Practical Gaming 2023

# Darragh Quinn

# T00226260

# Spider-Man Sandbox

# Gameplay

Game Summary: The game was designed to be a simple sandbox to test the movement and various web mechanics. The main focus of the game is the web swinging, and the methods that can be called to effect the player or webbedObjects while the grapple is in effect.

Controls:

Left Mouse Button Hold = Left Web Grapple.

Right Mouse Button Hold = Right Web Grapple.

F = Pull Player along leftWeb.

G = Pull Player along rightWeb.

Q = Pull Object attached to leftWeb.

E = Pull Object attached to rightWeb.

B = Fling Player towards Grapple Point.

O = Throw Object attached to left web.

# Coding

Under each of the following headings, please describe the concept, why is it or isn’t it useful/needed, where do you implement in your project, you may provide screenshots or cut and past code segments etc..

* Frame Rate Independence

This is a concept used in programming to ensure that the speed of an application or game remains consistent regardless of the computer or device it is running on. This is achieved by making sure that all movement, animation, and other time-based processes are calculated based on real-world time rather than the number of frames per second that the application is rendering.

* Interfaces

Interfaces are like contracts that define a set of methods and properties that a class must implement. Interfaces can be implemented by multiple classes, allowing them to share common behavior while still being able to have their unique properties and methods. This allows for code reusability and makes it easier to manage large codebases.

* Inheritance

Inheritance is the concept of creating a new class that is a modified version of an existing class. The new class, called the subclass or derived class, inherits the properties and methods of the existing class, called the superclass or base class. This allows the subclass to reuse existing code and add new functionality without having to rewrite the entire class from scratch.

* + The Movement class inherits from the MonoBehaviour class, which is a built-in class in Unity that provides basic functionality for game objects. The Movement class also overrides some of the methods provided by MonoBehaviour, such as Start() and Update(), to add custom functionality.

Brief

* Case pattern
  + The case pattern is a software design pattern that defines a set of actions to be taken based on the value of an input. This pattern is commonly used in switch statements, where each case statement defines a specific action to be taken based on the value of the input.
* Observer Pattern
  + This is a design pattern used in software development to allow objects to communicate with each other without being tightly coupled. In this pattern, an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes.
* Polymorphism
  + Polymorphism is the ability of an object to take on multiple forms. This means that a subclass can be treated as if it were an instance of its superclass, allowing for greater flexibility and code reuse. Polymorphism is often achieved through the use of interfaces or abstract classes.
  + The player can use both left and right grappling hooks to swing, but the code treats them as interchangeable. The StartGrapple(), StopGrapple(), and DrawRope() methods all take a SpringJoint parameter, which can be either the left or right web.
* Communication between scripts/game objects

In game development, scripts and game objects often need to communicate with each other to exchange information and trigger certain behaviors. Such as Direct references and Events.

LRLeft = LeftShooterTip.parent.GetComponent<LineRenderer>();

LRRight = RightShooterTip.parent.GetComponent<LineRenderer>();

These lines of code retrieve a reference to the LineRenderer component of the parent object of LeftShooterTip and RightShooterTip. The code also communicates with other scripts through function calls such as StartGrapple(), StopGrapple(), pullWeb(), pullObject(), WebZip(), DrawRope(), MovePlayer(), MyInput(), SpeedControl(), Jump(), and ResetJump().

* Instantiation and Prefabs
  + Instantiation is the process of creating a new instance of a game object at runtime, while prefabs are pre-made game objects that can be easily instantiated and customized. These concepts are commonly used in game development to reduce development time and improve performance.

Sample Code:

* + if (Input.GetKeyDown(KeyCode.K))
  + {
  + Shoot();
  + }
  + }
  + void Shoot()
  + {
  + GameObject webBall = Instantiate(webBallPrefab, shootingPoint.position, shootingPoint.rotation);
  + webBall.GetComponent<Rigidbody>().AddForce(Camera.main.transform.forward \* shootingForce, ForceMode.Impulse);
  + Destroy(webBall, 2f); // Change the lifetime of the web ball as needed
  + }
* Magic Numbers
  + Magic Numbers are hard-coded numeric values that are used in a program without explanation. They make code harder to read and maintain since their purpose is not immediately apparent. It is good practice to use constants or variables instead of magic numbers, so that they can be easily modified or reused throughout the program.
* Model Animation
  + This refers to the process of creating animations using 3D models. This process is commonly used in video game development, film, and other industries where realistic 3D animation is required.
* Self made models and or animations
  + This refers to the process of creating 3D models and animations from scratch, without the use of pre-made assets or templates. This technique requires advanced skills in 3D modeling and animation software and is commonly used by professional animators and game developers.
  + The sack boy prefab was a self-made model from my 3D Animation Module, which I have included as a test dummy of sorts.
* Interactions between objects/scripts
* In game development, interactions between objects/scripts refer to the communication and exchange of data between different game objects or scripts. For example, a player character may interact with a game object such as a weapon to pick it up, or a script may send a message to another script to trigger a specific behavior.

The Movement script is attached to a game object that is responsible for player movement. The script interacts with the LineRenderer components attached to the LeftShooterTip and RightShooterTip objects. The script also interacts with the SpringJoint components attached to the grappling hook.

* Proper code placement
  + Proper code placement refers to placing code in the appropriate locations within the game code architecture. This can help ensure that the code is organized and efficient, making it easier to maintain and update in the future. For example, code that controls the behavior of a specific game object may be placed in a separate script that is attached to that object.
  + private void Start()
  + {
  + rb = GetComponent<Rigidbody>();
  + rb.freezeRotation = true;
  + readyToJump = true;
  + LRLeft = LeftShooterTip.parent.GetComponent<LineRenderer>();
  + LRRight = RightShooterTip.parent.GetComponent<LineRenderer>();
  + }
  + private void Update()
  + {
  + // Get input values
  + float horizontal = Input.GetAxis("Horizontal");
  + float vertical = Input.GetAxis("Vertical");
  + // Calculate movement vector
  + Vector3 movement = new Vector3(horizontal, 0f, vertical) \* moveSpeed \* Time.deltaTime;
  + // Apply movement to Rigidbody
  + rb.MovePosition(transform.position + movement);
  + }
  + private void FixedUpdate()
  + {
  + // Get input values
  + float jumpInput = Input.GetAxis("Jump");
  + // Check if the player is on the ground
  + if (IsGrounded())
  + {
  + // Apply jump force if jump input is detected
  + if (jumpInput > 0f)
  + {
  + rb.AddForce(Vector3.up \* 5f, ForceMode.Impulse);
  + }
  + }
  + }
* Code repetition
  + Code repetition refers to writing the same code multiple times in different parts of a game project. This can be inefficient and can make it more difficult to maintain the code. Instead, developers can use functions or classes to encapsulate common functionality and reuse that code throughout the project. This can help make the code more organized and easier to maintain, as well as reducing the overall amount of code needed to implement the desired functionality.
  + There are some examples of Code Repetition in this code. For example, the code for starting and stopping a grapple is duplicated for the left and right mouse buttons:
  + if (Input.GetMouseButtonDown(1))
  + rightWeb = StartGrapple(transform, LRRight);
  + else if (Input.GetMouseButtonUp(1))
  + StopGrapple(rightWeb, LRRight);
  + if (Input.GetMouseButtonDown(0))
  + leftWeb = StartGrapple(transform, LRLeft);
  + else if (Input.GetMouseButtonUp(0))
  + StopGrapple(leftWeb, LRLeft);
  + This code could be refactored to remove the duplication and make it more maintainable. For example, you could define a GrappleButton class that encapsulates the logic for each grapple button, and then create instances of that class for the left and right buttons.

**Features:**

* StartGrapple
  + This method is responsible for starting the grapple mechanism. It is usually called when the player presses a button to initiate the grapple. The method first creates a raycast from the player's position in the forward direction and checks if it hits any object with a specific tag, like a wall or a target point. If the raycast hits an object, the grapple attaches to that object and the player is pulled towards it.
  + It’s the most important method in the entire project, as it is the basis for most of the movement-based gameplay. It took many failed attempts to get dual grapples working, along with fluid physics for the pendulum motion of the swing.

SpringJoint StartGrapple(Transform shooter, LineRenderer LR)

{

Debug.DrawRay(Camera.main.transform.position, Camera.main.transform.forward \* 20f, Color.blue, 3);

RaycastHit hit;

if (Physics.Raycast(Camera.main.transform.position, Camera.main.transform.forward, out hit, maxDistance))

{

//grapplePoint = hit.point;

SpringJoint Web = shooter.gameObject.AddComponent<SpringJoint>();

Web.autoConfigureConnectedAnchor = false;

Web.connectedAnchor = hit.point;

webbedObject = hit.transform;

Web.maxDistance = hit.distance \* 1f;

Web.minDistance = hit.distance \* 0.5f;

Web.spring = 2.5f;

Web.damper = 7f;

Web.massScale = 4.5f;

LR.positionCount = 2;

//currentGrapplePosition = WebTip.position;

return Web;

}

return null;

}

* PullWeb
  + This method is responsible for pulling the player towards the attached grapple point. It applies a force to the player's rigidbody in the direction of the grapple point. The amount of force applied is determined by a pullSpeed variable that can be adjusted to tweak the grapple behavior.

private void pullWeb(ref SpringJoint currentWeb)

{

currentWeb.maxDistance \*= 0.7f;

currentWeb.spring \*= 1.0075f;

}

* PullObject
  + This method is similar to PullWeb(), but instead of pulling the player towards the grapple point, it pulls a specified object towards the player. The method takes an object as a parameter and applies a force to its rigidbody in the direction of the player. This could be useful for creating puzzles in a level where the player needs to pull an object towards them to progress.

void pullObject(ref SpringJoint currentWeb)

{

if(webbedObject != null)

{

var pull = pullSpeed \* Time.deltaTime;

webbedObject.transform.position = Vector3.MoveTowards(webbedObject.transform.position, rb.position, pull);

}

}

* ThrowWebbedObject
  + The ThrowWebbedObject method is used to throw an object that has been webbed by the player. When the player presses the designated input key for throwing, this method is called. It first checks if an object is attached to the spring joint, and if so, releases the joint. It then calculates the direction to throw the object (in the direction the player is facing), and applies a force to the object's rigidbody in that direction. This creates the effect of the player throwing the object away from them.

public void ThrowWebbedObject(ref SpringJoint currentweb, float throwForce)

{

// Check if there is an object attached to the spring joint

if (leftWeb.connectedBody != null)

{

// Release the spring joint

leftWeb.connectedBody = null;

// Get the direction to throw the object

Vector3 throwDirection = transform.forward;

// Apply force to the rigidbody in the direction of the throw

Rigidbody rb = leftWeb.GetComponent<Rigidbody>();

if (rb != null)

{

rb.AddForce(throwDirection \* throwForce, ForceMode.Impulse);

}

}

}

* TimeTrial
  + Some time was also put into creating a TimeTrial script, which I intended on turning into a mini-game within the sandbox, where the player has to go through randomly generated checkpoint prefabs in the sandbox. Once all checkpoints have been entered, the timer is stopped and a message is displayed with how long it took to reach all of them. Unfortunately, I had difficulties getting the script to work with the MeshRenderer and checkpointPrefab, so it does not work in the Project. Checkpoints are generated within a random radius, but they are not visible and do not disappear once touched.
* public class TimeTrial : MonoBehaviour
* {
* public GameObject checkpointPrefab;
* public int numCheckpoints = 10;
* public float checkpointRadius = 100f;
* private List<GameObject> checkpoints = new List<GameObject>();
* private int checkpointsPassed = 0;
* private float startTime = 0f;
* void Start()
* {
* // Generate checkpoints
* for (int i = 0; i < numCheckpoints; i++)
* {
* Vector3 randomPos = Random.insideUnitSphere \* checkpointRadius;
* randomPos.y = 4f;
* GameObject checkpoint = Instantiate(checkpointPrefab, randomPos, Quaternion.identity);
* checkpoints.Add(checkpoint);
* MeshRenderer renderer = checkpoint.GetComponent<MeshRenderer>();
* renderer.material.color = Color.yellow;
* }
* // Start timer
* startTime = Time.time;
* }
* void Update()
* {
* // Check if all checkpoints have been passed
* if (checkpointsPassed >= numCheckpoints)
* {
* // End time trial and print results
* float timeElapsed = Time.time - startTime;
* Debug.Log("Time trial complete! Time: " + timeElapsed.ToString("F2") + "s");
* Destroy(gameObject);
* }
* }
* void OnTriggerEnter(Collider other)
* {
* // Check if the player collided with a checkpoint
* if (other.CompareTag("Checkpoint"))
* {
* // Remove checkpoint and increment count
* Destroy(other.gameObject);
* checkpointsPassed++;
* }
* }
* }

References: <https://www.youtube.com/@davegamedevelopment> for some of the Movement concepts and SpringJoint research.