

Adventure Game Tutorial

Difficulty: Intermediate

Source: <https://unity3d.com/learn/tutorials/projects/adventure-game-tutorial>

Platform: Windows 10 x64

Coding IDE: Visual Studio 2017 Community Edition

Unity Version: 2017.2.0f3 Personal

Contents

| | |
|---|----|
| The Player | 2 |
| Training Day Phases (Slides)..... | 2 |
| Phase 01 – The Player (Slides)..... | 3 |
| Creating the Adventure Game Tutorial Project for Phase 01 | 4 |
| Initial Build Steps..... | 4 |
| Animator State Machine | 6 |
| The Player (Continued) | 10 |
| Creating the Player Movement Script..... | 10 |
| PlayerMovement Script | 24 |
| Inventory | 27 |
| Phase 02 – Inventory (Slides) | 27 |
| Creating the Adventure Game Tutorial Project for Phase 02 | 27 |
| Initial Build Steps..... | 28 |
| Inventory Script..... | 44 |
| InventoryEditor Script..... | 45 |
| Conditions | 46 |
| ConditionCollection Script | 48 |
| ConditionCollectionEditor Script..... | 49 |
| Reactions..... | 50 |
| ReactionCollectionEditor Script | 51 |
| TextReaction Script | 53 |
| TextReactionEditor Script | 54 |
| Interactables | 54 |
| Interactable Script..... | 55 |
| InteractableEditor Script..... | 55 |
| Game State..... | 56 |

| | |
|------------------------------|----|
| SceneController Script | 57 |
| SaveData Script | 58 |

The Player

Training Day Phases (Slides)

Project Overview (Slides)

01 – 02 The Player

- Build a click to move animated character using:
 - EventSystem
 - NavMesh
 - Animator
 - Prefabs

03 Inventory

- Build a UI and Item Management System for Player Inventory
 - UI System
 - Editor Scripting

04 – 06 Interaction System

- Build a system allowing the Player to interact with the game
 - Conditions
 - Reactions
 - Interactables

04 Interaction System - Conditions

- Create a system to check the current game state
 - Scripting Patterns
 - Scriptable Objects
 - Generic Classes
 - Inheritance
 - Extension Methods

05 Interaction System - Reactions

- Create a system to perform actions based on condition state
 - Polymorphism
 - Further editor scripting
 - Serialization

06 Interaction System - Interactables

- Create a system to define what the player can interact with
 - Interactable Geometry
 - EventSystem
 - Interaction System Summary

07 Game State

- Creating a system to load scenes while preserving game state
 - Scene Manager
 - Scriptable Objects as *temporary* runtime data storage
 - Delegates
 - Lambda Expressions

Understanding the architecture of Scene Loading

Game Start

- Load a scene
 - Persistent scene that stays loaded throughout the game (first index of build) and manages:
 - Scene 1
 - Scene 2
 - Load level additive -- > Scene 1
 - Then Set Active Scene --> Scene 1
 - Unload active scene --> scene 1
 - Load level additive --> scene 2
 - Set active scene --> scene 2

Phase 01 – The Player (Slides)

The Player (Slides)

The Brief


- Create a **click to move** humanoid character for an adventure game
- When the user **clicks on the ground**, the character must **move to that location**
- **Interactable** object in the scene will be provided to our team for our character to interact with
 - We will get an opportunity to build parts of these later today!
- When the **Interactable** is clicked on, the character should approach the **interactionLocation** of the **Interactable**
 - **interactionLocation** is a **Transform** value saved as part of the **Interactable**
- On arrival, the character should match the **position** and **rotation** of the **interactionLocation** and call the **Interact** function of the **Interactable**
- When required, the character must play various **animations** in response to **trigger parameters** sent by the **Interactables**; specifically the supplied **trigger parameters**: *HighTake, MedTake, LowTake* and *AttemptTake*
- The character cannot be allowed to move while these animations are playing
 - Locked in place while interactions are playing

Approach

- **NavMesh** to define the walkable areas in the level
 - **EventSystem** to detect and handle user input and scripted interaction
 - **Animator** state machine to control and play all of the character animations; including idle, walking, and interaction
 - **Prefab** system to save the character so it can be easily added and used in any scene in the game
-

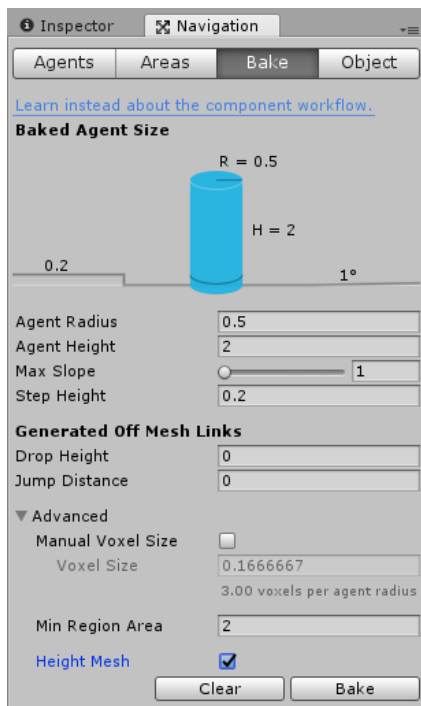
Creating the Adventure Game Tutorial Project for Phase 01

To create a project for Phase 01 of the Adventure Game Tutorial, do the following:

- Start Unity 2017.2.0f3
- Click the **New** link to create a new 3D project named **Unity_Tutorial_AdventureGame_Phase01** (or some other meaningful name) and then click the **Create project** button
- Click on the **Asset Store** tab, type **Adventure Game Tutorial** in the search field at the top of the **Asset Store** pane, and click the Search button  (or press **Enter**)
- Scroll down and choose **1/6 – Adventure Tutorial – The Player** from the results list
- Click the blue **Download** button
- Click the **Import** button on the **Importing Complete Project** popup message
- Click the **Import** button on the **Import Unity Package** popup message

Initial Build Steps

- Navigate to **Project** pane
- Expand the **Scenes** folder
- Drag the **SecurityRoom** scene into the Hierarchy
- In the Hierarchy pane, right-click the default **Untitled** scene was created when Unity initially opened and select **Remove Scene** from the context list
- Expand the **SecurityRoom** GameObject in the Hierarchy and select the **SecurityRoomEnvironment** GameObject
 - Check **Static** and click **Yes, change children** on the **Change Static Flags** popup message
 - Expand the **SecurityRoomEnvironment** GameObject
 - Multi-select the following children:
 - **BlackUnlit**
 - **FloorLightGlow**
 - **HologramLight**
 - **HologramLight02**
 - **SecurityGateBeams**
 - Uncheck the **Static** checkbox for the multi-selected children
- Open the **Navigation** pane by clicking **Window** from the menu items at the top and choosing **Navigation**
 - Select the **Bake** panel
 - Set the **Max Slope** to **1**
 - Set the **Step Height** to **0.2**
 - Under **Advanced**, set **Height Mesh** to **true**
 - Press the **Bake** button



The Event System (Slides)

- We need to be able to interact with our environment
- An event system has 3 requirements. Something to:
 - Send events
 - Physics Raycaster component attached to the camera
 - Every frame, it recasts into the scene looking for physics-based objects
 - Receive events
 - Colliders and Event Triggers
 - When a collider is hit with a raycast, then it will signal to the event trigger that an event has happened
 - Manage events
 - The EventSystem
 - GameObject that Unity creates for us (don't worry about it...)

Add an **Event System** with an **Audio Listener** component:


- Navigate to the **Hierarchy**
- Click the **Create** dropdown, select **UI**, and then select **Event System**
- In the Inspector pane, click **Add Component**
- Select **Audio** and then choose **Audio Listener**
- Save the scene!!!

Add a **Physics Raycaster** to the **SecurityRoom** GameObject:

- In the Hierarchy, expand the **SecurityRoom** GameObject
- Expand the **CameraRig** GameObject and select the **Camera** GameObject

- In the Inspector pane, click **Add Component**, select **Event**, and choose **Physics Raycaster** (Do not add a **Physics 2D Raycaster**!!)

Add the **Mesh Collider** to the **SecurityRoom** GameObject:

- In the Hierarchy, select the **SecurityRoom** GameObject
- Click **Add Component**, select **Physics**, and then choose **Mesh Collider**
- Using the Circle Select button , set the **Mesh** field to **SecurityRoomMeshCollider**


Add an **EventTrigger** component:

- In the Hierarchy, select the **SecurityRoom** GameObject
- Click **Add Component**, select **Event**, and choose **Event Trigger**
- Click **Add New Event Type** and select **PointerClick**
- Click the **+** button to add an event
- Leave the event **Object** field empty (e.g., leave the field displaying **None (Object)**)

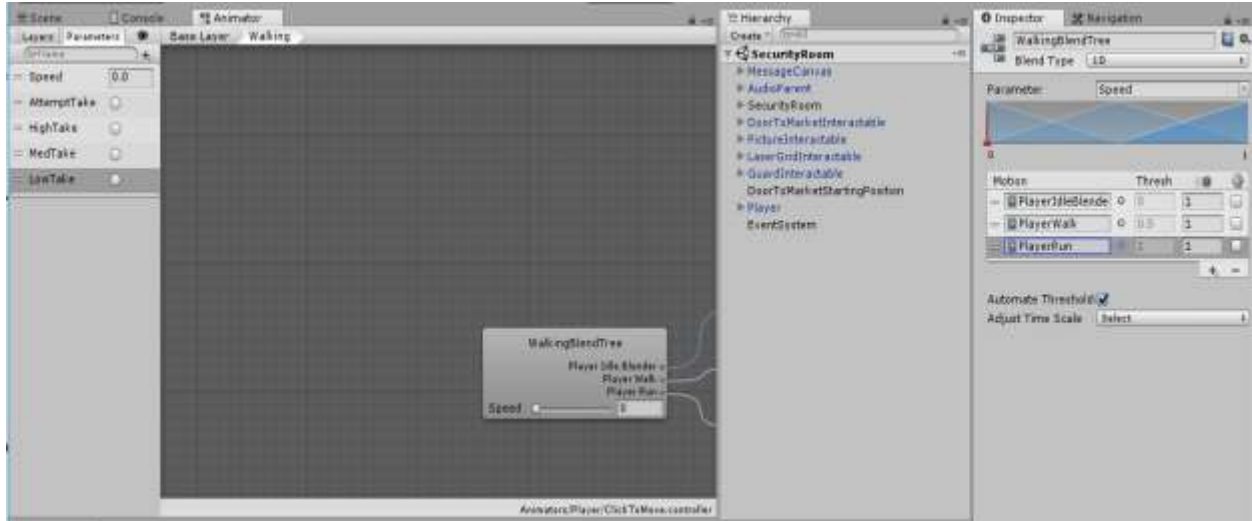
Save the scene!

Animator State Machine

Now that there is a NavMesh that defines the area the player can move along with the beginnings of the OnGroundClick interaction, it is time to move onto the player and creating the animator state machine.

- Navigate to the **Project** pane
- Select and expand the **Animators** folder
- Click the **Create** dropdown, select **Folder**, and name the new folder **Player**
- With the **Player** folder selected, click the **Create** dropdown and select **Animator Controller**
- Name the new controller **ClickToMove**
- Double-click on the **ClickToMove** to open the Animator pane to create the new parameters
- Select the **Parameters** tab
- Select the **+** button to create a new parameter
- Create a **Float** parameter named **Speed**
- Create four additional **Trigger** parameters using the **+** dropdown
 - Name the first trigger **AttemptTake**
 - Name the second trigger **HighTake**
 - Name the third trigger **MedTake**
 - Name the fourth trigger **LowTake**
- Navigate to the Animator pane's layout area
 - Right-click on the background
 - Select **Create State** and then select **From New Blend Tree**
 - Select the new **Blend Tree** and in the Inspector, change the name to **Walking**
 - Double-click on **Walking** to edit it
 - In the Inspector,
 - Rename **Blend Tree** to **WalkingBlendTree**
 - Click the **+** button below **Parameter** and select **Add Motion Field**
 - Repeat twice more to create a total of three motion fields
 - Using the Circle Select button  next to each of the motion fields, import the following animations:

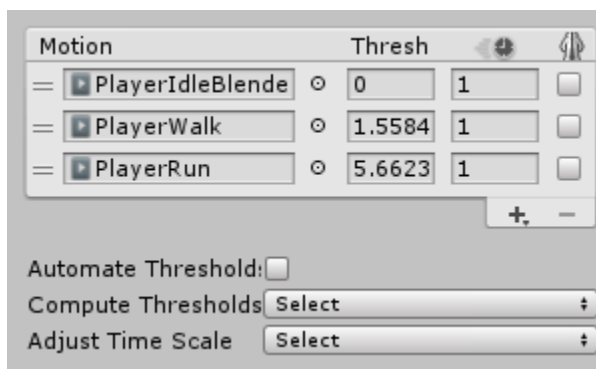
- First animation selected is **PlayerIdleBlender** to blend between the stationary player and the walking player animations
- Second animation selected is **PlayerWalk**
- Third animation selected **PlayerRun**



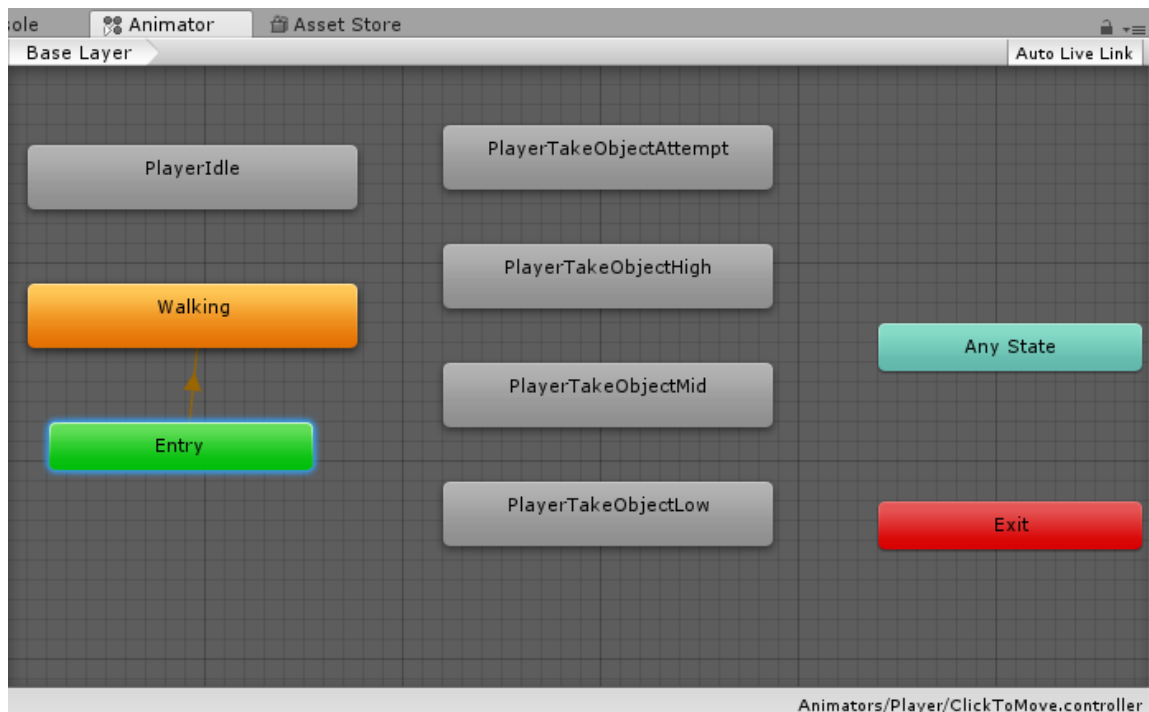
Within Unity, there is a concept of *Root Motion*. Each of the animations can move the character. In order to do that, a more complete system will be used through scripting. For now, be sure that when the speed is set, they accurately reflect how fast the animations play. For example, the **PlayerIdleBlender** will not move the character at all and has a threshold of zero. The **PlayerWalk** animation is a little faster, so that will have a slightly higher speed threshold. And the **PlayerRun** will be even faster, so that will be at the top threshold. Since Unity knows nothing about the three animations, it divides them evenly to make the thresholds **0**, **0.5**, and **1** which results in a relatively even graph.

To blend all the animations together in a different way than the default values:

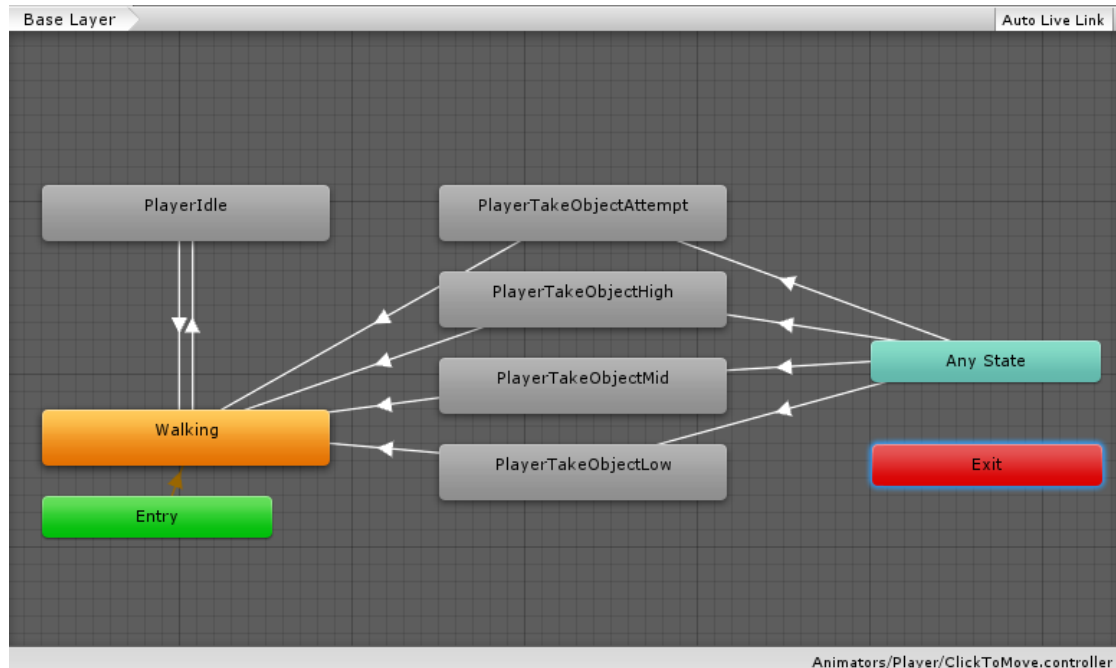
- Uncheck **Automate Threshold** so Unity does not divide them evenly.
- The thresholds will be based on the speed of the animations, and Unity will do that through the **Compute Thresholds** dropdown when **Speed** is selected. Here, Unity sets the values and then "walks away".



- In the Animator pane, return to the base layer by selecting the **Base Layer** tab in the upper, left corner of the Animator layout pane
- Navigate to the **Project** pane
 - Expand the **Animations** folder
 - Expand the **Player** folder to view the actual animations
 - Multi-select **Idle**, **TakeObjectAttempt**, and **TakeObjects** and drag into the layout of the Animator pane
 - Arrange the animations as shown:



- Now, create the transitions between **Walking** and **PlayerIdle**
 - Right-click on **Walking**, select **Make Transition**, and then left-click on **PlayerIdle**
 - Right-click on **PlayerIdle**, select **Make Transition**, and then left-click on **Walking**
- Create the transitions between **Any State** and the Attempt Takes:
 - Right-click on **Any State**, select **Make Transition**, and left-click on **PlayerTakeObjectAttempt**
 - Right-click on **Any State**, select **Make Transition**, and left-click on **PlayerTakeObjectHigh**
 - Right-click on **Any State**, select **Make Transition**, and left-click on **PlayerTakeObjectMid**
 - Right-click on **Any State**, select **Make Transition**, and left-click on **PlayerTakeObjectLow**
- Create the transitions between all of the Attempt Takes to **Walking**
 - Right-click on **PlayerTakeObjectAttempt**, select **Make Transition**, and left-click on **Walking**
 - Right-click on **PlayerTakeObjectHigh**, select **Make Transition**, and left-click on **Walking**
 - Right-click on **PlayerTakeObjectMid**, select **Make Transition**, and left-click on **Walking**
 - Right-click on **PlayerTakeObjectLow**, select **Make Transition**, and left-click on **Walking**



- Select the transition between **Walking** and **PlayerIdle**
 - In the Inspector, uncheck **Has Exit Time**
 - **Has Exit Time** only allows transition once the animation is a certain way through
 - Add a condition (required) by clicking the + button under **Conditions** and leaving it set to **Speed**
 - Change the **Greater** to **Less**
 - Change the value from **0** to **0.1**
- Select the transition between **PlayerIdle** and **Walking**
 - In the Inspector, uncheck **Has Exit Time**
 - Add a condition (required) by clicking the + button under **Conditions** and leaving it set to **Speed**
 - Change the **Greater** to **Less**
 - Change the value from **0** to **0.1**
- Select transition from **Any State** to **PlayerTakeObjectAttempt**
 - In the Inspector, add a condition using the + button and set to **AttemptTake**
- Select transition from **Any State** to **PlayerTakeObjectHigh**
 - In the Inspector, add a condition using the + button and set to **HighTake**
- Select transition from **Any State** to **PlayerTakeObjectMid**
 - In the Inspector, add a condition using the + button and set to **MedTake**
- Select transition from **Any State** to **PlayerTakeObjectLow**
 - In the Inspector, add a condition using the + button and set to **LowTake**

The brief states that the character should not move while in the "taking" states. To identify whether the character is in one of those states is to tell Unity that the character can only move when in one of the **Locomotion** states. To do this, the **Walking** and the **PlayerIdle** states are set to the **Locomotion** tag.

- Select **PlayerIdle** and in the Inspector, type **Locomotion** in the **Tag** field
- Select **Walking** and in the Inspector, type **Locomotion** in the **Tag** field
- Navigate to the **Scene** view and save the scene

Now, the Player prefab work:

- Navigate to the **Project** pane and expand the **Models** folder
 - Drag the **Player** model asset into the Hierarchy
 - Set the **Layer** on the **Player** GameObject to **Characters** and click **Yes, change children** popup message
 - Set the **Position** to **(-0.7, 0.0, 3.5)**
 - Set the **Rotation** to **(0.0, 180, 0.0)**
- Set the **Animator Controller** field using the Circle Select button to choose **ClickToMove**
- Click **Add Component**, select **Navigation**, and choose **Nav Mesh Agent**
 - Change the **Speed** property to **2**
 - Change the **Acceleration** property to **20**
 - Change the **Stopping Distance** property to **0.15**
- Drag the **Player** GameObject from the Hierarchy into the **Prefabs** folder in the **Project** pane
- Save the scene!

The Player (Continued)

Creating the Player Movement Script

Now, create the script to control the Player's movement.

- Navigate to the **Project** pane
- Expand the **Scripts** folder
- Select and expand the **MonoBehaviours** folder
- From the **Create** dropdown list, select **Folder**, and name the new folder **Player**
- With the new **Player** folder selected, click the **Create** dropdown list, and select **C# Script**
- Rename the new script to **PlayerMovement**
- Expand the **Prefabs** folder
- Drag the new **PlayerMovement** script onto the **Player** GameObject in the **Prefab** folder
- Select the **Player** prefab in the **Prefab** folder
- In the Inspector pane, double-click the **PlayerMovement** script to open it for editing in Visual Studio or MonoDevelop

What the **PlayerMovement** script needs to do:

- Normally, the player is moving around and aiming for a destination
- Since the animations are not perfect, some fine control is needed for approaching a destination
- When the player is within a certain distance of the destination, take finer control of the character to transition from moving to slowing using scripting
- Within a very small radius of the destination (right next to the destination), the player will be stopping by setting the character's location to the destination location, which is important when reaching for an item to have the character in the correct location and rotation rather than reaching into empty space

- Normally, the NavMeshAgent would be controlling the movement, but here the **PlayerMovement** script will be doing those operations

In the **PlayerMovement** script:

- Remove the **Start()** and **Update()** functions
- To use the **Event System** in Unity 5.5 and later, add two new namespaces at the top of the script:

```
using UnityEngine.AI; // To use EventSystem
using UnityEngine.EventSystems; // To use BaseEventData parameter
```

Now, to make some variables and write some functions for controlling the player's movement and exerting fine control. More variables and functions will be added as needed.

- Add the following **public** variables to the top of the script:

```
public Animator animator;
public NavMeshAgent agent;
```

- Create a **private Start()** function for setup

```
private void Start()
{
    // Setup stuff...
}
```

- In the **Start()** function, turn off the NavMeshAgent's ability to rotate the character

```
agent.updateRotation = false;
```

- As the player should not move when interacting, use a coroutine with a **private WaitForSeconds** variable to determine how long the player is going to wait for and a **public float** variable with a default value to determine how long that WaitForSeconds span will be. Create the two new variables above the **Start()** function:

```
public float inputHoldDelay = 0.5f;

private WaitForSeconds inputHoldWait;
```

- Return to the **Start()** function and set up the inputHoldWait using the based on the inputHoldDelay variable:

```
inputHoldWait = new WaitForSeconds(inputHoldDelay);
```

- Next, the character will be moving around in the scene, so the destination will need to be determined. For that, create a Vector3 variable above the **Start()** function for storing the destination the player is aiming for:

```
private Vector3 destinationPosition;
```

- When the player is close to the destination, the `Stopping()` function will be called and the player's position set, or snapped, to that destination. So in the `Start()` function, add the following line to set the destination position to the current position:

```
destinationPosition = transform.position;
```

- Next, create another `private` function for controlling how fast the player will move:

```
private void OnAnimatorMove()
{
    // NavMeshAgent velocity control
}
```

- Normally, the NavMeshAgent moves the character or the Animator moves the character using root motion. Here, combine those two things together by setting the speed of the NavMeshAgent based on how fast the Animator wants to move. The speed of the Animator is set based on calculations. The Animator will move based on how fast the player is going and assign the root motion accordingly. Instead of using the root motion directly, `OnAnimatorMove()` is used to override that root motion by setting the velocity of the NavMeshAgent to that root motion. The NavMeshAgent will be moving the character based on a rate determined by the Animator. Add the following line to the `OnAnimatorMove()` function:

```
agent.velocity = animator.deltaPosition / Time.deltaTime;
```

- All of the slowing, moving, and stopping speeds in the `OnAnimatorMove()` class still need to be handled, so create a new `private` `Update()` function for that purpose:

```
private void Update()
{
    // Distance based velocity calculations
}
```

Within `Update()`, a number of function calls for stopping, slowing, and moving the player will be created. After writing each of those three functions, return to `Update()` to finish writing the function. Since the `Update()` function will control the speed, both the stopping and slowing functions require an `out float` speed parameter to pass the player's velocity back to the `Update()` function for speed control. The moving function does not require a return parameter.

- First, check if the player is stopped by creating a `Stopping()` function with an `out float` speed parameter to pass the velocity back to the `Update()` function.

```
private void Stopping(out float speed)
{
    // Stopping stuff...
}
```

- Second, create a `Slowing()` function, also with an `out float` speed parameter for passing the velocity back to the `Update()` function along with another parameter to tell the `Slowing()` function how far it is to the destination.

```
private void Slowing(out float speed, float distanceToDestination)
```

```
{
    // Slowing stuff...
}
```

- Third, we need a `Moving()` function without any return parameters to deal with the general case.

```
private void Moving()
{
    // Moving stuff...
}
```

Within each of the three functions, the player movement is actually being controlled by `OnAnimatorMove()`. So within each of the three functions, only the player's rotation gets set. In the normal case, the player is moving. When the player approaches the destination, the player is slowing. And when the player actually arrives at the destination, the player is stopping.

- Returning to the `Update()` function, make sure nothing is happening when the player is planning on moving somewhere. To ensure that nothing is done while a path is pending, add an `if` statement that calls `return` to exit the function when `pathPending` is true.

```
if (agent.pathPending)
{
    return; // Do not do anything!
}
```

Given that the `NavMeshAgent` is not currently calculating a path, there is probably a path already, and working out how fast to move along that path based on the desired velocity of the `NavMeshAgent` is needed next. The `NavMeshAgent` has two velocities: The velocity it wants to go at and the velocity it's currently going at.

- After the `pathPending` check, set the velocity it actually moves at based on how fast it wants to move by creating a `float` variable called `speed` that is set equal to the `agent.desiredVelocity.magnitude` value.

```
float speed = agent.desiredVelocity.magnitude;
```

Next, determine which movement function to call. Is the player stopping, slowing, or moving? The first priority is to call `Stopping()` if the player is very, very close to the destination. The player needs to be at least within the stopping distance, but we want an even smaller radius by creating a number which makes a proportion of that stopping distance.

- Return to the variables section at the top of the script and create a `private const float` variable called `stopDistanceProportion` with a default value of `0.1f` to tell Unity to stop and set the position when within 10% of the stopping distance, which works out to be 1.5 cm in the game.

```
private const float stopDistanceProportion = 0.1f;
```

- Return to the `Update()` function and use the newly created variable to calculate if the player is within the inner stopping radius.

```
if (agent.remainingDistance <= agent.stoppingDistance * stopDistanceProportion)
{
    Stopping(out speed);
}
```

- Then, check to see if the character is within the outer stopping radius.

```
else if (agent.remainingDistance <= agent.stoppingDistance)
{
    Slowing(out speed, agent.remainingDistance);
}
```

- Given that the player is not really close to the destination or within the stopping distance of the destination, then the player is moving. But the player should only be turning if it's going fast enough. If the player stood still, but not near the destination, the player should not spin around really fast. The player needs to be moving before beginning to turn. In order to do that, create a check to verify how fast the player is currently going, and compare the result to a newly created default turn speed threshold variable before calling the `Moving()` function.

```
public float turnSpeedThreshold = 0.5f; // In variable section at top...

else if (speed > turnSpeedThreshold)
{
    Moving();
}
```

- Lastly, tell the Animator how fast the player is going by calling `Animator.SetFloat()`. Before doing that, determine what value to set. Normally when setting a parameter in the Animator, a `string` is used which needs to match the string 'Speed' that was set earlier in the Animator pane section. An integer can also be used which is faster and less error prone. To do that, find out what that integer is by creating a variable at the top of the script using a `private readonly int` variable with the word "hash" in the name which refers to a `string` representing an integer, and set the value based on the `static` Animator function `StringToHash()`.

```
private readonly int hashSpeedParam = Animator.StringToHash("Speed");
```

- To calculate how fast the speed parameter is going to be set, create a variable at the top of the script for damping the speed over time and set it to a default value. This default value is the amount of time over which the speed will change until reaching the newly set value. The Animator has a built-in system called damping parameters that gradually moves to the new speed rather than snapping to a new speed without any transition.

```
public float speedDampTime = 0.1f;
```

- Now, set the Animator speed at the bottom of the `Update()` function using the `animator.SetFloat()` represented by four parameters:
 - The hash variable just created.

- The new value the speed is being set to.
- The damping speed over time to be used for the transition.
- The time step used for damping.

```
animator.SetFloat(hashSpeedParam, speed, speedDampTime, Time.deltaTime);
```

- In the `Stopping()` function, exercise fine control over the player's position by preventing the `NavMeshAgent` from moving the character by setting `isStopped` to `true`.

```
agent.isStopped = true;
```

Note: `NavMeshAgent.Stop()` is obsolete: 'Set `isStopped` to `true` instead'.

- Next in the `Stopping()` function, snap the player's position to the destination being aimed for by setting the current position to the destination position.

```
transform.position = destinationPosition;
```

- And lastly in the `Stopping()` function, prevent the player from moving by setting the player's speed to 0.

```
speed = 0f;
```

- In the `Slowing()` function, exercise fine control over the player's position by preventing the `NavMeshAgent` from do anything by setting `isStopped` to `true`.

```
agent.isStopped = true;
```

Note: `NavMeshAgent.Stop()` is obsolete: 'Set `isStopped` to `true` instead'.

- Next in the `Slowing()` function, use the `Vector3.MoveTowards()` function to work out the player's position, and then gradually move the player's position towards the destination and base the speed on that distance of separation, which also requires an additional `float` variable at the top of the script for the maximum distance delta with a default value.

```
public float slowingSpeed = 0.175f; // In variable section at top..
```

```
transform.position = Vector3.MoveTowards(transform.position, destinationPosition,
    slowingSpeed * Time.deltaTime);
```

- Before calculating the player's speed, find out how close to the destination the player is as compared to the stopping distance (e.g., the proportional distance). When distance to the destination is very small approaching zero, the proportional distance will work out to a value of one.

```
float proportionalDistance = 1f - distanceToDestination / agent.stoppingDistance;
```

- The next line in `Slowing()` function will calculate the player's speed using `Mathf.Lerp` and interpolating between the slowing speed and zero based on the proportional distance. When

interpolating, a value of zero will give return the first value and a value of one will return the second value. The values in-between the first and second values will be an interpolation of the two values. When the proportional distance is one, the value will be zero. So when the distance to the destination is very small, the speed will get set to a very small value. When the distance to destination is close to the agent's stopping distance, then it will set the speed close to the slowing speed.

```
speed = Mathf.Lerp(slowingSpeed, 0f, proportionalDistance);
```

- As previously stated, the moving under normal circumstances is dealt with in the `OnAnimatorMove()` function. In the `Moving()` function, the player is rotated in the direction that the `NavMeshAgent` wants to move in by creating a `Quaternion` target rotation and setting it to the agent's desired velocity through the `Quaternion.LookRotation()` function.

```
Quaternion targetRotation = Quaternion.LookRotation(agent.desiredVelocity);
```

- After finding the target rotation, set the `transform.rotation` in the `Moving()` function equal to the `Quaternion.Lerp` between the player's current rotation and the player's target rotation. To figure out how fast to turn the player, create another `public float` variable, `turnSmoothing`, at the top of the script, set it to a default value, and then multiply that value by `Time.deltaTime` to calculate the third parameter of the `Quaternion.Lerp` function. Remember that the higher the value, the faster the turning, but the less smoothly; the lower the value, the slower the turning, but more smoothly.

```
public float turnSmoothing = 15f; // In variable section at top..
```

```
transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation,  
    turnSmoothing * Time.deltaTime);
```

- Now, set the player's destination by creating a function, `OnGroundClick()`, that is set in the Event Trigger and gets called when the ground is clicked on by the mouse. `OnGroundClick()` receives information about what is happening through a `BaseEventData` parameter called `data`. But, `BaseEventData` is not what is specifically wanted here. Instead, mouse `PointerClick` information and what is happening with the click at the current moment is needed. To get this information, cast the `BaseEventData` to `PointerEventData` by creating a new `PointerEventData` variable called `pData`. Then, cast the new variable by using the cast type, `PointerEventData`, in parentheses before the thing being cast outside of the parentheses, in this case the `BaseEventData` variable, `data`, which is passed into the `OnGroundClick()` function.

```
public void OnGroundClick(BaseEventData data)  
{  
    PointerEventData pData = (PointerEventData)data;  
}
```

- Now, use the new `pData` variable by finding a point on the `NavMesh` closest to the click using a function called `NavMesh.SamplePosition()` which works similarly to a `Raycast` in that it requires a 'hit' variable containing information about what was hit. Create the variable which will get set by the `NavMesh.SamplePosition()` call.


```
NavMeshHit hit;
```

- Just like a Raycast, a position is needed over which the NavMesh sample can happen. Create a new constant variable at the top of the script with a default value is 4f to refer to the distance away from the mouse click that the NavMesh can be sampled.

```
private const float navMeshSampleDistance = 4f;
```

- The NavMesh.SamplePosition() works like a Raycast and returns a bool for whether or not it is hit like a Raycast. Back in the OnGroundClick() function, call SamplePosition() within an if statement using the following four parameters:
 - The first parameter, pData.pointerCurrentRaycast.worldPosition, is the Vector3 position of the point in the world that the Raycast happening at the moment the click hits. An event is returned when a collider is hit and the point on that collider where the Raycast hit is returned.
 - The second parameter, out hit, returns all the information to the hit variable about what was hit.
 - The third parameter, navMeshSampleDistance, is the distance away from the click that is sampled over for potential hits.
 - The fourth parameter, NavMesh.AllAreas, refers to the areas we want to be checked, in this case, all the NavMesh available areas.

```
if (NavMesh.SamplePosition(pData.pointerCurrentRaycast.worldPosition, out hit,
                           navMeshSampleDistance, NavMesh.AllAreas))
{
    // Do stuff...
}
```

- So, what happens if something gets hit? The destination position gets set to hit.position, the position of the NavMesh that managed to get hit.

```
if (NavMesh.SamplePosition(pData.pointerCurrentRaycast.worldPosition, out hit,
                           navMeshSampleDistance, NavMesh.AllAreas))
{
    destinationPosition = hit.position;
}
```

- And if nothing gets hit? Find a location near by and have the player try to move towards the cursor wherever it has managed to click (e.g., the player will try to find a way to wherever the user clicked).

```
else
{
    destinationPosition = pData.pointerCurrentRaycast.worldPosition;
}
```

- After finding the appropriate position, tell the NavMeshAgent to use that location.

```
agent.SetDestination(destinationPosition);
```

- After telling the NavMeshAgent to stop, tell the agent to start moving again by setting NavMeshAgent.isStopped to `false`.

```
agent.isStopped = false;
```

Note: NavMeshAgent.Resume() is obsolete: 'Set isStopped to false instead'.

- Save the **PlayerMovement.cs** script.

Return to Unity and check the Console for errors.

- There will be a warning in the console regarding the variable made to specify a delay for dealing with what happens when things are interacted. The variable has not yet been used, and Unity throws a warning: *Assets/Scripts/MonoBehaviours/Player/PlayerMovement.cs(18,28): warning CS0414: The private field 'PlayerMovement.inputHoldWait' is assigned but its value is never used*

Now, check out the **PlayerMovement** script in the **Player** prefab.

- In the **Project** window, expand **Prefabs** and select **Player**.
- The **PlayerMovement** script now displays all the public fields.
- To populate the unassigned **Animator** and **Agent** fields:
 - Drag the **Player** prefab into the **Animator** slot.
 - Drag the **Player** prefab into the **Agent** slot.

Next, set up the **Event Trigger** that was left empty on the **SecurityRoom** GameObject.

- In the **Hierarchy**, select the **SecurityRoom** GameObject.
- The **Event Trigger** needs an object that runs the script. In this case, use the instance of the **Player** already in the scene hierarchy.
- With the **SecurityRoom** GameObject selected, drag the **Player** GameObject in the **Hierarchy** into the **Event Trigger Object** field.
- Referencing the **Player**, select the function dropdown list in the **Event Trigger**, select **PlayerMovement**, and choose **OnGroundClick**.

The **PointerClickEvent** is going to be looking at **MeshCollider** on the **SecurityRoom**. When it gets an **OnClick** event, it will find the **Player** to look at the **PlayerMovement** script and use the **OnGroundClick** appropriately.

- Save the scene.

The Player (Slide)

- Select the **Player** prefab
- On the **PlayerMovement** component:
 - Set up the reference to the Player's **Animator**
 - Set up the reference to the Player's **NavMeshAgent**
- Navigate to the **Hierarchy**
- Select the **SecurityRoom** game object
- Find the **Event Trigger**
- Drag the **Player** game object from the **Hierarchy** onto the **Object** field of the **Event Trigger**

- Select **PlayerMovement.OnGroundClick** in the function dropdown list of the **Event Trigger**
- Save the scene
- Test
 - Player can walk around but not interact with anything like the Security Gate or the Exit
- Exit Play mode

After exiting Play mode, return to the **PlayerMovement.cs** script to work on interaction.

First thing to do, is know what the interactable is that the player is heading towards.

- At the top of the script, create a new private variable called `CurrentInteractable` to store the interactable that the player is heading towards.

```
private Interactable currentInteractable;
```

- When arriving at the destination of an interactable, the player wants to interact with it. To deal with that, first modify the `Stopping()` function by determining if the player is heading towards an interactable by using an `if` statement at the bottom of the function to perform a check.

```
if (currentInteractable)
{
    // Do stuff...
}
```

When arriving at an interactable, make sure that the player is heading in the right direction. So, the player needs to be facing the direction that the interaction location sets.

- Inside the `if` statement's curly brackets in the `Stopping()` function, set the transform rotation to the rotation of the current interactable's location. Now that the player is facing the correct direction, call the `Interact()` function for the interactable. Afterwards, set the current interactable to `null` so the `Interact()` function is only called once.

```
if (currentInteractable)
{
    transform.rotation = currentInteractable.interactionLocation.rotation;
    currentInteractable.Interact();
    currentInteractable = null;
}
```

Now, wait for the interaction to happen before doing anything else by blocking the input.

- Create a new `private bool` variable at the top of the script to make sure that input is not accepted when interacting. By default, input will need to be handled so give the new variable a default value of `true`.

```
private bool handleInput = true;
```

Another thing that controls whether or not the player can move is if the player is in an Animator state that is tagged `Locomotion`.

- To check if the current player state is a Locomotion state, use the same hash method used for the Speed parameter check and create another `private readonly` integer variable for the Locomotion tag, again beginning the variable name with the word "hash".

```
private readonly int hashLocomotionTag = Animator.StringToHash("Locomotion");
```

- Now, create a new coroutine at the bottom of the script to determine how long to wait for an interaction to happen.

```
private IEnumerator WaitForInteraction()
{
    // Do stuff...
}
```

- No input will be handled here, so set the `handleInput` variable to `false`.

```
private IEnumerator WaitForInteraction()
{
    handleInput = false;

    // Do stuff...
}
```

- At the bottom of the coroutine after the interaction wait is over, set the `handleInput` variable back to `true`.

```
private IEnumerator WaitForInteraction()
{
    handleInput = false;

    // Do stuff...

    handleInput = true;
}
```

- In the middle of the coroutine, insert a `yield` statement to exit the code at this point to wait for the thing to the right of the `yield` statement, the `inputHoldWait` in `WaitForSeconds` that was cached during the `Start()` function, to be `true` before returning to the coroutine.

```
private IEnumerator WaitForInteraction()
{
    handleInput = false;

    yield return inputHoldWait;

    // Do more stuff...

    handleInput = true;
}
```

The other thing to wait for is the player to be in a Locomotion state.

- After the `yield` statement, use a while loop to check if the Animator base layer's current state tag is not equal to `hashLocomotionTag`. If the current state does not have the Locomotion tag, wait a single frame using a `yield return null` statement. So, every frame that is not in a Locomotion state enters the while loop.

```
private IEnumerator WaitForInteraction()
{
    handleInput = false;

    yield return inputHoldWait;

    while (animator.GetCurrentAnimatorStateInfo(0).tagHash != hashLocomotionTag)
    {
        yield return null;
    }

    handleInput = true;
}
```

- Returning to the if statement at the bottom of the `Stopping()` function, add a call to the new coroutine to deal with what happens when an `Interactable` is clicked on.

```
if (currentInteractable)
{
    transform.rotation = currentInteractable.interactionLocation.rotation;
    currentInteractable.Interact();
    currentInteractable = null;
    StartCoroutine(WaitForInteraction());
}
```

To deal with slowing down while approaching an Interactable since the player should not "snap" around suddenly to the new rotation of the Interactable rotation, add code in the `Slowing()` function to slowly interpolate the player's rotation towards the rotation of the Interactable so the player movement looks more natural.

- At the bottom of the `Slowing()` function, add a `Quaternion` variable using a conditional operator (e.g., `condition ? first_expression : second_expression;`) that checks if the player is heading towards an `Interactable`, then target the `Interactable` location rotation. If the player is not heading towards an `Interactable` location, then maintain the current rotation.

```
Quaternion targetRotation = currentInteractable ?
    currentInteractable.interactionLocation.rotation :
    transform.rotation;
```

- Now, use the target rotation to interpolate the player's rotation towards the new `Interactable` rotation. Like the speed, interpolate based upon the proportional distance that was worked out. The new rotation will equal a `Quaternion.Lerp` which interpolates between the current rotation and the target rotation by an amount based on the proportional distance.

```
transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation,
    proportionalDistance);
```

- Next, create a function under the `OnGroundClick()` function to allow the player to actually click on Interactables. The function takes an `Interactable` parameter so it can figure out which Interactable the player clicked on.

```
public void OnInteractableClick(Interactable interactable)
{
    // Do stuff...
}
```

- First thing to do in the new function is to make sure input can be handled by checking the `handleInput` variable. If the player is already interacting with something, don't do anything.

```
if (!handleInput)
{
    return; // Don't do anything
}
```

- If input can be handled, store the Interactable the player is heading towards by setting the `currentInteractable` variable to the `interactable` parameter being passed into the function.

```
currentInteractable = interactable;
```

- Next, set the destination the player is heading towards by setting the destination position to the current Interactable's position.

```
destinationPosition = currentInteractable.interactionLocation.position;
```

- With the destination position set, repeat the steps from the `OnGroundClick()` function to make sure the `NavMeshAgent` is heading towards the destination position with the `SetDestination()` function.

```
agent.SetDestination(destinationPosition);
```

- And, tell the agent to start moving again by setting `NavMeshAgent.isStopped` to `false`.

```
agent.isStopped = false;
```

Note: `NavMeshAgent.Resume()` is obsolete: 'Set `isStopped` to `false` instead'.

- At the top of the `OnGroundClick()` function, add a check to see if any input can be handled, and if not, return out of the function. This will handle the player not being able to move when the ground is clicked on but the player is already interacting with something.

```
if (!handleInput)
{
    return; // Don't do anything
}
```

- Also, if the player has clicked on an **Interactable** and then immediately clicked on the ground, do not accidentally activate the **Interactable**. Set the current **Interactable** to **null** to handle this case.

```
currentInteractable = null;
```

- Save the **PlayerMovement.cs** script.
 - Return to the Unity editor to check for errors in the **Console**.
-

The Player (Slides)

- **Interactables**, along with **Conditions** and **Reactions**, have been supplied to our team
 - We simply need to set them up to work with our new **Player click to move** system
 - Note: *We will be taking on the role of developing these systems later in the session*

 - In the Hierarchy, select **PictureInteractable**
 - Find the **Event Trigger (Script)** component
 - Drag the **Player** GameObject from the **Hierarchy** onto the **None (Object)** field of the **Event Trigger**
 - From the function dropdown list, select **PlayerMovement.OnInteractableClick(Interactable)**
 - Drag the **Interactable (Script)** GameObject (e.g., the **Interactable** component below the **EventTrigger Script** component) onto the **Parameter** field under the function dropdown list of the **Event Trigger (Script)** component
-
- Save the scene!!
 - Start the game to test the player's interaction with the picture of the face on the wall of the scene.
 - The interaction will play a sound clip and display text on the screen after the player clicks on the picture. Both the sound clip and the display text are *reactions* to the player's interaction with the picture.



PlayerMovement Script

```
using System.Collections;
using UnityEngine;
using UnityEngine.AI;
using UnityEngine.EventSystems;

public class PlayerMovement : MonoBehaviour
{
    public Animator animator;
    public NavMeshAgent agent;
    public float inputHoldDelay = 0.5f;
    public float turnSpeedThreshold = 0.5f;
    public float speedDampTime = 0.1f;
    public float slowingSpeed = 0.175f;
    public float turnSmoothing = 15f;

    private WaitForSeconds inputHoldWait;
    private Vector3 destinationPosition;
    private Interactable currentInteractable;
    private bool handleInput = true;

    private const float stopDistanceProportion = 0.1f;
    private const float navMeshSampleDistance = 4f;

    private readonly int hashSpeedParam = Animator.StringToHash("Speed");
    private readonly int hashLocomotionTag = Animator.StringToHash("Locomotion");

    private void Start()
    {

```



```

        agent.updateRotation = false;

        inputHoldWait = new WaitForSeconds(inputHoldDelay);

        destinationPosition = transform.position;
    }

    private void OnAnimatorMove()
    {
        agent.velocity = animator.deltaPosition / Time.deltaTime;
    }

    private void Update()
    {
        if (agent.pathPending)
        {
            return;
        }

        float speed = agent.desiredVelocity.magnitude;

        if (agent.remainingDistance <= agent.stoppingDistance * stopDistanceProportion)
        {
            Stopping(out speed);
        }
        else if (agent.remainingDistance <= agent.stoppingDistance)
        {
            Slowing(out speed, agent.remainingDistance);
        }
        else if (speed > turnSpeedThreshold)
        {
            Moving();
        }

        animator.SetFloat(hashSpeedParam, speed, speedDampTime, Time.deltaTime);
    }

    private void Stopping(out float speed)
    {
        agent.isStopped = true;
        transform.position = destinationPosition;
        speed = 0f;

        if (currentInteractable)
        {
            transform.rotation = currentInteractable.interactionLocation.rotation;
            currentInteractable.Interact();
            currentInteractable = null;
            StartCoroutine(WaitForInteraction());
        }
    }

    private void Slowing(out float speed, float distanceToDestination)
    {
        agent.isStopped = true;
        transform.position = Vector3.MoveTowards(transform.position, destinationPosition,
                                                slowingSpeed * Time.deltaTime);
    }

```

```

float proportionalDistance = 1f - distanceToDestination / agent.stoppingDistance;
speed = Mathf.Lerp(slowingSpeed, 0f, proportionalDistance);

Quaternion targetRotation = currentInteractable ?
                            currentInteractable.interactionLocation.rotation :
                            transform.rotation;

transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation,
                                     proportionalDistance);
}

private void Moving()
{
    Quaternion targetRotation = Quaternion.LookRotation(agent.desiredVelocity);
    transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation,
                                         turnSmoothing * Time.deltaTime);
}

public void OnGroundClick(BaseEventData data)
{
    if (!handleInput)
    {
        return;
    }

    currentInteractable = null;

    PointerEventData pData = (PointerEventData)data;
    NavMeshHit hit;

    if (NavMesh.SamplePosition(pData.pointerCurrentRaycast.worldPosition, out hit,
                              navMeshSampleDistance, NavMesh.AllAreas))
    {
        destinationPosition = hit.position;
    }
    else
    {
        destinationPosition = pData.pointerCurrentRaycast.worldPosition;
    }

    agent.SetDestination(destinationPosition);
    agent.isStopped = false;
}

public void OnInteractableClick(Interactable interactable)
{
    if (!handleInput)
    {
        return;
    }

    currentInteractable = interactable;

    destinationPosition = currentInteractable.interactionLocation.position;

    agent.SetDestination(destinationPosition);
    agent.isStopped = false;
}

```

```

private IEnumerator WaitForInteraction()
{
    handleInput = false;

    yield return inputHoldWait;

    while (animator.GetCurrentAnimatorStateInfo(0).tagHash != hashLocomotionTag)
    {
        yield return null;
    }

    handleInput = true;
}
}

```

Inventory

Phase 02 – Inventory (Slides)

Inventory (Slides)

Brief

- Create a **simple inventory system** with **persistent** content that is not lost during scene changes
 - Inventory stays the same between scene 1 and scene 2, and vice versa
- **Inventory Items** should be simple but easily extensible if the design changes
 - Sell, drop, upgrade, smelt, make MMO, equip, etc.
- The inventory should have two public functions:
 - AddItem
 - RemoveItem


(The problem with the complexity of an inventory is not really with the inventory system, but with the way it is displayed with the UI.)

- Simplify and improve the workflow of the project in the **Inspector** with regards to the **Inventory** and its **Items** through the use of custom inspectors

The Approach

- Use the **UI system** to display the inventory to the user
- Use **ScriptableObjects** to make a simple **Item** class which defines every possible inventory item and can easily be extended and referenced by the **Inventory**
- Create a **custom inspector** for the **Inventory** to improve the workflow of the project

To create a project for Phase 02 of the Adventure Game Tutorial, do the following:

- Start Unity 2017.2.0f3
- With the **3D** option selected, click the **New** link to create a new project named **Unity_Tutorial_AdventureGame_Phase02** (or some other meaningful name) and then click the blue **Create project** button
- Click on the **Asset Store** tab, type **Adventure Game Tutorial** in the search field at the top of the **Asset Store** pane, and click the Search button  (or press **Enter**)
- Scroll down and choose **2/6 – Adventure Tutorial – Inventory** from the results list
- Click the blue **Download** button
- Click the **Import** button on the **Importing Complete Project** popup message
- Click the **Import** button on the **Import Unity Package** popup message

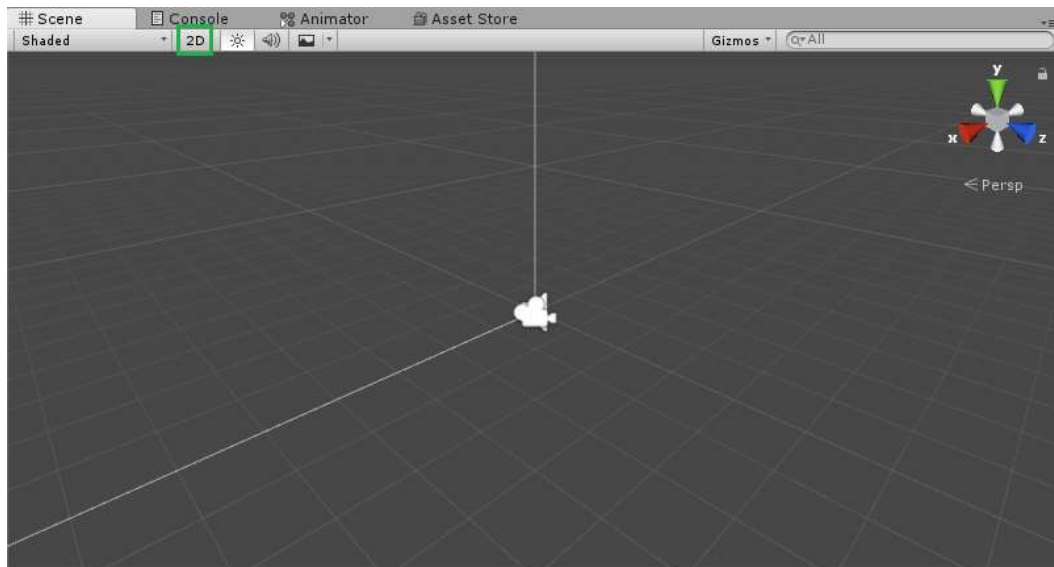
Initial Build Steps

Make sure to close MonoDevelop and Visual Studio so as to not get the two projects mixed up. To set up the Phase 02 project, do the following:

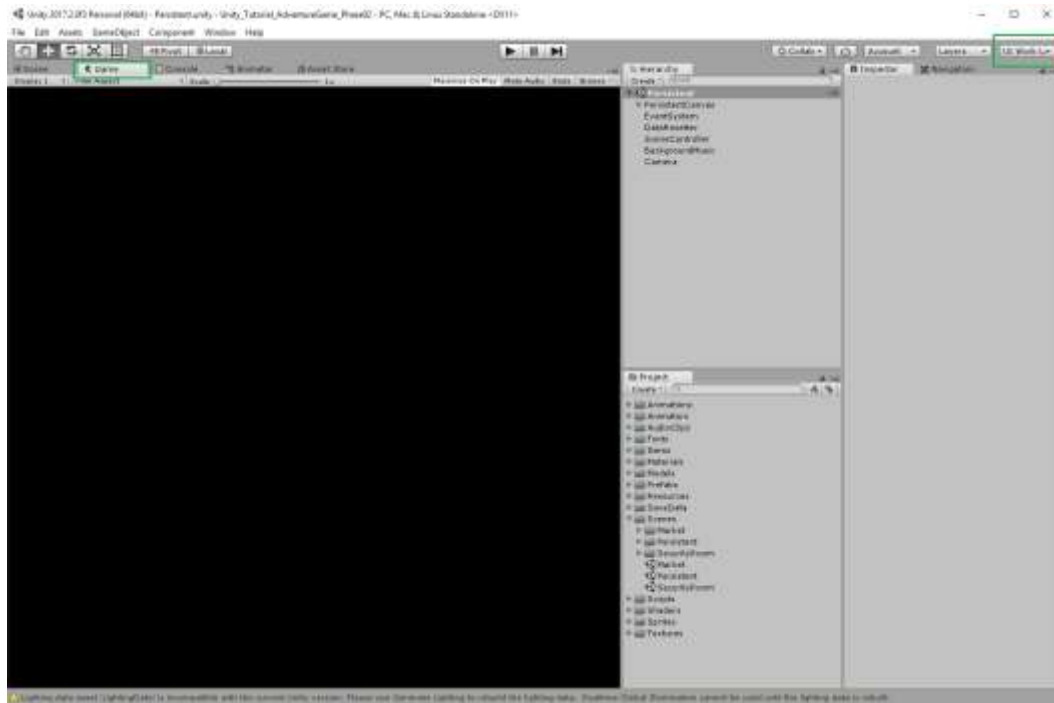
- Navigate to **Project** pane
- Expand the **Scenes** folder and double-click the **Persistent** scene to open it

The inventory system will need to be persistent and some of the basic information will be saved in the **Persistent** scene.

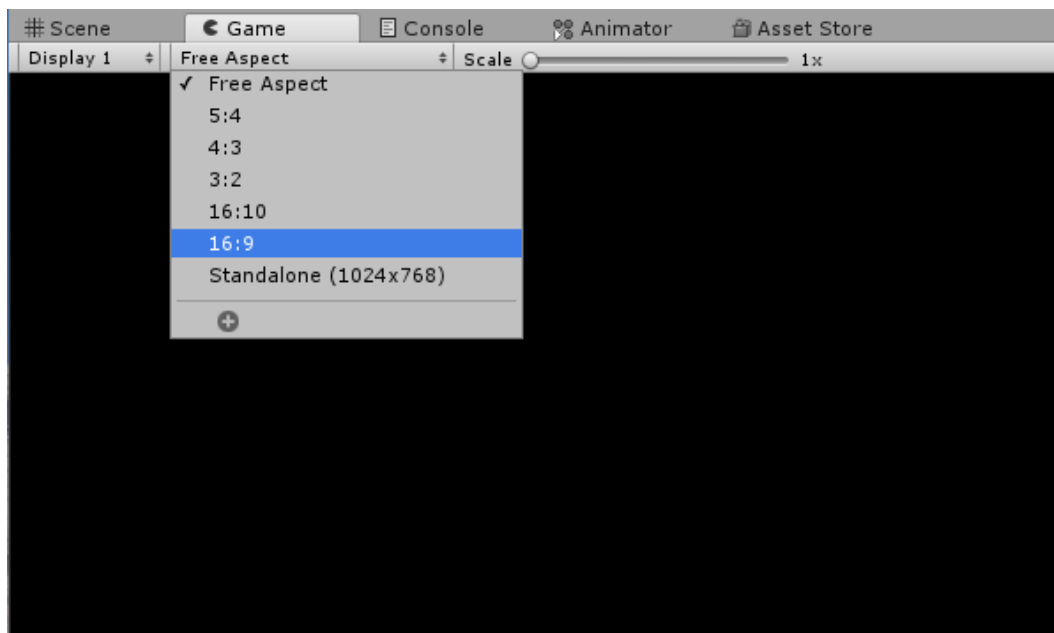
- In the **Scene** pane, click the **2D** button to switch modes for working with the UI



- To make it easier to work with the UI, grab the **Game** tab for the **Game** panel on the lower left of the Unity editor and drag it next to the **Scene** tab (green box on left in screenshot below). For ease of use, save this layout for later UI work by selecting the **Layout** dropdown list in the upper, right corner (green box on right in screenshot below), selecting **Save Layout...**, and providing a meaningful name like **UI Work Layout**.



- With the Game mode tab selected, change the aspect ratio by clicking on **Free Aspect** and then selecting the **16:9** from the dropdown list.



- In the Hierarchy pane, expand the **Persistent** scene and select the **PersistentCanvas**
- Switch from the **Game** mode to **Scene** mode and center the **PersistentCanvas** in the **Scene** pane

Inventory (Slide)

1. Navigate to the **Scenes** folder
 2. Open the **Persistent** scene
 3. Set the **Scene** view to **2D** mode
 4. Navigate to the **Game** view
 5. Set the **Aspect Ratio** to **16:9**
 6. Select and frame the **PersistentCanvas**
-

Understanding the UI in the Hierarchy Window (Slide)

1. The order of UI Elements in the Hierarchy window informs the UI system what order to render the UI Elements
2. The rendering order is from *the top to the bottom* which will render on screen from *the back to the front*

Hierarchy

UI Canvas

UI Object 01 – Background object rendered first

UI Object 02 – Drawn on top of the background object (e.g., mid ground), rendered second

UI Object 03 – Drawn on top of the mid ground object (e.g., foreground), rendered third

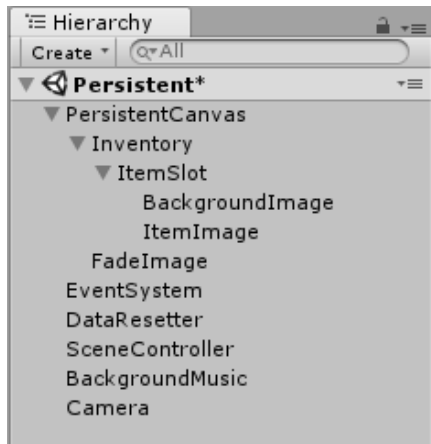
The order of the UI objects in the Hierarchy determines the order in which the objects are rendered in the UI. GameObjects at the **top** of the hierarchy are drawn **first**.


Inventory (Slide)

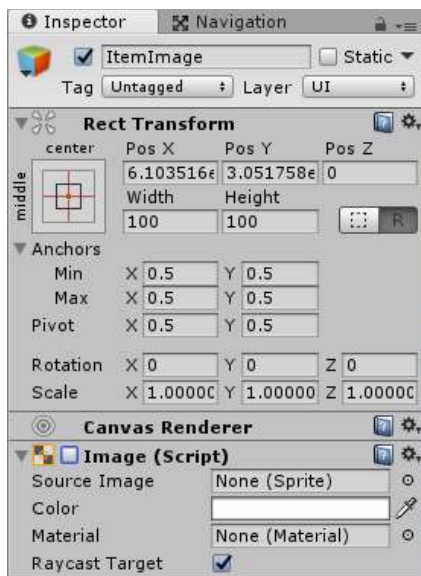
1. Navigate back to the **Scene** view
 2. With the **PersistentCanvas** selected, **Create an empty child GameObject**
 3. Name this new GameObject **Inventory**
 4. Make sure that it is the **first child** of **PersistentCanvas** and that it is *above* **FadelImage**
-

While in **Scene** mode, return to the **Hierarchy** pane:

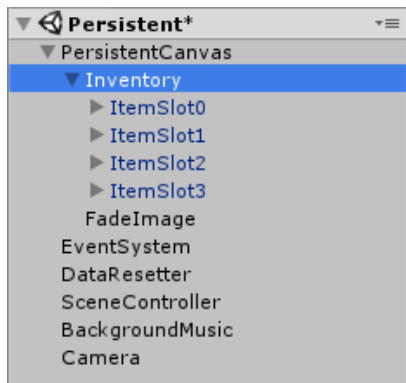
- Select the **PersistentCanvas** GameObject
- Right-click **PersistentCanvas** and select **Create Empty**
- Select the newly created **GameObject** and drag it above the existing **FadelImage** GameObject so it is at the top of the list under the **PersistentCanvas** where it will be rendered in the back since **FadelImage** is simply a black frame that is stretched across the entire canvas which will be later on in scene management to fade to black, do the scene change, and then fade back into the next scene.
- Select the new **GameObject** and rename it to **Inventory**
- With **Inventory** selected, right-click and select **Create Empty** again
- Rename the new **GameObject** to **ItemSlot**
- Right-click the **ItemSlot** GameObject, select **UI**, and then choose **Image**
- With the **Image** GameObject selected, press **CTRL-D** to duplicate the image
- Select the top **Image** and rename it to **BackgroundImage**
- Select the duplicate **Image (1)** and rename it to **ItemImage**



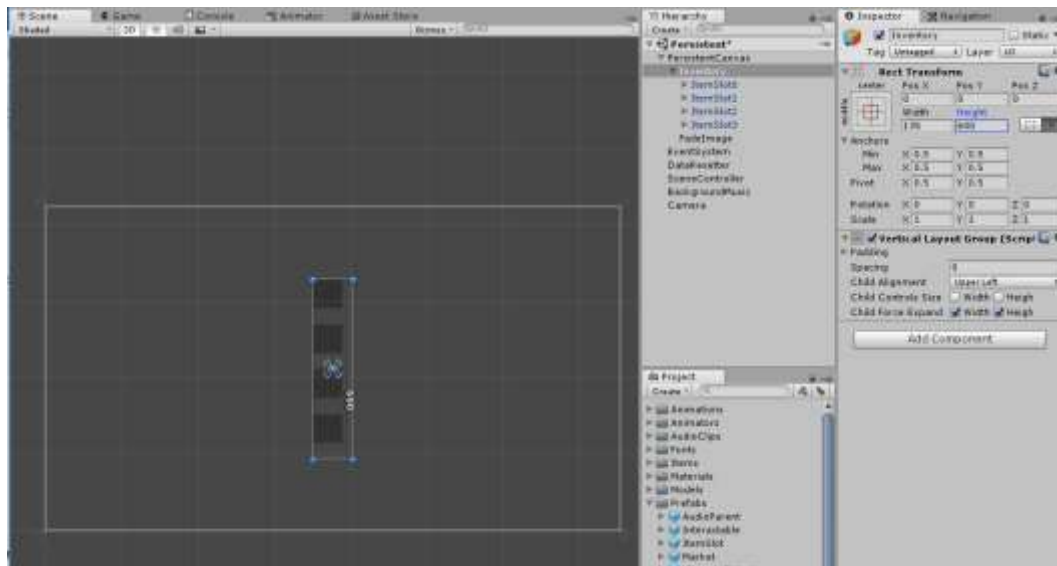
- With **BackgroundImage** selected, set the **Source Image** field in the **Image (Script)** component using the Circle Select button  and choosing **InventorySlotBG**
- With **ItemImage** selected, disable the **Image (Script)** component by unchecking the box on the component so the image background is displayed if there is no corresponding item image



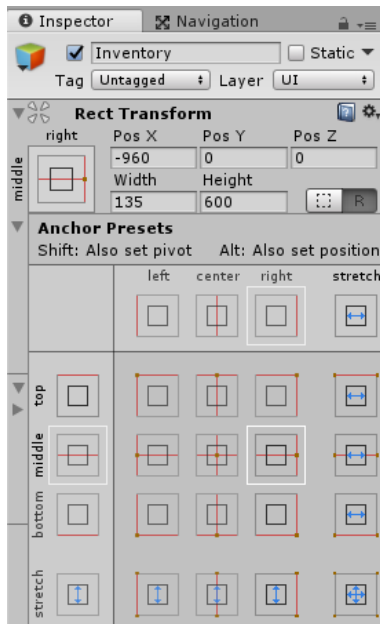
- Make the **ItemSlot** a prefab GameObject by dragging it from the Hierarchy pane onto the **Prefabs** folder in the Project pane as this is a good habit to build
- With the **ItemSlot** GameObject selected in the Hierarchy, press **CTRL-D** three (3) times to create three duplicate GameObjects for a total of four (4) **ItemSlot** GameObjects
- Rename the four GameObjects to **ItemSlot0**, **ItemSlot1**, **ItemSlot2**, and **ItemSlot3**



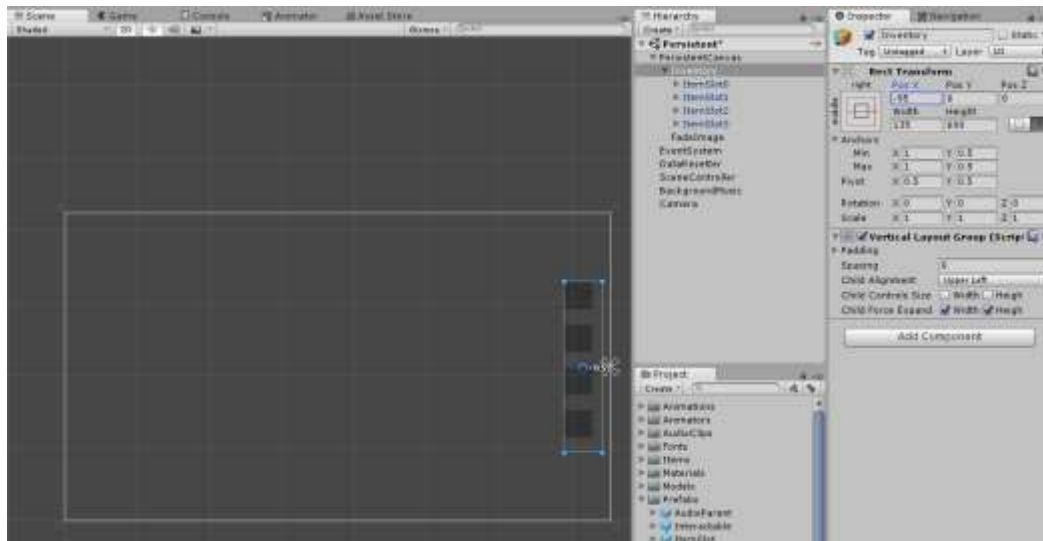
- Select the **Inventory** GameObject in the Hierarchy, click the **Add Component** button in the Inspector pane, select **Layout**, and choose **Vertical Layout Group**
- Change the **Inventory** GameObject's **Rect Transform** values to a **Width** of **135** and a **Height** of **600** to spread the inventory out more in a vertical bar



- From the **Anchor** dropdown in the Inspector pane, select **middle-right** to anchor the inventory to the middle of the right-hand side of the canvas



- In the Inspector, change the **Pos X** of the **Rect Transform** to **-95**



- Save the scene

Now that there is an Inventory, it is time to write a script for the Inventory. The first script to look at is *extremely complicated*, so it is already written, and it is the **Item** itself. The **Item** script is located in the **Scripts\ScriptableObjects\Inventory** folder.

- First off, it is a **Scriptable** object which means that it can be created as an asset, meaning there will be **Item** assets that sit in the **Project** folder.

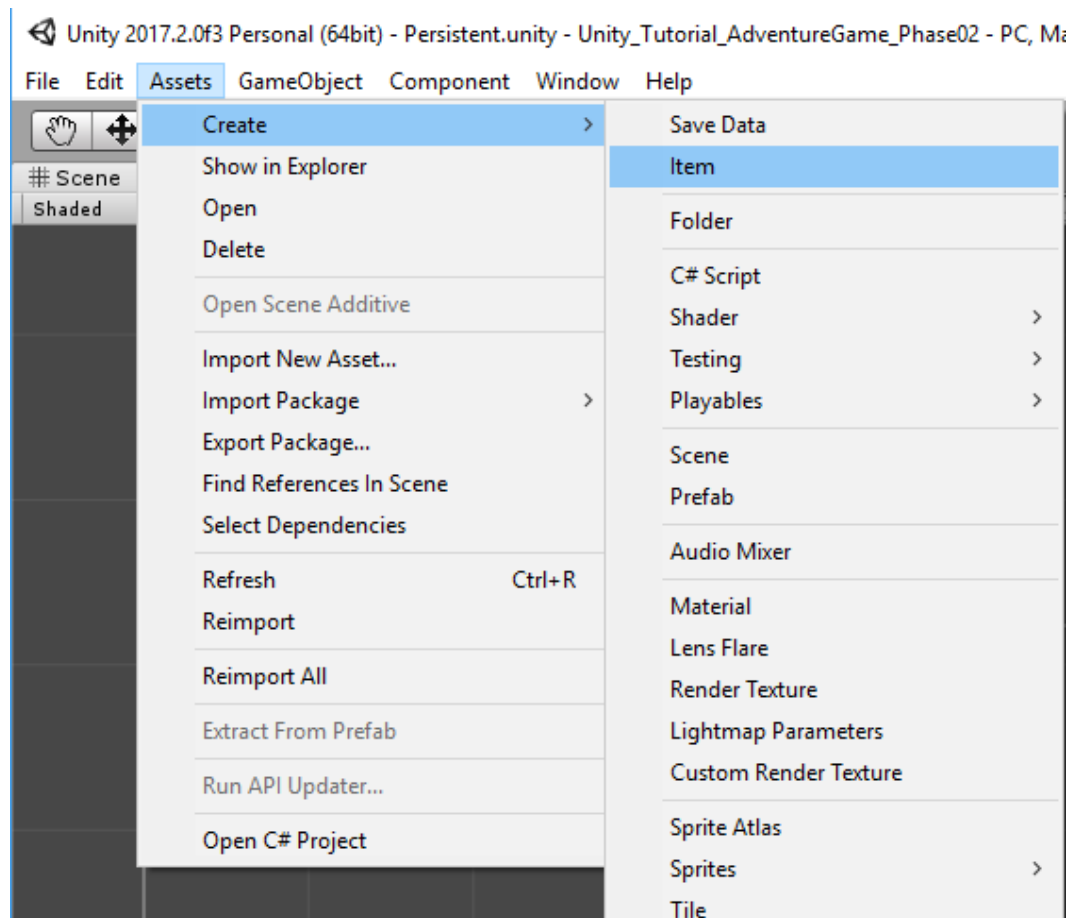
```
using UnityEngine;
```

```
[CreateAssetMenu]
```

```
public class Item : ScriptableObject
```

```
{
    public Sprite sprite;
}
```

- There's a **CreateAssetMenu** attribute so when assets are clicked on in the Unity editor, the **Assets** menu displays an option for creating **Items** based on the script



- The only variable in the script is the **Sprite** that represents the script which also makes the script very extensible for equipping items or representing items with GameObjects

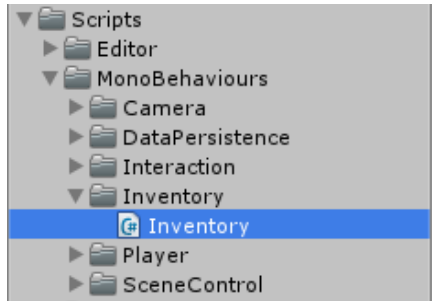
Inventory (Slide)

1. Select the **Scripts > MonoBehaviours > Inventory** folder
 2. Create a **C# script** called **Inventory**
 3. Open the **Inventory** script for editing
-

Create an **Inventory** script to control the player's inventory:

- In the Project pane, expand the **Scripts** folder and then expand the **MonoBehaviours** folder.
- With the **MonoBehaviours** folder selected, right-click and select **Create**, choose **Folder**, and name the folder **Inventory**.

- With the new **Inventory** folder selected, right-click the folder, select **Create**, choose **C# Script**, and name the script **Inventory**.



- Open the **Inventory** script for editing in MonoDevelop or Visual Studio.

The Inventory scripts needs to store items and display items. Things that will be needed:

- Access to image components to display the item images
- Access to an array of items for storage
- A function for adding items to the item array
- A function for removing items from the item array

There need to be as many items in the array as there are image components to display them.

With the **Inventory** script open for editing:

- Remove the existing **Start()** and **Update()** functions
- Add a **using** directive at the top of the script for working with the UI namespace to access the needed image components.

```
using UnityEngine.UI;
```

- Create a **public** constant variable for the number of Item slots and set it equal to 4 since that is how many Item slots are in the UI. **numItemSlots** is a **public** variable so the **InventoryEditor** that will be written later has access to the number of inventory slots.

```
public const int numItemSlots = 4;
```

- Create a **public** Image array that is initialized to the number of item slots.

```
public Image[] itemImages = new Image[numItemSlots];
```

- Create a **public** Item array that is also initialized to the number of item slots.

```
public Item[] items = new Item[numItemSlots];
```

- Create a **public** **AddItem()** function that can be called from anywhere and does not return anything. The function takes an Item parameter representing the Item being added to the Item array.

```
public void AddItem(Item itemToAdd)
```

```
{
    // Add stuff
}
```

- Create a for loop to iterate through all the Item slots until an empty slot is found, then set that empty slot to the Item being added, set the Item slot's image sprite to the sprite of the Item being added, and then enable the Item slot's image since the **Image (Script)** component was previously disabled on the **ItemImage** GameObjects when the component checkbox was cleared in the Unity editor. Once an empty slot has been found and the Item has been added, call **return** to exit the function.

```
for (int i = 0; i < items.Length; i++)
{
    if (items[i] == null)
    {
        items[i] = itemToAdd;
        itemImages[i].sprite = itemToAdd.sprite;
        itemImages[i].enabled = true;
        return;
    }
}
```

- Next, create a public RemoveItem function that can be called from anywhere and does not return anything. The function takes an Item parameter representing the Item being removed from the Item array.

```
public void RemoveItem(Item itemToRemove)
{
    // Remove stuff
}
```

- Create another for loop to iterate through all the Item slots until a slot is found containing the Item being removed, then set that Item slot to **null**, set the Item slot's image sprite to **null**, and then disable the Item slot's image by setting enabled back to **false**. Once the Item has been removed, call **return** to exit the function.

```
for (int i = 0; i < items.Length; i++)
{
    if (items[i] == itemToRemove)
    {
        items[i] = null;
        itemImages[i].sprite = null;
        itemImages[i].enabled = false;
        return;
    }
}
```

- Save the script and return to the Unity editor.

Inventory (Slide)

1. Navigate to **Scripts > ScriptableObjects > Interaction > Reactions > DelayedReactions** folder

2. Open the **LostItemReaction** script for editing
 3. Uncomment all the commented-out code (e.g., remove the `//` from lines 6, 11, and 17)
 4. Save the script and return to the editor
-

The **LostItemReaction** script is the reaction called when taking an item out of the player's inventory.

- After un-commenting lines 6, 11, and 17 in the **LostItemReaction** script, it should appear as follows:

```
public class LostItemReaction : DelayedReaction
{
    public Item item;

    private Inventory inventory;

    protected override void SpecificInit()
    {
        inventory = FindObjectOfType<Inventory> ();
    }

    protected override void ImmediateReaction()
    {
        inventory.RemoveItem (item);
    }
}
```

- Save the script and return to the Unity editor.

Next, do the same un-commenting process with the **PickedUpItemReaction** script in the same folder.

The **PickedUpItemReaction** script gets called when the player adds an item to their inventory.

- After un-commenting lines 6, 11, and 17 in the **PickedUpItemReaction** script, it should appear as follows:

```
public class PickedUpItemReaction : DelayedReaction
{
    public Item item;

    private Inventory inventory;

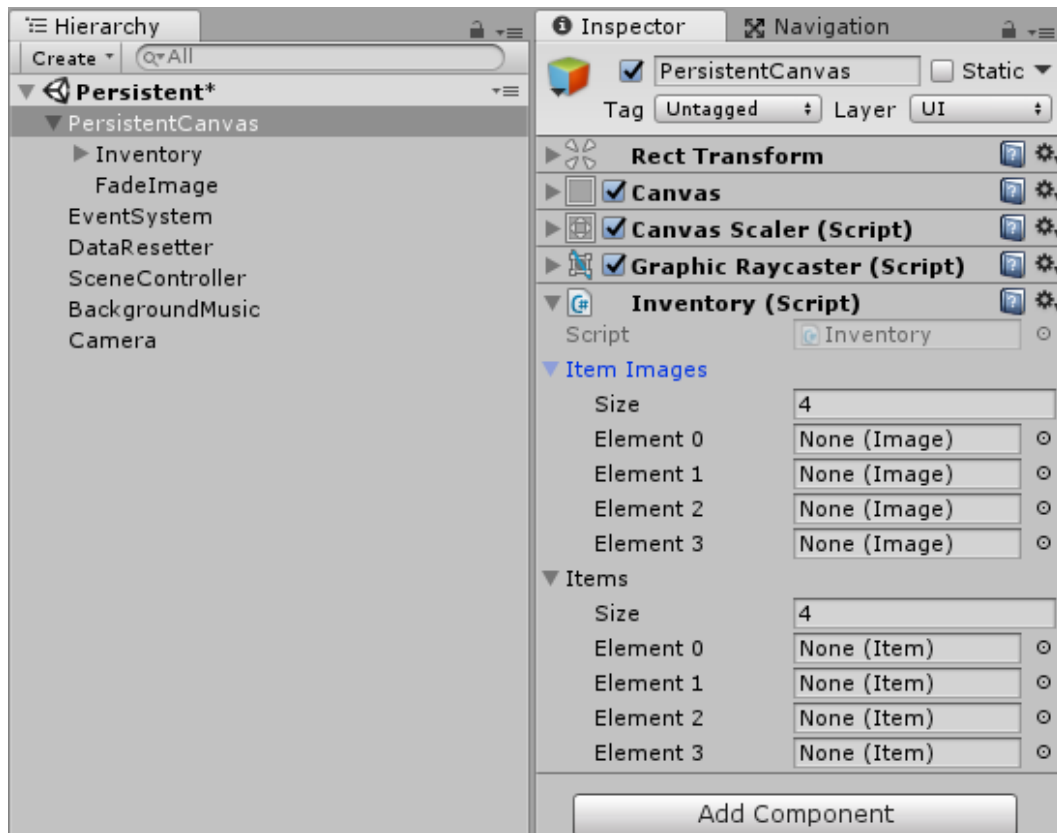
    protected override void SpecificInit()
    {
        inventory = FindObjectOfType<Inventory>();
    }

    protected override void ImmediateReaction()
    {
        inventory.AddItem(item);
    }
}
```

- Save the script and return to the Unity editor.

Now that the Inventory script has been created to manage the player's inventory, it needs to be used somehow.

- In the Project pane, select the **Inventory** script in the **Scripts\MonoBehaviours\Inventory** folder and drag it into the Hierarchy onto the **PersistentCanvas** GameObject. The **Inventory** needs to be in the **Persistent** scene because the inventory system needs to be persistent throughout the game. The standard Inspector view displays the two created arrays as shown below and the Item array and Image arrays are not yet associated with each other. Next, the two arrays will be displayed using a Custom Inspector.



Understanding the Custom Inspector (Slide)

At Run time, there are objects in the game. Within each object, there are a number of fields. The fields here are an Image array and an Item array.

At Edit time, an Editor is created that targets one of the objects that uses a SerializedObject to represent it. A SerializedObject is a generic representation of a run time object. Here, the SerializedObject will be looking at an Inventory. The SerializedObject has SerializedProperties that represent the fields of the run time object.

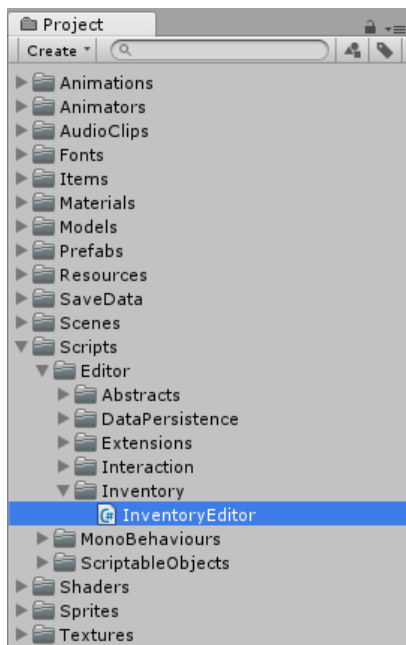
So, for the Inventory, there is a SerializedObject. For the Item array, there is a SerializedProperty. For the Image array, there is a SerializedProperty. All of those will be used to display things in the Editor.

Inventory (Slide)

1. Find the **Scripts > Editor > Inventory** folder
2. Create a **C# Script** called ***InventoryEditor***
3. Open the **InventoryEditor** script for editing

Create an **InventoryEditor** script to manage displaying the Inventory and Image arrays in the Inspector pane:

- In the Project pane, expand the **Scripts** folder and then expand the **Editor** folder.
- With the **Editor** folder selected, right-click and select **Create**, choose **Folder**, and name the folder ***Inventory***.
- With the new **Inventory** folder selected, right-click the folder, select **Create**, choose **C# Script**, and name the script ***InventoryEditor***.



- Open the **InventoryEditor** script for editing in MonoDevelop or Visual Studio,

Note: It is a common naming convention that when an editor is created for a particular class, one simply uses the name of the targeted class followed by the word "Editor". Here, the target is the Inventory class, so the editor is called the **InventoryEditor**.

With the **InventoryEditor** script open for editing:

- Remove the existing **Start()** and **Update()** functions
- Add a **using** directive at the top of the script for working with the UnityEditor namespace in order to have all the classes needed for making an editor.

```
using UnityEditor;
```

- Next, since this script will not be a MonoBehaviour that gets attached to a GameObject, replace the MonoBehaviour with Editor which is what is being created here.

```
public class InventoryEditor : Editor
```

- The Editor also needs to know what it will be looking at, so provide a target type using the CustomEditor attribute with a type of Inventory in parentheses. If the attribute is not there, the Editor will not work in the Inspector and will not target anything. The script should look like the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(Inventory))]
public class InventoryEditor : Editor
{
    // Do stuff..
}
```

- Create the serialized properties for each of the fields that need to be represented, in this case, the itemImages array and the items array:

```
private SerializedProperty itemImagesProperty;
private SerializedProperty itemsProperty;
```

- Provide names for the serialized properties to find by creating two constant strings. The constant strings are the preferable way to do this. In case any renaming needs to be performed in the future, the constant strings will make the task much easier and provide a single point of change in the code.

```
private const string inventoryPropItemImagesName = "itemImages";
private const string inventoryPropItemsName = "items";
```

The naming convention here breaks down into the first part referencing the class that the property will be part of (e.g., Inventory), then the shortened "Prop" for property, then the name of the field being targeted (e.g., ItemImages or Items), and finally the word "Name" to indicated that it's a `string` referring to the name of that property.

- Now, create a `private OnEnable()` function to find the serialized properties:

```
private void OnEnable()
{
    // Find serialized properties
}
```

- Find the itemImagesProperty using the FindProperty() function of SerializedObject, the representation of the the Inventory, with the constant inventoryPropItemImages which is set to "itemImages":


```
itemImagesProperty = serializedObject.FindProperty(inventoryPropItemImagesName);
```

- Next, find the itemsProperty using the same method:

```
itemsProperty = serializedObject.FindProperty(inventoryPropItemsName);
```

- At the top of the script, create a new `private bool` array to tell the Inspector whether or not an Item slot is displayed by the InventoryEditor:

```
private bool[] showItemSlots = new bool[Inventory.numItemSlots];
```

- Override the function `OnInspectorGUI ()` from the Editor class to change the way the Inspector displays the Inventory:

```
public override void OnInspectorGUI()
{
    // Do stuff...
}
```

- First make sure that the information in the serialized objects is up-to-date and matches the information actually in the Inventory by calling the `Update()` function on `serializedObject` at the beginning of `OnInspectorGUI()`:

```
serializedObject.Update();
```

Whenever a serialized property is changed, the representation within the serialized object is changed because the serialized properties do not belong to the Inventory, they belong to the serialized object. Whenever one of the serialized properties is changed, it changes the serialized object.

- To make sure the changes get back to the target, call the `ApplyModifiedProperties()` function on the serialized object at the end of `OnInspectorGUI()`:

```
serializedObject.ApplyModifiedProperties();
```

- In between the calls at the beginning and end of `OnInspectorGUI()` will be the GUI calls for each item slot. Since each item slot will basically be the same, create another function below `OnInspectorGUI()` that can be called multiple times and will take one index parameter specifying which slot is being used:

```
private void ItemSlotGUI(int index)
{
    // Do stuff
}
```

- Each item will be displayed in a box using a vertical layout group. To make a vertical layout group, use the `BeginVertical()` function of the `EditorGUILayout` class. To give it a specific style, use the `GUI.skin.box` parameter to indicate that all the things in the vertical layout group will be drawn in a box.

```
EditorGUILayout.BeginVertical(GUI.skin.box);
```

- At the end of the `ItemSlotGUI()` function, end the vertical layout group by using the `EndVertical()` function:

```
EditorGUILayout.EndVertical();
```

- Since it is a box, the rest of the GUI should not be drawn overlapping with the edit of the box, so indent everything slightly after the `BeginVertical()` call:

```
EditorGUI.indentLevel++;
```

- Accordingly, reset the indent before the `EndVertical()` call so the indent does not keep increasing:

```
EditorGUI.indentLevel--;
```

- First thing that needs to be displayed is a fold out, like the arrows in the Hierarchy for expanding or contracting lists of `GameObject` children, that display the rest of the GUI:

```
showItemSlots[index] = EditorGUILayout.Foldout(showItemSlots[index], "Item slot "
                                                + index);
```

The `Foldout()` function takes a `bool` that determines whether the fold out is open or closed. But, the function also takes into account whether or not the user has clicked on the frame. If the user clicks on it, it changes the value and returns the changed value. So, it takes in a show item slot value and then it will change that item slot if it needs to.

- To use the `Foldout()` `bool` value, create an if statement to check whether something from the `showItemSlots[index]` is being shown. If something is being shown, then display the item image and the item using the defaults for the serialized properties.

```
EditorGUILayout.PropertyField(itemImagesProperty.GetArrayElementAtIndex(index));
```

The array is represented by a serialized property. But that array has many sub-objects, also represented by serialized properties, which are found by using the `GetArrayElementAtIndex()` function which returns another serialized property of a specific element.

- Next, do the same thing but for the `itemsProperty`:

```
EditorGUILayout.PropertyField(itemsProperty.GetArrayElementAtIndex(index));
```

- Between the `Update()` and `ApplyModifiedProperties()` lines in `OnInspectorGUI()`, call the `ItemSlotGUI()` function for each item slot using a for loop to iterate through the number of item slots in the Inventory:

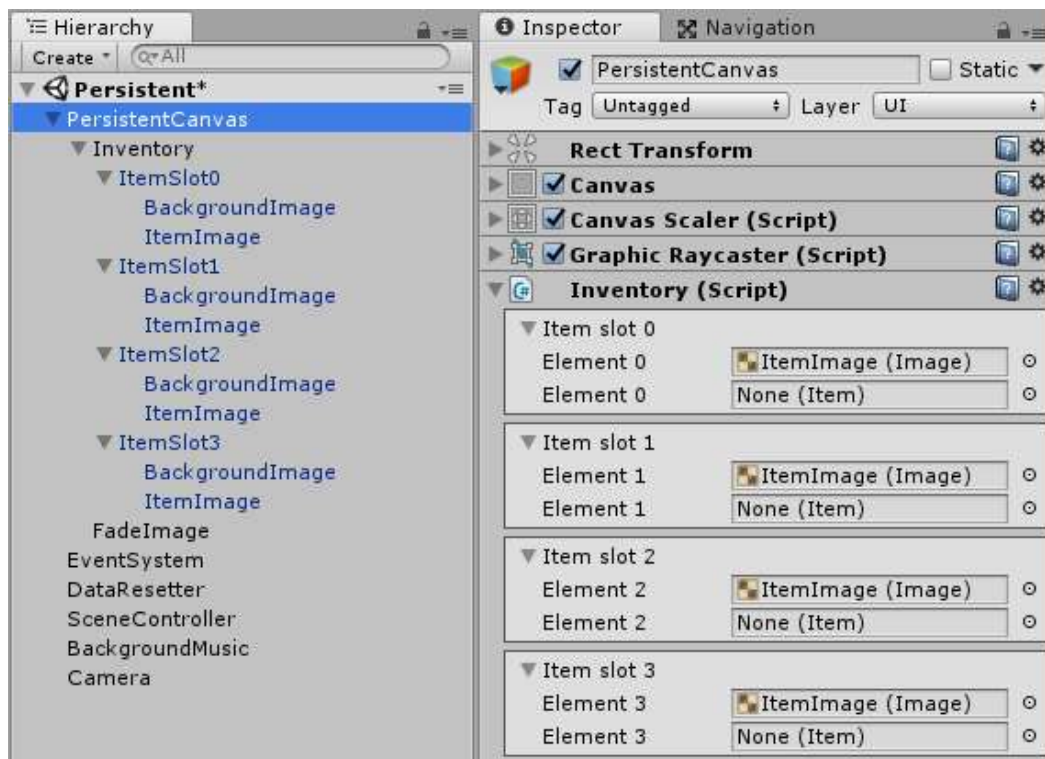
```
for (int i = 0; i < Inventory.numItemSlots; i++)
{
    ItemSlotGUI(i);
}
```

- Save the **InventoryEditor** script and return to the Unity editor.

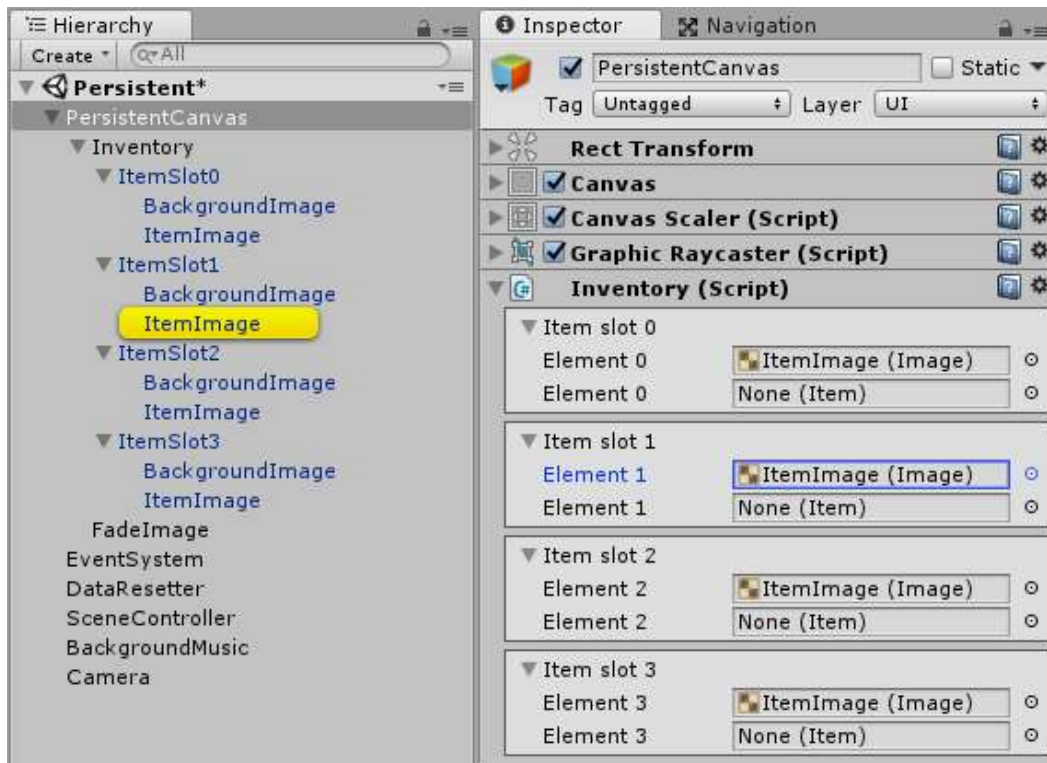
To observe the change in the Inspector, select the **PersistentCanvas** GameObject in the Hierarchy. The Inspector now displays the item slots together, drawn in a box, slightly indented, and the elements are next to each other rather than the two completely separate arrays.

Now, to populate the item slots, a reference to each individual **BackgroundImage** and **ItemImage** is required. To do this:

- In the Hierarchy, expand the **PersistentCanvas** GameObject, expand the **Inventory** GameObject, and then expand **ItemSlot0** through **ItemSlot3** so that the **ItemImage** child GameObjects are displayed for each item slot
- Drag the **ItemImage** from **ItemSlot0** to the **None (Image)** field of the top **Element 0** of **Item slot 0** in the Inspector pane
- Repeat for **ItemSlot1**, **ItemSlot2**, and **ItemSlot3** item images



- Click on each image element in the Inspector to doublecheck. As shown below, clicking on the **ItemImage (Image)** in the top **Element 1** field of **Item slot 1** in the Inspector causes the corresponding **ItemImage** for **ItemSlot1** to be highlighted in the Hierarchy.



- Save the scene.
- Enter Play mode to test the new UI.

The player can exit the first scene to the market scene. There the player can pick up the coin and the fish, then trade the coin for coffee, and finally scare the bird and pick up the dropped glasses. All the objects appear in the inventory vertical layout group on the right of the screen.

Inventory Script

```
using UnityEngine;
using UnityEngine.UI;

public class Inventory : MonoBehaviour {

    public Image[] itemImages = new Image[numItemSlots];
    public Item[] items = new Item[numItemSlots];

    public const int numItemSlots = 4;

    public void AddItem(Item itemToAdd)
    {
        for (int i = 0; i < items.Length; i++)
        {
            if (items[i] == null)
            {
                items[i] = itemToAdd;
                itemImages[i].sprite = itemToAdd.sprite;
            }
        }
    }
}
```

```

        itemImages[i].enabled = true;
        return;
    }
}

public void RemoveItem(Item itemToRemove)
{
    for (int i = 0; i < items.Length; i++)
    {
        if (items[i] == itemToRemove)
        {
            items[i] = null;
            itemImages[i].sprite = null;
            itemImages[i].enabled = false;
            return;
        }
    }
}
}

```

InventoryEditor Script

```

using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(Inventory))]
public class InventoryEditor : Editor {

    private SerializedProperty itemImagesProperty;
    private SerializedProperty itemsProperty;
    private bool[] showItemSlots = new bool[Inventory.numItemSlots];

    private const string inventoryPropItemImagesName = "itemImages";
    private const string inventoryPropItemsName = "items";

    private void OnEnable()
    {
        itemImagesProperty = serializedObject.FindProperty(inventoryPropItemImagesName);
        itemsProperty = serializedObject.FindProperty(inventoryPropItemsName);
    }

    public override void OnInspectorGUI()
    {
        serializedObject.Update();

        for (int i = 0; i < Inventory.numItemSlots; i++)
        {
            ItemSlotGUI(i);
        }

        serializedObject.ApplyModifiedProperties();
    }

    private void ItemSlotGUI(int index)
    {

```

```

EditorGUILayout.BeginVertical(GUI.skin.box);
EditorGUI.indentLevel++;

showItemSlots[index] = EditorGUILayout.Foldout(showItemSlots[index], "Item slot "
+ index);

if (showItemSlots[index])
{
    EditorGUILayout.PropertyField(itemImagesProperty.GetArrayElementAtIndex(index
));
    EditorGUILayout.PropertyField(itemsProperty.GetArrayElementAtIndex(index));
}

EditorGUI.indentLevel--;
EditorGUILayout.EndVertical();
}
}

```

Conditions

Phase 03 – Conditions (Slides)

Conditions (Slides)

Brief

- Make an interaction system that allows the player to interact with the game
- Change the game's state (e.g., collecting a coin)
- Remove the coin from the scene and the game knows that the coin has been collected
- Clicking on interactable objects in the scene triggers a series of reactions based on the game state
- Interaction system must support all game interactions except the player movement (interact, react, condition)

The Approach


- Create an **Interactable** GameObject which receives user click input as an OnClickEvent and will use the event system, and based on a set of conditions will call a series of reactions
- The **Interactable** GameObject will be:
 - **Stand-alone** and hold all logic and events
 - **Decoupled** from the actual props in the scene (e.g., the coin or the painting will not be an Interactable, but there will be an Interactable GameObject with a collider that can be set around the coin or the painting that will then trigger the interaction)
- The **Interactable** GameObject will have:
 - A **Collider** to detect clicks
 - An **EventTrigger** to process clicks
 - An **Interactable** component to control the logic of the interaction

Player moves to Interactable → Player clicks on the Interactable → The ConditionCollection will have a series of Conditions and will check to see if all of the conditions correct, and if they are not correct, move onto another ConditionCollection to see if those conditions are correct → If one of the ConditionCollections is correct, the a ReactionCollection is called to trigger a set of reactions (e.g., animation, text, etc. reactions)

- **Conditions** are:
 - (Simple) Data objects that contain only an **identifier** and a **boolean** state
 - Saved as **ScriptableObject** assets that can be used to compare the state of a **Condition**
 - **Reactions**:
 - Accommodate a wide variety of possible actions
 - Use **Inheritance** and **Polymorphism** to create specific **Reactions** for each possible type of action
 - **Reactions** will be **Encapsulated**
 - Each interactable does not need to know about what reactions it will play. All it needs to know is that there are reactions and it should play them under specific circumstances. A separate component, the ReactionCollection, that will have a React() function that will distribute its information and tell the scene what should happen.
 - The **Interactable** will have a single **object reference** to a **Reaction**
 - The **Interactable** will **call** a single **React** function regardless of how many actual **Reactions** there are
 - To improve workflow, **Custom Inspectors** will be created to accommodate all the different types of **Reactions**, **Conditions**, and **Interactables**
 -
-

Creating the Adventure Game Tutorial Project for Phase 03

To create a project for Phase 03 of the Adventure Game Tutorial, do the following:

- Start Unity 2017.2.0f3
- With the **3D** option selected, click the **New** link to create a new project named **Unity_Tutorial_AdventureGame_Phase03** (or some other meaningful name) and then click the blue **Create project** button
- Click on the **Asset Store** tab, type **Adventure Game Tutorial** in the search field at the top of the **Asset Store** pane, and click the Search button  (or press **Enter**)
- Scroll down and choose **3/6 – Adventure Tutorial – Conditions** from the results list
- Click the blue **Download** button
- Click the **Import** button on the **Importing Complete Project** popup message
- Click the **Import** button on the **Import Unity Package** popup message

Initial Build Steps

Conditions (Slides)

Brief

- Create a system to make **Reactions** conditional

Approach

- All Conditions will be **ScriptableObjects**
- Some **Conditions** will be saved as assets to represent the global state of the game
- Some **Conditions** will be instances in the scene which represent the required state

Make sure to close MonoDevelop and Visual Studio so as to not get the two projects mixed up. To set up the Phase 03 project, do the following:

- Navigate to **Project** pane
-

Stopped at 5:24

ConditionCollection Script

```
using UnityEngine;
public class ConditionCollection : ScriptableObject
{
    public string description;
    public Condition[] requiredConditions = new Condition[0];
    public ReactionCollection reactionCollection;
    public bool CheckAndReact()
    {
        for (int i = 0; i < requiredConditions.Length; i++)
        {
            if (!AllConditions.CheckCondition (requiredConditions[i]))
                return false;
        }
        if(reactionCollection)
            reactionCollection.React();
        return true;
    }
}
```



```
}
```

ConditionCollectionEditor Script

```
using UnityEngine;
using UnityEditor;
[CustomEditor(typeof(ConditionCollection))]
public class ConditionCollectionEditor : EditorWithSubEditors<ConditionEditor,
Condition>
{
    public SerializedProperty collectionsProperty;
    private ConditionCollection conditionCollection;
    private SerializedProperty descriptionProperty;
    private SerializedProperty conditionsProperty;
    private SerializedProperty reactionCollectionProperty;
    private const float conditionButtonWidth = 30f;
    private const float collectionButtonWidth = 125f;
    private const string conditionCollectionPropDescriptionName = "description";
    private const string conditionCollectionPropRequiredConditionsName =
"requiredConditions";
    private const string conditionCollectionPropReactionCollectionName =
"reactionCollection";
    private void OnEnable ()
    {
        conditionCollection = (ConditionCollection)target;
        if (target == null)
        {
            DestroyImmediate (this);
            return;
        }
        descriptionProperty =
serializedObject.FindProperty(conditionCollectionPropDescriptionName);
        conditionsProperty =
serializedObject.FindProperty(conditionCollectionPropRequiredConditionsName);
        reactionCollectionProperty =
serializedObject.FindProperty(conditionCollectionPropReactionCollectionName);
        CheckAndCreateSubEditors (conditionCollection.requiredConditions);
    }
    private void OnDisable ()
    {
        CleanupEditors ();
    }
    protected override void SubEditorSetup (ConditionEditor editor)
    {
        editor.editorType = ConditionEditor.EditorType.ConditionCollection;
        editor.conditionsProperty = conditionsProperty;
    }
    public override void OnInspectorGUI ()
    {
        serializedObject.Update ();
        CheckAndCreateSubEditors(conditionCollection.requiredConditions);

        EditorGUILayout.BeginVertical(GUI.skin.box);
        EditorGUI.indentLevel++;
        EditorGUILayout.BeginHorizontal();
        descriptionProperty.isExpanded =
EditorGUILayout.Foldout(descriptionProperty.isExpanded,
descriptionProperty.stringValue);
        if (GUILayout.Button("Remove Collection",
GUILayout.Width(collectionButtonWidth)))
```

```

        {
            collectionsProperty.RemoveFromObjectArray (conditionCollection);
        }
        EditorGUILayout.EndHorizontal();

        if (descriptionProperty.isExpanded)
        {
            ExpandedGUI ();
        }

        EditorGUI.indentLevel--;
        EditorGUILayout.EndVertical();
        serializedObject.ApplyModifiedProperties();
    }
    private void ExpandedGUI ()
    {
        EditorGUILayout.Space();
        EditorGUILayout.PropertyField(descriptionProperty);
        EditorGUILayout.Space();
        float space = EditorGUIUtility.currentViewWidth / 3f;
        EditorGUILayout.BeginHorizontal();
        EditorGUILayout.LabelField("Condition", GUILayout.Width(space));
        EditorGUILayout.LabelField("Satisfied?", GUILayout.Width(space));
        EditorGUILayout.LabelField("Add/Remove", GUILayout.Width(space));
        EditorGUILayout.EndHorizontal();
        EditorGUILayout.BeginVertical(GUI.skin.box);
        for (int i = 0; i < subEditors.Length; i++)
        {
            subEditors[i].OnInspectorGUI();
        }
        EditorGUILayout.EndHorizontal();
        EditorGUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace ();
        if (GUILayout.Button("+", GUILayout.Width(conditionButtonWidth)))
        {
            Condition newCondition = ConditionEditor.CreateCondition();
            conditionsProperty.AddToObjectArray(newCondition);
        }
        EditorGUILayout.EndHorizontal();
        EditorGUILayout.Space();
        EditorGUILayout.PropertyField(reactionCollectionProperty);
    }
    public static ConditionCollection CreateConditionCollection()
    {
        ConditionCollection newConditionCollection =
        CreateInstance<ConditionCollection>();
        newConditionCollection.description = "New condition collection";
        newConditionCollection.requiredConditions = new Condition[1];
        newConditionCollection.requiredConditions[0] =
        ConditionEditor.CreateCondition();
        return newConditionCollection;
    }
}

```

Reactions

ReactionCollectionEditor Script

```
using System;
using UnityEngine;
using System.Collections.Generic;
using UnityEditor;
[CustomEditor(typeof(ReactionCollection))]
public class ReactionCollectionEditor : EditorWithSubEditors<ReactionEditor, Reaction>
{
    private ReactionCollection reactionCollection;
    private SerializedProperty reactionsProperty;
    private Type[] reactionTypes;
    private string[] reactionTypeNames;
    private int selectedIndex;
    private const float dropAreaHeight = 50f;
    private const float controlSpacing = 5f;
    private const string reactionsPropName = "reactions";
    private readonly float verticalSpacing = EditorGUIUtility.standardVerticalSpacing;
    private void OnEnable ()
    {
        reactionCollection = (ReactionCollection)target;
        reactionsProperty = serializedObject.FindProperty(reactionsPropName);
        CheckAndCreateSubEditors (reactionCollection.reactions);
        SetReactionNamesArray ();
    }
    private void OnDisable ()
    {
        CleanupEditors ();
    }
    protected override void SubEditorSetup (ReactionEditor editor)
    {
        editor.reactionsProperty = reactionsProperty;
    }
    public override void OnInspectorGUI ()
    {
        serializedObject.Update ();
        CheckAndCreateSubEditors(reactionCollection.reactions);
        for (int i = 0; i < subEditors.Length; i++)
        {
            subEditors[i].OnInspectorGUI ();
        }
    }
}
```

```

        if (reactionCollection.reactions.Length > 0)
        {
            EditorGUILayout.Space();
            EditorGUILayout.Space ();
        }
        Rect fullWidthRect = GUILayoutUtility.GetRect(GUIContent.none, GUIStyle.none,
GUILayout.Height(dropAreaHeight + verticalSpacing));
        Rect leftAreaRect = fullWidthRect;
        leftAreaRect.y += verticalSpacing * 0.5f;
        leftAreaRect.width *= 0.5f;
        leftAreaRect.width -= controlSpacing * 0.5f;
        leftAreaRect.height = dropAreaHeight;
        Rect rightAreaRect = leftAreaRect;
        rightAreaRect.x += rightAreaRect.width + controlSpacing;
        TypeSelectionGUI (leftAreaRect);
        DragAndDropAreaGUI (rightAreaRect);
        DraggingAndDropping(rightAreaRect, this);
        serializedObject.ApplyModifiedProperties ();
    }
    private void TypeSelectionGUI (Rect containingRect)
    {
        Rect topHalf = containingRect;
        topHalf.height *= 0.5f;

        Rect bottomHalf = topHalf;
        bottomHalf.y += bottomHalf.height;
        selectedIndex = EditorGUI.Popup(topHalf, selectedIndex, reactionTypeNames);
        if (GUI.Button (bottomHalf, "Add Selected Reaction"))
        {
            Type reactionType = reactionTypes[selectedIndex];
            Reaction newReaction = ReactionEditor.CreateReaction (reactionType);
            reactionsProperty.AddToArray (newReaction);
        }
    }
    private static void DragAndDropAreaGUI (Rect containingRect)
    {
        GUIStyle centredStyle = GUI.skin.box;
        centredStyle.alignment = TextAnchor.MiddleCenter;
        centredStyle.normal.textColor = GUI.skin.button.normal.textColor;
        GUI.Box (containingRect, "Drop new Reactions here", centredStyle);
    }
    private static void DraggingAndDropping (Rect dropArea, ReactionCollectionEditor
editor)
    {
        Event currentEvent = Event.current;
        if (!dropArea.Contains (currentEvent.mousePosition))
            return;
        switch (currentEvent.type)
        {
            case EventType.DragUpdated:
                DragAndDrop.visualMode = IsDragValid () ? DragAndDropVisualMode.Link :
DragAndDropVisualMode.Rejected;
                currentEvent.Use ();
                break;
            case EventType.DragPerform:

                DragAndDrop.AcceptDrag ();

                for (int i = 0; i < DragAndDrop.objectReferences.Length; i++)
                {
                    MonoScript script = DragAndDrop.objectReferences[i] as MonoScript;
                    Type reactionType = script.GetClass();

```

```

        Reaction newReaction = ReactionEditor.CreateReaction
(reactionType);
        editor.reactionsProperty.AddToObjectArray (newReaction);
    }
    currentEvent.Use();
    break;
}
}
private static bool IsDragValid ()
{
    for (int i = 0; i < DragAndDrop.objectReferences.Length; i++)
    {
        if (DragAndDrop.objectReferences[i].GetType () != typeof (MonoScript))
            return false;

        MonoScript script = DragAndDrop.objectReferences[i] as MonoScript;
        Type scriptType = script.GetClass ();
        if (!scriptType.IsSubclassOf (typeof (Reaction)))
            return false;
        if (scriptType.IsAbstract)
            return false;
    }
    return true;
}
private void SetReactionNamesArray ()
{
    Type reactionType = typeof (Reaction);
    Type[] allTypes = reactionType.Assembly.GetTypes();
    List<Type> reactionSubTypeList = new List<Type>();
    for (int i = 0; i < allTypes.Length; i++)
    {
        if (allTypes[i].IsSubclassOf (reactionType) && !allTypes[i].IsAbstract)
        {
            reactionSubTypeList.Add(allTypes[i]);
        }
    }
    reactionTypes = reactionSubTypeList.ToArray();
    List<string> reactionTypeNameList = new List<string>();
    for (int i = 0; i < reactionTypes.Length; i++)
    {
        reactionTypeNameList.Add(reactionTypes[i].Name);
    }
    reactionTypeNames = reactionTypeNameList.ToArray();
}
}

```

TextReaction Script

```

using UnityEngine;
public class TextReaction : Reaction
{
    public string message;
    public Color textColor = Color.white;
    public float delay;
    private TextManager textManager;
    protected override void SpecificInit()
    {
        textManager = FindObjectOfType<TextManager> ();
    }
    protected override void ImmediateReaction()

```

```

    {
        textManager.DisplayMessage (message, textColor, delay);
    }
}

```

TextReactionEditor Script

```

using UnityEditor;
using UnityEngine;
[CustomEditor(typeof(TextReaction))]
public class TextReactionEditor : ReactionEditor
{
    private SerializedProperty messageProperty;
    private SerializedProperty textColorProperty;
    private SerializedProperty delayProperty;
    private const float messageGUILines = 3f;
    private const float areaWidthOffset = 19f;
    private const string textReactionPropMessageName = "message";
    private const string textReactionPropTextColorName = "textColor";
    private const string textReactionPropDelayName = "delay";
    protected override void Init ()
    {
        messageProperty = serializedObject.FindProperty (textReactionPropMessageName);
        textColorProperty = serializedObject.FindProperty
(textReactionPropTextColorName);
        delayProperty = serializedObject.FindProperty (textReactionPropDelayName);
    }
    protected override void DrawReaction ()
    {
        EditorGUILayout.BeginHorizontal ();
        EditorGUILayout.LabelField ("Message", GUILayout.Width
(EditorGUIUtility.labelWidth - areaWidthOffset));
        messageProperty.stringValue = EditorGUILayout.TextArea
(messageProperty.stringValue, GUILayout.Height (EditorGUIUtility.singleLineHeight *
messageGUILines));
        EditorGUILayout.EndHorizontal ();
        EditorGUILayout.PropertyField (textColorProperty);
        EditorGUILayout.PropertyField (delayProperty);
    }
    protected override string GetFoldoutLabel ()
    {
        return "Text Reaction";
    }
}

```

Interactables

Interactable Script

```
using UnityEngine;
public class Interactable : MonoBehaviour
{
    public Transform interactionLocation;
    public ConditionCollection[] conditionCollections = new ConditionCollection[0];
    public ReactionCollection defaultReactionCollection;
    public void Interact ()
    {
        for (int i = 0; i < conditionCollections.Length; i++)
        {
            if (conditionCollections[i].CheckAndReact ())
                return;
        }
        defaultReactionCollection.React ();
    }
}
```

InteractableEditor Script

```
using UnityEngine;
using UnityEditor;
[CustomEditor(typeof(Interactable))]
public class InteractableEditor : EditorWithSubEditors<ConditionCollectionEditor,
ConditionCollection>
{
    private Interactable interactable;
    private SerializedProperty interactionLocationProperty;
    private SerializedProperty collectionsProperty;
    private SerializedProperty defaultReactionCollectionProperty;
    private const float collectionButtonWidth = 125f;
    private const string interactablePropInteractionLocationName =
"interactionLocation";
    private const string interactablePropConditionCollectionsName =
"conditionCollections";
    private const string interactablePropDefaultReactionCollectionName =
"defaultReactionCollection";
    private void OnEnable ()
    {
        interactable = (Interactable)target;
        collectionsProperty =
serializedObject.FindProperty(interactablePropConditionCollectionsName);
        interactionLocationProperty =
serializedObject.FindProperty(interactablePropInteractionLocationName);
    }
}
```

```

        defaultReactionCollectionProperty =
serializedObject.FindProperty(interactablePropDefaultReactionCollectionName);

        CheckAndCreateSubEditors(interactable.conditionCollections);
    }
    private void OnDisable ()
    {
        CleanupEditors ();
    }
    protected override void SubEditorSetup(ConditionCollectionEditor editor)
    {
        editor.collectionsProperty = collectionsProperty;
    }
    public override void OnInspectorGUI ()
    {
        serializedObject.Update ();

        CheckAndCreateSubEditors(interactable.conditionCollections);

        EditorGUILayout.PropertyField (interactionLocationProperty);
        for (int i = 0; i < subEditors.Length; i++)
        {
            subEditors[i].OnInspectorGUI ();
            EditorGUILayout.Space ();
        }
        EditorGUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace ();
        if (GUILayout.Button("Add Collection",
GUILayout.Width(collectionButtonWidth)))
        {
            ConditionCollection newCollection =
ConditionCollectionEditor.CreateConditionCollection ();
            collectionsProperty.AddToObjectArray (newCollection);
        }
        EditorGUILayout.EndHorizontal ();
        EditorGUILayout.Space ();
        EditorGUILayout.PropertyField (defaultReactionCollectionProperty);
        serializedObject.ApplyModifiedProperties ();
    }
}

```

Game State

SceneController Script

```
using System;
using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;
public class SceneController : MonoBehaviour
{
    public event Action BeforeSceneUnload;
    public event Action AfterSceneLoad;
    public CanvasGroup faderCanvasGroup;
    public float fadeDuration = 1f;
    public string startingSceneName = "SecurityRoom";
    public string initialStartingPositionName = "DoorToMarket";
    public SaveData playerSaveData;

    private bool isFading;
    private IEnumerator Start ()
    {
        faderCanvasGroup.alpha = 1f;
        playerSaveData.Save (PlayerMovement.startingPositionKey,
initialStartingPositionName);
        yield return StartCoroutine (LoadSceneAndSetActive (startingSceneName));
        StartCoroutine (Fade (0f));
    }
    public void FadeAndLoadScene (SceneReaction sceneReaction)
    {
        if (!isFading)
        {
            StartCoroutine (FadeAndSwitchScenes (sceneReaction.sceneName));
        }
    }
    private IEnumerator FadeAndSwitchScenes (string sceneName)
    {
        yield return StartCoroutine (Fade (1f));
        if (BeforeSceneUnload != null)
            BeforeSceneUnload ();
        yield return SceneManager.UnloadSceneAsync (SceneManager.GetActiveScene
().buildIndex);
        yield return StartCoroutine (LoadSceneAndSetActive (sceneName));
        if (AfterSceneLoad != null)
            AfterSceneLoad ();

        yield return StartCoroutine (Fade (0f));
    }
    private IEnumerator LoadSceneAndSetActive (string sceneName)
    {
        yield return SceneManager.LoadSceneAsync (sceneName, LoadSceneMode.Additive);
        Scene newlyLoadedScene = SceneManager.GetSceneAt (SceneManager.sceneCount -
1);
        SceneManager.SetActiveScene (newlyLoadedScene);
    }
    private IEnumerator Fade (float finalAlpha)
```

```

    {
        isFading = true;
        faderCanvasGroup.blocksRaycasts = true;
        float fadeSpeed = Mathf.Abs (faderCanvasGroup.alpha - finalAlpha) /
fadeDuration;
        while (!Mathf.Approximately (faderCanvasGroup.alpha, finalAlpha))
        {
            faderCanvasGroup.alpha = Mathf.MoveTowards (faderCanvasGroup.alpha,
finalAlpha,
                fadeSpeed * Time.deltaTime);
            yield return null;
        }
        isFading = false;
        faderCanvasGroup.blocksRaycasts = false;
    }
}

```

SaveData Script

```

using System;
using UnityEngine;
using System.Collections.Generic;
[CreateAssetMenu]
public class SaveData : ResettableScriptableObject
{
    [Serializable]
    public class KeyValuePairLists<T>
    {
        public List<string> keys = new List<string>();
        public List<T> values = new List<T>();
        public void Clear ()
        {
            keys.Clear ();
            values.Clear ();
        }
        public void TrySetValue (string key, T value)
        {
            int index = keys.FindIndex(x => x == key);
            if (index > -1)
            {
                values[index] = value;
            }
            else
            {
                keys.Add (key);
                values.Add (value);
            }
        }
        public bool TryGetValue (string key, ref T value)
        {
            int index = keys.FindIndex(x => x == key);
            if (index > -1)
            {
                value = values[index];
                return true;
            }
            return false;
        }
    }
}

```

```

    public KeyValuePairLists<bool> boolKeyValuePairLists = new KeyValuePairLists<bool>
    ();
    public KeyValuePairLists<int> intKeyValuePairLists = new KeyValuePairLists<int>();
    public KeyValuePairLists<string> stringKeyValuePairLists = new
KeyValuePairLists<string>();
    public KeyValuePairLists<Vector3> vector3KeyValuePairLists = new
KeyValuePairLists<Vector3>();
    public KeyValuePairLists<Quaternion> quaternionKeyValuePairLists = new
KeyValuePairLists<Quaternion>();
    public override void Reset ()
    {
        boolKeyValuePairLists.Clear ();
        intKeyValuePairLists.Clear ();
        stringKeyValuePairLists.Clear ();
        vector3KeyValuePairLists.Clear ();
        quaternionKeyValuePairLists.Clear ();
    }
    private void Save<T>(KeyValuePairLists<T> lists, string key, T value)
    {
        lists.TrySetValue(key, value);
    }
    private bool Load<T>(KeyValuePairLists<T> lists, string key, ref T value)
    {
        return lists.TryGetValue(key, ref value);
    }
    public void Save (string key, bool value)
    {
        Save(boolKeyValuePairLists, key, value);
    }
    public void Save (string key, int value)
    {
        Save(intKeyValuePairLists, key, value);
    }
    public void Save (string key, string value)
    {
        Save(stringKeyValuePairLists, key, value);
    }
    public void Save (string key, Vector3 value)
    {
        Save(vector3KeyValuePairLists, key, value);
    }
    public void Save (string key, Quaternion value)
    {
        Save(quaternionKeyValuePairLists, key, value);
    }
    public bool Load (string key, ref bool value)
    {
        return Load(boolKeyValuePairLists, key, ref value);
    }
    public bool Load (string key, ref int value)
    {
        return Load (intKeyValuePairLists, key, ref value);
    }
    public bool Load (string key, ref string value)
    {
        return Load (stringKeyValuePairLists, key, ref value);
    }
    public bool Load (string key, ref Vector3 value)
    {
        return Load(vector3KeyValuePairLists, key, ref value);
    }
    public bool Load (string key, ref Quaternion value)
    {

```

```
        return Load (quaternionKeyValuePairLists, key, ref value);  
    }  
}
```