# Create – Applications From Ideas
# Written Response Submission Template

Please see Assessment Overview and Performance Task Directions for Student for the task directions and recommended word counts.

**Program Purpose and Development**

2a)

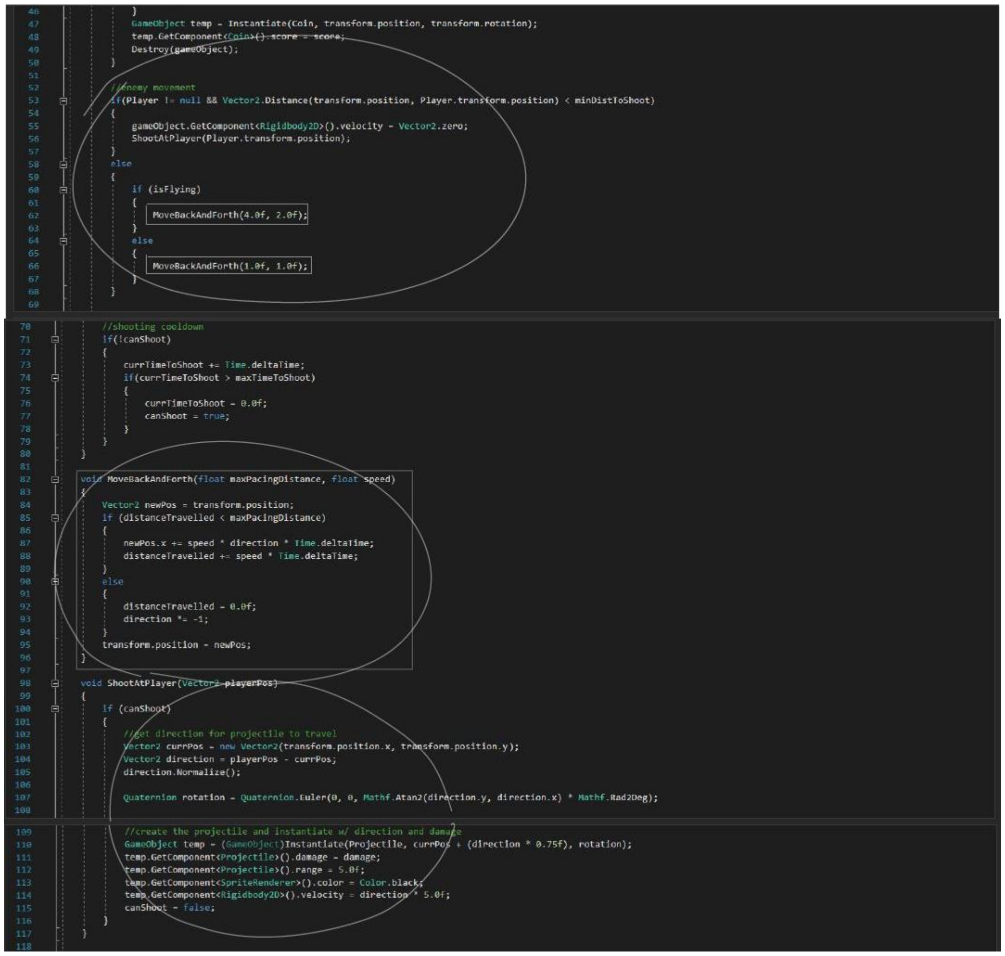
This game was programmed in C# with the purpose to entertain people. In the game, the player (white cube) travels through an endlessly-generated environment, trying to get the most points before they die. The player gains points as it progresses. When the player gets hit by the enemy's projectiles, its health decreases. The video shows the game being played, highlighting features like enemy targeting, inventory management, "sprinting," level generation (for platforms, enemies, and weapons), and text display. It begins with the player jumping onto platforms and evading enemies. The player sprints to reach the high platform and continues to jump and avoid enemies. They pick up another weapon, dropping their current weapon and holding the new one. They then shoot an enemy and pick up the dropped coin for additional points, after which their health drops to zero after getting shot by another enemy, marking the end of the game.

2b)

I worked independently to make an endless platformer game. I began by creating classes that would manage the player (stats and movement, inventory) and a class to control the enemy's behavior. Next, I made a class to endlessly generate platforms, enemies, and weapons, a class to scroll camera through the level, and a class to manage the UI. Finally, I made classes to store data for in-game objects like weapons and coins.
One difficulty was with generating platforms. I wanted them to randomly generate, but I aslo wanted to make sure that the level was playable. To solve this, I created an array of predetermined heights and generated platforms of random lengths at a randomly-chosen height. The heights were set to be challenging, but not impossible to reach with the player controls.
Another difficulty I was with the enemy attack. I wanted the enemy to to shoot projectiles when it was near the player. However, this caused it to shoot a constant stream of projectiles instead of shooting projectiles one after the other.

To solve this, I had a cooldown for the enemy, so every time it shot a projectile, it would have to wait 2 seconds before shooting again.

2c)

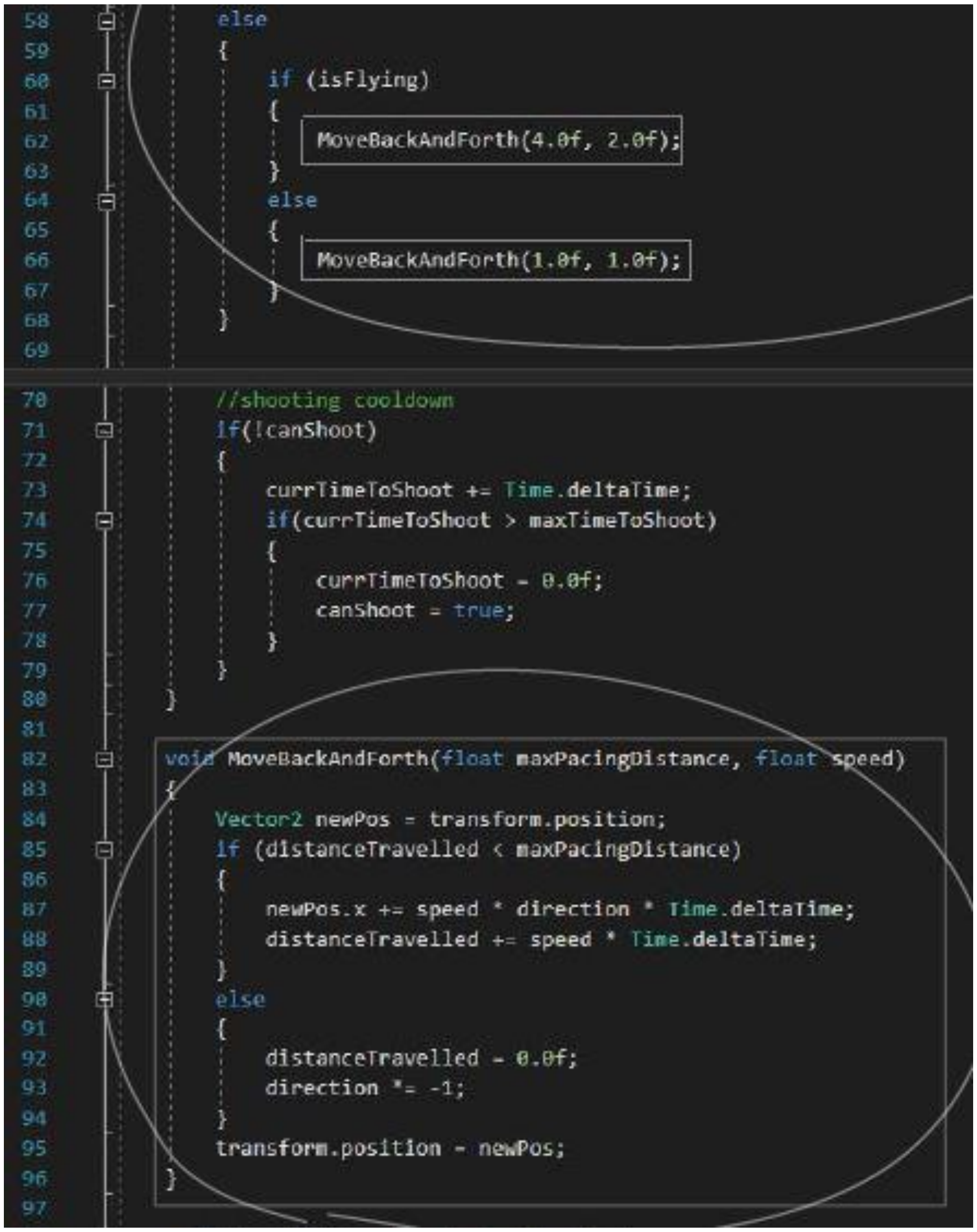


```
            }
            GameObject temp = Instantiate(Coin, transform.position, transform.rotation);
            temp.GetComponent<Coin>().score = score;
            Destroy(gameObject);
        }

        //enemy movement
        if(Player != null && Vector2.Distance(transform.position, Player.transform.position) < minDistToShoot)
        {
            gameObject.GetComponent<Rigidbody2D>().velocity = Vector2.zero;
            ShootAtPlayer(Player.transform.position);
        }
        else
        {
            if (isFlying)
            {
                MoveBackAndForth(4.0f, 2.0f);
            }
            else
            {
                MoveBackAndForth(1.0f, 1.0f);
            }
        }

        //shooting cooldown
        if(!canShoot)
        {
            currTimeToShoot += Time.deltaTime;
            if(currTimeToShoot > maxTimeToShoot)
            {
                currTimeToShoot = 0.0f;
                canShoot = true;
            }
        }
    }

    void MoveBackAndForth(float maxPacingDistance, float speed)
    {
        Vector2 newPos = transform.position;
        if (distanceTravelled < maxPacingDistance)
        {
            newPos.x += speed * direction * Time.deltaTime;
            distanceTravelled += speed * Time.deltaTime;
        }
        else
        {
            distanceTravelled = 0.0f;
            direction *= -1;
        }
        transform.position = newPos;
    }

    void ShootAtPlayer(Vector2 playerPos)
    {
        if (canShoot)
        {
            //get direction for projectile to travel
            Vector2 currPos = new Vector2(transform.position.x, transform.position.y);
            Vector2 direction = playerPos - currPos;
            direction.Normalize();

            Quaternion rotation = Quaternion.Euler(0, 0, Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg);

            //create the projectile and instantiate w/ direction and damage
            GameObject temp = (GameObject)Instantiate(Projectile, currPos + (direction * 0.75f), rotation);
            temp.GetComponent<Projectile>().damage = damage;
            temp.GetComponent<Projectile>().range = 5.0f;
            temp.GetComponent<SpriteRenderer>().color = Color.black;
            temp.GetComponent<Rigidbody2D>().velocity = direction * 5.0f;
            canShoot = false;
        }
    }
```

This algorithm is important because it controls the enemy's behavior, which adds to the enjoyment of the game by adding additional challenges for the player, and giving them new ways of travel through the level.
The first part consists of if-statements that check the distance between the player and enemy. If it's less than 5 meters (minDistToShoot) from the player, it stops moving and shoots. Otherwise, it moves back and forth. Because there are two enemy types, there is an additional check to determine whether the enemy is flying, which affects its movement pattern. This algorithm calls 2 additional algorithms detailing what to do in each scenario.
When moving back and forth, the algorithm claculates how far the enemy has travelled in one direction and checks if it has travelled the max pacing distance. If it has, it turns around and moves in the opposite direction, repeating the same calculation.
When shooting at the player, the algorithm finds the direction between the enemy and player using a normalized vecor and sends a projectile in that direction.

These different algorithms come together to create a new, combined algorithm that controls the enemy's behavior- how and when it acts under different circumstances

2d)



```
            else
            {
                if (isFlying)
                {
                    MoveBackAndForth(4.0f, 2.0f);
                }
                else
                {
                    MoveBackAndForth(1.0f, 1.0f);
                }
            }

            //shooting cooldown
            if(!canShoot)
            {
                currTimeToShoot += Time.deltaTime;
                if(currTimeToShoot > maxTimeToShoot)
                {
                    currTimeToShoot = 0.0f;
                    canShoot = true;
                }
            }
        }

        void MoveBackAndForth(float maxPacingDistance, float speed)
        {
            Vector2 newPos = transform.position;
            if (distanceTravelled < maxPacingDistance)
            {
                newPos.x += speed * direction * Time.deltaTime;
                distanceTravelled += speed * Time.deltaTime;
            }
            else
            {
                distanceTravelled = 0.0f;
                direction *= -1;
            }
            transform.position = newPos;
        }
```

This abstraction is used to simplify the enemy's movement. There are two different types of enemies: a flying enemy and a grounded enemy. The two have similar movement patterns (they both pace back and forth), but they move in different intervals, and with different speeds and travel distances. Therefore, the code is split into multiple parts.
First, the program determines whether the enemy is flying or grounded. Then, based on what type of enemy it is, it calls MoveBackAndForth() with different parameters for pacing distance and speed. MovebackAndForth() then takes those parameters and moves the enemy accordingly.

This abstraction reduces the complexity of the code by making it easier to read and making sure that the same code isn't repeated twice in the update function with a few tweaks to the variable values. Instead of having repeated code, having a separate function with parameters simplifies and condenses the code, making it easier to manage and implement the enemy's movement. This also makes it easier to edit the enemies' movement patterns, because rather than searching through and changing large segments of code, there are only two values that need to be changed, and they are easier to find in the program.