocity, my script determines the height that Cosmo jumps based on how long the jump button is held, small taps result in little hops, and a long press lets Cosmo "hang" in the air for a defined time. Falling back to the ground is different too. If Cosmo does a small hop, he is defined to have a smaller fall multiplier than if he jumps higher, this makes the impact of jumps feel more like other games player's have been exposed to (The first ever Super Mario being a great example of a game with this mechanic).

The script doesn't just focus on noticeable features. One of the most complicated parts of this script is determining drag. That is, if Cosmo is on the ground, than he should have a different friction than he would have if he is in the air. To do this, I defined two methods, one for ApplyGroundLinearDrag(), and another for ApplyAirLinearDrag(). Here are a couple screenshots:

```
private void FixedUpdate()
{
    CheckCollision();
    H_Movement();

    if (isGrounded)
    {
        jumps = resetJumps;
        ApplyGroundLinearDrag();
    }
    else if (onWallLeft || onWallRight)
    {
        ApplyAirLinearDrag();
        FallMultiplier();
    }
    else
    {
        ApplyAirLinearDrag();
        FallMultiplier();
    }
}
```

```
private void ApplyGroundLinearDrag()
{
    if (Mathf.Abs(dirX) < 0.2f || changingDirection)
    {
        rb.drag = groundLinearDrag;
    }
    else
    {
        rb.drag = 0;
    }
}

//Applies "extra" gravity to make jump feel less floaty
private void ApplyAirLinearDrag()
{
    {
        rb.drag = airLinearDrag;
    }
}
```

In the ApplyGroundLinearDrag() Method, I check if the player is holding left or right on their controller and if they are changing directions. If they aren't, then the game applies a drag force against the direction they were moving to simulate friction. Changing directions is defined best in my comments here:

```
//Snappier direction change.
//true if rigidbody is moving right and the horizontal input is less than 0 (left key is being pressed)
//or
//true if rigidbody is moving left and the horizontal input is greater than 0 (right key is being pressed)
private bool changingDirection => (rb.velocity.x > 0f && dirX < 0f) || (rb.velocity.x < 0f && dirX > 0f);
```

In the ApplyAirLinearDragMethod(), we simply apply the drag if the method is called. Now to address FixedUpdate(). FixedUpdate() is a base method in Unity's dictionary that is used for game physics because it runs at a fixed interval independent of the frame rate. Here, I check if Cosmo is grounded and if they are, the game calls the ApplyGroundLinearDrag() method. If Cosmo is next to a wall, the game calls the ApplyAirLinearDragMethod() as well as the FallMultiplier() method.

Over time, this movement script got more and more complex, but this is a good look at some of the basics. Most of the ideas in this player movement script were found through research and my biggest hurdle was understanding how exactly they worked so that I could put them

together. But with this unique game idea, I quickly left the realm of researchable code and had to do my own problem solving.

Easily, the most important feature of The Cave is the Shadow GameObject. Here's the idea; I wanted to create a reason for the player to be able to double jump. This came in the form of an "otherworldly" diamond collectible that when the player touches, breaks the barrier between Cosmo and his understanding of the truth for a quick second. In game terms, this means that if you collide with a diamond, as the diamond disappears, the Shadow appears, the camera focuses on it, and you are granted an extra jump despite not touching the ground.

In order to make this work, I created a child game object of the player which I named and tagged as Shadow, and gave it an Animator that references the player's animation script and copies their movement states. Simple enough. Next was the problem of turning it on/off and assigning cameras. This next script will be best explained accompanied by the code:

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

public class ShadowVisualization : MonoBehaviour
// Script which turns off the shadow and causes it to reappear if the player touches a diamond.
// Then the shadow is always front and center of camera.
{
    private SpriteRenderer sprite;
    private Animator anim;
    public bool hasLight;
    [SerializeField] private CinemachineBrain brain;
    private int lowPriority;
    private int highPriority;
    [SerializeField] private CinemachineVirtualCamera cam1;
    [SerializeField] private CinemachineVirtualCamera cam2;



    // Start is called before the first frame update. Using it to grab parts of shadow and define priority for use in methods.
    void Start()
    {

        sprite = transform.GetChild(0).GetComponent<SpriteRenderer>();
        anim = transform.GetChild(0).GetComponent<Animator>();
        hasLight = false;
        sprite.enabled = false;
        anim.enabled = false;
        lowPriority = 1;
        highPriority = lowPriority + 1;
        cam1.Priority  = highPriority;
        cam2.Priority = lowPriority;

    }

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

    // Method that is used to check if the player has collected "the light at the end of the tunnel"
    public void LightCheck()
    {
        if (hasLight == true)
        {
            sprite.enabled = true;
            anim.enabled = true;

        }
    }
```

For this to work, I found a unity package for smart camera movement, called Cinemachine. I then created two virtual cameras which have a built in priority setting. This setting determines which camera is used during gameplay. One is following the player, the other is following the Shadow. Next, I needed to define a low priority and high priority variable that I could move between the cameras. Easy, add one to whatever low priority is, and we get high priority. Then, because at the end of the game I have the player reach the “light”, I define a method that turns on the shadow indefinitely if a bool, hasLight, is true. Now for the collision code:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Void"))
    {
        cam2.Priority = lowPriority;
        cam1.Priority = highPriority;
    }
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Diamond") && hasLight != true)
    {
        sprite.enabled = true;
        anim.enabled = true;
        cam1.Priority = lowPriority;
        cam2.Priority = highPriority;

        StartCoroutine("JumpScare");
    }
    if (collision.gameObject.CompareTag("Light"))
    {
        hasLight = true;
        cam1.Priority = lowPriority;
        cam2.Priority = highPriority;
    }
}

IEnumerator JumpScare()
{
    yield return new WaitForSeconds(1);
    sprite.enabled = false;
    anim.enabled = false;
    cam2.Priority = lowPriority;
    cam1.Priority = highPriority;
}
```

Here I've set up "if loops" that check if the player collides with an object, which camera and what shadow parts should turn on and off. The most notable thing here is the diamond collison. First, I have to ensure that if the player hasLight, that they don't loose it by colliding with a diamond. This is done by only checking the collision with a diamond if the player hasn't collided with the light object and thus hasLight. Next, if we collide with a diamond, we want to switch cameras and see the shadow for a small moment. This is done with a Couroutine, which lets me use a timer inside the function to turn the shadow back off and switch the cameras back to their original state after the timer finishes. Pretty neat huh?

Finally, we have to address player death. In order to create a scoring system, I decided to create a checkpoint system so that the player doesn't have to oneshot every obstacle in the game in order to reach the end. (some of them are hard!) To do this, I have yet another script on the player character. Here it is:

```
public class playerDeath : MonoBehaviour
// Script which makes the player unable to move, plays a death animation, and
// resets the player at the last checkpoint the last checkpoint they touched
// resets the player if they have light and die
{
    public GameObject cpParent;
    private Transform cpGrabber;
    private Vector3 spawnPoint;
    public GameObject player;
    private Rigidbody2D rb;
    private Animator anim;
    private SpriteRenderer sprite;
    private int deathCount;

    // Start is called before the first frame update
    private void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
        sprite = GetComponent<SpriteRenderer>();
        deathCount = 0;

        //grabs the position of the starting area
        cpGrabber = cpParent.transform.GetChild(0);
        player.transform.position = cpGrabber.position;

    }

   private void OnCollisionEnter2D(Collision2D collision)
   {
        if (collision.gameObject.CompareTag("Traps"))
        {
            Die();
        }
   }

    // turn the player's movement off and play the death animation
    // add one death to the death counter
   private void Die()
   {
    rb.bodyType = RigidbodyType2D.Static;
    anim.SetTrigger("Death");
    deathCount ++;
   }

    // when colliding with a checkpoint, save the checkpoints tranform in cpGrabber
   private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.CompareTag("Checkpoint"))
        {
            cpGrabber = other.transform;
        }
    }

    //Reset Player and move them to the last checkpoint they touched
    private void Respawn()
    {
        rb.bodyType = RigidbodyType2D.Dynamic;
        anim.ResetTrigger("Death");
        player.transform.position = cpGrabber.position;
        anim.Play("Shadow_Idle");

    }
}
```

First, I created a game object that I called checkpoint and added a box collider 2D set as isTrigger to it. In start, I grab the transform of that checkpoint. Then, if a player collides with a game object tagged as a trap, call the Die() method. Death results in stiff, rigid… Body2D's so I set the type to static and then call the death animation. In the death animation, I call the respawn method after waiting a few moments to let the player regret their actions. The respawn function resets the player and moves them to the last checkpoint they touched. Storing checkpoints that

have been touched is accomplished by setting a grabber variable to the transform of the object that was collided with.

I have really only scratched the surface of the original code I've added to this game, but the deadline to turn this project in is fast approaching, and I've already gone way over the four page goal. I suppose you'll just have to play the game to see the rest of it's features, but I hope this paper has given some behind the scenes insight to how I approached developing a weird game, based on a weird story, from a weird philosopher with unique ideas about the weird world.