User Manual
v1.13

CONTENTS

CHAPTER II: ADVANCED FEATURES

CHAPTER III: EXTENDING FUNCTIONALITY

INTRODUCTION

Thank you for purchasing Adventure Creator! This is a toolkit for Unity that provides full functionality of an adventure game engine – navigation, inventory, characters, conversations, cutscenes, saving and loading and more are all possible without coding. If you want to customise the interface or integrate the kit with other systems, you will have to dive into the framework, but the code is clean and intuitive.

If you supply the graphics and animation, Adventure Creator can be told what to do with it. If you're new to Unity, it's advisable that you get to grips with the basics of the Unity interface first, since Adventure Creator is tightly integrated into it. This manual assumes a working knowledge of Unity's interface and basic concepts. The Unity website provides an excellent introduction to the interface at unity3d.com/learn.

Once you've imported the Adventure Creator package, you'll find a new option under Unity's Window menu. Choose **Window → Adventure Creator** to bring up the main interface and dock it in a vertical space. You'll need it handy for much of your game's development.

A demo game has been created to demonstrate many of the kit's features in simple ways. It is available to play in a browser at iceboxstudios.co.uk/adventure-creator/demo

The package includes the full source to this demo game. It's a good idea to spend some time studying it, as this manual makes a number of references to it to demonstrate the various steps to creating an adventure game. Refer to section 1.2 for instructions in setting up the demo.

This manual also offers an insight into the way Adventure Creator works, from a coding perspective, so that if you want to extend it, you've got a head start.

CHAPTER I: THE BASICS

1.0 - SETTING UP

1.1 - PREREQUISITES

Adventure Creator makes use of a few tags and layers that will need to be defined before your game will run properly. Before you do anything else, create the following tags in the Tags Manager: **GameEngine**, **PersistentEngine**, **FirstPersonCamera**, and **NavMesh**.

You will then need to assign these two of these tags to the appropriate prefabs. From within the Assets → AdventureCreator subdirectory, tag the listed prefabs as follows:

- Prefabs/Automatic/GameEngine – Tag as **GameEngine**
- Resources/PersistentEngine – Tag as **PersistentEngine**
- Demo/Resources/Tin Pot/First Person Cam – Tag as **FirstPersonCamera**

Next, the Input Manager will need to be updated. Which axes you will need to define depend on what control method you decide to make use of – refer to the Input And Navigation section (2.0) for a rundown of the various requirements. At it's most basic, you need to define an axis called **Menu**. This defines which key brings up the in-game pause menu - traditionally the Escape key. The Horizontal and Vertical axes are also required, but these are created by default by Unity when starting a new Unity project.

Adventure Creator relies on the Legacy animation system, so you'll need to make sure you can provide legacy Animation Clips of your characters and animated objects.

It's also important to keep objects on the correct layer. Most objects, unless stated otherwise, should be moved to the **Ignore Raycast** layer, especially if you intend to make a Point and Click-style game, as you don't want scene geometry to interfere with your interface.

To create and run a game, you will also need a References file and manager assets. These assets are specific to your game. The default managers refer to the demo game's set, so in order to create your own game, you will also have to create your own set of managers. Refer to section 1.3 for more on managers.

1.2 - RUNNING THE DEMO

The included demo is a short single-scene game in which wrestling “celebrity” Brain and his robot companion Tin Pot try to film their own promotional video for Adventure Creator. You can load the game's scene file by opening **Assets → AdventureCreator → Demo → Scenes → Basement.unity** in your project folder.

If you have not made any changes to Adventure Creator since installing it, refer to the previous section and assign the tags, layers and input axes as described.

If you problems running the demo game, check that the main Adventure Creator window is referring to the demo managers. Refer to section 1.3 for details on how to do this.

To keep things straightforward, the demo is designed to be played using the Point And Click control style, but it can be updated easily if you want to play it in Direct (cursor-driven) or First Person styles. You can change the game's control style easily enough within the Settings Manager, though you will need to do a bit more to make the scene release-ready. Refer to section 2.0 for a rundown on what you'll need to do.

1.3 - THE ADVENTURE CREATOR WINDOW

Adventure Creator makes use of six "managers" that each store data for a different aspect. These managers are created and accessed within the main Adventure Creator window (Window → Adventure Creator). They are required to properly run a game made with Adventure Creator. You can tab between each manager from the top of the window. The following is a list of what each manager provides control over:

Scene manager – Handles scene-specific settings, such as the player's default position, and also provides quick-buttons to creating prefabs like Hotspots and Markers.

Settings manager – Handles project-wide settings, such as the player's prefab, the game's control style, and cursor settings.

Actions manager – Defines the list of actions available to cutscenes and interactions.

Variables manager – Manages a list of user-defined integers and booleans that can be used for game logic throughout the project.

Inventory manager – Manages a list of items that the player can pick up, as well as the responses for inventory interactions.

Speech manager – Lists the spoken lines written in the game, assigning each one a unique ID for audio files, and manages subtitle translations.

Each manager is a separate asset file. By default, a set of "demo" managers are referred to by the main Adventure Creator window, as these contain inventory items, variables and other data required by the demo.

To make a new game, without affecting the demo project, you will need to create and assign a new set of managers. This is easy to do. Each manager asset is referenced at the top of it's respective tab in the main Adventure Creator. If you remove the reference (by selecting the reference field and pressing backspace), Adventure Creator will ask if you want to create a new manager asset.

Click the "Create new" button, and Adventure Creator will create – and automatically reference – a new manager asset. It will attempt to create this asset file inside Assets → AdventureCreator → Managers, so ensure that this directory exists. Note that in order to run the demo, you must first switch each manager reference back to it's "Demo" counterpart. You can do so from the top of each tab, or from the Resources asset itself, explained below.

The main Adventure Creator window stores it's links to these manager asset files inside a file called **References**. This file MUST be placed in a folder called **Resources** in order to work properly. If such a file is not present, you will receive a prompt to automatically create one when you call up the main Adventure Creator window. Adventure Creator will attempt to create the file inside Assets → AdventureCreator → Resources, so ensure that this directory exists.

1.4 - PREPARING A NEW SCENE

With the mangers defined (see section 1.3), open the Adventure Creator window and click on the **Scene** tab. From here, you can create the GameObjects needed for an adventure game.

With a new scene, click **Organise room objects**. Adventure Creator will place the required prefabs into the scene's hierarchy. The essential objects are GameEngine, PlayerStart, and MainCamera, but a number of folders (identified with underscores) will also be created to help keep things organised. These folders are not necessary, but are helpful. As you use the Scene Manager to create Hotspots, Conversations and other Adventure Creator prefabs, it will place them into the relevant folders automatically.

You will notice a blue arrow has appeared at the centre of the scene. This is our PlayerStart object, which marks the default position and rotation of our player when the scene begins. This is our default PlayerStart because it has been set automatically by the Scene Manager, underneath **Scene Settings**. We can have multiple PlayerStart objects in a scene, so that the player can begin at different points depending on which scene they just came from. The **Previous Scene** variable in the PlayerStart's inspector dictates which scene the player must have travelled from for this PlayerStart to be used. If no appropriate object is found, or if the game is started from this scene, the default will be used instead.

We will want to build the set that will form the backdrop to our scene. The **_SetGeometry** folder is the intended place for it. Be sure to move this geometry onto the **Ignore Raycast** layer, to avoid unwanted interference with the interface – the game "discovers" hotspots and other interactive objects by seeking out objects on the **Default** layer.

With the set in place, we now need to define the space that the player can move around. Even in a Point And Click-style game, we need to place a floor down to stop the player from falling through the scene. Click **CollisionCube**, and a blue cube will be created in the scene. Manipulate the cube's transforms so that the top face covers the set's floor. It can extend beyond the floor in the X and Z directions – this object is purely a "barrier" to prevent the player from falling.

Now we will want a Navigation Mesh, or NavMesh. NavMeshes are 3D geometry that dictate where the player can navigate to inside the set. Click **NavMesh**, and assign a mesh to it's Mesh Collider component. You can use a simple cube, but a custom piece of geometry will be probably be needed. You can examine the demo game's NavMeshes in Assets → AdventureCreator → Demo → Models → NavMesh.

Underneath **Scene Settings** in the Scene Manager, assign this NavMesh to **Default NavMesh** – this will make sure that – out of any NavMesh objects present – only this one will be on the correct layer at runtime. Refer to section 2.6 for more on NavMeshes.

If we want to play a cutscene or run some kind of logic when the scene begins, we create a Cutscene prefab and refer to it in the **Cutscene on start** field, just under the Default NavMesh field. See section 4.3 for more on cutscenes.

We also need to define our initial camera, or GameCamera. GameCameras are separate from our

MainCamera, in that they dictate the available positions that our MainCamera can take. Only the MainCamera is active throughout the scene, but it will copy it's associated GameCamera's position, rotation and field of view. Click **GameCamera** in the Scene Manager, and position the newly-created camera to a point you're happy with. Next, find your default PlayerStart object - you can do so quickly by clicking on the **Default PlayerStart** field to highlight it. In the PlayerStart inspector, click the **Camera On Start** field and select your new GameCamera. When the scene begins, the player will appear at this PlayerStart, and the MainCamera will appear at this GameCamera.

Lastly, we need to define our player prefab, and set our control style. You can define both of these things in the Settings Manager. Click the **Settings** tab and change the options to your liking. For details on how to set up a Player prefab, see section 3.1. Do not place the Player prefab into the scene – Adventure Creator will do this for you when the game starts.

2.0 - INPUT AND NAVIGATION

2.1 - OVERVIEW OF CONTROL STYLES

Adventure Creator comes with three available control styles: Point And Click, Direct, and First Person. You can choose between them in the Settings Manager. Note that the control style only affects how the player can navigate the scene – interacting with hotspots and characters is still cursor-driven.

Three input types are also available: Mouse And Keyboard, Controller, and Touch Screen. Any Controller input is available for all three control styles, but the axes that you must define in the Input Manager are very specific. The following sections detail which axes are needed for each setting.

2.2 - POINT AND CLICK CONTROL

Point And Click control is the default control style. Most adventure games, like Monkey Island and The Longest Journey are controlled in this way. If you left-click your cursor in the scene but not over an interactive object, the player will make their way there. Double-clicking will make the player run, if they are able to.

To make a traditional mouse-driven point and click game, you do not need to define any axes beyond the Menu key described in section 1.1.

This style makes heavy use of pathfinding to move the player around the scene, so it's essential that your Player prefab has the **Paths** component attached. You will need to define a NavMesh for every scene – see section 2.6 for more information.

You will also need to create at least one **CollisionCube** in every scene to act as a floor (see section 1.4).

2.3 - DIRECT CONTROL

Direct control allows you to control the player's movement directly, with either the keyboard or a controller. Telltale's The Walking Dead is a recent example of a game that employs this control style.

For non-Touch Screen input, the **Horizontal**, **Vertical** and **Run** axes must be defined in the Input Manager for Direct control to work. Unity should have already defined the first two by default upon starting a new project. If not, refer to Unity's documentation on how to set them up. The Run axis should be a keystroke, such as left shift, that you can hold down to make the player run.

For Touch Screen input, Direct control works by touching on the screen and dragging without letting go, similar to how Telltale's Tales Of Monkey Island plays. The drag distances required to make the Player character walk and run can be set in the Setting Manager.

When creating interactions and moving the player in cutscenes, you will probably need to make use of pathfinding, even if you are not making a Point And Click game, so follow the same steps to create a NavMesh, assign a Paths component to the Player prefab, and add a CollisionCube floor as outlined in the previous section.

When under Direct control, the player does not take notice of the NavMesh. Instead, the player is bounded by **CollisionCube** and **CollisionCylinder** objects, which act as invisible walls to prevent the player from clipping through the set. Use the Scene Manager to populate the scene with these colliders. To avoid clutter, you can also use the Scene Manager to hide these objects once you've finished placing them. Click **Off** next to Collision under the Visibility section to hide them. The same can be done with Hotspot and Trigger objects.

Try adding CollisionCubes to the demo scene in such a way that they cover the walls and props, and switch the control style to Direct. Without the collision objects, the player will still be controlled by the horizontal and vertical keys, but will clip through the set geometry.

2.4 - FIRST-PERSON CONTROL

First-Person control lets you navigate your game from the player character's point of view. Technically, it is an extension of direct control, meaning you should first follow the steps explained in the previous section – it requires **Horizontal**, **Vertical** and **Run** axes to be defined (if non-Touch Screen input), CollisionCube objects to be placed in the scene, and – optionally – a NavMesh to be created, if you wish to control player movement during Cutscenes.

If you are not using Touch Screen as your input, you also need to define a **ToggleCursor** axis. This button will let you toggle between using the cursor to move around the screen, and using it to turn the Player character's head.

Touch Screen input works with First Person control in the same way as Direct control. The drag distances required to make the Player character walk and run can be set in the Setting Manager.

Two additional axes need to be defined in your Input Manager: **MouseHorizontal** and **MouseVertical**. Set the Axis fields to **X axis** and **Y axis** respectively, and both Types to **Mouse Movement**. You will need to play with the numerical settings to discover what works best for you, but a Dead setting of 0.001 and Sensitivity of 0.02 have been found to give good results.

This control style makes use of both keyboard and mouse – the mouse is used to turn the character as well as look up and down, while the keyboard is used to move forward, backward and sideways.

A camera that acts as the player's point of view must also be created. This must be a child object of the Player prefab. Drag your Player prefab into a scene, attach a regular camera as a child (GameObject → Create Other → Camera) and position it inside, or just in front of, the player's head. Add a **First Person Camera** script, and tag the object as **FirstPersonCamera**. This camera will be now used during normal gameplay. Update the prefab (click Apply in the Prefab settings), and remove the Player from the scene.

The demo's player prefab, Tin Pot, comes with such a camera already attached. Navigate to Assets → AdventureCreator → Demo → Resources → Tin Pot in your Project window to see it.

2.5 - CONTROLLER SETUP

All three control styles can be used with a controller. Each requires a **Menu** button, and **Horizontal** and **Vertical** axes. To play your game using an XBox controller, set the Menu button to **joystick button 9**, and the Horizontal and Vertical Axes to **X axis** and **Y axis** respectively, and their Type to **Joystick Axis**. To allow for First Person toggling between cursor modes, declare an axis called **ToggleCursor** (as explained in 2.4), and set it's Positive button to **joystick button 19**.

You also need to define your cursor input. Create two more axes called **ControllerCursorHorizontal** and **ControllerCursorVertical**, setting the Axis to **3rd axis** and **4th axis**, and both Types to **Joystick Axis**.

Finally, you need to define your A and B buttons – the controller equivalent of a mouse's left and right clicks. Create two buttons: **Controller_A** and **Controller_B**, set the Positive Buttons to **joystick button 16** and **joystick button 17** (for an XBox controller), and the Types to **Key or Mouse Button**.

Now you can use the Settings Manager to freely switch between Input types as you develop your game.

2.6 - TOUCH-SCREEN INPUT

Adventure Creator can be used to make games for iOS and Android platforms, but all three control styles. When interacting with objects using Touch Screen input, the interface changes slightly – touching a Hotspot once will highlight it, and touching it again will interact with it. Touching away from a highlighted Hotspot will de-select it.

Examining Hotspots and Inventory items – which is done by a right-click for Mouse input – is done by placing a second finger down on the screen while the first finger is still touching. You can simulate this effect in the Unity Editor by right-clicking on a Hotspot while the left mouse button is held down.

2.7 - NAVIGATION MESHES

Navigation Meshes, or NavMeshes, define the area of your scene that the player and NPCs can navigate. Whether you have multiple NavMeshes in your scene or just one, you must always declare the default inside the **Scene Manager** (the first tab in the main Adventure Creator window). Doing so will ensure your active NavMesh is on the correct layer, known as "NavMesh". Note that Navigation Meshes may not work if their GameObject is set to a non-zero rotation.

NavMeshes are created via the Scene Manager's Navigation panel. A NavMesh object must have a Navigation script, and a Mesh Collider component. The Mesh Collider should be set to **Is trigger?**, and it's mesh set to a custom navigation mesh for the scene.

Such a mesh is an invisible mesh that, looking top-down over the scene, marks out the floor space that can be walked on. It need not stick rigidly to the floor in the Y-axis, but it's recommended to keep it close. It's also recommended to reduce the mesh's vertex count to it's bare minimum, since Adventure Creator's pathfinding algorithm refers to these vertices when calculating a path.

While this algorithm does take other objects, such as CollisionCubes, into account when calculating a path, it may occasionally give unexpected results, so it's best to have multiple NavMeshes in your scene if your scene layout is going to change. For example, if a scene involves two rooms separated by a door that can be both open and closed, you should create three NavMeshes: one for each room, and another that contains both rooms with the connection between.

The demo scene provides another example. In the scene's Hierarchy, _Navigation → _NavMesh contains two NavMeshes: one for when the barrel is standing to the side, the other for when the barrel is tipped over in the middle of the room. Click on each object, with the Mesh Collider component open to see the difference between the two.

If you are creating a game of very large scale, you may find that you need to increase the size of the **NavMesh ray length**, which you can adjust inside the Settings Manager.

If you are a coder, or have another pathfinding algorithm that you wish to replace the default with, the pathfinding result is returned via the Navigation Mesh script's **GetPointsArray** function. It takes two parameters – a start point and an end point – and returns an array of Vector3s. You can replace this function with your own to override the default pathfinding algorithm.

3.0 - CREATING CHARACTERS

3.1 - THE PLAYER

A game cannot run until a Player is defined. Even if the game is purely first-person, and the player is never seen, a prefab still needs to be created.

Your game's player character is defined in the Settings Manager (from the main Adventure Creator window). The Player prefab is created by the GameEngine object at runtime – it should not be present in the scene when you run the game. So that it can be instantiated, the Player prefab is required to be placed in a folder called **Resources**.

The Player prefab must be tagged as **Player**, but it's name need not be such. Only the root object should be given this tag, while both the root object and any children should be placed in the **Ignore Raycast** layer.

Since Adventure Creator relies on Legacy animation, the Player prefab should have an **Animation** component. The Animations array ought to be cleared, since the Character script will replace them with it's own set at runtime.

If you intend to have speech audio, add an **Audio Source** component as well. The other built-in components required are **Capsule Collider** (or whichever Collider component is appropriate), and **Rigidbody**. Position the Capsule Collider such that it covers the Player's mesh, and lock the Rigidbody's rotation, as this will be controlled in script.

Finally, add the scripts **Player** and **Paths** as components to the prefab. You can leave the Paths script alone, but you'll need to define the public fields in the Player script. These are mainly to do with animation, which is detailed further in section 3.4.

Once your Player prefab is ready, place it in the Resources folder and assign it using the Settings Manager. The demo's player character, Tin Pot, is available for study in Assets → AdventureCreator → Demo → Resources.

3.2 - NPCS

NPCs are computer-controlled characters that the player can interact with in the game. They can walk, run, speak, and be animated just as the player can, but require a little more work to place in the game.

An NPC should be untagged. As with the Player prefab, add and arrange the **Animation**, **Audio Source**, **Capsule Collider**, **Rigidbody** and **Paths** components, and then the **NPC** script. The public variables in the *NPC* script are identical to those in the *Player* script.

If you intend to make the character interactive, you'll need to move the root object onto the **Default** layer. Then add on the **Hotspot** script, and optionally the **Highlight** script. The *Highlight* script will make the character brighter when the cursor is over them – find the **Object to highlight** field in the Hotspot inspector, and select the NPC. For more on using the Hotspot inspector to create NPC interactions, refer to section 4.2.

NPCs in the scene can be manipulated in-game using Actions, and with the Hotspot script, they can be interacted with in the same way Hotspot prefabs are. For full instructions, refer to section 4.0.

3.3 - CHARACTER MOVEMENT

Interactions, Cutscenes, Dialogue Options and Triggers can all be used to control a character's movement, and also restrict a player's movement during gameplay.

Characters can move in two ways: by dynamically pathfinding their way between two points, or by following a pre-determined route designed in the Scene view. Both of these methods involve the **Paths** script.

Any character you wish to pathfind to somewhere must have the *Paths* script attached to their GameObject. A NavMesh must also be defined in the scene: refer to the earlier section 2.6 for more. The **Character: Move to point** Action can then be used to make the character navigate the scene. See section 4.1 for more on Actions. If a Character wants to pathfind but no NavMesh is set, they will simply move in a straight line directly to their destination.

To make a character move along a pre-set path, you first need to create that path as a separate object. From the Scene Manager, click **Path** under the Navigation panel. If you don't see that button available, make sure you have set up your scene correctly – see section 1.4.

Once you have created a new Path object, you should see a blue circle appear at the origin of your scene. The blue circle represents the starting point of your path. Move it to an appropriate place in your scene. Then use the Paths inspector to create your path: clicking **Add node** will make another transform gizmo appear close to the blue circle, with the number 1 appearing below it. This means it is the first node, or path point, beyond the starting point. A blue line connects nodes together, allowing you to visualise the path your character will take.

Note that the the elevation of a path's nodes are unimportant unless you check the **Override gravity?** box in the Paths inspector. Doing so will cause the character to move to each node's point on the Y-axis, as well as the X and Z. This is useful if you want a character to fly, for example. You can also make the character walking along this path wait for a time at each node, by supplying a **Wait time**.

Once you have set up your pre-determined path, you can use the **Character: Move along path** Action to move either the player or an NPC along it. Again, see section 4.1 for more on Actions.

Predetermined paths can also be used to restrict player movement during gameplay. You can use the **Player: Constrain** Action to assign a Paths object to the player, which will mean the player can only move along that path. Note that this feature only works with the Direct and First Person control styles.

For coders, a Character's path is determined by their *activePath* variable. If a Character is pathfinding, this *activePath* will refer to their own Paths component. The *targetNode* and *prevNode* integers are used to determine where on the path a character is, and in which direction they are travelling. When they reach a node, a function in the Paths script returns the next node number they should aim for.

3.3 - CHARACTER ANIMATION

Adventure Creator makes use of Unity's Legacy animation system, so your character files will need to be imported as such. The Player and NPC inspectors have public fields for five "standard" animations: **Idle**, **Walk**, **Run**, **Turn Left** and **Turn Right**. These animations are played automatically as appropriate. Note that the turning animations are only played when the character is turning while standing still. If an animation is missing, the character will still move even if they are not animated.

The Player and NPC inspectors also have fields for six bone transforms: **Upper Body**, **Left Arm**, **Right Arm**, **Neck**, **Left Hand**, and **Right Hand**. The first four of these are used as mixing transforms (that is, to isolate animations to specific body parts) when using the **Character: Animate** Action. The two hand bones are used as reference when instructing a Character to hold an object, via the **Character: Hold object** Action.

When you play a custom animation on a Character, you can define an animation layer for it to be played on, from the base layer at the bottom, to the mouth layer at the top. By keeping your animations on separate layers, you can mix them together to create new animations. The demo provides a good example of this when Brain talks to the player while in his chair. He is playing his idle animation on the Base layer, turning his head left on the Neck layer, bobbing his head on the Head layer, changing his expression on the Face layer, and moving his lips on the Mouth layer. It's generally a good idea to only play one animation per layer at any one time.

The **Character: Animate** Action can also stop animations, change the standard animations, and reset a Character to idle. It also takes care of removing old animations that are no longer playing in the Character's Animation component.

When you select a custom animation to play, you can also choose if that animation is blended with or added on top of existing animations. If you are having trouble getting an additive animation to play properly, make sure that all keyframed bones in that animation start from their rest position.

The **Dialogue: Play Speech** Action also allows for two more animations: Head and Mouth. These fields act as shortcuts to play custom animations in the correct way. The Head animation is used to vary a character's head motion as they say a line, for example a nod if they are agreeing with something. This is an Additive animation played once on the Head layer. The Mouth animation is used to let the character animate their lips as they talk. You can either supply a generic "talking" animation, or a line-specific lip-sync animation. This is a Blend animation played once on the Mouth layer. Note that it is also possible to animate a character's jaw bone automatically, according to the sound they are making: see section 8.1 for more.

If you have an alternative solution for lip-syncing, for example FaceFX, you can modify the **Dialogue: Play Speech** Action to suit your needs: all actions are self-contained scripts, isolated from one other.

4.0 - INTERACTIONS

4.1 - ACTIONS AND ACTIONLISTS

Actions are at the core of any interaction in Adventure Creator. They are singular events, each performing a specific task, such as giving the Player an Inventory item, and making an NPC talk.

ActionLists are chains of Actions. Every time the player clicks on a Hotspot in the scene, walks through a Trigger mesh, begins a Cutscene, or clicks on a dialogue option in a Conversation, an ActionList is run.

Some Actions act as logic gates, allowing different Actions or ActionLists to be run conditionally. For example, when the player tries to buy something from a shop, we can use the **Inventory: Check** Action to determine if they have enough money.

The Actions available to you during development can be modified by using the Actions Manager. The standard actions available are:

Camera: Fade
Fades the camera in or out. The fade speed can be adjusted, and the game can optionally pause until it finishes. The fade colour is set in the MainCamera prefab.

Camera: Switch
Moves the MainCamera to the position, rotation and field of view of a specified GameCamera. Can be instantaneous or transition over time.

Character: Animate
Affects a Character's animation. Can play or stop a custom animation, change a standard animation (idle, walk or run), or revert the Character to idle.

Character: Face object
Makes a Character turn, either instantly or over time. Can turn to face another object, or copy that object's facing direction.

Character: Hold object
Parents a GameObject to a Character's hand transform, as chosen in the Character's inspector. The local transforms of the GameObject will be cleared.

Character: Move along path
Moves the Character along a pre-determined path. Will adhere to the speed setting selected in the relevant Paths object. Can also be used to stop a character from moving.

Character: Move to point
Moves a character to a given Marker object. By default, the character will attempt to pathfind their way to the marker, but can optionally just move in a straight line.

Dialogue: Play speech
Makes a Character talk, or – if no Character is specified – displays a message. Subtitles only appear if they are enabled from the Options menu.

Dialogue: Start conversation
Enters Conversation mode, and displays the available dialogue options in a specified conversation. Will end the currently-running ActionList.

Dialogue: Toggle option
Sets the display of a dialogue option. Can hide, show, and lock options.

Engine: Change NavMesh
Changes the active NavMesh that the Player and NPCs refer to for pathfinding.

Engine: Change scene
Moves the Player to a new scene. The scene must be listed in Unity's Build Settings.

Engine: Change timescale
Changes the timescale to a value between 0 and 1. This allows for slow-motion effects.

Engine: Check previous scene
Queries the last scene that the player visited.

Engine: Pause game
Waits a set time before either performing the next Action or giving control back to the player.

Engine: Play Sound
Triggers a Sound object to start playing. Can be used to fade sounds in or out.

Hotspot: Rename
Renames a Hotspot, or an NPC with a Hotspot component.

Inventory: Add or remove
Adds or removes an item from the Player's inventory. Items are defined in the Inventory Manager. If the player can carry multiple amounts of the item, more options will show.

Inventory: Check
Queries whether or not the player is carrying an item. If the player can carry multiple amounts of the item, more options will show.

Inventory: Select
Selects a chosen inventory item, as though the player clicked on it in the inventory menu. Will optionally add the specified item to the inventory if it is not currently held.

Object: Animate
Causes a GameObject to play or stop an animation. Legacy animation is required.

Object: Send message
Sends a given message to a GameObject. Can be either a message commonly-used by Adventure Creator (Interact, TurnOn, etc) or a custom one, with an integer argument.

Object: Set parent
Parent one GameObject to another. Can also set the child's local position and rotation.

Object: Teleport
Moves a GameObject to a Marker instantly. Can also copy the Marker's rotation.

Object: Transform
Transforms a GameObject over time, by or to a given amount. The GameObject must have a *Moveable* script attached.

Object: Visibility
Hides or shows a GameObject. Can optionally affect the GameObject's children.

Player: Constrain
Locks and unlocks various aspects of Player control, from available move directions to ability to save. When using Direct or First Person control, can also be used to specify a Path object to restrict movement to.

Variable: Check
Queries the value of a global boolean or integer declared in the Variables Manager.

Variable: Set
Sets the value of a global boolean or integer declared in the Variables Manager.

Nearly all Actions have an **After running** field. With this, you can choose what happens after an Action has been performed. You can use this to stop the ActionList, skip to another Action within that ActionList, or run a different Cutscene. Note that if a Cutscene is run, the ActionList from which it was called from will cease.

For coders, an ActionList (and by extension, a *Cutscene*, *Interaction*, *Trigger* and *DialogueOption* script) is run by calling its Interact function. The following sections will describe the various ways in which ActionLists are used.

4.2 - HOTSPOTS

Hotspots are used to create ways for the player to interact with the scene. We can position and scale **Hotspot** prefabs over geometry, and define Interaction for them that run when clicked on. Hotspots need to be on the **Default** layer to be recognised by the control scripts.

To create a Hotspot, open the Scene Manager and click Hotspot under the Logic panel. A yellow cube will appear at the scene origin, marking the region that the mouse cursor must hover over in order to select it. The name of the Hotspot is what will appear as the label when selected, but you can override this with the **Label (if not object name)** field in the **Hotspot** inspector – this is useful for differentiating multiple hotspots with the same label – an example being the two Barrel Hotspots in the demo.

Transform the Hotspot until it is in a sensible place – typically covering a piece of set geometry that you want the player to interact with. If there is such a piece of set geometry, you can make that object glow when the Hotspot is selected by adding a **Highlight** script to it, and then referring to it from the Hotspot's **Object to highlight** field in the Hotspot inspector.

Within the Hotspot's inspector, you can define it's associated interactions by using the panels at the bottom. Click the + icon in the **Use interaction** panel, and a new interaction slot will be created, as shown by the panel that appears underneath. The Use interaction is called when the player left-clicks on the Hotspot with the mouse (or presses the Controller_A input when using the Controller input type).

The **Interaction** field is a reference to the Interaction ActionList that will run when the player "uses" the hotspot. You can create an Interaction object from the Scene Manager, but it is easier to click **Auto-create** to the right of the Interaction field. Doing so will create, rename, and link a new Interaction object within the scene, which you can then select and modify to define what happens when the Interaction runs.

This new Interaction object is an ActionList. One Action will have been created automatically – the default Action as defined in the Actions Manager. You can change this Action, modify it's settings, and add, remove and re-arrange other Actions.

Back to the Hotspot inspector: the **Icon** field in the **Use interaction** panel lets you to choose the display icon that the cursor changes to when selected, provided the Settings Manager allows for cursor changes. Cursor settings are kept in the Settings Manager.

The **Player action** field dictates what the Player character does before this interaction is run, e.g. turn to face the hotspot, or walk towards it. The Player can be made to walk to a specific Marker if this is set to **Walk To Marker**, but a **Marker** object must also be defined, further up in the Hotspot inspector. You can create a new Marker from the Navigation panel in the Scene Manager.

While Hotspots can only have one Use interaction and one Examine interaction, they can have multiple Inventory interactions, with each Inventory interaction handling the use of one type of item on the object.

When you create an Inventory interaction, a drop-down box will appear in the panel where you can choose which inventory item is associated with it. If you want to create a default response (i.e. "I can't use that there!") to using an inventory item on a hotspot without creating the same interaction multiple times, you can define an **Unhandled event** in the Inventory Manager. This is described further in section 5.0.

Hotspots can be turned on and off using the **Object: Send message** Action. A Hotspot is declared "on" only if it is on the Default layer.

If you are creating a game of very large scale, you may find that you need to increase the size of the **Hotspot ray length**, which you can adjust inside the Settings Manager.

4.3 - CUTSCENES

A **Cutscene** is an ActionList object that is run when an Action triggers it. They are created by clicking the Cutscene button under the Scene Manager's Logic panel. Cutscene objects are invisible and cannot be interacted with directly by the player – their position is unimportant.

Cutscenes can be used to make cinematics in your game, change objects or variables instantaneously, and run other Cutscenes conditionally.

For example, the demo scene features a cutscene called OnStart, which is run when the scene is started. This cutscene simply checks the state of the variable **Played intro**, and acts accordingly. If the boolean is true, it plays the opening cinematic: the Cutscene named Intro1. If it is false, it moves Brain onto the chair and plays OnLoad, which sets up the room to it's post-introduction state (knocking over the canvas, for example). This is useful for debugging – by changing the state of **Played intro** within the Variables Manager, you can either watch the opening cutscene or skip to the main section of gameplay.

Nearly all Actions can call Cutscenes after running. The Scene Manager contains two further Cutscene fields: **Cutscene on start**, and **Cutscene on load**. **Cutscene on start** is run when the scene begins – whether by entering from another scene, or by starting the game from that scene. **Cutscene on load** is run when a scene is loaded thanks to a loaded saved-game, regardless of whether or not the player was in that scene when the saved-game was loaded.

The Cutscene inspector also features two of fields that separate it from the Interactions inspector: **Start delay** and **Auto-save after running**. The latter will create a saved game after it is run – see section 7.3. The **Start delay** determines the time (in seconds) that the Cutscene will wait before running it's Actions. If a **Kill** message is sent to a delayed Cutscene (using the **Object: Send message** Action), the cutscene will not run when the time runs out. This feature can be used to create timed sequences in your game.

4.4 - TRIGGERS

A **Trigger** is an ActionList that runs when the Player object passes through it. Similar to the Hotspot prefab, a Trigger must be positioned in the Scene view appropriately. Triggers can be created using the Scene Manager underneath the Logic panel.

The Trigger inspector contains a **Type** field. This is used to make the Trigger run either whenever the Player object is inside it (**Continuous**), or simply when the Player object enters it for the first time (**On enter**). The Continuous option is generally the more reliable.

The demo makes use of Triggers to affect the camera as the player navigates the scene. The default gameplay camera is NavCam1, but when the player heads to the far end of the room, a Trigger object changes the camera to NavCam2.

Triggers can be turned on and off using the **Object: Send message** Action. A Trigger turns itself off by disabling it's Collider component.

4.5 - CONVERSATIONS

A **Conversation** object lets the player choose from a list of on-screen dialogue options, which when combined allow them to converse with an NPC. They are started by using the **Dialogue: Start conversation** Action. Even if no NPC is present, they can still be used to provide the player with a choice of words.

You can create a new conversation by clicking Conversation under the Scene Manager's Logic panel. Like Cutscene objects, a Conversation object is never physically seen by the player, so it's position is irrelevant.

Similar to how a Hotspot defines fields for multiple Interaction objects, a Conversation manages fields for multiple **Dialogue Option** objects. As you create Dialogue Options, you can give each one a **Label** (the text that appears on screen), set it's initial **Enabled** state, and choose whether or not it will return to the conversation (that is, display the available options once again) once it has run. Dialogue Options will not appear to the player unless they are enabled. You can create Dialogue Options either from the Scene Manager, or by clicking "Auto-create" in the Conversation inspector.

Dialogue Options can be enabled and disabled using the **Conversation: Toggle option** Action. This is useful for preventing the player from saying the same thing twice. Options can also be locked, to ignore future calls to be turned on or off.

Conversations can also be timed, similar to those in Telltale's The Walking Dead game. If you check the **Is timed?** checkbox at the top of the Conversation inspector, you can specify the length of time that the conversation will display. You must also select a default Dialogue Option to run when the timer runs out. If a default option is not chosen, the conversation will end when the timer runs out.

4.6 - ARROW PROMPTS

An **Arrow Prompt** is an on-screen indicator that the player can perform an action by pushing a directional key. This is similar to the Quick-Time Events that are employed in Telltale's The Walking Dead game.

The demo game makes use of Arrow Prompts when the player clicks on the barrel for the first time. Left and right arrows appear on the screen – pushing Left causes the robot to push the barrel over, while pushing Right makes him leave it alone.

Arrow Prompts are created by clicking Arrow Prompt under the Scene Manager's Logic panel. As with Conversations and Cutscenes, Arrow Prompt objects are invisible and their transforms are unimportant.

You can use the Arrow Prompt inspector to provide any combination of up, down, left and right arrows. You can modify the icon of each arrow, and supply a Cutscene that will run when the appropriate key is pressed, or arrow lis clicked, by the player. The Arrow Prompts will be disabled automatically once this happens.

Arrow Prompts have a type field, which determines how they may be interacted with. They can be set to only respond to directional movement (i.e. cursor keys), cursor clicks, or both.

While a set of Arrow Prompts are on-screen, the player's regluar movement control is disabled. To make a set of Arrow Prompts appear, the object must be turned on using the **Object: Send Message** action and choosing **Turn On** as the message to send.

4.7 - SOUNDS

A **Sound** object provides additional settings for Audio Sources, allows volumes to be adjusted by the player, and allows sound to be played using the **Engine: Play sound** Action.

Sound objects are created by clicking the Sound button under the Scene Manger's Logic panel. You can set up your sound using the Audio Source component as normal, but the **Volume** field will be overridden. Instead, you can use the **Relative volume** field in the Sound inspector to adjust it's sound level. This way, you can adjust the volume relative to other sounds of the same type (e.g. music or SFX).

The **Sound type** pop-up lets you designate which category of sound the object will play. This will affect it's overall volume, since the game allows the player to choose the volume of Music, SFX and Speech audio from the Options menu. Choosing “Other” will make the Options menu ignore the volume for this object, making it independent from the rest of the game.

The **Engine: Play Sound** Action can control Sound objects by playing, stopping and fading audio. You can also change the sound clip that is being played, but this is not recommended for audio that will likely be looping when the game is saved, since any change in a Sound object's **Audio Clip** will not be stored in the save data.

5.0 - INVENTORY

5.1 - DECLARING INVENTORY ITEMS

The items that the player can pick up over the course of the game are collectively known as the **Inventory**. Items that the player is holding are displayed in the game's Inventory menu, and can be used, examined and combined with Hotspots, NPCs and and other items.

Inventory items are declared and modified using the Inventory Manager – the fifth tab in the main AdventureCreator window. Each item has fields for a unique name, icon texture, and checkboxes to determine if it is carried by the player when starting the game, and determining if the player can carry multiple units of it. This is useful for things like currency.

When the player is carrying one or more items, the Inventory menu will be enabled, unless it has been disabled with the **Player: Constrain** Action. By default, "using" an item will select it, and the cursor will change to the item's texture. You can then use that item on Hotspots, NPCs and other items.

However, you may want to something more specific to occur when the player clicks on a particular item. To do this, you can assign an InvActionList to the item's **Use** field (under Standard events). InvActionLists are like ActionLists, but are stored in asset files. Creating them is detailed in the next section. Using InvActionLists, you can also set interactions for examining an item, combining one item with another, and defining **Unhandled events**.

Unhandled events are "default" interactions that are run when no specific interaction has been defined. We can use these events to create standard messages to the player when they try to combine or use items in way the developer hasn't catered for.

For example, the demo game makes use of the **Use on hotspot** unhandled event. When the player tries using the "Fake sword" item on the pinboard, which doesn't have an interaction defined for such a scenario, the game will run this unhandled event – causing the robot to reply, "I can't cut that."

5.2 - SCENE-INDEPENDENT INVENTORY HANDLING

If our game allows inventory items to be examined and combined with each other, we will want such interactions to be scene-independent, so that they can run regardless of the scene that the player is in. **Inventory ActionLists** allow us to do this.

Inventory ActionLists, or InvActionLists, are like regular ActionLists in that they chain together Actions to perform a series of commands when run. But while ActionLists are defined inside a scene, Inventory ActionLists are separate asset files. They are created by right-clicking inside the Project window and choosing Create → Inventory ActionList.

Inventory ActionLists are more restrictive than regular ActionLists. Their actions cannot be re-arranged, and they cannot refer to scene-specific objects. Therefore, they are best used to perform simple tasks, such as making the Player character say something or re-arranging the Player's inventory. When one or more Actions exist within the Inventory ActionList, the asset file will create a hierarchy of individual Action assets. When this happens, you can no longer rename the original asset file, so be sure to do so before creating your Actions.

5.3 - MANAGING INVENTORY IN-GAME

Once declared in the Inventory Manager, Inventory items can be added to and removed from the Player by using the **Inventory: Add or remove** Action. If multiple units of the same item can be carried (by ticking the **Can carry multiple?** checkbox in the Inventory Manager), then this Action will also allow you to affect the number of units that the player is carrying. For example, if you have created an item that represents currency, you can use this Action to handle a shop transaction – removing the player's amount of money by 50.

To use an Inventory item on a particular Hotspot or NPC, refer back to section 4.2.

The **Inventory: Check** Action is used to perform different Actions based on what the player is carrying. Again, if multiple units of the same item can be carried, this Action will allow you to make a specific query about how many units of that item the player is carrying. Returning to our shop example, we can use this Action to determine if the player has enough money to buy an item, and issue a response accordingly.

For coders, the player's inventory at runtime is stored in the PersistentEngine object's **Runtime Inventory** component. A public list inside this script called **localItems** stores the player's inventory. Inventory items are given an ID number when created in the Inventory Manager, which is used by the *RuntimeInventory* script to determine which item is being affected. This allows items to be removed and inserted in between each other, without destroying their references.

6.0 - GLOBAL VARIABLES

6.1 - DECLARING GLOBAL VARIABLES

Global Variables are variables that can be used in game logic regardless of scene, to keep track of progress. For example, the demo game makes use of a boolean variable called **Tried lifting canvas**, which is used to incite a different reaction from the Player character when clicking on the canvas Hotspot multiple times.

They can also be used to debug your game. The demo games makes use of another boolean called **Played intro**, which can be set to true during development to skip the opening cutscene when the game is started.

Global Variables are defined in the Variables Manager. Variables can be either booleans (true-or-false flags) or integers (whole numbers). It is important to set a variable's type before using it in game logic. You can also give each variable a name, and an initial state.

Note that using the Variables Manager to change a variable's state while the game is running will not affect the game's current instance of those variables, and changes made to the Variables Manager will survive when the game is stopped.

6.2 - MANAGING VARIABLES IN-GAME

Variables are set and checked purely using Actions. They are set using the **Variable: Set** Action, and queried using the **Variable: Check** Action.

For coders: when the game begins, the PersistentEngine object's **RuntimeVariables** component makes a local copy of the variables, keeping them separate from the Variables Manager during runtime. These are stored in a public list called localVars, and – like inventory items – rely on ID numbers to determine which variable is being referenced.

CHAPTER II: ADVANCED FEATURES

7.0 - SAVING AND LOADING

7.1 - OVERVIEW

Adventure Creator is capable of saving and loading a player's progress with little effort on the developer's part. However, it is important to understand the way in which it does so, and what exactly is recorded in order to create an effective save system for your game.

When a game is saved, Adventure Creator stores two types of data: main and room. Main data includes the player's position, the camera's position, the current scene, and the state of the inventory and global variables. This data is stored automatically.

Room data is scene-specific, and only consists of data that has been flagged for saving. Flagging GameObjects for saving is a simple matter of attaching the appropriate *Remember* scripts to them. For example, to save the state of a conversation's dialogue options, the **Remember Conversation** script must be attached. A full description of the various *Remember* scripts is given in the next section.

So long as an object has been flagged appropriately, it's data will be saved and loaded. Adventure Creator will also handle the storage and return of room data as the player navigates different scenes.

Saved game files are stored in Unity's **Application.persistentDataPath**. You can display this path in the Unity console by calling `print(Application.persistentDataPath);` in a script. Note that saving and loading is not possible on the WebPlayer platform.

While Adventure Creator is capable of storing most of the essential data needed to properly save a game, it is not capable of saving changes to a GameObject's reference to an external asset file – the most important effect of this being that changes in animation are not saved. However, it is easy to work around this by using **Global Variables**, which are always saved.

Global Variables can be used to revert objects and NPCs to their appropriate animation states when the game is loaded by calling upon them in the scene's **Cutscene on load** field, found in the Scene Manager. The demo provides several examples of this, all of which are described in section 7.5.

The save system will also not save dynamic paths, i.e. those generated by characters when pathfinding. Thus, if the Player character is pathfinding do a point as the game is saved, they will no longer be moving when that game is loaded back.

For coders, saved games can be accessed via the *SaveSystem* script, which is attached to the PersistentEngine, created at runtime. SaveSystem contains the public functions *LoadGame*, *SaveGame*, and *SaveNewGame* for use in handling save files. The *LoadGame* and *SaveGame* functions require the slot number as a parameter. The slot number, which refers to the list of saved games in the menus, is also appended to the save's filename.

For example, the following code will load the first save game slot:

```
GameObject.FindWithTag (Tags.persistentEngine).GetComponent
<SaveSystem>().LoadGame (0);
```

Bear in mind that, in order to function correctly, a scene must be in Unity's Build Settings in order to load a saved game from it, and it must have finished initialising correctly before transitioning to a new one. Do not call *LoadGame* from within either the *Awake* or *Start* functions.

7.2 - SAVING INDIVIDUAL OBJECTS

Unless it is either the GameEngine, PersistentEngine, the Player, or the MainCamera, GameObjects in your scene cannot be saved without an associated ID number. The **ConstantID** script, when attached to a GameObject, will provide such a variable, and will not change upon restarting Unity.

We must add a *ConstantID* script to any GameCamera that the MainCamera can be switched to when saving is enabled (that is, during normal gameplay). In doing so, we allow the referenced object (in this case, the GameCamera), to be "visible" to the save system, and thus allow it to be saved. For the same reason, we must attach a *ConstantID* scipt to each NavMesh if a scene has more than one.

However, to record specific data about the GameObject, rather than merely the reference to it, we must use *Remember* scripts. *Remember* scripts are a subclass of ConstantID, which give the save system instructions to also save specific things about that GameObject. For example, the *RememberName* script ensures that the name of it's associated GameObject will be saved. The following is a list of the various *Remember* scripts, and when they are used.

RememberConversation – Attaching this script to any Conversation object will make sure the on/off/lock state of it's DialogueOptions will be saved. This is a part of the Conversation prefab by default.

RememberHotspot – Attaching this script to any Hotspot object will make sure it's visibility to the player's cursor is saved. Technically, it records whether or not the object is on the "Default" layer or not. Attach this to Hotspots that may be turned on or off during gameplay.

RememberName – Attaching this script to any GameObject will record any changes in said GameObject's name. Attach this to any GameObject affected by the **Hotspot: Rename** Action.

RememberNPC – Attaching this script to any NPC will record that NPC's transform, active path, and visibility to the player's cursor. Note that in order to also save the active path, a *ConstantID* script must also be present on the Path object.

RememberTransform – Attaching this script to any GameObject to save it's transform data.

AdventureCreator will save the data of any object with these scripts attached. To reduce the size of save files, it is wise to use them only when necessary. For example, the demo's opening Conversation, IntroConv, has no *RememberConversation* script, since the states of it's DialogueOptions never change.

The following is a general guideline for setting a scene up correctly for saving and loading:

- All **GameCamera** objects that are used during normal gameplay (that is, not purely used during cutscenes) have a **ConstantID** script.

- All **Conversation** objects with DialogueOptions that can change have a **ConstantID** script.
- All Path objects that characters may be using upon saving have a **ConstantID** script.
- If the active NavMesh can change during a scene, give all NavMeshes a **ConstantID** script.
- All **Hotspot** objects that can be turned on or off have a **RememberHotspot** script.
- All NPCs have a **RememberNPC** script.
- All other GameObject types that are moved during gameplay (through it's Transform, not just Animation component) have a **RememberTransform** script.

Coders who wish to build their own *Remember* scripts may do so from within the *LevelStorage* script. The bottom of this script contains classes for the existing data containers, and can be copied and used as a basis for further work. *Remember* scripts are empty subclasses of *ConstantID* – the code for handling them is all within the *LevelStorage* script.

7.3 - AUTOSAVING

Save slot “0” is reserved for Autosaving, and will appear as such in the in-game Save and Load menus.

Autosaves can be made via Cutscenes. At the top of a Cutscene's inspector, tick the **Auto-save after running?** box to save the same automatically once the Cutscene has run. Be aware that this will only occur if the Cutscene does not “branch off” onto another Cutscene object. That is, none of the Actions within the Cutscene are set to **Run Cutscene** in their **After running** field.

Coders may wish to load a saved-game automatically at certain times, e.g. if the player gets a “Game Over” after being spotted by a guard. This can be done by calling the SaveSystem script's *LoadGame* function:

```
GameObject.FindWithTag (Tags.persistentEngine).GetComponent
<SaveSystem>().LoadGame (0);
```

7.4 - OPTIONS DATA

Options data is independent from save data, allowing option changes to "survive" when a saved game is loaded. They are stored in Unity's PlayerPrefs, under the *Options* key.

You can clear stored options data via the **Clear options cache** button at the bottom of the Settings Manager, in the main Adventure Creator window.

By default, options data is confined to subtitle display, language and the volume of speech, sound effects and music. They are loaded when the game begins, and saved whenever a change is made in the Options Menu, as defined in the *MenuSystem* script. See section 9.1 for more on the *MenuSystem* script.

For coders, Option variables are stored in the *OptionsData* class. The public instance of this class is in the *Options* script, which is attached to the PersistentEngine, created at runtime.

7.5 - HOW THE DEMO DOES IT

The demo game, while simple, demonstrates a fully-functioning save and load system.

The first step to creating such a system is to be aware of the conditions under which saving is possible. While loading is possible at any time, a game can only be saved during normal gameplay (that is, not during cutscenes or conversations). For that reason, the player cannot save progress in the demo until the introduction cinematic has played, and we can use this knowledge to make assumptions about the state of the scene when the game loads.

We know that during normal gameplay, Brain will be sat in the chair, and the canvas will be tipped over. Therefore, the *OnLoad* Cutscene (which is set as the **Cutscene on Load** in the demo's Scene Manager) sets up the animation states for Brain, the chair and the canvas. The Player object is also reset to idle, since it may have been playing a custom animation when the player requested a saved game to load.

The state of the barrel, which may have been tipped over by the player while playing, is not so certain. Since the barrel moves through animation, rather than having it's Transform component altered, we must play the correct animation in the *OnLoad* Cutscene. For this, we make use of a Global Variable (as defined in the Variables Manager) called *Tipped barrel* – a boolean which is set to become *true* when the player tips the barrel over – see the *BarrelTip* Cutscene. The state of this variable is saved automatically, so we can check it's value in OnLoad, and change the barrel's animation accordingly.

The rest of the save system is set up by careful placement of *ConstantID* and *Remember* scripts on certain GameObjects:

- **ConstantID** placed on the *NavCam1* and *NavCam2* GameCameras ensures the reference to the active camera is stored. Only these cameras require this script, since the game can only be saved during normal gameplay.
- **RememberNPC** placed on *Brain* ensures his transformation is stored
- **RememberConversation** placed on *BrainConv* (the main conversation object) ensures the “enabled” state of each DialogueOption is stored
- **RememberHotspot** placed on the *Sword*, *Barrel1* and *Barrel2* Hotspots ensures their visibility to the player's cursor is stored. The *Sword* Hotspot is turned off as the player picks it up, and the *Barrel1* Hotspot is replaced with *Barrel2* when the player pushes the barrel over.
- **RememberTransform** placed on the *Sword* mesh (inside the _SetGeometry folder) ensures its transformation is stored. When the player picks the sword up, the mesh object is hidden from view by moving it to a far-off Marker called *HideMarker*.

Additionally, the demo game makes use of a Global Variable called *Played intro*. By setting this boolean to *true* before starting the game, the game will skip the opening cinematic, which is useful when testing the scene during development. The *OnStart* Cutscene, which runs when the scene begins (having been set in the Scene Manager), checks the state of this variable and either plays the intro, or skips it by moving Brain to the chair and running *OnLoad*.

8.0 - SPEECH MANAGEMENT

8.1 - AUDIO FILES

Audio files for speech are dynamically loaded at runtime, meaning you do not have to create a link between each **Dialogue: Play speech** Action and it's associated AudioClip. Such clips are loaded by giving each line of speech an ID number – a unique integer assigned by the Speech Manager.

You can access the Speech Manager from the final tab in the main Adventure Creator window (Window → Adventure Creator). Clicking **Update list** will cause the manager to search all scenes added to the Build Settings (File → Build Settings) and give an ID to every **Dialogue: Play Speech** Action. It will also do the same to lines contained in InvActionLists referenced in the Inventory Manager – see section 5.2.

You will first be prompted to save the open scene. This is a non-destructive process: IDs will only be added to those Actions without one. When it has finished, the speech lines found in the game will be listed underneath, along with the name of the speaker and ID number. Note that speech lines that are blank, or without a speaker, will not be listed.

Audio files can now be made for the game. Clicking **Create script sheet** will produce a script file called **GameScript.txt** in the root Assets folder, which voice actors can be given. This script file also contains the filename that each line is expected to have. The filename syntax is "Character name" + "ID number", while the audio format and extension can be any that Unity recognises. For example, Brain's line, "Hey, little robot!" has an ID number of 21, meaning the associated audio file is named *Brain21.mp3*. Player lines are always named as *Player*, rather than the Player prefab's name.

Audio files for speech are placed in a Speech subfolder inside your Resources directory, as described in the Speech Manager.

While mouth animations can be applied to each **Dialogue: Play speech** Action (see section 3.4), Adventure Creator also features a method of moving a character's mouth automatically, based on the volume of their speech audio. This is used by the demo game to animate the robot's jaw without a need for line-specific animation clips.

To make use of this feature, simply add the **AutoLipSync** script to a character, and modify the public variables in the inspector as needed. A "jaw bone" Transform will need to be assigned, as will the axis to rotate the bone on.

8.2 - MANAGING TRANSLATIONS

Adventure Creator comes with translation support – that is, the language of displayed subtitles. A game's active language is stored in it's Options Data (see the Options section) and is controlled by the player in the Options menu.

Within the Speech Manager, you can add and remove supported translations underneath the Languages panel. Each translation adds an additional text field beneath each line in the Speech lines panel. You can enter your translated lines into these boxes. As an example, the demo game comes with a French translation.

By default, only subtitles are translated, but you can also play alternative audio files when different languages are selected. By checking the "Audio translations?" box in the Settings Manager, the engine will look for different audio files inside a subfolder of the translation's name. To use the previous section's example, a French translation of Brain's line 21 will be placed in *Resources → Speech → French → Brain21.mp3*.

CHAPTER III: EXTENDING FUNCTIONALITY

9.0 - MENUS

9.1 - OVERVIEW

Each element of the game's user interface – from the Pause and Options menus to the cursor label and inventory bar – are created using menus.

Adventure Creator's menu system is designed to be both powerful and simple to customise. The menus are all defined in the **MenuSystem** script, which is a component of the *GameEngine* prefab, found within Assets → AdventureCreator → Prefabs → Automatic.

Simple cosmetic changes, such as the textures, fonts and colours can be made within the *MenuSystem* inspector. Note that the cursor appearance is handled within the SettingsManager.

For wider changes, a little scripting knowledge is required, but the principles are quite simple. A list of *MenuElement* subclasses populate a *Menu* class. The constructor of a *Menu* class defines properties such as size, orientation, and display type. Each *MenuElement* subclass gives the menu different functionality. For example, the *MenuButton* class displays a clickable button.

The default MenuSystem script creates eight menus:

- **pauseMenu** – The menu that appears when the player presses the "Menu" button (as defined in the Input Manager, see section 1.1). It's Save and Quit buttons will only appear when appropriate (the Quit button, for example, is not necessary on the Web Player platform).
- **optionsMenu** – Provides a list of available options. Changing any option will cause the Options Data to update (see the Options section).
- **saveMenu** – Provides a list of saved games to overwrite. Also provides a "new save" button, if there is an available slot.
- **loadMenu** – Provides a list of saved games to load.
- **inventoryMenu** – Displays all Inventory items carried by the player.
- **InGameMenu** – Displays a button to access the pause menu.
- **conversationMenu** – Displays the current Conversation's available dialogue options. Will also display a timer bar if the Conversation is timed.
- **hotspotMenu** – Displays the label of the Hotspot, Inventory item or NPC that the player's cursor is hovering over. Will always position itself above the cursor.
- **subsMenu** – Displays the current speaker and speech line, as supplied by the **Dialogue: Speech line** Action.

Each menu shares the same font type, size and colour. Fonts are scaled according to the screen size. The *backgroundTexture* variable defines the shared background that most menus have.

The *Menu* and *MenuElement* classes are defined in the header, and populated in the *Start* function. The public function *ProcessClick* is called every time the player clicks on an element. The function is sent arguments to determine which element was clicked on (and which slot, if the element displays a list of items, as is the case of inventory and saved games). The function takes these arguments and acts accordingly. For example, if the player clicks on a saved game slot in the Load Menu, the *ProcessClick* function closes the menu, and loads the saved game represented by that slot.

All *Menu* and *MenuElement* classes can be kept private, save for three MenuElements: pauseMenu's Save button, and the two labels that populate subsMenu.

Most changes needed to the menu system should be possible just within the *MenuSystem* script, so it is a good idea to amend it rather than start from scratch. Additional *MenuElement* subclasses can be written for integration, as well. The next section provides a detailed description of each *MenuElement* subclass, and how to construct your own menus.

9.2 - MENU SCRIPTING

The following is a scripting guide for creating new menus. Please refer to the *MenuSystem* script, found in Assets → AdventureCreator → Scripts → Menu for examples on how the default menu system is built.

The *Menu* class has the following constructor:

```
Menu (float _spacing, Orientation _orientation, AppearType _appearType, Vector2
_defaultElementSize)
```

The *_spacing* float refers to the size of the border that surrounds each menu element. Like all menu-based size parameters, this is given as a decimal of the screen size – a *_spacing* of 1f will take up the entire screen.

The *_orientation* defines whether the menu elements are arranged vertically or horizontally. If a particular element contains more than one "slot" (or sub-element, e.g. a list of saved games), then that orientation is defined separately.

The *_appearType* controls under what circumstances the menu appears. The available choices are:

- **AppearType.Manual** – This type of menu will only appear and disappear when called by a script. Call the menu's *TurnOn* and *TurnOff* functions to do this.
- **AppearType.MouseOverInventory** – This is a special case type used for the inventory bar. It will appear if the cursor is hovering over it, and it has not been locked via the Player: Constrain Action. For Touch Screen input, there is an additional setting in the Settings Manager to keep it on during gameplay.
- **AppearType.DialogueOptions** – This type of menu will during conversations.
- **AppearType.OnMenuButton** – This type of menu will toggle on and off when the player presses the "Menu" button, as defined in the Input Settings.
- **AppearType.OnHotspot** – This type of menu appears when the cursor is hovering over a clickable item, whether an NPC, hotspot, or inventory item.
- **AppearType.OnSpeech** – This type of menu appears when a speech line is being played, and subtitles are enabled.
- **AppearType.Gameplay** – This type of menu appears during normal gameplay.

The *_defaultElementSize* controls how large each menu element will be, unless otherwise specified. Like the _spacing parameter, this is a decimal of the screen size. The following public functions are available in the *Menu* class to aid in customisation:

- **SetBackground** – Applies a stretched background texture to the menu
- **SetCentre** – Centres the menu at a screen size-relative point on the screen
- **Align** – Positions the menu in one of the screen corners or along an edge
- **SetSize** – Resizes the menu to supplied screen-relative dimensions
- **AutoResize** – Resizes the menu automatically based on it's visible elements
- **Centre** – Positions the menu in the middle of the screen

If a menu's *appearType* is set to AppearType.Manual, it must be programatically enabled and disabled using the **TurnOn** and **TurnOff** functions.

The **Add** function is used to add pre-declared elements to a menu. The order in which elements are added dictates the order in which they appear.

Menu elements are created by declaring a *MenuElement* subclass. The following subclasses are provided:

- **MenuLabel** – A simple label, that cannot be interacted with
- **MenuButton** – A text button that can be clicked on
- **MenuInventoryBox** – A list of items carried by the player
- **MenuSavesList** – A list of saved games
- **MenuDialogList** – The available options in the active Conversation
- **MenuTimer** – A horizontal countdown bar for timed Conversations
- **MenuToggle** – A label that toggles "on" and "off" when clicked on
- **MenuCycle** – A label that cycles between an array of strings when clicked on
- **MenuSlider** – A horizontal bar whose size represents a decimal amount

Each *MenuElement* subclass has it's own constructor parameters – refer to their individual scripts to see how they are used. Each one shares the following public functions that are used to customise display:

- **SetBackground** – Assigns a stretched texture over the entire element
- **SetSize** – Resizes the element to supplied screen-relative dimensions
- **SetAbsoluteSize** – Resizes each element to absolute (pixel) dimensions

MenuSavesList, *MenuInventoryBox* and *MenuDialogList* contain "slots", or sub-elements so that multiple items can be displayed within one. For these elements, the SetSize and SetAbsoluteSize functions dictate the size of each slot, rather than the entire element.

The *MenuSystem* script declares it's Menus, MenuElements, font and textures in it's header. The menus are built in the Start function by adding the MenuElement objects to their appropriate Menus. Here, the size and position of each menu is also declared.

Two public style variables, *normalStyle* and *highlightedStyle* are defined in *MenuSystem* for use by *PlayerMenus*. These two styles are used by each menu to define their font styling.

When the player clicks on a menu element, a function called *ProcessClick* is run inside *MenuSystem*. This function dictates the action to take, based on menu, element, element slot, and the button that was clicked.

10.0 - INTEGRATING NEW CODE

10.1 - CUSTOM SCRIPTS

You can call a custom script – or rather, run a function within a custom script – from an ActionList by using the **Object: Send message** Action. This Action can be used to send a message to another object. If that object holds any scripts that declare a function with the message's name, then that function will run.

A drop-down list in the **Object: Send message** Action allows you to choose the message that the Action will send. As well as a number of standard messages used by Adventure Creator, you can supply a custom message and, optionally, an integer to pass as a parameter.

As an example, let's say we want to integrate an Achievement script into our game. This script causes an achievement message to display on the screen – the message displayed being determined by an integer. Our script might contain the following function:

```
DisplayAchievement (int achievement_number) {}
```

Suppose, as part of a Cutscene, we want "achievement 3" to appear. We simply add an **Object: Send Message** Action to our Cutscene, select our **Message to send** parameter as **Custom**, enter **DisplayAchievement** as our **Method name**, tick the **Pass integer to method?** checkbox, and enter the number **3** into the **Integer to send** field.

Note that when a custom script is called in this way, it will run in parallel to the ActionList it is called from – the game will not wait for the custom script to finish before continuing. In order to write a script that integrates more directly within the ActionList system, you may want to write your own Action – the process of which is outlined in the next section.

10.2 - CUSTOM ACTIONS

For an introduction to Actions and their functions, refer to the Actions And ActionLists section.

Each Action used by Adventure Creator is a self-contained script file. They can be enabled and disabled by using the Actions Manager. When **Refresh list** is clicked, the Actions Manager searches a given folder for scripts with the ".cs" file extension, and lists them, so that they can be enabled, disabled, and have a default set. Only one folder can be defined at a time, so all action files must be placed together.

Note that this button will not work if the game's platform is set to Webplayer during development.

Each Action is a subclass of the Action base class, and it's implementation and inspector GUI are written together. By writing a new Action subclass script and placing it inside the same folder as the other Action scripts, it can be integrated into the system for use with in ActionLists and InvActionLists.

To be properly visible inside the Actions Manager, a new Action must have it's *title* field defined within it's constructor. An override function called *ShowGUI* is used to display parameters inside it's inspector.

An ActionLists' Actions are stored inside a list variable. When an ActionList runs, it calls upon it's first Action (via the Action's *Run* function), and then subsequent Actions based on the previous Action's command – either continuing to the next Action, skipping to a pre-determined number, or halting. The ActionList will only move onto it's next Action once the current one is no longer running, as set with the *isRunning* boolean.

The Action's *Run* function returns a float, which corresponds to the time that the ActionList will wait before running the Action again, to see if the Action's task has been completed. A protected float called *defaultPauseTime* is often called by Actions when the time taken to complete a task cannot be calculated.

Once an Action's task has been completed, ActionList will call that Action's *End* function. This is already written in the Action base class file, and does not normally need to be overridden. The End function returns an integer that represents the Action that the ActionList should next run, if not the one that immediately follows. This is how ActionLists can skip certain Actions. If the returned value is -1, the ActionList will stop and gameplay will resume. If it is -2, the ActionList will stop but assume another ActionList has taken over game-pausing duties. If the returned value is zero, the ActionList will attempt to run the next Action as normal.

To assist in the creation of new Actions and help with the understanding of how they work, a template Action script – with comments – has been written, called *ActionTemplate.cs*. It can be found in Assets → AdventureCreator → Scripts → ActionList.

11.0 – HOW IT WORKS

11.1 - OVERVIEW

The following section gives a broad overview of the system behind Adventure Creator, so that it can be built-upon to cater to a scripter's needs. The sections that follow provide a more in-depth explanation of more specific elements.

The single most important variable used by Adventure Creator is the **gameState**, a public variable inside the StateHandler script, which is referenced by the majority of the game's other scripts. The GameState enum can take the following four values: Normal, Cutscene, DialogueOptions, and Paused. To take control away from the player, the gameState must be set to Cutscene, and returned to Normal to resume gameplay.

The StateHandler script is a component of the PersistentEngine prefab, one of two "engine" objects that are required for Adventure Creator to work. **PersistentEngine** is scene-independent, in that it's data survives scene loading, and **GameEngine** is scene-specific, and does not survive scene loading.

Scripts find the active GameEngine, PersistentEngine, and the Player prefabs by the tags of the same name, so there must only ever be one object with each tag present in the scene at any one time.

Of the two engines and the player, only the GameEngine is present when the scene begins. A component inside GameEngine, called **Kickstarter**, will create an instance of both the PersistentEngine and the Player prefabs if none exist. This way, a game can begin from any scene, which is useful for game testing.

The GameEngine is used to store and handle data that does not need to be carried to the next scene. It houses the player control scripts, the menu system, scene-specific settings and the dialogue-handling script.

The PersistentEngine is used to store and handle data that needs to transfer from one scene to the next. It houses the options and game saving scripts, saved room data as well as local instances of the variables and inventory items defined in the managers.

Only one active camera is ever present in the game – the **MainCamera**. The other cameras, including the optional First Person Camera, are merely used as reference by the main camera. When assigned an "activeCamera" variable, the main camera will imitate that camera's position, rotation, and field of view.

Most project-specific data is stored in the various Manager assets, as explained in the next section.

11.2 - MANAGERS

Each Manager is it's own asset file. A project can have multiple Managers of the same type in it's Assets folder, but only one of each will be used by the game. Which one it uses is determined by the **References** file, which must exist directly in a Resources directory.

The References script defines one public field for each manager. The AdvGame script contains a static function called **GetReferences**, which can be used to quickly obtain each of the Managers being used by the game. For example, `AdvGame.GetReferences ().settingsManager` will return the active Settings Manager.

Changes made in the managers, except those made in the Scene Manager, survive assembly reloads. Because most scripts that refer to them do so only when needed, you can make changes to your game while the game is running (such as changing the control style). The global variables defined in the Variables Manager, however, are kept separate at runtime. The RuntimeVariables script, attached to the PersistentEngine prefab, store a local copy of the variables when the game starts. This way, when Cutscenes and other game logic affect the state of the variables, the changes are not passed recursively back to the Variables Manager.

11.3 - PATHS

Paths are used to provide navigation to Characters so that they can move around a scene. The Paths script stores a list of Vector3s, as well as additional variables for things like move speed, path type, and so on.

These Vector3s, or nodes, describe a route through the scene. A Character moving along a Path will loop, ping-pong between the ends, or move to random nodes depending on the path type. A Character moves along a path by assigning that Character's **activePath** variable by using the **SetPath** function. A Character's **targetNode** and **prevNode** integers, which represent the target node and previously-visited node respectively, allows the Character to determine where on the path they are, and which direction they should be moving in.

A Character moving along a Path will refer to that Path's script to determine it's course of action upon reaching a node. The public function **GetNextNode** returns a node integer, which will be -1 if the Character has reached the end of the Path.

If a Character prefab has a Paths script attached, it can dynamically alter this Path mid-game. This is the basis of pathfinding. The **MoveAlongPoints** function, inside the Character script, will re-build it's own Path according to a supplied array of Vector3s, effectively re-assigning that Path's nodes.

The array of Vector3s for this function is generated by the NavigationMesh script. The **GetPointsArray** function requires a starting Vector3 and a target Vector3, and analyses the active NavMesh to produce the array. If you wish to replace the default pathfinding algorithm with your own, this is the function to override.

11.4 - PLAYER CONTROL

The Player is controlled indirectly, via five scripts on the GameEngine. They are:

- **PlayerMenus** – Handles the display of menus according to their AppearType
- **PlayerCursor** – Handles the display of the cursor
- **PlayerInput** – "Reads in" the player's input buttons and axes
- **PlayerInteraction** – Handles the processing of clicks on interactive objects
- **PlayerMovement** – Handles the movement of the Player object during gameplay

Each script is independent from the others as best can be. The PlayerInput script does not refer to any of the others, but all others refer to it. By keeping scripts separate in this way, it is not difficult to provide the player with more control.