Quest 10 - Steps

1) Setup

Quest 10 builds off of Quest 9 and uses the same project and scene.

Import zelda_world.png and use the following Import Settings:
Sprite Mode = Multiple
Pixels Per Unity = 16
Filter Mode = Point
Compression = None

Hit Apply whenever prompted.

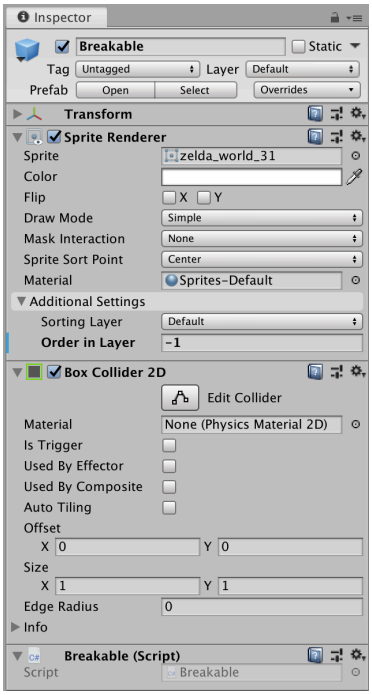
Open Sprite Editor and use the following Slice settings:
Type = Grid by Cell Size
Pixel Size X = 16
Pixel Size Y = 16

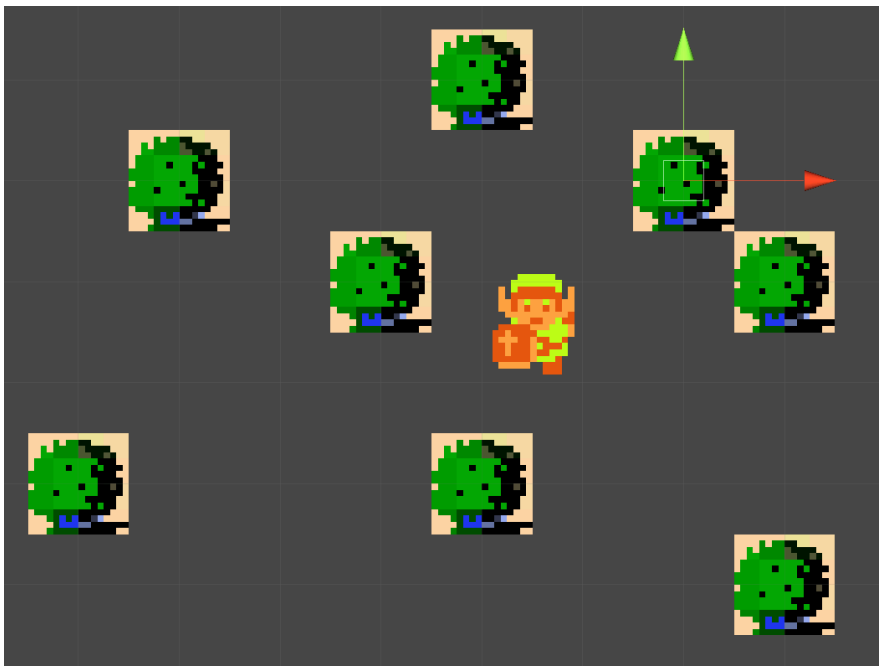
Hit Slice and Apply whenever prompted.

2) Destructible Environment

Create a new Scene game object called Breakable and attach a Sprite Renderer (with appropriate Sprite) and BoxCollider2D to it. The Sprite Component's Order in Layer should be -1. Create a Breakable.cs script and attach it to the object.

Prefab this game object and duplicate **at least 4** of them to place around the player.





Open Breakable.cs and implement a public Break function that destroys the obstacle.

```
public class Breakable : MonoBehaviour
{
    public void Break() {
        Destroy(gameObject);
    }
}
```

3) Facing Direction

In order for our player to attack, we need to keep track of what direction they are facing. An enumeration is a nice way for defining your own data type for keeping track of such information. In PlayerController.cs, create an enumeration for Direction.

```
using UnityEngine;

public enum Direction {
    Up = 0,
    Down = 1,
    Left = 2,
    Right = 3
}

public class PlayerController : MonoBehaviour
{
```

Within the PlayerController class, define a variable for keeping track of the player's facing direction. We will also need to access the SpriteRenderer soon.

```
public class PlayerController : MonoBehaviour
{
    // Outlets
    Rigidbody2D _rigidbody;
    Animator _animator;
    SpriteRenderer _spriteRenderer;

    // State Tracking
    public Direction facingDirection;
```

Fill the SpriteRenderer reference during the start event.

```
// Methods
void Start() {
    _rigidbody = GetComponent<Rigidbody2D>();
    _animator = GetComponent<Animator>();
    _spriteRenderer = GetComponent<SpriteRenderer>();
}
```

Ultimately, the animation state will drive what direction we are facing, so we will check the currently rendering player sprite to determine facing direction. Overall the flow looks something like this:

Controller Input -> Physics Forces -> Animation Blending -> Visible Sprite -> Facing Direction

Because animations happen after the Update loop, we will put our facing logic in LateUpdate which happens after animations have resolved.

```
void LateUpdate() {
    if(String.Equals(_spriteRenderer.sprite.name, "zelda1_8")) {
        facingDirection = Direction.Up;
    } else if(String.Equals(_spriteRenderer.sprite.name, "zelda1_4")) {
        facingDirection = Direction.Down;
    } else if(String.Equals(_spriteRenderer.sprite.name, "zelda1_10")) {
        facingDirection = Direction.Left;
    } else if(String.Equals(_spriteRenderer.sprite.name, "zelda1_6")) {
        facingDirection = Direction.Right;
    }
}
```

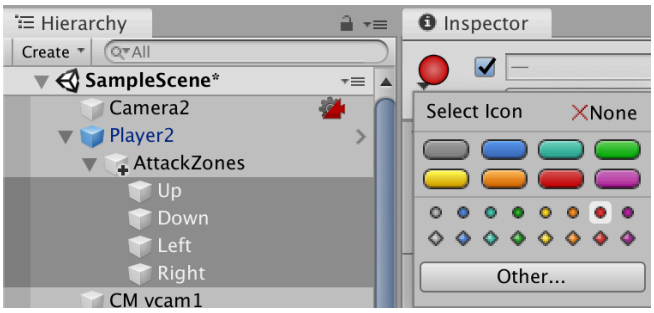
If you named your sprites differently, you MUST type your own matching names.

Playtest your game and inspect the PlayerController to check if Facing Direction updates appropriately for all 8 idle/walk scenarios.

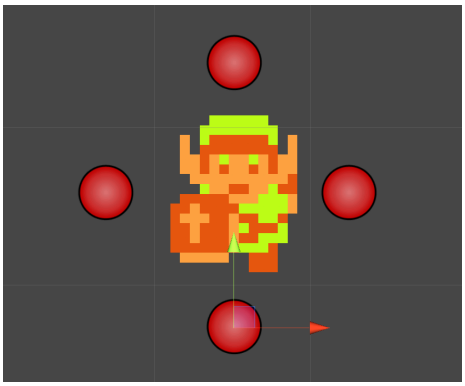
4) Attacks

Now that we know what direction the player is facing, we can define the 4 attack zones where the player will deal damage. Create an empty child object within Player and name it “AttackZones”. Make sure it’s Position XYZ are set to 0.

Next create four empty game object children under AttackZones and name them Up, Down, Left, and Right. To help with positioning, change the game object icon on these four children to the red dot.



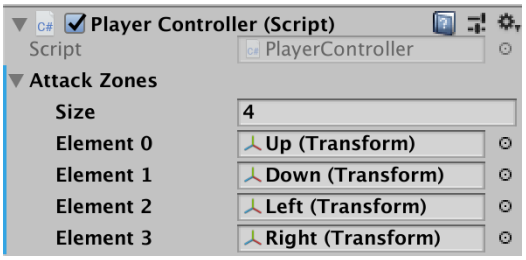
These positions will determine where the player can do damage. Arrange them around the player like the following diagram as their name suggests (up is above, down is below, etc).



PlayerController needs to keep track of these four tracking points.

```
// Outlets
Rigidbody2D _rigidbody;
Animator _animator;
SpriteRenderer _spriteRenderer;
public Transform[] attackZones;
```

Fill in the blanks for the attack zones in the inspector. The order in this array MUST MATCH the order defined by your index numbers in the enumeration.




```
using UnityEngine;

public enum Direction {
    Up = 0,
    Down = 1,
    Left = 2,
    Right = 3
}

public class PlayerController : MonoBehaviour
{
```

Whenever the player attacks, we will use Physics' OverlapCircle to check for targets under our various attack zones. For example, if the player attacks and we are facing up, we will check within a circle at attack zone 0 which represents up.

```
if(Input.GetKeyDown(KeyCode.Space)) {
    _animator.SetTrigger("attack");

    // Convert enumeration to an index
    int facingDirectionIndex = (int)facingDirection;

    // Get attack zone from index
    Transform attackZone = attackZones[facingDirectionIndex];

    // What objects are within a circle at that attack zone?
    Collider2D[] hits = Physics2D.OverlapCircleAll(attackZone.position, 0.1f);

    // Handle each hit target
    foreach(Collider2D hit in hits) {
        Breakable breakableObject = hit.GetComponent<Breakable>();
        if(breakableObject) {
            breakableObject.Break();
        }
    }
}
```

When OverlapCircleAll returns results, we loop through all of them and process each accordingly depending on what components it has.

Playtest your game and ensure you can break targets from all 4 directions.