Unit Tests

- ### Background

  Since Burger Breakout is made in the Unity game engine, unity supplies a few tools for making unit tests possible. It uses a NUnit testing framework which works for the C# files. There are two types of tests that are able to run in Unity which are PlayMode tests and EditMode tests. The EditMode tests will interact with whatever environment you are currently on in the editor, and the PlayMode tests will interact with the entire game regardless of where you are in the editor.

- ### Currently Done

  Currently we have two folders set up for unit tests for both PlayMode and EditMode. We have written some unit tests for the in-game menu and the player to ensure they work with our future changes. For example, we have some unit tests in EditMode which assure that all the necessary dependencies are present in a level environment for it to run correctly. We also have some other tests in PlayMode which create mocked clicks on the UI buttons to ensure that their functionality is still intact.

  Here are screenshot examples of some of our unit tests...

```
    [UnityTest]
    0 references
    public IEnumerator MenuCanvasShouldNotBeNullOnLoad()
    {
        SceneManager.LoadScene("Menu");
        yield return null;
        var mainMenuCanvas = GameObject.Find("Canvas");

        Assert.IsNotNull(mainMenuCanvas);
    }

    0 references
    public IEnumerator MainMenuShouldNotBeNullOnLoad()
    {
        SceneManager.LoadScene("Menu");
        yield return null;
        var mainMenu = GameObject.Find("MainMenu");

        Assert.IsNotNull(mainMenu);
    }

    [UnityTest]
    0 references
    public IEnumerator NewGameButtonShouldNotBeNull()
    {
        SceneManager.LoadScene("Menu");
        yield return null;
        var newGameButton = GameObject.Find("NewGameButton");

        Assert.IsNotNull(newGameButton);
    }

    [UnityTest]
    0 references
    public IEnumerator NewGameButtonShouldLoadLevel1OnClick()
    {
        SceneManager.LoadScene("Menu");
        yield return null;
        var eventSystem = EventSystem.current;
        var newGameButton = GameObject.Find("NewGameButton");
        ExecuteEvents.Execute(newGameButton.gameObject, new BaseEventData(eventSystem), ExecuteEvents.submitHandler);

        yield return null;
        var newScene = SceneManager.GetActiveScene().name;

        Assert.AreEqual("Level1", newScene);
    }
```

```
public class PlayerTests
{
    0 references
    public GameObject player;
    [Test]
    0 references | Run Test | Debug Test
    public void CharacterShouldBeInGameSpaceOnOpen()
    {
        var player = GameObject.FindGameObjectWithTag("Player");
        Assert.IsNotNull(player);
    }

    [Test]
    0 references | Run Test | Debug Test
    public void CharacterShouldHaveRigidBody2D()
    {
        var player = GameObject.FindGameObjectWithTag("Player");
        Assert.IsNotNull(player.GetComponent<Rigidbody2D>());
    }

    [Test]
    0 references | Run Test | Debug Test
    public void CharacterShouldHaveSpriteRenderer()
    {
        var player = GameObject.FindGameObjectWithTag("Player");
        Assert.IsNotNull(player.GetComponent<SpriteRenderer>());
    }

    [Test]
    0 references | Run Test | Debug Test
    public void CharacterShouldHaveBoxCollider2D()
    {
        var player = GameObject.FindGameObjectWithTag("Player");
        Assert.IsNotNull(player.GetComponent<BoxCollider2D>());
    }

    [Test]
    0 references | Run Test | Debug Test
    public void CharacterShouldHaveAnimator()
    {
        var player = GameObject.FindGameObjectWithTag("Player");
        Assert.IsNotNull(player.GetComponent<Animator>());
    }
}
```

- **Plan For Future**

  We plan to continue making these unit tests to test the entirety of the game so that we do not have to manually test it in order to ensure it works as we want it to.

Use Case Tests

- **Plan For Future**

  Going forwards, each time we create a new use case, we will create an alternative section for it so that we can better write the unit tests to cover valid and invalid cases. Not all of our unit tests currently cover all the valid/invalid scenarios of all the use cases so we will likely need to expand those.

Here are some example use cases...

Use Case	Load Game
Actors	- Player

Actor:	**System:**
1. Actor clicks UI button 3. Actor chooses selection	2. System opens 'Yes' or 'No' prompt 4. System loads data 5. System closes prompt 6. System opens game display

Alternative:
3a: Actor decides to close game instead
3a1: System closes prompt and shuts game down
3b. Actor clicks 'No'
3b1. System closes prompt
4a. Actor does not have any data in system
4a1. System tells actor they don't have data and starts new game for them
4b: System can't load player's data for outside reason
4b1: System tells player it can't load data and to try again

Test Cases:
1. User clicks ‘Yes’, system finds data and loads game
2. User clicks ‘No’, system closes prompt
3. User exits game, system closes prompt and shuts down game
4. Actor has no data in system, system starts new game
5. Actor has data, system loads the data
6. System database has issues, prompts player to try again

Use Case	Change Controls
Actors	- Player

Actor:	**System:**
1. Actor clicks pause game 3. Actor clicks change control UI 5. Actor clicks control 7. Actor presses desired input	2. System opens pause screen UI 4. System opens change control UI 6. System listens for input 8. System verifies input 9. System binds input to action

Alternative:
5a. Actor decides to unpause game instead
5a1. Game closes pause and change control UI and resumes game state
8a. Input overlaps other action’s input
8a1. System tells actor that key is already bound to another action

Test Cases:
1. Actor unpauses game, system restores game state and closes UI
2. Actor gives valid input, system rebinds the action to input key
3. Actor gives invalid input, system tells actor that key is already bound to another action

Acceptance Testing

Ex.

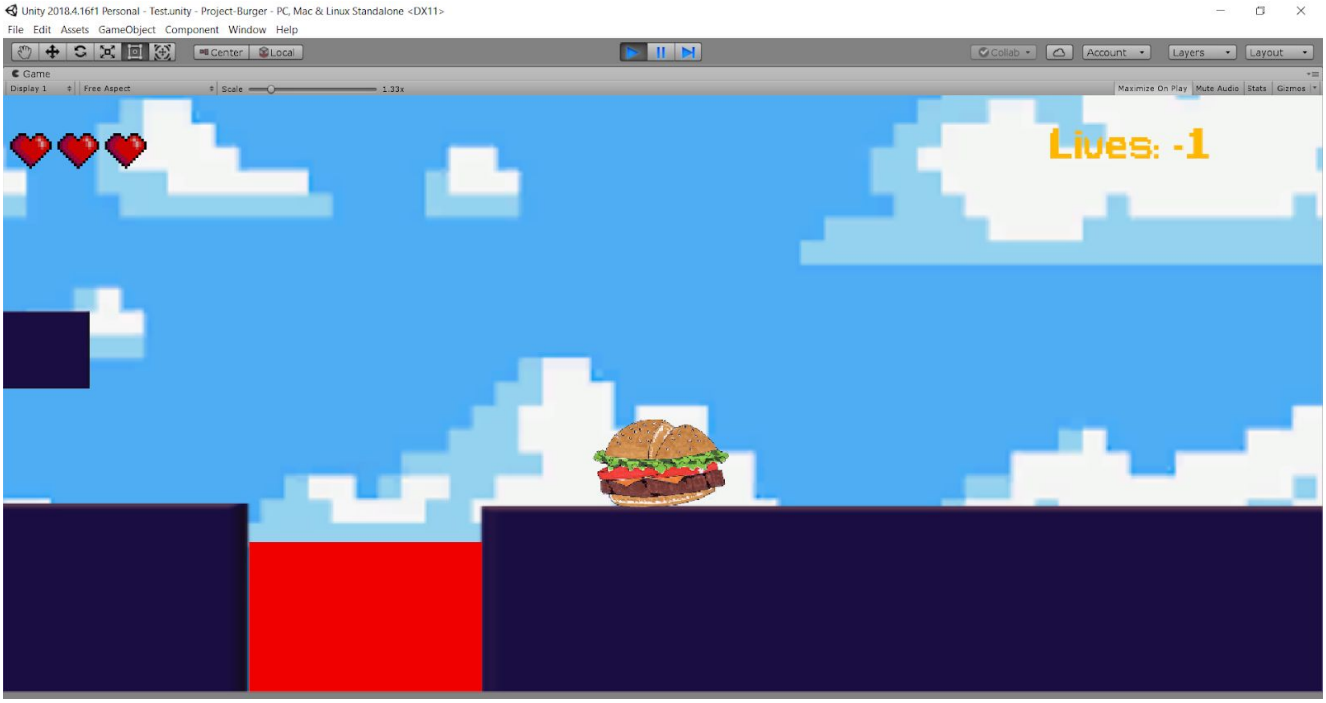
Valid	Invalid
(1) Player dies when health depleted. (2) Game Over when lives are depleted. (3) Player jumps if grounded and Space key is pressed. (4) Player moves in the horizontal direction according to key press. (5) Player instantiates at the start of the game. (6) Player is damaged upon touching an enemy/obstacle.	(7) Player does not die when health is depleted. (8) Game is not over when lives are depleted. (9) Player does not jump if grounded and Space key is pressed. (10) Player jumps when not grounded and space key is pressed. (11) Player is not instantiated at the start of the game. (12) Player is not damaged upon touching an enemy/obstacle.

- **Currently Done**

For our acceptance testing, we have performed two beta tests based on the validity conditions above. The tests were performed internally as opposed to third-parties or customers. The tests involved playthroughs of the various levels in the game. Testers played the game in the manner it was developed for, as well as in manners meant to “break” the game to reveal bugs.

Beta Test #1 - Horizontal movement is acceptable, momentary lapses in ability to jump when grounded and spacebar pressed. Player is damaged and dies when health is depleted, lives deplete when player dies but the game does not end upon depleting all lives.

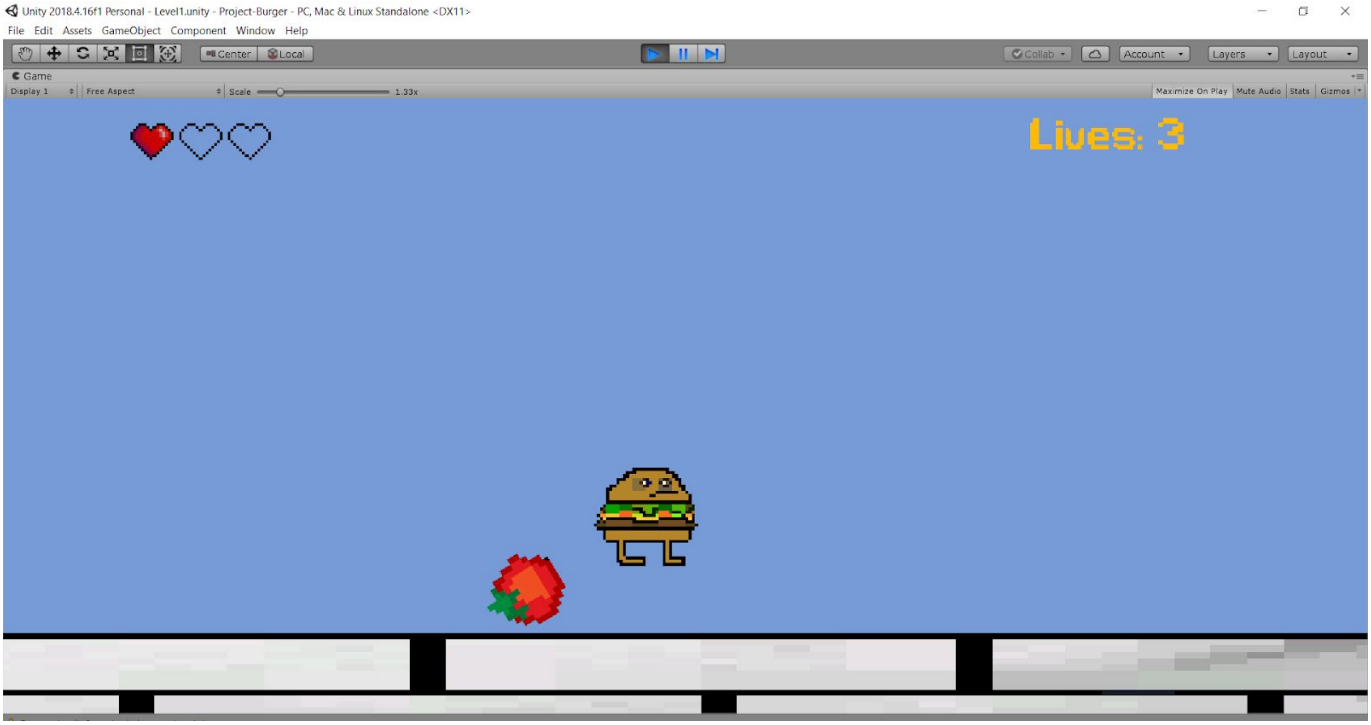
Conditions met: 1,4,5,6,8,9



Beta Test #1 - Screenshot above demonstrating game not ending when lives are depleted.

Beta Test #2 - Movement is acceptable, game ends when lives are depleted, minor issue with randomly not being able to jump when grounded and spacebar pressed persists.

Conditions met: 1,2,4,5,6,9



Beta Test #2 - Screenshot above demonstrating improvements since the previous test performed.

- **Plan For Future**

Going forward, we will conduct a public beta test involving users and third-parties, so we may receive some external feedback into the acceptability of our game. After our game is finalized and released, new content which will be released in future updates will require further closed-beta testing to ensure proper operation and function of all new features.