Aila Batista
Anthony Ellertson
GIMM 110
4 November 2023

Rhetorical Analysis

In the making of my individual game, I struggled a lot. The programming was certainly difficult, but what made it much harder was the design choices I had to make. I wasn't sure where I wanted to go with my game except that I wanted it to be a platformer. I was able to come up with a simple concept which I used in my animation: a cat that has had a fish it discovered stolen away by a rat. This effectively set up my game so the cat is the playable character while the rat is the enemy/boss that needs to be defeated. I chose to make it a 16 bit pixel game using a limited palette of pastel colors because I wanted to have a soft and cute style. I also felt that it would be more manageable to use this style to create my game within the time constraints that I had.

Now to delve into the code, there are roughly eleven important pieces of code I want to go over. First I want to discuss the inheritance communication that I used. I made a rat class to base the rat enemies on which sets the health and how much damage they take while also containing two other methods, one of which defines the enemy's death and the other defining damage upon collision. At the beginning of this class I created a variable which would hold a numerical value that is the health of the rat and is used in the GetHealth() virtual method below it. The TakeDamage() virtual method contains the code that the health value will go down by the value of damage which is the parameter that this method will accept when called elsewhere. This method also contains an if statement that when the value of health reaches zero or lower it will call the Die() method. This method is a private void so it can not accessible by the child classes to be changed (it can not be overridden) and it destroys the game object which in this case would be the rat that holds this script. Lastly, there is a private OnCollisionEnter2D method that uses data from unity's built in system to detect a collision on the rat. There is an if statement inside that runs if there is a collision with a game object that specifically has the tag "DamageDealer". This is where the value of damage is assigned that will be used in the TakeDamage() method which is called inside this method to be used. For the purposes of my game this was all I needed in order to begin setting up the rat class so that I could have defeatable enemies.

```
public class Rat : MonoBehaviour
{
   public int health = 10;
   public virtual int GetHealth()
   {
      return health;
   }
   public virtual void TakeDamage(int damage)
   {
      health -= damage;
      if (health <= 0)
      {
         Die();
      }
   }
   private void Die()
   {
      Destroy(gameObject);
   }
   private void OnCollisionEnter2D(Collision2D collision)
   {
      if (collision.gameObject.CompareTag("DamageDealer"))
      {
         int damage = 5;
         TakeDamage(damage);
      }
   }
}
```

Then I created two child classes called RatEnemy and SuperRat. RatEnemy has no changes and uses the exact data given from the parent class Rat as shown below in the left column: two override methods of the virtual methods in the parent class calling the base data. Then to the right there is the SuperRat child class that does make changes to the override methods. In the first method GetHealth(), the health of the super rat is changed to 25. Next in the TakeDamage method, there are a couple of things going on. The code is almost identical, except now there is a piece of code using SceneManager to load to a different scene upon the death of the super rat. In order to do this I had to write the code again along with the private method Die, since I would not have been able to get access to the original if statement otherwise.

```
public class RatEnemy : Rat
{
   public override int GetHealth()
   {
      return base.GetHealth();
   }

   public override void TakeDamage(int damage)
   {
      base.TakeDamage(damage);
   }
}
```

```
public class SuperRat : Rat
{
   public override int GetHealth()
   {
      return health = 25;
   }
   public override void TakeDamage(int damage)
   {
      health -= damage;
      if (health <= 0)
      {
         Die();
         SceneManager.LoadScene("End");
      }
   }
   private void Die()
   {
```

```
            Destroy(gameObject);
        }
    }
```

This next piece of code begins some of the physics changes I added to my game. This code is what I wrote to incorporate a double jump. I created a variable named jumpCount which holds a numerical value which is assigned to be 0. This is important because as shown below, the jump method has an if statement with a condition called isGrounded only allowing the player to jump when touching the ground. I did not want to change this because it used code from the platformer stub provided to the class and I wanted to protect the integrity of the stub as much as I could while still making my own additions. So, I assigned this variable in order to add an additional condition that when jumpCount is less than 2 you can still jump. Everytime you jump, the count is increased by one (jumpCount++). But there is also another if statement nested inside the first that when you are grounded, the jumpCount returns to 0. This is another important detail because if the jump count is not reset each time the player is touching the ground, then the double jump may stop working (which is something I struggled with before reaching this result.) The previous code had been written in a way that jumpCount was only reset after the entire code had run including the double jump. The problem with this is that if I did not double jump, the counter would remain at one since it did not reset upon being grounded (only after double jumping.) Once I jumped again, it would go up to two and a double jump could not be performed for the rest of the game.

```
int jumpCount = 0;
void Jump()
{
    if (Input.GetButtonDown("Jump") && (isGrounded || jumpCount < 2))
    {
        if (isGrounded)
        {
            jumpCount = 0;
        }
        rb.AddForce(new Vector2(0f, jumpForce), ForceMode2D.Impulse);
        jumpCount++;
    }
}
```

This code incorporates knockback into my game when the player collides with the rats. Admittedly, this code is very buggy within the game and I still have not gotten around to solving it. To start, there is a private OnCollisionEnter2D method so we can use the collisions detected by unity. There is an if statement inside this method that when the player collides specifically

with a game object that has the tag “RatEnemy”, it will call from the method TakeDamage() assigning the value of damage as 10 within the parentheses. The TakeDamage() method will decrease the value of current health by the value of damage. Then it accesses a method SetHealth() from the healthBar script which is referenced earlier in the script, although not shown here. Inside the parentheses the currentHealth is assigned, the purpose of this being to set the health bar inside the game to display the value of the current health as it is changed from taking damage. Returning back to the if statement in the previous method, there are three pieces of code below where TakeDamage(10) is called. The first line creates a new vector position using the coordinates of the player called playerPosition. The next line creates another vector called knockbackDirection which uses playerPosition and the point of collision in order to calculate the vector. This vector is normalized so it only has a magnitude of 1 which should give the knockback consistent strength no matter how far the distance between the player and point of collision (this part I had to research because I didn’t fully understand, so I am a little uncertain about this information). The last line adds force to the player’s rigidbody and the type of force is defined by ForceMode2D.Impulse which specifies that a sudden force is added player making move in the direction calculated in knockbackDirection with the force of knockbackForce (knockbackForce is assigned earlier in the script that is not shown below).

```
private void OnCollisionEnter2D(Collision2D collision)
{
   if (collision.gameObject.CompareTag ("RatEnemy"))
   {
      TakeDamage(10);
      Vector2 playerPosition = new Vector2(transform.position.x, transform.position.y);
      Vector2 knockbackDirection = (playerPosition - collision.contacts[0].point).normalized;
      rb.AddForce(knockbackDirection * knockbackForce, ForceMode2D.Impulse);
   }
}
void TakeDamage (int damage)
{
   currentHealth -= damage;
   healthBar.SetHealth(currentHealth);
   Debug.Log("Player took " + damage + " damage. Current health: " + currentHealth);
}
```

This code incorporates a dash in my game, but like the knockback code, this is also buggy and works in a different way than I had intended. Instead of dashing forward in whatever direction the player is facing, I have discovered that when I jump the player can dash back down. I like this outcome because I think it creates opportunities in the gameplay where it is necessary to stop the player’s jump and quickly return to the ground so damage from enemies or traps can

be avoided. Right now it doesn't have any particular usefulness but, nonetheless, I will be searching for where I went wrong with my initial goal and how to improve the current state of the code.

The dash code begins with the Dash() method. Inside there is an if statement that when the left shift key is pressed and isDashing is not true, the contents of the statement will be performed. First I will discuss the isDashing variable which is a bool assigned to false earlier in the code that is not shown below. This is done in order to prevent dashing again before the code has been fully executed. In the if statement when both conditions have been met isDashing is set to true for the reason I have specified earlier. Under that, a float variable is declared called dashDirection which is assigned the value of the direction the sprite is facing as dictated by sprite.flipX. The code that follows that, "? -1 : 1" means that if sprite.flipX is true the value of dashDirection will be -1 and when it is false it will be 1. Then the next statement is changing the player's velocity based on the dashDirection and dashSpeed (dashSpeed is a variable assigned earlier in the code which is not shown below). Lastly the coroutine method StopDashing() is called (Which IEnumerator is a coroutine method). This method has the player stop dashing after the condition dashDuration is met (this is another variable assigned earlier in the code which is not shown below). The player velocity on the x axis is then set to zero, stopping it while the y velocity remains unchanged. The variable isDashing is set to false and the player can now dash again. There is a problem with this part because the new vector setting the horizontal velocity to zero is assuming that the player is dashing horizontally and not vertically which is the way it happens to work in my code.

```
private void Dash()
{
    if (Input.GetKeyDown(KeyCode.LeftShift) && !isDashing)
    {
        isDashing = true;
        float dashDirection = sprite.flipX ? -1 : 1;
        rb.velocity = new Vector2(dashDirection * dashSpeed, 0);
        StartCoroutine(StopDashing());
    }
}
private IEnumerator StopDashing()
{
    yield return new WaitForSeconds(dashDuration);
    rb.velocity = new Vector2(0, rb.velocity.y);
    isDashing = false;
}
```

This class for projectiles the player can shoot has a lot to cover. It begins with a series of variables and references to the game object's information. Firstly, projectilePrefab references the game object allowing it to be used within this code. Next firePoint references the position of the game object from which it will be fired. The sprite renderer is also referenced as playerSprite so that the direction the object is facing can be determined. The projectileForce float set at 10 determines the force at which the projectile is shot while projectileLifetime float set at 5 determines the duration the projectile will continue to be shot before being destroyed. There is also a bool canShoot set at true which will be used later to prevent the player from shooting until the code for the first projectile has been completely executed.

After that there is an if statement in the update method which will start the Shoot() method when the left mouse button is clicked and canShoot is true. Directly below that is the Shoot() method which, when called, sets canShoot to false, preventing the player from shooting. Then a new projectile game object is created using the projectilePrefab game object and setting it at the firePoint position. A new vector is also called projectilePosition is then created which is used in the following if statements to change which direction the projectile is shot based on which direction the player is facing. The next set of code looks similar to the previous code, but calls the rigidbody of the game object and applies force to it depending on the direction that is being faced. Lastly, two coroutine methods are called. EnableShooting(1f) sets canShoot to true after a 1 second delay. DestroyProjectileAfterLifetime(newProjectile, projectileLifetime) destroys the newProjectile after a 5 second delay. At the very end I created an OnCollisionEnter2D() method so that the projectile would also be destroyed upon collision with another game object.

```
public class Projectile : MonoBehaviour
{
    public GameObject projectilePrefab;
    public Transform firePoint;
    public float projectileForce = 10f;
    public float projectileLifetime = 5f;
    private bool canShoot = true;
    public SpriteRenderer playerSprite;
    void Update()
    {
        if (Input.GetMouseButtonDown(0) && canShoot)
        {
            Shoot();
        }
    }
    void Shoot()
    {
        canShoot = false;
        GameObject newProjectile = Instantiate(projectilePrefab, firePoint.position, transform.rotation);
```

```
        Vector3 projectilePosition = firePoint.position;
        if (playerSprite.flipX)
        {
            projectilePosition = firePoint.position - transform.right * 1f; // Shift it left.
        }
        else
        {
            projectilePosition = firePoint.position + transform.right * 1f; // Shift it right.
        }
        Rigidbody2D rb = newProjectile.GetComponent<Rigidbody2D>();
        if (rb != null)
        {
            if (playerSprite.flipX)
            {
                rb.AddForce(transform.right * projectileForce, ForceMode2D.Impulse);
            }
            else
            {
                rb.AddForce(transform.right * -projectileForce, ForceMode2D.Impulse);
            }
        }
        StartCoroutine(EnableShooting(1f));
        StartCoroutine(DestroyProjectileAfterLifetime(newProjectile, projectileLifetime));
    }
    IEnumerator EnableShooting(float delay)
    {
        yield return new WaitForSeconds(delay);
        canShoot = true;
    }
    IEnumerator DestroyProjectileAfterLifetime(GameObject projectileToDestroy, float lifetime)
    {
        yield return new WaitForSeconds(lifetime);
        Destroy(projectileToDestroy);
    }
    void OnCollisionEnter2D(Collision2D collision)
{
    Destroy(gameObject);
}
}
```

The code for my moving platform includes two different classes. I borrowed this code from the 2D platformer tutorial that we followed for homework. The first class Waypoint Follower creates a serialized field called waypoints that within the unity engine I am able to apply a separate game object to that can be referenced. A serialized float variable, speed is also created, assigned the value 2, but can be changed within the unity engine. There is also a numerical float variable called currentWaypointIndex which is assigned the value zero and is meant to keep track of the current waypoints index. Then in the update method there is an if statement which checks if the distance from position of the game object and the waypoint is less than .1 units meaning that the game object has reached the waypoint. If so then the currentWaypointIndex will increase by one. Below that is another if statement that will reset the currentWaypointIndex to 0 when the index becomes greater than or equal to the total number of waypoints which will allow it to loop. At the bottom is the movement code which will transform the position of the game object to move back and forth between the waypoints.

The other class is StickyPlatform which has two methods OnTriggerEnter2D() and OnTriggerExit2D. This allows the collider on the platform (for this code I made another collider) to act as a trigger, specifically if it collides with a game object named Player. Once the player collides with the platform it will set the platform as a parent to the player so that they will move together. Then when the player exits and they stop colliding this code is nulled.

```
public class WaypointFollower : MonoBehaviour
{
    [SerializeField] private GameObject[] waypoints;
    private int currentWaypointIndex = 0;
    [SerializeField] private float speed = 2f;
    private void Update()
    {
        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);
    }
}
```

```
public class StickyPlatform : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.name == "Player")
        {
            collision.gameObject.transform.SetParent(transform);
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.name == "Player")
        {
            collision.gameObject.transform.SetParent(null);
        }
    }
}
```

This code is the player death logic I created which I believe is also borrowed from the 2D platformer tutorial but adapted to fit with my health bar code which I also borrowed from a tutorial. It starts by referencing the players rigidbody and assigning it to the variable rb and also referencing the Healthbar script and assigning it to the variable healthBar. Then in the start method the first line of code is looking for an object that has the Healthbar script as a component. In the next line with the if statement when this component is found it subscribes the Die() method to event OnHealthZero in the Healthbar script. Lastly, in the start method the rigidbody component of the game object is grabbed so it can be used. Next there is the Die() method which starts when the health of the player reaches zero. The first line of code sets the rigidbody of the player to static so it can no longer be moved then calls a coroutine method RestartLevelAfterDelay(2f). That method reloads the scene to the very beginning after 2 seconds.

```
public class PlayerLife : MonoBehaviour
{
    private Rigidbody2D rb;
```

```
    private Healthbar healthBar;
    private void Start()
    {
        healthBar = FindObjectOfType<Healthbar>();
        if (healthBar != null)
        {
            healthBar.OnHealthZero += Die;
        }
        rb = GetComponent<Rigidbody2D>();
    }
    private void Die()
    {
        rb.bodyType = RigidbodyType2D.Static;
        StartCoroutine(RestartLevelAfterDelay(2f));
    }
    private IEnumerator RestartLevelAfterDelay(float delay)
    {
        yield return new WaitForSeconds(delay);
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
}
```

The very last thing I will discuss is this code for my scoring system. This code is from the 2D platformer stub provided which I did not change to protect the integrity of the stub and it suited my purpose. The first line of this code references a text component and stores it in a variable called text. Then the next line creates a static variable called coinAmount which means it can be accessed from another script (it is used in a separate script called CoinCollect which is attached to a fish game object that when collided with increases the cointAmount in this script). In the start method it gets the text component and sets it to the variable text so that it can be used in the code. Finally in the update method it sets the text component to the value of coinAmount. The ".ToString()" converts the integer to a string so it can be displayed as text.

```
public class ScoreCounter : MonoBehaviour
{
    Text text;
    public static int coinAmount;
    void Start()
    {
        text = GetComponent<Text>();
    }
    void Update()
    {
        text.text = coinAmount.ToString();
    }
}
```
