Practical Gaming 2022

# Joseph Diggins

# T00172862

# Doge

# Gameplay

W – Move Forward

A – Move Left

S – Move Backwards

D – Move Right

Space – Jump

Shift + Direction - Sprint

Left Mouse – Sword Attack

Right Mouse – Defend

E – Interact

# Coding

* Frame Rate Independence
* This concept is useful because if a game is not Framerate independent users will have different gameplay experiences. I.e., if a game runs at 30fps on a slow computer and 60fps on a fast one, the user on the 30fps will have a significantly worse experience due to their game progressing at half the speed of the other computer. Framerate independent games ensure that both users play at the same speed, no matter their respective framerates.
  + Character movement.

transform.position += characterSpeed \* moveDirection \* Time.deltaTime;

* + Character rotation:

transform.rotation = Quaternion.Slerp(transform.rotation, Quaternion.LookRotation(moveDirection), Time.deltaTime \* rotate Speed);

* Interfaces
* Interfaces are useful because they allow multiple classes to use the same methods and doesn’t depend on the object type. It is useful because it allows for multiple inheritance in C# using a single class.
  + iDamageable
* Inheritance
* Inheritance is useful because it allows you to create new classes that reuse & modify the behaviour defined in other classes. Using my game as an example, I have an enemy class, which is the base class, that the specific enemies inherit from. All the enemies have the same enemy states but with modified behaviours for some states. Inheritance allows for the reuse of code for these enemies instead of re-writing the enemy script for each specific enemy.
  + ShellEnemy script inherits from EnemyController script.
  + SlimeEnemy script inherits from EnemyController script.
  + Coin & Heart inherit from Collectables script.
* Case pattern
* Case pattern allows you to use a single expression to switch between cases. It is useful because it is significantly faster than an if/else when the number of conditions is large. In my game I used a case pattern in the enemy controller script to allow changes to the characters state.

switch (isCurrently)

{

case enemyState.patrol:

//code

break;

case enemyState.moveToTarget:

//code

break;

case enemyState.attack:

//code

break;

case enemyState.flippingOver:

//code

break;

case enemyState.upsideDown:

//code

break;

case enemyState.flippingBack:

//code

break;

case enemyState.dying:

//code

Break;

}

* Observer Pattern
* Observer pattern allows objects to notify other objects about changes in their state. It is useful to apply when you have several objects depending on that object and are required to do something when the dependent objects state changes or updates.
* Polymorphism
* Polymorphism is the ability of an object to take on many forms. It is useful because it allows you to code an interface that reduces coupling, increases reusability, and makes your code easier to read. OnTriggerEnter is a form of polymorphism. When the OnTrigger function is called, we don’t know which type of collider is being used. Each objects specific collider is passed into the function. Since all these colliders inherit from the Collider parent class, any of them will work.

* private void OnTriggerEnter(Collider other)

{

if (other.gameObject.tag == "Sword" && player.Attacking)

{

swordHit();

}

if (isCurrently == enemyState.upsideDown && other.gameObject.tag == "Player" && !player.Grounded)

{

take\_damage(amtDamage);

}

if (isCurrently != enemyState.upsideDown && other.gameObject.tag == "Player")

{

player.take\_damage(amtDamage);

}

}

* Communication between scripts/game objects
* Communication between scripts is hugely important in gaming. Communication from one script to another allows us to update variable. E.g., in the shell enemy script, we communicate with the PlayerController script in order to determine if we inflict damage to the player or to the enemy object itself.
* public class ShellEnemy : EnemyController

{

PlayerController player;

new void Start()

{

base.Start();

player = FindObjectOfType<PlayerController>();

}

private void OnTriggerEnter(Collider other)

{

if (other.gameObject.tag == "Sword" && player.Attacking)

{

swordHit();

}

if (isCurrently == enemyState.upsideDown && other.gameObject.tag == "Player" && !player.Grounded)

{

take\_damage(amtDamage);

}

if (isCurrently != enemyState.upsideDown && other.gameObject.tag == "Player")

{

player.take\_damage(amtDamage);

}

}

}

* + We communicate with the player script in the first if statement to determine if the player is attacking. If they are then the method swordHit() will be called.
  + In the second if statement, we are again communicating with the player script to determine if the player is in the air when they collide. The enemy will take damage if the condition is true.
  + Finally, if the third condition is true, we communicate with the player script for the player to take damage.
* Instantiation and Prefabs
* Instantiation is used to create new objects at runtime. In my game I’ve used instantiation to create projectiles for the wizard. It is also useful for cloning objects & scripts.
* Instantiate(projectilePrefab, wand.position, transform.rotation);
* Prefabs are components that allows Game Objects to be stored in the project for reuse. They can be shared between scenes without having to be configured again. The map design in my game is done through numerous prefabs.
* Magic Numbers
* Magic numbers is the direct use of a number within the code. Magic numbers are not useful and should be avoided. By avoiding the use of magic numbers, it improves the readability of the code, and it is also a lot easier to maintain. I have avoided magic numbers by creating constant data types and using the name of the data type within the code.
  + private const float jumpHeight = 5.5f;

rb.AddForce(new Vector3(0, jumpHeight, 0), ForceMode.Impulse);

* + private const float rotateSpeed = 5f;

transform.rotation = Quaternion.Slerp(transform.rotation, Quaternion.LookRotation(moveDirection), Time.deltaTime \* rotateSpeed);

* + public const int amtDamage = 50;

internal void swordHit()

{

take\_damage(amtDamage);

}

* Model Animation (https://en.wikipedia.org/wiki/Model\_animation)
* Model animation is a form of stop motion animation designed to merge with live-action footage to create the illusion of a real-world fantasy sequence. It is useful in games development because it makes the objects look more realistic and persuasive than a flat 2D equivalent.
  + Walking/Running animation when the character is moving.
  + Attacking/Defending
  + Jumping
  + Dying
* Self made models and or animations
  + My self-made model appears at the start of the game at the entrance to the castle.
* Interactions between objects/scripts
* Interactions between objects are useful for several reasons. Depending on the type of game they may allow for a character to be upgraded, hints and new quests. In my game the player can interact with the chest using the ‘E’ key when they are close by to open them and get 10 coins.
* if (isPlayerClose)

{

if (Input.GetKey(KeyCode.E) || player.Attacking)

{

Open();

}

}

* Propper code placement
* Proper code placement is having good clean code. It is useful for a variety of reason.
  + Easier & Quicker to maintain
  + Easier to understand
  + Reusability
  + Easier to test
* Code repetition
* Code repetition is the duplication of code throughout your project. It is inefficient to duplicate code and depending on the scale could affect the performance of your project. To reduce code repetition one can, use loops, trees, methods etc.
* Health System
* Dynamic health system used to keep track of the characters health.

public void UpdateHealth()

{

for (int i = 0; i < hearts.Length; i++)

{

if (i < currentHealth)

{

hearts[i].color = Color.red;

}

else

{

hearts[i].color = Color.black;

}

}

}

* Hidden Gem
* Gem spawns when all coins in the level have been collected. (80 coins)