



*What will we create?*

We will create a ***Point & Click Adventure Game***  
***in 6 phases***

In each phase we will create a specific system for  
the game as if we were working in a games studio

By the end of the day we will have presented all the  
major systems for this game

*What will we create?*

***The Point & Click Adventure Game has been designed to a level beyond most tutorial projects***

This project is rated as ***intermediate*** in both concept and execution

In this tutorial we will cover the most important points of the project in lecture.

*The order of the tutorial.*

*Introduction*

*Project Overview*

**Phase 01 - The Player**

**Phase 02 - Inventory**

**Phase 03 - Conditions**

**Phase 04 - Reactions**

**Phase 05 - Interactables**

**Phase 06 - Game State**

# *What you need to know*

- Phase Asset Store projects:
  - Each phase has it's own Asset Store project you will need to download and import
  - There is also a completed Asset Store project with annotations

*Let's Get  
Started!*



# Overview



# Project Overview

## Key Topics for this presentation

# Project Overview

## Key Topics for this presentation

- Player Control

# Project Overview

## Key Topics for this presentation

- Player Control
- Inventory

# Project Overview

## Key Topics for this presentation

- Player Control
- Inventory
- Interaction System

# Project Overview

## **Key Topics for this presentation**

- Player Control
- Inventory
- Interaction System
- Game State

# Project Overview

*Understanding the basic relationships of the project*

# Project Overview

## *Event System*



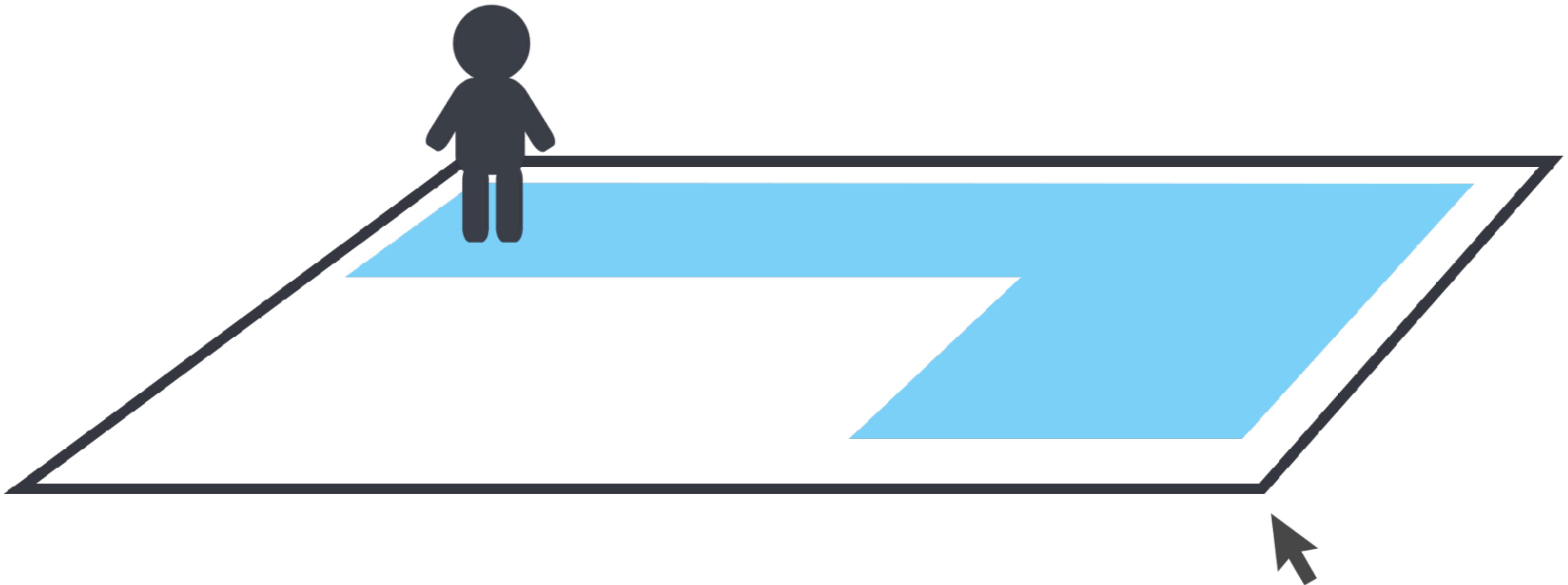
# Project Overview

## ***Navmesh***



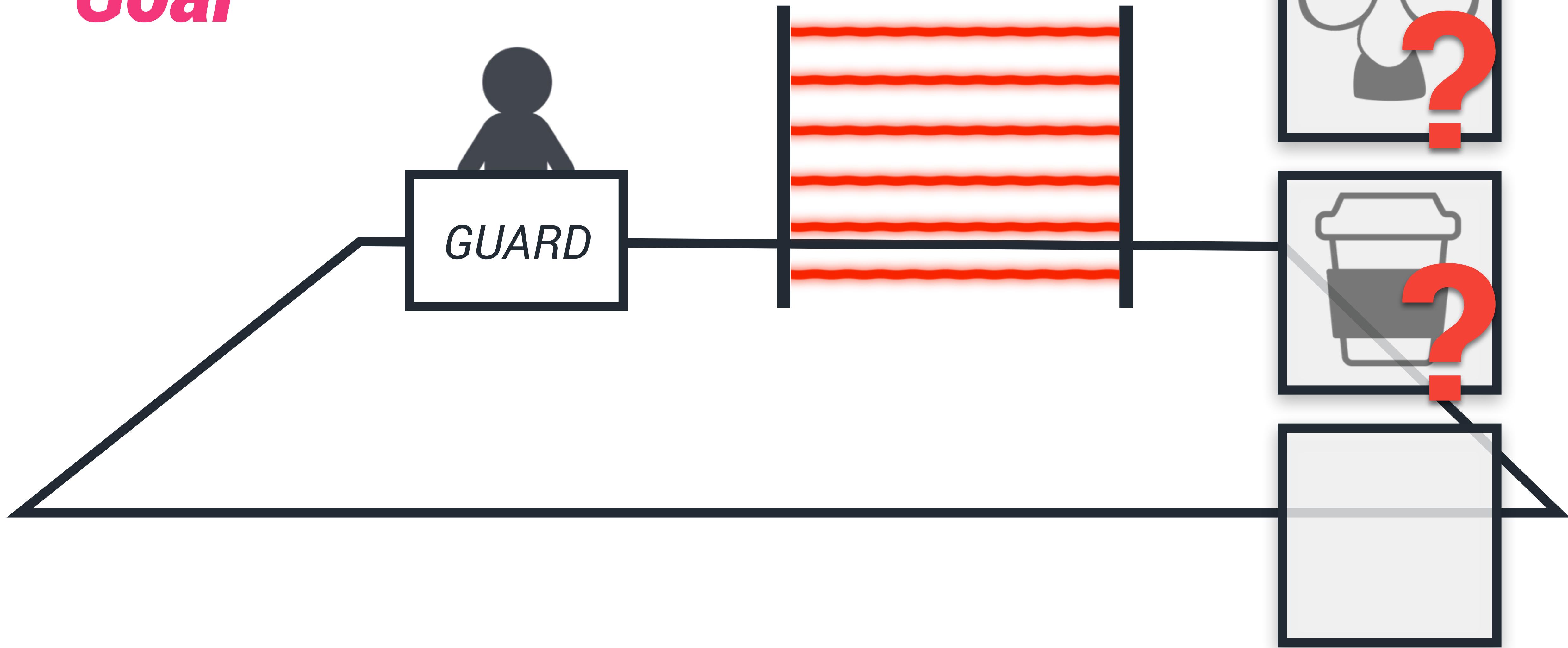
# Project Overview

## ***Animator***



# Project Overview

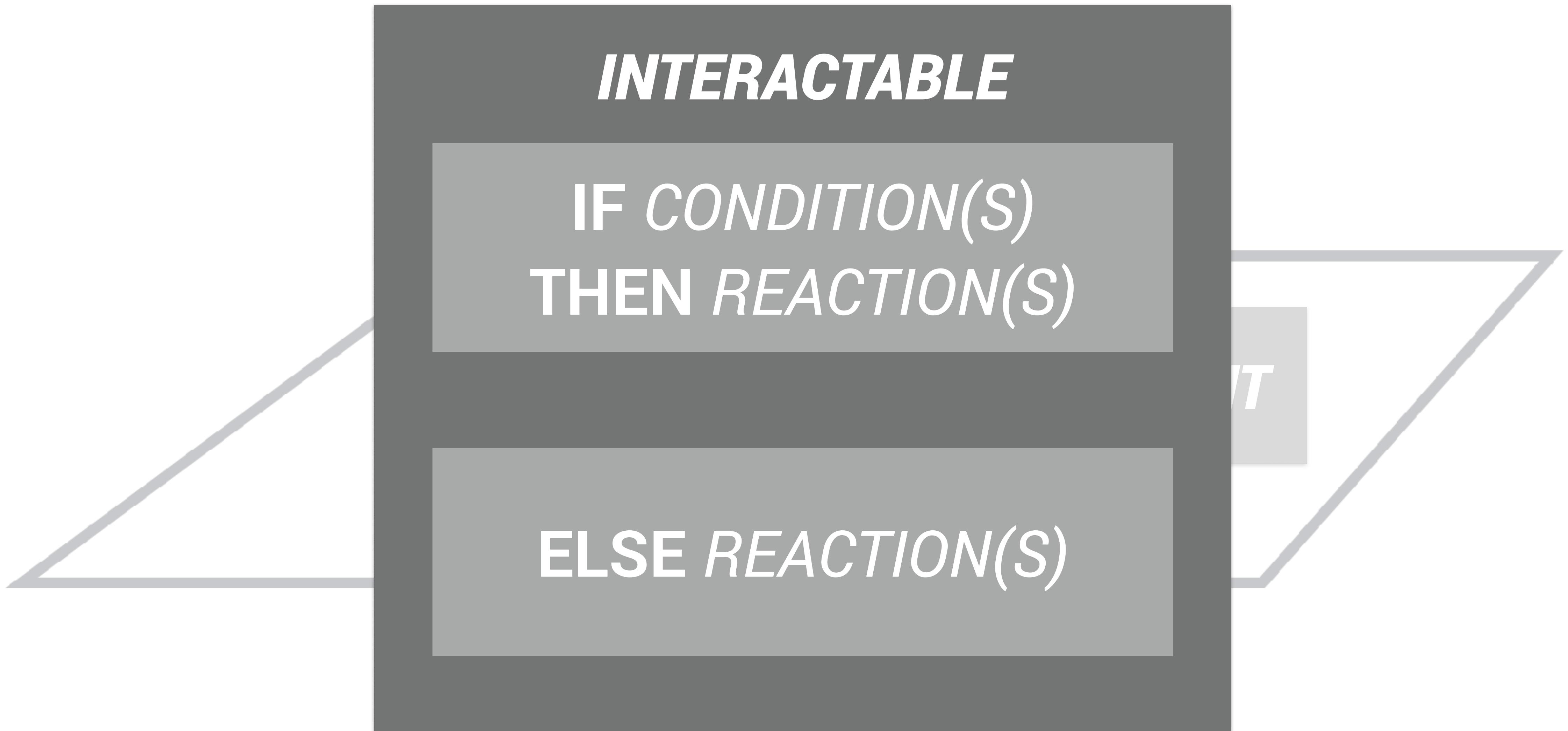
## *Goal*



# Project Overview



# Project Overview



# Project Overview



# Project Overview

## ***FISH INTERACTABLE***

**IF PICKED UP FISH**

**THEN**

- T** *"I don't want another fish"*
- 🔊** *Play Voiceover*

**ELSE**

- ▶** *Play Animation*
- 📝** *Store Fish in Inventory*
- 🔊** *Collected Item Sound*
- 🔊** *Play Voiceover*
- T** *"I doubt this is a clue"*
- ?** *Set pickedUpFish = true*

# Project Overview

## ***GUARD INTERACTABLE***

**IF HAS > GLASSES & COFFEE**

**THEN**

- ▶ *Switch off Gate Lasers*
- ▶ *“Hey Frank, go on through!”*
- ▶ *Play LaserPowerDown*
- ▶ *Play Voiceover*
- ? *LaserOff = True*

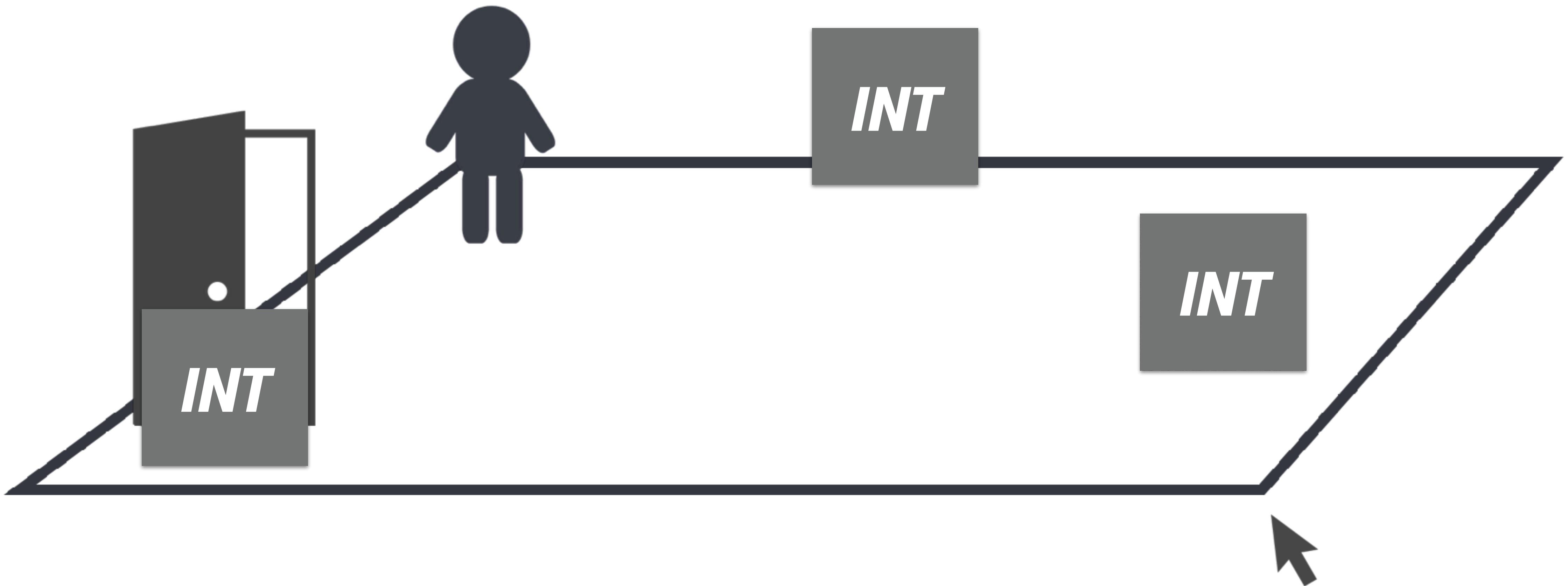
**IF HAS > GLASSES**

- ▶ *“Go get the boss some Coffee”*
- ▶ *Play Voiceover*

**ELSE**

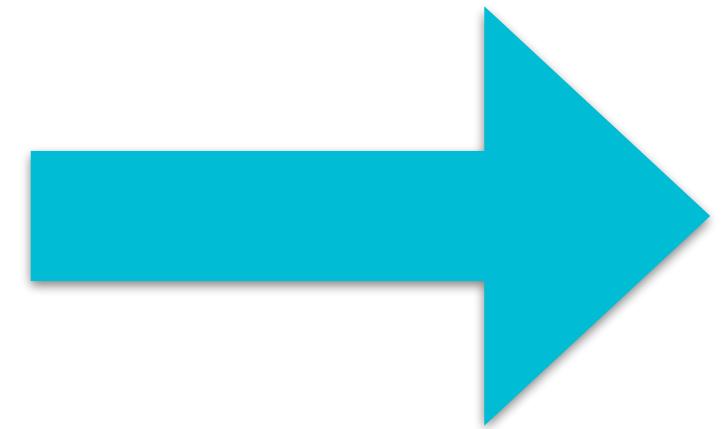
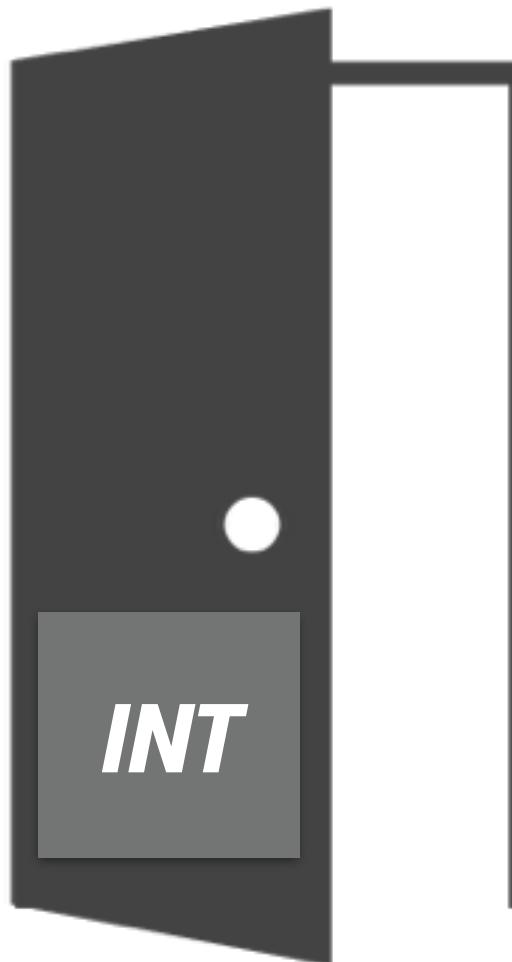
- ▶ *“Get outta here!”*
- ▶ *Play Voiceover*

# Project Overview



Project Overview

# ***Scene Management***

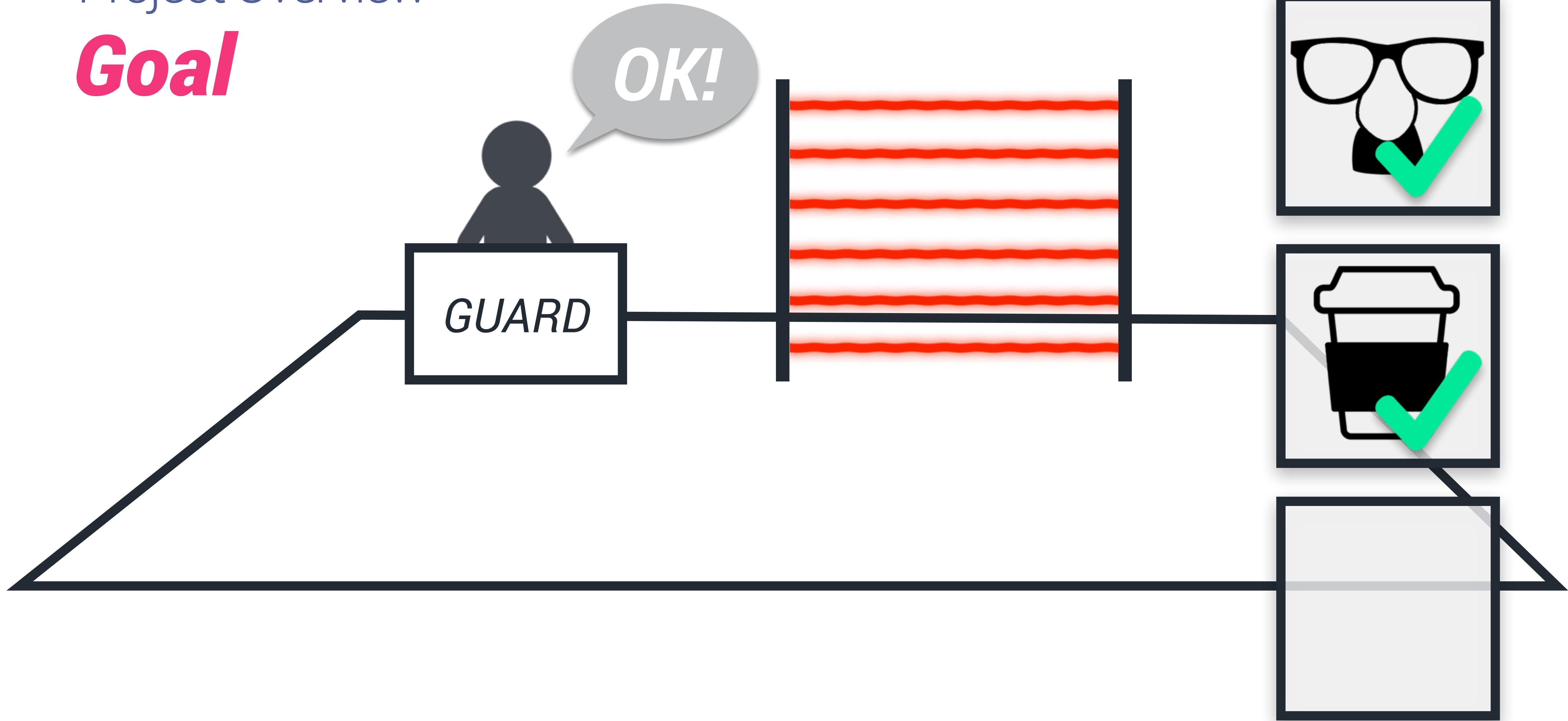


***Load Scene***

*Maintain Inventory  
& Scene State*

# Project Overview

## *Goal*



# Project Overview

*What we will learn in each phase*

# Project Overview

## The Player

# Project Overview

## The Player

*Build a click to move animated character using:*

# Project Overview

## The Player

*Build a click to move animated character using:*

- EventSystem

# Project Overview

## The Player

*Build a click to move animated character using:*

- EventSystem
- NavMesh

# Project Overview

## The Player

*Build a click to move animated character using:*

- EventSystem
- NavMesh
- Animator

# Project Overview

## The Player

*Build a click to move animated character using:*

- EventSystem
- NavMesh
- Animator
- Prefabs

# Project Overview

## Inventory

# Project Overview

## Inventory

*Build a UI and Item Management System for Player Inventory*

# Project Overview

## Inventory

*Build a UI and Item Management System for Player Inventory*

- UI System

# Project Overview

## Inventory

*Build a UI and Item Management System for Player Inventory*

- UI System
- Editor Scripting

# Project Overview

## Interaction System

# Project Overview

## Interaction System

*Build a system allowing the Player to interact with the game*

# Project Overview

## Interaction System

*Build a system allowing the Player to interact with the game*

- Conditions

# Project Overview

## Interaction System

*Build a system allowing the Player to interact with the game*

- Conditions
- Reactions

# Project Overview

## Interaction System

*Build a system allowing the Player to interact with the game*

- Conditions
- Reactions
- Interactables

# Project Overview

## Interaction System - Conditions

# Project Overview

## Interaction System - Conditions

*Create a system to check the current game state*

# Project Overview

## Interaction System - Conditions

*Create a system to check the current game state*

- Scripting Patterns

# Project Overview

## Interaction System - Conditions

*Create a system to check the current game state*

- Scripting Patterns
- Scriptable Objects

# Project Overview

## Interaction System - Conditions

*Create a system to check the current game state*

- Scripting Patterns
- Scriptable Objects
- Generic Classes

# Project Overview

## Interaction System - Conditions

*Create a system to check the current game state*

- Scripting Patterns
- Scriptable Objects
- Generic Classes
- Inheritance

# Project Overview

## Interaction System - Conditions

*Create a system to check the current game state*

- Scripting Patterns
- Scriptable Objects
- Generic Classes
- Inheritance
- Extension Methods

# Project Overview

## Interaction System - Reactions

# Project Overview

## Interaction System - Reactions

*Create a system to perform actions based on condition state*

# Project Overview

## Interaction System - Reactions

*Create a system to perform actions based on condition state*

- Polymorphism

# Project Overview

## Interaction System - Reactions

*Create a system to perform actions based on condition state*

- Polymorphism
- Further editor scripting

# Project Overview

## Interaction System - Reactions

*Create a system to perform actions based on condition state*

- Polymorphism
- Further editor scripting
- Serialization

# Project Overview

## Interaction System - Interactables

# Project Overview

## Interaction System - Interactables

*Create a system to define what the player can interact with*

# Project Overview

## Interaction System - Interactables

*Create a system to define what the player can interact with*

- Interactable Geometry

# Project Overview

## Interaction System - Interactables

*Create a system to define what the player can interact with*

- Interactable Geometry
- EventSystem

# Project Overview

## Interaction System - Interactables

*Create a system to define what the player can interact with*

- Interactable Geometry
- EventSystem
- Interaction System Summary

# Project Overview

## Game State

# Project Overview

## Game State

*Creating a system to load scenes while preserving game state*

# Project Overview

## Game State

*Creating a system to load scenes while preserving game state*

- Scene Manager

# Project Overview

## Game State

*Creating a system to load scenes while preserving game state*

- Scene Manager
- ScriptableObjects as *temporary* runtime data storage

# Project Overview

## Game State

*Creating a system to load scenes while preserving game state*

- Scene Manager
- ScriptableObjects as *temporary* runtime data storage
- Delegates

# Project Overview

## Game State

*Creating a system to load scenes while preserving game state*

- Scene Manager
- ScriptableObjects as *temporary* runtime data storage
- Delegates
- Lambda Expressions

# Project Overview

*Understanding the architecture of Scene Loading*

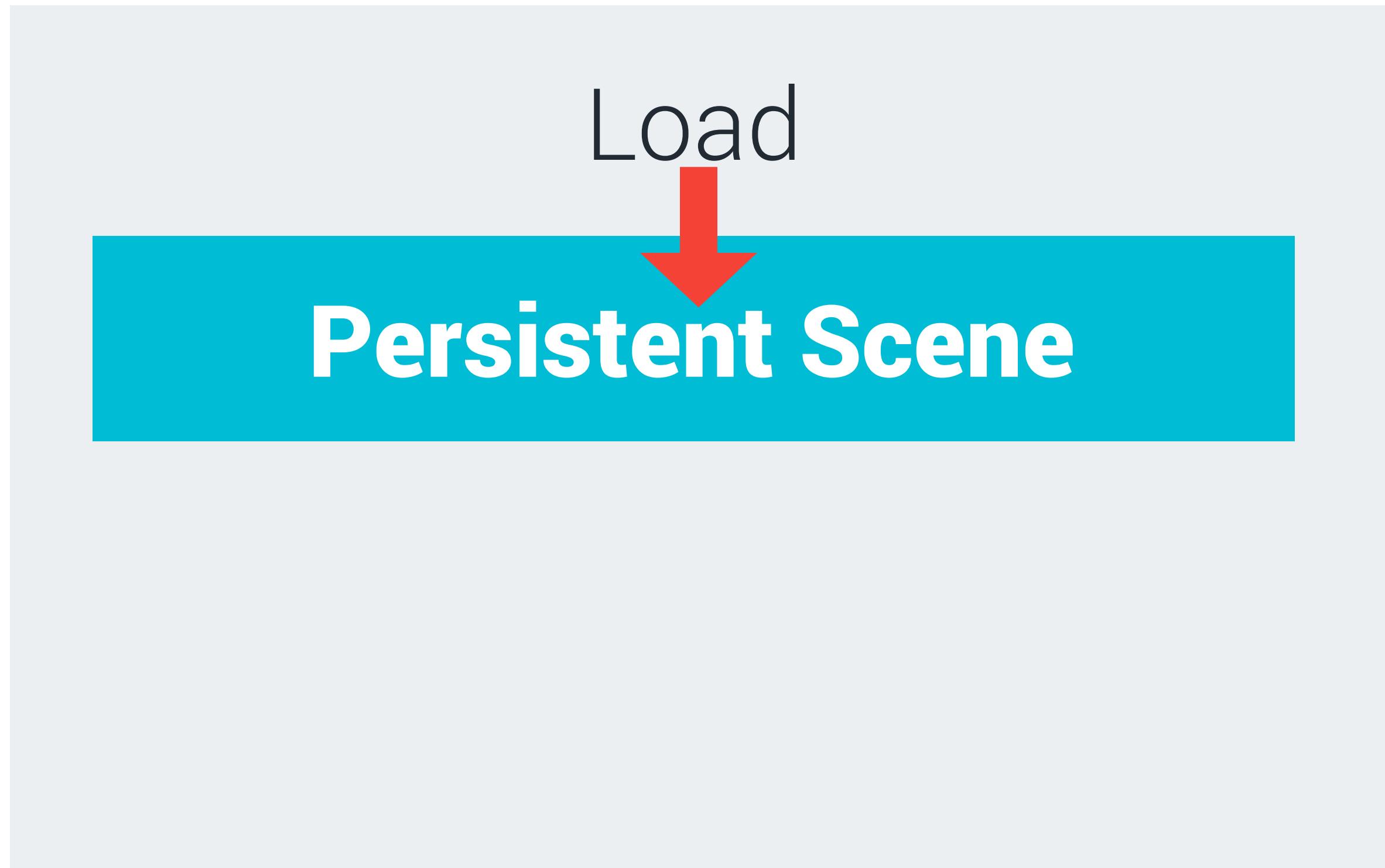
# Project Overview

Game Start

# Project Overview

Load

# Project Overview



# Project Overview

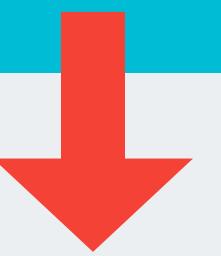
**Persistent Scene**

**Scene 1**

**Scene 2**

# Project Overview

**Persistent Scene**

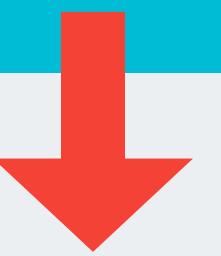


**Scene 1**

**Scene 2**

# Project Overview

**Persistent Scene**



Load Level Additive

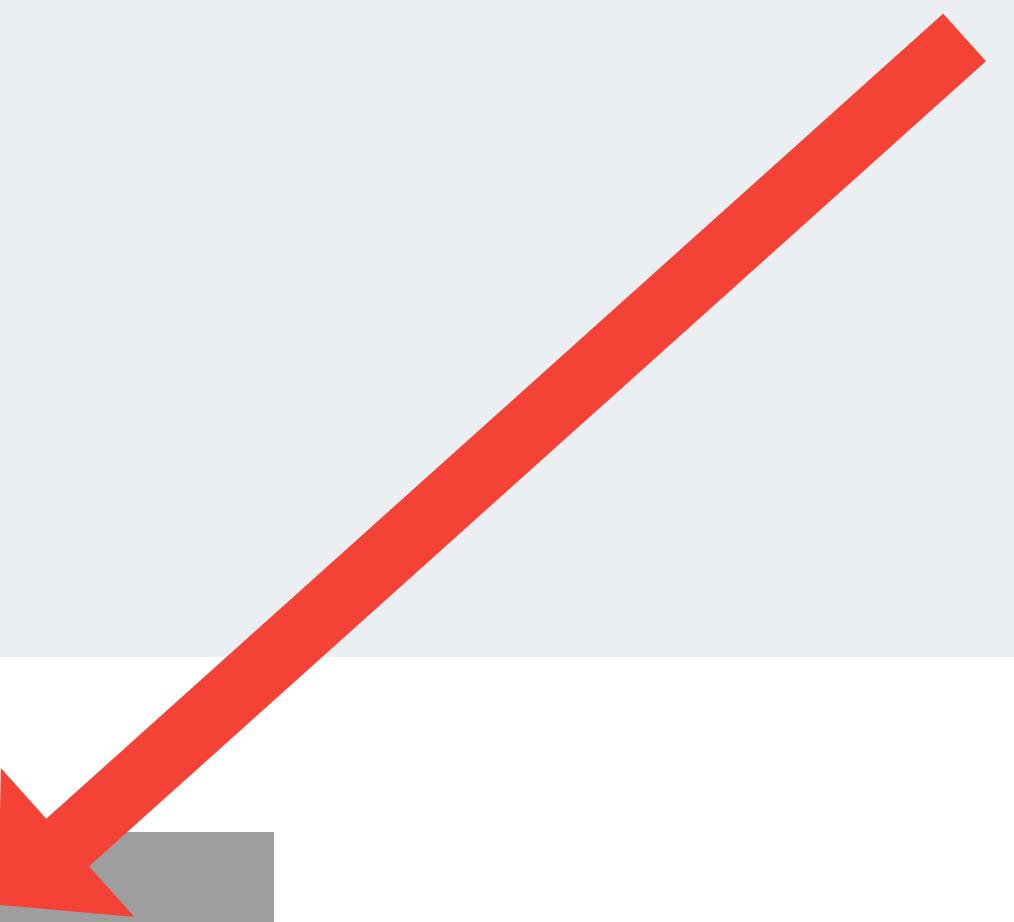
**Scene 1**

**Scene 2**

# Project Overview

## Persistent Scene

Load Level Additive



**Scene 1**

**Scene 2**

# Project Overview

## Persistent Scene

Load Level Additive



Scene 1

Scene 2

# Project Overview

## Persistent Scene

Set Active Scene

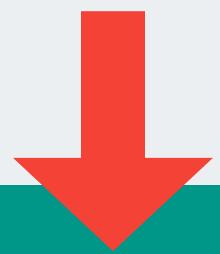
**Scene 1**

**Scene 2**

# Project Overview

## Persistent Scene

Set Active Scene



**Scene 1**

**Scene 2**

# Project Overview

**Persistent Scene**

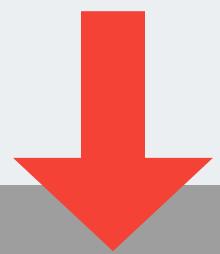
**Scene 1**

**Scene 2**

# Project Overview

## Persistent Scene

Unload Scene



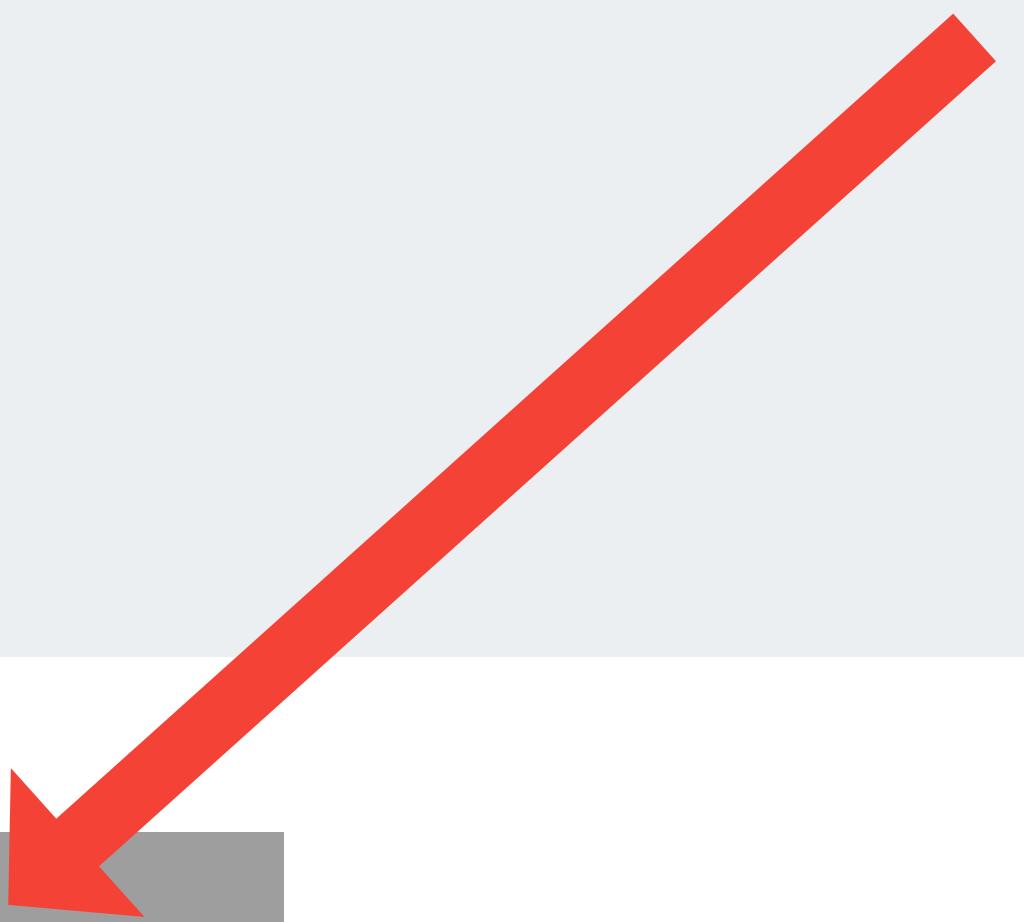
Scene 1

Scene 2

# Project Overview

## Persistent Scene

Unload Scene



**Scene 1**

**Scene 2**

# Project Overview

**Persistent Scene**

**Scene 1**

**Scene 2**

# Project Overview

## Persistent Scene

Load Level Additive

**Scene 1**

**Scene 2**

# Project Overview

## Persistent Scene

Load Level Additive

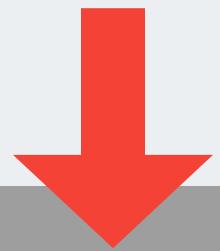
Scene 1

Scene 2

# Project Overview

## Persistent Scene

Load Level Additive



Scene 2

Scene 1

# Project Overview

**Persistent Scene**

Set Active Scene

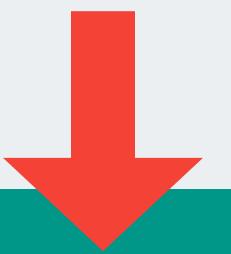
**Scene 2**

**Scene 1**

# Project Overview

## Persistent Scene

Set Active Scene



**Scene 2**

**Scene 1**

# Phase 01

## *The Player*

*Download the Asset Store package:*

**1/6 - Adventure Tutorial - The Player**



# *The Brief*



## *Brief*

- Create a **click to move** humanoid character for an adventure game



## *Brief*

- When the user **clicks on the ground**, the character must **move to that location**

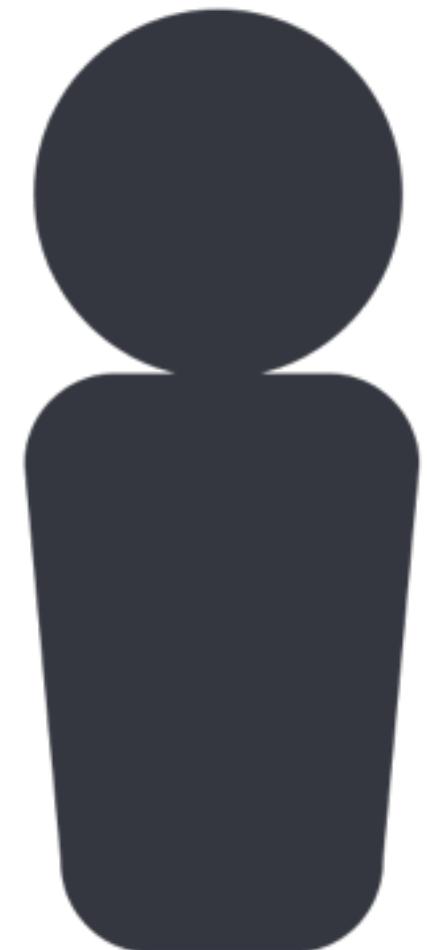


## Brief

- **Interactable** objects in the scene will be provided to our team for our character to interact with

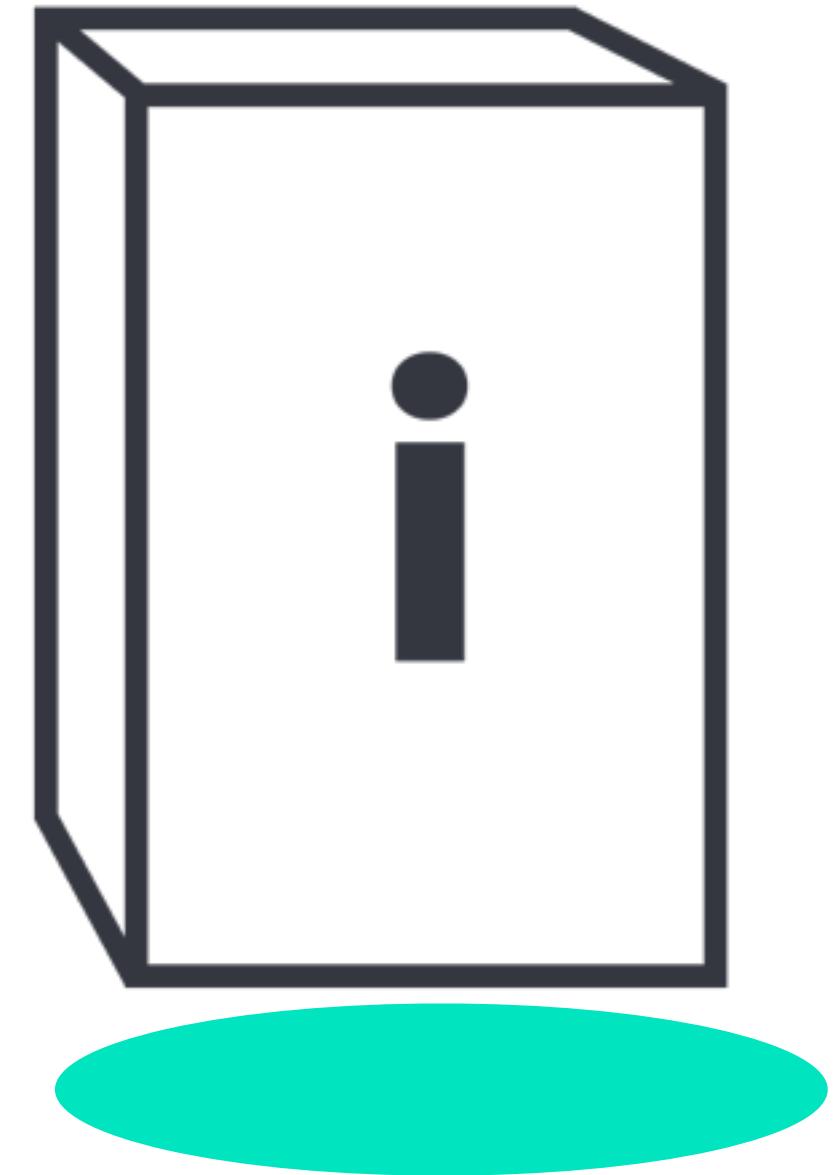


*We will get an opportunity to build parts of these later today!*



## Brief

- When an **Interactable** is clicked on, the character should approach the **interactionLocation** of the **Interactable**



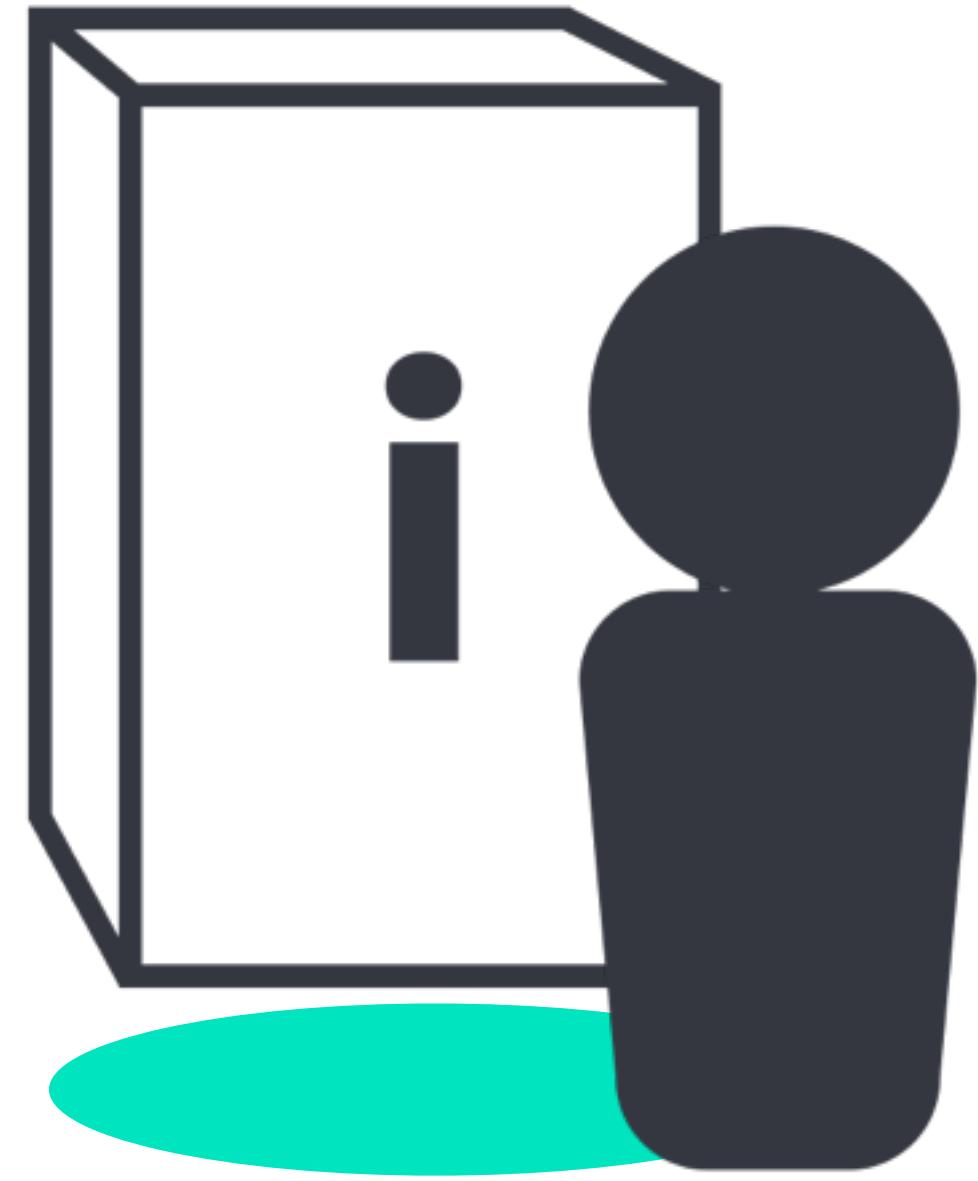
*InteractionLocation*



***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**

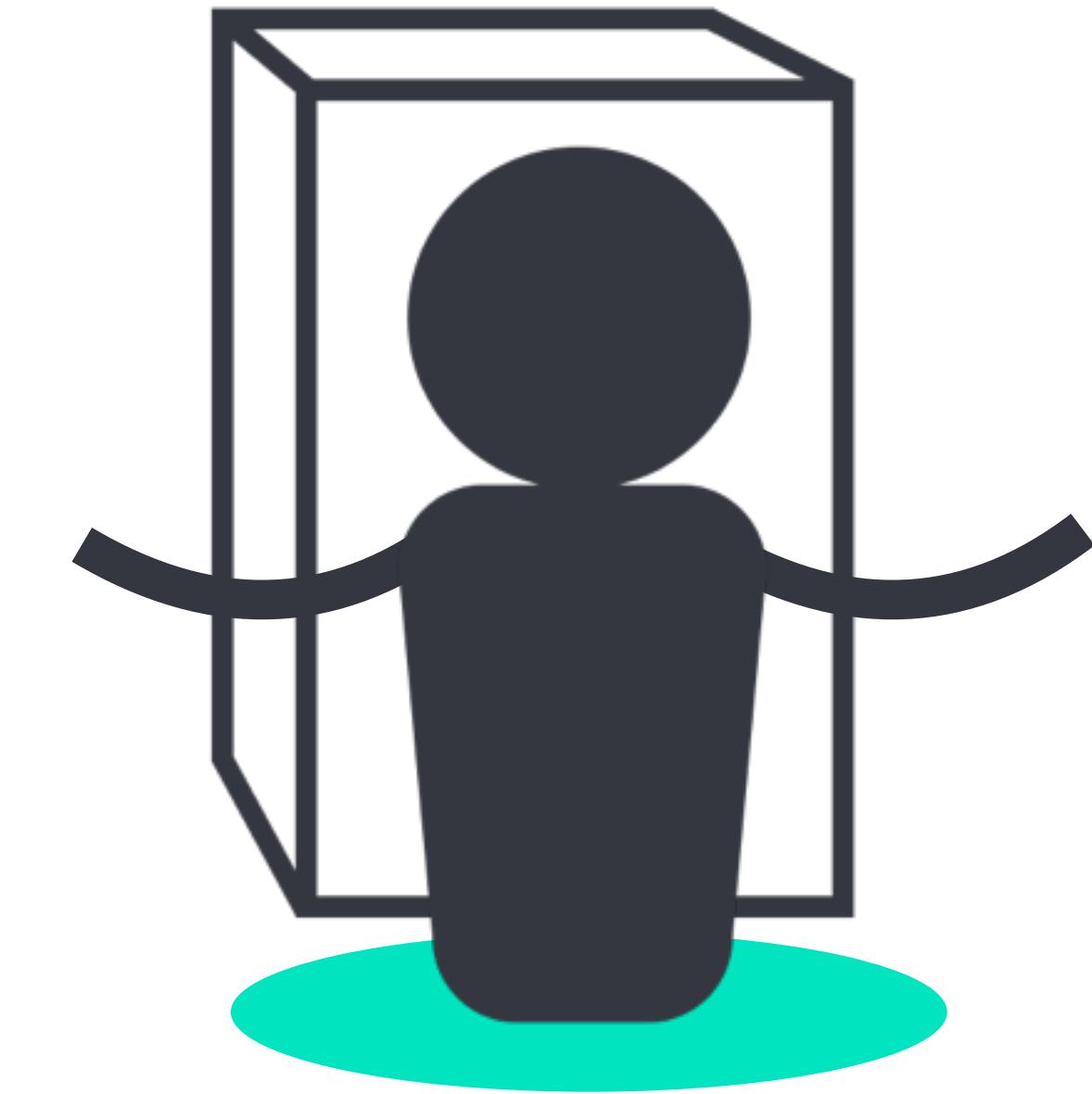
## Interact



**InteractionLocation**

- 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*

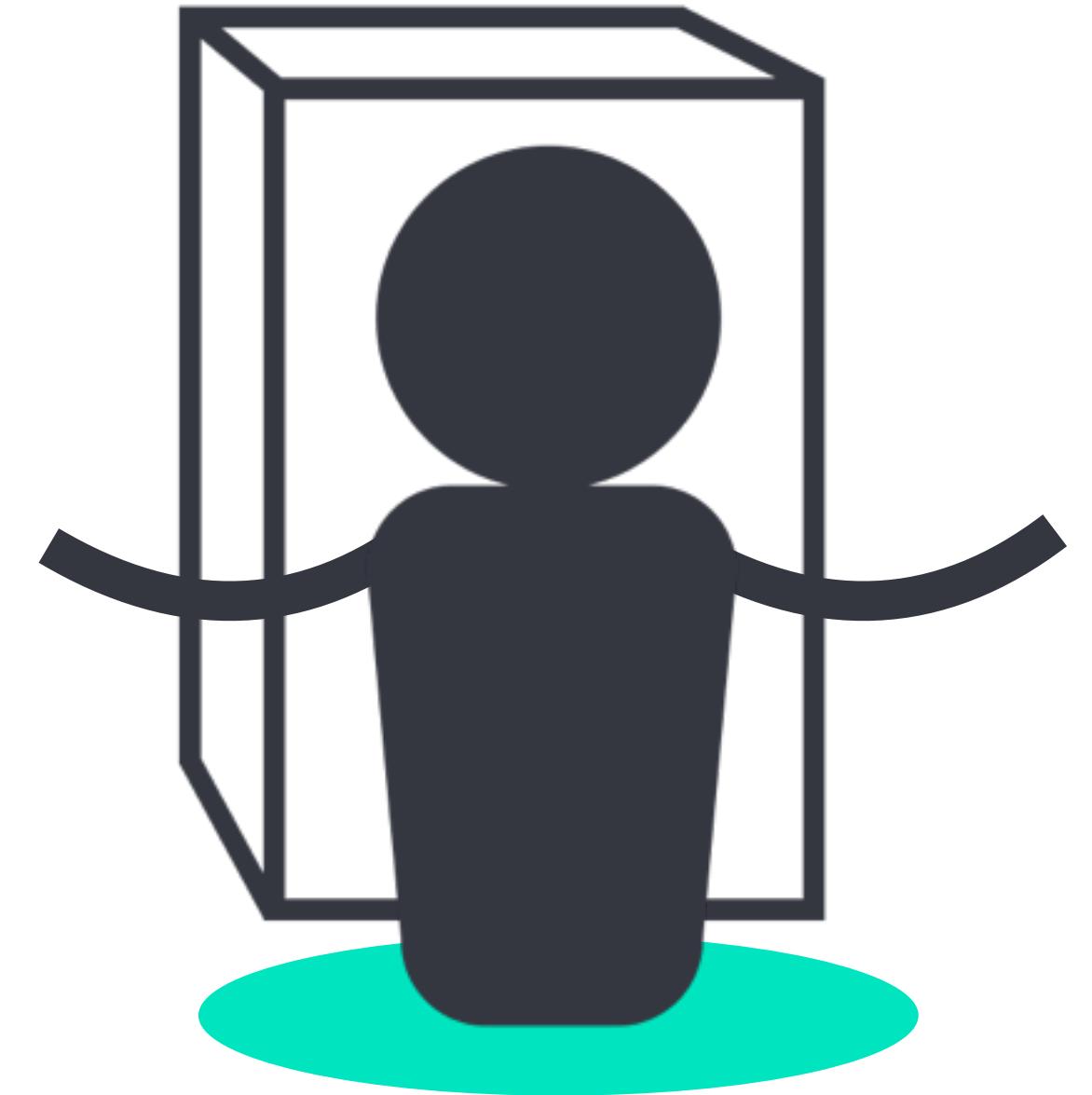
## Interact



*InteractionLocation*

## *Brief*

- The character cannot be allowed to move while these animations are playing

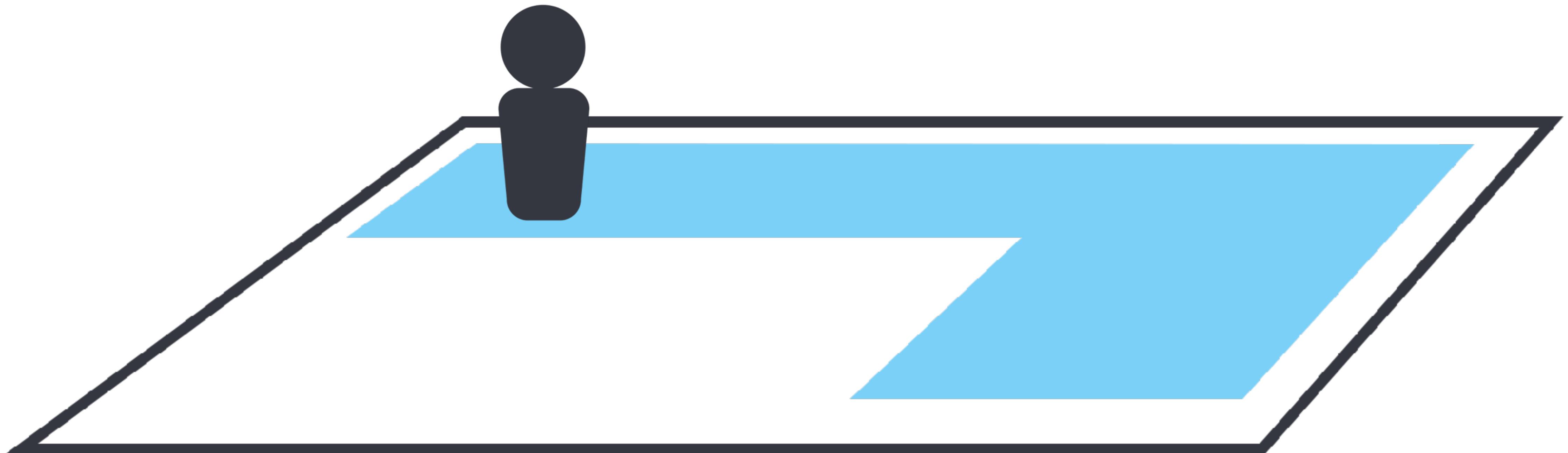


*InteractionLocation*

# *The Approach*

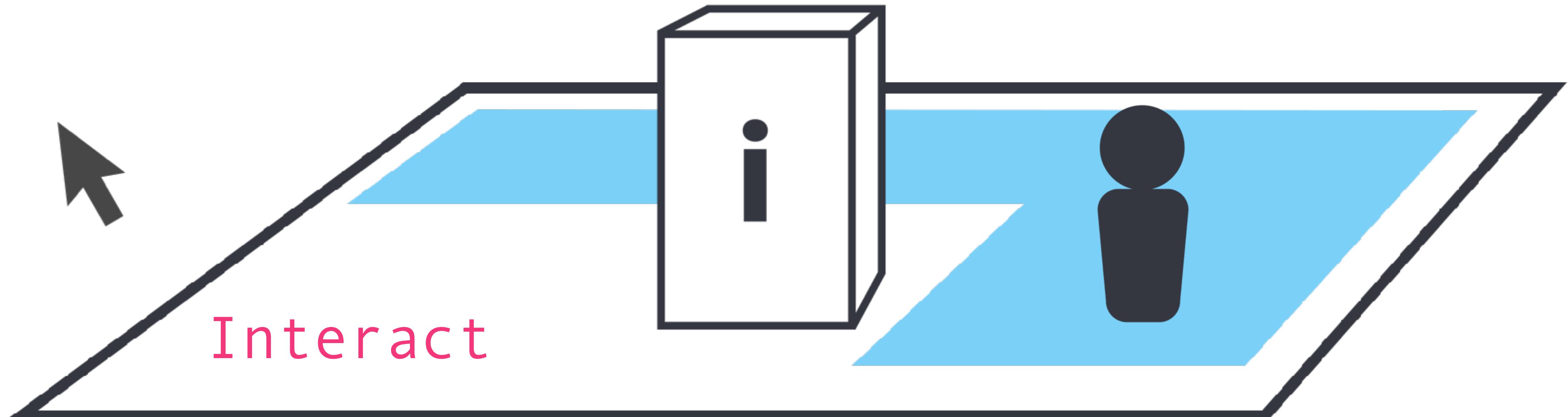
## *Approach*

- **NavMesh** to define the walkable areas in the levels



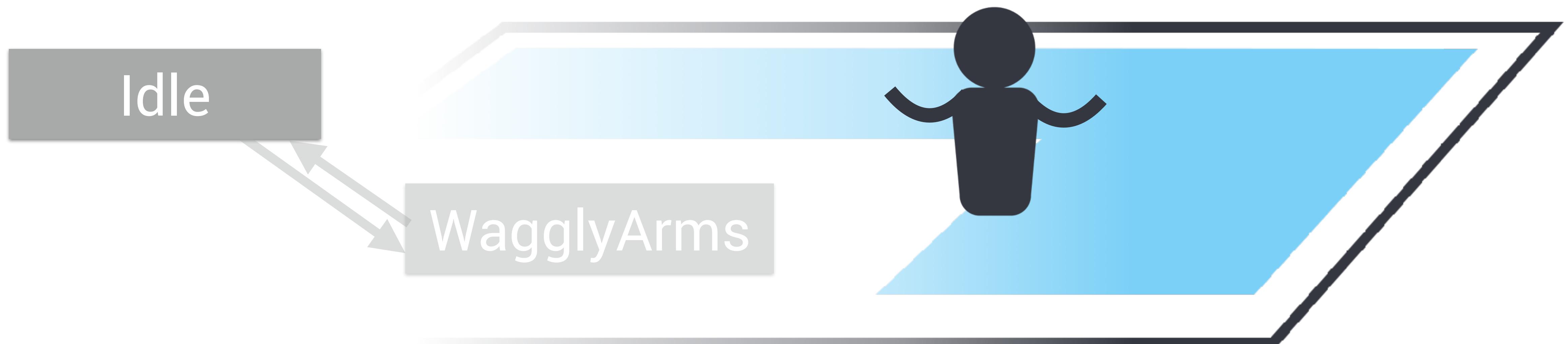
## *Approach*

- **Event Systems** to detect and handle user input and scripted interaction



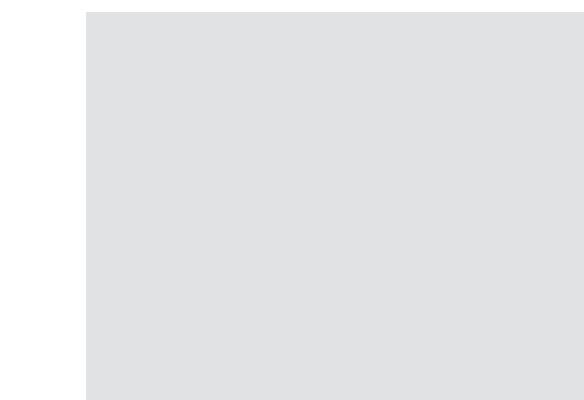
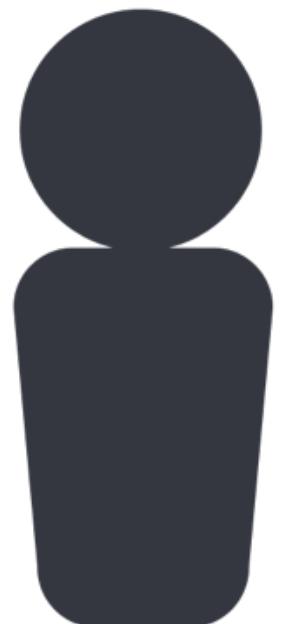
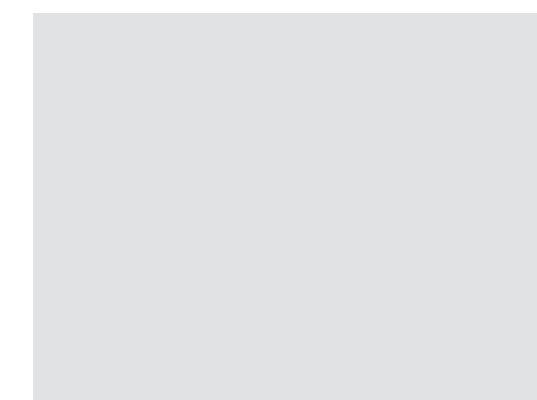
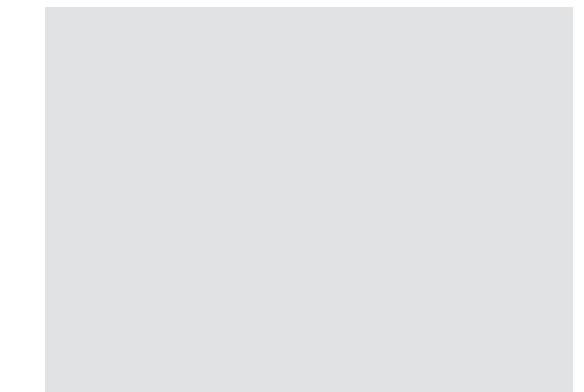
## *Approach*

- **Animator** state machine to control and play all of the character animations; including idle, walking and interaction



## *Approach*

- **Prefab system** to save the character so it can be easily added and used in any scene in the game



# *The Steps*



1. With the correct Asset Store package imported:

## ***1/6 - Adventure Tutorial - The Player***

2. Navigate to the **Project** window
3. Expand the **Scenes** folder
4. Open the **SecurityRoom** scene

1. Navigate to the **Hierarchy** window
2. Expand the **SecurityRoom** GameObject and select the **SecurityRoomEnvironment** GameObject
3. Check the **Static** checkbox
4. Select **Yes, change children**

1. **Expand the SecurityRoomEnvironment**
2. **Multi-Select** the children called:
  - **BlackUnlit**
  - **FloorLightGlow**
  - **HologramLight**
  - **HologramLight02**
  - **SecurityGateBeams**
3. **Uncheck** the **Static** checkbox

1. Open the **Navigation** window
2. Select the **Bake** panel
3. Set the **Max Slope** to **1**
4. Set the **Step Height** to **0.2**
5. Under ‘Advanced’ set **Height Mesh** to **true**
6. Press the **Bake** button

1. Navigate to the **Inspector** window
2. ***Save the scene***

The Player

Phase 1/6

## *Event Systems*

## The Event System

- We need to be able to interact with our environment
- An event system has 3 requirements. Something to -
  1. Send events
  2. Receive events
  3. Manage events

## The Event System

### 1. Send events

Physics Raycaster component

### 2. Receive events

Colliders & Event Triggers

### 3. Manage events

The EventSystem

## The Event System

### 1. Send events

*Physics Raycaster component*

### 2. Receive events

Colliders & Event Triggers

### 3. Manage events

The EventSystem

## The Event System

1. Send events

Physics Raycaster component

2. Receive events

Colliders & Event Triggers

3. Manage events

The EventSystem

## The Event System

1. Send events

Physics Raycaster component

2. Receive events

*Colliders & Event Triggers*

3. Manage events

The EventSystem

## The Event System

1. Send events

Physics Raycaster component

2. Receive events

Colliders & Event Triggers

3. Manage events

The EventSystem

## The Event System

1. Send events

Physics Raycaster component

2. Receive events

Colliders & Event Triggers

3. Manage events

*The EventSystem*

1. Navigate to the Hierarchy
2. ***Create > UI > Event System***
3. ***Add an AudioListener component***
4. ***Save the scene***

1. **Expand the SecurityRoom and CameraRig GameObjects**
2. **Select the Camera GameObject**
3. **Add a Physics Raycaster component.**
  - Make sure this is *not* a *Physics 2D Raycaster*

1. **Select the SecurityRoom GameObject**
2. **Add a MeshCollider component**
3. **Set the Mesh property to SecurityRoomMeshCollider**

1. Add an **EventTrigger** component
2. Click **Add New Event Type** and select **Pointer Click**
3. Click the + button to add an event
4. Leave the event **empty!**
5. **Save the scene**

The Player

Phase 1/6

## *Animator State Machine*

1. Navigate to the **Project** window
2. With the **Animators > Player** folder selected, **create an Animator Controller**
3. **Name it ClickToMove**

1. Open the Animator window
2. Select the Parameters panel
3. To create a new Parameter, select the + dropdown menu
4. Create a float parameter and name it *Speed*

## 1. Create 4 trigger parameters and name them:

- **AttemptTake**
- **HighTake**
- **MedTake**
- **LowTake**

1. Navigate to the **Layout** area
2. Right-click on the background and select:  
***Create State > From New Blend Tree***
3. Select the new **Blend Tree**
4. In the **Inspector** window, name the state ***Walking***
5. Double click on the state to edit the blend tree

1. Click on **BlendTree**
2. Rename **BlendTree** to ***WalkingBlendTree***
3. To make a new **Motion Field**, click the **+** button and select **Add Motion Field**
4. Repeat this **2 more times** to make a total of **3 Motion Fields** in all

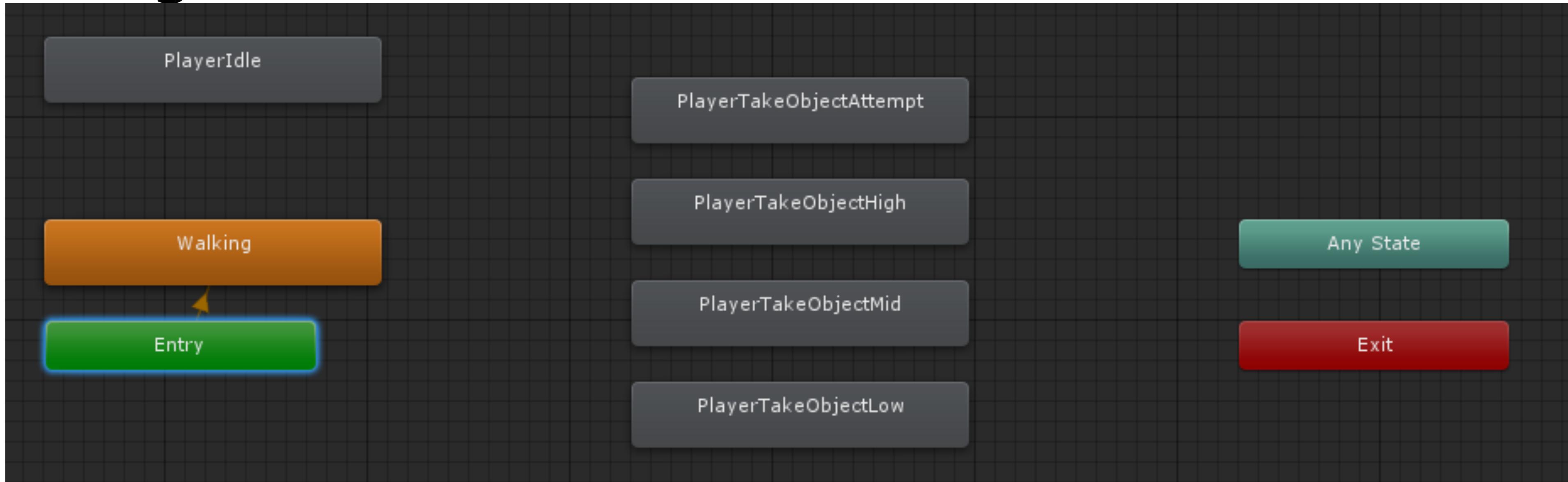
## 1. Using **Circle Select** find & assign:

- **PlayerIdleBlender**
- **PlayerWalk**
- **PlayerRun**

1. **Uncheck *Automate Thresholds***
2. **Click on the **Compute Thresholds** dropdown menu**
3. **Select **Speed****
4. **Return to **Base Layer** using Breadcrumb**

1. Navigate to the **Project** window
2. Expand the ***Animations > Player*** folder

- 1. Select *Idle*, *TakeObjectAttempt* & *TakeObjects***
- 2. Drag these into Animator window**
- 3. Arrange Animations as shown:**



1. **Right click on Walking**

2. **Select *Make Transition***

3. **Left click on PlayerIdle**

*This makes a **Transition from Walking to PlayerIdle***

4. **Make a Transition from PlayerIdle back to Walking**

1. Make Transitions from Any State to each of the **PlayerTakeObject...** states:

- **Attempt**
- **High**
- **Mid**
- **Low**

2. Make Transitions from each of the **PlayerTakeObject...** states to **Walking**

1. Select the **Transition** from Walking to PlayerIdle
2. Uncheck **has exit time**
3. Under the **condition list** click the **+** button to add a condition and set the parameters as:
  - **Speed (*already set*)**
  - **Less**
  - **0.1**

1. Select the **Transition** from **PlayerIdle** to **Walking**
2. Uncheck ***has exit time***
3. Under the **condition list** click the **+** button to add a condition
  - **Speed (*already set*)**
  - **Greater (*already set*)**
  - **0.1**

1. Select each of the **Transitions** from **Any State** to each of the **PlayerTakeObject...** states
2. Add a new **condition** and set the **parameter** the appropriate **trigger** using the parameter dropdown:
  - **AttemptTake**
  - **HighTake**
  - **MedTake**
  - **LowTake**

1. Select **PlayerIdle**
2. Set the **tag** to **Locomotion**
3. Select **Walking**
4. Set the tag to **Locomotion**
5. Navigate to the Scene window
6. ***Save the scene***

# The Player

# Phase 1/6

*Player Prefab*

1. Navigate to the **Project** window
2. Expand the **Models** folder
3. Drag the **Player** model asset into the **Hierarchy**
4. Set the **Layer** on the **Player** GameObject to **Character**
5. Set **Position -0.7, 0.0, 3.5**
6. Set the **Rotation 0.0, 180, 0.0**

1. Using **Circle select** - set **Controller** to *ClickToMove*
2. Add a new component: **NavMeshAgent**
3. Change the **Speed** property to 2
4. Change the **Acceleration** property to 20
5. Change the **Stopping Distance** to 0.15

1. Select and Drag the Player GameObject to the Project window > Prefabs folder
2. Navigate to ***Scripts/Monobehaviours/Player*** folder
3. Create a new C# Script called ***PlayerMovement***
4. Drag the **PlayerMovement** script onto the **Player Prefab**
5. Open the **PlayerMovement** script for editing

*PlayerMovement script*

The Player

Phase 1/6

**Destination**

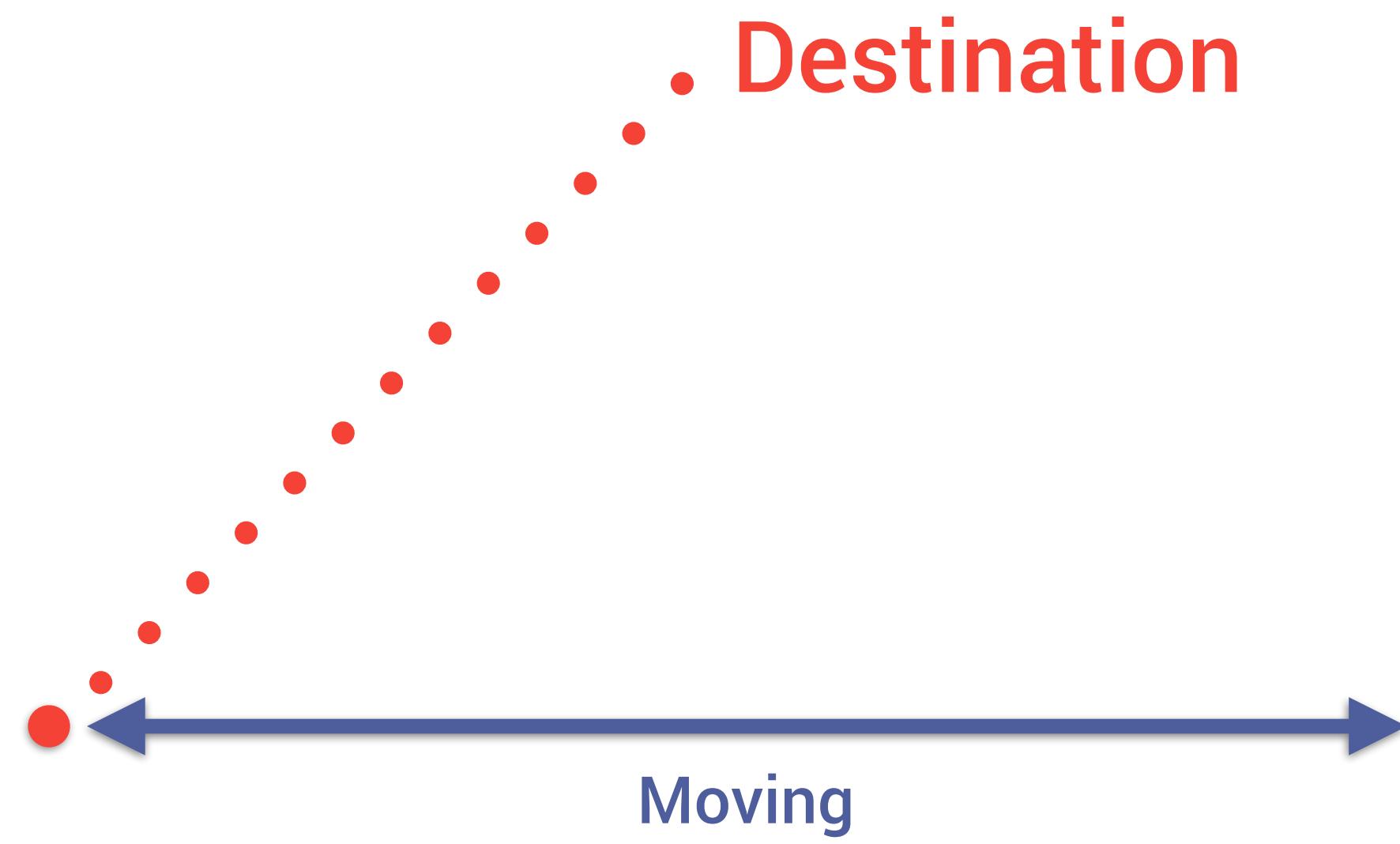
# The Player

Phase 1/6



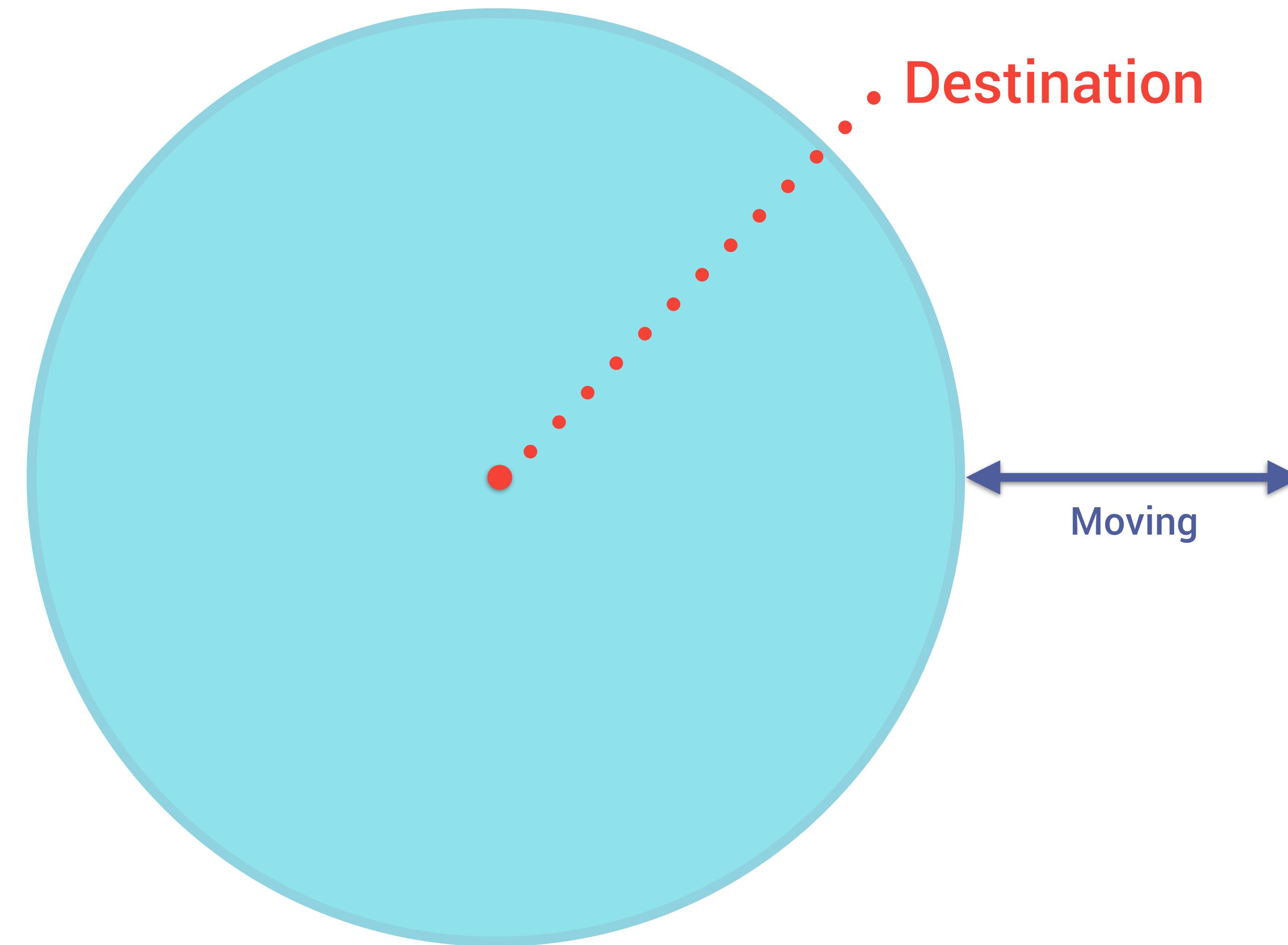
# The Player

Phase 1/6



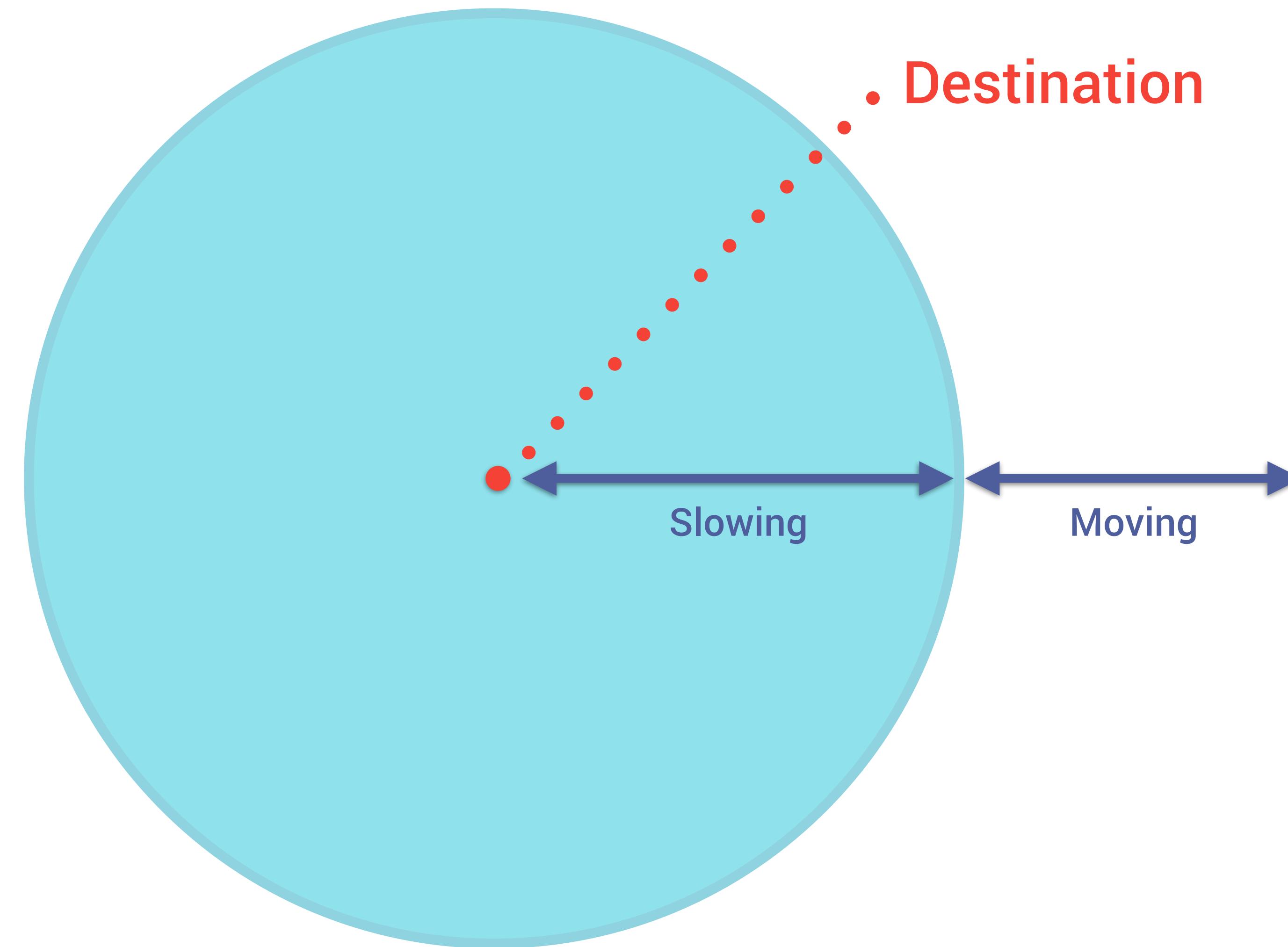
# The Player

Phase 1/6



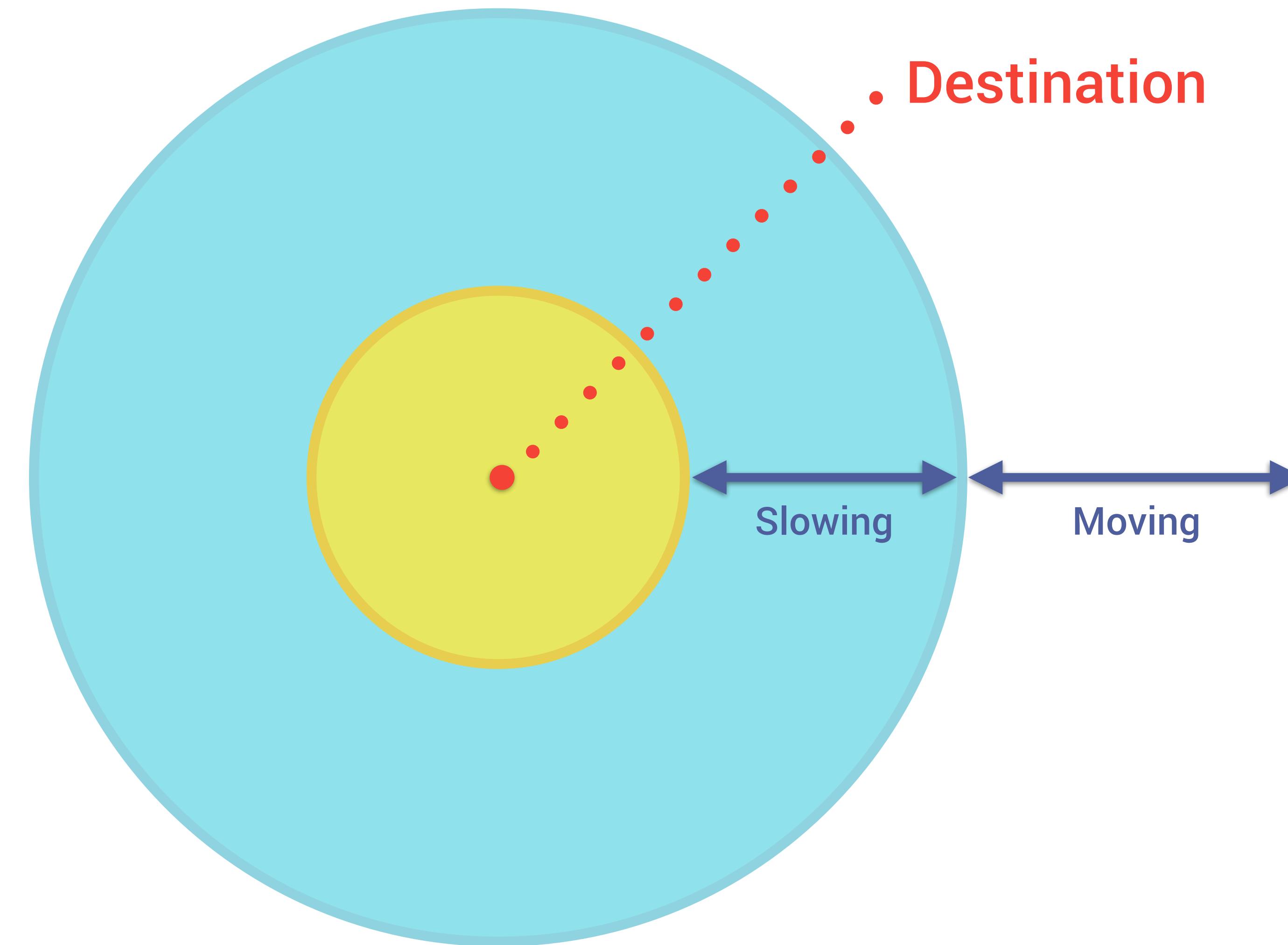
# The Player

Phase 1/6



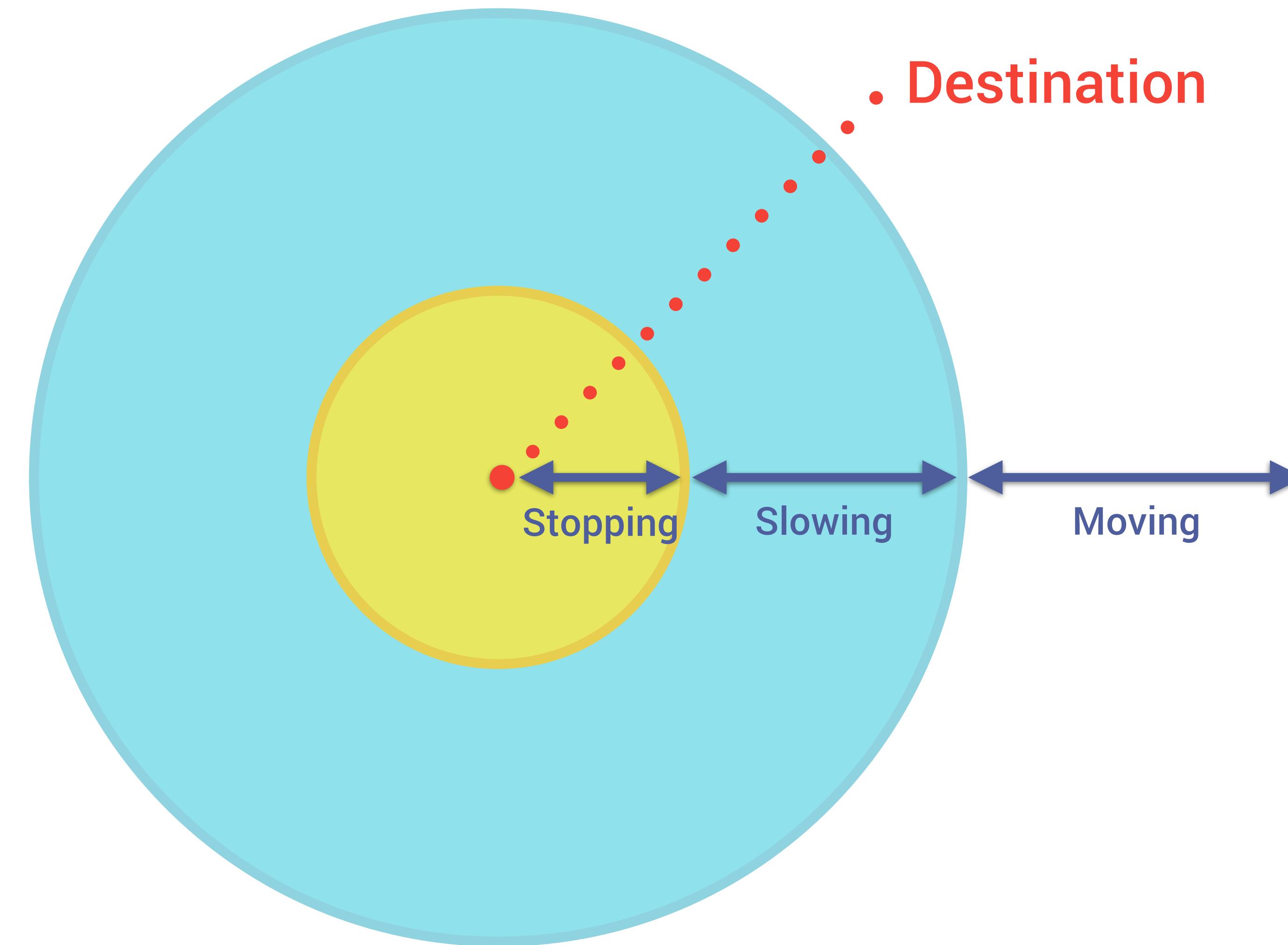
# The Player

Phase 1/6



# The Player

Phase 1/6



1. **Save the script**
2. **Return to the Unity Editor**

1. Select the **Player Prefab**
2. On the **PlayerMovement** component:
  - Setup the **reference** to the Player's **Animator**
  - Setup the **reference** to the Player's **NavMeshAgent**

1. Navigate to the Hierarchy
2. Select **SecurityRoom** GameObject
3. Find the Event Trigger
4. Drag the Player GameObject from the Hierarchy onto the **object** field of the Event Trigger
5. Select **PlayerMovement.OnGroundClick()**

**1. *Save the Scene***

**2. Test**

**3. *Exit Play Mode!***

## 1. Open the **PlayerMovement** script for **Editing**

1. **Save the script**
2. **Return to the Unity Editor**

## *Working with Interactables*

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

1. Select ***PictureInteractable***
2. Find the **Event Trigger** component
3. Drag the **Player** GameObject from the **Hierarchy** onto the **object** field of the **Event Trigger**
4. Select **PlayerMovement.OnInteractableClick()**
5. Drag the **Interactable** GameObject or the **Interactable** component below onto the **EventTrigger's Parameter** field

**1. *Save the Scene***

**2. Test**

**3. *Exit Play Mode!***

*End of*  
**Phase 01**



# Phase 02

# *Inventory*

*Download the Asset Store package:*  
**2/6 - Adventure Tutorial - Inventory**



# *The Brief*

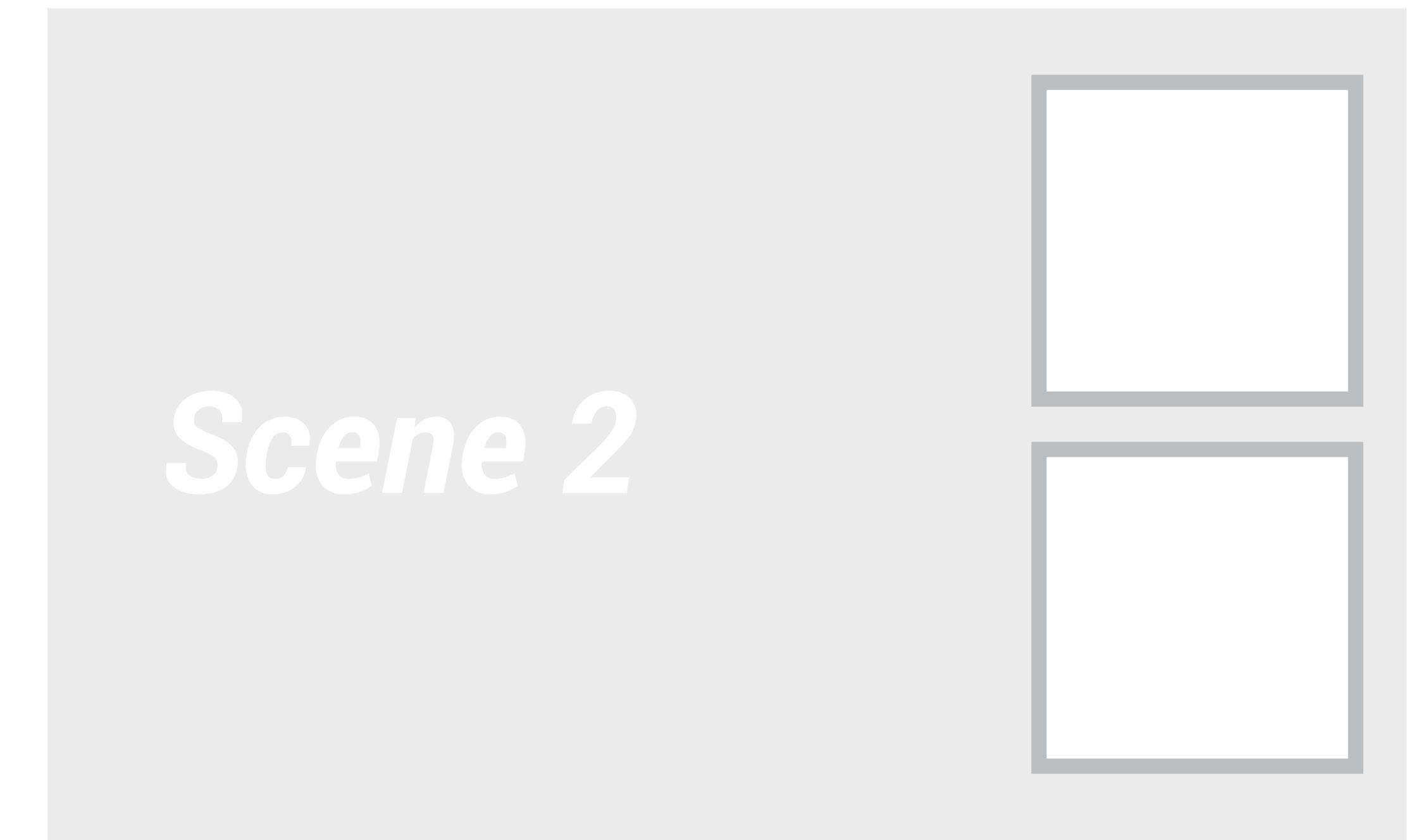


## Brief

- Create a **simple inventory system** with **persistent** content that is not lost during scene changes



*Scene 1*



*Scene 2*

## Brief

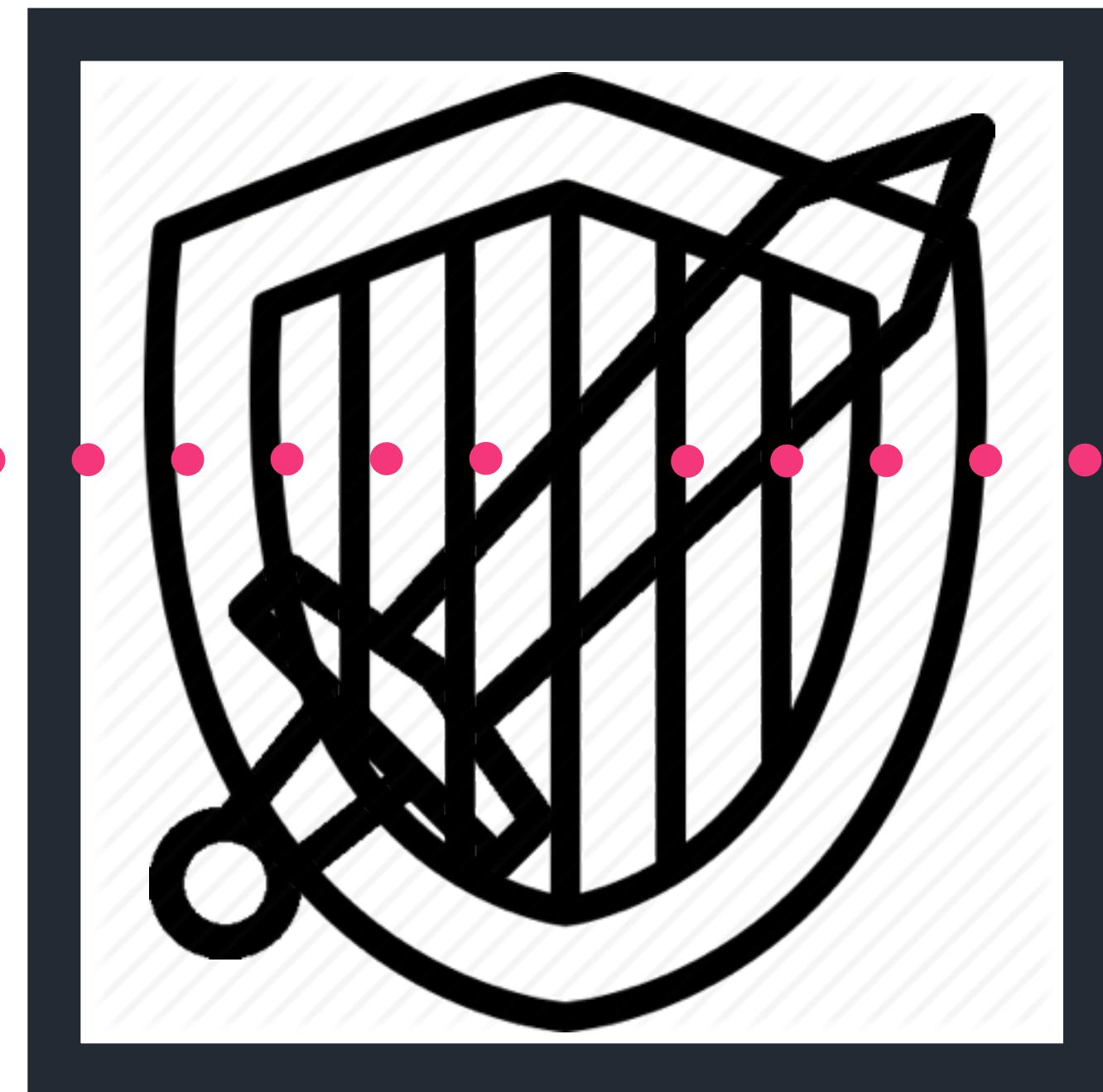
- **Inventory Items** should be simple but easily extensible if the design changes



## Brief

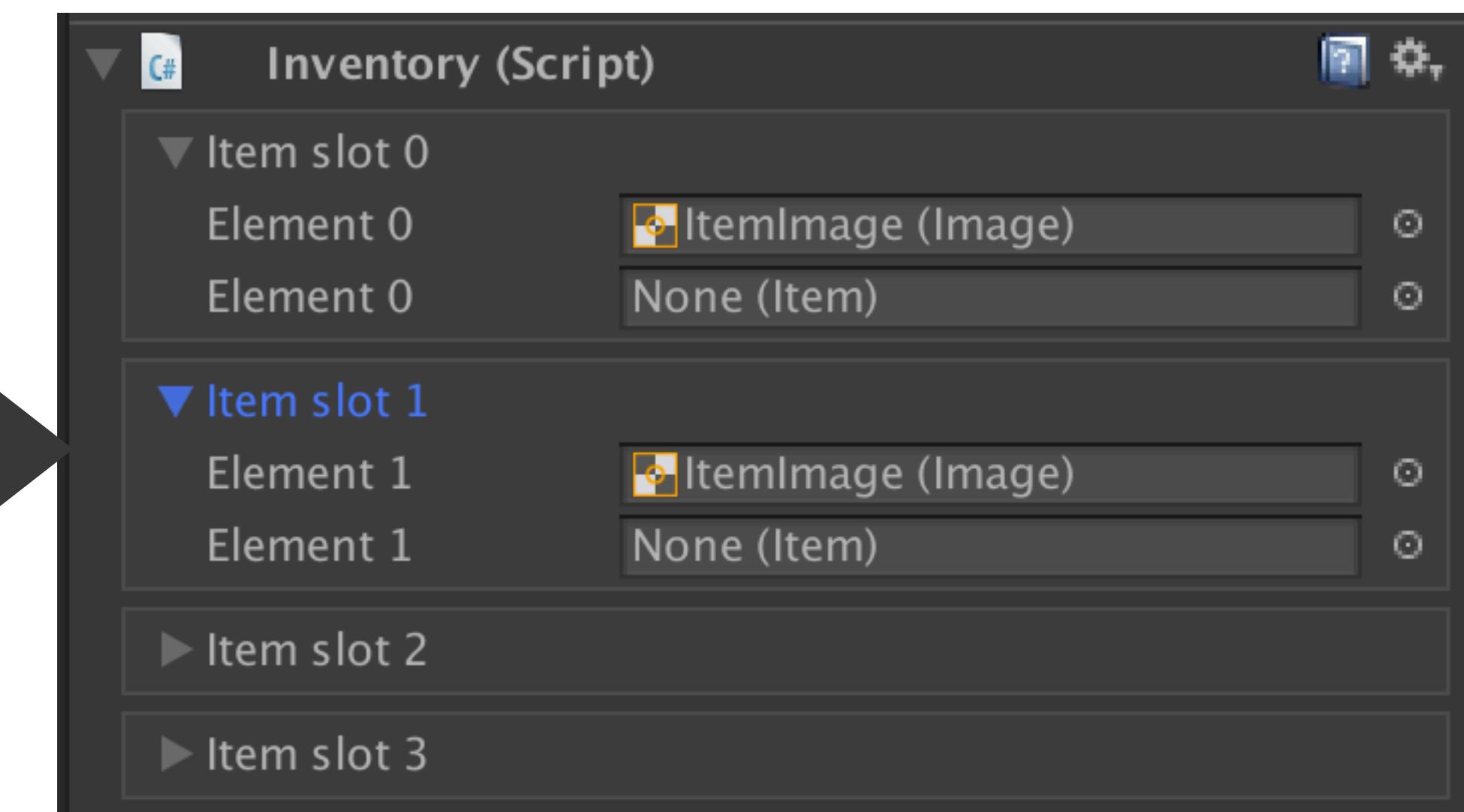
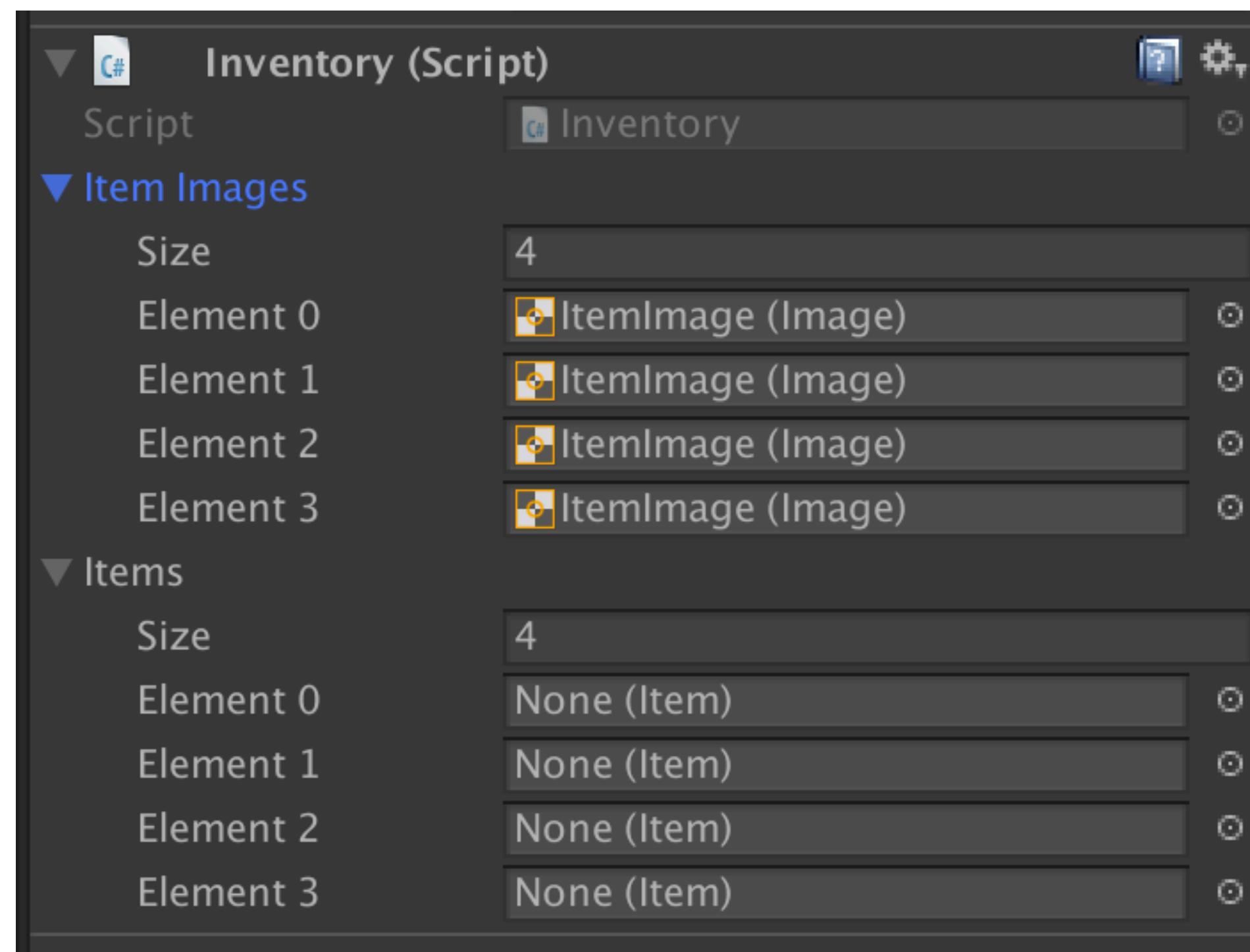
- The inventory should have two public functions:  
**AddItem & RemoveItem**

AddItem • • • • • RemoveItem



## Brief

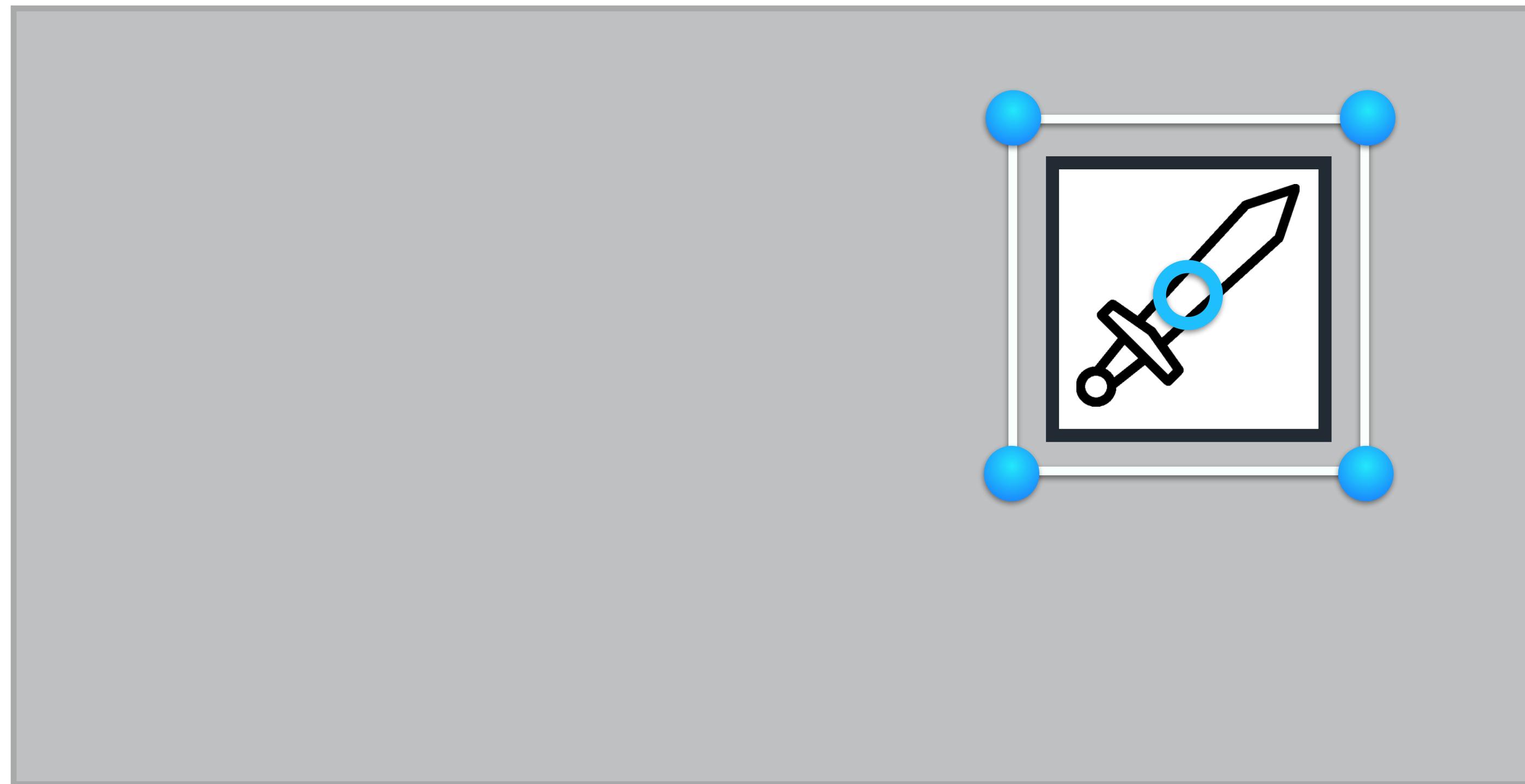
- Simplify and improve the workflow of the project in the **Inspector** with regards to the **Inventory** and its **Items**



# *The Approach*

## *Approach*

- Use the **UI system** to display the inventory to the user

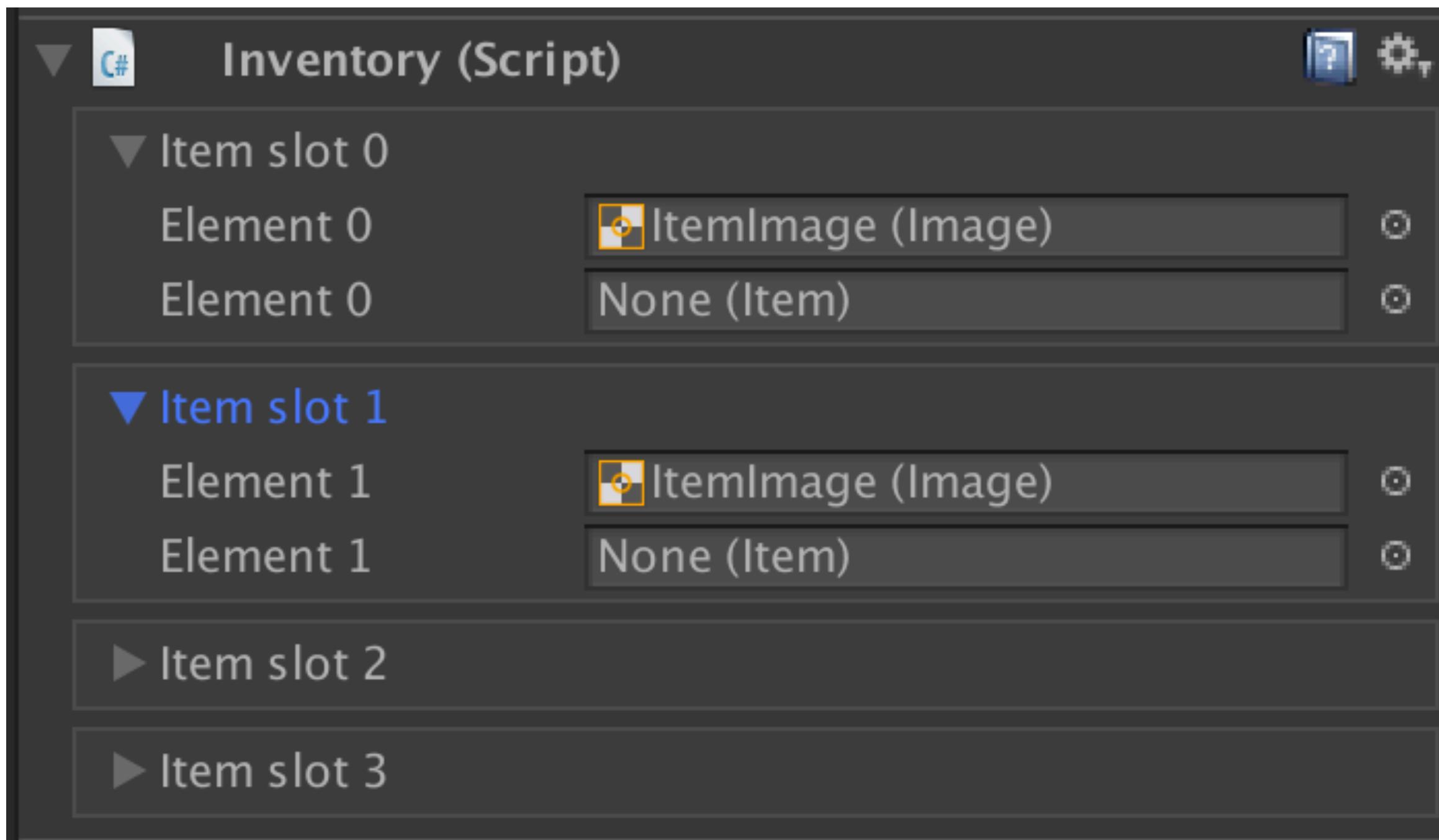


## *Approach*

- Use **ScriptableObjects** to make a simple **Item** class which defines every possible inventory item and can easily be extended and referenced by the **Inventory**

## Approach

- Create a **custom inspector** for the **Inventory** to improve the workflow of the project



# *The Steps*



1. Please import the Asset Store package:

***2/6 - Adventure Tutorial - Inventory***

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*

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*

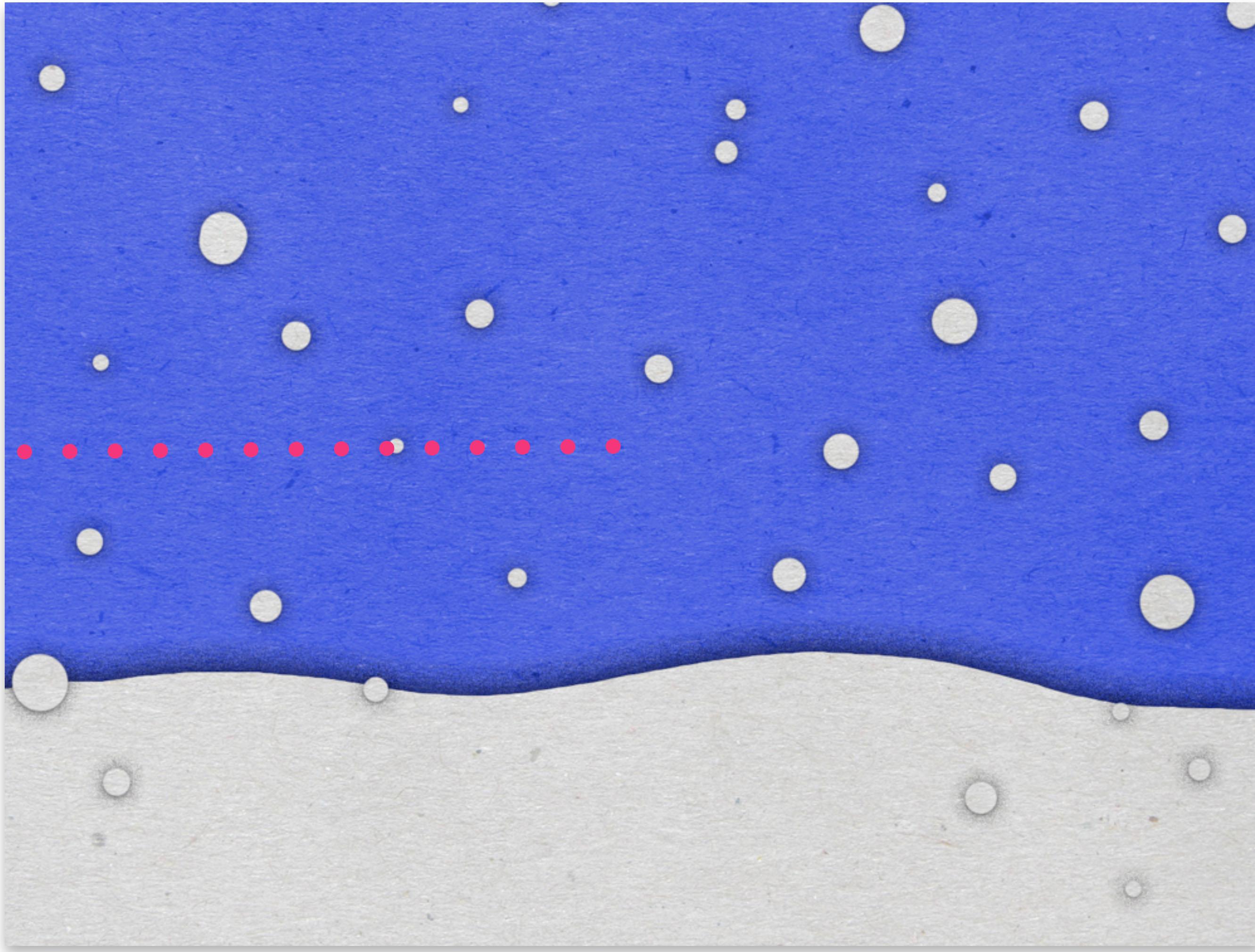
## Hierarchy

UI Canvas

## Hierarchy

UI Canvas

UI Object 01 ...

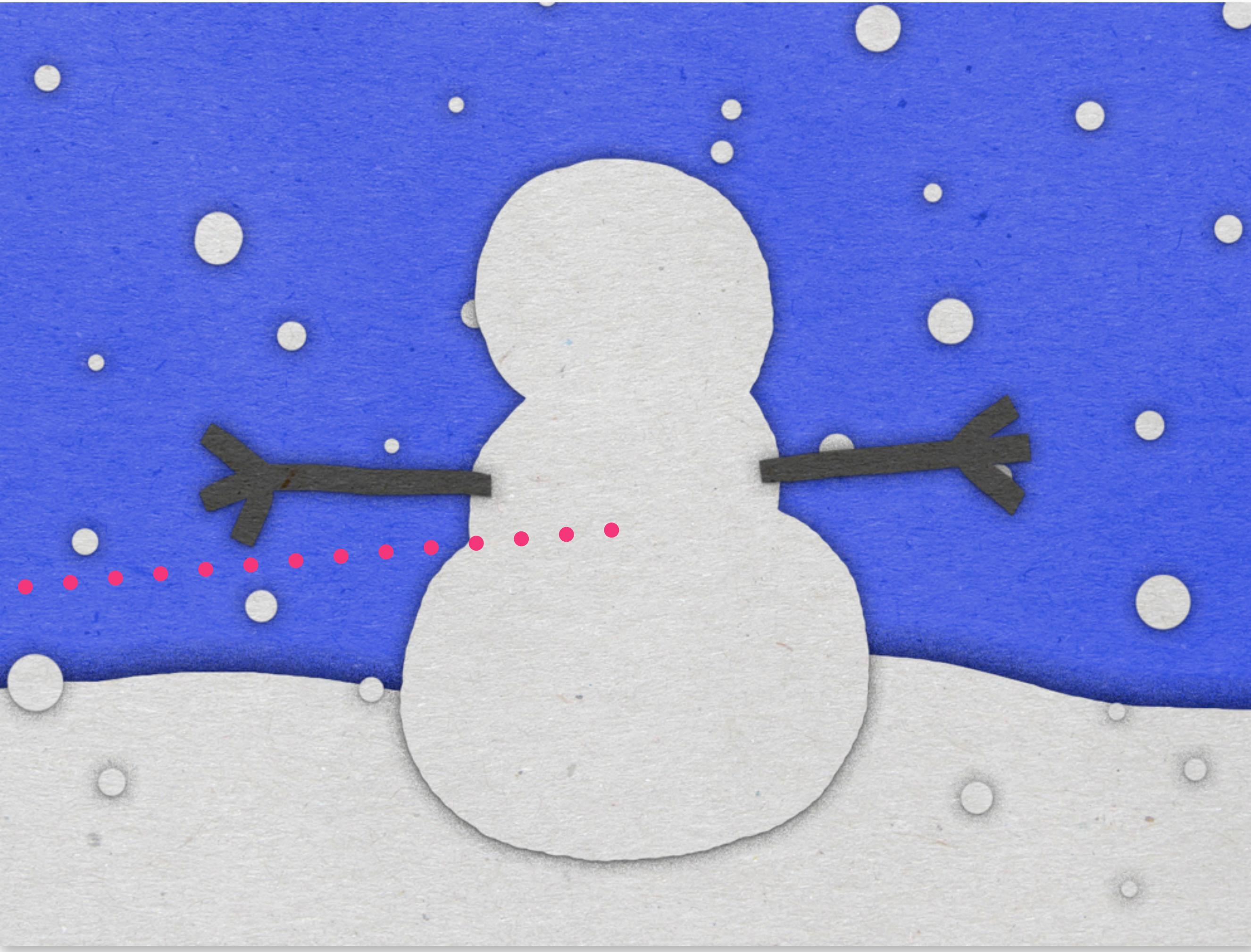
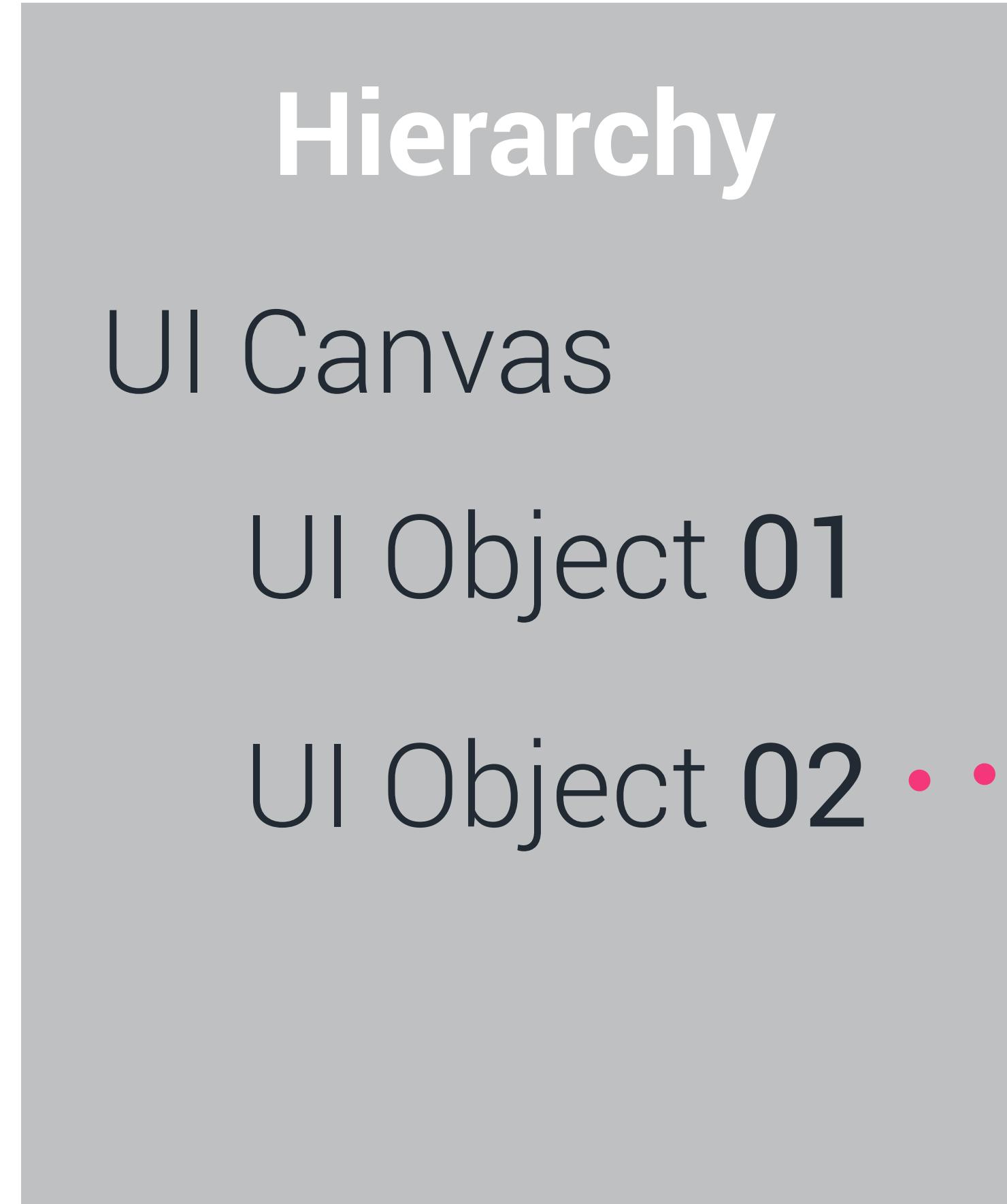


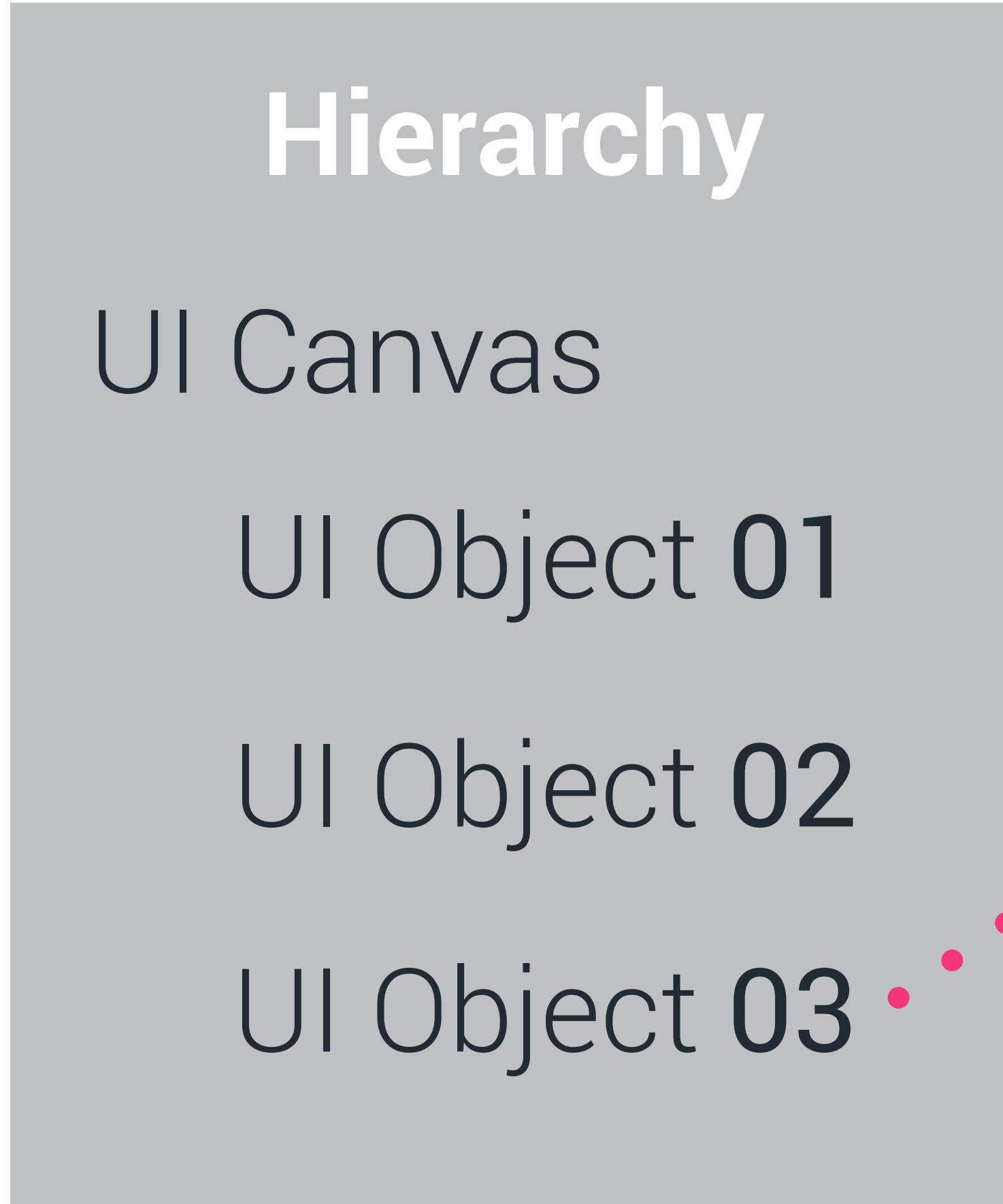
## Hierarchy

UI Canvas

UI Object 01

UI Object 02





### Hierarchy

UI Canvas

UI Object 01

UI Object 02

UI Object 03

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 FaderImage*

1. **Create a child of Inventory called ItemSlot**
2. As a **child of ItemSlot, Create a new *UI > Image* GameObject**
3. **Duplicate the Image GameObject**
4. **Name the first child Image *BackgroundImage***
5. **Name the second *ItemImage***

1. **Select the BackgroundImage GameObject**
2. **Set the Image component's *Source Image* to InventorySlotBG**
3. **Select the ItemImage GameObject**
4. **Disable the Image component**

1. Drag the **ItemSlot** GameObject into the **Prefabs** folder to make it a prefab
2. Return to the **Hierarchy** window
3. Duplicate the **ItemSlot** GameObject 3 times so there are 4 **ItemSlots** in total
4. Name them: **ItemSlot0**, **ItemSlot1**, **ItemSlot2** and **ItemSlot3**

- 1. Select the Inventory GameObject**
- 2. Add a Vertical Layout Group component**

1. Find the Inventory's Rect Transform component
2. Set the width to 135 and the height to 600
3. From the anchor selection dropdown,  
choose ***middle-right***
4. Set the position to **-95, 0, 0**

## 1. *Save the scene*

### *Item script*

## *Inventory script*

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

## ***1. Save the script and return to the editor***

1. **Navigate to *Scripts > ScriptableObjects > Interaction > Reactions > DelayedReactions* folder**
2. **Open the `LostItemReaction` script for editing**
3. **Uncomment all the commented-out code**
4. ***Save the script and return to the editor***

1. **Navigate to *Scripts > ScriptableObjects > Interaction > Reactions > DelayedReactions* folder**
2. **Open the PickedUpItemReaction script for editing**
3. **Uncomment all the commented-out code**
- 4. *Save the script and return to the editor***

## 1. Add an **Inventory** component to the **PersistentCanvas** GameObject

- **Note that the *Inventory* component has two separate and unassociated arrays**

## *Understanding the Custom Inspector*

Inventory

Phase 2/6

Runtime

Inventory

Phase 2/6

Runtime

**Object**

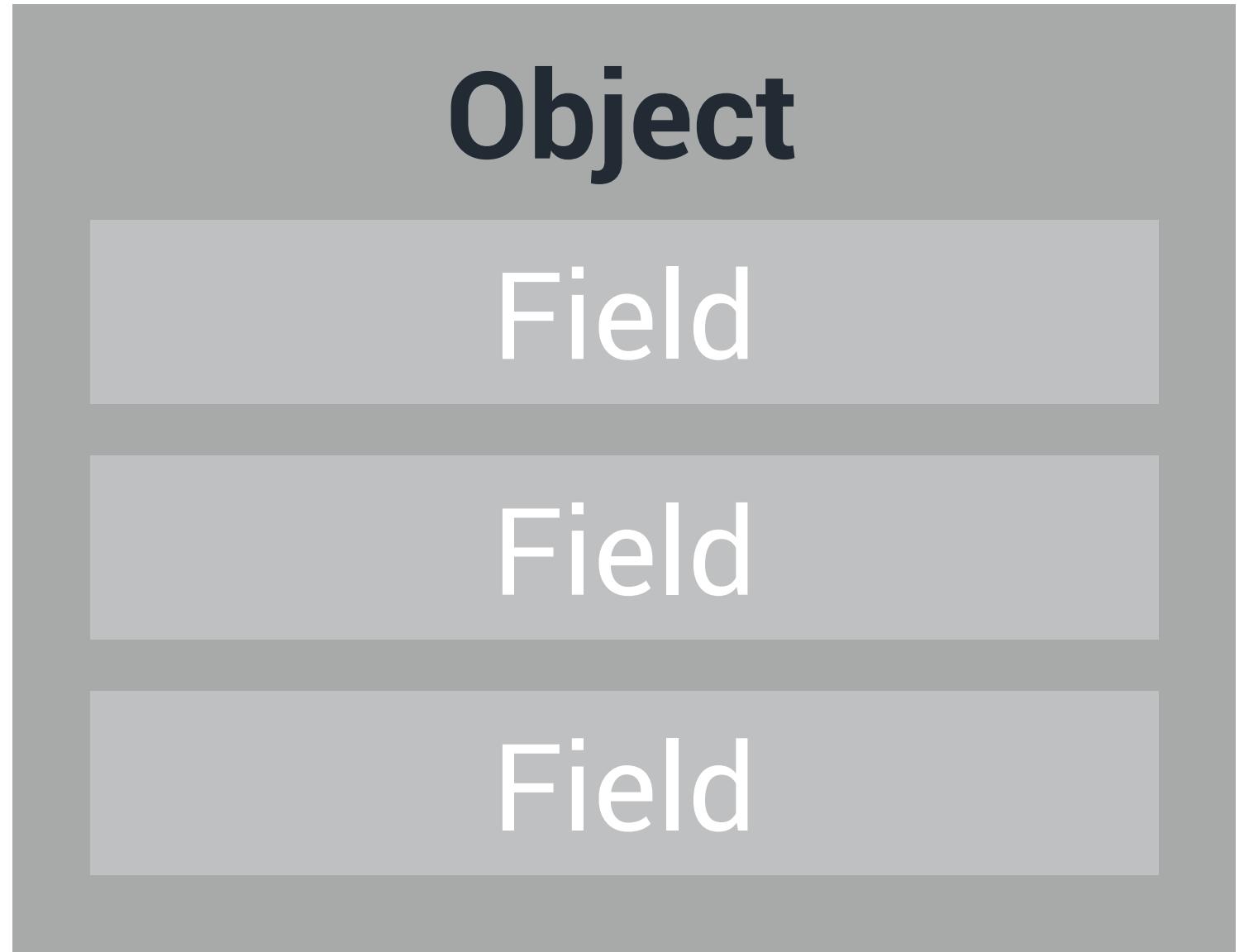
### Runtime

**Object**

Field

Field

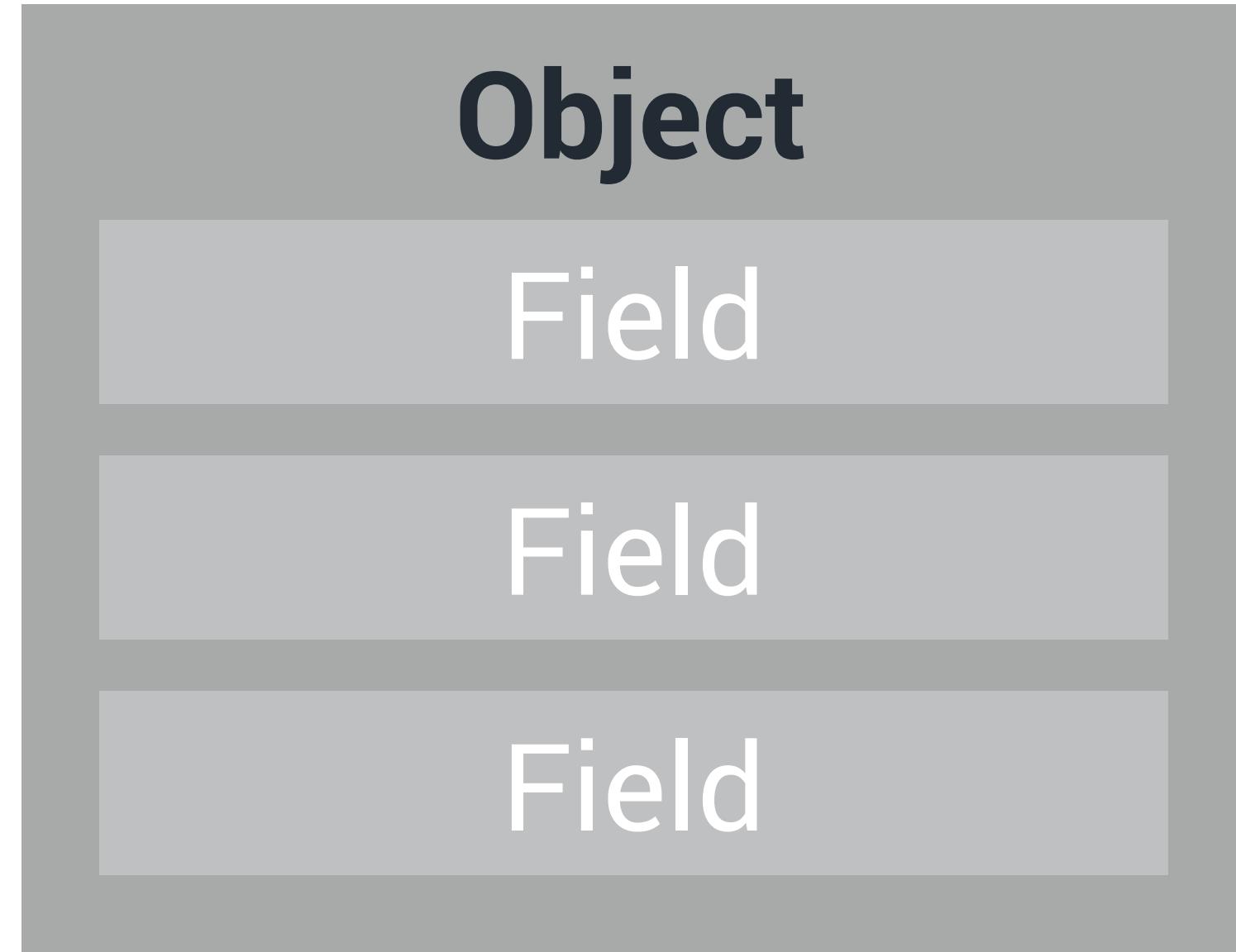
Field



# Inventory

Phase 2/6

Runtime



Edit Time

# Inventory

Phase 2/6

Runtime

**Object**

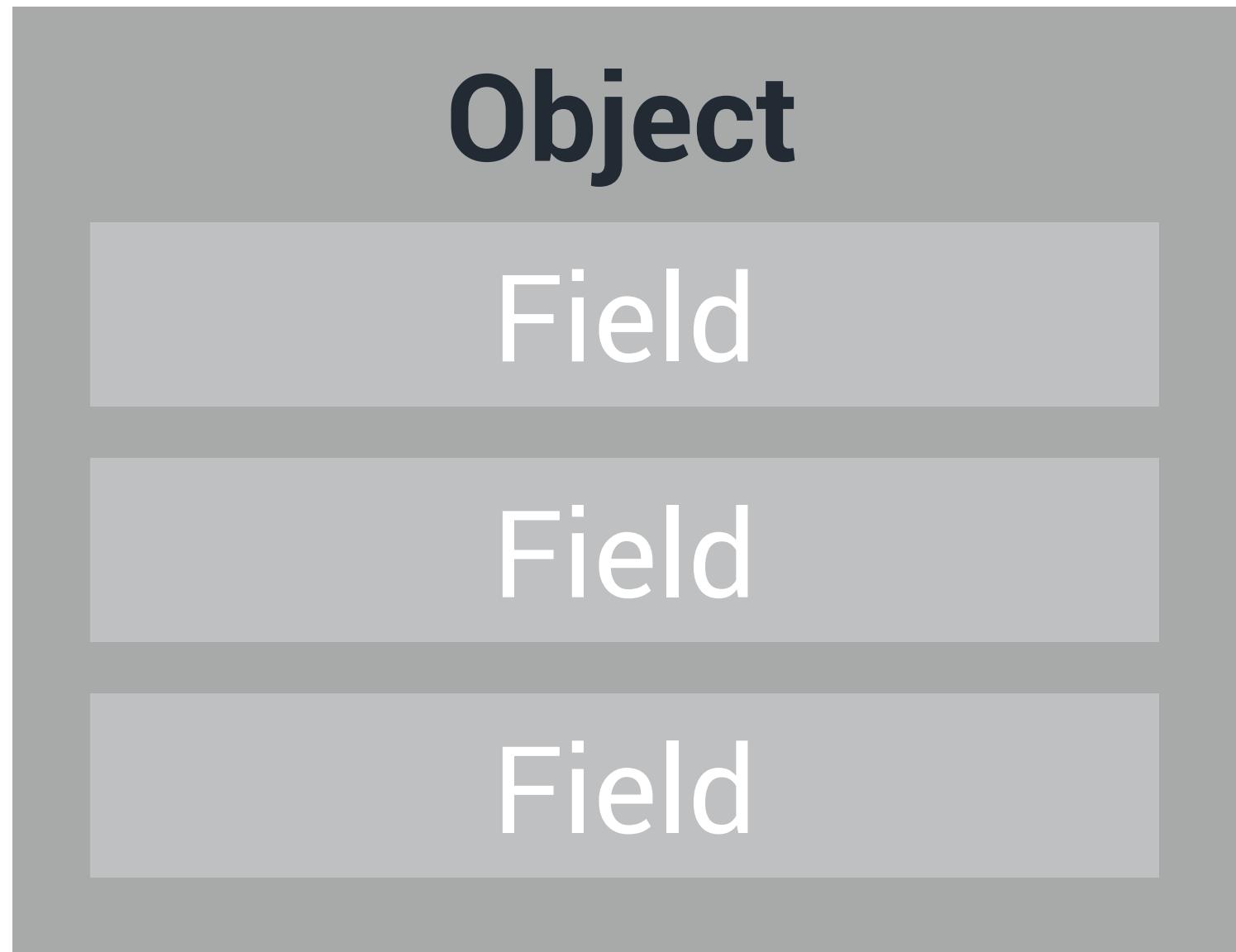
Field

Field

Field

Edit Time

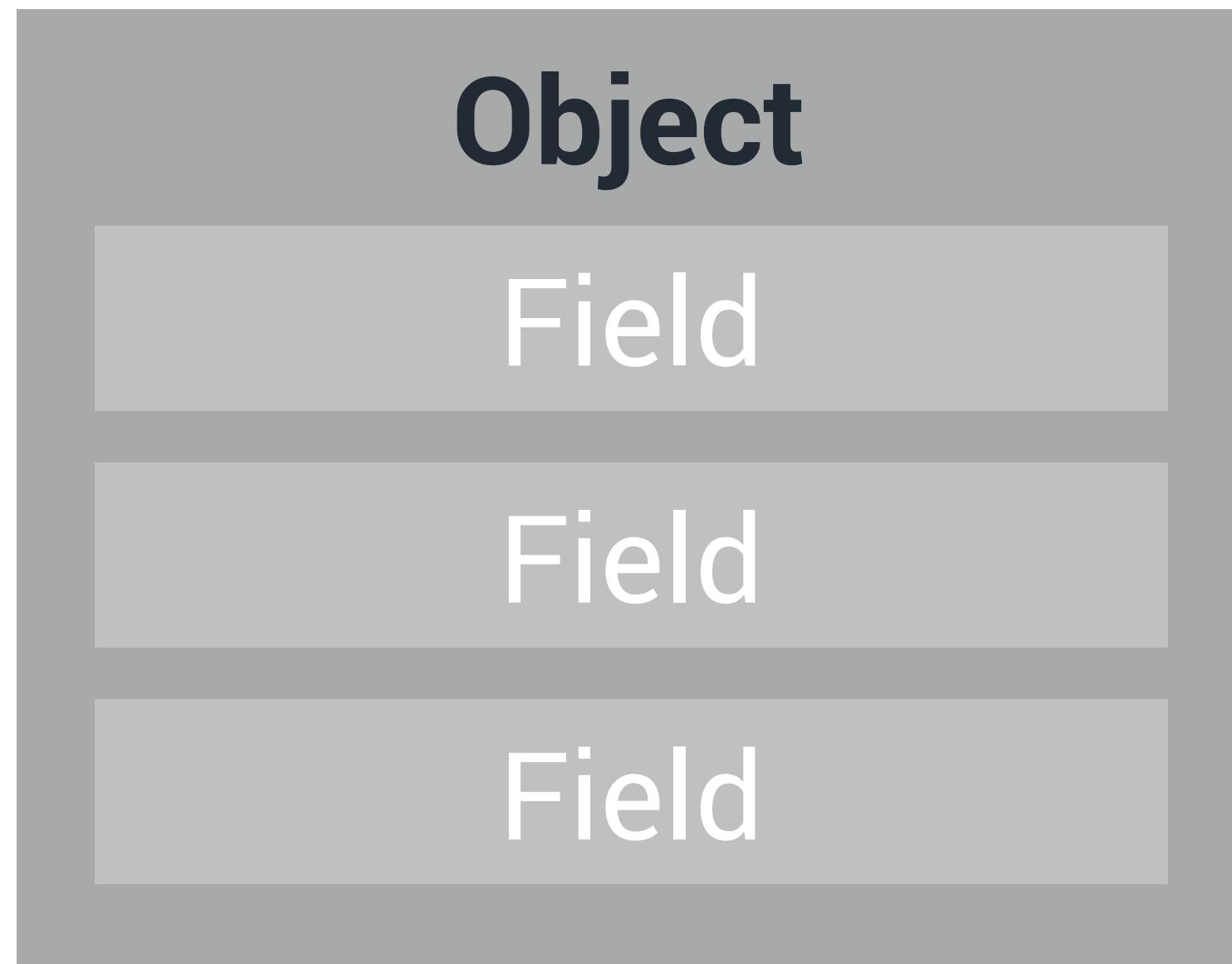
**Editor**



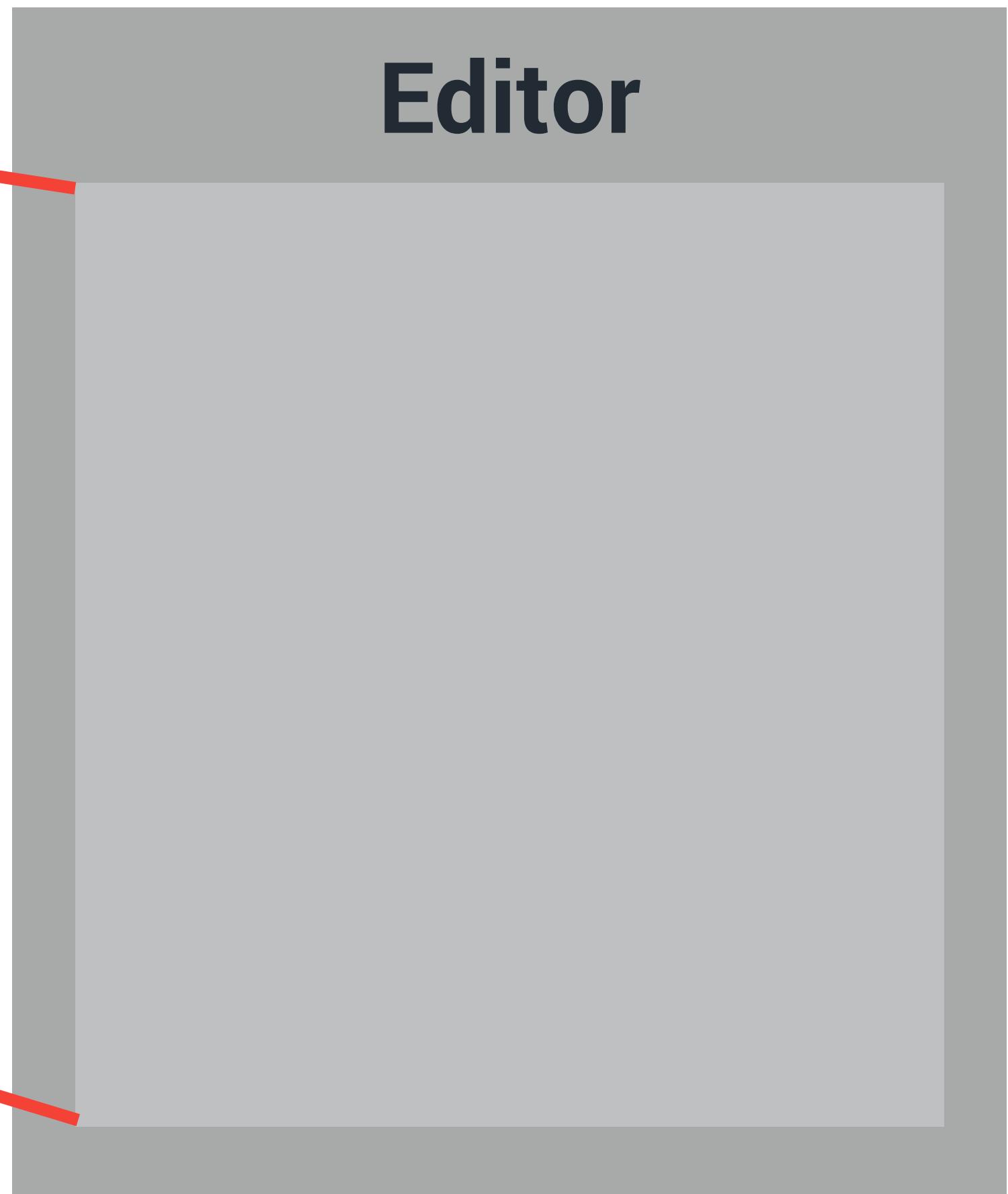
# Inventory

Phase 2/6

Runtime



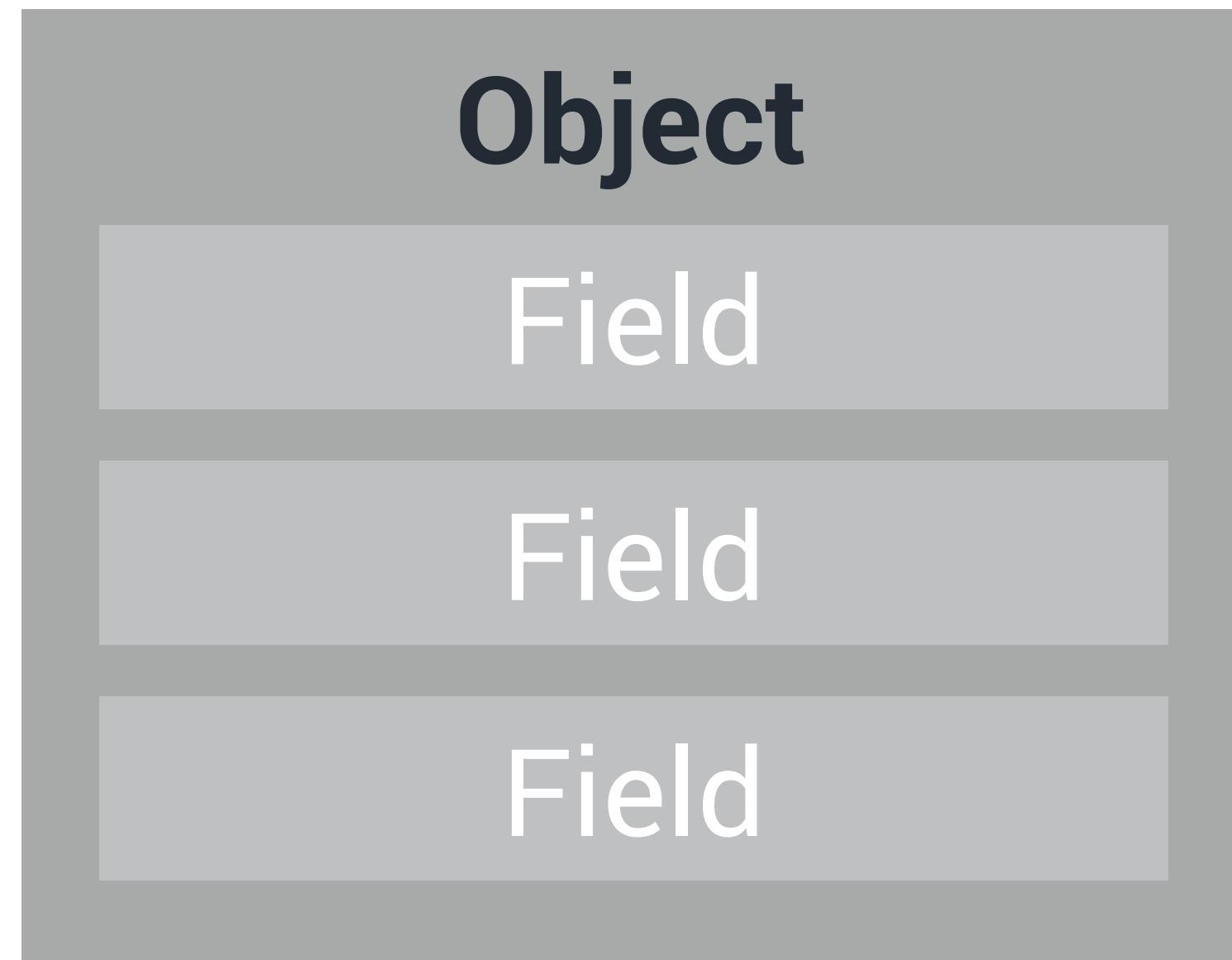
Edit Time



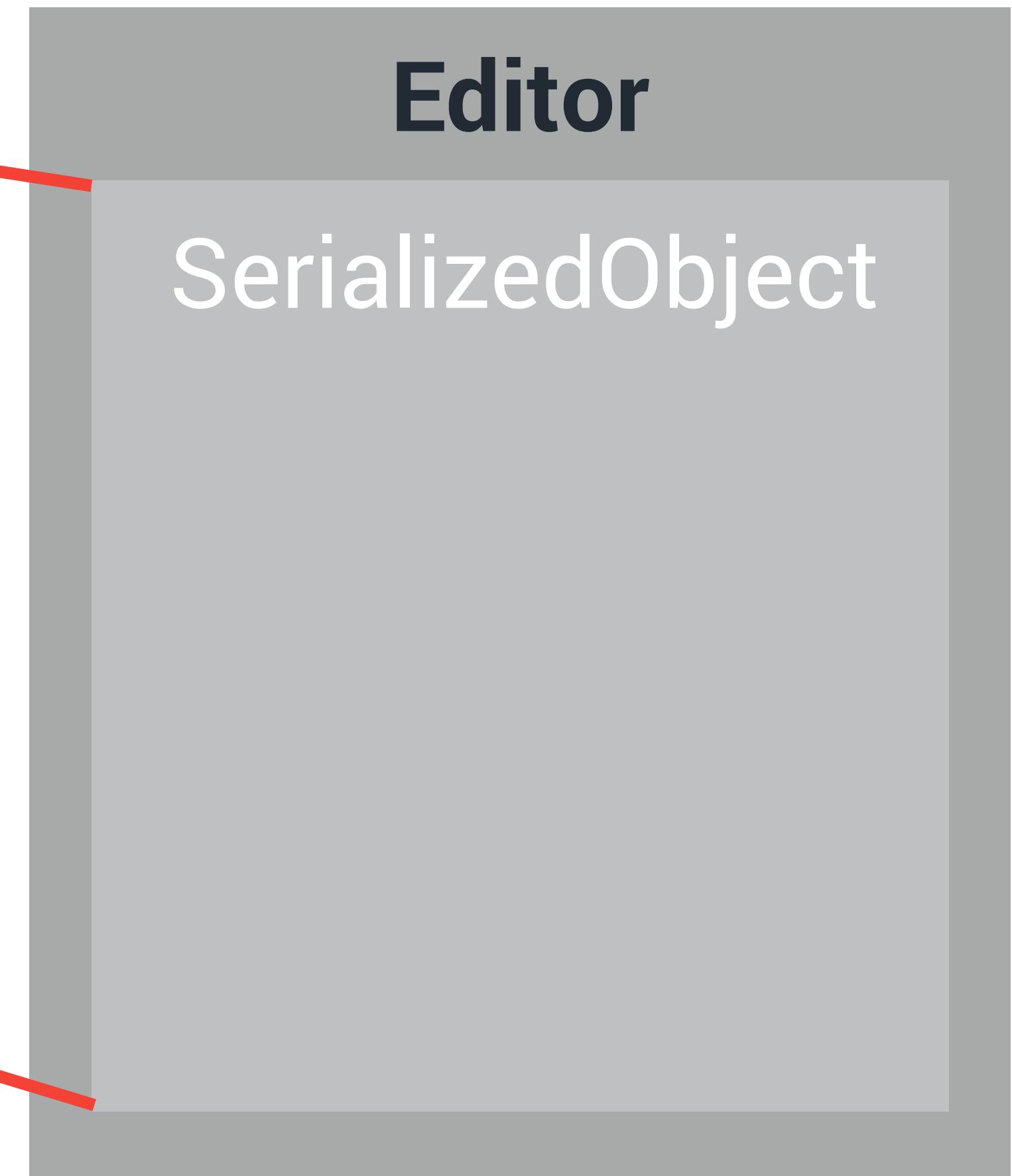
# Inventory

Phase 2/6

Runtime



Edit Time



# Inventory

Phase 2/6

Runtime

**Object**

Field

Field

Field

Edit Time

**Editor**

SerializedObject

SerializedProperty

SerializedProperty

SerializedProperty

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

## ***1. Save the script and return to the editor***

- 1. Observe the change in the Inspector**
- 2. Drag the ItemImage GameObject from each ItemSlot to the appropriate Image object field on the Inventory inspector**
- 3. *Save the scene***

1. **Test by entering play mode and picking up objects in the Market scene**
2. ***Exit Play Mode!***

*End of*  
**Phase 02**



# Phases 03 - 05

## *Interaction*



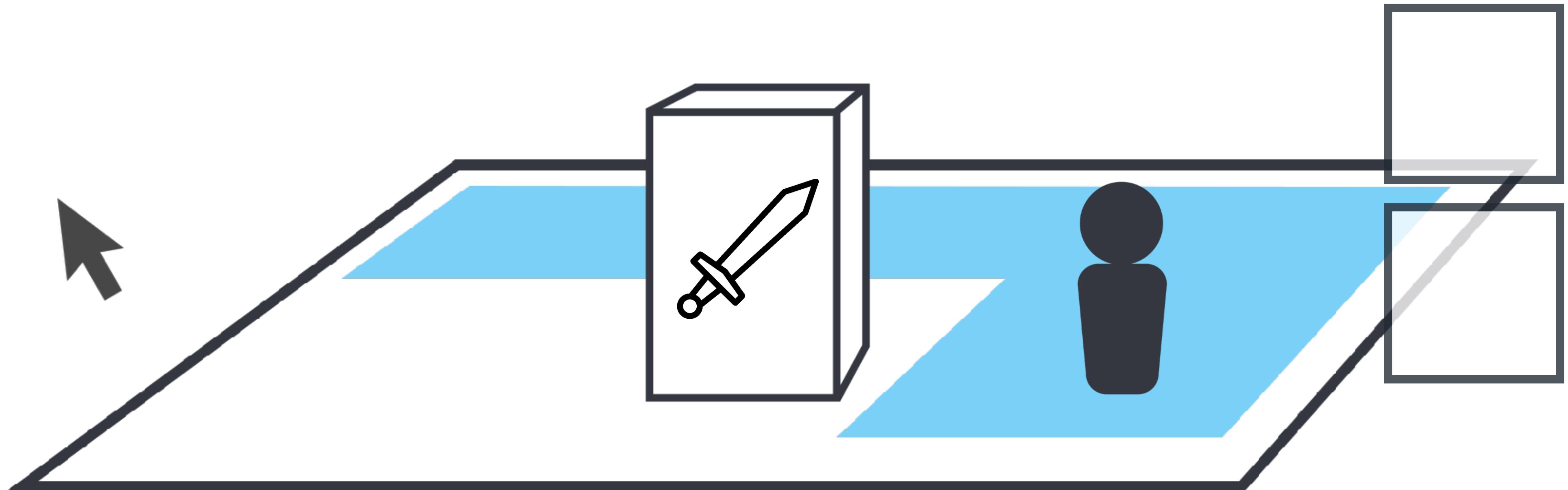
# *The Brief*



# Interaction System

Phases 3-5

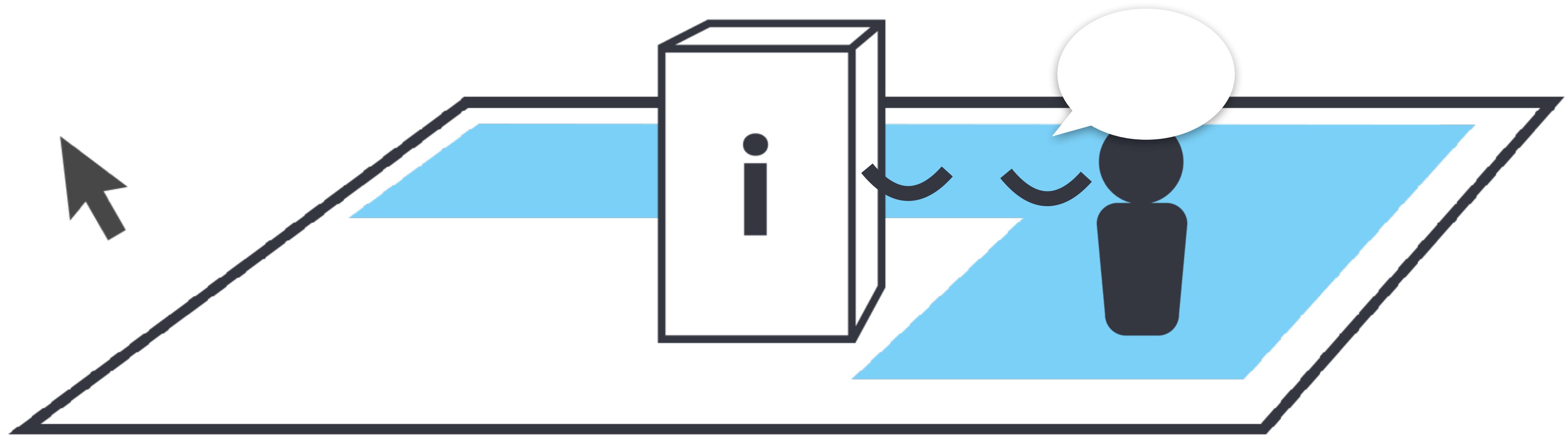
*Brief*



# Interaction System

Phases 3-5

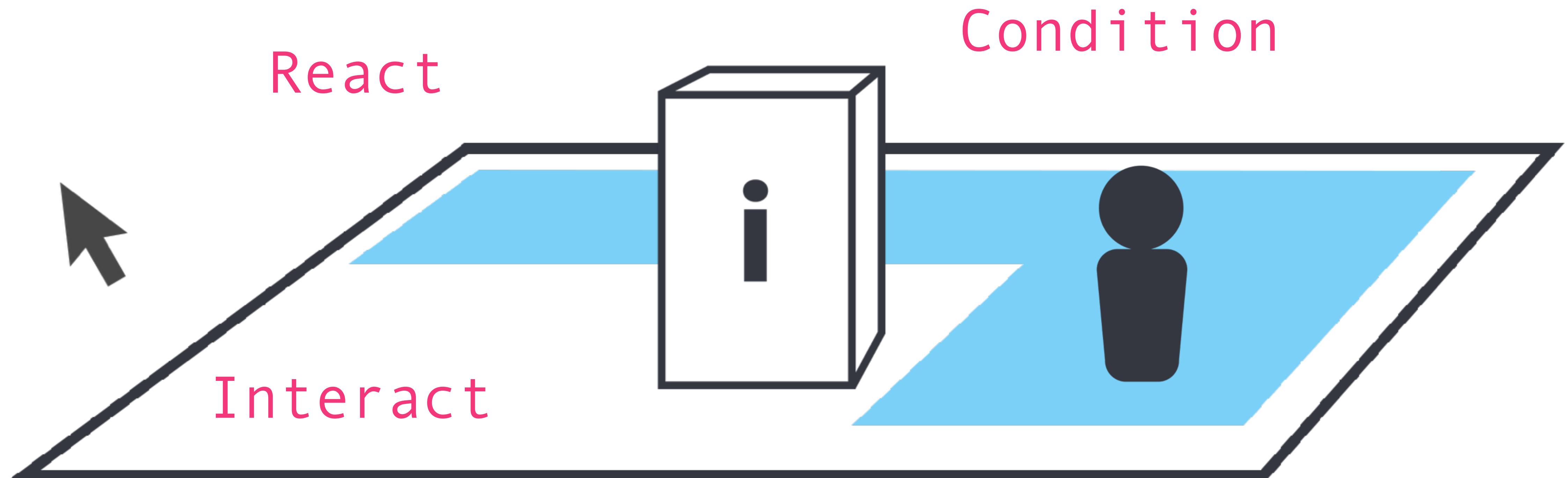
*Brief*



# Interaction System

Phases 3-5

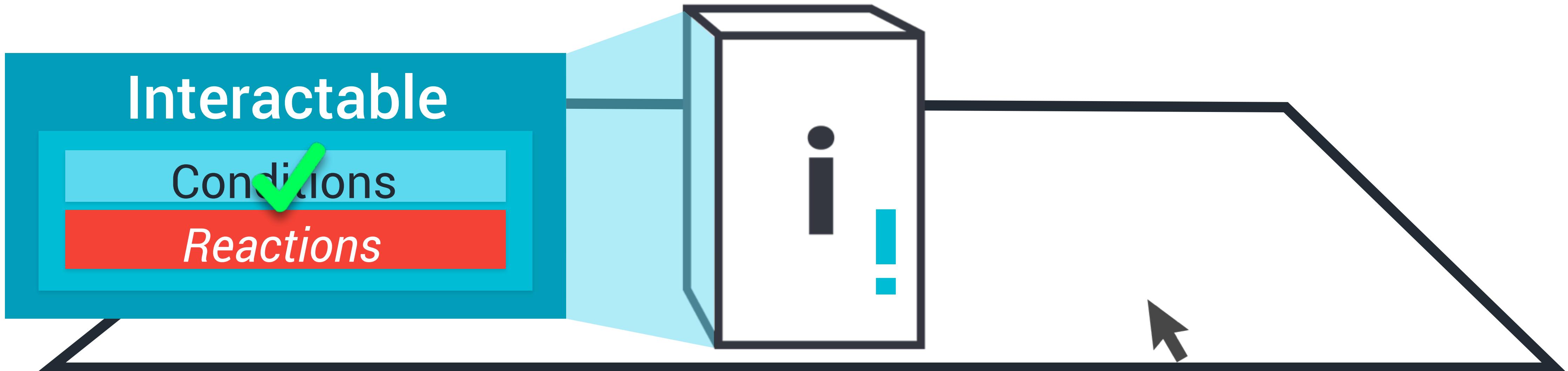
*Brief*



# *The Approach*

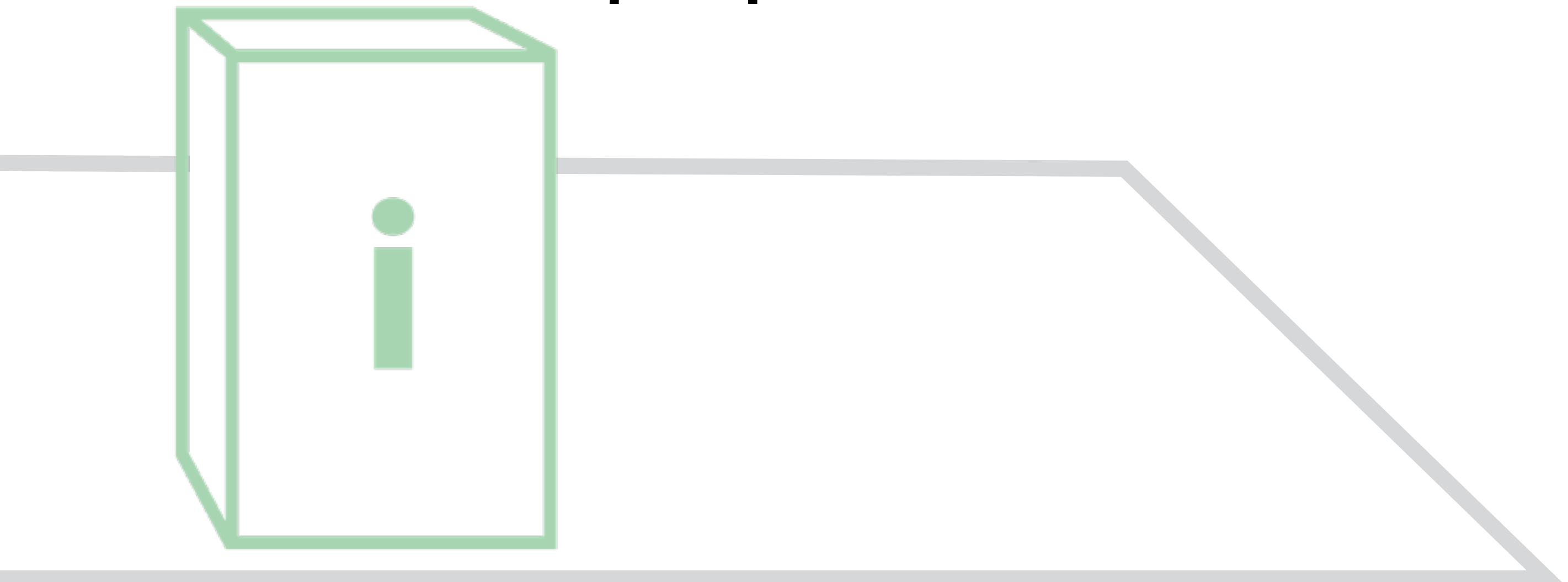
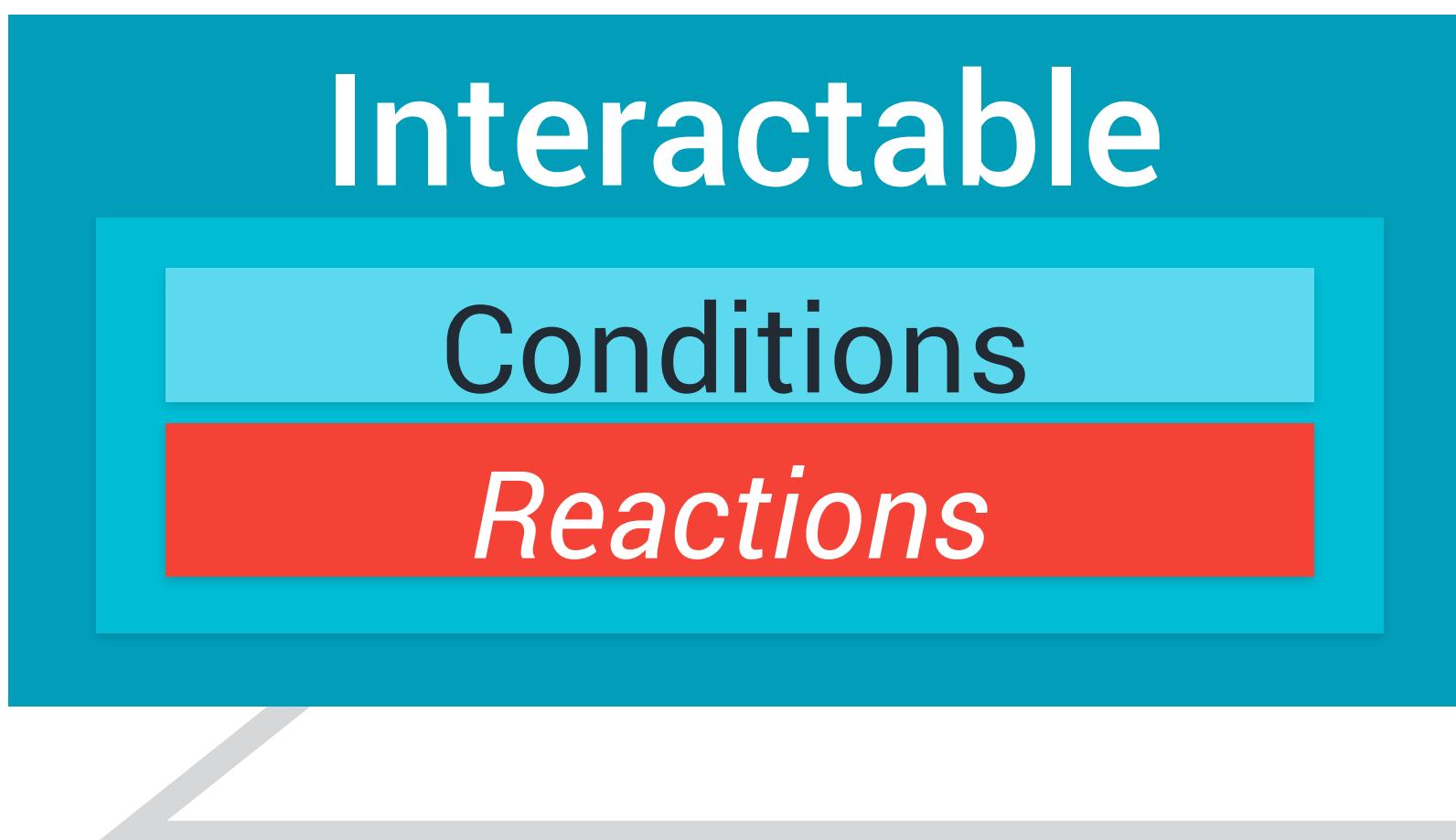
### Approach

- Create an **Interactable** GameObject > OnClickEvent  
> Condition > Reaction



### Approach

- The **Interactable** GameObject will be:
  - **stand-alone** and hold all logic & events
  - **decoupled** from the actual props in the scene



### *Approach*

- 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

# Interaction System Approach

*Player moves to  
Interactable*

Interactable

ConditionCollection

Condition ✓

ReactionCollection

Reaction

Reaction

Phases 3-5

## *Approach*

- **Conditions** are:

### *Approach*

- **Conditions** are:
  - Data objects that contain only an **identifier** and a **boolean**

### *Approach*

- **Conditions** are:
  - Data objects that contain only an **identifier** and a **boolean**
  - Saved as **ScriptableObject** assets that can be used to compare the state of a **Condition**

### *Approach*

- **Reactions:**

## *Approach*

- **Reactions:**
  - Accommodate a wide variety of possible actions

### *Approach*

- **Reactions:**
  - Accommodate a wide variety of possible actions
  - Use **Inheritance** and **Polymorphism** to create specific **Reactions** for each possible type of action

## *Approach*

- **Reactions will be Encapsulated**

### *Approach*

- **Reactions will be Encapsulated**
- **The Interactable will have a single object reference to a Reaction**

### *Approach*

- **Reactions will be Encapsulated**
- **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**

### *Approach*

- To improve workflow **Custom Inspectors** will be created to accommodate all of the different types of **Reactions, Conditions and Interactables**

# Phase 03

# *Conditions*

*Download the Asset Store package:*  
**3/6 - Adventure Tutorial - Conditions**



# *The Brief*



## *Brief*

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

# *The Approach*

## *Approach*

- All Conditions will be **ScriptableObject**s
- 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

Interactable

### Interactable Conditions

### Interactable Conditions *Description*

**Interactable  
Conditions**  
*Description*  
*Satisfied*

**Interactable  
Conditions**  
*Description*  
*Satisfied*

AllConditions

**Interactable  
Conditions**  
*Description*  
*Satisfied*

**All Conditions**  
**Conditions**  
*Description*  
*Satisfied*





Reaction

# Conditions

Phase 3/6



# Conditions

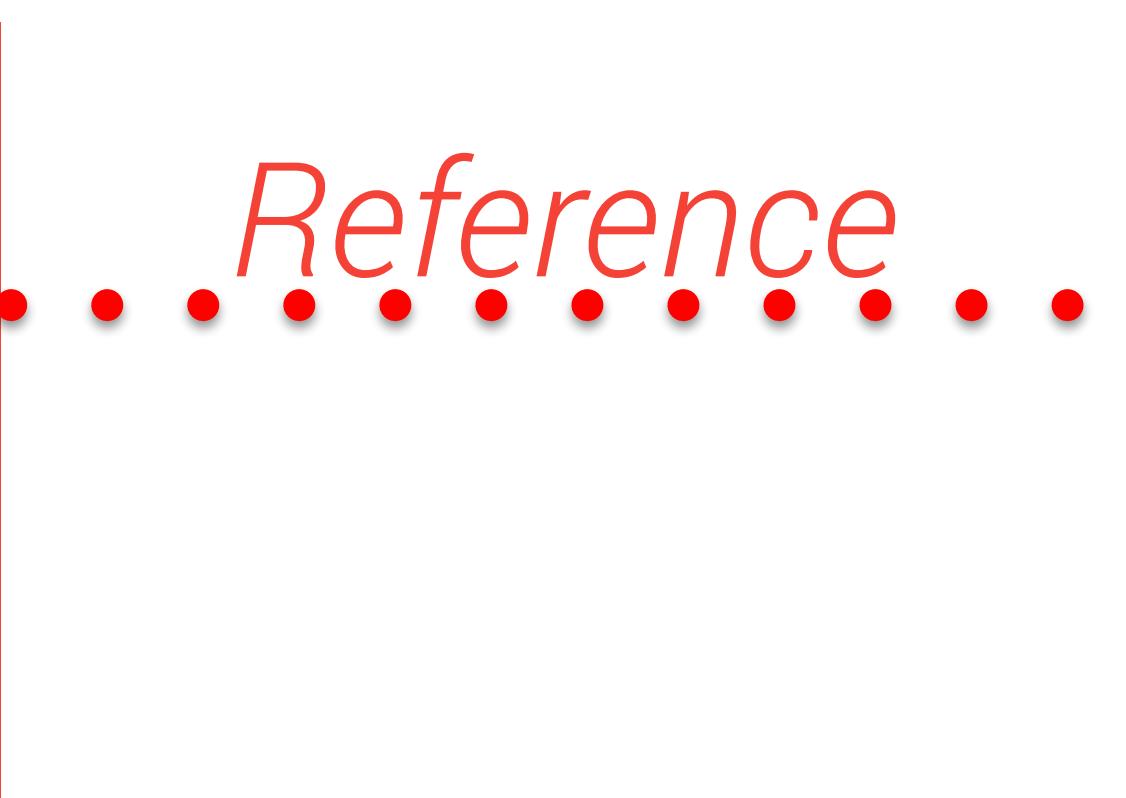
Phase 3/6

**Reaction**  
**Conditions**  
*Description*  
*Satisfied*

**All Conditions**  
**Conditions**  
*Description*  
*Satisfied*

# Conditions

Phase 3/6



# Conditions

Phase 3/6





## *Approach*

- **Conditions** will have an integer **hash** representing their description
- Comparing the **hash** will be **more efficient** than comparing the descriptions as strings

# *The Steps*



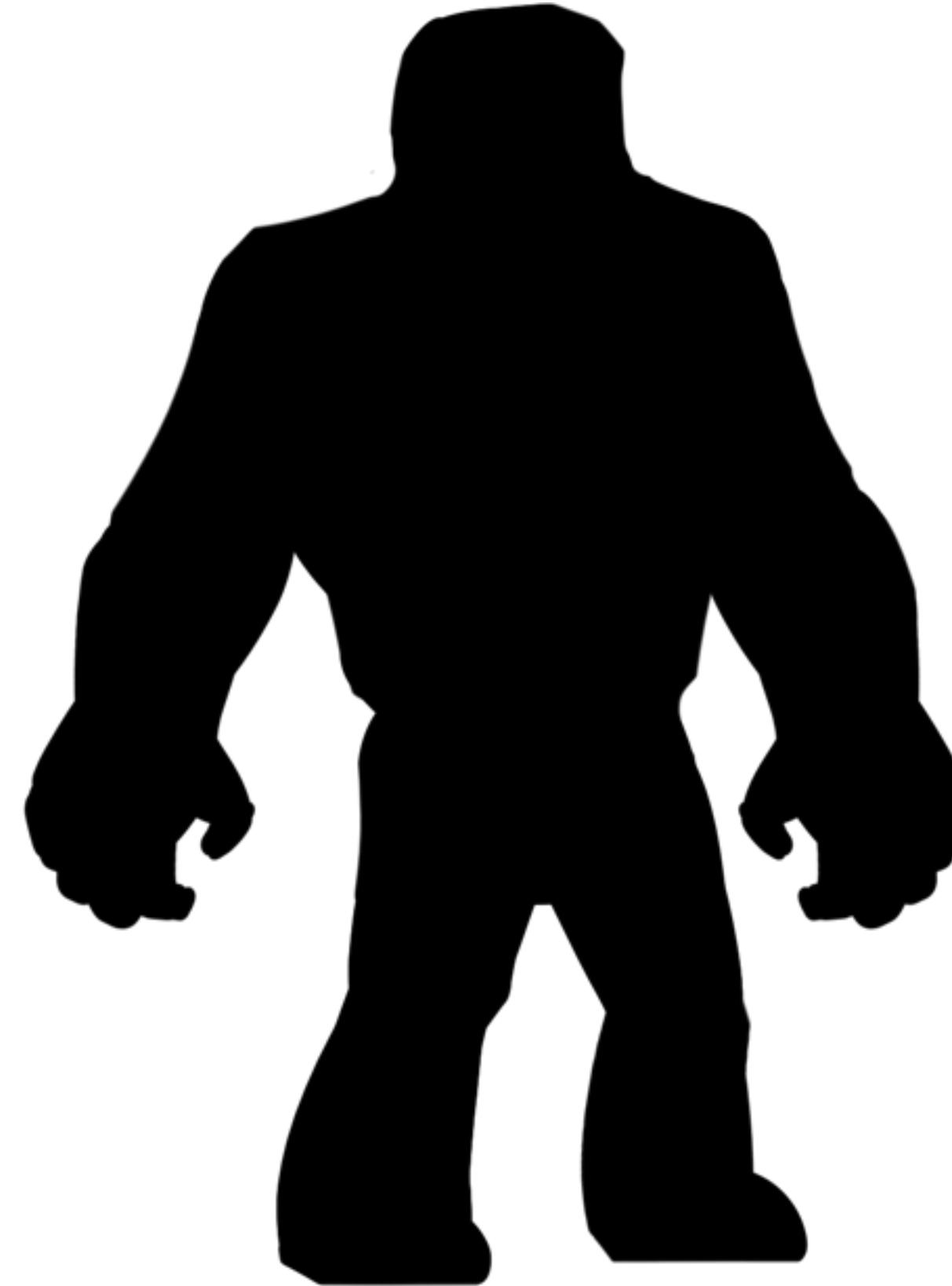
1. Please import the Asset Store package:

***3/6 - Adventure Tutorial - Conditions***

## *The Condition script*

## *The AllConditions script*

# Conditions



public class Enemy



public class Orc: Enemy



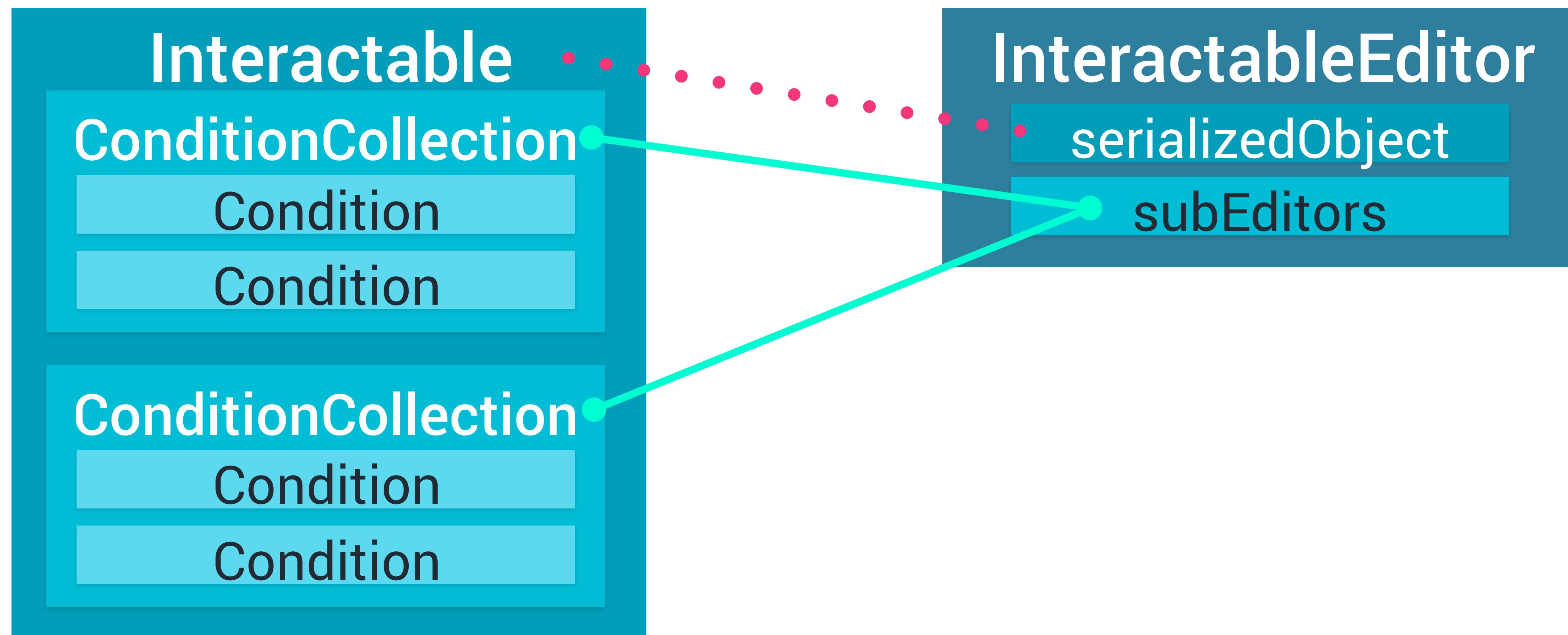
public class OrcChieftan: Orc

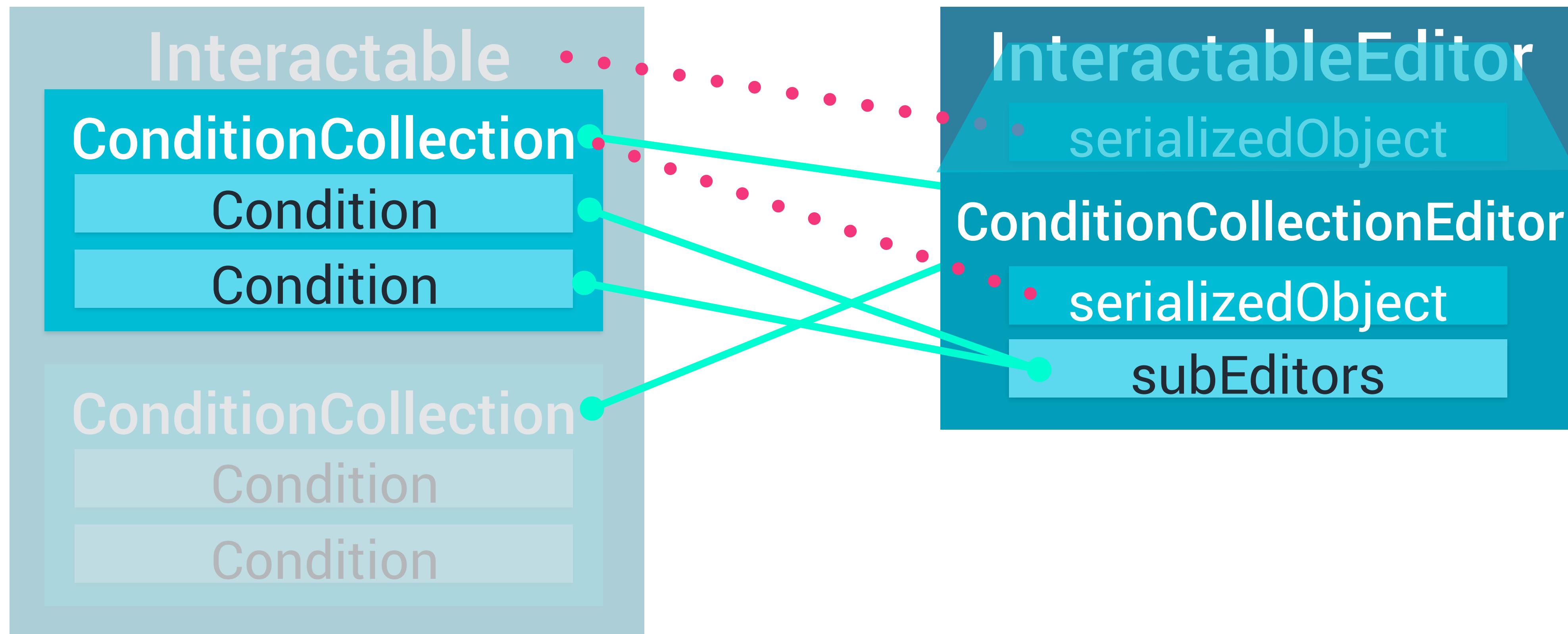
## *The AllConditions script*

1. **Navigate to *Scripts > ScriptableObjects > Interaction > Conditions* folder**
2. **Open the ConditionCollection script for editing**

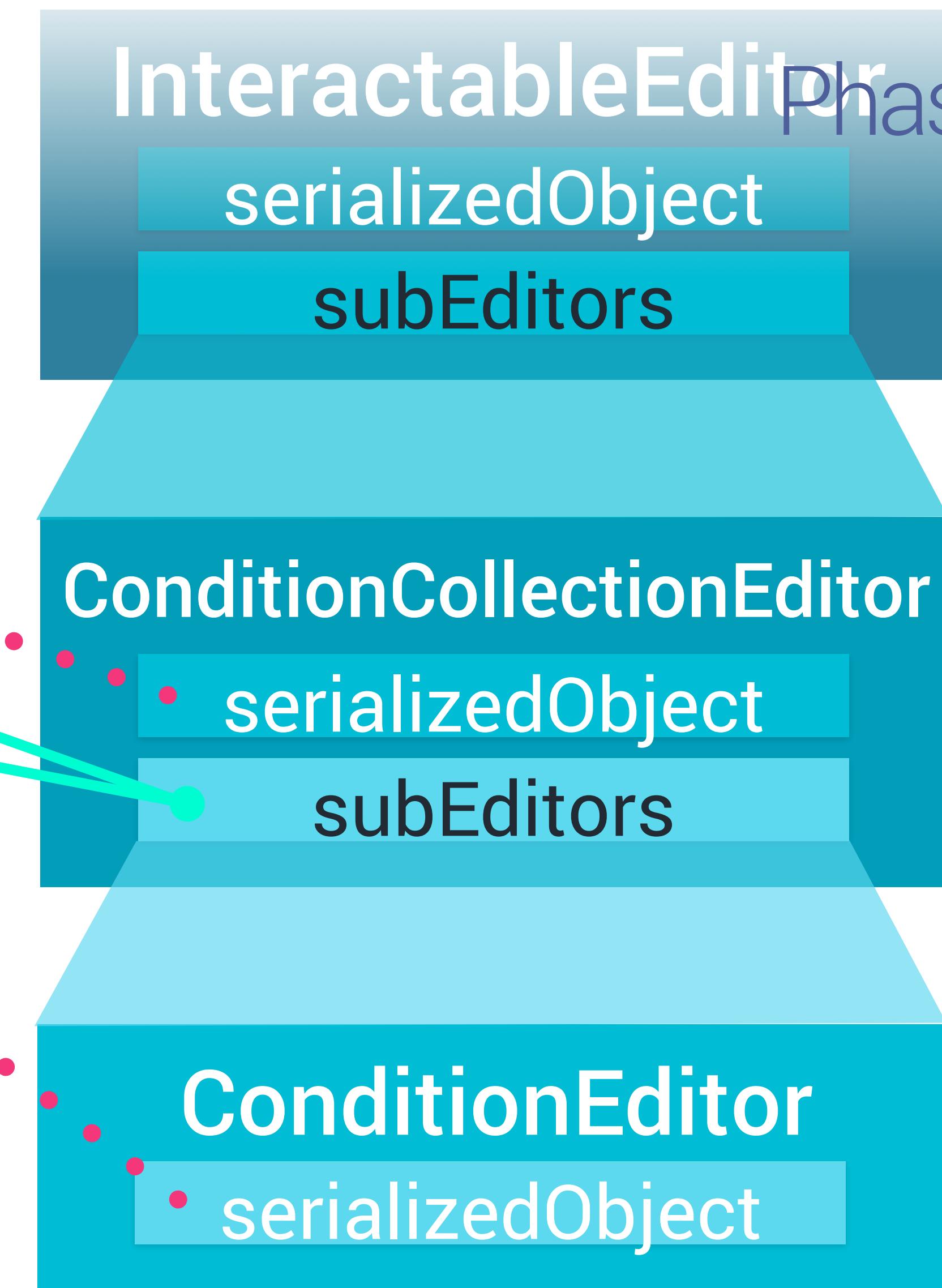
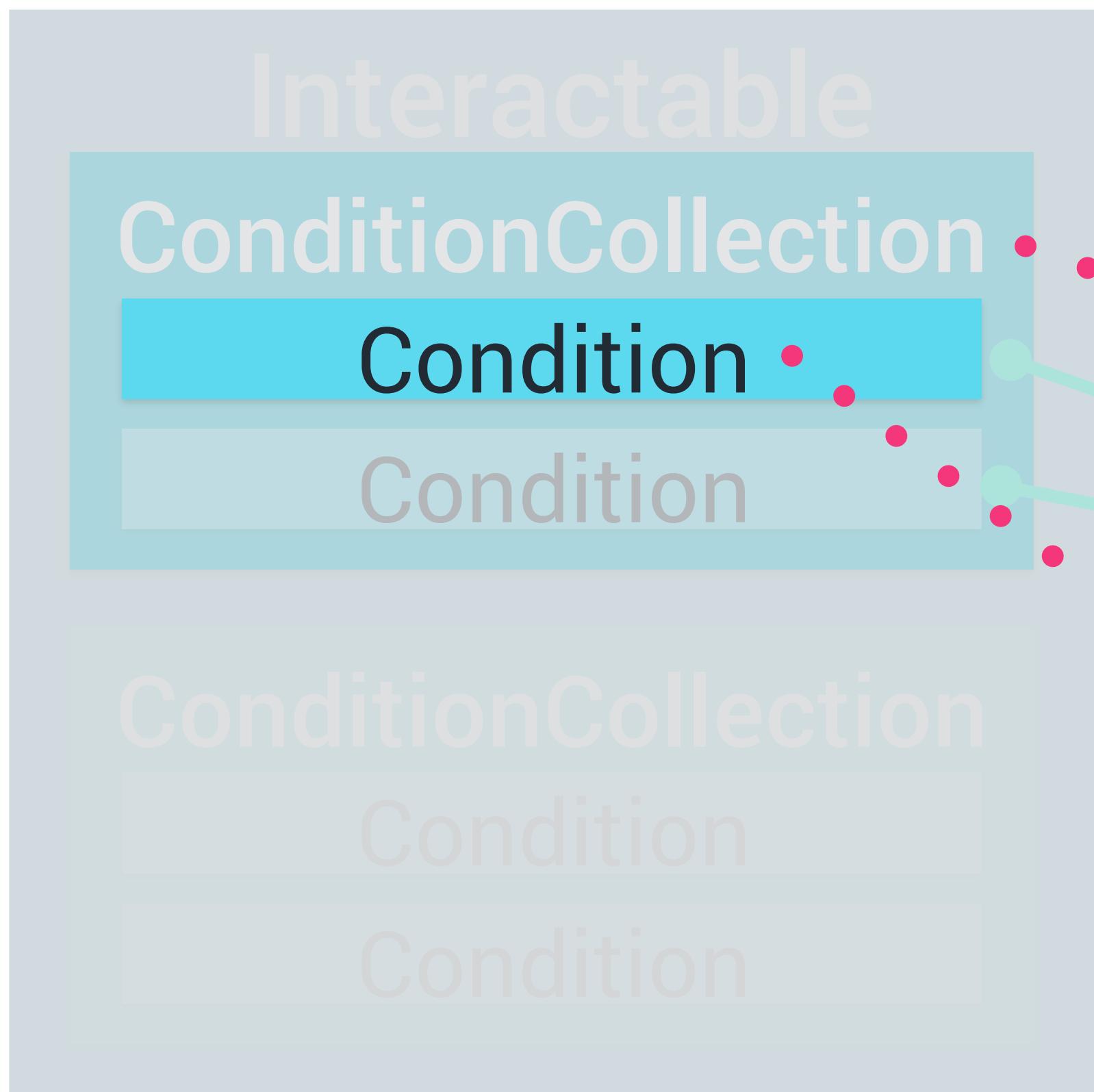
## ***1. Save the script and return to the editor***

*The EditorWithSubEditors script*





# Conditions



Conditions

Phase 3/6

*ConditionCollectionEditor*

1. **Navigate to the *Scripts > Editor > Interaction > Conditions* folder**
2. **Select the ConditionCollectionEditor script and open it for editing**

*The RemoveFromObjectArray extension method of the  
SerializedPropertyExtensions script*

## ***1. Save the script and return to the editor***

1. **Navigate the Scenes folder in the Project window**
2. **Open the SecurityRoom scene**
3. **Select the LaserGridInteractive GameObject**
4. On the **Interactive** component, click the **Add Collection** button
5. Set the **Description** of the new condition collection to **LaserGridOff**

1. From the dropdown, set the **Condition** to ***LasersDeactivated***
2. Check the **Satisfied** box
3. Expand the **LaserGridInteractable**
4. Drag the **BeamsOffReaction** from the **hierarchy** onto the **Interactable Reaction** field for the **Condition Collection**

*End of*  
**Phase 03**



# Phase 04

# *Reactions*

*Download the Asset Store package:*  
**4/6 - Adventure Tutorial - Reactions**



# *The Brief*



## Brief

- **Create a system that performs a series of actions based on Conditions and the current Game State when the Player Clicks on an Interactable**

## Brief

- There can be many different types of **Reactions** such as:
  - Play an animation
  - Play a sound
  - Display text on screen
  - Add the item to the inventory
  - Disable a GameObject in the scene
  - Change a global condition to acknowledge the pickup

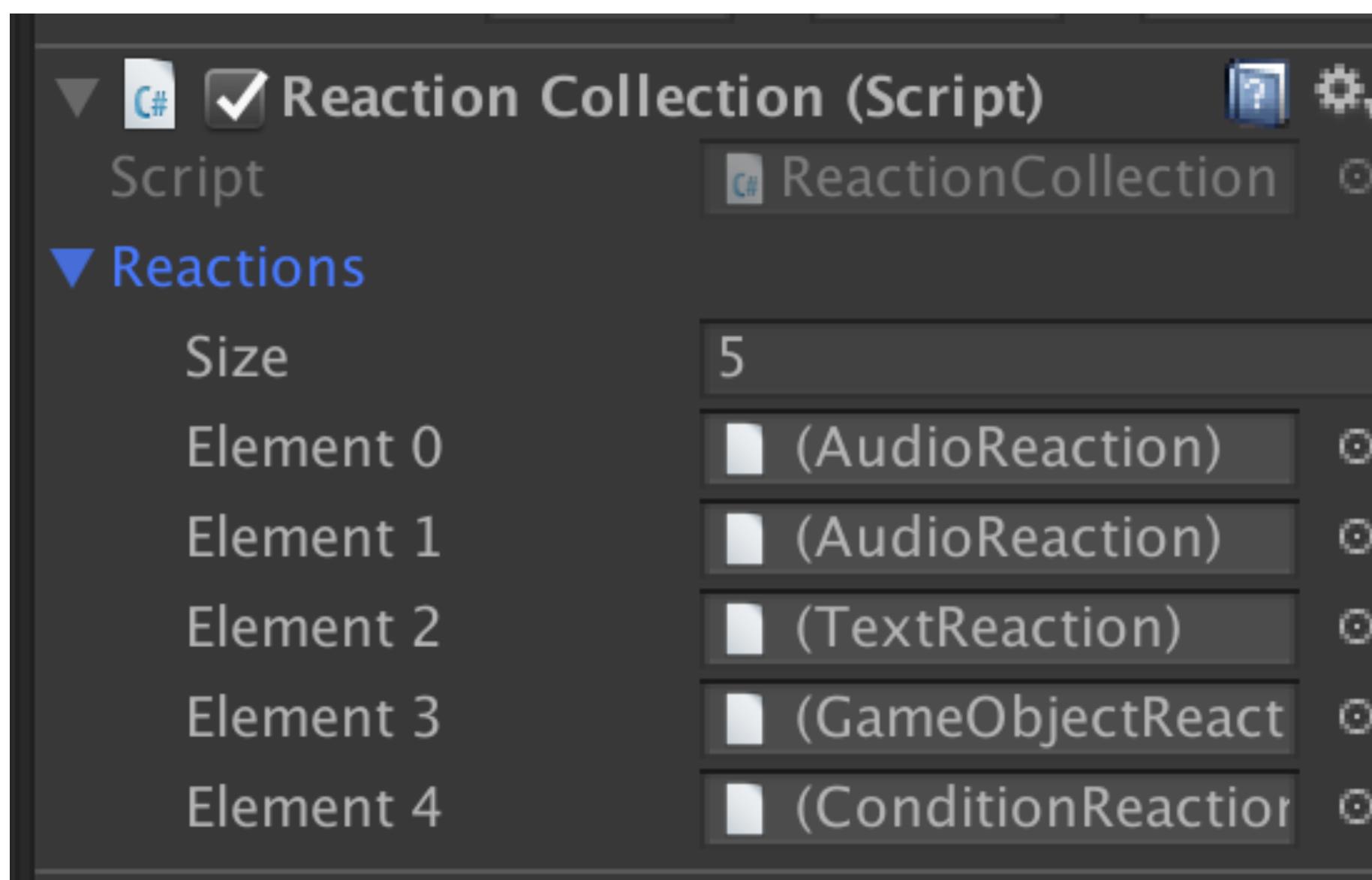
## Brief

- Depending on type, some **Reactions** must have an **optional delay** while others must be **instant**



*Brief*

- Each type of Reaction will need its own **Custom Editor**



# *The Approach*

## *Approach*

- Collections of **Reactions** will be encapsulated, initialized and called as part of a **ReactionCollection** script which contains a single public **React** function

## *Approach*

- Individual **Reactions** will all be **different classes** inheriting from a **base Reaction class** as this will maintain the key functionality across all the variations of **Reactions** and allow them to be treated as **Reactions due to Polymorphism**

## *Approach*

- **Reactions** will derive from **ScriptableObject** which allows **Polymorphic Serialization** so that **custom inspectors** can be created based on the type of the **Reaction**

# *The Steps*



1. Please import the Asset Store package:

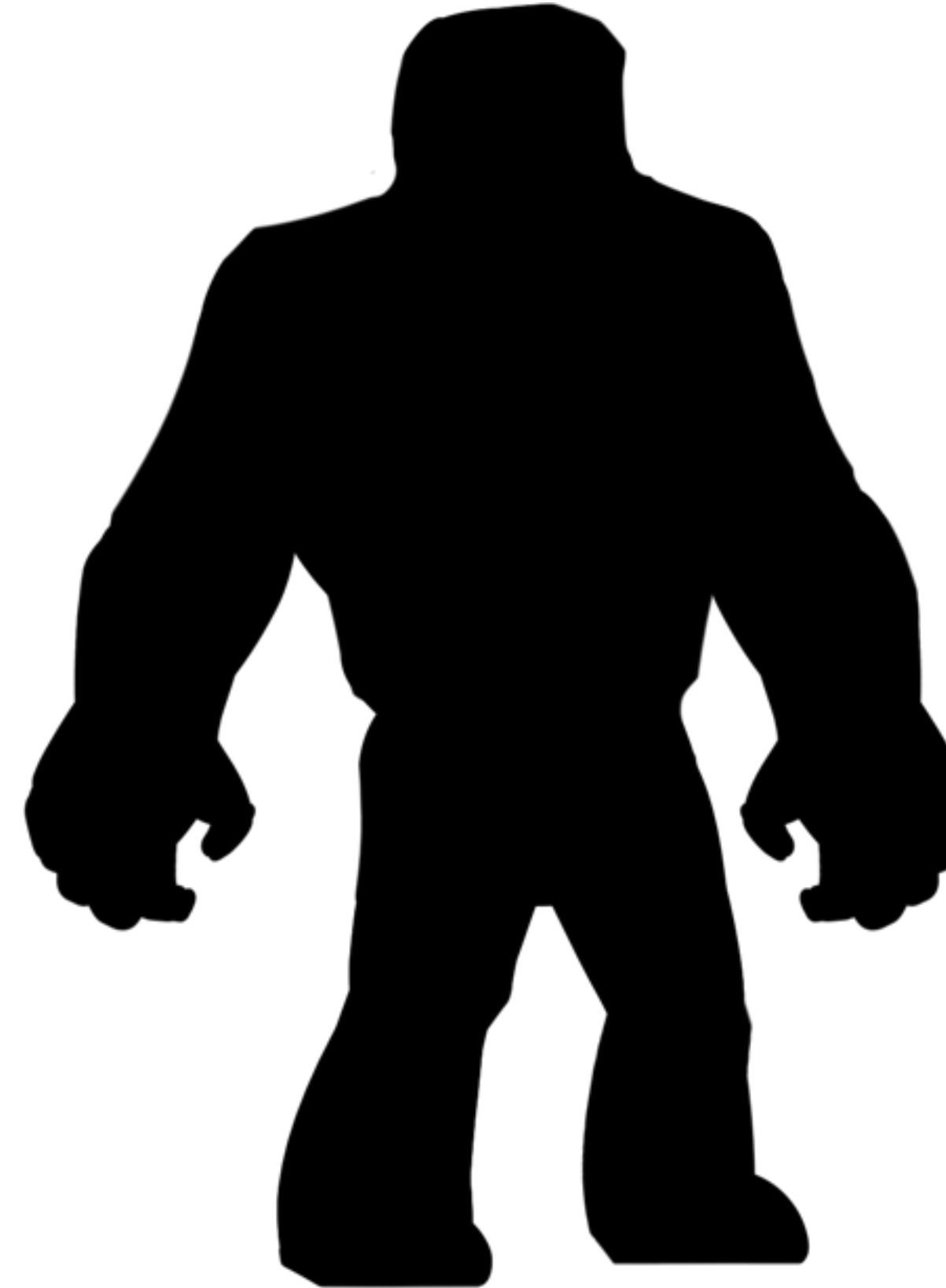
***4/6 - Adventure Tutorial - Reactions***

*ReactionCollection script*

## *Inheritance and Polymorphism*

# Reactions

Phase 4/6



public class Enemy



public class Orc : Enemy

public class Goblin : Enemy



public class OrcChieftan : Orc

# Reactions

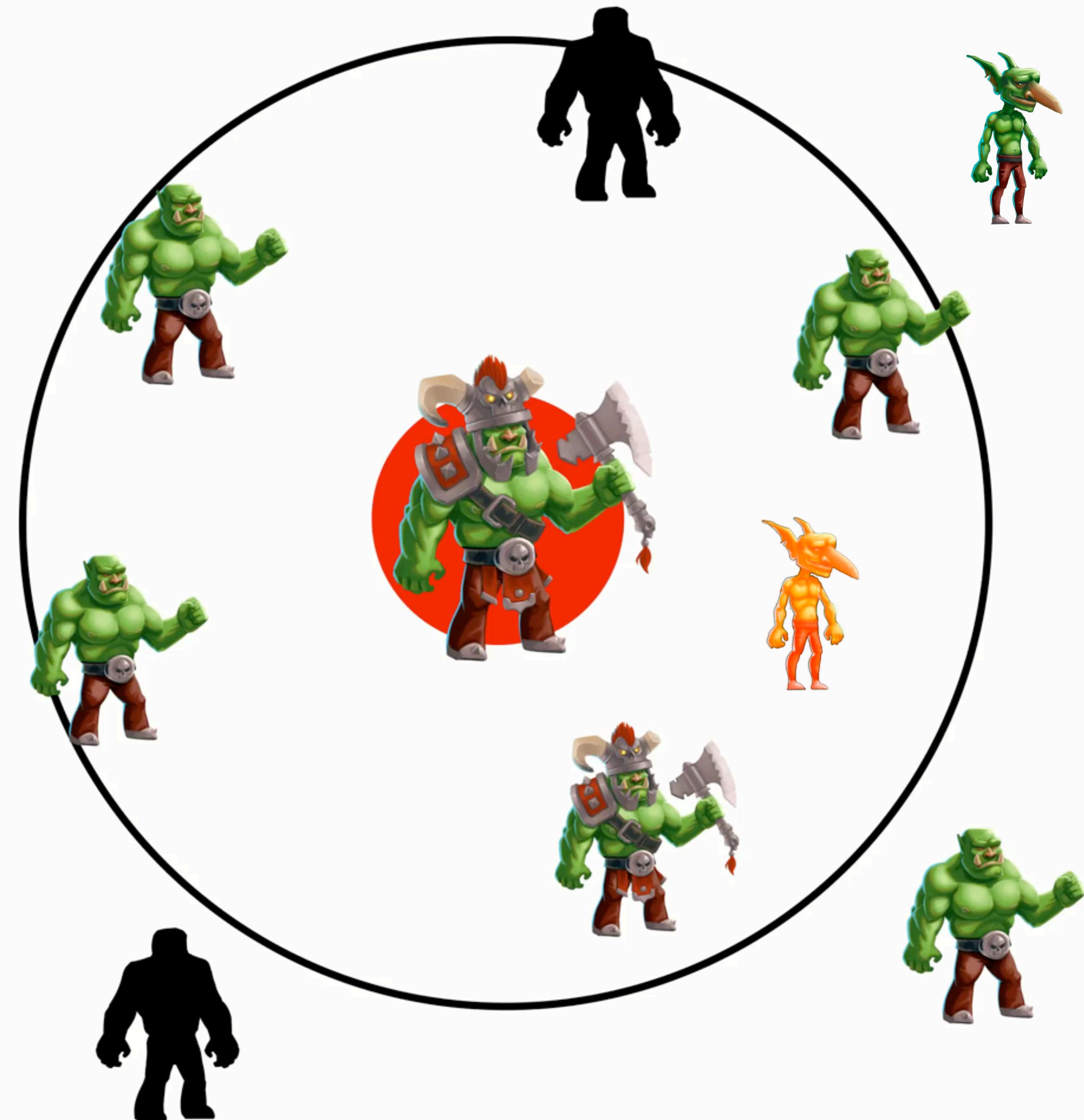
Phase 4/6



Unite 2016 - Training Day

# Reactions

Phase 4/6



# Reactions

Phase 4/6



## *Scriptable Objects and Polymorphic Serialization*

# Reactions

Phase 4/6

## Reaction

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

# Reactions

Phase 4/6

## Reaction

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

ReactionCollection

# Reactions

Phase 4/6

## Reaction

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

ReactionCollection

Reaction

Reaction

Reaction

## Reaction

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

ReactionCollection

Reaction

Reaction

Reaction

## Reaction : ScriptableObject

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

## Reaction

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

ReactionCollection

Reaction

Reaction

Reaction

## Reaction : ScriptableObject

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

ReactionCollection

# Reactions

Phase 4/6

## Reaction

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

ReactionCollection

Reaction

Reaction

Reaction

## Reaction : ScriptableObject

```
public Reaction[] reactions; // TextReaction, AudioReaction, AnimationReaction
```

ReactionCollection

TextReaction

AudioReaction

AnimationReaction

*Return to ReactionCollection script*

1. **Navigate to the *Scripts > Editor > Interaction***
2. **Find the ReactionCollectionEditor**
3. **Open the ReactionCollectionEditor script for editing**
4. This is the finished custom editor we will be working on:

## ***1. Save the script and return to the editor***

1. **Create a new Empty GameObject**
2. **Add an ReactionCollection component**
3. **Navigate to *Scripts > ScriptableObjects > Interaction > Reactions* folder**
4. **Drag either a DelayedReactions or ImmediateReactions onto the Drop area of the ReactionCollection component**

1. **Delete** the test GameObject you just made

*Creating a specific Reaction type*

1. **Navigate to the *Scripts > ScriptableObjects > Interaction > Reactions > ImmediateReactions* folder.**
2. **Create a new C# script**
3. **Name it TextReaction**
4. **Open the script for editing**

## 1. *Save the script and return to the editor*

1. **Navigate to the *Scripts > Editor > Interaction > ReactionEditors* folder.**
2. **Create a new C# script**
3. **Name it *TextReactionEditor***
4. **Open the *TextReactionEditor* script for editing**

## 1. *Save the script and return to the editor*

1. Open the **SecurityRoom** scene
2. In the **Hierarchy**, find the **DefaultReaction** child of the **PictureInteractable**
3. Add a **TextReaction** using either the dropdown or the drop area
4. Set the Message of the Text Reaction to “*He looks pretty trustworthy.*”
5. **Save the scene**

1. Open the Persistent scene and test

2. *Exit Play mode!*

*End of*  
**Phase 04**



# Phase 05

# *Interactables*

*Download the Asset Store package:*  
**5/6 - Adventure Tutorial - Interactables**



# *The Brief*



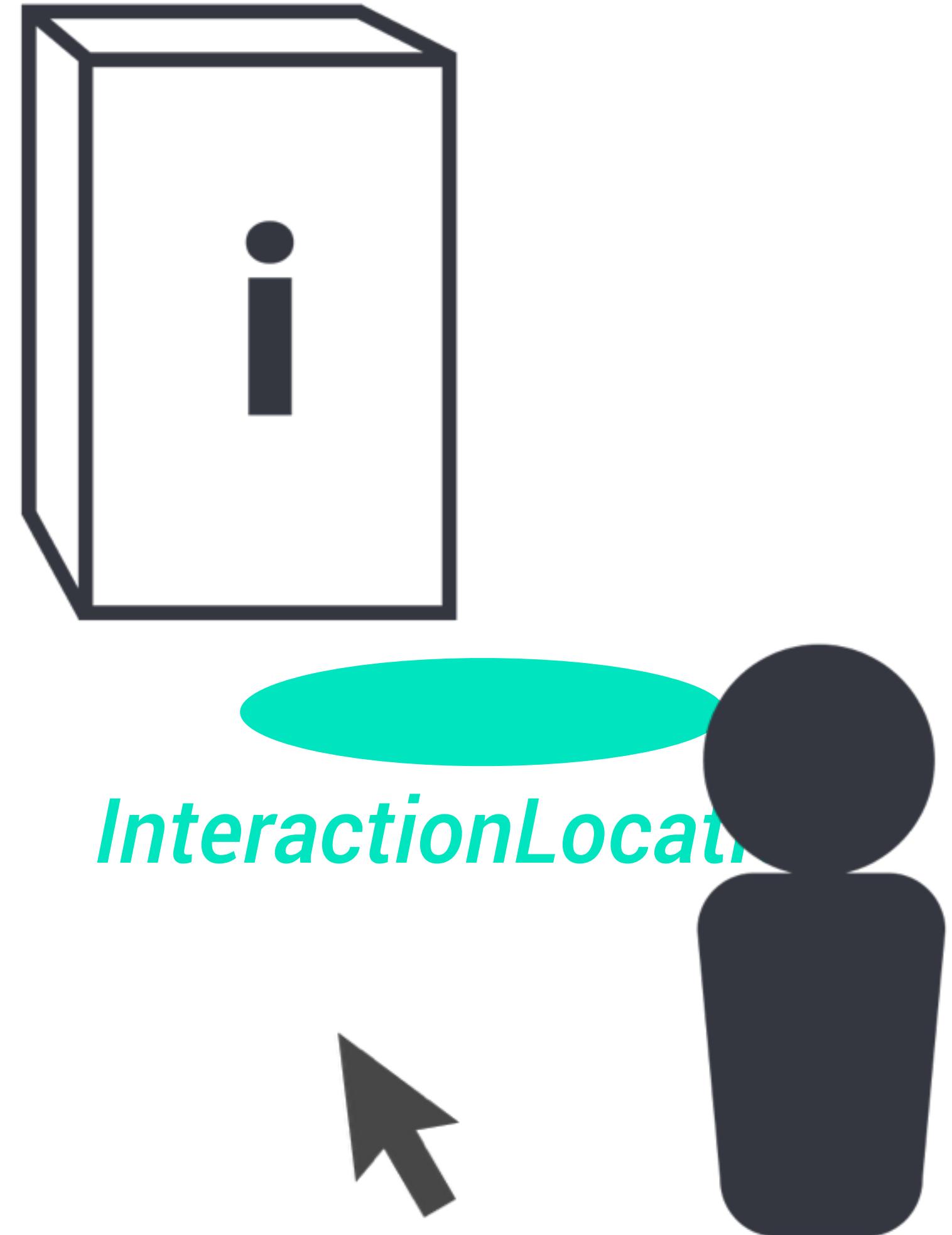
*Brief*

**Create an Interactable system to process Player input and tie Conditions and Reactions together**



## Brief

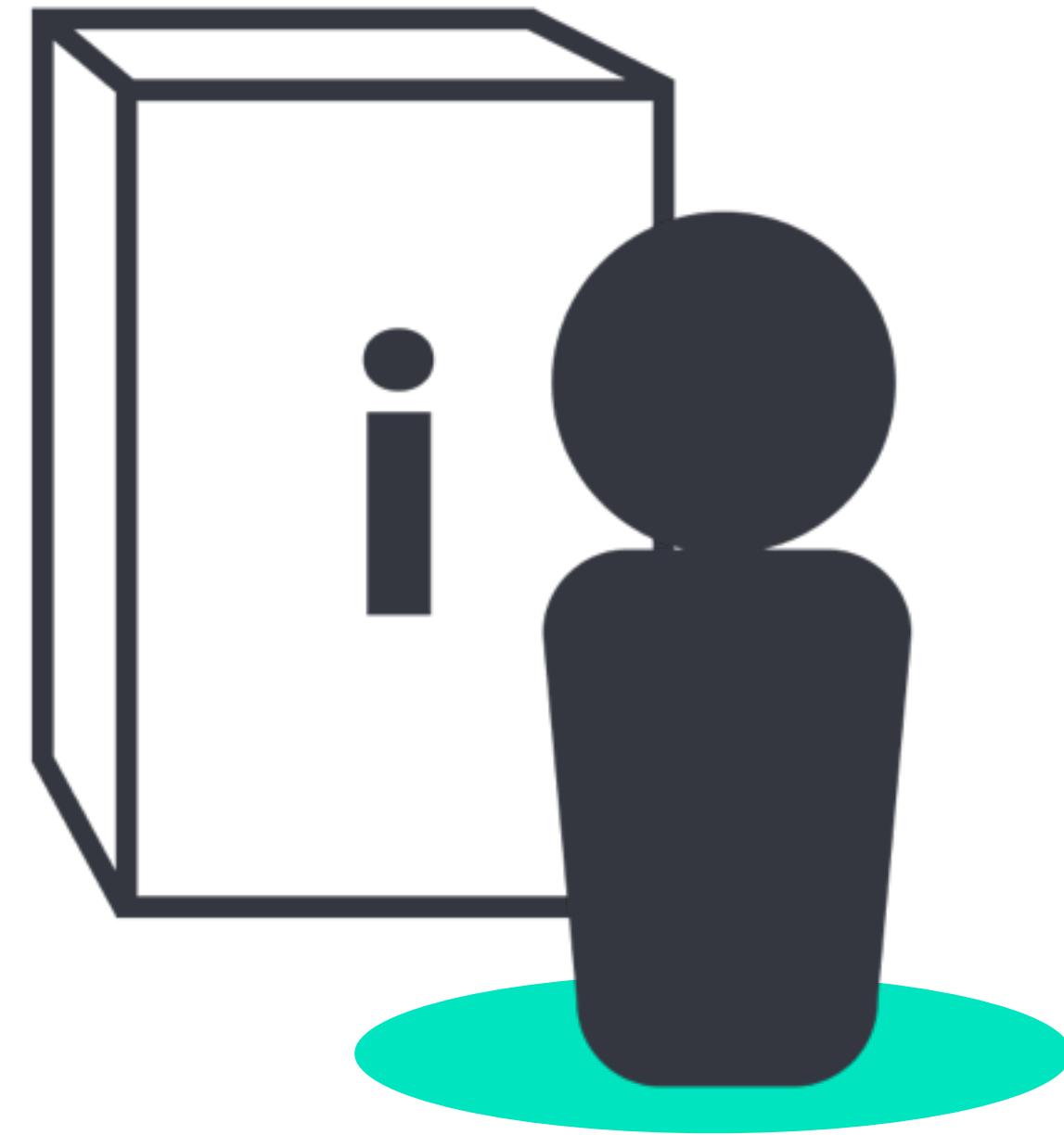
When an **Interactable** is clicked on the **Player** must move to a specific location in the scene marked by a **Transform reference** called **InteractionLocation**



## Brief

Upon reaching the **InteractionLocation** the player must call a public **Interact** function which will in turn call an appropriate **Reaction** collection based on a check of **Conditions and Game State**

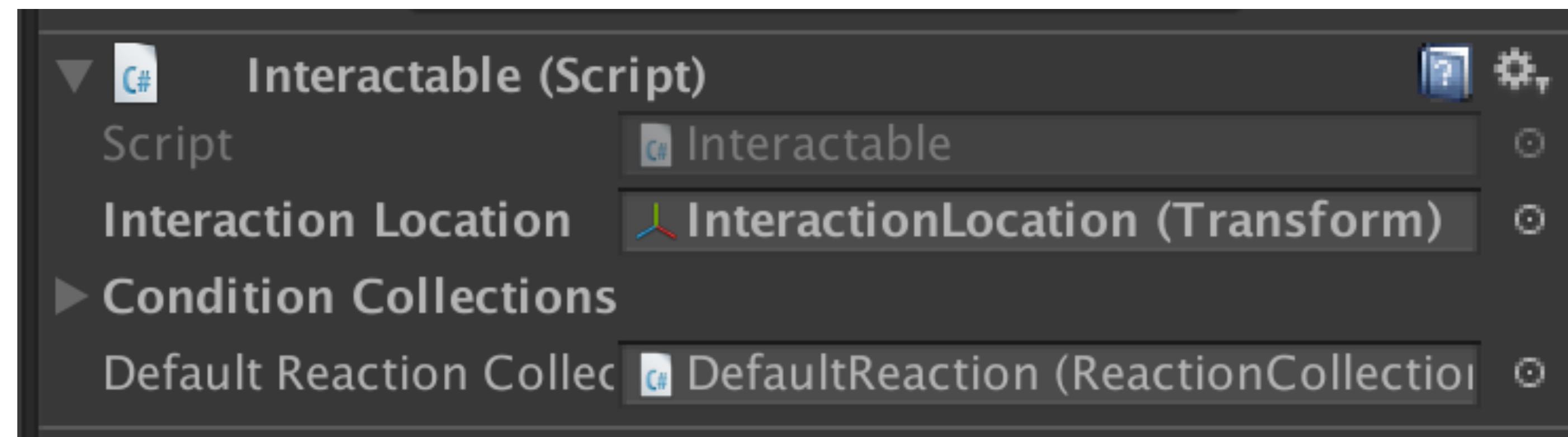
## Interact



**InteractionLocation**

## Brief

**Create a custom inspector for the Interactable to make it easier to understand**



# *The Approach*

## *Approach*

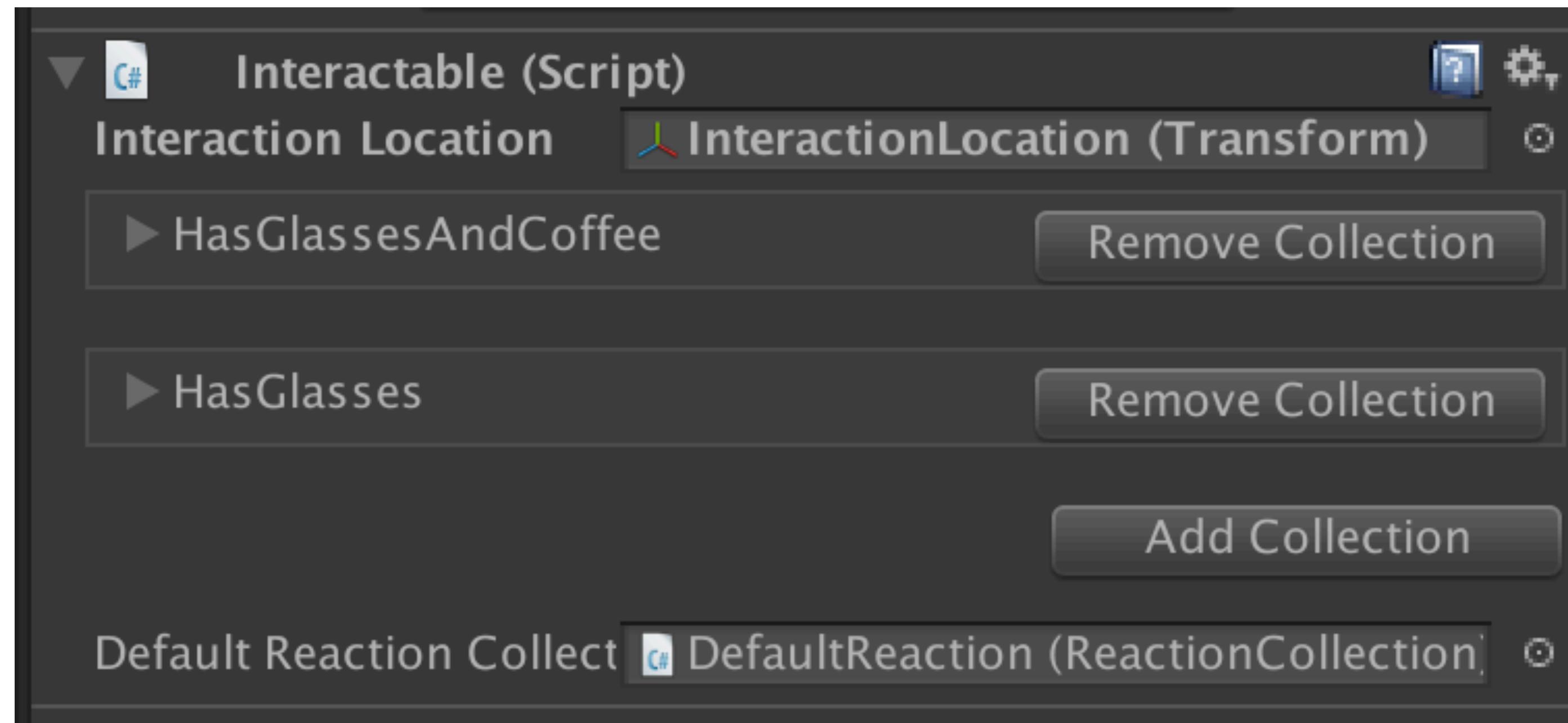
- The Interactable will have:
  - An **EventTrigger** component to registers clicks
  - A **Box Collider** to define the volume to interact with

## *Approach*

- When clicked, the **Interactable** sets the current destination for the **Player**
- When the **Player** arrives, the **Interact** function will be called by the **PlayerMovement** script

## Approach

- Create a **Custom Inspector** for the **Interactable** component



# *The Steps*



1. Please import the Asset Store package:

***5/6 - Adventure Tutorial - Interactables***

- 1. Navigate to the Scenes folder**
- 2. Open the SecurityRoom scene**

*The Event System*

## The Event System

### 1. Send events

- 1. Physics Raycaster component

### 2. Receive events

- 1. Colliders & Event Triggers

### 3. Manage events

- 1. The Event System

## The Event System

### 1. Send events

1. *Physics Raycaster component*

### 2. Receive events

1. Colliders & Event Triggers

### 3. Manage events

1. The Event System

## The Event System

1. Send events

1. Physics Raycaster component

**2. Receive events**

1. Colliders & Event Triggers

3. Manage events

1. The Event System

## The Event System

1. Send events

1. Physics Raycaster component

**2. Receive events**

1. *Colliders & Event Triggers*

3. Manage events

1. The Event System

## The Event System

1. Send events

1. Physics Raycaster component

2. Receive events

1. Colliders & Event Triggers

**3. Manage events**

1. The Event System

## The Event System

1. Send events

1. Physics Raycaster component

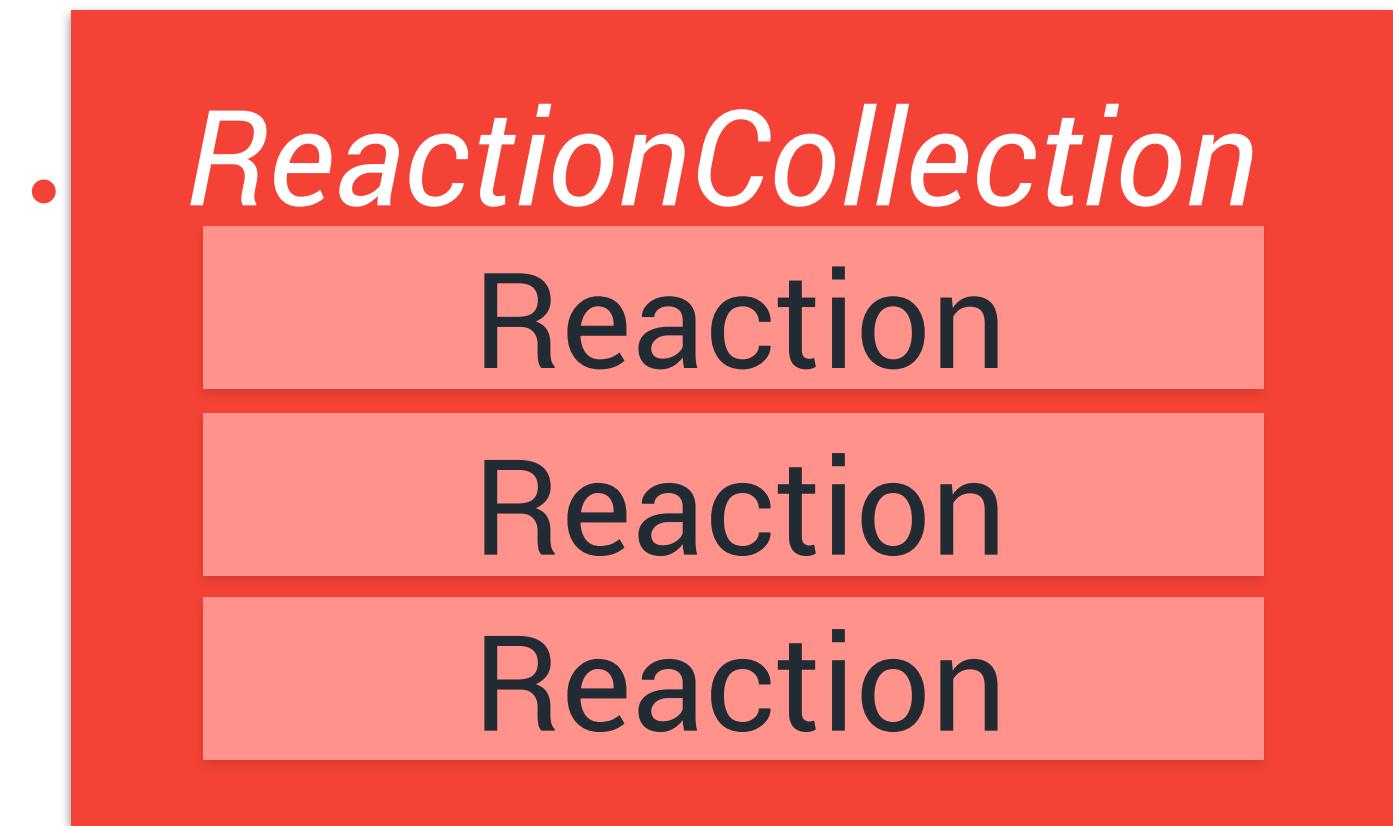
2. Receive events

1. Colliders & Event Triggers

3. Manage events

1. *The Event System*

## 1. To recap the structure of **Interactables**...



## Guard Interactable Glasses and Coffee

Has Glasses

Has Coffee

*Glasses and Coffee*

Glasses

Has Glasses

*Glasses Reaction*

*Default Reaction*

*Glasses and Coffee*

Audio Reaction

Text Reaction

Condition Reaction

## *The Interactable Script*

1. **Navigate to the *Scripts > MonoBehaviours > Interaction* folders**
2. **Create a new C# script**
3. **Name it Interactable**
4. **Open the Interactable script for editing**

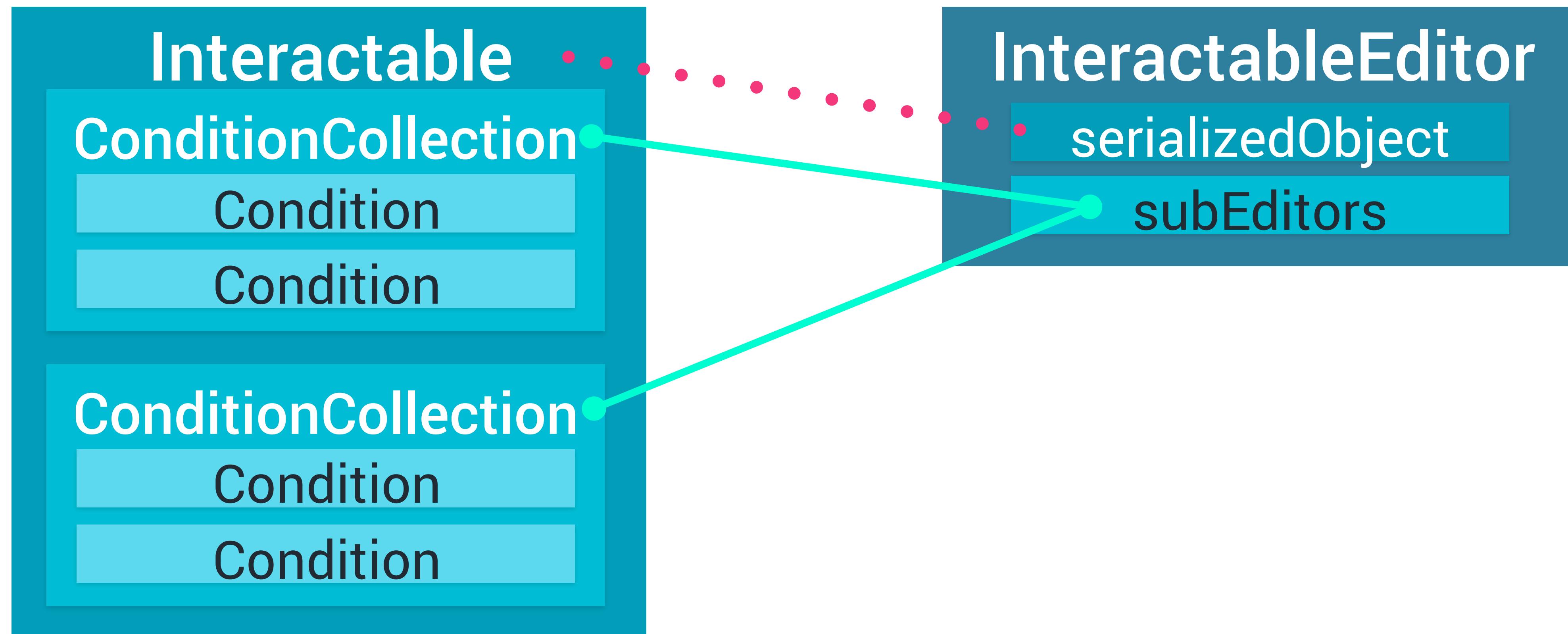
## 1. *Save the script and return to the editor*

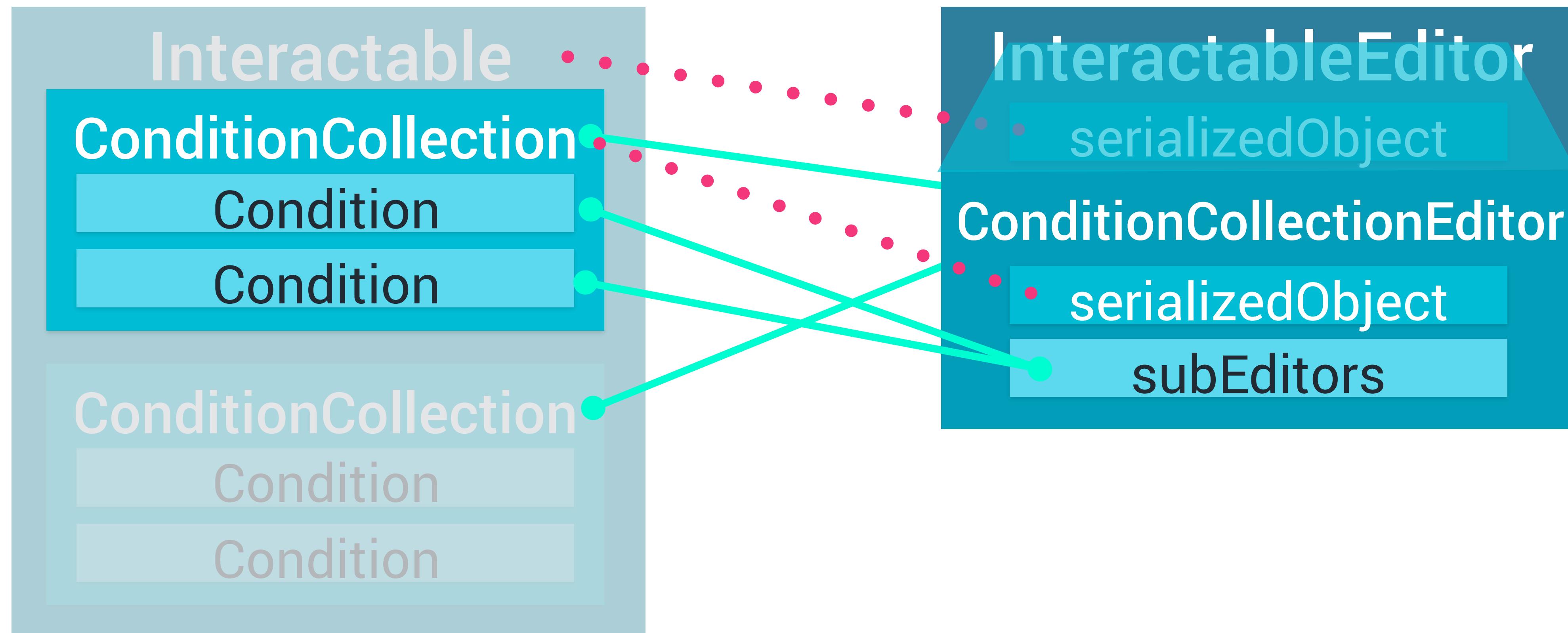
## *The PlayerMovement Script*

1. **Expand the *Scripts > MonoBehaviours > Player***
2. **Open the *PlayerMovement* script for editing**

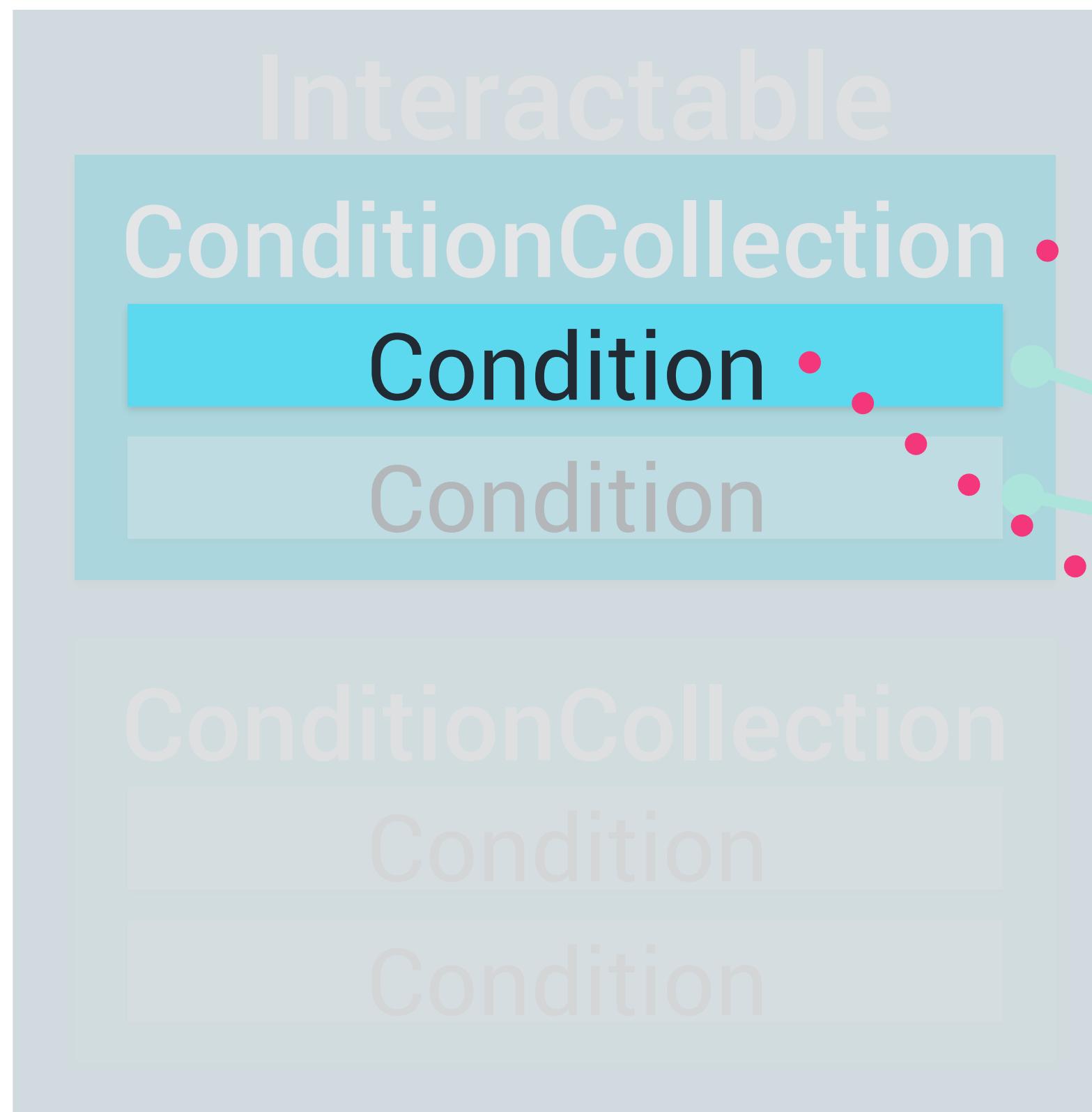
## 1. *Save the script and return to the editor*

*The EditorWithSubEditors Class*





# Interactables



InteractableEditor

Phase 5/6

serializedObject

subEditors

ConditionCollectionEditor

serializedObject

subEditors

ConditionEditor

serializedObject

## *The InteractableEditor script*

1. **Navigate to the *Scripts > Editor > Interaction* folder**
2. **Open the `InteractableEditor` script for editing**

## 1. *Save the script and return to the editor*

## *The Interactable Prefab*

1. If it is not already open, **Navigate to the Scenes folder** and **open the SecurityRoom scene**
2. **Create an empty GameObject**
3. **Name it Interactable**

1. Add a **BoxCollider** component
2. Add an **EventTrigger** component
3. Add a new **Pointer Click Event Type** to the **Event Trigger** component
4. Add a new **event** to the **Pointer Click event list**
5. Add an **Interactable** component

1. **Create two children of the Interactable GameObject**
2. **Name the first child *InteractionLocation***
3. **Name the second child *DefaultReaction***
4. **Select the DefaultReaction GameObject**
5. **Add a ReactionCollection component**

1. Select the **Interactable** GameObject
2. Assign the **InteractionLocation** to the **Interaction Location** field on the **Interactable** component
3. Assign the **DefaultReaction** to the **Default Reaction** field on the **Interactable** component

- 1. Drag the Interactable GameObject from the hierarchy into the Prefabs folder**

## *Creating The PictureInteractive*

1. Rename the **Interactable** GameObject to **PictureInteractive**
2. Set the position to **3.18, 0, 0.24**
3. On the **BoxCollider** component, set the **Center** to **0, 2.5, 0**
4. On the **BoxCollider** component, set the **Size** to **0.25, 2.3, 3.8**

1. On the **Pointer Click** event of the **EventTrigger**, drag on the **Player** **GameObject**
2. From the function dropdown select **PlayerMovement > OnInteractiveClick**
3. Drag the **Interactable** component onto the **object** **parameter field**

1. **Select the InteractionLocation child GameObject**
2. **Set its position to -1.5, 0, 0**
3. **Set its rotation to 0, 90, 0**

1. **Select the DefaultReaction child GameObject**
2. **Add a TextReaction and an AudioReaction to the ReactionCollection component**
3. **Set the Message of the Text Reaction to “*He looks pretty trustworthy.*”**
4. **Set the Audio Source of the Audio Reaction to VO**
5. **Set the Audio Clip of the Audio Reaction to *PlayerThisGuyLooksTrustworthy***

1. ***Save the scene***
2. **Open the Persistent scene**
3. **Test**
4. ***Exit Play Mode!***

*End of*  
**Phase 05**



# Phase 06

## *Game State*

*Download the Asset Store package:*  
**6/6 - Adventure Tutorial - Game State**



# *The Brief*



*Brief*

## Create a system to transition between scenes

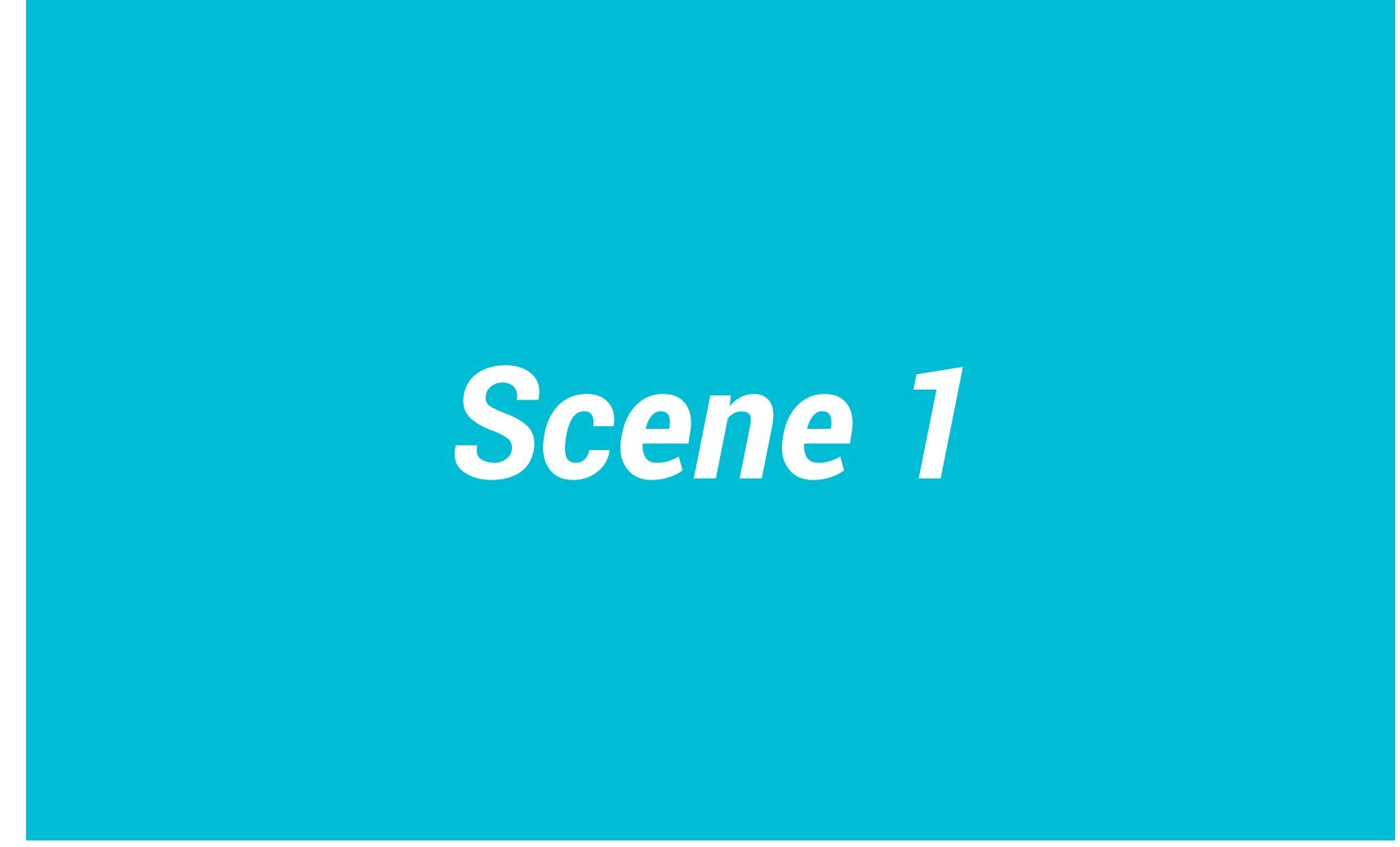
- Important information must be retained between scenes (eg: inventory state)



*Brief*

## Create a system to transition between scenes

- Fade to black during scene transitions



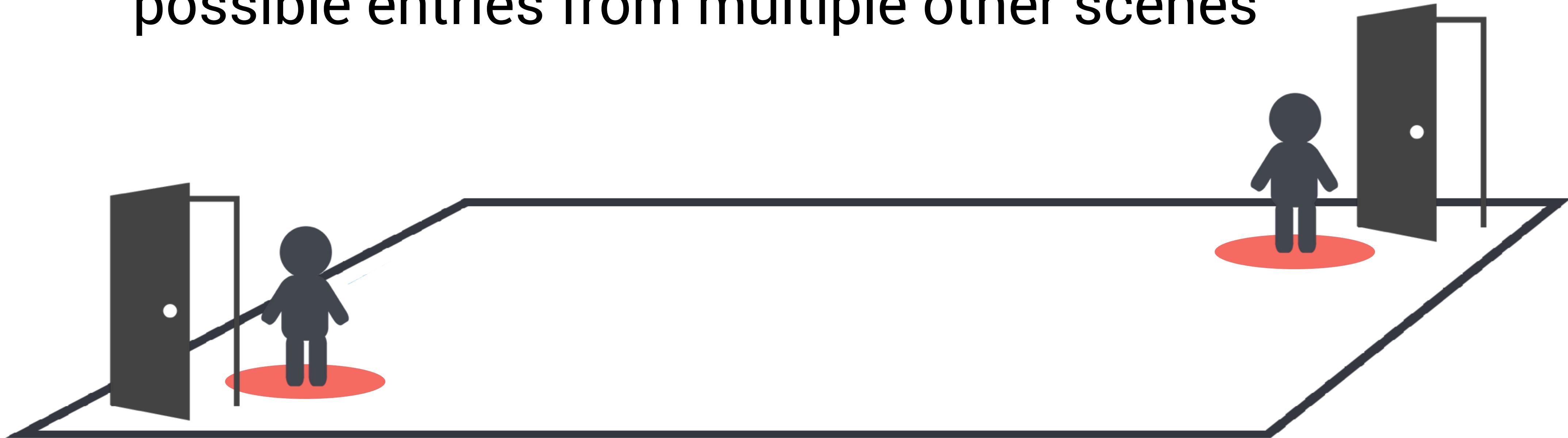
*Scene 1*

*Scene Loading...*

## Brief

### Create a system to transition between scenes

- Multiple spawn points need to be supported due to possible entries from multiple other scenes



## *Brief*

### **Create a system to save data between scene loads and unloads**

- Upon unloading a scene information such as the position of a GameObject must be stored so that on returning to the scene the information can be used again

# *The Approach*

## *Approach*

- The project architecture will be based on a main, or “persistent”, scene that will stay loaded all times

Persistent Scene

## *Approach*

- This “persistent” scene will handle the loading and unloading of other scenes

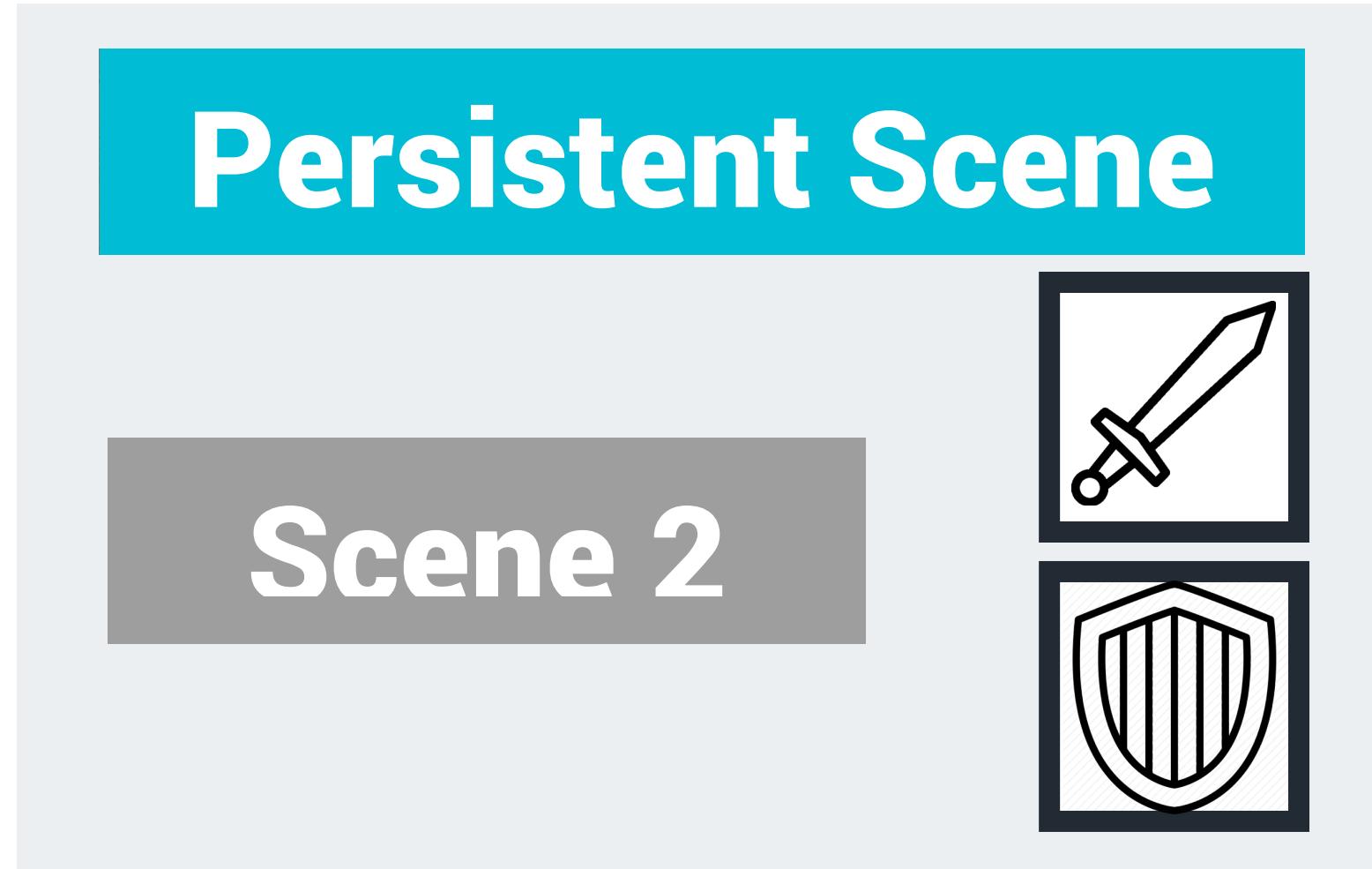
**Persistent Scene**

**Scene 1**

**Scene 2**

## Approach

- Certain information that needs to persist throughout all scenes (eg: the inventory) can be stored in the “persistent” scene



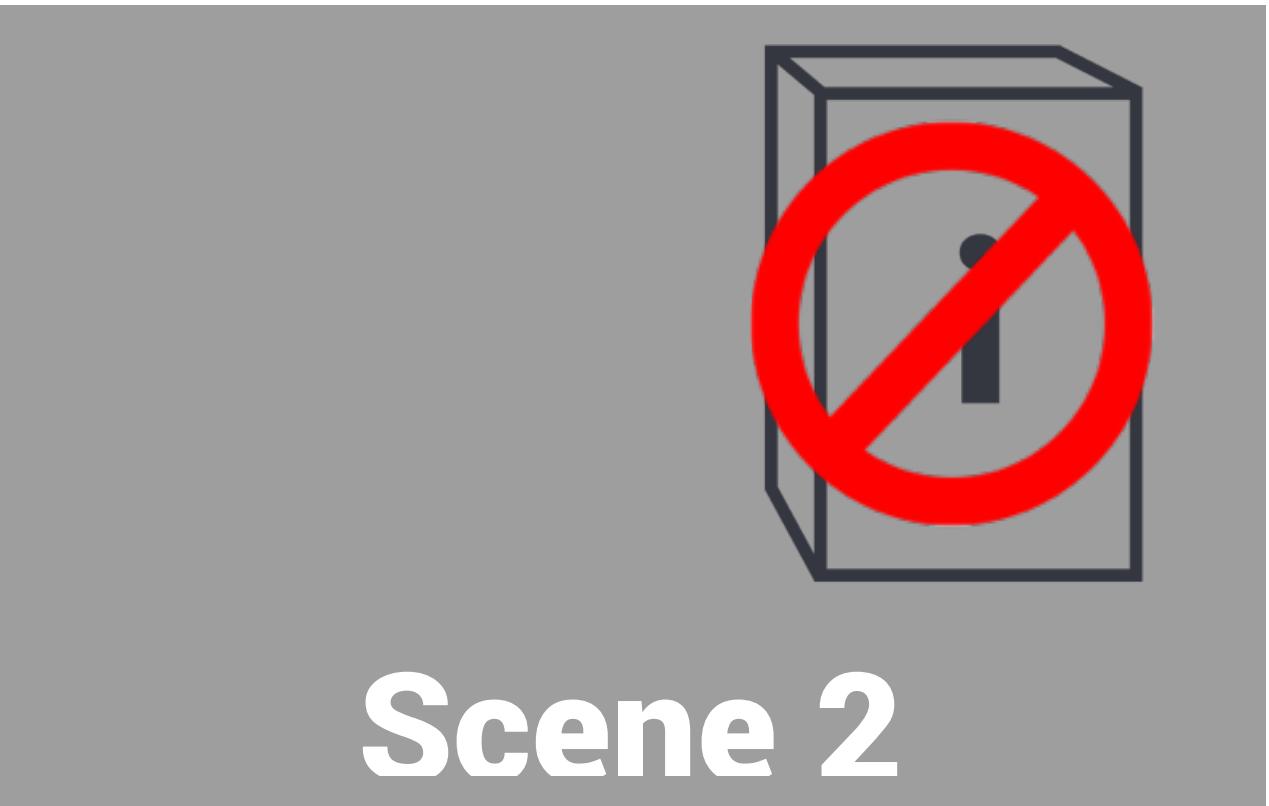
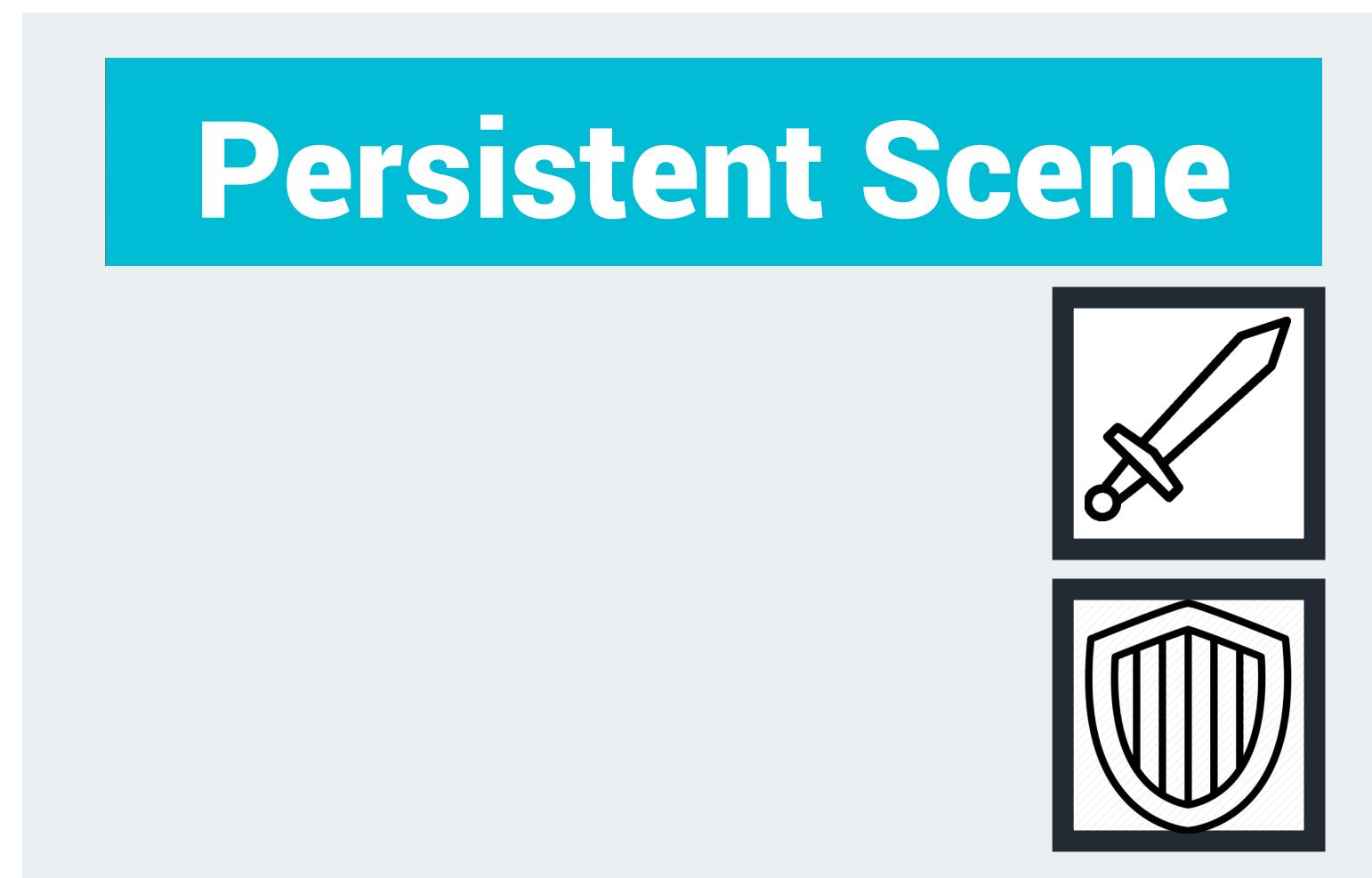
Scene 1

## Approach

- The Persistent scene is not suitable to store all information, such as details about GameObjects that exist only in one particular scene (eg: the bird)



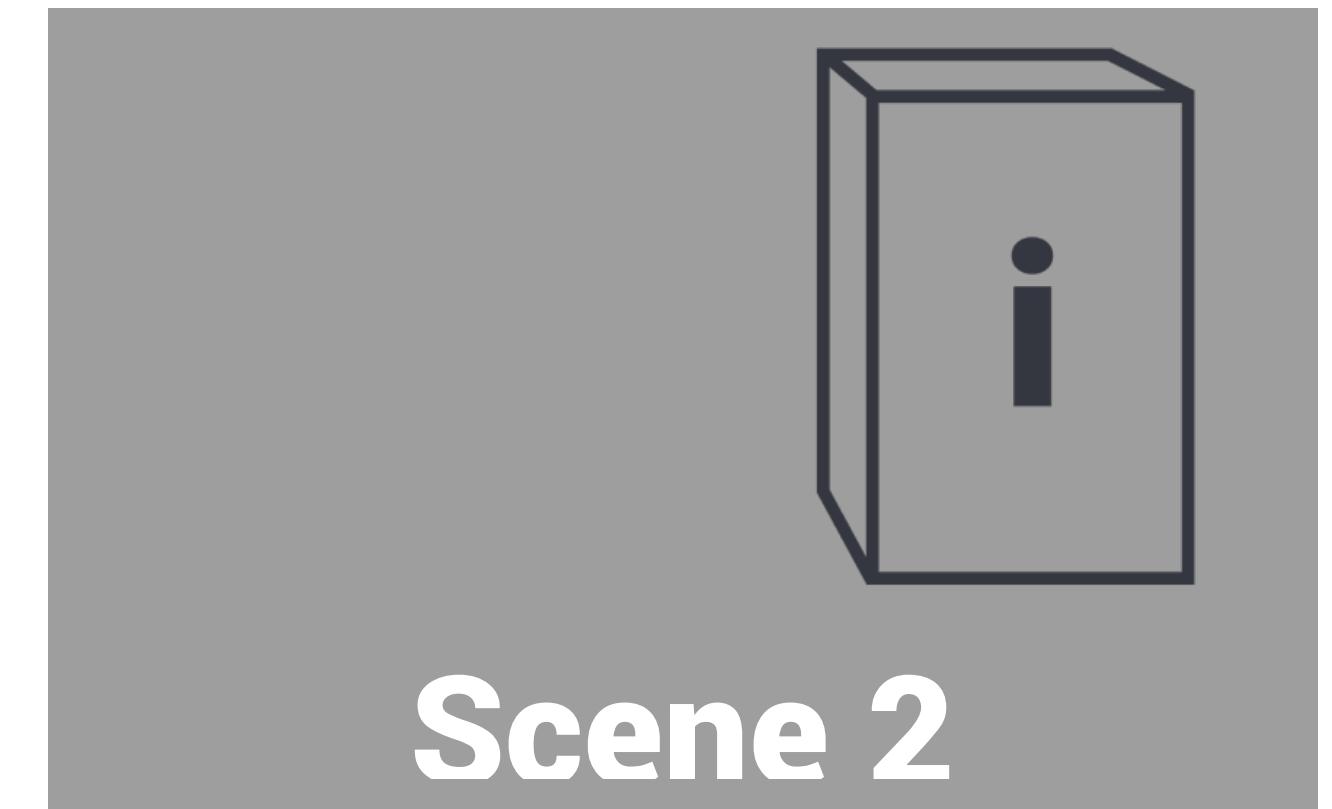
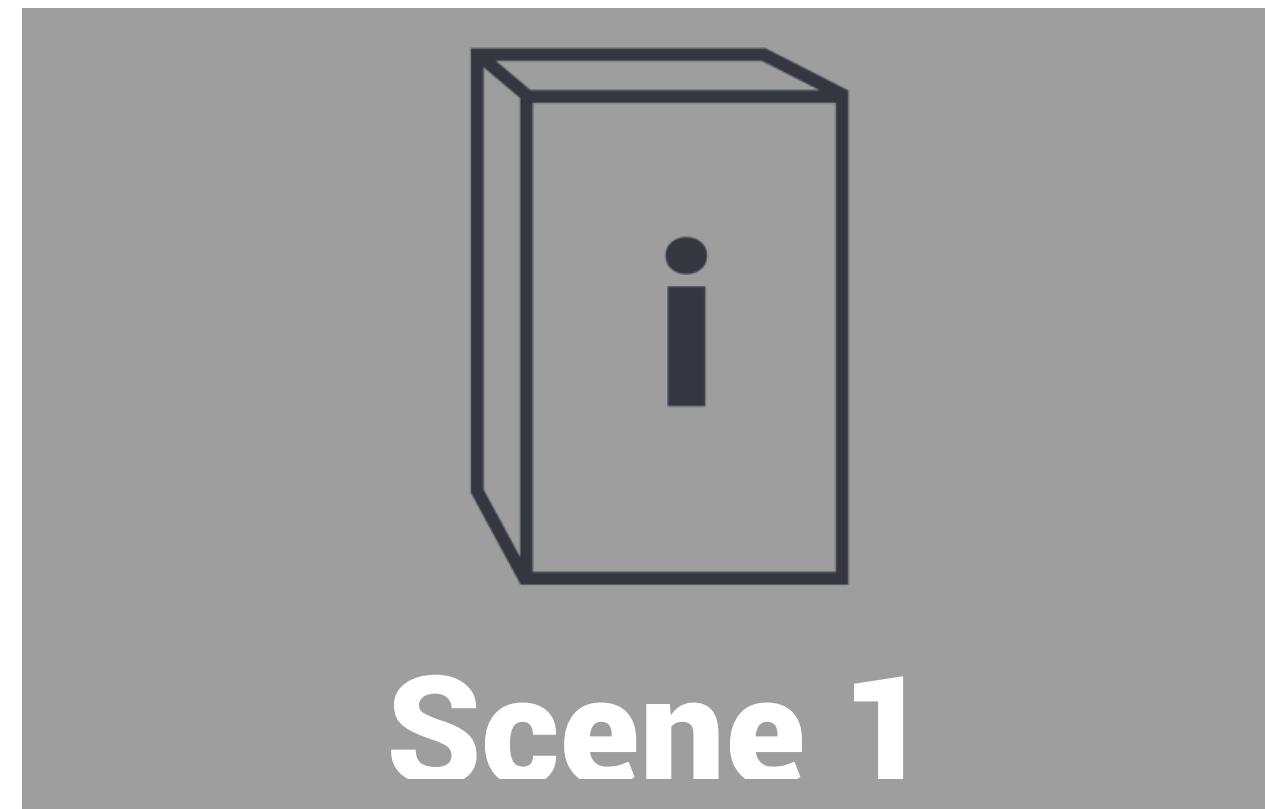
**Scene 1**



**Scene 2**

## Approach

- ScriptableObject assets will be used to *temporarily* store this data so that it can be loaded again on return to a scene



## *The SceneController Architecture*

**Show on screen**

Scene 1

**Loaded in memory**

Persistent

Scene 1

**Show on screen**

Scene 1

**Loaded in memory**

Persistent

Scene 1

**Show on screen**

Scene 1

**Loaded in memory**

Persistent

Scene 1



**Show on screen**

Scene 2

**Loaded in memory**

Persistent

Scene 2



**Show on screen**

Scene 2

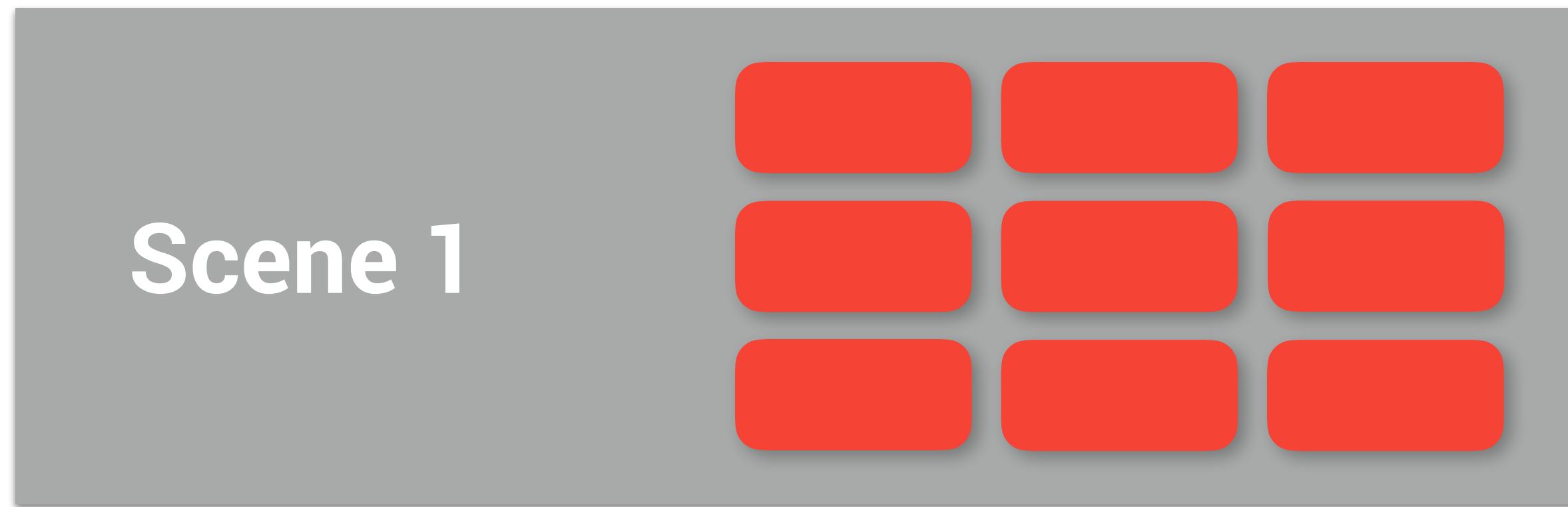
**Loaded in memory**

Persistent

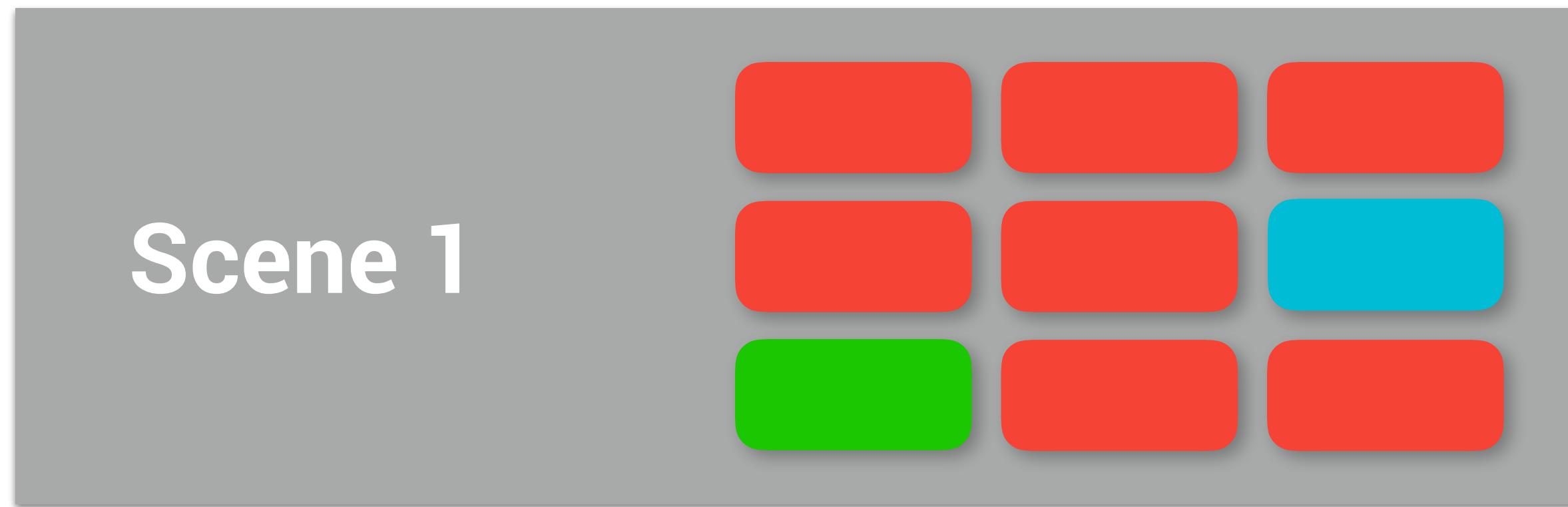
Scene 2

## *The SaveData Architecture*

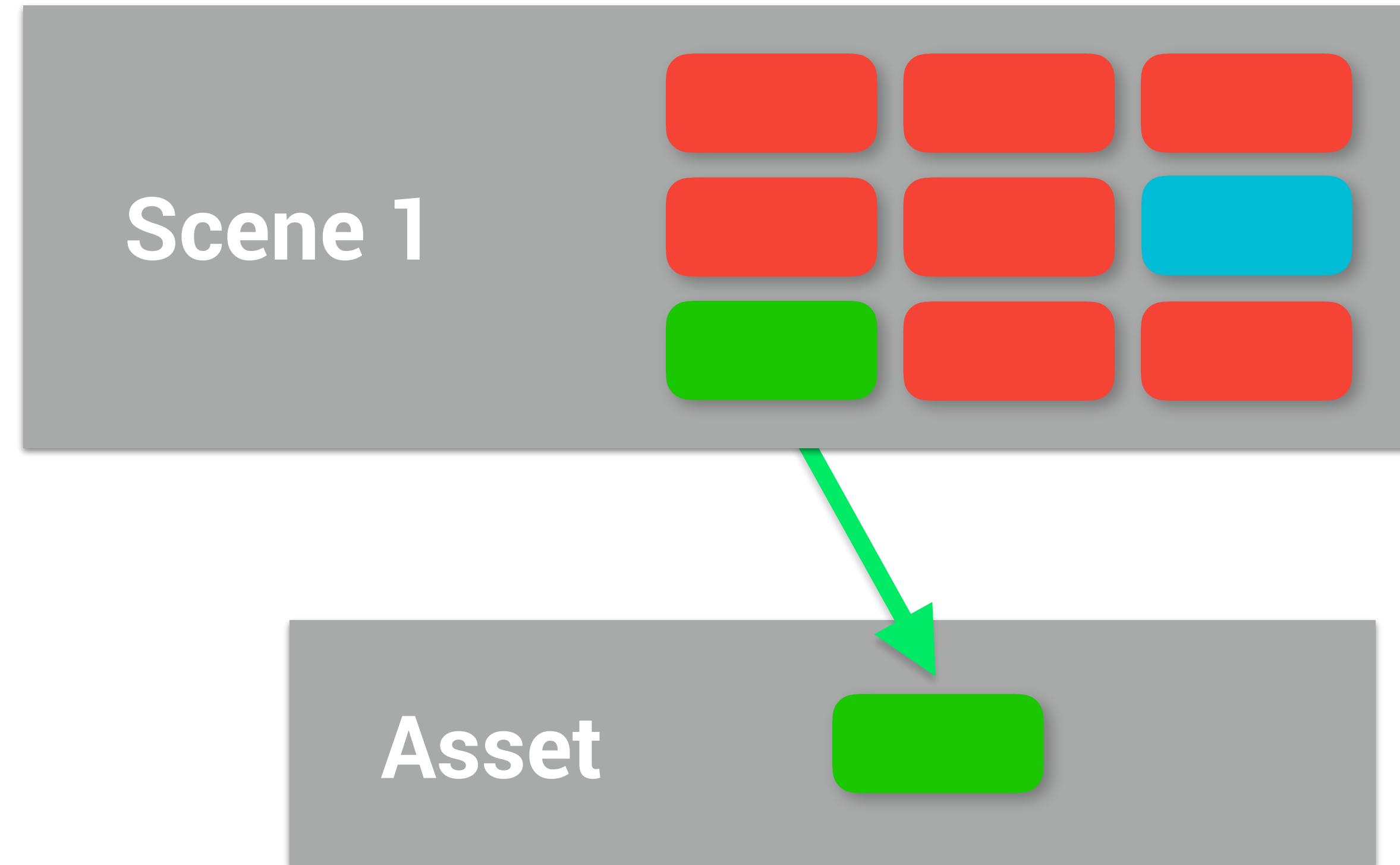
Scene 1 is loaded



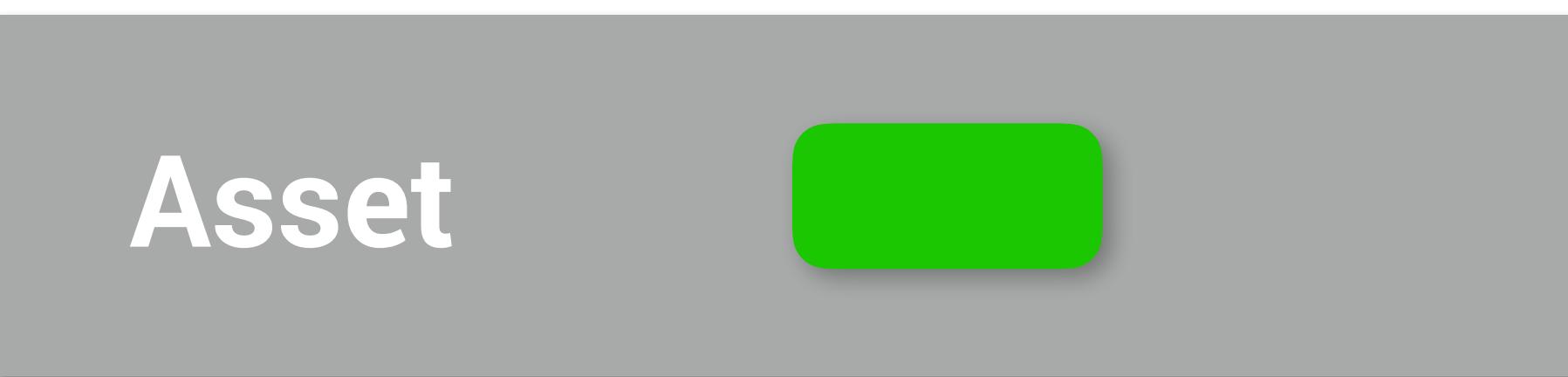
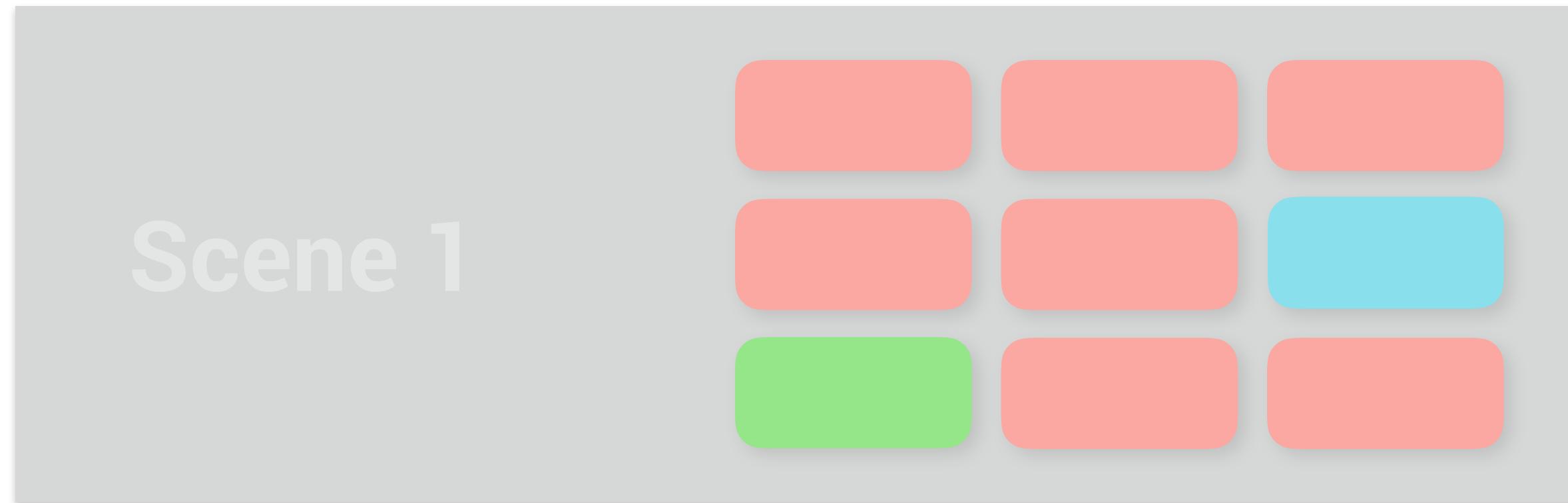
Scene 1 changes state



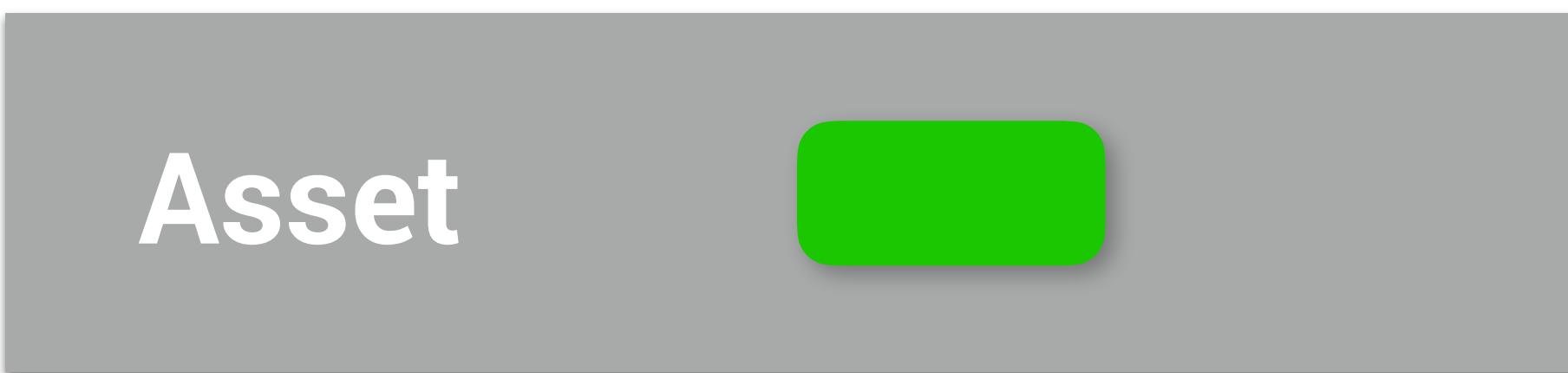
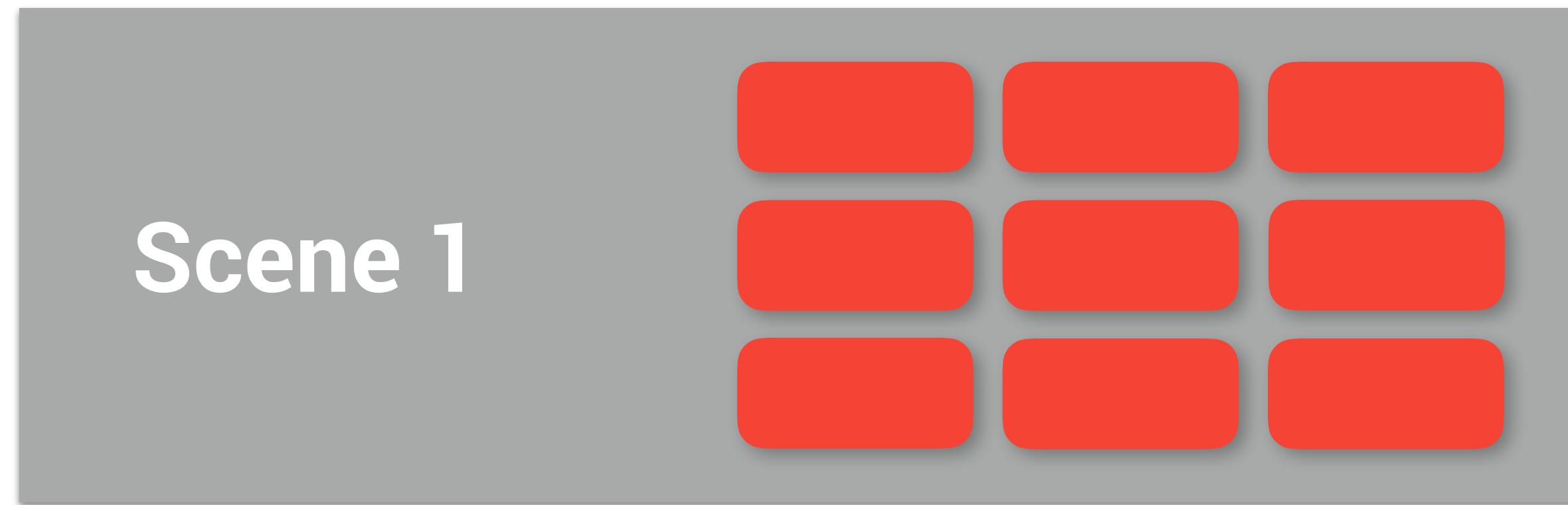
Scene 1 saves some changes to an asset



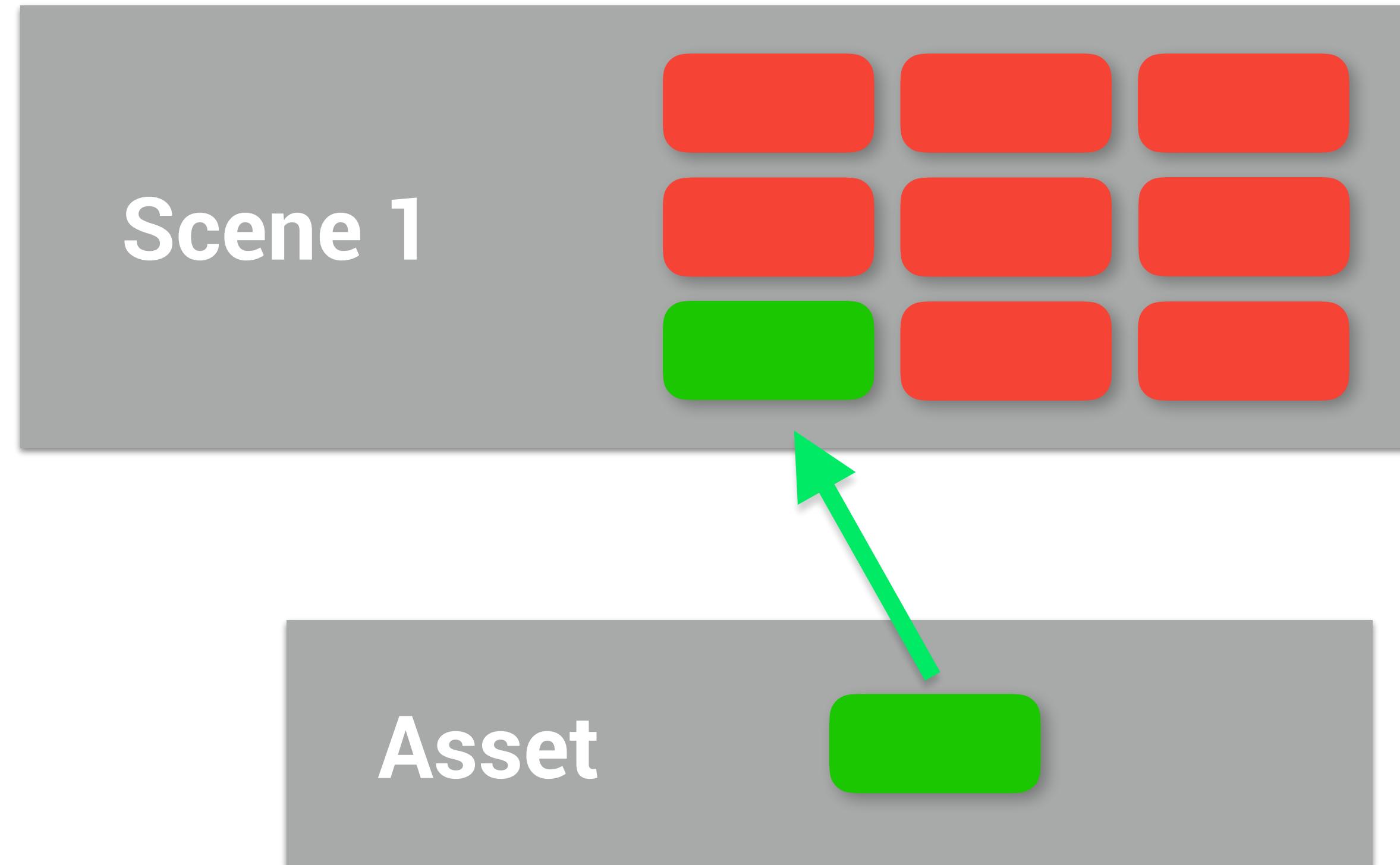
Scene 1 is unloaded



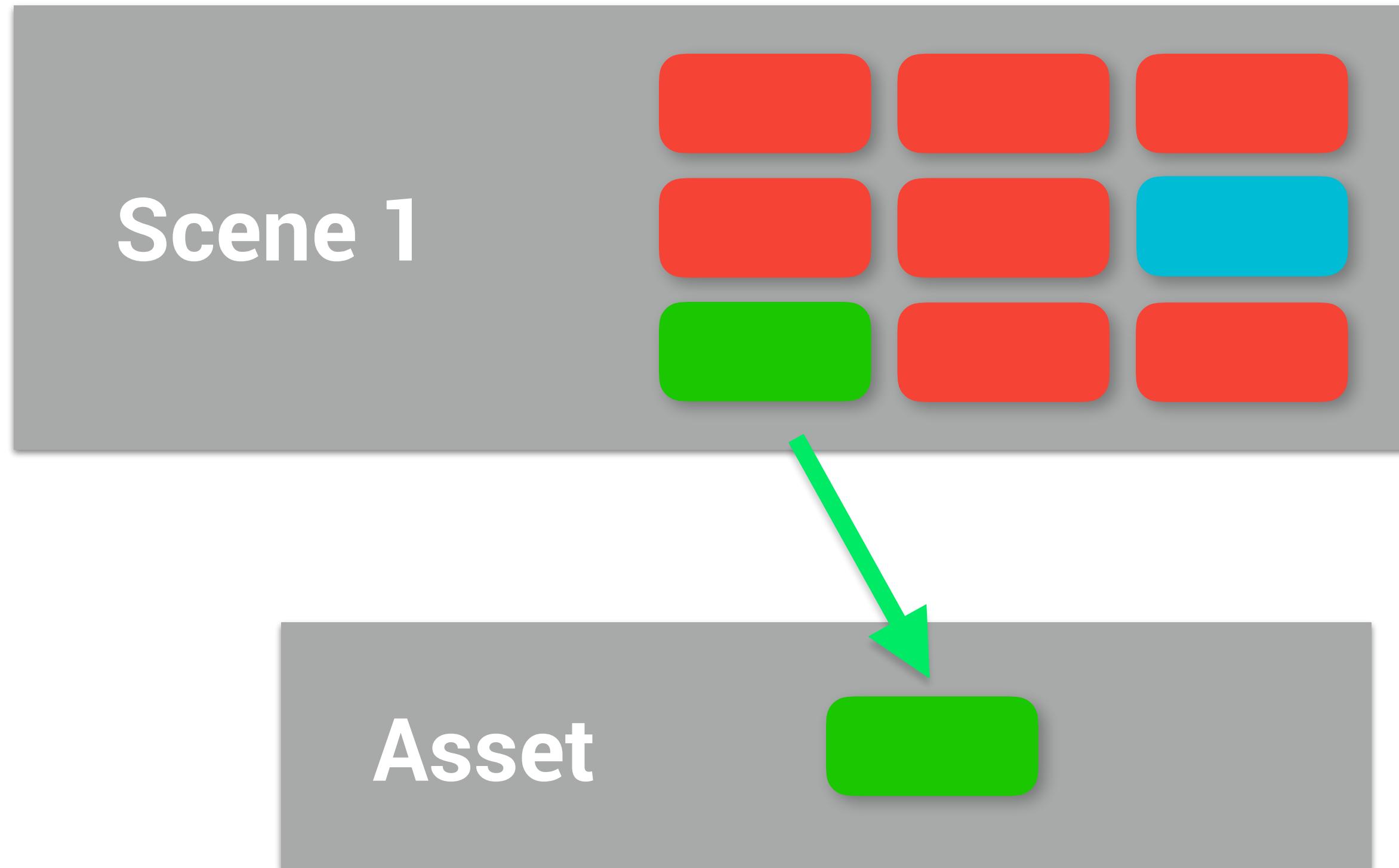
Scene 1 is reloaded



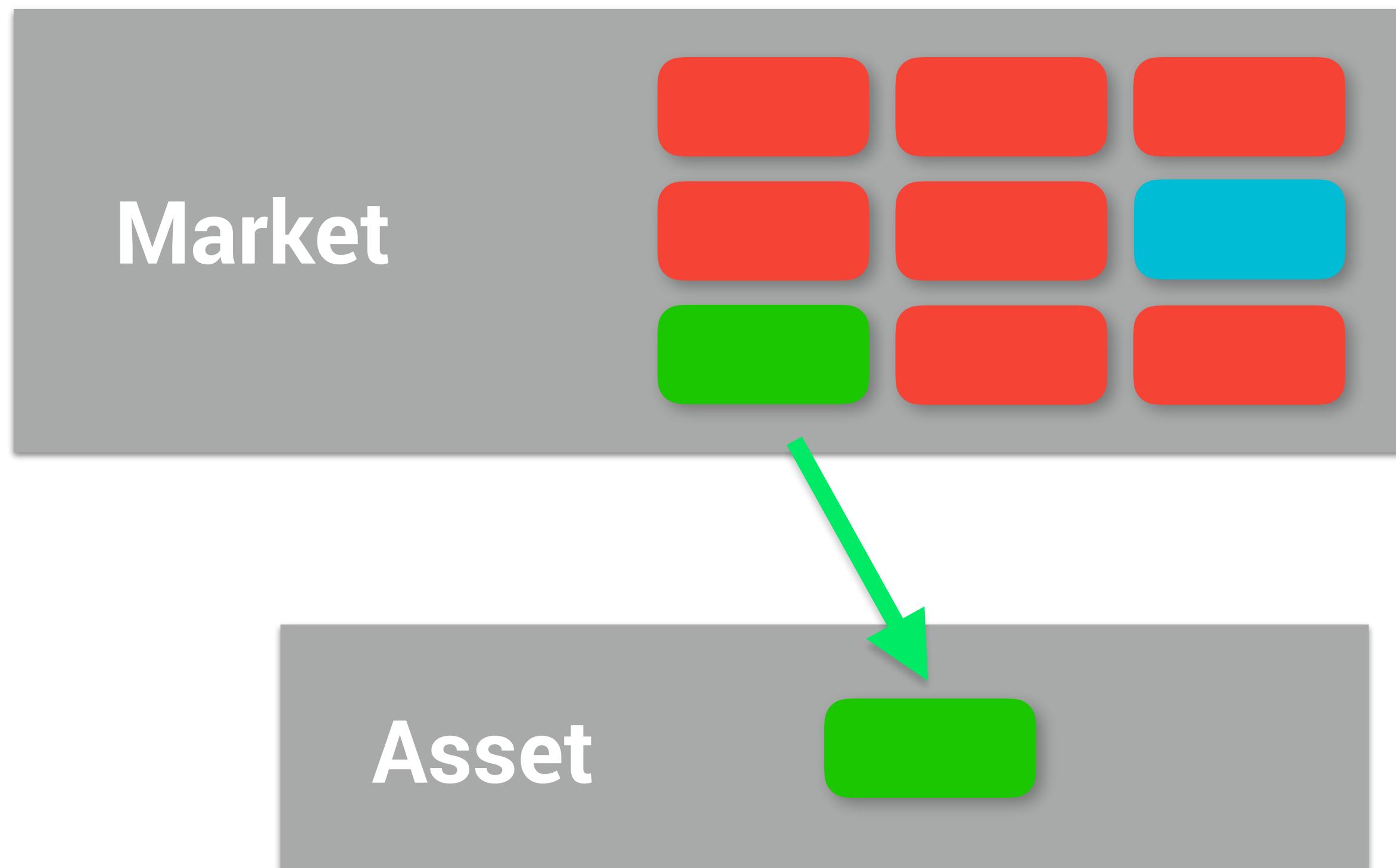
Scene 1 loads some data from an asset



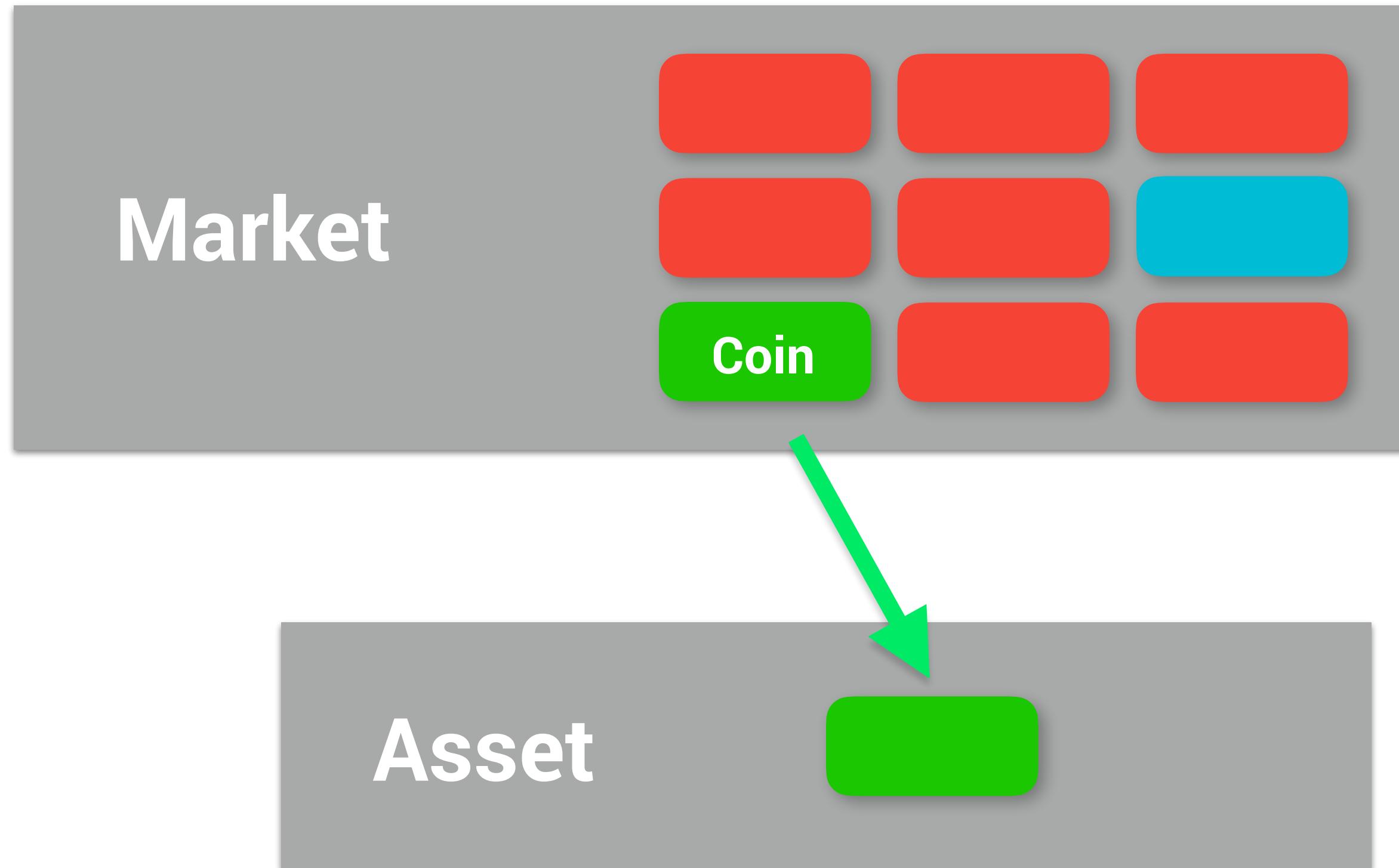
## Specific save data example



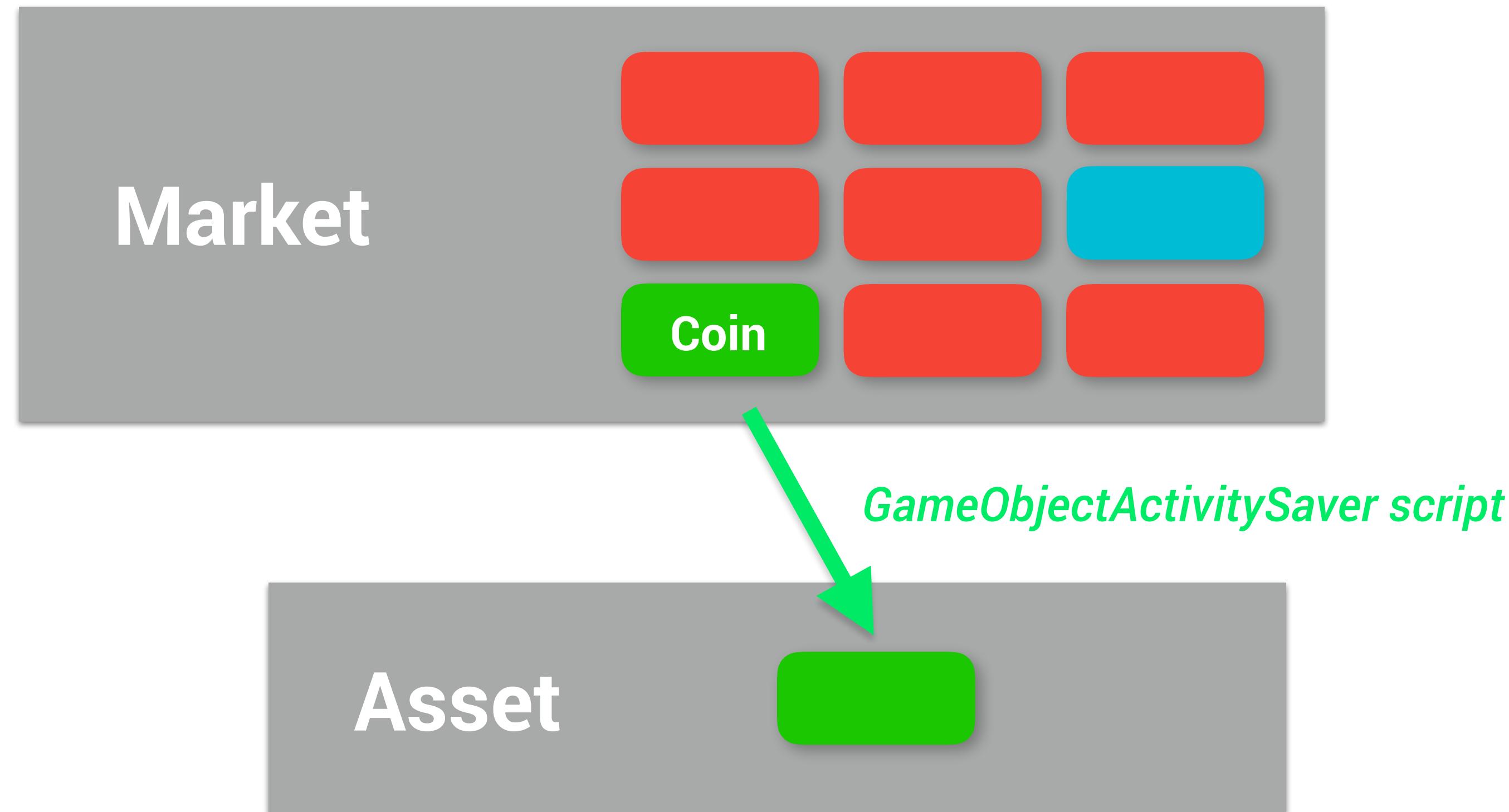
## Specific save data example



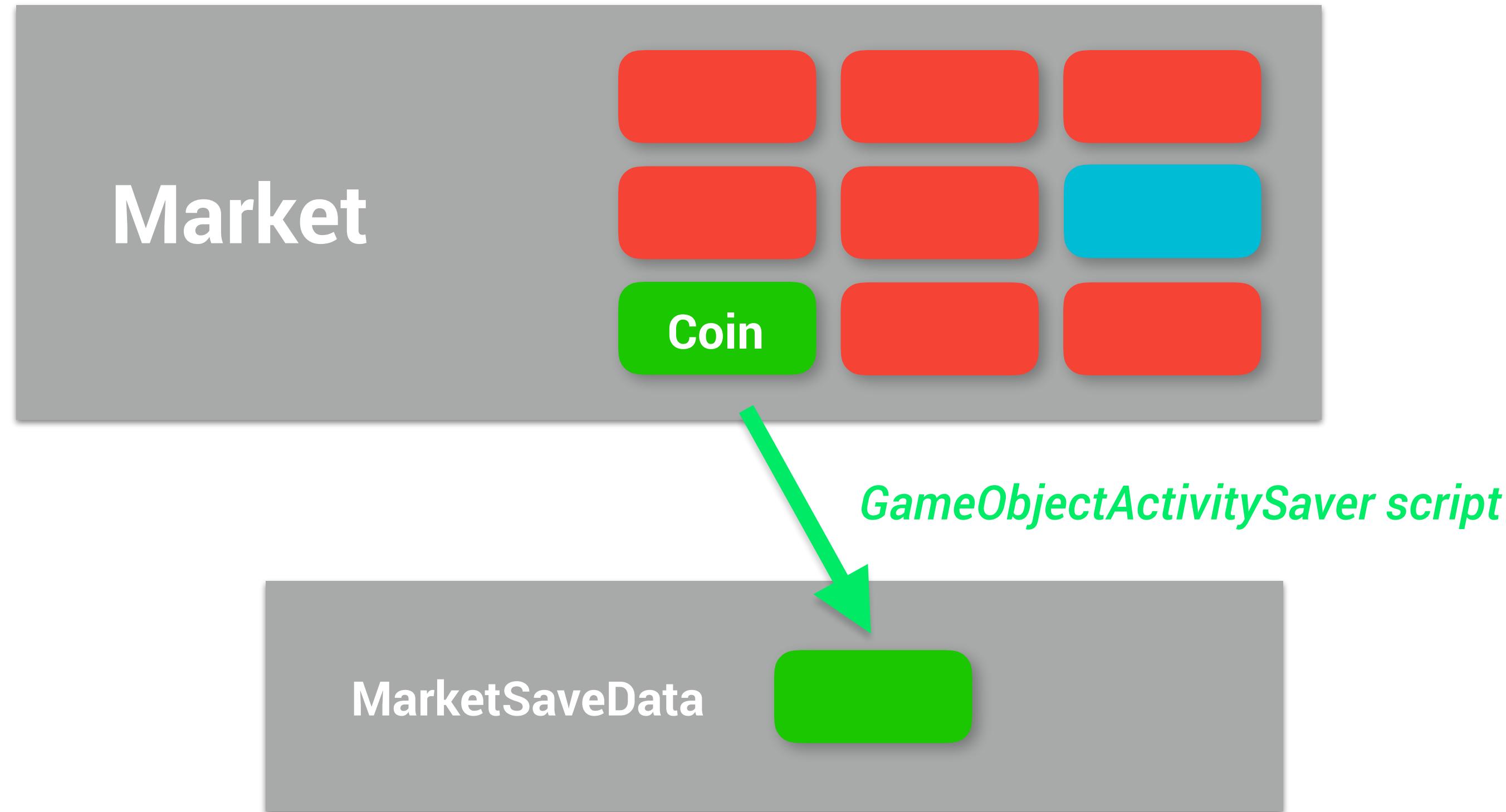
## Specific save data example



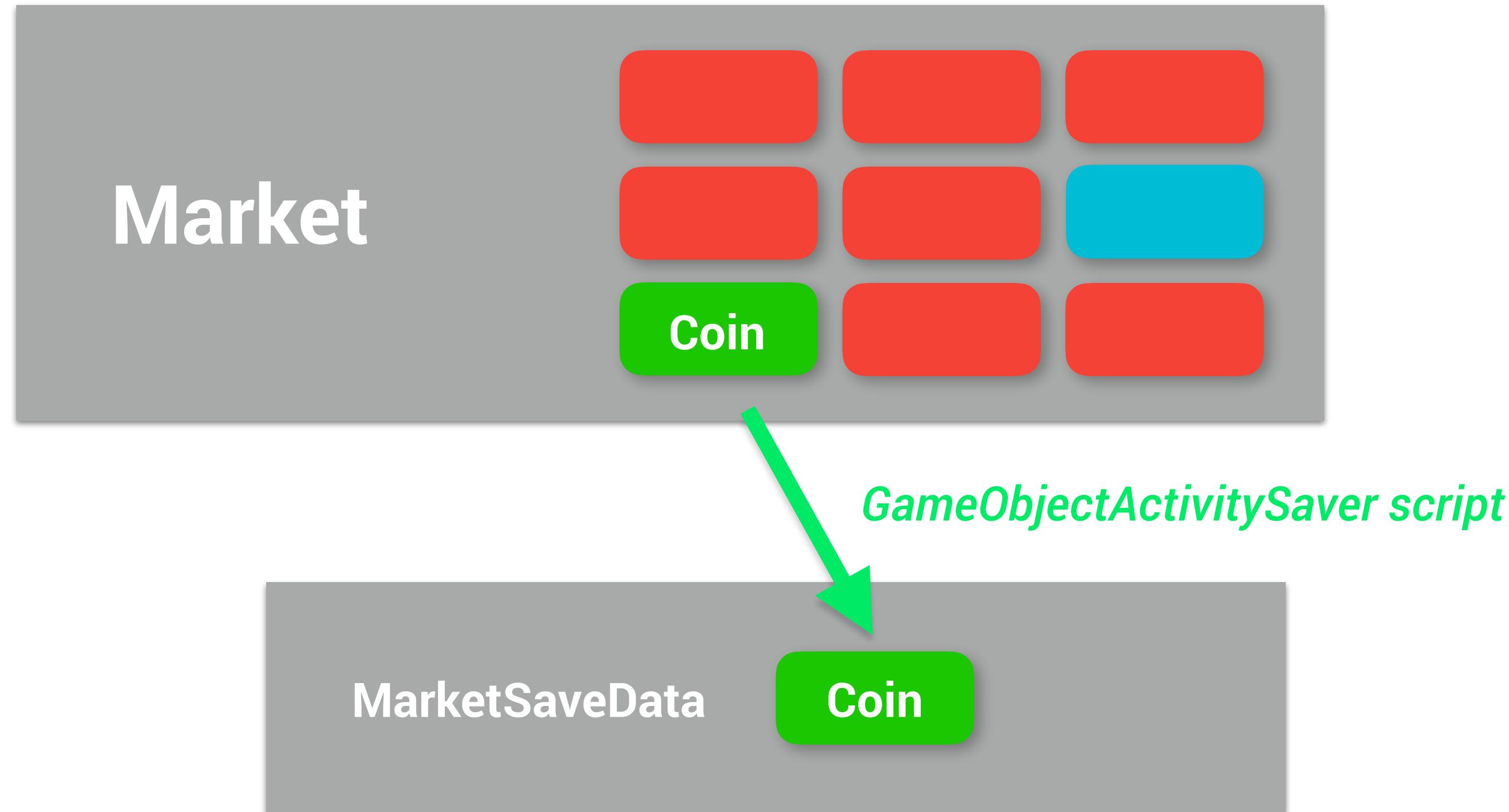
## Specific save data example



## Specific save data example



## Specific save data example



# *The Steps*



*Delegate, Events and Lambda Expressions*

1. Please import the Asset Store package:

**6/6 - Adventure Tutorial - Game State**

## Delegates

1. Delegates are variables that can store *functions* instead of *data*

## Delegates

1. Delegates are variables that can store *functions* instead of *data*
2. The basic **Action** is a specific type of **Delegate** that represents a **function** that returns **void** and has no **parameters**

## Delegates

1. Functions are stored in **delegates** as follows:

```
void MyFunction ()  
{  
}
```

```
Action myAction = MyFunction;
```

## Delegates

1. The functions stored in **delegates** can be called as follows

```
myAction();
```

## Delegates

1. Actions also come in generic types

```
Action<int> myIntegerAction;
```

2. The above **Action** can store a function that returns void and takes in a single integer parameter as below

```
public void MyFunction (int myInt)  
{  
}
```

```
Action<int> myIntegerAction = MyFunction;
```

## Delegates

1. One of the most important things about **delegates** is that they are stored like **variables**, as such they can be passed to **functions** like a **variable**

```
public void MyFunction (Action myActionFunction)  
{  
}
```

## Delegates

1. A **delegate** can be **subscribed to** by using the **+ =** operator and **unsubscribed from** using the **- =** operator

```
myAction += MyFunction;  
myAction -= MyFunction;
```

## Events

1. *Note that the events we are about to discuss are C# events which have nothing to do with the Event System in Unity and are solely a scripting feature*

## Events

1. A **delegate** can be used as an **event** by putting the keyword 'event' in its definition

```
public event Action myEventAction;
```

## Events

1. It is very important that once an **event** has been **subscribed to**, that it is **unsubscribed from** before the object is destroyed

## Events

1. It is very important that once an **event** has been **subscribed to**, that it is **unsubscribed from** before the object is destroyed
2. **Events** that are still subscribed to will not be **Garbage Collected** and will therefore cause **memory leaks** if they are left subscribed

## Events

1. Calling **events** works the same as calling **delegates**, this will call all of the **functions** which have subscribed to the **event**

```
public event Action myEventAction;
```

```
myEventAction += MyFunction;
```

```
myEventAction();
```

## Events

1. The key point about **events** is that the **triggering class** does not need to tell all **concerned classes** to call a specific function
2. Instead, **concerned classes** can **listen** for an **event** and then act accordingly

## Lambda Expressions

1. **Lambda expressions** are a short-hand way of writing functions that can be stored in **delegates**
2. They have the following syntax:

'variable names' => 'expression(s) using those variables'

## Lambda Expressions

1. The variables declared for a lambda often very short
2. Variables are usually implicitly typed, for example:

```
x => x == someValue
```

3. In the above case, x implicitly has the same type as someValue

## Lambda Expressions

1. Lambda expressions are generally quite short
2. Here are some examples of delegates using Lambda Expressions

```
Action myAction = () => Debug.Log("something");
```

```
Action<int> myAction = x => x * x;
```

## Lambda Expressions

1. When expressions are longer or there are multiple expressions, braces are used

```
Action<string> myAction = name =>
{
    name.ToUpper();
    print (name);
};
```

## Lambda Expressions

1. One key reason to use lambda expressions is to simplify our code

## Lambda Expressions

This:

```
Action<string> myAction = SomeFunction;  
  
private void SomeFunction (string name)  
{  
    print(name);  
}
```

Becomes this:

```
Action<string> myAction = name => {print (name);};
```

*SceneController Script*

- 1. Navigate to the Scripts > MonoBehaviours > SceneControl**
- 2. Open the SceneController script for editing**

## ***1. Save the script and return to the editor***

## *SaveData Script*

- 1. Navigate to the Scripts > ScriptableObjects > DataPersistence folder**
- 2. Open the SaveData script for editing**

```
private int GetIndex (string key)
{
    for (int i = 0; i < keys.Count; i++)
    {
        if (keys[i] == key)
            return i;
    }
    return -1;
}
```

## ***1. Save the script and return to the editor***

*The Saver Script and its children*

1. Open the Persistent scene
2. Test by entering Play Mode
3. *Exit Play Mode*

*End of*  
**Phase 06**

