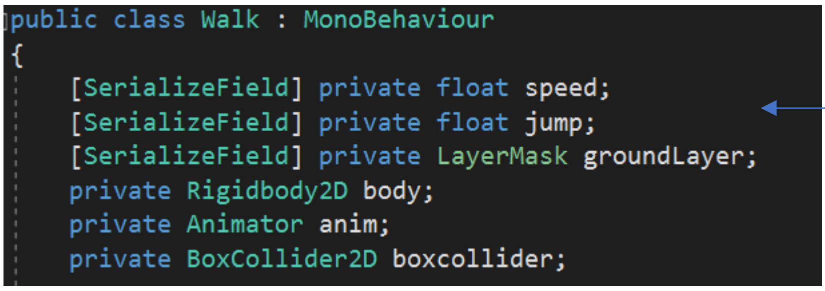
Techniques:

1. Encapsulation
2. Importing libraries with the keyword ‘using’- Unity methods
3. Selection statements and operators
4. Inheritance
5. Event Handling
6. GUI

1. Encapsulation

Encapsulation is used to set data under a single unit. It is used to prevent data from being accessed by code outside the data which is encapsulated (“C# - Encapsulation.”, 1). It is used since the success criteria 1, 2 and 3 require a defined set of variables which are not communicated with the rest of the program.

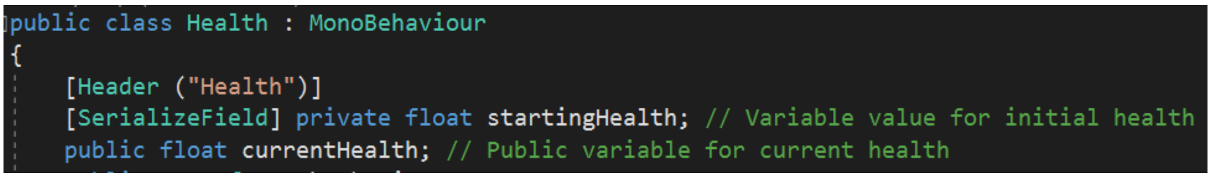
The *Health* class and the *Walk* class both make use of encapsulation.



```
public class Walk : MonoBehaviour
{
    [SerializeField] private float speed;
    [SerializeField] private float jump;
    [SerializeField] private LayerMask groundLayer;
    private Rigidbody2D body;
    private Animator anim;
    private BoxCollider2D boxcollider;
```

These variables only apply to the character and are thus private inside the *Walk* class which determines character movement

As the character’s movement variables such as speed and jump are constant in every execution, it is useful to store it in a private attributes. Moreover, the Boxcollider2D, Rigidbody2D and Animator objects are also defined with private attributes as they belong specifically to only the player, and not other game elements since these are predefined objects in Unity.

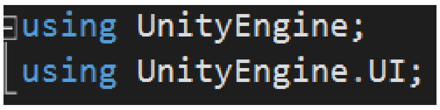


```
public class Health : MonoBehaviour
{
    [Header ("Health")]
    [SerializeField] private float startingHealth; // Variable value for initial health
    public float currentHealth; // Public variable for current health
```

Similarly, the startingHealth is always constant and therefore has a private attribute. However, the currentHealth needs to be accessed by other classes and thus has a public attribute. Without a public attribute, it would be protected within the Health class and inaccessible by others.

2. Importing libraries using the keyword ‘using’ to apply Unity methods

The Unity game engine provides a predefined set of variables and methods through libraries. The program uses several methods specific to the Unity engine in order to achieve success criteria 1, 4, 5 and 6 which are criteria possible only through a video game. These methods were predefined in the UnityEngine and UnityEngine.UI library and were used since they do not have to be repeatedly written as well as do not have to be debugged, which reduces the development time.

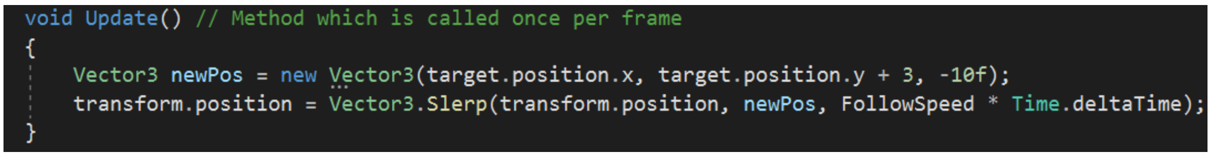


```
using UnityEngine;
using UnityEngine.UI;
```

Unity Engine libraries for predefined methods and GUI tools

Update():

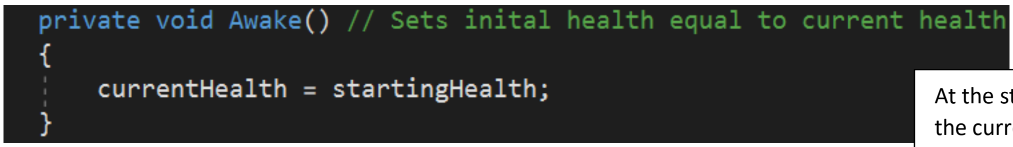
The update method is a specialized method which is called every frame, and is used for the elements which are in constant motions, such as the character and the camera.



```
void Update() // Method which is called once per frame
{
    Vector3 newPos = new Vector3(target.position.x, target.position.y + 3, -10f);
    transform.position = Vector3.Slerp(transform.position, newPos, FollowSpeed * Time.deltaTime);
}
```

The camera follows the character and hence its position is transformed every frame

Awake():

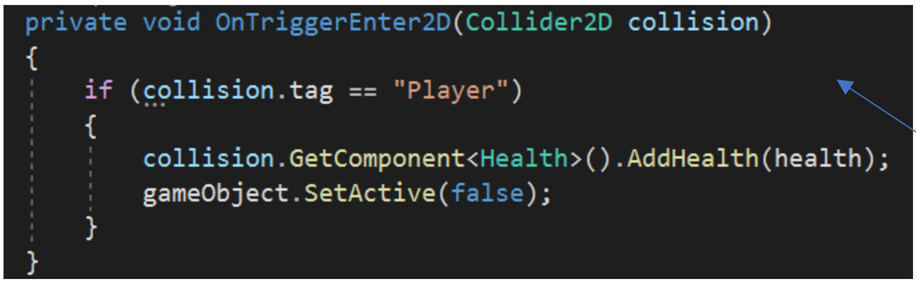


```
private void Awake() // Sets inital health equal to current health
{
    currentHealth = startingHealth;
}
```

At the start of the program, the currentHealth is set to the specific startingHealth value

The awake method is called at the start of the program and is used for getting components such as health, body and animation.

OnTriggerEnter2D():



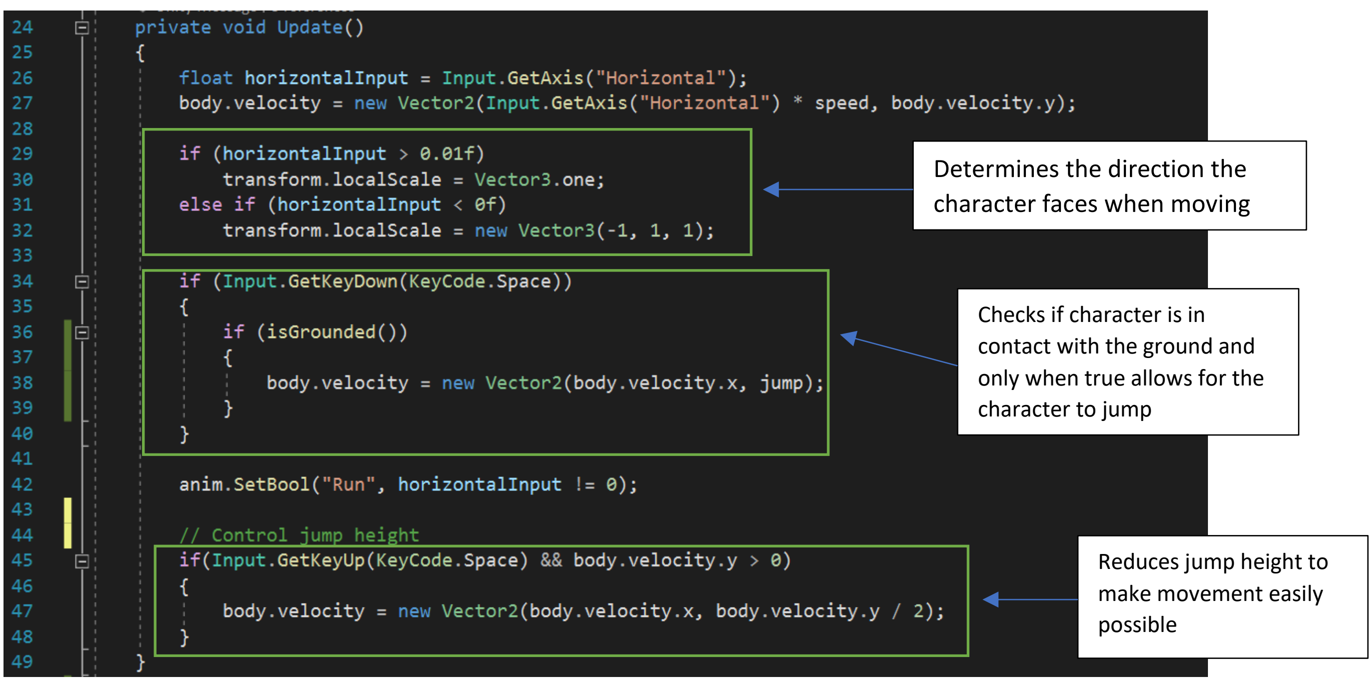
```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "Player")
    {
        collision.GetComponent<Health>().AddHealth(health);
        gameObject.SetActive(false);
    }
}
```

The method checks a Boolean value for the collision with a object with the “Player” tag

This is a method which is used along with 2D box colliders to simulate physics in the game. The method is called when an object enters a 2D box collider.

3. Selection statements- if(), else if(), nested if() statement. Boolean logical & Comparison operators

An *if* selection statement is used along with an *else if* statement to select one of two statements to execute based on the value of a Boolean expression() (Wagner, 1). Comparison and Boolean logical operators are also used to check for input and numerical values (Pkulikov, 1).



```
24  private void Update()
25  {
26      float horizontalInput = Input.GetAxis("Horizontal");
27      body.velocity = new Vector2(Input.GetAxis("Horizontal") * speed, body.velocity.y);
28
29      if (horizontalInput > 0.01f)
30          transform.localScale = Vector3.one;
31      else if (horizontalInput < 0f)
32          transform.localScale = new Vector3(-1, 1, 1);
33
34      if (Input.GetKeyDown(KeyCode.Space))
35      {
36          if (isGrounded())
37          {
38              body.velocity = new Vector2(body.velocity.x, jump);
39          }
40      }
41
42      anim.SetBool("Run", horizontalInput != 0);
43
44      // Control jump height
45      if(Input.GetKeyUp(KeyCode.Space) && body.velocity.y > 0)
46      {
47          body.velocity = new Vector2(body.velocity.x, body.velocity.y / 2);
48      }
49  }
```

Determines the direction the character faces when moving

Checks if character is in contact with the ground and only when true allows for the character to jump

Reduces jump height to make movement easily possible

Line 29 transforms the players vector to face right by having a positive x coordinate when the horizontalInput is towards the right. This is checked by the '>' operator to check that input is towards the right direction and hence has a positive value above 0.01. The else if statement similarly checks if the player is facing left when a negative axis input value is received thus smaller than '<' 0.

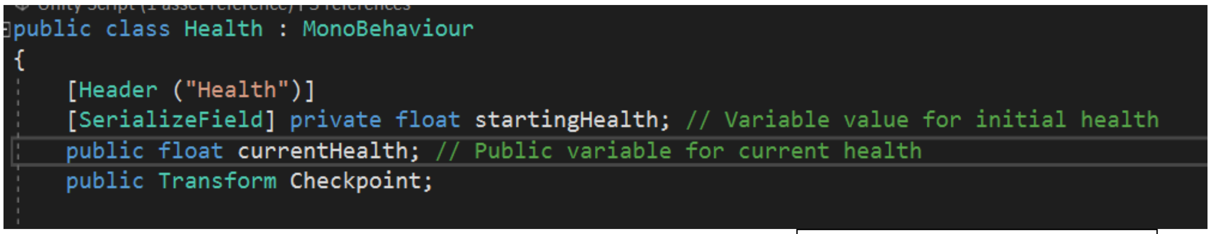
Another logical operator is used to run the animation. This is run when the player is moving therefore the horizontalInput is not equal to 0 and is checked by the '!=' operator.

Lastly, the jump height has to be controlled as it should realistically be lesser than the horizontal speed. Hence, the y velocity is halved. However, this code is run when both the spacebar input is received and the y velocity is greater than 0 as the change in y velocity has to occur before halving it. To check for both conditions the '&&' operator is used. This prevents the need to write separate if() statements.

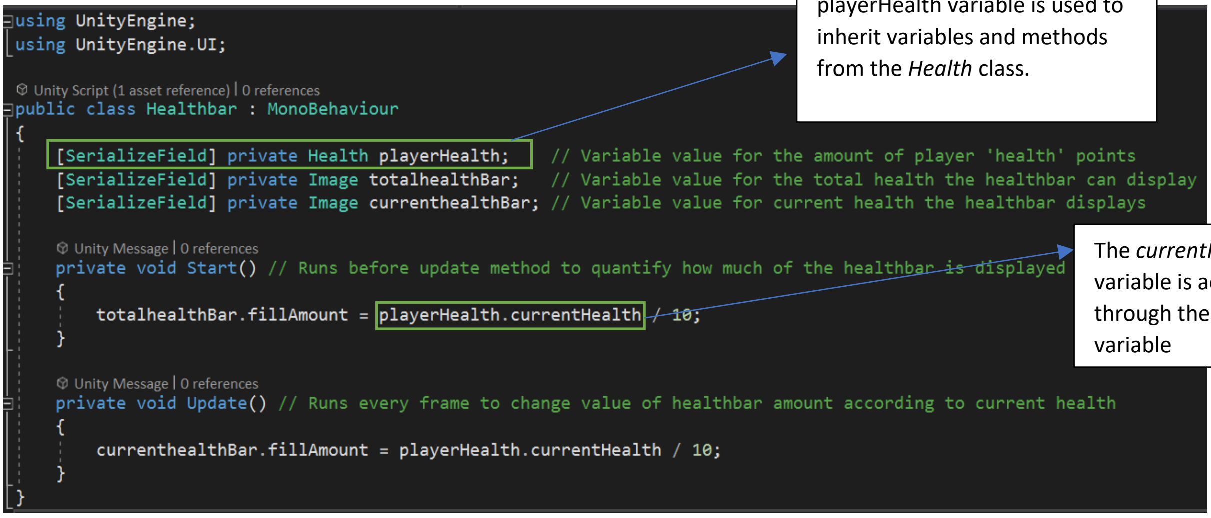
Instead of *if* statements, switch case statements could also be used but they were not used since there are only a maximum of 2 different cases possible for every *if* statement used therefore *if* statements are a more efficient solution.

4. Inheritance

Inheritance allows for new classes to be created which can be reused and extended in other classes (Wagner, 1). When the player loses health or reaches the end of the game, that has to be communicated with the UI component to display the health or the game completion screen in order to validate success criteria 5. Thus it is important for all the different components to work together such as between the classes UIManager and Health. By defining the playerHealth as a object of class 'Health', the class is extended and methods and variables in the Health class can be used by referencing through the variable. Ex: playerHealth.currentHealth is used to access the float variable from the health class.



```
public class Health : MonoBehaviour
{
    [Header ("Health")]
    [SerializeField] private float startingHealth; // Variable value for initial health
    public float currentHealth; // Public variable for current health
    public Transform Checkpoint;
```



```
using UnityEngine;
using UnityEngine.UI;

Unity Script (1 asset reference) | 0 references
public class Healthbar : MonoBehaviour
{
    [SerializeField] private Health playerHealth;   // Variable value for the amount of player 'health' points
    [SerializeField] private Image totalhealthBar;   // Variable value for the total health the healthbar can display
    [SerializeField] private Image currenthealthBar; // Variable value for current health the healthbar displays

    Unity Message | 0 references
    private void Start() // Runs before update method to quantify how much of the healthbar is displayed
    {
        totalhealthBar.fillAmount = playerHealth.currentHealth / 10;
    }

    Unity Message | 0 references
    private void Update() // Runs every frame to change value of healthbar amount according to current health
    {
        currenthealthBar.fillAmount = playerHealth.currentHealth / 10;
    }
}
```

playerHealth variable is used to inherit variables and methods from the *Health* class.

The *currentHealth* variable is accessed through the *playerHealth* variable

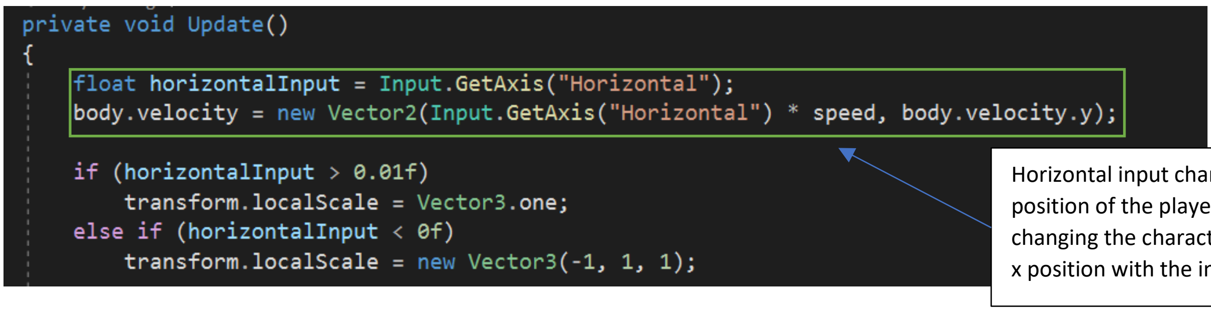
A private static class could also have been defined for this purpose, however since the *Health* class has variable values and methods which are continuously run, inheritance is more suitable.

5. Event Handling

In this program, event handling is implemented in the functioning of the player's movement and how the character interacts with other game objects. All the events are necessary for success criteria 1, 2, 3, 6 and 7 by checking for events continuously through the update() method.

Events handled:

Player Movement-



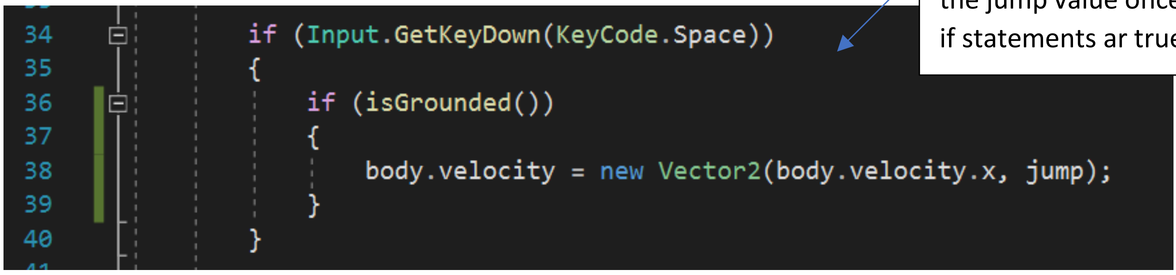
```
private void Update()
{
    float horizontalInput = Input.GetAxis("Horizontal");
    body.velocity = new Vector2(Input.GetAxis("Horizontal") * speed, body.velocity.y);

    if (horizontalInput > 0.01f)
        transform.localScale = Vector3.one;
    else if (horizontalInput < 0f)
        transform.localScale = new Vector3(-1, 1, 1);
```

Horizontal input changes the position of the player by changing the character's vector x position with the input value

The Input.GetAxis() method is used to check for inputs to the program and "horizontal" specifies the right and left direction keys. When the input is received, the body.velocity, which refers to the character, is set as the direction of input received multiplied by the variable speed. This changes the horizontal location of the character and hence allows movement.

Player Jump-

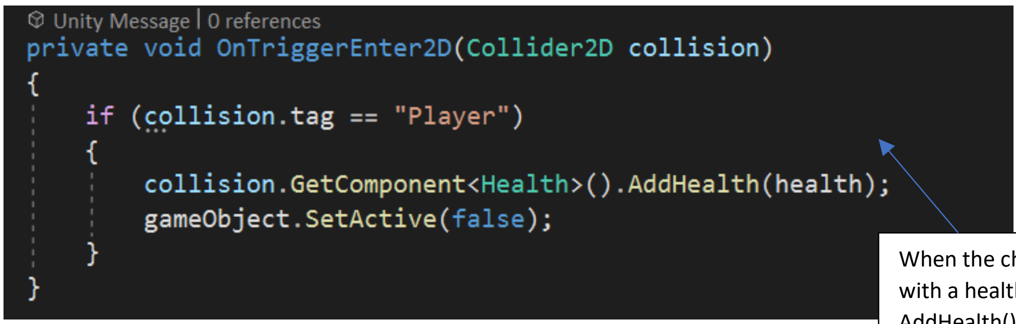
The jump movement occurs by changing the character's y position with the jump value once both if statements ar true



```
if (Input.GetKeyDown(KeyCode.Space))
{
    if (isGrounded())
    {
        body.velocity = new Vector2(body.velocity.x, jump);
    }
}
```

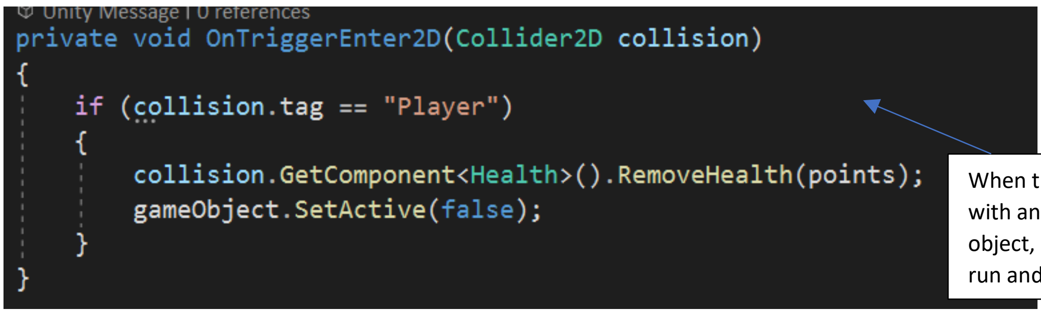
To determine the player jump, a nested if() statement is used. The first if statement checks for user input when the spacebar is pressed, and the nested if statement checks for the isGrounded() [In Appendix] to check if the player is in contact with the ground. If both statements are true then the code is run, which changes the y velocity to the jump variable which enables the player to jump.

Player pickups-



```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "Player")
    {
        collision.GetComponent<Health>().AddHealth(health);
        gameObject.SetActive(false);
    }
}
```

When the character collides with a healthy food object, AddHealth() is run and the object is disabled

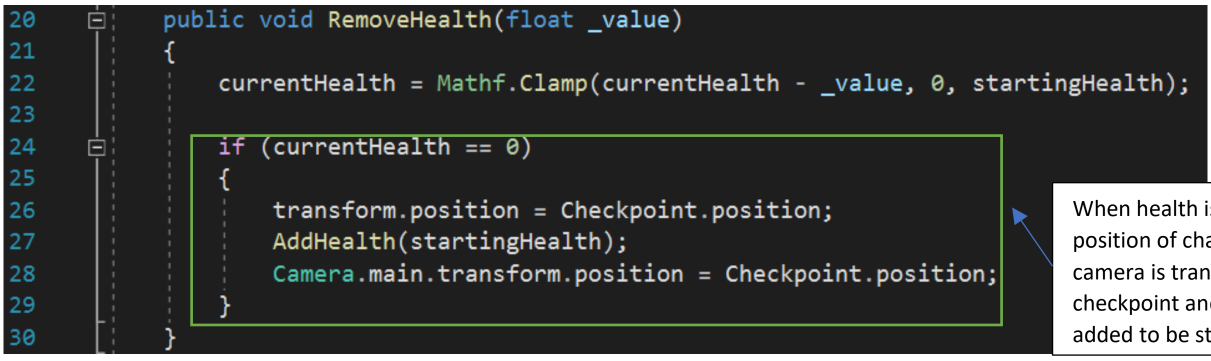


```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "Player")
    {
        collision.GetComponent<Health>().RemoveHealth(points);
        gameObject.SetActive(false);
    }
}
```

When the character collides with an uhealthy food object, RemoveHealth() is run and the object is

The "Player" tag in both codes to identify when the player touches an object using the collision.tag parameter. A tag is used in Unity to differentiate different types of game objects. The collision parameter detects 2D collisions, in this case using tags. The player game object is given this tag in the Unity editor which allows the code to recognize the collision.

Respawn-



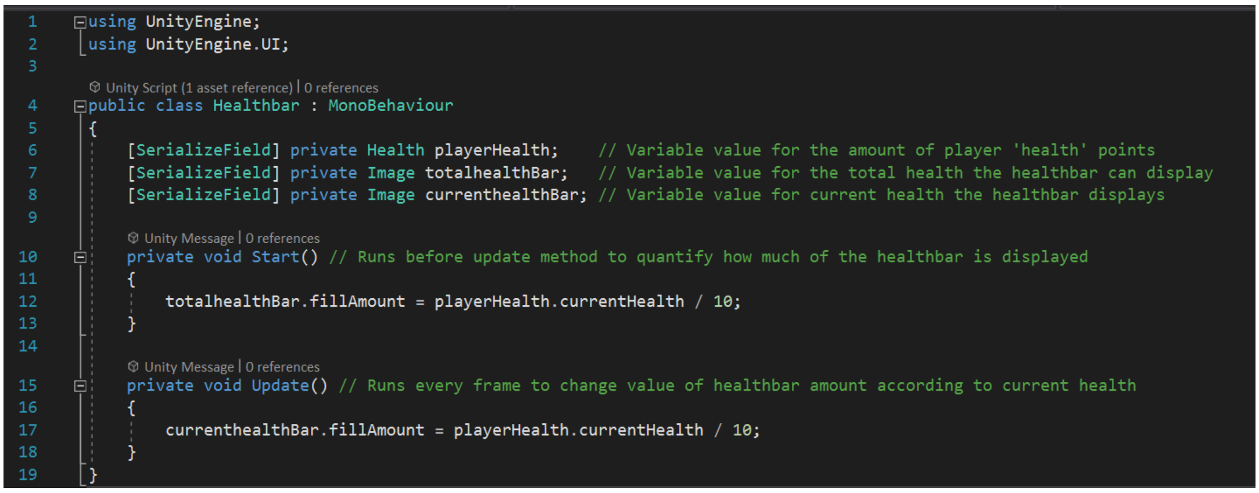
```
public void RemoveHealth(float _value)
{
    currentHealth = Mathf.Clamp(currentHealth - _value, 0, startingHealth);

    if (currentHealth == 0)
    {
        transform.position = Checkpoint.position;
        AddHealth(startingHealth);
        Camera.main.transform.position = Checkpoint.position;
    }
}
```

When health is 0, the position of character and camera is transformed to the checkpoint and health is added to be startingHealth

When the playerHealth value reaches 0, the if statement in the Health class checks this and runs a set of code. This is checked by the logical operator '=='.

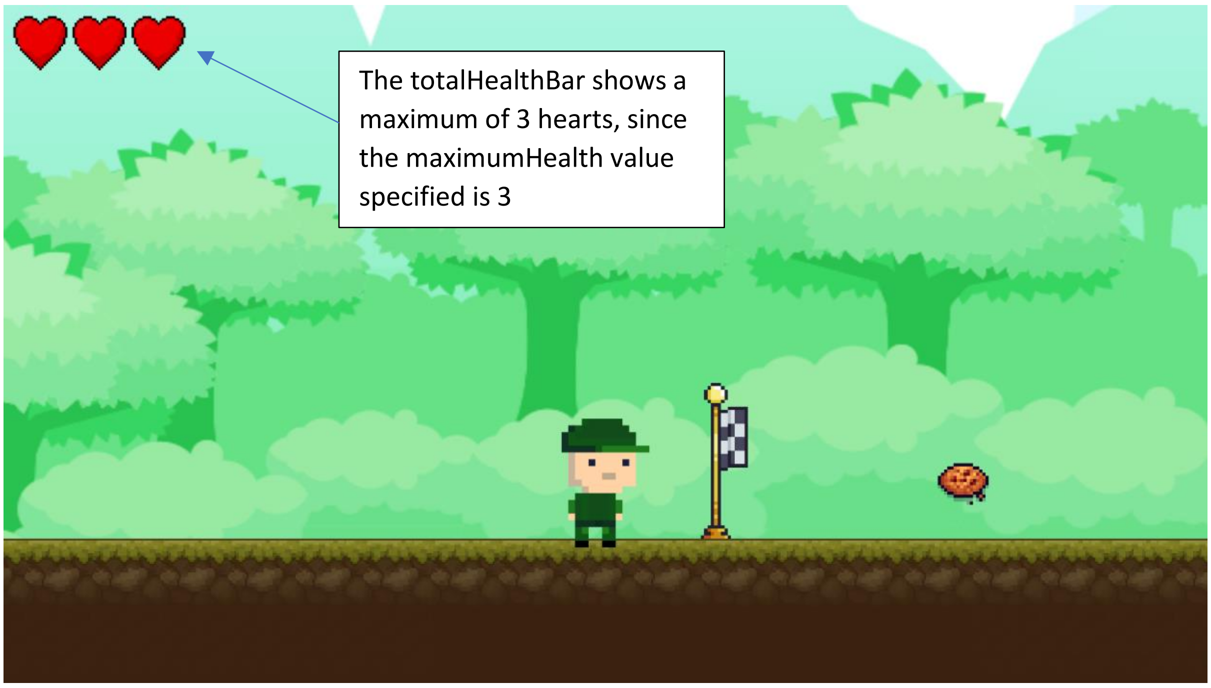
6. GUI

GUI refers to graphical user interface which is a part of a program which allows for communication between the user and the system using symbols and pointing devices ("Graphical User Interface.", 1). This is important for success criteria 5 & 7 to communicate with the user the amount of health and the game over screen.

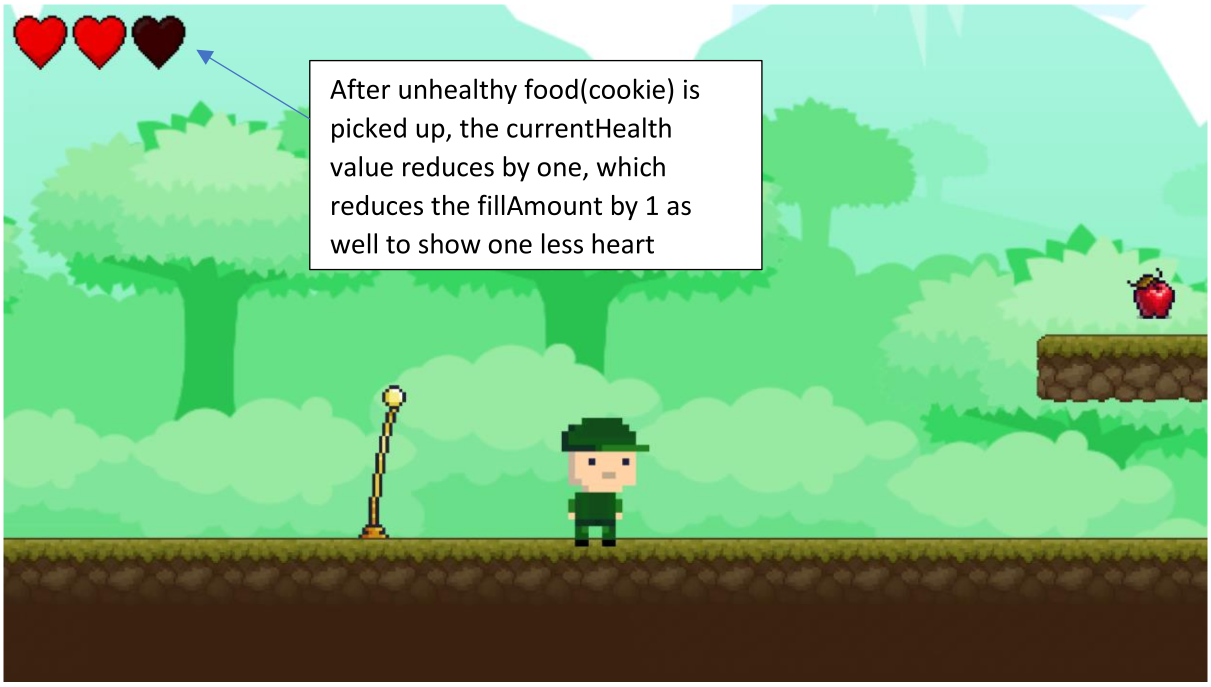


```
using UnityEngine;
using UnityEngine.UI;

Unity Script (1 asset reference) | 0 references
public class Healthbar : MonoBehaviour
{
    [SerializeField] private Health playerHealth;    // Variable value for the amount of player 'health' points
    [SerializeField] private Image totalhealthBar;   // Variable value for the total health the healthbar can display
    [SerializeField] private Image currenthealthBar; // Variable value for current health the healthbar displays

    Unity Message | 0 references
    private void Start() // Runs before update method to quantify how much of the healthbar is displayed
    {
        totalhealthBar.fillAmount = playerHealth.currentHealth / 10;
    }

    Unity Message | 0 references
    private void Update() // Runs every frame to change value of healthbar amount according to current health
    {
        currenthealthBar.fillAmount = playerHealth.currentHealth / 10;
    }
}
```

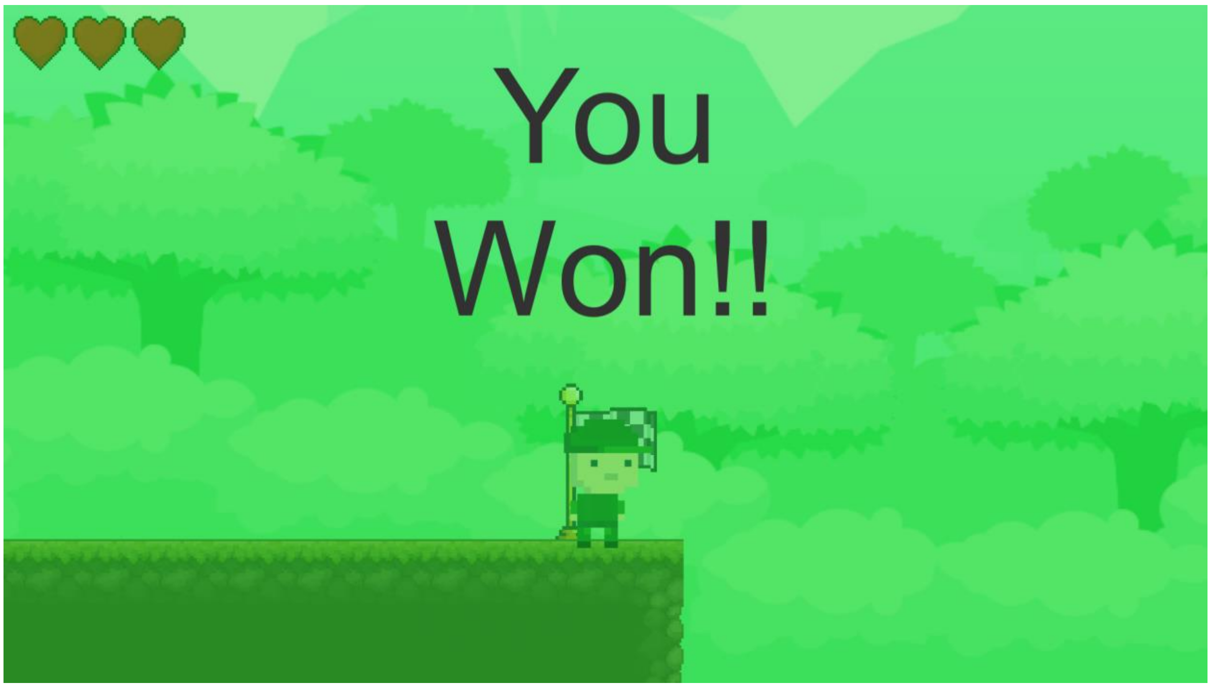
The GUI is defined by the two images totalHealthBar and currentHealthBar. The image used for both of them is the same image which consists of 10 inidividul hearts. Both fields are Serialized which means that their values can be set by the user and in the program.


The totalHealthBar shows a maximum of 3 hearts, since the maximumHealth value specified is 3

The code in line 12 uses the method of fillAmount, which determines how much of the image is shown. The totalHealthBar image is shown, and is divided into 10 equal parts. The playerHealth is used to access the currentHealthvalue from the Health class. Depending on the currentHealth, those amounts of hearts are shown out of the 10 total.


After unhealthy food(cookie) is picked up, the currentHealth value reduces by one, which reduces the fillAmount by 1 as well to show one less heart

The second part of the GUI is the “You Won!” screen. This is shown when the player reaches the end of the level.



The gameOverScreen, which is designed in the Unity game engine, is shown using the SetActive() method, after the player collides with the flag at the end, recognized through the tag “Endpoint”.

Bibliography

Wagner, Bill. “If and Switch Statements - Select Execution Path among Branches.” *If and Switch Statements - Select Execution Path among Branches.* | *Microsoft Learn*, https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements.

Wagner, Bill. “Inheritance.” *Inheritance* | *Microsoft Learn*, https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance.

“Graphical User Interface.” *Encyclopædia Britannica*, Encyclopædia Britannica, Inc., https://www.britannica.com/technology/graphical-user-interface.

Pkulikov. “Boolean Logical Operators - the Boolean and, or, Not, and XOR Operators.” *Boolean Logical Operators - the Boolean and, or, Not, and Xor Operators* | *Microsoft Learn*, https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators.

Pkulikov. “Comparison Operators - Order Items Using the Greater than and Less than Operators.” *Comparison Operators - Order Items Using the Greater than and Less than Operators* | *Microsoft Learn*, https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/comparison-operators.

“C# - Encapsulation.” *TutorialsPoint*, TutorialsPoint, 28 July 2021, https://www.tutorialspoint.com/csharp/csharp_encapsulation.htm.