

Assignment 2

Enterprise & Innovation Group Project

Hassan Bahzad

BAH18837264

Table of Contents

Introduction	2
Product Description	2
Product Illustration	2
Main Character	2
Background	3
Weapon Pickups	3
Enemy Sprites	4
Minimum Viable Product (MVP)	4
Playable Character	4
Obstacles and Enemies	4
Power-Ups	5
Weapons	5
Level Design	5
Scoring System	5
Product Differentiation	6
Product Features	7
Player Movement	7
Shooting Mechanics	10
Item Pickup Mechanics	12
Enemy AI	14
Planned Features	15
Product Scalability	15
Extra Scripts	16

Introduction

Within this report I will be discussing the ideation and development of our product. I am the lead animator, graphic designer and game developer of LastMinuteGames, which means I will be going over what has been done for our latest project; “Sugar Crash.”

Product Description

We at LastMinuteGames set out on a mission to create video games that would go on to become classics and give people their fondest childhood memories. We ended up coming up with the idea of a game called “Sugar Crash” which is a rogue-like game that randomises the weapons, enemies, maps, and other features of the game with every run.

The game would take place in a candy noir themed city and would have candy enemies (the mafia) attack the player throughout the streets. The player would then run through each street and defeat the hordes of enemies coming to attack them. The player will eventually reach a boss fight they must defeat to then move to the next stage.

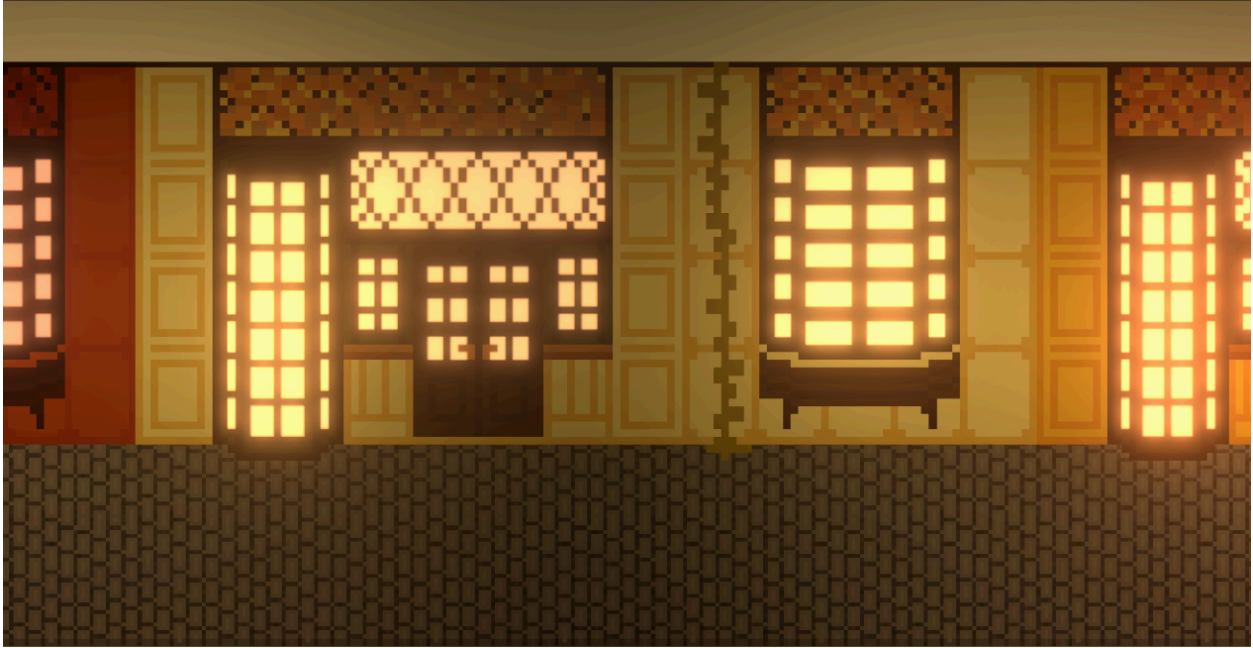
Product Illustration

Main Character



This is the design of the main character, the weapon animation changes depending on the weapon used, however as of right now these are the only two weapons implemented into the game.

Background



As for right now, these buildings are the only background objects in view, however this will be how the world will be lit up and styled.

Weapon Pickups



The weapon drops will glow as shown above, this is to properly set them aside from other non-pickable drops in the game.

These screenshots should indicate how the game is going to look and feel, unfortunately there isn't much more to show in regards to art style as most the work done over the last couple months has been polishing and adding features to the code end of the project.

Enemy Sprites



These images represent the enemy styles within the game, we plan to add more variety in weapon types and such for each enemy, but as of right now this is the style we are working towards.

Minimum Viable Product (MVP)

Game Title: Sugar Crash

Overview: Sugar Crash is a 2D rogue-like dungeon crawler video game that has a player run through the streets of a candy noir themed city. The player will defeat hordes of candy enemies and collect power ups, new weapons and abilities along the way.

MVP Features:

Playable Character

- The MVP will have a single character that the user can control. This will be the protagonist.
- The character will have the basic movement setup, such as walking and shooting.
- These are necessary for the functionality of the game. The playable character will have their base animations and design completed to reflect the theme of the world.

Obstacles and Enemies

- The game will include a variety of enemies.

- Obstacles such as car barricades, will block the player from progressing to a new stage until the street the player is on is clear of all enemies.
- Enemies will be animated and their base functionality of shooting, moving and dying will be implemented.

Power-Ups

- Players will be able to find power-ups after every street they clear which will grant the player new abilities or upgraded stats.
- These powerups will consist of Faster Shooting, Faster Reloading, More Health, Heal Over Time, More Damage, and more.

Weapons

- A variety of weapons will be implemented within the game.
- Each weapon will have their own unique animations and shooting styles
- Weapons can be found through the streets, or be purchased within vending machines somewhere within the level.

Level Design

- The MVP will include a minimum of one complete stage.
- The stage will have randomised weapons, enemies and potentially a randomised map layout.
- The objective of this stage is to clear all the streets of enemies, however in the full release of the game, there will be a final boss to fight to progress to the next stage of the game.

Scoring System

- Enemies killed will give players more points which they can spend on new weapons, upgrades or cosmetics in-game. However, the MVP will only utilise the score so players can buy weapons from the vending machines.

- Points earned will vary depending on the enemy defeated by the player.

Product Differentiation

Sugar Crash is different to other rogue-like games as it offers a fresh new aesthetic for players to roam in. Across all rogue-like dungeon crawlers, there is no point in revisiting old rooms you cleared, and the only rewards for clearing rooms is to gain loot or to find the exit.

Our game offers a brand new approach, where the new events will occur on previously cleared streets, such as friendly characters roaming around or hidden quests. Furthermore, clearing rooms will give loot and points to the player, however the more the city is cleared, the more respect is earned from the city. Respect will work as a levelling system within the game, the higher the respect, the more weapons and enemies are unlocked.

Our gunplay is similar to a fan favourite game known as Enter the Gungeon, however, we tweaked the gameplay and camera movement to be less based on bullet positionings and instead the game relies on the players ability to react fast and dodge faster.

Enter the Gungeon works by having all sorts of calculations being done at all times so that not too many bullets are flying at the same time, making it impossible for the player to survive. However, our game isn't intended to have as many bullets flying that would justify having such a mechanic in the game.

Product Features

Player Movement

The movement in the game is quite basic, using WASD as the standard control set for moving up, left, down and right. The player can use the left mouse button for the standard control of shooting towards the direction of the mouse cursor, and in future versions of the game, the shift button will be used as the standard dodge button.

The game is set up to use Unity's new input system, which allows me to enable or disable the movement at any given moment.

```
0 references
private void OnEnable()
{
    playerMovement.action.Enable();
    mousePosition.action.Enable();
}

0 references
private void OnDisable()
{
    playerMovement.action.Disable();
    mousePosition.action.Disable();
}
```

The code above enables the controls of movement and the detection of the mouse position.

```

0 references
private void Update()
{
    // Read player movement input
    Vector2 movementInput = playerMovement.action.ReadValue<Vector2>();

    // Move the player based on the input
    Vector3 movement = new Vector3(movementInput.x, movementInput.y, 0f);
    transform.Translate(movement * playerSpeed * Time.deltaTime, Space.World);

    // Store the current mouse position
    currentMousePosition = mousePosition.action.ReadValue<Vector2>();

    if (movement.magnitude > 0.1f && !playerMoving)
    {
        playerMoving = true;

        // Play running animation
        bodyAnim.SetBool("isMoving", true);
    }
    else if (playerMoving && movement.magnitude < 0.1f)
    {
        playerMoving = false;

        // Play idle animation
        bodyAnim.SetBool("isMoving", false);
    }

    FlipPlayer();
    RotateWeaponHolder();
}

```

This code is triggered every frame that runs, using this function the player's position is translated to a new position on the screen smoothly. If the magnitude of the player is above 0.1 then the player running animation begins, and stops when the magnitude is less than 0.1.

```

1 reference
private void FlipPlayer()
{
    Vector3 mouseWorldPosition = Camera.main.ScreenToWorldPoint(new Vector3(currentMousePosition.x, currentMousePosition.y, Camera.main.nearClipPlane));
    Vector3 playerToMouse = mouseWorldPosition - transform.position;

    bodyAnim.SetFloat("mouseVertical", playerToMouse.y);
    bodyAnim.SetFloat("mouseHorizontal", playerToMouse.x);

    if (playerToMouse.x >= 0)
    {
        // Face player to the right
        transform.localScale = new Vector3(1f, 1f, 1f);
        weaponHolder.localScale = new Vector3(1f, 1f, 1f);
    }
    else if (playerToMouse.x < 0)
    {
        // Face player to the left
        transform.localScale = new Vector3(-1f, 1f, 1f);
        weaponHolder.localScale = new Vector3(-1f, -1f, 1f);
    }

    // Set sorting order based on mouse position
    SpriteRenderer weaponHolderSpriteRenderer = weaponHolder.GetComponent<SpriteRenderer>();

    if (weaponHolderSpriteRenderer == null)
        return;

    if (playerToMouse.y > 1)
    {
        // Mouse is above the player, set sorting order below the player
        weaponHolderSpriteRenderer.sortingOrder = transform.GetComponent<SpriteRenderer>().sortingOrder - 1;
    }
    else
    {
        // Mouse is below or to the side of the player, set sorting order above the player
        weaponHolderSpriteRenderer.sortingOrder = transform.GetComponent<SpriteRenderer>().sortingOrder + 1;
    }
}

```

This function makes it so that the player faces the direction of the mouse cursor. This flips the player so that they face it at all times, which also saves time when animating the movement animations of the player.

```
1 reference
private void RotateWeaponHolder()
{
    if (hasWeapon) // Only rotate if the player has a weapon
    {
        // Get the mouse position in world coordinates
        Vector3 mouseWorldPosition = Camera.main.ScreenToWorldPoint(new Vector3(currentMousePosition.x, currentMousePosition.y, Camera.main.nearClipPlane));

        // Calculate the direction from the weapon holder to the mouse position
        Vector3 direction = mouseWorldPosition - weaponHolder.position;

        // Calculate the angle between the direction vector and the right vector (1, 0)
        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;

        // Create a Quaternion rotation based on the calculated angle
        Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.forward);

        // Apply the rotation to the WeaponHolder
        weaponHolder.rotation = rotation;
    }
}
```

Since the weapon holder is a child of the player character, I decided to make the weapon rotate to face the mouse position at all times.

Shooting Mechanics

The weapons in the game have currently got two firing modes.

```
5 references
public enum FiringMode
{
    2 references
    SemiAutomatic,
    2 references
    Automatic,
}
```

Semi Automatic, and fully automatic. These two make it so the player will either just hold down the mouse click to shoot, or will have to tap the mouse button to shoot.

```
2 references
void PrimaryAttack(InputAction.CallbackContext context)
{
    if (isAttacking)
        return;

    switch (currentFiringMode)
    {
        case FiringMode.SemiAutomatic:
            weaponAnim.SetTrigger("PrimaryAttack");
            SetAnimationSpeed(animationSpeedMultiplier);
            break;

        case FiringModeAutomatic:
            weaponAnim.SetBool("PrimaryAttack", true);
            SetAnimationSpeed(animationSpeedMultiplier);
            break;
    }

    isAttacking = true;
}
```

When the player does trigger the primary attack function by left clicking, it will trigger the firing animation. The animation speed varies depending on the power ups the player has, but by default the animation will play at a normal speed (1x.)

Within the animation itself, a function will be called within the script;

```
// ----- SHOTGUN PROJECTILES -----

0 references
public void ShootPrimary_Shotgun_Projectiles()
{
    // Implement shotgun firing logic
    int numberOfProjectiles = 5;

    float spreadAngle = 7f; // Fixed angle between bullets

    for (int i = 0; i < numberOfProjectiles; i++)
    {
        float bulletAngle = i * spreadAngle - (spreadAngle * (numberOfProjectiles - 1) / 2f);
        Quaternion spreadRotation = Quaternion.Euler(0f, 0f, bulletAngle);
        Instantiate(primaryProjectileObject[Random.Range(0, primaryProjectileObject.Length)], shotPoint.position, shotPoint.rotation * spreadRotation);
    }
}
}
```

The above code is an example of the shotgun projectile. Once the animation triggers this function, projectiles will spawn at specific angles and therefore give the shotgun projectile effect.

Once these projectiles have been instantiated, they each will have their own script called “ProjectileScript.”

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
0 references
public class ProjectileScript : MonoBehaviour
{
    1 reference
    public float projectileDamage = 5f;
    1 reference
    public float speed = 5f;

    2 references
    private Rigidbody2D rb;
    1 reference
    public float removeObjectDelay = 5f;

    0 references
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        rb.velocity = transform.right * speed;
        Destroy(gameObject, removeObjectDelay);
    }

    0 references
    void OnTriggerEnter2D(Collider2D sender)
    {
        if(sender.tag == transform.tag)
            return;

        if(sender.GetComponent<TakeDamageScript>())
        {
            sender.GetComponent<TakeDamageScript>().TakeDamage(projectileDamage);
            Destroy(gameObject);
        }
    }
}
```

This script is simple, it moves the projectile in the direction it was facing when it was spawned. If the projectile doesn't hit anything that contains the "TakeDamageScript" within a specific amount of time, the projectile will be destroyed from the game.

The projectiles will be spawned at the "ShotPoint" object, which is attached to the weapon holder object mentioned before.



This means that when the weapon rotates as mentioned earlier in this report, the shotpoint also rotates with the weapon. Therefore, the projectiles will always fly in the direction the player is aiming.

Item Pickup Mechanics

Weapons and Items will share a script which will first detect if the player is in range. They do this by checking when an object with the "Player" tag enters its collider box. This will then change a bool called "playerInRange" to true and false if the player leaves.

```
0 references
void OnTriggerEnter2D(Collider2D sender)
{
    if (sender.CompareTag("Player"))
    {
        playerInRange = true;
    }
}

0 references
void OnTriggerExit2D(Collider2D sender)
{
    if (sender.CompareTag("Player"))
    {
        playerInRange = false;
    }
}
```

The code above shows how this works, the sender collider being the collider that enters the trigger box.

```
2 references
private void PickupWeapon(InputAction.CallbackContext context)
{
    if (!playerInRange)
        return;

    GameObject player = GameObject.FindGameObjectWithTag("Player");
    Transform weaponHolder = player.transform.Find("WeaponHolder");

    // Check if player is holding a weapon already
    if (player.GetComponent<PlayerController>().hasWeapon)
    {
        Transform existingWeapon = weaponHolder.GetChild(0);
        existingWeapon.GetComponent<DropWeaponScript>().DropWeaponFunction();
        Destroy(existingWeapon.gameObject);
    }
    else
    {
        player.GetComponent<PlayerController>().hasWeapon = true;
    }

    GameObject weaponInstance = Instantiate(weaponObject, weaponHolder);

    if (weaponHolder != null)
    {
        weaponInstance.transform.parent = weaponHolder;
        Destroy(gameObject);
    }
}
```

Once the player is in range, the player can press the interact button ('E') to trigger the PickupWeapon function. This function will replace whatever weapon the player is holding with the item being interacted with.

If the player is holding a weapon already, the "DropWeaponFunction" function will be called, this function drops the held weapon onto the ground.

Enemy AI

Enemies will have two types of weapon variation, ranged and melee. For the ranged weapons I used a simplistic approach that will make the enemy fire their weapon after a specific amount of time

```
0 references
private void Start()
{
    anim = GetComponent<Animator>();
    // Start shooting coroutine
    StartCoroutine(ShootCoroutine());
}

1 reference
private IEnumerator ShootCoroutine()
{
    while (true)
    {
        // Wait for the specified frequency
        yield return new WaitForSeconds(1f / shotFrequency);

        // Instantiate the projectile at the shot point position and rotation
        anim.SetTrigger("Attack");
    }
}

0 references
public void EnemyTakeShot()
{
    Instantiate(projectilePrefab, shotPoint.position, shotPoint.rotation);
}
```

Planned Features

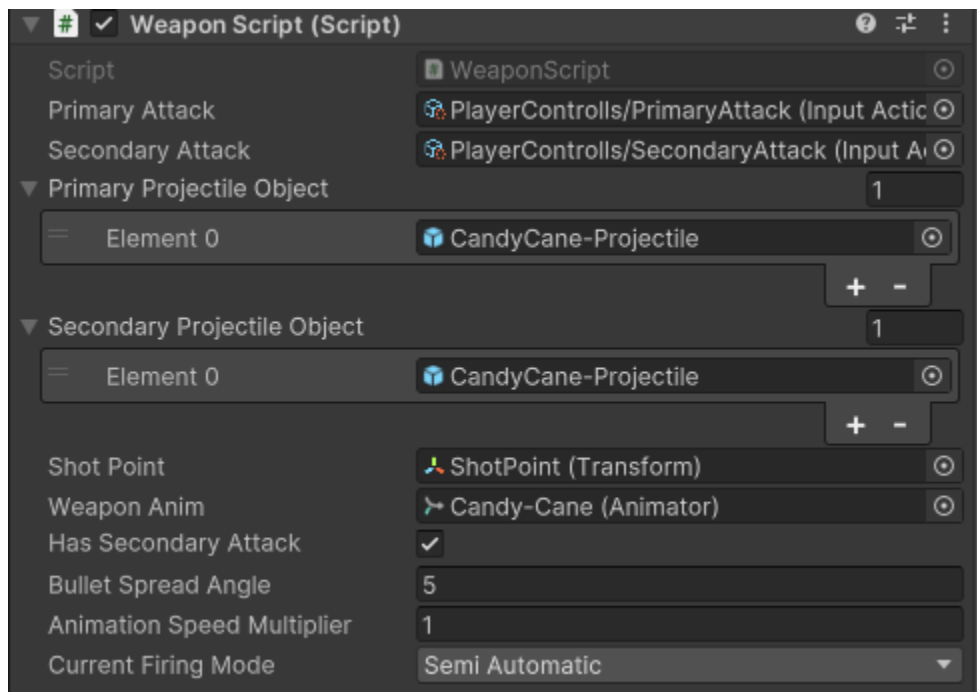
There are still many features to be added to the game, things such as boss fights, weapon shops, power up scripts, optimisation, advanced bullet handling, more animations and enemies, etc..

The major addition that needs to be made before the MVP can be complete is the procedural generation of the map. We have a few ideas on how we are going to implement this, however as of right now no method has accomplished the task we set.

This report is meant to only go over some of the basic features the game is meant to have for the alpha version of the game.

Product Scalability

The game's code, animation handling, enemies, and weapon scripts are all written with the intent to make the game as scalable as possible. Each script can be utilised on different weapons, enemies and projectiles, ensuring the game runs as effectively as possible.



An example of this would be the weapon script (as shown above.) This script can be applied to every weapon in the game, and will allow custom projectiles, secondary attacks and the firing mode to be changed for each weapon object.

Each script works well with one another, which makes further development of the game a lot easier to maintain and understand.

Extra Scripts

The above mentioned features are not the only scripts written for the game so far, the below screenshot shows only a few of the extra scripts written to allow the game to function as smoothly as it does.

Animation handling, health tracking, custom physics, camera movement, and a lot more is handled within the scripts below.

