User Manual
v1.51d

Table of Contents

INTRODUCTION

Adventure Creator 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.

Adventure Creator can be used to make 2D, 2.5D and 3D adventure games.

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 top menu. Choose **Adventure Creator → Editors → Game Editor** 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.

Three demo games have been created to demonstrate many of the kit's features in simple ways. They are available to play in a browser at adventurecreator.org.

The base Adventure Creator package includes the full source to the 2D and 3D demo games. It's a good idea to spend some time studying them, 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.

The Physics Demo's source is available as a separate package, and can be downloaded for free on the Adventure Creator website: adventurecreator.org.

When you are ready to begin making your own game, you can use the included New Game Wizard to handle the first steps for you, which is found in **Adventure Creator → Getting started → New Game Wizard**.

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 inputs and layers that will need to be defined before your game will run properly. Before you do anything else, create the following two Layers, by going to Edit → Project settings → Tags and Layers:

- NavMesh
- BackgroundImage

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 section 2.0 for a rundown of the various requirements. To play the demo game without errors, 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 supports both 3D animation via Unity's Legacy and Mecanim systems, as well as 2D animation via either 2D Toolkit or Unity's own 2D framework. For 2D Toolkit support, a preprocessor define must be set - refer to section 3.6 on how to do this.

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. However, do not do this if you are making a game with Ultimate FPS.

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 GAMES

Adventure Creator comes with two simple demo games – a 3D game and a 2D game – that show off the basic workflow involved.

Each demo game has its own set of Managers (see section 1.3), which must be loaded into the system before the game can be played. You can load these Managers easily by going to **Adventure Creator → Getting started** from the top toolbar, and selecting the demo game you want to set up.

Once the correct Managers are loaded in, it is then safe to load that demo's scene file. The scenes for the 3D and 2D Demo games are found in **Assets → AdventureCreator → Demo → Scenes**, and **Assets → AdventureCreator → 2DDemo → Scenes** respectively.

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

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

To keep things straightforward, both demos are designed to be played using the Point And Click movement method, but it can be updated easily if you want to play it in Direct (cursor key-driven) or First Person methods (3D Demo only). You can change the game's movement method 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 eight “managers” that each store data for a different aspect. These managers are created and accessed within the main Adventure Creator window (**Adventure Creator → Game editor** in the top toolbar). 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 an editor for handling and 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.

Cursor manager – Manages cursor settings, and the available icons when interacting with objects and NPCs.

Menu manager – Provides a visual editor for managing menus, with options for both their display and functionality.

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, you can use the included New Game Wizard to create a new set of managers and quickly define your game's main settings. You can bring up the wizard from **Window → Adventure Creator → New Game Wizard**. These managers will be listed inside a “ManagerPackage” asset file. For convenience, you can bulk-assign your managers by double-clicking this file.

Alternatively, you can manually create and assign your manager files. Each manager asset is referenced at the top of its respective tab in the main Adventure Creator. If you remove the reference (by selecting the reference field and pressing backspace), you will be asked 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 its “Demo” counterpart.

While you familiarise yourself with Adventure Creator, it may be helpful to rely on some of the Demo managers as you build your game – particularly the Menu, Actions and Cursor Managers.

The main Adventure Creator window stores its 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 3D SCENE

With the managers defined (see section 1.3), open the Adventure Creator window and click on the Settings tab, and set your **Camera perspective** (found underneath **Camera settings**) to **3D**. Then change to the **Scene** tab, from where you can create the GameObjects needed for an adventure game.

With a new scene, the top of the Scene Manager will have two **Organise room objects** buttons: **With folders** and **Without folders**. Both of these buttons will set up your scene to be useable by Adventure Creator – the only difference is whether or not "helper" folders (identified with underscores) will also be created to help keep things organised. These folders are not necessary, but are helpful, so it's recommended to click **With folders**. As you use the Scene Manager to create Hotspots, Conversations and other Adventure Creator prefabs, it will place them into the relevant folders automatically.

Adventure Creator will require its own **MainCamera** prefab – if it detects that another camera is present, then it will ask you if you would like to replace it completely, or attempt to convert it into a camera that Adventure Creator can use.

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.

If you wish the Player's position upon entering a new scene to be based on their position in the previous scene, you can define a **Relative Marker** when using the **Engine: Change scene** Action. A tutorial on using this feature can be found online here.

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. You can either supply a custom mesh made inside a separate 3D application, or use Unity's Navigation tools to bake a NavMesh directly inside your scene. Refer to sections 2.11 to 2.13 for more on both types of NavMesh creation.

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 5.4 for more on Cutscenes. In the Settings Manager, we can use the **ActionList on start game** field to define an ActionList asset that runs when the game begins, before the scene-specific Cutscene. See section 5.8 for more on ActionList assets.

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 its associated GameCamera's position, rotation and field of view. Choose **GameCamera** in the Scene Manager and then **Add new**, 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.

When making a 3D scene, we also have the option of using a traditional third-person camera, which follows and rotates around the Player dynamically. From the Scene Manager, click **GameCameraThirdPerson**, followed by **Add new**, to create one and set it up in the Inspector. Based on your settings, additional input axes may be required – these will be listed in the Inspector for you.

Lastly, we need to define our player prefab, and set our control options. You can define 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 sections 3.0. Do not place the Player prefab into the scene – Adventure Creator will do this for you when the game starts.

1.5 - PREPARING A 2D SCENE

This section only deals with the workflow specific to a game in 2D, so be sure to read the previous section beforehand – even if you are not making a 3D game.

To begin making a 2D game, first go to the Settings Manager and set the **Camera perspective** to **2D**. A new pop-up will appear called **Moving and Turning**. This field determines how the cameras, sprites, Hotspots and Navigation Meshes relate to one another, and is very important. It can take the following values:

Unity 2D – The game is played in Unity's own "2D" view with orthographic cameras, and Characters move purely in the X/Y plane – and scaled automatically to create a depth effect. The game makes use of 2D physics components like Box2D and Polygon Colliders, rather than their 3D equivalents. The designer can make use of Polygon Collider pathfinding.

Top Down – The game is played in the X/Z plane, and the camera looks down "from above". Characters move purely in the X/Z plane – and scaled automatically to create a depth effect. 3D physics components are required, but the designer can make use of Unity's built-in Navigation pathfinding.

World Space – The game is played with perspective cameras, with the main "background sprite" behind all Characters. Characters move in 3D space, with no need for "cheating" a depth effect.

Local Space – Similar to World Space, only Characters will move and turn according to a Marker's position on-screen, rather than its position in 3D space.

The default setting of this field is **Unity 2D**, which means you work with Unity's own "2D" view in the X and Y co-ordinates, and this is the recommended way of working.

Adventure Creator's 2D scene (found in **Assets → AdventureCreator → 2D Demo → Scenes → Park**) relies on a Unity 2D workflow.

In both Unity 2D and Top Down modes, Characters do not move closer to or further away from the camera. To create a sense of depth, and to position Characters in front of and behind scene objects appropriately, a **Sorting Map** is used to draw objects in the correct order, regardless of their distance from the camera. A Sorting Map can cause Characters to shrink when moving upwards (in screen-space), and be displayed behind objects beneath them.

A Sorting Map automates the sorting order of any GameObject's Renderer component when the **FollowSortingMap** script is attached to it. Check the 2D Demo scene to see this in action – the Scene Settings (within the Scene Manager) has a SortingMap object defined, and this object (which can also be created from the Navigation panel below) defines how position affects a Renderer's sorting. This particular sorting map is set to affect a sprite's **Order in layer**, but it could be used to affect **Sorting layer** instead. For the Player Character to then be affected by the sorting map, the FollowSortingMap

component is added to the Player prefab's Sprite Child (found within **Assets** → **AdventureCreator** → **2D Demo** → **Resources** → **Brain2D** → **Sprite**).

A Sorting Map can also be used to adjust a Character sprite's scale and movement, by checking the appropriate boxes in its Inspector. For smooth scaling, you can simply set the start and end scales, and click **Interpolate in-between scales** to automatically set the intermediate scales according to their relative position.

By default, all **FollowSortingMap** components will follow the **Default SortingMap**, as set in the Scene Manager. However, a FollowSortingMap can be assigned to follow any SortingMap in the scene – either by modifying its Inspector, or by using the **Character: Change rendering** Action (see Section 5.2).

Since Unity 2D and Top Down modes effectively "fake" perspective this way, you may wish for your Characters to move vertically more slowly than horizontally (in screen-space). Within the Settings Manager, you can adjust the **Vertical movement factor** slider to do just this. This factor can also be overridden per-scene within the Scene Manager.

Your choice of Moving And Turning will affect which pathfinding methods are available to you. Unity 2D can make use of Mesh Collider and Polygon Collider, while the others can make use of Mesh Collider and Unity Navigation. The 2D Demo's Park scene makes use of Polygon Collider-based pathfinding, which allows you to define an arbitrary shape using the Polygon Collider 2D component. You can learn more about manipulating Polygon Colliders on the Unity website here.

While Unity 2D and Top Down fake perspective, you can also opt to have your Characters move in 3D space instead, just as they would in a regular 3D scene, with World Space and Screen Space.

In both of these modes, the camera assumes that it is working in the X/Y plane (the same that the Scene window's "2D" button switches to). Characters do not share the same depth as the geometry, but instead move around the scene in 3D space. For this reason, NavMeshes must also be laid out in 3D space.

As with a regular 3D scene, World Space mode causes Characters to refer to the exact 3D position of Markers and Hotspots when moving and turning. Screen Space mode, however, causes Characters to refer to their position on-screen instead. For example, if a Hotspot is closer to the camera than the Player, but raised such that it is vertically above the Player on-screen, the player will look "behind" himself when set to turn towards it. And when the Player is told to walk to a Marker, he will instead look to the Marker's apparent position on the NavMesh as "seen" by the camera. This allows you to give all Hotspots and Markers the same z-depth as the set geometry, allowing you to work almost entirely in the 2D view once a NavMesh has been set up. Note that PlayerStart objects and Markers used to teleport Characters must still be placed in the correct position in 3D space.

1.6 - UPDATING ADVENTURE CREATOR

Adventure Creator is frequently updated with new features, and it's a good idea to download the latest update when it becomes available.

You can update Adventure Creator from your Unity Asset Store account. Choose **Windows → Asset Store** from the top toolbar, then click on the **Download Manager** icon at the top.

Upon updating, Adventure Creator will revert the manager asset references (see section 1.3) to the Demo game's set. You can avoid this by not importing the References asset when you import the update – the References asset is found in your **Assets → AdventureCreator → Resources** folder.

If the manager asset references are replaced, however, it is simple to change them back to your own game's: if you used the New Game Wizard to create your managers, a ManagerPackage asset file would have been created in your game's folder. Simply select the asset, and either double-click on it, or click on **Assign managers** within its Inspector to re-assign your own game's manager asset files.

Important note: Because of the way Unity treats Complete Project assets, your Build, Player and Input settings may also be overwritten unless you uncheck these Project settings when you import. These settings can be found either at the very top or the bottom of the import list.

1.7 - MINIMALLY IMPORTING ADVENTURE CREATOR

Adventure Creator's package size is quite large, as it includes both the 2D and 3D Demo game source files. However, these demo files are not technically necessary for Adventure Creator to compile, and can add unnecessary disk space to your project (and build) once you no longer need them. It is, however, necessary to import the full Adventure Creator package when using it for the first time.

When you use the New Game Wizard (see Introduction), you are given an option about your game's interface. If you choose to use the default UI system for your game, then your Menu and Cursor Managers (see section 1.3) will make use of certain graphics and ActionLists used by the Demo game. If this is the case, the following folders will still be required:

/Assets/AdventureCreator/Demo/InvActionLists
/Assets/AdventureCreator/Demo/MenuActionLists
/Assets/AdventureCreator/Demo/UI

2.0 - INPUT AND NAVIGATION

2.1 - OVERVIEW

Adventure Creator comes with four available movement methods: Point And Click, Direct, First Person, and Drag. You can choose between them in the Settings Manager. Note that the movement method only affects how the player can navigate the scene – interacting with hotspots and characters is still dependent on your game's Interaction method – see section 5.1.

Three input methods are also available: Mouse And Keyboard, Keyboard Or Controller, and Touch Screen. The axes that you must define in the Input Manager are very specific. The following sections detail which axes are needed for each setting.

The Settings Manager will list your game's available Input Axes based on the settings chosen. If the **Assume inputs are defined?** option beneath this list is checked, these Inputs will be required, but performance to the game will be increased.

Pathfinding is an essential part of any adventure game. When the Player character moves around a scene during a Point And Click game, or when characters are moving as part of a Cutscene, pathfinding algorithms are used to determine a path from one point to another. Adventure Creator provides two methods of pathfinding: **Unity Navigation**, and **Mesh Collider**. The pathfinding method can be set on a per-scene basis within the Scene Manager, under the Scene Settings panel. These methods are explained in sections 2.11 to 2.13.

By default, characters will make one path calculation before moving to a set point in the scene. However, the Settings Manager's **Pathfinding update time (s)** value can be used to enforce regular recalculations as they move.

Be aware that Adventure Creator's functions for detecting input can be overridden by using delegates, allowing you to create or integrate a custom input manager. More on this topic can be found in section 12.9.

2.2 - POINT AND CLICK CONTROL

Point And Click control is the default movement method. 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. For Context Sensitive interactions (see section 5.1), double-clicking will make the player run if they are able to, but this can be changed to make the interaction run instantly, as a per-Hotspot setting.

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. However, you can also use the Input buttons **InteractionA** and **InteractionB** in place of the left- and right-mouse buttons respectively.

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 sections 2.11 to 2.13 for more information.

If you are making a 3D game that involves gravity, you will also need to create at least one collider in every scene to act as a floor. This can be done easily using **Collision Cubes** (see section 1.4).

There are several options related to how the player's destination when clicking is determined. The **NavMesh search %** (in the Settings Manager) allows you to choose how far from the cursor the game will search for a NavMesh, if one was not clicked on directly. If this is greater than zero, you can use the **NavMesh search direction** to determine if the search is conducted radially outward from the cursor, or straight down.

You can optionally supply a **Click marker** prefab (also in the Settings Manager), which appears in the scene when you click, at the player character's intended destination. The demo game makes use of the default ClickMarker prefab, found in AdventureCreator → Prefabs → Navigation.

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 movement method.

For non-Touch Screen input, the **Horizontal**, **Vertical** and **Run** axes must be defined in the Input Manager for Direct control to work. **ToggleRun** and **Jump** are also valid, though Jump is only available for 3D Characters. 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 button (or axis) should be a keystroke, such as left shift, that you can hold down to make the player run.

If you want the intensity of the Horizontal or Vertical axes to affect the player's speed, check **Input magnitude affects speed?** in the Settings Manager. And checking **Account for player's position on screen?** will cause pressing “down” (for example) to result in the player walking towards the camera, rather than just away from the camera's point of view.

By default, the Player will turn naturally as they move, according to the “Turn speed” defined in their Inspector. However, checking **Turn instantly when under player control?** in the Settings Manager will enable snap-turning – an effect that works best with 2D games.

Also for non-Touch Screen input, you can choose if movement keys direct the player relative to the camera, or relative to the character (also known as “Tank controls”). And if it's relative to the camera, you can also opt to limit movement to 4 or 8 directions, rather than letting the player move in any direction. Note that if the camera cuts to a different angle, the player character will continue moving in his original direction until the user moves the input direction – this prevents the player moving in an unintended direction if the angle changes sharply.

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 movement method 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. **ToggleRun** and **Jump** are also valid axes.

For non-touch screen input, you can either rely on the mouse for turning (free-aiming) or opt to move and turn solely with the keyboard. The latter is chosen by setting the **Direct movement type** (under Movement settings in the Settings Manager) to **Tank Controls**.

Otherwise, you will 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.

Two additional axes need to be defined in your Input Manager: **CursorHorizontal** and **CursorVertical**. 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.

First Person control also works with Touch Screen input by dragging a finger on the screen. The drag distances required to make the Player character walk and run can be set in the Setting Manager, under **Movement settings**. The **First person movement** field under **Touch Screen settings** allows you to choose how exactly you move in first-person – for example, using one finger to turn, and two fingers to move.

Touch Screen input can also work 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. You can also set touch-dragging to affect just the Player's direction, rather than position, in the **Movement settings** panel. Here, you can also control the jump speed.

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 **First Person Camera** inspector gives various options, allowing for head-bobbing when moving, and the ability to zoom in and out using the mouse wheel. Note that for the latter feature, the **Mouse ScrollWheel** axis must be defined in the Input Manager. This axis requires values of 1000, 0, 0.1 for Gravity, Dead and Sensitivity respectively, a

Type of Mouse Movement, and the Axis set to 3rd.

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.

This camera will always be the “active” one during normal gameplay – you can still switch the camera during Cutscenes, but once a Cutscene ends the First Person camera will be reverted to automatically. This camera is also used during Conversations (see section 5.7), but this can be changed with the **Run Conversations in first-person?** option in the Settings Manager.

If you wish to disable the First Person camera temporarily during gameplay (for a fixed-perspective close-up, for example), you can use the **Engine: Manage systems** Action to change the Movement method to None temporarily.

2.5 - DRAG CONTROL

Drag control acts much like Direct control when Touch Screen is used as an input, only it is better equipped for mouse and keyboard input. In this mode, the player can navigate a scene by clicking and dragging while holding the mouse or button down. The drag distances required to make the Player character walk and run can be set in the Setting Manager.

2.6 - STRAIGHT-TO-CURSOR CONTROL

Straight To Cursor control causes the Player to move towards the cursor whenever the mouse button (or tap, for Touch Screens) is held down.

From the **Movement settings** in the Settings Manager, you can determine how far away the Player must be to start running, and how closely the Player will follow the cursor - use higher values if the Player starts circling the cursor continuously.

You can also determine whether or not a single-click will cause the Player to move – much like regular Point And Click movement.

Note, however, that this mode does not use pathfinding - the Player will only ever move in a straight line. Therefore, unless you want pathfinding in Cutscenes or for NPCs, you do not need to set up a Navigation method for your scenes. You simply need a Collider that is able to "receive" the mouse clicks / button presses. A **Collision Cube**, that marks out the floor and placed on the Default layer, will suffice.

If you require the Player to pathfind their way to the cursor, use Point And Click movement instead.

2.7 - ULTIMATE FPS INTEGRATION

Adventure Creator can make use of the Ultimate FPS asset, which is a separate package available on the Unity Asset Store, and a popular choice for those making First Person games.

Like Adventure Creator, Ultimate FPS is a Complete Project asset, so it will overwrite your Inputs, Tags and Layers settings when you import it. Since it has more changes to make than Adventure Creator, it is advised to import it first, let it overwrite your settings, then import Adventure Creator, and make the required settings changes manually (following the steps in section 1.1).

Integrating UFPS with Adventure Creator comes down to three key steps:

1. Set your game's **Movement method** (in the Settings Manager) to **First Person**.
2. Add the **UltimateFPSIsPresent** scripting define symbol to your game's Player settings. You can find this field in Edit → Project Settings → Player.
3. Add the **UltimateFPSIntegration** components to your UFPS character. This script will add other components it requires automatically.

Alternatively, ready-made player prefabs are available on Adventure Creator's website here.

Just as with regular First Person movement, the mouse cursor can be locked (allowing free-aiming), and unlocked (allowing menu navigation but preventing camera rotation). The default state of the cursor can also be set within the Settings Manager.

When a Menu that pauses the game is enabled, the mouse cursor is automatically unlocked. The cursor lock can be toggled automatically by the player when the **ToggleCursor** input button is triggered. Additionally, the **Player: Constrain** Action can be used to disable free-aiming at any time.

The integration script, however, is isolated from the rest of Adventure Creator. This means that, should you need to amend the way the two assets work together, you can do so by duplicating the script and instead working with an amended copy. The original **UltimateFPSIntegration.cs** script can be found in /Assets/AdventureCreator/Scripts/Static.

If your UFPS player includes a HUD, you will write a simple custom script that control its display according to your gameState enum, which can read with **AC.KickStarter.stateHandler.gameState**. The Scripting Guide provides a full list of AC's public variables and functions.

2.8 - KEYBOARD AND CONTROLLER SETUP

All four movement methods can be used with either a keyboard or 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 its Positive button to **joystick button 19**.

You also need to define your cursor input. Create two more axes called **CursorHorizontal** and **CursorVertical**, setting the Axis to **3rd axis** and **4th axis**, and both Types to **Joystick Axis**. The speed of the cursor can be adjusted in your scene's GameEngine object – the PlayerInput component's **Cursor move speed value**.

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

To play your game with a keyboard, define all of the above input axes, but set the type of each to **Key or Mouse Button**, and the positive and negative buttons to your desired mapping.

2.9 - TOUCH-SCREEN INPUT

Adventure Creator can be used to make games for iOS and Android platforms, using all four movement methods. By default, interacting with Hotspots in Touch Screen mode is a two-step process: 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. You can disable this feature, and revert back to one-touch interactions, from the **Touch Screen settings** section of the Settings Manager.

In Context Sensitive interaction mode (see section 5.1), objects are examined 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.

By default, the cursor position will always be same as the touch position (or first touch position, if multiple touches are made). Optionally, the cursor can be dragged instead by dragging with a finger – its position will remain relative to the touch position instead. To enable this feature, check the **Drag cursor with touch?** box within the Settings Manager.

2.10 - OUYA INTEGRATION

Adventure Creator can be used to make games for the OUYA, which is a console based on the Android platform.

To deploy your game to the OUYA, you must first add the “OUYAIsPresent” preprocessor define to your Player Settings. To do this, choose Edit → Project Settings → Player from the top toolbar, in the Configuration section, enter **OUYAIsPresent** in the **Scripting Define Symbols** textbox for your chosen platform.

Next, you must add the **OuyaIntegration** script component to any scene that you wish to be able to make use of the Ouya controller in. It is recommended that you add this to your game's PersistentEngine prefab (found in /Assets/AdventureCreator/Resources), as this will always be present during gameplay.

Adventure Creator's OUYA integration automatically maps the OUYA's controller to the most commonly-used Input axes and buttons:

OUYA button	**Associated Adventure Creator input**
Left stick	Horizontal, Vertical (axes)
Right stick	CursorHorizontal, CursorVertical (axes)
O	InteractionA
U	Jump
Y	EndCutscene
A	InteractionB
L1	Run
R1	ToggleCursor
R3	FlashHotspots
Menu	Menu

The **OuyaIntegration** script, however, is isolated from the rest of Adventure Creator. This means that if you need to change the controls, you can do so easily by duplicating the script and instead making use of a modified version. The original **OuyaIntegration.cs** script can be found in /Assets/AdventureCreator/Scripts/Static.

2.11 - MESH COLLIDER PATHFINDING

Mesh Collider-based pathfinding is the default pathfinding method, and involves creating custom 3D meshes to mark out the area over which characters can walk. Because of the need for mesh creation, it is harder to set up than the Unity Navigation method (explained in the next section), but is dynamic – different NavMeshes can be swapped out when the layout of the scene changes.

Once the **Pathfinding method** field in the Scene Manager has been set to **Mesh Collider**, the Navigation panel will allow you to create a **NavMesh** prefab.

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.

You must then assign a mesh to the NavMesh prefab by assigning a custom mesh to the Mesh Collider component's Mesh field.

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 its 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.

Navigation Meshes can be made visible when not selected via the Scene Manager's Visibility panel. Provided your scene has an active NavMesh with a Mesh Renderer component, it can be shown and hidden using the On and Off buttons.

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. You can also adjust the **NavMesh Search %**, to change the vertical distance (as a percentage of the Screen's height) that the cursor will search beneath it for a walkable area.

2.12 - UNITY NAVIGATION PATHFINDING

Unity Navigation-based pathfinding relies on Unity's built-in NavMesh tool. It is easier to set up than Mesh Collider-based pathfinding, but isn't dynamic – the area over which characters in a scene can walk must be fixed.

Once the **Pathfinding method** field in the Scene Manager has been set to **Unity Navigation**, the Navigation panel will allow you to create a **NavMeshSegment** prefab. By placing down NavMeshSegments and positioning them over your set's floor, you can mark out the area in which characters can move. The Scene Manager's Visibility panel lets you hide and show such segments. The **StaticObstacle** prefab is also available in the Navigation panel – this can be used to define areas in which characters cannot walk.

When you have laid out the segments, open the Navigation window by clicking on Window → Navigation within the main menu, and click the **Bake** button. The newly-created NavMesh will be represented by a blue area. You can adjust how rigidly the NavMesh follows your segments with the Radius and Height values in the Bake tab – just make sure that the NavMesh itself is slightly above the floor itself.

Done correctly, Adventure Creator will then make use of this NavMesh when pathfinding is required – but be sure not to delete the original NavMeshSegment prefabs, as these are also used by the engine.

Note: You do not necessarily need to use NavMeshSegment prefabs in order to build your NavMesh – this is just a convenience. If you are not making use of point-and-click movement (see Section 2.2), then a baked NavMesh using Unity's Navigation tab is enough. In order to use point-and-click movement, colliders that mark out the NavMesh's area must be placed on the **NavMesh** layer and marked as **Navigation Static** – this is all the NavMeshSegment prefab really is, and can be replaced with custom collider(s) if desired.

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. You can also adjust the **NavMesh Search %**, to change the vertical distance (as a percentage of the Screen's height) that the cursor will search beneath it for a walkable area.

If you prefer, you can also make use of Unity's own NavMeshAgent component to move characters around a NavMesh. To do this, simply add the component, as well as the **NavMeshAgentIntegration** script. This script will give control to the NavMeshAgent instead. This script is isolated from the rest of Adventure Creator – meaning you can amend it (or a duplicate of it) to tailor it to your specific needs without causing undesired problems elsewhere.

2.13 - POLYGON COLLIDER PATHFINDING

Polygon Collider-based pathfinding is only a valid option when making a “Unity 2D” 2D game (see section 1.5). Like Mesh Collider pathfinding, it involves the creation of a custom NavMesh, only it can be created directly in the Unity editor. And also like Mesh Collider pathfinding, it is dynamic – different NavMeshes can be swapped out when the layout of the scene changes.

Once the **Pathfinding method** field in the Scene Manager has been set to **Polygon Collider**, the Navigation panel will allow you to create a **NavMesh2D** prefab.

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 NavMesh2D is on the correct layer, known as “NavMesh”.

You must then modify the shape of the Polygon Collider (represented in the Scene View by a green pentagon) to mark out the area that Characters can move. It's recommended to reduce the number of points to its bare minimum, since Adventure Creator's pathfinding algorithm refers to these vertices when calculating a path.

To create “holes” in your NavMesh, you can define additional Polygon Colliders in the NavigationMesh Inspector that will be subtracted by the main NavMesh at runtime. This can be useful if your scene includes e.g. a tree that the Player can walk around.

This method can also be used to add walkable areas together – if the “hole” Polygon Collider overlaps the boundary of the original NavMesh, then it will be added onto the NavMesh instead – rather than being subtracted.

You can add and remove holes to your NavMesh in-game by using the **Engine: Change scene setting** Action (see section 5.2).

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.

In the Settings Manager, you can also adjust the **NavMesh Search %**, to change the vertical distance (as a percentage of the Screen's height) that the cursor will search beneath it for a walkable area.

Note that you may encounter problems if your NavMesh's scale is too small, which may be the case if you are using low-resolution (e.g. 320x240) backgrounds. You can easily tell if your scale is wrong by looking at the white squares that break up a Character's path when pathfinding – they should be tiny dots in the Scene window compared with the rest of the scene. If they are overly large, your scene is likely too small and you will

have to scale up your geometry. You can easily scale up your scene sprites by reducing the **Pixels to Units** setting when importing sprites.

If you have multiple characters moving around at once, then you can use the **Character evasion** setting on the Navigation Mesh component to have them attempt to move around each other. Be aware that this will have an impact performance, and when building to mobile platforms, it is strongly recommended to set this option to **None**.

For a further performance boost, you can lower the **Accuracy** slider. The optimal value of this slider will depend on your game's scale, NavMesh size, and target platform, but should generally only be set below 1 if you experience slowdown when pathfinding.

2.14 - ACTIVE INPUTS

Active Inputs are a series of pre-defined Input buttons that, when pressed during a particular game-state (i.e. during gameplay, or in a cutscene) triggers an ActionList asset (see section 5.8). This is useful if, for example, you want to switch to a menu screen that's in a separate scene when you press the "Escape" key.

To access the Active Inputs Editor window, choose **Adventure Creator -> Editors -> Active Inputs** from the top toolbar. You can then define how many Active Inputs you want. Each Input requires a button name (which must be defined in Unity's Input Manager), which state that the game must be in for it to register, and the ActionList to run when it is triggered. If you wish for an Active Input to work during multiple game states (i.e. during both gameplay AND cutscenes) you must define two separate Inputs - one for each state.

Note that your list of Active Inputs is stored within the Settings Manager asset. If you change your Settings Manager, any Active Inputs previously defined will no longer be present.

3.0 - CREATING CHARACTERS

3.1 - INTRODUCTION

Two types of characters are supported in Adventure Creator: the Player, and NPCs. The steps involved to create either type is largely similar, and the differences are detailed in the following section. This section will cover the elements that all Characters must have.

The quickest way to get characters into your game is to use the Character Wizard, which can be opening from Unity's top toolbar, under **Adventure Creator → Editors → Character wizard**. The wizard can be used to add the necessary scripts and components onto a model or sprite, but the components themselves will still need tweaking afterwards – for example, colliders will need to be resized and animation properties will need to be correctly set. The components necessary are covered below.

NPCs require the NPC script, while the Player character requires the Player script attached to its GameObject. The inspectors for both the Player and NPC scripts are identical, and have fields grouped into three panels: **Standard animations** (either 2D or 3D, depending on the chosen animation engine), **Movement values**, and **Dialogue settings**.

The first field you should set for any new Character is its **Animation engine**. Adventure Creator supports both 3D animation via Unity's Legacy and Mecanim systems, as well as 2D animation via either 2D Toolkit or Unity's own 2D framework.

Regardless of animation engine, most Characters will require the following components:

- **Audio Source** – Needed for speech audio; no audio clip required
- **Collider** – A Capsule works best for 3D, and a Circle for “Unity 2D” 2D mode
- **Rigidbody** – With the three rotation axes locked (or a Rigidbody2D for Unity 2D games)
- **Paths** – Required for pathfinding around the scene; no additional tweaking required

Characters can still move without a Rigidbody, which can be processor-intensive if your game features many of them. Consider removing them from NPCs, and Players that do not need to pass through Triggers, or move vertically in the scene. You can also use a **Character Controller** in place of a Rigidybody and Capsule Collider, giving you the ability to limit a Player's slope limit, for example.

Refer to section 3.5 for details on how to set up a character for 3D animation, and sections 3.6 and 3.7 for 2D animation. Movement values are typical speed factors for walking, running, turning, accelerating and decelerating. The Dialogue settings panel provides a field for a portrait graphic, which will be displayed when a character talks if the subtitles menu contains a Graphic element of type Dialogue Portrait, and a text

colour, which will be used if the game's subtitles menu element allow it.

The Rigidbody settings panel offers some handy options regarding the Character's physics. If a scene's floor contains no variations in height, the **Ignore gravity?** checkbox can be used to omit the need for a Collision-based floor – the 2D Demo Player Character, Brain2D, makes use of this feature. Conversely, if a Character is expected to move on particularly steep slopes, the **Freeze when Idle?** checkbox can be used to stop them from sliding downward when standing still.

A Character's inspector also provides fields for audio clips that play when moving – to make audible footsteps, for example. So that a Character can speak aloud at the same time, however, a separate “Sound child” must be provided to allow for a second Audio Source. Add an empty GameObject to your Character's prefab (GameObject → Create Empty), and add both the Audio Source and Sound components. Set the Sound inspector's **Sound type** to SFX, and finally drag this child object into the Character's **Sound child** field. An example Character set up in this way can be found in Assets → Adventure Creator → Demo → NPCs → Brain.

The Character: Animate action can be used to change a Character's footstep sounds mid-game, however for the change to be registered by the save game files, be sure to place the new sounds in your game's Resources folder.

3.2 - THE PLAYER

If your game's **Movement method** (in the Settings Manager) is anything other than **None**, then you will need to define a Player. Even if the game is purely first-person, and the player is never seen, a prefab still needs to be created. Follow the steps outlined in the previous section before continuing, being sure to add a Player script to the Character.

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 start the game. So that it can be instantiated, the Player prefab is required to be placed in a folder called **Resources**. However, if another Player object is present in a new scene, the default Player will be temporarily replaced by this object while this scene is loaded.

The Player prefab must be tagged as **Player**, but its 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.

You will also need to attend to the Player's animation settings – refer to section 3.5 for 3D animation, or sections 3.6 and 3.7 for 2D animation.

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.

Most games involve only one Player prefab, however it is also possible to have a game that involves several Player characters, that you can switch between in-game. Within the Settings Manager, set **Player switching** to **Allow**, and you will be able to define multiple Player prefabs, as well as the default. The **Player: Switch** Action can then be called during gameplay to change the current active Player. Note, however, that only one Player prefab can be present in the scene at a time – if you wish to have two Player characters present together, you must use the **Player: Switch** Action's ability to “swap out” the inactive Player prefab with an NPC prefab that has the same model and animation set.

The ability to switch Player prefabs is also useful for giving your Player character costume changes. A different character model can be used by a different prefab, and swapped out as needed. When this is the case, you will likely want your prefabs to share the same Inventory. To allow this, just click the **Share same Inventory?** checkbox in the Settings Manager when **Player Switching** is set to **Allow**.

3.3 - 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.

Begin by following the steps in section 3.1, giving the Character an NPC script, and then making sure the NPC is untagged. It is a good idea to make your NPC a prefab, so that it can be re-used in other scenes.

If you intend to make the NPC interactive, you'll need to add the **Hotspot** component, as well as a **Collider** component with **Is Trigger?** Checked.

IMPORTANT: If your NPC is sprite-based, these components should be placed on the sprite itself. If your NPC is a 3D model, they should instead be placed on the root object. It is also necessary to move the Hotspot object onto the **Default** layer.

The optional **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 5.3.

You will also need to attend to the NPC's animation settings – refer to section 3.5 for 3D animation, or sections 3.6 and 3.7 for 2D animation.

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 5.0.

When a scene features NPCs – particularly ones that move around – the player may occasionally find themselves stuck because an NPC is in their way. To prevent this, NPCs can be made to keep away from the player if they get too close. In the NPC inspector, check **Keep out of Player's way?**, and set the minimum distance that they should keep between themselves and the player.

3.4 - 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 both walk and run. The **Minimum run distance** on a Player / NPC Inspector controls the minimum distance between the character and its target required for running to be possible.

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 sections 2.11 to 2.13 for more. The **Character: Move to point** Action can then be used to make the character navigate the scene. See section 5.2 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. This Action can also be used to make Characters float to their destination (temporarily, if they normally obey gravity), allowing for scenes that take place in space or underwater.

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**.

For greater control, you can also run a Cutscene (see section 5.4) or ActionList asset (see section 5.8) when a Character reaches each node. The Character involved can be sent as a parameter to this ActionList if parameters are enabled (see section 5.13). 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 5.2 for more on Actions.

Because object scaling varies from game to game, you may need to adjust the **Destination accuracy** slider in the Settings Manager. This slider determines how "close is close enough" when it comes to determining if a Character has reached their destination. This is visualised as a yellow sphere gizmo by the Character's feet in the Scene window.

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 movement methods.

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.

3.5 - CHARACTER ANIMATION (LEGACY)

If your Character engine is set to Legacy (see section 3.1), make sure that your animation files have been imported as such. Next, attach an **Animation** component to your Character. The Animations array ought to be cleared, since the Character script will replace them with its own set at runtime.

Note: If your game is using Top-Down 2D (see section 1.5) combined with 3D Characters (as in Runaway or The Longest Journey, for instance), your Animation component needs to be a **child object** of the Character, and must be set as your **Animation child** in the Character's inspector. Also be aware that in order for 3D models to work correctly with sprites and Sorting Maps, they may need shaders that have “ZWrite Off” set to them.

When using Legacy animation, the inspector for both the Player and NPC scripts will contain panels called **Standard 3D animations** and **Bone transforms**. Standard animations (Idle, Walk, etc.) are played automatically when appropriate. Note that the turning animations are only played when the character is turning on the spot. If an animation is missing, the character will still move even if they are not animated.

The bone transform fields are required for custom animations. The first four of these (Upper body, neck, left arm and right arm) 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.

Adventure Creator also features a number of ways to animate your Character lip-syncing, including making use of FaceFX, and these are covered in sections 10.5 and 10.6).

To animate expressions on your Characters by using blend shapes, you can use of the **Object: Blend shape** Action in conjunction with the **Shapeable** script. The Shapeable script is covered in section 8.1.

3.6 - CHARACTER ANIMATION (SPRITES UNITY)

Adventure Creator can rely on Unity's own 2D framework for character animation by setting the Character engine field in the Settings Manager. Unity's built-in 2D framework uses a variant of the Mecanim system. For that reason, you must be familiar with using the Animator component and window. You can read up on the Mecanim system from the official Unity documentation.

To create a 2D character, you will need to make a "Sprite child" object that holds your character sprite. Add an Animator component to an empty GameObject - this will be your Sprite child. You can either then add a Sprite Renderer component to this Object, or attach multiple Sprite Renderers as child Objects for layered animation (such as the "hero" character in Unity's own 2D example project). Assign your Animator component a Controller, and set up your animation clips using the Animation and Animator windows.

To create an animation clip, select your Animator component, and dock the Animation window. Create a new clip, rename it, and drag the clip's individual frames onto the Dope Sheet. Adjust the samples value to affect playback speed, and then drag the clip's asset file into the Animator window. For standard animations (explained below), you do not need to deal with Transitions or Parameters, as Adventure Creator will play the appropriate animations automatically.

2D Characters have four standard animation types that play automatically: Idle, Walk, Run, and Talk. The names of these animations are entered manually into the inspector. 2D Characters can face either four or eight directions, depending on the inspector's Diagonal sprites? checkbox. Such directions are determined via suffixes in your sprite's animation names. For example, if a Character's walk animation is “Walk”, the walk-right animation should be named “Walk_R”, and walk-left animation “Walk_L”. The following suffixes to the animation names are understood by Adventure Creator:

- _R → Right
- _L → Left
- _U → Up
- _D → Down
- _UR → Up-right
- _UL → Up-left
- _DR → Down-right
- _DL → Down-left

If your left- and right-facing sprites are merely mirror images of each other, you only need supply one or the other. Within a 2D character's “Standard 2D animations panel”, set the **Frame flipping** value to Left Mirrors Right to only rely on right-facing animations, or Right Mirrors Left for the opposite. By default, this option will only affect standard animations, such as Idle, Walk and Run – to make it affect custom animations as well, check **Flip custom animations?**.

If you need help keeping track of which animations are required for your character, based on the options you've chosen, you can open up the **List expected animations?** foldout box in the character's Inspector to see a full breakdown.

If your animation clips rely on sprite transforms, rather than swapping out frames, you can use the **Crossfade between states?** checkbox to smooth transitions.

If you are going for a retro effect, you can use the **Only move when sprite changes?** checkbox so that the character's movement is less smooth – which can be useful when working with low-resolution sprites. This is equivalent to Adventure Game Studio's "Anti-glide mode".

The **Character: Animation** Action alters when Characters use the Unity 2D framework, but note that when playing non-standard animations, you may need to add Transitions to your Animation Controller to properly control how the animation finishes playing. The Dialogue: Play Speech Action is given additional animation options, allowing playback of animations on varying layers.

To handle collision, you should add a **Circle Collider** at the base of your Character's root object (that is, it covers the feet), making sure it's not a trigger. If you are making an NPC that is clickable, you will need to add a second collider, a **Box Collider 2D**, onto the sprite child, as well as the **Hotspot** script (see section 5.3).

3.7 - CHARACTER ANIMATION (MECANIM)

Support for Mecanim in Adventure Creator is intended for designers who wish for greater control over their animation than that which Legacy provides. While Legacy animation allows designers to simply "give" Adventure Creator an animation and specify how and when to play it, Mecanim leaves the handling of animations up to the designer, while giving Adventure Creator control over certain parameters in the Character's Controller.

Players and NPCs running with Mecanim require an **Animator** component. Their Inspectors will have a new panel: **Mecanim parameters**. To move a Character in the scene, Adventure Creator requires that their Animation Controller has a float parameter that determines their movement speed. By default, the name of this parameter is "Speed", but this can be changed in the Inspector's **Move speed float** field.

Note: If your game is using Top-Down 2D (see section 1.5) combined with 3D Characters (as in Runaway or The Longest Journey, for instance), your Animator component needs to be a **child object** of the Character, and must be set as your **Animator child** in the Character's inspector. Also be aware that in order for 3D models to work correctly with sprites and Sorting Maps, they may need shaders that have "ZWrite Off" set to them.

A bool parameter is also required, which is set to true when the Character is speaking. By default, the name of this parameter is "IsTalking", but this can be changed in the Inspector's **Talk bool** field. When using the **Dialogue: Play speech** Action, you can also define a specific head and mouth animation to play, provided that the animation is already in your Mecanim FSM. The layers on which the head and mouth animations are played on are set in the Character's Inspector.

When a Character is standing still, their assigned **Move speed float** parameter is set to zero. When they are made to walk or run, this parameter is set to their **Walk speed scale** and **Run speed scale** respectively – both of which are defined in the Inspector's **Movement settings**. With this knowledge, a designer can set up an FSM inside their Controller to play Idle, Walk and Run animations accordingly. Turning can also be catered for, by supplying a **Turn float** parameter. This will be set to -1 and +1 when the Character turns left and right respectively.

One of the biggest advantages with Mecanim characters is the ability to rely on root motion, which allows for more natural movement when walking around a scene. To enable this, just check **Apply Root Motion** on your Character's Animator component, as you would normally. You can also choose how much influence the system has over the Character's turning: when the **Root motion turning** slider is set to zero, then the Character script will handle all turning manually; when set to one, then all turning will be expected to be performed by the Mecanim controller.

For more natural movement still, you can also check **Slow movement near wall colliders?**, which will cause the Character to stop moving when facing a wall (or any Collider on a layer you choose). This is most suited to Direct control games (see

section 2.3).

Mecanim-based Characters can also be animated according to their vertical speed, to allow for custom animations when falling, or climbing stairs. The **Vertical movement float** parameter outputs any change in a Character's position along the Y-axis.

As with Legacy animation, a Mecanim-based Character can also be assigned left and right hand bones, for use with the **Character: Hold object** Action.

The **Character: Animate** Action can be used by Mecanim-based Characters to change the value of any parameter in their Controller. It can also be used to change which parameter is used as the **Move speed float** and **Talk bool**. By changing these, it is possible to "redirect" the Controller to play different "standard" animations, such as Walking and Talking.

Adventure Creator also features a number of ways to animate your Character lip-syncing, including making use of FaceFX, and these are covered in sections 10.5 and 10.6).

To animate Characters using blend shapes, you can use of the **Object: Blend shape** Action in conjunction with the **Shapeable** script. The Shapeable script is covered in section 8.1.

3.8 - CHARACTER ANIMATION (SPRITES 2D TOOLKIT)

2D Toolkit is a separate Unity asset that provides sprite functionality, and is available for purchase at www.unikronsoftware.com/2dtoolkit. Support for it can be enabled via the **Animation engine** field in the Player and NPC Inspectors. You will then be prompted to define the "tk2DIsPresent" preprocessor. This can be done by:

- Opening the Player Settings, and entering **tk2DIsPresent** into the **Scripting Define Symbols** field for your game's platform.

To create a 2D character, set up a sprite and animation library using 2D Toolkit (refer to 2D Toolkit's own documentation on how to do this), and add the animated sprite as a **child** of the Character prefab. Then, drag the sprite object into the Character's **Sprite child** field in the Player or NPC inspector. Note that 2D Character sprites should have an Anchor of **Lower Center**.

2D Characters have four standard animation types that play automatically: Idle, Walk, Run, and Talk. The names of these animations are entered manually into the inspector. 2D Characters can face either four or eight directions, depending on the inspector's **Diagonal sprites?** checkbox. Such directions are determined via suffixes in your sprite's animation names. For example, if a Character's walk animation is "Walk", the walk-right animation should be named "Walk_R", and walk-left animation "Walk_L". The following suffixes to the animation names are understood by Adventure Creator:

- _R – Right
- _L – Left
- _U – Up
- _D – Down
- _UR – Up-right
- _UL – Up-left
- _DR – Down-right
- _DL – Down-left

If your left- and right-facing sprites are merely mirror images of each other, you only need supply one or the other. Within a 2D character's "Standard 2D animations panel", set the **Frame flipping** value to Left Mirrors Right to only rely on right-facing animations, or Right Mirrors Left for the opposite. By default, this option will only affect standard animations, such as Idle, Walk and Run – to make it affect custom animations as well, check **Flip custom animations?**. Note that all animations must be present inside the same animation library.

The **Character: Animation** Action is altered when Characters use the 2DToolkit engine, but functionality is the same. You can stop and start custom animations by name, altering their wrap mode as needed. The Dialogue: Play Speech Action also removes animation options, since talking animations are played automatically.

3.9 - CHARACTER ANIMATION (SPRITES UNITY COMPLEX)

Much like how the Mecanim integration is designed to give 3D game designers greater control than Legacy, Sprites Unity Complex is designed to give 2D game designers greater control than Sprites Unity, allowing for smooth transitions between animations – such as Broken Sword-style animated transitions from one walking direction to another.

Rather than requiring the names of animation clips for Adventure Creator to automatically call upon, Sprites Unity Complex leaves the handling of animations up to the designer, while giving Adventure Creator control over certain parameters in the Character's Controller – this allows the designer to make use of them however they like.

To create a 2D character, you will need to make a "Sprite child" object that holds your character sprite. Add an Animator component to an empty GameObject - this will be your Sprite child. You can either then add a Sprite Renderer component to this Object, or attach multiple Sprite Renderers as child Objects for layered animation (such as the "hero" character in Unity's own 2D example project). Assign your Animator component a Controller, and set up your animation clips using the Animation and Animator windows.

To create an animation, select your Animator component, and dock the Animation window. Add a new clip, rename it, and drag the clip's individual frames onto the Dope Sheet. Adjust the samples to affect playback speed, and then drag the clip's asset file into the Animator window.

Character with Sprites Unity Complex have a new panel in their Inspector: **Mecanim parameters**. To move a Character in the scene, Adventure Creator requires that their Animation Controller has a float parameter that determines their movement speed. By default, the name of this parameter is “Speed”, but this can be changed in the Inspector's **Move speed float** field.

When a Character is standing still, their assigned **Move speed float** parameter is set to zero. When they are made to walk or run, this parameter is set to their **Walk speed scale** and **Run speed scale** respectively – both of which are defined in the Inspector's **Movement settings**.

The direction that a Character faces is also output in the form of the **Direction integer**. The value that it takes depends on the Character's facing direction: (note that the final four are only available if the **Diagonal sprites** option is checked)

- 0 – Down
- 1 – Left
- 2 - Right
- 3 – Up
- 4 – Down-left
- 5 – Down-right
- 6 – Up-left

- 7 – Up-right

For greater control, the angle (in degrees) can be output to the **Angle float** parameter. This angle is zero when the Character faces down, and increases to 360 as the Character rotates clockwise.

When a Character talks, the **Talk bool** parameter is set to true. Turning can also be catered for, by supplying a **Turn float** parameter. This will be set to -1 and +1 when turning left and right.

If you are going for a retro effect, you can use the **Only move when sprite changes?** checkbox so that the character's movement is less smooth – which can be useful when working with low-resolution sprites. This is equivalent to Adventure Game Studio's "Anti-glide mode".

For more natural movement, you can check **Slow movement near wall colliders?**, which will cause the Character to slow their movement when facing a wall (or any Collider on a layer you choose). This is most suited to Direct control games (see section 2.3).

3.10 - HEAD TURNING

To increase realism when making 3D games, we may sometimes wish for our Characters to turn their heads to look at specific objects - rather than turning their entire bodies. This is possible when using both the Legacy and Mecanim animation engines.

If you are using Unity 5 (Free or Pro) or Unity 4 Pro, and your Character has a Humanoid Mecanim rig (as set in the Rig settings of their model file), then you can make use of automatic IK head turning – just check **IK head-turning?** within the Character's Inspector. By default, the character's Capsule Collider will be used to estimate the Character's height, but you can set this explicitly by defining the Character's **Neck bone** transform in their Inspector.

Otherwise, four animation clips are required: one each for looking up, down, left and right. When creating these clips for a Legacy-based Character, these clips should animate just the head, start in the rest position and end in the extreme position. These clips can then be assigned in the Character's Inspector. A **Neck bone** should also be supplied, to isolate the animations from the rest of the body - the demo game's Tin Pot Character provides an example of this.

For Mecanim-based Characters, you can update your FSM with two float parameters that determine the head's Yaw (left-and-right) and Pitch (up-and-down) rotation offset. The names of these parameters must be determined within the Character's Inspector.

Characters will turn their heads sideways by a maximum 60 degrees sideways, and 30 degrees up and down.

Once the animations have been properly integrated, you can make a Character face an object by using the **Character: Face object** Action, and setting Face with to Head. A Character will continue to face the supplied GameObject until the Action is run again with Stop looking? checked.

The Player Character can also be made to face the active Hotspot - from the Settings Manager, check **Player turns head to active?** underneath **Hotspot settings**. This can be temporarily disabled mid-game using the **Player: Constrain** Action. By default, the Player will turn to face the Hotspot's centre, but you can override the position by defining a **Centre point (override)** within the Hotspot's Inspector. This option will affect all Hotspots by default, but individual Hotspots can be ignored by this feature from their Inspectors.

3.11 - PRECISION MOVEMENT

As with many engines, when characters in Adventure Creator move along a path, they'll determine whether their destination is reached or not according to their distance from it. If they are within a pre-set threshold, then they'll be considered "close enough" and will stop moving. This is typical of 3D game engines, where it is often impossible to attain an “absolute zero” difference between the character and their intended destination.

You can amend this threshold via the **Destination accuracy** slider in the Settings Manager. Lower values will allow characters to not have to be so close to their targets, and higher values will require them to be much closer. The larger your game's scale is (compared to Unity's base scale), the lower you'll generally want this value to be. You may need to experiment a little to get the right value, but the default of 0.8 is generally fine for most games.

If you require precision in your movement, you will need to raise this value. Be aware, however, that this may bring about "overshooting" if this is too high - especially if your character's deceleration value is too low (meaning they take too long to slow down). If you ramp this value all the way up to 1, however, you can enable the **Attempt to be super-accurate?** setting beneath it. This setting will force characters to land on the exact point they are supposed to, but will come with a "sliding" effect that may be obvious under certain circumstances. The list below outlines some steps you can try to reduce this effect and attain more natural, precise movement:

- If you are using Mecanim's root motion feature, then make use of a Blend Tree to scale movement speed with animation speed. This will allow the character to slow down more naturally as they approach their target.
- If a character overshoots when running, increase their **Minimum run distance** value. If a character is running this far from their target, they'll slow to a walk.
- The **Deceleration** value affects at what point a character begins to slow down - lower values will cause them to slow down sooner. If you find that the character slows down so prematurely that they can't reach their destination, try raising this value. Note that a value of 0 will cause it to copy the **Acceleration** value.

4.0 - CAMERA PERSPECTIVES

4.1 - INTRODUCTION

As you create your game, you will place many cameras down in your scene. Most of these will be GameCameras, which are never used directly to view your game from, but rather are used as “reference points” for the main camera. The MainCamera prefab attaches itself to whichever GameCamera is currently active, and copies its position, rotation, field of view, orthographic type and other camera properties.

Adventure Creator allows for 2D, 2.5D and 3D adventure games, but also for combinations: for example, a 3D game in which the characters are all sprites, or a sidescrolling game in which the characters are 3D models. The camera perspective that your game takes is defined in the **Camera settings** panel within the Settings Manager.

This setting affects which GameCamera prefabs are available within the Scene Manager. However, you can still use any type of GameCamera in your game, regardless of the perspective setting you've chosen – just drag them manually from AdventureCreator → Prefabs → Cameras into your scene hierarchy.

Each type of GameCamera comes with different settings, and these will each be explained in the following sections.

For all camera and perspective types, Adventure Creator also provides widescreen and letterboxing support. Also in the Settings Manager's **Camera settings** panel is the option to **Force aspect ratio**. When checked, you can manually set your game's aspect ratio, regardless of the resolution of your game's build. A ratio of less than one will cause a Letterbox effect, while a ratio greater than one will cause a Widescreen effect. Black borders will be created to fill the screen, and the Cursor and Menus will be prevented from entering them.

4.2 - 3D CAMERAS

3D cameras, or **GameCameras**, are the default camera type in Adventure Creator, and provide the biggest amount of control over their movement in 3D space. Position, spin, pitch and field of view can all be controlled independently by unchecking the **Lock?** Toggle beside each.

When at least one axis is unlocked, a panel to affect the camera's target appears in the Inspector. By default, this is the player, but other GameObjects can be used instead if the **Target is player?** checkbox is unchecked. The speed at which the camera follows its target can also be controlled.

When an axis becomes unlocked, the method by which that axis is affected can be set. For example, the X-axis position can be set to react to the target's X-axis position, Z-axis position, position across the viewport or position away from it. The way in which this “input” results in the axis' final position depends on the Influence and Offset values, and limits can be set using the Constrain panel.

The **Side scrolling** option allows the camera to behave like a more traditional 2D adventure game camera, in which the camera only moves when the player nears the edge of the screen.

The spin rotation panel has an additional option: **Look At Target**, with a height offset, which is a simple way of ensuring the camera is always centred on the target.

To determine the best values for a GameCamera's inspector, it is often easier to tweak them while the game is running, copy their values (via the cog icon to the top-right of the inspector), and paste them back in once the game has been stopped.

GameCameras also have a **Cursor influence** panel, which allows the camera to appear to subtly “follow” the player's cursor around the screen.

Though their default projection is Perspective, GameCameras can also be set to Othographic. It is important, however, that the scene's Navigation Mesh is always visible to the camera if you are making a Point And Click game – if you are making one with Orthographic camera, be sure to rotate them downward so that they can see the NavMeshes.

If you are using Unity Pro, you can make use of post-processing features such as depth-of-field. GameCameras have a **Depth of field** setting that you can call upon in such post-processing scripts. The focal distance can either be set manually, or tied to the camera's target object. A custom script can call upon the focal distance with this simple command, which will return the value as a float:

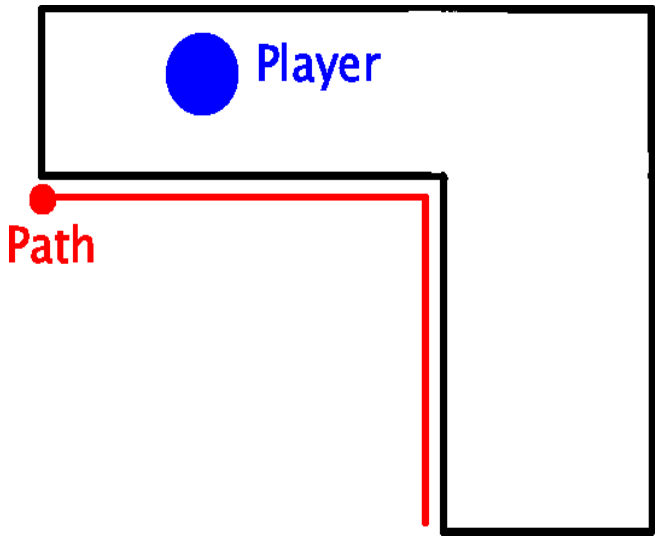
```
AC.Kickstarter.mainCamera.GetFocalDistance ();
```

4.3 - ANIMATED CAMERAS

The **GameCameraAnimated** prefab is an alternative to the regular GameCamera, and only listed in the Scene Manager when your game's perspective is set to 3D (in the Settings Manager). It is a Camera that can play back a Unity-made animation when made active. This allows for more dynamic and interesting camerawork during Cutscenes, for example.

Additionally, a GameCamerAnimated can be made to play a fixed frame from an animation based on its Target's point along a Path. A Path is a prefab that describes a series of nodes, and is also available in the Scene Manager. When the Camera's Target is at the start of the assigned Path, the Camera will play the first frame of the animation. When the Target is at the end of the Path, the Camera will play the last frame, and in-between frames will be interpolated appropriately. This allows for controlled camerawork as the Player moves along a specific section of floor, and will familiar to players of the God Of War series of games.

Do be aware, however, that such Paths must be kept to one side of the Target at all times, and the nodes must be positioned such that the reflex angles (> 180 degrees) between them must face the Target. If the Path were to be used for a corner, for example, the bird's eye view should look like this:



4.4 - 2D CAMERAS

The 2D camera, or **GameCamera2D**, provides fewer options than its 3D counterpart, but emulates the behaviour of traditional 2D adventure game cameras, whether the set is actually in 2D or not. Note than in order to work properly, a GameCamera2D must be facing the forward Z axis – if it is not, a button will appear in its inspector to correct its rotation.

GameCamera2D's do not physically move in the Scene. Rather, their projection matrices change – they "pan" across the screen yet the perspective remains the same. 3D sets will appear to be in 2D, and 2D backgrounds can be used in their place. Traditional 2D adventure games can be created by combining sprite-based characters and objects with 2D cameras.

A GameCamera2D can move horizontally and vertically, or be locked in either direction. When at least one axis is unlocked, options to control the camera's target will appear, as with the GameCamera in the previous section.

In either direction, the **Track freedom** variable determines how far, in Unity co-ordinates, the target must move from the camera's percieved centre before the camera begins to follow – a freedom of zero will keep the target in the centre of the screen. As before, the camera's movement can be constrained. Even when the game is not running, the **Offset** variable can be used to position the camera's perceived centre, so that appropriate constrain values can be determined.

As the GameCamera2D pans around your scene, you may want to have foreground and background elements panning at different speeds, to achieve a depth effect. To do this, attach a **Parallax2D** script to any GameObject to make it move with the camera. The depth value controls its panning speed – background objects should have positive values, while foreground objects should have negative values.

If you are using Unity 2D as your game's perspective, the Scene Manager will provide another 2D camera type: **GameCamera 2D Drag**. This camera can be used to pan horizontally or vertically by clicking and dragging with a mouse, or by touching and dragging on a touch-screen. As clicks will still be used to initiate player movement, this camera type is recommended for games that do not make use of Point And Click movement.

4.5 - 2.5D CAMERAS

The 2.5D camera, or GameCamera25D, facilitates the development of games that use pre-rendered backgrounds with (traditionally) 3D characters. Such cameras are fixed, and cannot move, but are useful because they can have background images assigned to them, which play underneath the rest of the scene's geometry, and are not kept in scene space. This allows a scene to have many cameras in a 2.5D game, without having to keep track of an equal number of background planes.

Such background images require a little more set-up, however. A BackgroundCamera prefab must exist, and each background image is stored within a separate BackgroundImage prefab.

When the Camera perspective option in the Settings Manager is set to 2.5D, the Scene Manager will be altered to allow for these prefabs. Clicking **Organise room objects** will now see to it that the **BackgroundCamera** prefab is created automatically, and **BackgroundImage** appears as an available prefab in the new **Set geometry** panel.

It is required that the MainCamera and the BackgroundCamera have the correct **Culling Masks**. First, ensure that the BackgroundImage layer has been defined in the Tags manager (as described in section 1.1). Then, find the MainCamera's Culling Mask, and uncheck this layer, causing the popup box to appear as “Mixed ...”. Finally, find the BackgroundCamera, and set the Culling Mask to nothing but the BackgroundImage layer.

Background images are assigned to their own BackgroundImage prefab via the **Texture** field in the **GUITexture** inspector. It is these BackgroundImage prefabs that are assigned to the GameCamera25Ds. When a GameCamera25D has been assigned a BackgroundImage prefab, a button labelled **Set as active** appears in its inspector. Clicking this allows you to preview its view, plus the background image, in the Game Window when the scene is not running.

It is recommended to make use of the **Force aspect ratio?** Option in the Settings Manager when working with background images, as this will ensure that they are proportionally correct. When working, be sure to have your Game Window's perspective set to match your chosen aspect ratio.

When handling scene sprites in 2.5D games, you can add SceneObstacle prefabs within the Scene Manager. This prefab type contains the **Align To Camera** script component to position them correctly. More on this component can be found in section 8.5.

It may be necessary to only show certain GameObjects when a particular camera is active. Simply add the **LimitVisibility** script component to an object, and you can limit its visibility by camera. This works for both mesh objects and Unity sprites.

4.6 - CUSTOM CAMERAS

Although Adventure Creator is equipped with a wide variety of camera types, it may be that you require or prefer to use a custom camera type in your scene – for example, a camera-based asset from Unity's Asset Store.

Adventure Creator makes it simple to incorporate such cameras. Bear in mind that the only active Camera within your scene should generally be the MainCamera prefab – even when custom cameras are involved. To that end, be sure to disable your custom camera's **Camera** component – Adventure Creator will still read its values, but there won't be a conflict in rendering.

Next, simply attach the "_Camera" script component to your custom camera's GameObject. Your camera can then be referenced by Adventure Creator's various camera-based Actions, and the ability to set it as the "Default camera" within the Scene Manager.

4.7 - SIMPLE CAMERAS

A **SimpleCamera**, as listed in the Scene Manager when making a 3D game, is the most simple – and the least processor-intensive – of all camera types. It cannot be moved in-game, but if you use it purely for still shots, then it will save more memory than the regular GameCamera prefab.

4.8 - VR CAMERAS

While Adventure Creator isn't geared towards VR experiences, its still possible to allow a scene to be viewed through a VR camera, using Unity's VR tools included with Unity 5.1.0 and above.

To allow your scene to be viewed through a VR camera, the MainCamera object in your Hierarchy must be split into two objects: The MainCamera component on the base, tagged as MainCamera, with the Camera component itself attached as a child object.

A VR-ready MainCamera prefab is available for you to use in your scene. First, delete the existing MainCamera object in your Hierarchy, and then drag in MainCameraVR from your Project window. The MainCameraVR prefab can be found in **Assets / AdventureCreator / Prefabs / Camera**.

5.0 - INTERACTIONS

5.1 - INTRODUCTION

The core of any adventure game involves interacting with the game world by clicking on objects and characters. In Adventure Creator, you lay down Hotspots, which in turn run interactions when clicked on.

Hotspots shine one at a time as the cursor hovers over them, but if you define an Input axis called **FlashHotspots**, you can use it to briefly illuminate all hotspots on the screen during gameplay.

Hotspots and interactions are detailed in the sections that follow, but first it is important to choose your game's **Interaction method**, as this will affect your Hotspots' inspectors and Cursor Manager.

The Interaction method is set within the Settings Manager, underneath **Interface settings**, and can be set to the following: **Context Sensitive**, **Choose Interaction Then Hotspot**, and **Choose Hotspot Then Interaction**

Context Sensitive is the default Interaction method, and allow Hotspots to have single Use and Examine interactions, as well as multiple Inventory interactions. When playing a game, you activate hotspots by hovering over them, and either left-click to run the Use interaction, or right-click to run the Examine interaction. Though multiple Use interactions can exist on a Hotspot, only the first active interaction will be registered – the **Hotspot: Change interaction** Action can be used to disable and enable specific Use interactions (see section 5.2).

When you create a Use interaction, you can optionally supply an icon to display when the mouse hovers over it, to give the player a sense of what their clicking will result in – for example, you can set a “talk” icon to display when the player hovers their cursor over an NPC. The Cursor Manager is used to define these icons – you can add, remove and set textures, as well as choose which icon serves as the Examine icon when no Use interaction is available. The Cursor Manager is also used to define the default cursor, an optional “Cutscene” cursor, as well as store a few other cursor-related settings.

Icons can also be animated, by supplying a texture with multiple “frames” of equal size. The number of rows, columns, and frames must be explicitly stated.

The **Choose Interaction Then Hotspot** method allows a hotspot to have multiple Use interactions, but no Examine interactions. The player first chooses a cursor “mode” based on the icons defined in the Cursor Manager, and then clicks on a Hotspot. The Hotspot will then run its defined Use interaction with an associated icon that matches the cursor. To change the cursor mode, the player can either right-click to cycle through available icons, trigger the **CycleCursors** input button, or select one via a list in a Menu (see section 11.1 for more on Menus). This allows for an interface similar to those used

in early 90s Sierra games, such as Kings Quest V. If the player uses a Hotspot in a way that is not defined, a fallback "unhandled" interaction can be set in the Cursor Manager. This method can also be used to recreate the classic "nine verb" interface made popular by LucasArts. Each icon listed in the Cursor Manager can be mapped to an input button – the appropriate name is listed automatically. If an input button of the expected name is pressed during gameplay, the associated icon will be selected automatically. Specific icons can also be chosen to act as "use" and "give" inventory interactions: see section 6.2.

The **Choose Hotspot Then Interaction** method is similar to the previous, only an interaction is chosen from a list of available types once the Hotspot has first been selected. There are a number of ways that an interaction can be chosen. By default, this is done by clicking a Menu that appears once a Hotspot has been clicked on. On this Menu, there is a list of available interactions to choose from. Inventory items can also be selected in this way. Such a Menu must have an Appear Type of **On Interaction** - see the Demo Manager's Interaction Menu for an example of how to set this up. With this method, Inventory items can also have multiple interactions associated with them.

The way in which interactions are chosen is set with the **Select Interactions by** setting in the Settings Manager. The default method, **Clicking Menu**, is described above. **Cycling Menu And Clicking Hotspot** also involves an Interaction Menu, only here the Interactions are chosen by pressing Inputs rather than hovering over them with the cursor. The buttons **CycleInteractionsLeft** and **CycleInteractionsRight**, and axis **CycleInteractions**, let you cycle through the various interactions available, and the selected interaction will run when the Hotspot is clicked on. The final option, **Cycling Cursor And Clicking Hotspot**, removes the need for a Menu, and simply changes the Cursor icon to represent the selected interaction. The right-mouse-button, or **CycleCursors** input button, can be used to cycle through the various interactions (and **CycleCursorsBack** will cycle in reverse). Optionally, the cursor can be cycled automatically once an Interaction is run.

When the Interaction method is set to **Choose Hotspot Then Interaction**, you are also given the option for Inventory interactions to either behave in the same way (that is, define multiple interactions per item that the player chooses from), or to act as they do when the method is set to **ContextSensitive**. This option is also given to Hotspots that only have a single "Use" Interaction defined.

The way in which Hotspots are detected can also be modified, via the **Hotspot detection method** field in the Settings Manager. By default, this is set to **Mouse Over**, which uses the cursor's position to highlight Hotspots underneath it. This can be changed to **Player Vicinity**, which causes a Hotspot to be highlighted when it enters a Trigger volume attached to the player. Combined with a Movement method of Direct, and an Input method of Keyboard or Controller, it is possible to make a game with similar controls to Grim Fandango.

If the **Hotspot detection method** is set to **Player Vicinity** and the **Input method** is set to either **Direct** or **First Person**, then you can additionally set the **Hotspots in vicinity**

setting, which determines if the selected Hotspot is simply the one closest to the Player, or if the Player can cycle through multiple Hotspots in the vicinity. If the latter is chosen, you can use the Input buttons **CycleHotspotsLeft** and **CycleHotspotsRight**, as well as the Input axis **CycleHotspots**, to allow the player to cycle through the available Hotspots.

To create a Player Vicinity trigger object, add an empty GameObject to your Player prefab as a child object. Leave it untagged, and move it to the **Ignore Raycast** layer. Then add the **Sphere Collider** (or a **Circle Collider 2D** if your game is in Unity 2D mode) and **Detect Hotspots** components, and position the sphere collider such that its centre is slightly in front of the player, and radius extends a few feet outward. Then check the **Is Trigger?** checkbox, and re-apply the prefab. Finally, assign the GameObject as the **Hotspot detector** child in your Player prefab's Inspector.

The demo player character, Tin Pot, contains such a trigger for examination. It can be found in Assets → Adventure Creator → Demo → Resources.

5.2 - 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. Each Action can be clicked on to display a description and other information, as well as give the ability to choose a "default" Action. The standard actions available are:

ActionList: Check running
Queries whether or not a supplied ActionList is currently running. By looping the **If condition is not met** field back onto itself, this will effectively "wait" until the supplied ActionList has completed before continuing.

ActionList: Check parameter
Queries the value of a parameter sent to the parent ActionList by the **ActionList: Run** and **ActionList: Set parameter** Actions. Parameters are covered in section 5.13.

ActionList: Comment
Stores text for editor display only, which is useful for keeping track of complex lists. The comment text can optionally be sent to the Console window when the Action is run.

ActionList: Kill
Instantly stops a scene or asset-based ActionList from running.

ActionList: Pause or resume
Pauses or resumes an ActionList. When instructed to pause, any currently-running Actions will first be completed. If you wish to save the pause-state of a scene-based ActionList, you must add a Constant ID component (see section 9.2). To save the pause-state of an ActionList asset, you must give it a unique name and place it in a Resources asset folder.

ActionList: Run
Runs any ActionList (either scene-based like Cutscenes, Triggers and Interactions, or ActionList assets). If the new ActionList to be run has parameters (see section 5.13), then can be set within this Action.

ActionList: Run in parallel
Runs any subsequent Actions (whether in the same list or in a new one) simultaneously. This is useful when making complex cutscenes that require timing to be exact.

ActionList: Set parameter
Sets the value of a single parameter of an ActionList. To set the value of all parameters at once, use the **ActionList: Run** Action. Parameters are covered in section 5.13.

Camera: Crossfade
Crossfades the camera from its current GameCamera to a new one, over a specified time.

Camera: Fade
Fades the camera in or out. The fade speed can be adjusted, as can the overlay texture – this is black by default.

Camera: Rotate third-person
Rotates the "GameCamera Third Person" (available in 3D games) to a fixed angle – either in World Space or relative to its target.

Camera: Shake
Causes the camera to shake, giving an earthquake screen effect. The method of shaking, i.e. moving or rotating, depends on the type of camera the Main Camera is linked to.

Camera: Split-screen
Displays two cameras on the screen at once, arranged either horizontally or vertically. Which camera is the "main" (i.e. which one responds to mouse clicks) can also be set.

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), change a footstep sound, or revert the Character to idle.

Character: Change rendering
Overrides a Character's scale, sorting order, sprite direction or Sorting Map. This is intended mainly for 2D games.

Character: Face direction
Makes a Character turn, either instantly or over time, to face a direction relative

to the camera – i.e. up, down, left or right.

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. Note that this action only works with 3D characters. If the GameObject is a prefab, and not present in the scene at runtime, it will first be instantiated.

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, or resume moving along a path if it was previously stopped.

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.

Character: NPC Follow
Makes an NPC follow another Character, whether it be a fellow NPC or the Player. If they exceed a maximum distance from their target, they will run towards them. Note that making an NPC move via another Action will make them stop following anyone.

Character: Rename
Changes the display name of a Character when subtitles are used.

Character: Switch portrait
Changes the “speaking” graphic used by Characters. To display this graphic in a Menu, place a Graphic element of type Dialogue Portrait in a Menu of Appear type: When Speech Plays. If the new graphic is placed in a Resources folder, it will be stored in saved game files.

Container: Add or remove
Adds or removes Inventory items from a Container.

Container: Check
Queries the contents of a Container for a stored Item, and reacts accordingly.

Container: Open
Opens a chosen Container, causing any Menu of Appear type: On Container to open. To close the Container, simply close the Menu.

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. A "thinking" effect can be produced by opting to not play any animation.

Dialogue: Start conversation
Enters Conversation mode, and displays the available dialogue options in a specified conversation. Optionally, the Actions that are run once an option is chosen can be overridden here – otherwise, the ActionList will end when this is run.

Dialogue: Stop speech
Ends any currently-playing speech instantly.

Dialogue: Wait for speech
Waits until a particular character has finished speaking. This is most useful when the line in question has been set to play in the background.

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

Engine: Change scene
Moves the Player to a new scene. The scene must be listed in Unity's Build Settings. By default, the screen will cut to black during the transition, but the last frame of the current scene can instead be overlaid. This allows for cinematic effects: if the next scene fades in, it will cause a crossfade effect; if the next scene doesn't fade, it will cause a straight cut. If asynchronous loading is enabled (see section 9.6), this Action can also be used to preload a scene only, so that loading time is reduced when it is next opened.

Engine: Change scene setting
Changes any of the following scene parameters: NavMesh, Default PlayerStart, Sorting Map, Tint Map, Cutscene On Load, Cutscene On Start, or Cutscene On Variable Change. When the NavMesh is a Polygon Collider, this Action can also be used to add or remove holes from it.

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

Engine: Check platform
Queries the platform that the game is running on or being built for, which is useful when creating platform-specific content.

Engine: Check scene
Queries either the current scene, or the last one visited.

Engine: End game
Ends the current game, either by loading an autosave, restarting or quitting the game executable.

Engine: Manage systems
Enables and disables individual systems within Adventure Creator, such as Interactions. Can also be used to change the "Movement method", as set in the Settings Manager, but note that this change will not be recorded in save games.

Engine: Wait
Waits a set time before continuing.

Engine: Play movie clip
Plays a MovieClip texture. By default, the MovieClip will be played full-screen, but on desktop platforms you also have the option of playing the MovieClip on a specific object and material in the scene.

Hotspot: Change interaction
Enables and disables specific Interactions on a Hotspot.

Hotspot: Check interaction enabled
Checks if a specific Interaction on a Hotspot is currently enabled.

Hotspot: Enable or disable
Turns a Hotspot on or off. To record the state of a Hotspot in save games, be sure to add the **RememberHotspot** script to the Hotspot in question.

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

Input: Check
Checks to see if the Player is pressing a mouse key, touching the screen, or pushing a button/axis as defined in Unity's Input Manager at the time it is run. Run this continuously by looping it onto itself to create QTEs and other sequences that require a specific key to be pressed to continue gameplay.

Input: QTE
Initiates a Quick Time Event for a set duration. The QTE type can either be a single key-press, holding a button down, or button-mashing. The Input button must be defined in Unity's Input Manager. For more on Quick Time Events, see section 5.17.

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: Check selected
Queries whether or not the chosen item, or no item, is currently selected.

Inventory: Crafting
Either clears the current arrangement of crafting ingredients, or evaluates them to create an appropriate result (if this is not done automatically by the recipe itself).

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.

Menu: Change state
Provides various options to show and hide both menus and menu elements.

Menu: Check num slots
Queries the number of slots that a given Menu Element has. This can be used on an "Inventory Box" Element, for example, to determine how many Inventory items the Player is carrying.

Menu: Check state
Queries the visibility of menu elements, and the enabled or locked state of menus.

Menu: Set Input Box text
Alters the text within an "Input Box" Menu Element (see section 11.2). The new text can either be entered within the Action itself, or taken from a String Global Variable (see section 7.1).

Menu: Set Journal page
Changes the selected page Journal (see section 11.2) to a specific index.

Moveable: Check held by player
Queries whether or not a Draggable or PickUp object is currently being manipulated.

Moveable: Check track position
Queries how far a Draggable object is along its track.

Moveable: Set track position
Moves a Draggable object along its track automatically to a specific point. The effect will be disabled once the object reaches the intended point, or the Action is run again with the speed value set as a negative number.

Object: Add or remove
Instantiates or deletes GameObjects within the current scene. To ensure this works with save games correctly, place any prefabs to be added in a Resources asset folder.

Object: Animate
Causes a GameObject to play or stop an animation, or modify a Blend Shape. The available options will differ depending on the chosen animation engine.

Object: Blend shape
Animates a Skinned Mesh Renderer's blend shape by a chosen amount. If the Shapeable script attached to the renderer has grouped multiple shapes into a group, all other shapes in that group will be deactivated. See section 8.1 for more on the Shapable script.

Object: Call event
Invokes a public function present on another GameObject.

Object: Change material
Changes the material on any scene-based mesh object.

Object: Change Tint map
Changes which Tint map a Follow Tint Map component uses, and the intensity of the effect (see section 8.9).

Object: Check presence
Use to determine if a particular GameObject or prefab is present in the current scene.

Object: Check visiblity
Use to determine if a particular GameObject is visible, either at all in the current scene, or within the view of the currently-active camera.

Object: Fade sprite
Fades a sprite in out out over a set time. The sprite in question must have the **SpriteFader** component attached to it. To save the state of a sprite's fade, give it the **RememberVisibility** component.

Object: Highlight
Gives a glow effect to any mesh object with the Highlight script component attached to it. Can also be used to make Inventory items glow, making it useful for tutorial sections.

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. The final position can optionally be made relative to the active camera, or the player. For example, if the Marker's position is (0, 0, 1) and **Positon relative to** is set to **Relative To Active Camera**, then the object will be teleported in front of the camera.

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

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

Player: Check
Queries which Player prefab is currently being controlled. This only applies to games for which Player switching has been allowed in the Settings Manager.

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

Player: Switch
Swaps out the Player prefab mid-game. If the new prefab has been used before, you can restore that prefab's position data – otherwise you can set the position or scene of the new player. This Action only applies to games for which Player switching has been allowed in the Settings Manager. Unless all Players share the same inventory (as set in the Settings Manager), you can also opt to transfer the previous Player's inventory onto the new Player.

Save: Check
Queries whether or not saving is currently possible (to aid in the display of a “Save” Menu Button, for example), whether or not a particular save slot is filled, or how many profiles or save game files have been created (to aid in the display of a “Continue game” Button, for example).

Save: Manage profiles
Creates, renames and deletes save-game profiles, if they're enabled (see section 9.8). If the ActionList that contains this Action has an Integer parameter (see section 5.13), then it can be set when called from a Button Menu Element.

Save: Manage saves
Renames and deletes save game files, as referenced by the slot index number of a SavesList Menu Element (see section 11.2). If the ActionList that contains this

Action has an Integer parameter (see section 5.13), then it can be set when called from a Button or SavesList Menu Element.

Save: Save or load
Allows you to load, continue the last-recorded, or save, a save-game file. Note that saving will not be permitted if any gameplay-blocking ActionList other than the one that contains this Action is running, or if a Conversation is active. The **Save: Check** Action can be used to determine if a save will succeed beforehand. This Action can also be used to optionally load a game selectively, i.e. only certain elements, such as inventory or variables.

Sound: Change volume
Alters the 'relative volume' of any Sound object. Be sure to add the RememberSound component to any Sound object whose volume changes you wish to be saved.

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

Sound: Play music
Handles the playback of a music track listed in the Music Storage window. For more about this Action, see section 5.11.

Sound: Play one-shot
Plays an AudioClip once, and without the need for a Sound object or AudioSource component. The sound will be treated as SFX, and its volume will be set as such. Though easier to use than the “Sound: Play” Action, sounds triggered with this Action will not be able to fade or be interrupted.

Sound: Set Mixer snapshot
Unity 5 only. Transitions to a single or multiple Audio Mixer snapshots.

Third-Party: Cinema Director
Runs a Cutscene built with Cinema Director. Note that Cinema Director is a separate Unity Asset, and the CinemaDirectorIsPresent preprocessor must be defined for this to work (see section 12.1).

Third-Party: PlayMaker
Calls a specified Event within a PlayMaker FSM. Note that PlayMaker is a separate Unity Asset, and the PlayMakerIsPresent preprocessor must be defined for this to work (see section 12.1).

Variable: Assign preset
Bulk-assigns all Global or Local Variables to pre-determined values. See section 7.5 for more on Variable presets. Optionally, Variables linked to Options Data (see section 9.4) can be ignored.

Variable: Check

Queries the value of both Global and Local Variables declared in the Variables Manager. Variables can be compared with a fixed value, or with the values of other Variables.

Variable: Check random number
Picks a number at random between zero and a specified integer – the value of which determines which subsequent Action is run next.

Variable: Copy
Copies the value of one Variable to another. This can be between Global and Local Variables, but only of those with the same type, such as Integer or Float. Integer and Pop-up Variables, however, can be transferred between each other.

Variable: Pop Up switch
Uses the value of a Pop Up Variable to determine which Action is run next. An option for each possible value the Variable can take will be displayed, allowing for different subsequent Actions to run.

Variable: Run sequence
Uses the value of an integer Variable to determine which Action is run next. The value is incremented by one each time (and reset to zero when a limit is reached), allowing for different subsequent Actions to play each time the Action is run.

Variable: Set
Sets the value of both Global and Local Variables, as declared in the Variables Manager. Integers can be set to absolute, incremented or assigned a random value. Strings can also be set to the value of a MenuInput element (see section 11.2), while Integers, Booleans and Floats can also be set to the value of a Mecanim parameter. When setting Integers and Floats, you can also opt to type in a formula (e.g. 2 + 3 *4), which can also include tokens of the form [var:ID] to denote the value of a Variable, where ID is the unique number given to a Variable in the Variables Manager. Note that formulas do not currently work on the Windows Phone 8 platform.

Most 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. To call another ActionList and have it run in parallel, use the **Engine: Run ActionList** Action instead.

To aid in testing, ActionLists can be run at any time while the game is running – just click **Run now** at the top of the Inspector. Actions can also be set to pause the Unity Editor just before they are run – allowing you to debug any problems more easily. This is done via the **Toggle breakpoint** option in an Action's context menu:

Actions can be deleted, copied, re-arranged, disabled and more via a context menu accessed via the top-right button on each Action inspector. A “node layout” window of

any ActionList can be displayed by clicking the **ActionList Editor** button at the top of the Inspector (also available via the icon in the Hierachy window). This window is designed to help make complex ActionLists easier to follow: Actions appear as nodes, which you can re-arrange and re-connect to one another by dragging "wires" from output sockets on the right to input sockets on the left. You can select multiple Actions at a time by dragging a box around them (hold the Shift key to add newly-selected Actions to the existing), and manipulate them in bulk by right-clicking to bring up a context menu.

When editing an ActionList within the ActionList Editor, comments can also be written above an Action. To enable comments on an Action, click on the cog to the top-right of the Action and select **Toggle comment**. Comments will be visible even if the Action itself is collapsed, allowing you to get an overview of what an ActionList does even if all the Actions are collapsed.

By default, Actions are designed to be arranged vertically (clicking "Auto-arrange" will arrange them neatly this way). However, you can arrange them horizontally if you prefer – choose your desired arrangement at the top of the **Actions Manager**.

The ActionList Editor also can be locked to a particular ActionList: click the yellow padlock icon to the bottom-left of the window, and it will lock itself to the ActionList it is currently viewing. Clicking the icon again will unlock it. Properties of the ActionList can be toggled by clicking the button to the bottom-right of the window.

By default, only one instance of the ActionList Editor can be open at once: if you open a new one, it will replace any existing windows. However, it is possible to allow for multiple instances of the window – allowing for separate windows that display separate ActionLists at once. Enabling this feature can be done from the top of the **Actions Manager**.

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.

5.3 - 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 Interactions 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, followed by **Add new**. 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 you have a MeshRenderer object selected before the Hotspot is created, the Scene Manager will give you an option to have the Hotspot surround it automatically. You can make an 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. The Highlight script will also affect GUITexture components attached to the same GameObject. If you wish to implement a custom highlight, rather than the default brightening effect, you can disable the effect and call the Highlight script's **GetHighlightIntensity** function in a custom script. Events can also be set to trigger when a highlight is enabled or disabled.

Within the Hotspot's inspector, you can define its 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 **InteractionA** input). Unless your **Interaction method** (see section 5.1) is set to Context Sensitive, you can define multiple Use interactions.

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 its settings (such as its colour when displayed in the ActionList Editor), 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.
If the Interaction method is set to Context Sensitive, you can also define an Examine interaction, which runs when the player right-clicks. You can also 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 6.0.

Hotspots can be turned on and off using the **Hotspot: Enable or disable** Action. A Hotspot is declared “on” only if it is on the Default layer. Additionally, you can limit a Hotspot's interactivity by assigning a GameCamera in the Inspector's **Limit to camera** field. When assigned, the Hotspot will only be active if the chosen camera is also active.

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.

By default, scene-based Interaction prefabs are used to handle what happens when a Hotspot is clicked on, but there are alternatives. Setting the Hotspot's **Interaction source** to **Asset File** allows you to call ActionList assets instead. This is useful for building game logic when you don't have access to the scene, for example when building a game as part of a team.

The **Interaction source** setting can also be set to **Custom Script**. You will then be able to send a message to a GameObject of your choice. This is useful if you wish to hard-code your interactions instead of relying on Actions.

5.4 - 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, followed by **Add new**. 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 its 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 three further Cutscene fields: O**n start**, **On load**, and **On variable change**. **On start** is run when the scene begins – whether by entering from another scene, or by starting the game from that scene. **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. **On variable change** is run whenever a Variable (as defined in the Variables Manager) has been altered. Note that this Cutscene will only run once the ActionList that contained the variable change has completed.

Cutscenes, like Interactions and Triggers, will pause gameplay by default when they run. If you wish to make a particular ActionList run during gameplay instead, simply choose **Run In Background** in the **When running** pop-up box.

The Cutscene inspector also features two more fields that separate it from the Interaction and Trigger inspectors: **Start delay** and **Auto-save after running**. The latter will create a saved game after it is run – see section 9.3. The **Start delay** determines the time (in seconds) that the Cutscene will wait before running its 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.

5.5 - SKIPPING CUTSCENES

Cutscenes – and all scene-based ActionLists – can be set to be skippable by the player. When made so, the player can skip the ActionList by pressing the **EndCutscene** input button, as defined by the Input Manager.

Skipping an ActionList still causes any game logic to execute: Variables will still be changed, Inventory items will still be added or removed, and objects will still be moved to their expected “end” position. However, additional work may be required if your ActionList involves playing non-Legacy animations.

Because some animations may be intended to continue playing once the Action finishes – or continue to another FSM state in Mecanim, they must still be played when an ActionList is skipped. Therefore, it is necessary to end your ActionList with Actions that place your objects and Characters in their correct animation state. For instance, if the Player waves during a Cutscene before returning to Idle, you should end your ActionList with an additonal **Character: Animate** Action that specficially returns the Player to the Idle animation, even if this happens naturally when the ActionList plays normally.

The 2D Demo's Park scene contains examples of this necessity: the “Intro2” Cutscene ends by playing the BirdHide animation on the Bird NPC, even though this animation is played by the FSM when the Cutscene plays uninterrupted.

Note that these additional steps are unnecessary for Legacy animation Actions: they will be skipped as expected.

Also to note, if your ActionList makes use of any **ActionList: Run in parallel** Actions: although the list will be skipped instantaneously, chains of Actions defined by the parallel Action will be skipped in order, and each chain to completion before the next chain is skipped.

5.6 - TRIGGERS

A **Trigger** is an ActionList that runs when an object passes through it. By default, it will only respond to the Player, but it can also be set to react to pre-determined objects, any object with a specific component, or any object with a specific tag. 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 object is inside it (**Continuous**), when the object enters it for the first time (**On enter**), or when the object exits its space (**On exit**). The Continuous option is generally the more reliable.

The 3D 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 its Collider component.

A more advanced feature of Triggers is that the GameObject it detects can be passed to its list of Actions as a parameter. For more on ActionList parameters, see section 5.13.

Important note: In order for a Trigger to work, either the Trigger or the object it is to detect must have a **Rigidbody** component. This is also true for 2D Triggers, only with a **Rigidbody2D** component.

5.7 - 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. Dialogue options are displayed in-game within a DialogueList element (see section 11.2).

You can create a new conversation by clicking **Conversation** under the Scene Manager's Logic panel, followed by **Add new**. Like Cutscene objects, a Conversation object is never physically seen by the player, so its position in 3D space is irrelevant.

The Conversation's Inspector window allows you to manage the options that appear on-screen. Options can be displayed as text or as icons, and can be enabled or disabled at any time. What happens when the player chooses an option can be handled in two ways: within the Conversation Inspector, or within the Action that calls the Conversation.

The first method involves the creation of **Dialogue Option** objects – with each object holding an ActionList that determines what happens when its associated option is chosen. The Conversation Inspector and Editor windows make it simple to auto-create such objects and link them to an option. When a Dialogue Option has run, you can choose what happens next: either to stop running that Conversation, return to the list of options, or to run a new Conversation. You can set this within the Conversation Inspector, using the **When finished** popup.

Linked Dialogue Options are considered the “default” when an option is chosen. However, it is possible to override these defaults when the Conversation is initiated with the **Dialogue: Start conversation** Action. This is the second method: by checking **Override defaults?**, you can define what happens when an option is clicked within the ActionList itself – whether it be to run other Actions in the same list, or to call a separate Cutscene. Be aware that this has no **When finished** option like the Conversation Inspector method does – you will have to call the same **Dialogue: Start converstation** Action manually if you want to return to the same options afterwards.

Dialogue Options can be enabled and disabled using the **Dialogue: 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.

Sometimes, a player may only be faced with one dialogue option – when this happens, this option can be played automatically. To do this, just check the **Auto-play lone option** checkbox at the top of the Conversation inspector.

Particularly if your game is keyboard-controlled, you can make it easier for your player to select options by linking them to numeric keys on your keyboard. Just check **Dialogue options can be selected with number keys?** in the Settings Manager. This option also allows you to trigger options with inputs mapped to **DialogueOptionX**, where *'X'* is the index number of the option to trigger.

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.

You can also open a useful **Conversation Editor** from the top of the Conversation Inspector. From there, you can view all assigned Dialogue Options that the selected Conversation has, and click on them to display their ActionLists in the Inspector window. If you select a new Conversation object, you can quickly revert back to the previous one from the top-left of the window.

Just like Hotspots, Conversations allow you to change the **Interaction source** in their Inspectors. By default, linked Dialogue Options are scene-based lists, but you can also make use of ActionList asset files (see section 5.8). You can also set this to **Custom Script**. You will then be able to send a message to a GameObject of your choice. This is useful if you wish to hard-code your responses instead of relying on Actions.

5.8 - ACTIONLIST ASSETS

Within a given scene, we can place Cutscenes, Triggers, Interactions and Dialogue Options - each of which are ActionLists that can manipulate objects within that scene. But sometimes, we'll want an ActionList to run no matter which scene is currently loaded - for example, when examining an Inventory item. And when working as a team on a large game, we may also want to be able to create ActionLists for a scene without interfering with anyone else's work.

With Adventure Creator, we can create an **ActionList asset**, which is an ActionList that exists as a physical file within your project folder. This asset is created by right-clicking inside the Project window, and choosing Create -> Adventure Creator -> ActionList. Double-clicking this asset will open it within the ActionList Editor.

ActionList assets are mainly used for Inventory interactions, Menu functions, and common tasks that can occur in any scene. For example, the Demo game's Pause menu (as found in the Demo_MenuManager asset) runs the "DeselectInventory" ActionList when it turns on. This ActionList de-selects any active Inventory item, making sure the main cursor is always displayed when navigating the Pause menu.

ActionList assets can also manipulate scene objects. When dragging an object onto a asset's field, a “Constant ID” number will appear beneath it. A **Constant ID** number acts as a unique reference to the object, so that it can still be found by the asset when another scene is open. If a GameObject has no Constant ID when it is assigned to an ActionList asset's field, one will be generated automatically by giving itself a *ConstantID* script. If the GameObject is not currently visible to the field (e.g. because another scene is open), the **Search scenes** button underneath the field can be used to search all scenes listed in the game's Build Settings for the object in question.

When a GameObject has a *ConstantID* script, its ID number can be manually set. If you wish for an asset-based Action to manipulate different objects in different scenes, you can assign the same ID number to each object to ensure that the Action will always be carried out.

ActionList assets can be called by using the **Engine: Run ActionList** Action and setting the **Source** field to **Asset File**. Similarly, Cutscenes and other scene-based ActionLists can also refer to an asset file for their Actions.

ActionLists assets can also be referenced from Hotspots and Conversations in a similar way. This is useful for creating "global" interactions, such as Conversations that can take place over more than one scene.

ActionList assets can be converted to scene-based ActionLists (like Cutscenes), and vice-versa, via the cog icon to the top-right of the Inspector. As scene-based ActionLists cannot be stored as assets directly, if you want to transfer one to another scene, then it is recommended to convert it to an ActionList asset, and then convert it back to a scene-based ActionList in the new scene.

5.9 - 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. When relying on Touch Screen input, you can also activate arrows by swiping in the given direction.

Arrow Prompts are created by clicking **Arrow Prompt** under the Scene Manager's Logic panel, followed by **Add new**. 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 is 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 regular 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.

5.10 - 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 **Sound: Play** 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 its 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 its 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. As speech audio is automatically set to the correct volume without the need for a Sound object, the “Speech” option in the pop-up is only necessary for playing other sounds at the same volume.

If you are working with Unity 5, you have the added option of linking your Sound objects to Audio Mixer Groups. Mixer Groups can be set within the Settings Manager, under **Audio Settings**. Volume parameters for each sound type will also need to entered, and created in the Mixer Group. If an AudioSource has no Audio Mixer Group assigned in its **Output** field, then it will be assigned automatically based on the **Sound type** in the Sound component. This is also true for the AudioSource components used by characters.

The **Sound: Play** 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.

By default, sounds do not carry over when changing scene, but you may wish to have e.g. ambient sounds continue playing as you navigate the game. To have a Sound object survive a scene change, check the **Play across scenes?** checkbox, and move the prefab into the root of your scene's hierarchy. The prefab cannot survive a scene load unless it has no parent GameObject.

Though the Sound object can be used to play music, it is recommended to use the dedicated music system for music playback – see section 5.11.

5.11 - MUSIC

Whereas sound effects and speech audio are generally tied to specific GameObjects in a scene, music tracks can played independently to both the scene, and the objects within them.

By using the **Sound: Play music** Action, music tracks can be played, queued, looped and stopped at any time. The state of the music, and the queued playlist, is saved automatically.

In order to play music using this Action, a music track must first be listed in the Music Storage window. This window can be accessed from the Action, and it's here that AudioClips are assigned, as well as their relative volumes. Only tracks listed in this window will be available to use in the Action.

Note: When music tracks are assigned in this window, the associated data is stored in the Settings Manager. Therefore, be aware that if you change your Settings Manager asset file, you will also have to update the Music Storage window with your tracks.

5.12 - CONTAINERS

A **Container** is a scene-based list of Inventory items (see section 6.1 for more on Inventory items). This allows for gameplay such as treasure chests, that the Player can open and take from within, and storage chests with which the Player can store away items for later use.

A Container can have a default set of items, that can be changed during gameplay, either through Actions or through Menus. To “open” a Container, use the **Container: Open** Action. To view the contents of a Container, your Menu Manager must contain a Menu with an *Appear type* set to **On Container**. The Demo_MenuManager asset provides such a Menu, and with this items can be transferred both ways between the Container and the Player's Inventory.

To store the contents of a Container in save game files, a Container requires that the RememberContainer script be attached, but this is true of all Container prefabs by default.

5.13 - ACTIONLIST PARAMETERS

Sometimes, we'll want our game to perform the same task several times, but with subtle differences each time - for example, whenever the Player picks up an item, we want its associated Hotspot to be disabled, its GameObject to be made invisible, and the Item to be added to the inventory.

ActionList parameters allow us to use the same ActionList to perform the same tasks to different objects - just as parameters do when programming functions. In the example above, parameters could be used to create a single ActionList that disables any Hotspot, hides any GameObject, and adds any Item to the Player's inventory. This ActionList would then be called every time the Player takes an item - with the specific objects referenced by the parameters being set each time.

Parameters are available to Cutscenes, and asset-based ActionLists. To enable them, check **Use parameters?** in the properties box at the top of the Inspector. A new box will appear: enter the number of parameters you wish to use, and you can then re-name and choose the type of each one. Parameters can reference Strings, Floats, Integers, Booleans, GameObjects, Unity Objects (such as Materials and AudioClips), Inventory items, Global Variables and Local Variables - though Local Variables are not compatible with asset-based ActionLists.

Whenever an Action can make use of a parameter, the list of parameters will appear inside the Action as a drop-down box. Choose a parameter to "override" the Action's default field, and have it be replaced by the parameter. Note that this drop-down box will only appear if at least one parameter of matching type has been defined.

Sometimes the function of Actions change based on their field settings. For example, the Variable: Check Action will give an option to check for true or false when querying a boolean, and will give an option to check a numerical value when querying an integer. Note than when a parameter is assigned to an Action, it will assume the same UI and functionality that it had before the parameter was set.

Once an ActionList has been set up to take parameters, these parameters can be passed to it by calling it from another ActionList with the **ActionList: Run** Action. When the **ActionList: Run** Action is used to trigger an ActionList with parameters, you can set the fields for each parameter. You can also use this Action to transfer its ActionList's own parameters that have been set using another such Action.

To set the values of individual parameters, use the **ActionList: Set parameter** Action, or alternatively set them through custom scripting: the ActionList and ActionListAsset classes both store their parameters as Lists, which can be modified through script.

5.14 - DRAGGABLE OBJECTS

Adventure Creator allows for moveable objects that can be manipulated directly with the mouse cursor or touches, creating a greater sense of presence and interactivity - particular with first person games. Such objects fall into two categories: **Draggable** and **PickUp** objects. Draggable objects are constrained: they can only be moved in certain ways: either along a set plane, only rotated, or along pre-defined **Tracks**.

To hold a Draggable object, simply touch (or click) it and hold. The object will be let go when the touch or click is let go. How far away an object can be is determined by the **Moveable ray length** field in the Settings Manager.

The Scene Manager provides a default Draggable prefab, which contains all the components necessary, but no mesh. It's advisable to add the required mesh to it as a child object, because the Draggable's position and rotation is determined by its Drag mode - therefore, adding a mesh as a child object allows its Transform settings to be set safely.

Key to the way a moveable object behaves is its Rigidbody settings. The Drag and Angular Drag values are locked to 20 when an object is held, so it's the Mass value that affects how quickly a held object moves. A Mass of 1 gives a 1:1 relationship between the movement of the mouse or touch and the movement of the object. Higher values will require more movement from the player to move the object, which lower values will require less.

Draggable objects are at their most useful when locked to a Track. There are three Track types, each available as prefabs in the Scene Manager: **Straight Track**, **Curved Track**, and **Lever Track**. When placed in the scene, a Track prefab will display a white line in the Scene window that represents the locus an attached Draggable object can take, with a grey circle at the start, and a white circle at the end. When moving along a Track, a Draggable object works best when its base Collider component is a **Sphere Collider**, so it's a good idea to keep this component even if you add other Colliders to it in child objects.

A **Straight Track** is used to constrain a Draggable object along a straight line. Rotation effects can also be added, to make the object roll as it moves, or turn in a screw-like motion. A **Curved Track** prefab is used to constrain a Draggable object along a curved line. If the line is looped to form a circle, the number of possible revolutions can also be set.

A **Hinge Track** prefab is used to pivot a Draggable object about its centre. Its position is locked, and can only be rotated in a circular motion. Like the Curved Track, it can also be looped. This Track type is useful for objects such as doors and levers. If the camera is going to be looking at a hinged Draggable head-on, it's recommended to enable the **Align drag vector to front** setting.

When a Draggable object is moved, its **Interaction on move** ActionList will be run. If this Interaction begins with a **Moveable: Check track position** Action, it can be looped

onto itself to be run again until the desired position has been reached. The **Moveable: Set track position** Action can be used to move the Draggable object automatically.

Draggable objects can be turned on and off by using the **Object: Send message** Action on them. They start the scene enabled, but this can be changed with the **Remember Moveable** script, which is attached to the prefab by default.

The Moveable_Drag Inspector also allows you to reduce the player's movement when it is being manipulated, which is particularly helpful when creating immersive first-person games.

5.15 - PICKUP OBJECTS

PickUp objects, unlike Draggable objects, can be freely manipulated. To hold a PickUp object, simply touch (or click) it and hold. The object will be let go when the touch or click is let go. How far away an object can be is determined by the **Moveable ray length** field in the Settings Manager.

PickUp objects can also be rotated, brought closer to the screen, and thrown. The required Input axes to perform these extra functions are listed in the Settings Manager's Required inputs section.

The Scene Manager provides a default PickUp prefab, which contains all the components necessary, but no mesh. The prefab uses a Sphere Collider by default, but this can be replaced if necessary with another.

Key to the way a moveable object behaves is its Rigidbody settings. The Drag and Angular Drag values are locked to 20 when an object is held, so it's the Mass value that affects how quickly a held object moves. A Mass of 1 gives a 1:1 relationship between the movement of the mouse or touch and the movement of the object. Higher values will require more movement from the player to move the object, which lower values will require less.

Triggers can be placed in the scene to determine if a PickUp object has been placed in the correct position. A Trigger can be set to detect the PickUp object in question (or all PickUps in general by choosing **Any Object With Component** and entering the component name **Moveable_PickUp**), so that a sequence of Actions will run when the object enters it.

PickUp objects can be turned on and off by using the **Object: Send message** Action on them. They start the scene enabled, but this can be changed with the Remember Moveable script, which is attached to the prefab by default.

The Moveable_PickUp Inspector also allows you to reduce the player's movement when it is being manipulated, which is particularly helpful when creating immersive first-person games.

5.16 - CUSTOM CURSORS

As stated in section 5.1, the Cursor Manager is used to define the Interactions available in your game. You can add, remove and set textures, animate them, as well as define the rules for which cursors appear – such as the ability to display a dedicated "walk" cursor when hovering over a Navigation Mesh.

The Cursor Manager can also be used to determine if cursors are rendered in Hardware or Software mode. Software mode, the default, hides the hardware cursor and displays the correct cursor as a texture in its place. While it can be slower on older systems, it enjoys wider support on more platforms. Hardware mode, on the other hand, replaces the system's hardware cursor completely, and can often be faster. Note, however, that as of September 2014, Hardware cursors do not work correctly in the Unity Editor on Macs – this is a Unity-based problem, but still works correctly in the game is Built.

The "click offset" can also be set for each cursor. In Software mode, this offset represents how far the click point is from the cursor's centre, as a decimal of its size. In Hardware mode, the offset represents how far the click point is from the cursor's top-left, in exact pixels.

Cursors defined under the **Interaction icons** panel can also be referenced in the Menu Manager (see section 11.1), to change the cursor when the mouse hovers over a particular element.

5.17 - QUICK TIME EVENTS

Quick Time Events, or QTEs, are isolated moments of gameplay that require the player to press a key, or a combination of keys, within a time limit. The event is considered "won" if the keys are pressed correctly, and "lost" otherwise. Adventure Creator makes it easy to trigger such events, by using the **Input: QTE Action**. When this Action is run, regular gameplay is disabled, and the Action waits until the player has either won or lost.

QTEs can have several "win" requirements: a single button-press, a button held down for a set time, or a button pressed repeatedly (i.e. "button mashing"). The button name defined in the Action must correspond to an Input button defined in the Input Manager (found in **Edit -> Project settings -> Input** in the top toolbar). What happens when the player wins or loses is dictated by the Action's **If condition is met** and **If conditions is not met** fields respectively.

A Menu name can also be supplied to the Action. So long as this Menu's **Appear type** is set to **Manual**, then it will be displayed automatically for the duration of the QTE - making it suitable to act as a "button prompt" to tell the player what to do. Timer menu elements (see section 11.2) are useful here: a Timer's Timer type can be set to either **Quick Time Event Remaining** (how long longer the QTE will last) or **Quick Time Event Progress** (how much progress the player has made). If such a Timer is visible when a QTE is active, then it will represent that QTE.

Additionally, if the Menu is linked to Unity UI (see section 11.3), then it can also be animated when the player wins, loses, or presses a correct button. To prepare a Unity UI-linked Menu for animating, attach an **Animator** component to the base **Canvas** component. Adventure Creator requires that three animation states be present: **Win** and **Lose**, and either **Hold** or **Hit** (depending on the QTE's type: the Action will describe which states it requires). If not all animations are required (e.g. Win but not Lose), then empty states of the same name can be used instead.

A series of QTE tutorials can be found here.

6.0 - INVENTORY

6.1 - 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. If your **Interaction method** (see section 5.1) is not set to "Context Sensitive", they can also be "given" to NPCs (see section 6.2).

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.

Adventure Creator provides different methods of inventory handling – that is, whether or not Items are selected before choosing the Hotspot to use them on, or whether the Hotspot is chosen before selecting which Item to use. You can also make use of an **Active** and **Selected** texture for each Item, based on your chosen settings. An Active texture is displayed when an Inventory cursor is hovering over a Hotspot, or when the item is selected. The cursor effect that is employed, be it simple or pulsing, can be modified in the Settings Manager. A Selected texture will override the Active within an InventoryBox element, if one is provided.

Alternatively, a **Cursor** can also be defined for the Inventory item, provided that the Cursor Manager has been set to change when an item is selected. This allows you to assign a dedicated Cursor graphic, which may also be animated.

Inventory items can also be sorted into categories. Categories can be created, renamed and removed from the top of the Inventory Manager. Once multiple categories exist, each Inventory item can be sorted into one of them. Categorising inventory items allows them to be sorted when creating inventory menus – for example, by categorising spell items under "Spells", you can create a spell inventory menu that just displays spells.

Inventory items can also have properties assigned to them – see section 6.5.

How Inventory items are manipulated are determined by their various "Interactions" fields. Which fields are displayed depends on what your various interaction options in the Settings Manager are set to. These are outlined in the next section.

6.2 - INVENTORY HANDLING

By default, an Inventory item is selected by left-clicking on its icon within an InventoryBox menu element (see section 11.1 for more on Menus). Once it is selected, it can be used on a Hotspot by creating Inventory interactions within that Hotspot, or be combined with another item.

Items can be examined by right-clicking (or by dropping a selected item onto itself, if set in the Settings Manager). And if you wish for the player to do something other than select it when left-clicking on it (for example, opening a separate "close up" scene), you can override the "select" behaviour.

The Standard interactions panel that appears when an item is being edited in the Inventory Manager allows you to choose what happens for each of these various interactions. For example, you can make an "Examine" interaction, a "Use" interaction, or an interaction for the various combinations with other items.

Such interactions are stored in ActionList asset files - these are scene-independent lists (see section 5.8), which mean they can be run regardless of which scene the player might be in.

Exactly what "Standard interactions" are available, however, depends on your chosen Interaction method, as set in the Settings Manager (see section 5.1). If your game makes use of **Choose Interaction Then Hotspot** mode or **Choose Hotspot Then Interaction mode**, then you can make use of as many interactions as you like for your Hotspots (as set in the Cursor Manager). When in either of these modes, you can opt to make use of these interactions, rather than the standard "Use" and "Examine", on your Inventory items as well. In your Settings Manager, just change the "Inventory interactions" option from "Single" to "Multiple". You will now be able to define an Interaction for each cursor type on your Inventory items.

These two modes also allow you to “give” items to NPCs. When you declare the Inventory interactions on a Hotspot that's attached to an NPC, you can choose whether it's a “Use” or a “Give” interaction. The **Use (item) on (object)** syntax can be set globally within the Cursor Manager, or per-item in the Inventory Manager. By default, Inventory interactions are to be “used” when selected – if done so manually using the **Inventory: Select** Action, however, you can change this. You can also select items in “give” mode using the method below:

When set to "Multiple", you can still select Items by using the **Inventory: Select**" Action in their interaction asset files. However, this can be quite tedious if you always want the same cursor type to select an item - for example, "Use". So, you can opt to make this the default behaviour: in the Settings Manager, check **Select item if Interaction is unhandled?**, and you can choose which cursor mode will select an item. The same method can be used to select items for "giving" them to NPCs.

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.”

In addition to defining global unhandled events, you can also define per-item unhandled events, which override the defaults. Found under the **Standard interactions** section of an Inventory Item's editor, you can use these to elicit a response from the Player that's unique to that Item.

It is also possible to define per-Hotspot unhandled events, which will take precedent over any unhandled event defined in the Inventory Manager. This is useful for Hotspots like trashcans, where you can make the player say something like “I don't want to throw that away.” when any item is used on it.

You can also pass the Hotspot that was clicked on to initiate the unhandled interaction as a GameObject parameter (see Section 5.13) – just click **Pass Hotspot as GameObject parameter?** at the top of the Inventory Manager. This will apply for per-Hotspot interactions as well.

6.3 - 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 5.3.

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.

6.4 - CRAFTING

As well as combining two items together to solve puzzles, you can also define crafting “recipes” that allow for multiple items to be combined to create a new one. This allows for Minecraft-style crafting elements in your game.

Beneath the list of Items in the Inventory Manager, you can declare any number of recipes. Each recipe requires a number of Items as “ingredients”, and a resulting Item that is produced when the ingredients are combined. Recipes can optionally be made to require a specific crafting pattern – that is, the arrangement of ingredients in the Crafting menu element.

If an ingredient's Item has **Can carry multiple** checked, you can also determine the number of instances of this item required. For example, a recipe to create a lit torch may require one empty torch and two batteries.

To craft items, ingredients must be placed into a Crafting menu element. A Crafting element has two types: Ingredients and Output. When the correct arrangement of items are placed in a Crafting box of the Ingredient type, the resulting item can be selected from a Crafting box of the Output type. If the recipe (as declared in the Inventory Manager) has **Result is automatic** checked, the resulting Item will appear instantly in the Output box – otherwise it will require the **Inventory: Crafting** Action to be run to create the recipe.

To demonstrate the appropriate use of the Crafting menu element, both the Demo and Demo2D Menu Manager assets provide a Crafting menu, which you can copy into your own Menu Manager. Note that this menu's **Appear type** is set to **Manual**, and you will have to decide for yourself how this Menu opens in your game – whether it be by the **Menu: Change state** Action, or opening from another Menu, or some other means.

6.5 - INVENTORY PROPERTIES

Inventory properties are variables that you can define for Inventory items, which each item having its own value for each variable. This makes it possible to give an item "stats" such as weight or value. Properties can either be defined for all items, or for those within a specific category. Properties can be created and managed within the **Properties** tab of the Inventory Manager.

When an item can make use of a property, it is listed at the bottom of that item's Settings panel, and its values can then be set.

An item's properties can be displayed in a Label menu element (see section 11.2), but its true value lies when used with custom scripting. The GetProperty function within the InvItem class can be used to retrieve any property, and unique values, that the item holds. The RuntimeInventory script holds all inventory items held by the player, allowing including the variable lastClickedItem, which stores the last item that the player clicked on.

See the scripting guide for a more thorough description of public functions and variables available to custom scripts.

7.0 - VARIABLES

7.1 - DECLARING VARIABLES

You can define both Global and Local Variables. **Global Variables** are scene-independent, and can be retrieved and modified regardless of scene, while **Local Variables** are specific to a scene, and cannot be read outside of that scene.

Variables are used to to keep track of progress, and alter gameplay accordingly. 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.

Variables can be one of five types:

Boolean: Either “true” or “false”
Integer: A whole number
String: Some text
Float: A number with a decimal point
Pop Up: A string chosen from a drop-down list of pre-determined choices

All Variables are defined in the Variables Manager, but Local Variables can only be defined once **Organise room objects** has been clicked in the Scene Manager. Variables can be either booleans (true-or-false flags), integers (whole numbers), floats (numbers with decimals) or strings (a line of text). 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. If you have many Variables, you may find it easier to locate them in Actions if you placed a forward-slash in their name (e.g. “Options/IsFullScreen”), as they will be grouped up in lists according to the letters before the slash.

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. For debugging, the “live” values of Variables can be seen during gameplay by checking **Show realtime values?** at the top of the manager.

As well as being used “behind the scenes” to affect game logic, Variables can also be displayed on screen, either via a MenuLabel (see section 11.2), via a Character's dialogue, or when typing formulas into the **Variable: Set** Action. The token syntax **[var:ID]** will be replaced by the value of the associated Variable. For example, [var:2]

will display the value of Global Variable 2 listed in the Variables Manager. A Variable's ID number is listed next to it in the Variable Manager – note that this number is not necessarily the same as its order in the list.

Local Variables make use of a slightly different token syntax: **[localvar:ID]**.

7.2 - MANAGING VARIABLES IN-GAME

Variables are set using the **Variable: Set** Action, and queried using the **Variable: Check** Action. The Variable: Set Action can also be used to transfer the value of a MenuInput Menu Element (see section 11.2) to a String Variable, and Mecanim parameter values to Boolean and Integer Variables.

The **Variable: Copy** Action can be used to transfer a Variable's value to another of the same type – this is useful for temporarily backing up a value, or for converting a Local Variable to a Global one.

For coders: when the game begins, the PersistentEngine object's **GlobalVariables** component makes a copy of the global variables, keeping them separate from the Variables Manager during runtime. These are stored in a public list called globalVars, and – like inventory items – rely on ID numbers to determine which variable is being referenced. Local variables are stored in the GameEngine object's **LocalVariables** component, inside the localVars list. The Scripting guide has more information on these scripts.

7.3 - LINKING WITH PLAYMAKER VARIABLES

If you have the popular PlayMaker asset, which is a separate Unity asset to Adventure Creator, you can sync Adventure Creator's Global Variables with PlayMaker's Global Variables. To do so, simply select the Variable you wish to link, and change its **Link to** value to **Playmaker Global Variable**.

If you have not done so already, you will be prompted to add the **PlayMakerIsPresent** scripting define symbol to your game's Player Settings. You can find this field from Edit → Project Settings → Player.

Once you have entered this, you can then enter the name of the PlayMaker Global Variable you wish to link to. Bear in mind that the two variables must match type: if you are linking a PlayMaker float, you must do so with an Adventure Creator float as well.

You can also choose whether or not PlayMaker determines the initial value of the Adventure Creator Global Variable. Generally, your game should only affect either the PlayMaker or Adventure Creator Variable, rather than both. When a PlayMaker Variable is changed, its value is “downloaded” to Adventure Creator only when it is needed – i.e. when the Variable: Check Action is used to determine its value.

By linking Variables in this way, you can save the value of PlayMaker Global Variables automatically.

7.4 - LINKING WITH OPTIONS DATA

Global Variables can be used to form custom options data, which is independent of save game data. See section 9.4 on how to do this.

7.5 - VARIABLE PRESETS

Variable presets allow you to bulk-assign all Global or Local Variable values at once. This is particularly useful for debugging and testing, since you can use them to quickly assign your variables to states they'll be at specific points in your game.

Presets are listed and defined in the **Preset configurations** panel of the Variables Manager. Once a preset has been created, you can assign a Variable's preset value within the Variable's properties panel. A preset value will the Variable's default value until it is modified.

When the game is running, a preset can be assigned by selecting it in the Variables Manager and clicking **Bulk-assign**. Presets can also be assigned by using the **Variable: Assign preset** Action, which can be useful if you need to ensure all players have the exact same variable values at some point during gameplay.

8.0 - MISCELLANEOUS SCRIPTS

8.1 - SHAPEABLE

The **Shapeable** script is used to manage multiple blend shapes that are present on the **Skinned Mesh Renderer** that it is attached to.

When working with blend shapes, it is often the case that you will want to group some together, and only ever have one within a group to be “active”, while the others are not. For example, you may want to group a Character's various eyebrow expressions into a group, so that only one can be active at a time.

Attach the Shapeable script to a Skinned Mesh Renderer, and you will be able to define as many **shape groups** as you like. A group can contain any number of **shape keys**, which each correspond to a different blend shape.

Once a Shapeable script has been set up, it can then be manipulated by the **Object: Blend shape** Action. This Action can be used to make one key in a group the “active” one – all others will be disabled. This can be performed over time, however, for smooth transitions.

This concept is used by the NPC Brain in the 3D Demo. The ExpressionHappy and ExpressionSad blend shapes are grouped together, and called upon in the Basemene scene.

8.2 - MOVEABLE

In order to manipulate a GameObject's Transform component with the **Object: Transform** Action, the **Moveable** script must be attached. Simply attach the script to the GameObject, and its Transform can be manipulated.

8.3 - PARALLAX 2D

In games made in Unity 2D mode (as set by the **Moving and turning** popup in the Settings Manager), the camera does not physically move. When it pans sideways, the perspective remains fixed.

As this happens, all objects in the scene will move across the game window at the same rate, regardless of their distance from the camera. Therefore, the **Parallax 2D** script can be used to achieve a depth affect by causing them to move with the camera – the speed at which it follows will determine how far away it feels.

Attach the **Parallax 2D** script to a background sprite, and assign a Depth value. The more positive the value, the further away it will seem. The more negative, the closer it will seem. This script is used on the foreground and background objects in the 2D demo's Park scene.

For more advanced effects, it is also possible to limit the parallax movement to within pre-set boundaries in both the X and Y directions. Just check **Constrain?** within each directional box to set upper and lower bounds.

8.4 - LIMIT VISIBILITY

Particularly in 2.5D games, you may wish for an object to be visible only when a particular camera is the active one. Attach the **Limit Visibility** script, and you can limit its visibility to a certain camera – and optionally its children, too.

8.5 - ALIGN TO CAMERA

The **Align To Camera** component is used to aid in the correct placement of scene sprites when building 2.5D games. 2.5D games require that scene sprites (i.e. those that the player can walk in front of and behind) are placed in 3D space, so that the player can be viewed from a perspective-correct camera. Therefore, it's necessary to align these sprites such that they face a camera that may not be pointing along an axis - which can be difficult when trying to do manually.

By attaching the Align To Camera component to a scene sprite, you can have it automatically face a camera. Once it is aligned, its depth (the distance from this camera) can be controlled within the component's Inspector. Optionally, you can lock the sprite's percieved scale when the depth is adjusted. This will cause the sprite to get larger as it moves further away, ensuring it has the same "screen-space" when viewed through the camera.

8.6 - PARTICLE SWITCH

When you create a Unity Particle System, you may wish to turn it on at some point during gameplay, rather than play it continually. For example, a fireplace would only need to produce smoke when it's lit.

With the **Particle Switch** script, you can turn the Particle System on and off easily with the **Object: Send message** Action. Add the script to your Particle System, then point to it with the Object: Send message Action. The “Turn On” and “Turn Off” messages will perform as expected, but the “Interact” message will cause it to emit all of its particles once.

8.7 - LIGHT SWITCH

When you create a Unity Light, you may wish to turn it on at some point during gameplay, rather than play it continually. For example, a lamp would only emit light if the player has plugged it into a wall socket.

With the **Light Switch** script, you can turn the Light on and off easily with the **Object: Send message** Action. Add the script to your Light, then point to it with the Object: Send message Action. The “Turn On” and “Turn Off” messages will perform as expected.

8.8 - SPRITE FADER

In order to manipulate a Sprite's transparency with the **Object: Fade sprite** Action, the **Sprite Fader** script must be attached. Simply attach the script to the Sprite, and its transparency can be manipulated.

8.9 - TINT MAPS

If your game is in 2D (and makes use of **Unity 2D** for your **Moving and turning** setting - see section 1.5), then you can make use of Tint maps to alter the colour of sprites as they move around a scene. This allows you to easily create dynamic lighting effects, such as having your Player get darker when they enter a shaded portion of the background.

A Tint map can be created under the Camera section of the Scene Manager, and assigned as the default at the top - underneath **Scene settings**. A new Tint map will appear 10 units in the Z-axis, but its Z-position is not actually important, as it can be hidden when the game begins. What is important is its scale in the X and Y directions - after creating it, stretch it out so that it covers the same area as your background graphic. You can then supply a "tint" texture to its Inspector. This texture will tint any sprites that "follow" it - but pure white will not have an effect. Such sprites will be tinted according to their position over the Tint map.

IMPORTANT: The texture you supply must be readable by Unity. This is a simple but crucial step: within its properties Inspector, set its **Texture Type** to **Advanced**, and check **Read/Write Enabled**.

To make a sprite follow a Tint map, simply add the **Follow Tint Map** script component. This component will normally follow the scene's default Tint map, as defined in the Scene Manager, but you can also supply a separate Tint map if you prefer.

You can also adjust the intensity of the tinting effect. These values can also be changed mid-game by using the **Object: Change Tint map** Action - allowing you to change the Tint effect dynamically, e.g. when the player turns on on a light switch. Just be sure to add the **Remember Visibility** component to the sprite as well if you do – as this will ensure any such changes are recorded in save game files (see section 9.2).

A tutorial on working with Tint maps can be found here.

CHAPTER II: ADVANCED FEATURES

9.0 - SAVING AND LOADING

9.1 - 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.

A player can save or load a game by accessing a SavesList menu element (see section 11.2). After saving or loading, an ActionList asset can be optionally run – which is useful for doing things like setting up menus correctly afterwards.

The name of save files is – by default – the same as your project name, but you can replace this from the Settings Manger, under **Save game settings**. It is here that you can also tell Adventure Creator to save a screenshot as well as a save file (note: Screenshots are disabled for WebPlayer and Android platforms), the maximum number of allowed save files, and their display order in SavesList Menu Elements (see section 11.2)

When a game is saved, Adventure Creator stores two types of data: main and room. Main data includes the player's position, the player's standard animations (2D sprites only), the camera's position, the current scene, and the state of the inventory, global variables, and menus. 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, its 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.

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 9.5.

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

Save games can also be loaded using the **Save: Save or load** Action. A more advanced save menu can be created by using this Action in conjunction with ActionList parameters (see section 5.13). This Action can also be used to only load a save file selectively, so that only certain elements of a save file are loaded.

For coders, saved games can be accessed via the *SaveSystem* script, which is attached to the PersistentEngine, created at runtime. SaveSystem contains several static functions to aid in save management: *LoadSave*, *SaveGame*, *SaveNewGame*, *LoadAutoSave*, *GetNumSlots* and *GetSaveSlotName*. Most 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:

```
SaveSystem.LoadGame (0);
```

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.

9.2 - SAVING INDIVIDUAL OBJECTS

Saving object data within a scene is done by adding the correct component to that object. Starting with v1.50, you can have Adventure Creator attempt to "auto-tag" your game's objects with the correct components for you – just click **Auto-add save components to GameObjects** at the top of the Settings Manager.

However, it is still important to understand the principles involved in saving scene object data – in case your requirements are slightly different:

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* script 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 its associated GameObject will be saved. The following is a list of the various *Remember* scripts, and when they are used.

RememberAnimator – Attaching this script to any Animator component will cause the current animation, parameter values and layer weights of that Animator to be saved. If an animation is in mid-transition when saving occurs, only the "transition-to" animation will be saved. Note that this component is only available in Unity 5.

RememberCollider – Attaching this script to any Collider or Trigger will make sure the enabled state of its Collider component will be saved.

RememberContainer – Attaching this script to any Container object will make sure the items stored within will be saved. This is a part of the Container prefab by default.

RememberConversation – Attaching this script to any Conversation object will make sure the on/off/lock state of its 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 its visibility to the player's cursor, any changes in its name, and any changes in its Interaction states are saved.

RememberMaterial – Attaching this script to any Renderer object will record any changes in its Materials. It will search your game's Resources folder when looking for Materials to place on it, and these must have unique filenames.

RememberMoveable – Attaching this script to any Draggable or PickUp object will record its position and rotation, as well as the position along any track it may be attached to.

RememberName – Attaching this script to any GameObject will record any changes in said GameObject's name.

RememberNavMesh2D – Attaching this script to any NavMesh2D prefab will record any changes in its hole structure. Holes (in the form of Polygon Colliders) can be added and removed from NavMesh2D prefabs during gameplay to add pathfinding obstacles.

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. Changes in “standard animations” (walk, idle, etc) are also saved – if the NPC uses Legacy animation, the animations must be placed in a Resources asset folder and have a unique filename.

RememberShapable – Attach this to any GameObject that also has a Shapeable script. Shapeable scripts can be used to manipulate blend shapes on a Skinned Mesh Renderer, and the RememberShapeable script will record any changes made.

RememberSound – Attach this to any Sound object that you wish to save. What its playing, and its track position will be recorded. If you need to record its change in sound clip (if it changes mid-game, for example), just place the AudioClip in question inside an asset folder named Resources.

RememberTransform – Attach this script to any GameObject to save its transform data. Can optionally save a GameObject's presence in the scene, if it is deleted or instantiated during gameplay – but only if the GameObject is a prefab inside a Resources asset folder. Can also be used to save which GameObject is its parent, provided that GameObject has either a *ConstantID* script of its own, or is an assigned Hand Bone of a Character.

RememberTrigger – Attach this script to any Trigger prefab to save its on/off state.

RememberVisibility – Attach this script to any GameObject to save its renderer component's “enabled” state, and optionally that of its children. This works for both mesh and sprite renderers. When attached to a sprite renderer, it can also be made to record its colour or transparency. It will also save the state of any

attached Sprite Fader (see section 8.8) or Follow Tint Map component (see section 8.9).

AdventureCreator will save the data of any object with these scripts attached. Objects marked to be saved have a save icon beside them in the Hierarchy window. To reduce the size of save files, it is wise to use them only when necessary. For example, the 3D Demo's opening Conversation, IntroConv, has no *RememberConversation* script, since the states of its DialogueOptions never change.

Some scripts also allow the state they record to be "off" when the game begins. For example, a Hotspot can be disabled by default, by attaching a RememberHotspot script to it, and setting its **Hotspot state on start** to **Off**.

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 Paths 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 its Transform, not just Animation component) have a **RememberTransform** script.

Animation, however, cannot be saved so easily. It is for this reason that the **On load** Cutscene field is provided in the Scene Manager – as this will be run whenever a scene is loaded via a save game file. In this Cutscene, you can set up any animation states that you need to, based on the values of Global or Local variables that you've defined. The 3D Demo scene, Basement, provides an example of this.

9.3 - 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 game 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.

Script functions to programatically-load an auto-save, and others, can be found in section 12.7.

9.4 - 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 a key that is based on your game's name.

All Adventure Creator games have five options by default: whether or not subtitles are on, the game's language, and the volume levels of music, speech, and sound effects. These options can be changed in the Options Menu that both the Demo_MenuManager and Demo2D_MenuManagers carry. You can also view and edit this data, as well as reset them to their default values, in the Settings Manager.

Options data is loaded when the game begins, and saved whenever a change is made to any of them.

It is possible to create custom options in your game by way of Variables. The **Link to** property of a Global Variable, as listed in the Variables Manager, can be set to **Options Data**. When this is done, the Variable's value will be stored in the PlayerPrefs, and not in save game files. You can then use the **Variable: Set** Action, or **Cycle**, **Toggle** and **Slider** Menu Elements to affect the value.

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.

9.5 - 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 its 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 its 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*.

9.6 - LOADING SCREENS

If your game features complex scenes, or it is played on older hardware, it may take a few seconds to transition between scenes. In this case, you may wish to create a loading screen, that appears during this pause to alert the player that the game is loading.

You can do this by creating a dedicated "loading" scene, which is displayed during transitions. This does not need to be an "Adventure Creator" scene - it can merely be a camera with a sprite texture in front of it.

Create a scene you wish to act at the loading screen, and add it to your game's list of Scenes in build from the Build Settings. Then, check the **Use loading screen?** box in the Settings Manager, and supply the scene number of your loading screen.

Additionally, or instead of, you can opt to make use of asynchronous loading. This feature allows you to load scenes in the background, allowing animation to continue for a short time while the next scene loads. By checking **Load scenes asynchronously?** in the Settings Manager, you can then provide a delay time before and after the load process – which is useful if you want some nice loading effects.

For example, you can create a “Loading” menu with a progress bar – a technique that is covered in this tutorial.

9.7 - IMPORTING SAVES FROM OTHER GAMES

If you are making a multi-episodic game that spans multiple projects, you can use the SavesList menu element (see section 11.2) to import saved games from a previous project. This will import the save file's Global Variables (and thus the player's choices) into the current game.

When a SavesList's **Click action** is set to **Import**, you must supply an **Import product name** (as set in the other project's PlayerSettings window), and **Import save filename** (as set in the other project's Settings Manager). Note that there are three requirements for this to work:

- The other project's Company name (as set in the PlayerSettings window) must be identical to the current project.
- The two projects must share exactly the same Global Variables – it is recommended to copy the VariablesManager asset and use it in both projects.
- Because of Unity's security features, this feature only works on standalone platforms (PC, Mac and Linux).

When this SavesList is displayed, it will then list any import files it can find. If the player chooses a file, an ActionList can be defined that will run if the import is successful.

It is also possible to only limit the available import files to those in which a particular Global Variable has been set to true. This is useful if you only want players to be able to import a save if they have reached the end of the previous game.

9.8 - SAVE PROFILES

Player profiles allow you to separate save game files and options settings by the player who created them. This is useful for a number of reasons: for instance, it means that one player cannot accidentally delete another player's save files. It also allows options such as the language to be unique to the person playing. However, they are quite an advanced topic, and it's recommended to be familiar with Menus, ActionLists, and ActionList parameters before working with them.

Profiles can be enabled at the top of the Settings Manager: **Enable save game profiles?**. Your game will automatically create a new profile if none exist - it is not possible to have a game with no profiles. You can now use the ProfilesList Menu Element (see section 11.2). This element will list all profiles created by the user - though the currently-active profile can optionally be hidden. When a profile in this menu element is clicked, it will be selected. To display the current profile non-interactively, a Label can be created with a **Label type** of **Active Save Profil**e.

Profiles can be created, renamed, and deleted using the **Save: Manage profiles** Action. When a new profile is created or renamed, its name can be set by the value of a String Global Variable (see section 7.1). You can have the player enter a name of their choice by using an Input menu element, and using the **Variable: Set** Action to store the Input box's contents in the String Global Variable. When a profile is deleted, any associated save game files will also be deleted, so you may want to have make a confirmation box appear before performing this.

To provide the ability to rename or delete profiles in the form of Button Menu Elements beside your list of profiles, it is recommended to make use of ActionList parameters (see section 5.13) to condense the number of ActionLists you need to make. If a Button Menu Element is set to run an ActionList that has an Integer parameter, then the parameter can be set within the Button's properties. If you set this parameter to match the slot index number of the profile list beside it (indices start from zero), you can use just one **Save: Manage profiles** Action to handle the deletion of any profile.

A basic Profiles menu is included in the Demo_MenuManager asset file, and can be used to create, delete and switch profiles. To make use of it, enable profiles within the Settings Manager, and then un-hide the ProfilesButton element within the Pause menu.

10.0 - SPEECH MANAGEMENT

10.1 - 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 its 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 → Game editor). Clicking **Gather lines** 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 (among other things). It will also do the same to lines contained in any ActionList assets referenced by your game.

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. By default, lines spoken by the Player are denoted as "Player" - if you only rely on one Player prefab, the Speech Manager will give you an option to rely on the prefab's actual name instead (though you will need to gather the lines again to update them).

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.

Alternatively, you can choose to manually assign audio files to a speech line by un-checking **Auto-name speech audio files?** in the Speech Manager. You can then assign files via the field that appears when clicking on a line's entry below. Note that if your game makes use of lip-syncing files, then these too will have to be manually assigned.

Audio files can now be made for the game. By default, audio files are found automatically according to their name – the expected folder and filename can be found by clicking on a speech line's entry within the Speech Manager. You can generate a file to give to voice actors, or to use as a reference sheet for which filename each line needs, by exporting a script sheet (see section 10.3).

If a speech line has no associated Character, it is considered a narration line, and the audio file is named *Narration*. In order to play narration audio, your scene must either contain a Player prefab with an **AudioSource** component, or have a **Default Sound**

prefab assigned in the Scene Manager.

Audio files for speech are placed in a Speech subfolder inside your Resources directory. Viewing a speech lines properties will display its expected filename and folder – if **Place audio files in speaker subfolders?** is checked, then audio files will need to be placed in subfolders with the name of the speaking character. For example, Brain21's audio file would be placed in Resources/Brain/.

If your game's character engine is set to Legacy (see section 3.1), you can use the **Dialogue: Play Speech** Action to set a specific animation to play per line. However, Adventure Creator also features a number of ways to animate your Character's lips when talking, and these are covered in sections 10.5 and 10.6).

Adventure Creator also supports “audio ducking” - while speech audio plays, all other audio can be made to quieten slightly so that the speech can be better heard. The amount by which SFX and Music volumes are reduced are set within the Speech Manager.

10.2 - MANAGING TRANSLATIONS

Adventure Creator comes with full translation support for your game's text, specifically:

- Speech lines
- Hotspots and Dialogue Option labels
- NPC names (if set to something other than their GameObject's name)
- Journal pages
- Menu text
- Inventory Item names
- Pop-up and String Variables.
- Cursor names and prefixes

A game's active language is stored in its Options Data (see section 9.4) and is controlled by the player in the Options menu.

Before you can translate text, you must first gather it in the Speech Manager (using the same **Gather Text** button as discussed in section 10.1). Once text is gathered, clicking on it within the Speech Manager will reveal a number of properties, including any translations if they exist. Lines of text can be filtered by scene, type, and more – including any custom description you choose to give a line of text when viewing its properties.

Within the Speech Manager, you can add and remove supported translations underneath the Languages panel. While translations for each line of text can be edited within the Speech Manager, it is recommended to edit them in an external spreadsheet.

You can click **Export translation** to generate a CSV file which contains your game's text. Be sure to click **Gather text** beforehand, to properly collect each line to be translated. You can use a spreadsheet application, such as Excel or OpenOffice, to edit the CSV file and click **Import translation** within the Speech Manager to update the translation. Note that the CSV delimiter is a pipe (|) character. As an example, the demo game comes with a French translation.

If your game supports multiple translations, and you wish to update them all from just one file, you can also use the Speech Manager to export and import all translations in a single CSV file. HTML script sheets can also be created for each translation.

You can also tell Adventure Creator to only rely on translated text, and ignore the original text that was entered in Unity inspectors. This is useful if you would like to update your game's main language in a CSV file as well – you can simply create a new translation that serves as the game's main language (even if it's in the same language as the original game text), and use that as the default translation.

You can also play alternative speech 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*.

If you are using an Arabic translation, text will need to read right-to-left. To that end, if your translation is labelled “Arabic”, then Input menu elements will automatically be flipped.

Translations can be edited or imported at runtime by accessing the **RuntimeLanguages** script during gameplay (see section 12.7).

10.3 - SCRIPT SHEETS

Once your game's speech text has been collected in the Speech Manager (see section 10.1), it can be exported as an HTML script sheet to hand out to voice actors. Within the Speech Manager, click **Create script sheet...** to bring up the Script Sheet window.

By default, all speech lines listed in the Speech Manager will be exported as an HTML file. However, the Script Sheet window gives you the option of limiting these lines by character name, or by Speech Tag. If your game supports multiple languages, you can also select which language to export lines in.

Speech Tags are labels that you can assign **Dialogue: Play speech** Actions, and are useful if you want voice actors to just get a script with lines related to a particular cutscene or interactive sequence. They are created within the Speech Manager by clicking **Edit speech tags**. You can add and remove tags in the window that then pops up.

Once tags are enabled and defined, any ActionList (both scene- and asset-based) that contains a **Dialogue: Play speech** Action can be assigned to one within its list of properties. When the speech lines within are next gathered into the Speech Manager, they will be listed with this tag.

10.4 - SPEECH TOKENS

Speech tokens are snippets of text that, when inserted into a Character's line of dialogue, have a dynamic effect. The following tokens are recognised:

[continue]
If the dialogue this is placed in is not running in the background, then from this point onward it will be. This is useful if, for example, you want to cut the camera on a particular word, mid-sentence.

[hold]
Like [continue] above, the ActionList will carry on when this token is displayed on screen. However, the speech itself will remain on the screen indefinitely, until the **Dialogue: Stop speech** Action is used to end it. This is useful, for example, if you want a character's last-spoken line to remain on the screen when the player is presented with a dialogue option.

[expression:Name]
Changes the character's expression, if **Use expressions?** is checked within their Inspector, to the one named “Name”. More on expressions can be found in section 10.7.

[var:ID]
Replaces the token with the value of a Global Variable, where “ID” is the ID number of the referenced Variable. The replacement token of any Variable is listed in its properties in the Variables Manager.

[localvar:ID]
Replaces the token with the value of a Local Variable, where “ID” is the ID number of the referenced Variable. The replacement token of any Variable is listed in its properties in the Variables Manager.

[wait]
Removes the token, and only displays the speech text up to the point at which it was placed. The Character will not continue speaking until the player clicks/taps. Note that **Subtitles can be skipped?** must be enabled in the Speech Manager.

[wait:X]
Removes the token, and only displays the speech text up to the point at which it was placed. The Character will wait X seconds before continuing to speak. The value of X can be either an integer or a decimal.

[token:ID]
Replaces the token with a string assigned by calling the SetCustomToken function inside RuntimeVariables, where “ID” is the ID number of the custom token. A tutorial on using this token can be found here.

10.5 - FACEFX INTEGRATION

FaceFX is a popular application used to create facial animation from speech audio, and can be used on Characters inside Adventure Creator. To begin using FaceFX's XML files, you must first download and import the official FaceFX Unity plugin from http://unitydemos.facefx.com.s3.amazonaws.com/FaceFXBonesMorph.unitypackage.

Next, go to the Speech Manager and set **Lip syncing** to **Face FX**. You will be prompted to add the **FaceFXIsPresent** scripting define symbol to your game's Player Settings. You can find this field from Edit → Project Settings → Player.

Adventure Creator will now auto-detect any FaceFX scripts attached to your Characters when they speak. These scripts must be, or must derive from, **FaceFXControllerScript_Base**, but can be attached either to the root Character gameobject, or on a child gameobject. If both a script and speech audio clip (see section 10.1) is detected, Adventure Creator will attempt to play an XML file of the same name as the clip, but with “Default_” inserted before it.

For example, if the audio file is named “Player2”, then the assumed XML name is “Default_Player2”.

10.6 - LIP SYNCING

Animating your Characters convincingly makes a big difference to a game's quality, and there are a number of methods available. For Legacy-based Characters, you can opt to play a custom facial animation for each line of dialogue. This animation is set explicitly within each **Dialogue: Play speech** Action.

However, it's often unfeasible to keyframe an animation for every line of dialogue, and this option is only available to 3D characters. So, Adventure Creator provides several ways to animate Characters' lips automatically – a process known as lip-syncing.

Aside from the FaceFX integration mentioned in section 10.5, Adventure Creator has support for a number of free alternatives, that can be used to animate 2D sprites, 3D models, as well as portrait graphics.

Bringing lip-syncing to your game is a two-step process. First, the dialogue to be spoken must be broken down into a list of phonemes, or lip-shapes. Second, the phonemes are mapped onto the speaking Character in some way – whether it be onto the Character's gameobject, or their portrait graphic. All of these settings are handled in the **Speech Manager**.

Within the Speech Manager, the **Lip syncing** setting determines how the phonemes for each line of dialogue are found. Setting this to **FaceFX** requires no more steps other than those described in the previous section. The other available options are:

From Speech Text
This option will generate phonemes automatically based on the speech text. This option is best used when there is no accompanying audio – it won't always be a totally accurate approximation, but it will give the Character's animation some noticeable variety.

Read Pamela File
This option will make use of the phonemes generated by a Pamela file. Pamela is a free Windows application that can generate phonemes, and is a good choice for fine-tuning a lip-syncing animation. It can be downloaded from http://users.monash.edu.au/~myless/catnap/pamela3/.

Read Sapi File
This option will make use of the phonemes generated by a SAPI file. SAPI is another free Windows application, and is a good choice for quickly and automatically creating phonemes in bulk. Adventure Creator's Physics demo makes use of this option. SAPI can be downloaded from http://www.annosoft.com/sapi_lipsync/docs/index.html.

Read Papagayo File
This option will make use of the phonemes generated by Papagoyo, a free, cross-platform lip-sync tool that's easy to use. It can be downloaded from http://www.lostmarble.com/papagayo/.

Salsa 2D
This option will make use of the 2D lip-syncing features of SALSA With RandomEyes, which is a separate Unity asset available to purchase from http://crazyminnowstudio.com/projects/salsa-with-randomeyes-lipsync. While the Salsa 3D script component can be used on 3D characters independently of Adventure Creator, Salsa 2D cannot – because 2D characters can face multiple directions, and therefore need different sets of "talking" frames. To get around this problem, simply choose this option, and add the **Salsa 2D** to your character's base object (which should also have an AudioSource), and ignore its sprite fields. Adventure Creator will instead make use of the sprite animations you provided in the NPC / Player components, and use Salsa 2D to perform the lip-syncing processing. You will also need to add the SalsaIsPresent Scripting Define Symbol – see section 12.1.

Rogo Lip Sync
This option will make use of Rogo Digital's Lip Sync, which is a separate Unity asset available to purchase from https://www.assetstore.unity3d.com/en/#!/content/32117. LipSyncData files are generated as normal, and must then be named/placed according to the description below. The lip-sync animation will then be played automatically provided the speech line has an ID, and the speaking character has a LipSync script component attached to the root GameObject.

Pamela, SAPI, Rogo Digitial LipSync, and Papagayo files are automatically detected by Adventure Creator in the same way that audio speech files are (see section 10.1). They must be placed in a **Resources/Lipsync** folder, and be of the same filename as they're relevant audio file, only with a **.txt** extension (or **.asset** for Rogo Digital LipSync). For example, if an audio file is "Resources/Speech/Player2.mp3", its accompanying phoneme file would be "Resources/Lipsync/Player2.txt". Lipsync files can also work with translations.

For each of the options listed above, you will also need to map each phoneme to an animation frame. Click on the **Phonemes Editor** to bring up the mapping UI. Multiple phonemes can be mapped to the same frame by separating them with a slash "/". The **Revert to defaults** button will map appropriate phonemes to your chosen Lip-sync method, but it will likely require further tweaking.

Once your phonemes are being generated and mapped to frames, you can now use them to animate your Characters. The **Perform lipsync on** setting chooses how: **Portrait** will animate a Character's portrait graphic (assuming it's an animated texture), **Portrait And Game Object** will also animate the Character's GameObject, and **Game Object Texture** will animate a texture on a Character's Skinned Mesh Renderer. The method by **Portrait And Game Object** affects the Character is based on your Character's chosen animation engine:

For **Sprites Unity**-based Characters, each lip-sync frame will correspond to a frame in

the Character's talking animation. This animation is assumed to be of the same number of frames as have been declared in the Phonemes Editor.

For **Sprites Unity Complex**-based Characters, the current lip-sync frame can be output to the Mecanim controller by declaring a **Phoneme integer** parameter in the Character's Inspector.

For **Legacy** and **Mecanim**-based Characters, lip-syncing works by manipulating blend shapes. Each lip-sync frame will be mapped to a particular blend shape, as declared by the **Shapeable** script. All blend shapes used to animate the mouth must be placed in the same group (see section 8.1 for more on shape groups), and the group to affect is then declared in the Character's inspector. The Physics demo makes use of blend shapes to animate Brain in the opening cutscene.

To use **Game Object Texture** mode, a **LipSyncTexture** script component must be attached to the Character's root gameobject. Once attached, it will provide texture replacement fields that correspond to each phoneme frame.

Before lip-syncing features were introduced, Adventure Creator also featured a method of moving a 3D 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.

10.7 - FACIAL EXPRESSIONS

If a character uses Mecanim animation (see section 3.7), or relies on portrait graphics when speaking (see section 3.1), then their expression can be changed mid-speech by using the **[expression:Name]** token within the speech text itself (other tokens are available as well - see section 10.4). The "Name" part of this token refers to the label given to an expression defined in the Player or NPC Inspector, underneath **Dialogue settings**, once **Use expressions?** is checked. Here, multiple expressions can be created and managed - each with their own portrait graphic, and ID number.

The ID number is fixed, and displayed just above the expression's "Name" field. If your character uses Mecanim animation, then your chosen **Expression ID integer** parameter (as set under the Mecanim parameters panel) will be set to this value when the expression is triggered.

A tutorial on working with character expressions can be found here.

11.0 - MENUS

11.1 - OVERVIEW

Each element of the game's user interface – from the Pause and Options menus to the Hotspot label and dialogue option list – are created using Menus.

Adventure Creator's menu system is designed to be powerful yet simple to use. All Menus are defined within the **Menu Manager**. From here, you can define your game's Menus, and edit both their appearance and functionality.

Menus can either be designed with Adventure Creator's built in Menu system, or by referencing one made with Unity UI (see section 11.3).

The demo game's Menu Manager asset provides a fully-functioning menu system for an adventure game. Rather than create a new menu system from scratch, you may prefer to duplicate this asset and then edit it to your liking. Note that Menu Manager assets group up all Menu and Menu Element assets together - you will have to select all the assets (or just the root asset) before duplicating to properly copy it.

The following is a run-down of the default menu system:

Pause – Displays buttons to access the Options, Save and Load menus, as well as to resume gameplay or quit the game. Appears either when player presses the "Menu" axis (as defined in the Input Settings), or clicks the "Menu" button on the InGame menu.

Options – Allows the changing of various in-game options, such as audio levels and language. Accessed only via the Pause Menu.

Save – Lists available saved games to overwrite, with an option to save a new file if the limit has not been reached. Accessed only via the Pause Menu.

Load – Lists available saved games to load. Accessed only via the Pause Menu.

Inventory – Displays any Inventory items held by the player. Accessed by hovering the mouse over the top of the screen during normal gameplay, once the player is carrying something.

InGame – Displays a single button to access the Pause Menu. Always visible during normal gameplay.

Conversation – Lists the available Dialogue Options in the active Conversation. Only visible when a Conversation is active.

Hotspot – Displays a text label showing the name of any Hotspot or Inventory item underneath the cursor. Only visible during normal gameplay.

Subtitles – Displays the name and dialogue of the currently speaking Character. Only visible if Subtitles are set to On from within the Options Menu.

Interaction – Displays a list of available Interaction choices for a selected Hotspot. Not used by the demo game - only visible when the game's Interaction method is set to Choose Hotspot Then Interaction (see section 5.1).

Container – Displays the contents of the active Container, and also the Player's inventory. Clicking on item in one list will transfer it to the other.

When editing a menu, you can choose to have it display in your Game window when your game is not running, to get a live preview of how it will look. Just click the **Test in Game Window?** checkbox at the top of the Menu Manager.

Upon selecting a Menu to edit, its properties box will appear beneath the list of defined Menus. From here you can give it a title (for reference purposes only), assign a background texture, choose its orientation, position, size, and more. The main property to define is its **Appear type**. Menus can be made to appear and disappear automatically, based on their Appear type. The Appear type can be set to one of the following:

Manual – Deactivates the Menu by default. Can only be shown and hidden using the Menu: Change State Action or through scripting. Pause this menu if you wish to access it via a keyboard or controller.

Mouse Over – Activates the Menu only when the player's cursor hovers over its defined area.

During Conversation – Activates the Menu only when a list of dialogue options, by way of a Conversation prefab, is made available. Pause this menu if you wish to access it via a keyboard or controller.

During Cutscene – Activates the Menu whenever gameplay is blocked, but not paused. This can be used to add e.g. black borders on the screen during a cutscene. The **Clickable in cutscenes?** option must be set for a Menu to be interactive at this time.

On Container – Activates the Menu when a Container has been opened using the **Container: Open** Action.

On Input Key – Identical to Manual, only a Toggle key can also be used to activate and deactivate the Menu. The key's name must match that of an axis in the Input settings.

On Interaction – Activates the Menu once the player clicks to select a Hotspot.

Will deactivate again when the cursor leaves its defined area (as set in the Settings Manager). Only for use with the Interaction method Choose Hotspot Then Interaction (see section 5.1). Pause this menu if you wish to access it via a keyboard or controller.

On Hotspot – Activates the Menu whenever the cursor is hovering over a Hotspot or Inventory item during normal gameplay.

When Speech Plays – Activates the Menu when a speech line is active, but only if Subtitles are set to On. By default, the menu will only be used for the most recently-activated line – if a line is still playing in the background, for example, then the text will be removed. However, by checking **Duplicate for multiple lines?**, each speech line will be displayed in its own copy of the menu. This is useful for games in which dialogue lines overlap, and subtitle text is displayed above those speaking. Can additionally be filtered to only display for Character (either all characters or specific ones), Narration lines (those without an associated Character), and/or lines that play normally or in the background.

During Gameplay – Activates the Menu only during normal gameplay.

Menus can be locked, and won't be shown until unlocked, by checking the **Start game locked off?** checkbox. They can also be locked via the **Menu: Change state** Action in-game.

Some Menus can be set to pause the game when activated. The Menu Manager allows for a **Pause background texture** to be defined, which will cover the screen when this happens. The “GUI depth” (a Unity property) of Adventure Creator's menus can also be set, in case of conflict with other GUI scripts and assets.

In the case that a Menu is purely for the display of information (for example, a score bar), it can be made non-interactive by checking **Ignore Cursor clicks?** within its properties box.

Menus can also be made to run an **ActionList asset** when they turn on or off (see section 5.8). The Demo game's Pause menu makes use of this by triggering an Action that deselects the active Inventory, ensuring the main cursor is always active when the Pause menu is active.

11.2 - MENU ELEMENTS

A Menu is nothing until Elements are added to it. Menu Elements make up a Menu's functionality, in the form of Labels, Buttons, Icons and more. When a Menu is selected, its Elements are listed underneath the properties box. From here you can create, remove and edit Elements, as well as re-arrange their order.

Appearance settings such as font, colour and size are defined on a per-Element basis, allowing for deep customisation of Menu appearances. All elements can separately made visible and invisible, both by default or changes during gameplay via the Menu: Change State Action. If an Element's Position popup is set to **Aligned**, it will arrange itself automatically in line with other Elements.

Before you make a new Element, you must first choose its type, via the **Element type** pop-up box. Elements vary in appearance and functionality depending on their type. Some types, such as Labels, are non-interactive while others, such as Buttons, can be clicked on. When an Element is selected, its properties box is displayed.

The following list of Element types are available:

Button – A simple button that can be clicked on. The **Click type** defines what happens when clicked on. Among the available options are **Run Action List** and **Custom Script**, both of which are described in the following section. Buttons can also be used to scroll through a list of inventory items, turn pages in a journal, and simulate input keys. If an ActionList that has an Integer parameter is assigned to run when clicked (see section 5.13), then the value of this parameter can be set within the Button's properties.

Crafting – Provides a grid for placing down crafting ingredients, and a space for the resulting output Item. Change the **Crafting element type** to choose between these two functions.

Cycle – A label that cycles through elements of a string array when clicked on. The **Cycle type** defines its link to other data, and can be used to affect integer Variables. An ActionList can be assigned to run when clicked.

DialogList – Displays Dialogue options of the active Conversation. Can either be a list of all available options, or be forced to always display the "n'th" option. Its click functionality is handled automatically. If the active Conversation has more active options than the DialogList is set to display, the visible options can be shifted by using a Button Element with a **Click Type** of **Offset Element Slot**.

Drag – Allows either the entire Menu or a single Element to be dragged within a pre-defined boundary. This also works on Elements that are larger than the Menu they're contained in, making it useful for displaying documents that are larger than the screen.

Graphic – Allows a texture to be drawn. Unlike the Label element, which can

also display a texture, the one in a Graphic can also be animated – if the texture is made up of frames of equal size.

Input – A label that can be edited via keyboard when the element is active. Can be used for password-input puzzles. Optionally, a Button on the same Menu can be "triggered" when the Enter key is pressed. The text value can be set to a String variable by using the **Variable: Set** Action.

Interaction – Displays a single Icon as defined in the Cursor Manager. Its click functionality is handled automatically, based on the game's **Interaction method** (see section 5.1).

InventoryBox – Displays a list of inventory items carried by the player – either with icons or with names. If its **Inventory box type** is set to **Hotspot Based**, this list will be further limited to items catered for by the active Hotspot. If instead it's set to **Default**, the number of slots shown can be limited. If this is the case, and more items are carried than displayed, a Button Element with a **Click Type** of **Offset Element Slot** can be used to shift the displayed items. If it's set to **Display Selected**, it will show the icon of the currently selected item – note that this is for visual purposes only. However, if set to **Display Last Selected**, it will provide the last-selected item as a clickable icon, providing an easy way for the player to re-select their last-used item.

Journal – Displays text from a number of pages. Pages can be flipped through by using a Button Element with a **Click Type** of **Offset Journal**. New pages can be added during gameplay, or existing ones removed, via the **Menu: Change state** Action. The active page can be set manually using the **Menu: Set Journal page** Action. Variable and custom tokens described in section 10.4 can also be used in the label's text. If you want to display two pages at once, you can create two Journals side-by-side, and have the second reference the first by changing it's Type to **Display Existing Journal**.

Label – Displays a text box that cannot be clicked on. If the **Label type** is not set to **Normal**, the text label will be replaced in-game automatically. For example, setting it to **Hotspot** will cause the label to change to the currently-active Hotspot, and setting it to **Variable** will cause the label to show the value of pre-defined Variables. Variable and custom tokens described in section 10.4 can also be used in the label's text.

ProfilesList – If save game profiles are enabled (see section 9.8), this element will display any profiles that the user has created. Clicking on a profile will make it the active one, and any lists of save games will be updated to reflect the new profile. The currently-active profile can be optionally left out.

SavesList – Displays either a list of saved games found in the file system, or a specific save slot number. The **Click action** is used to determine if clicking loads them, overwrites them, or imports them from another project (see section 9.7). If set to **Save**, an option to save a new game can be made to show if the number of

save files does not exceed the maximum set. Optionally, an ActionList asset can also be run once the game is saved. If the Settings Manager has been set to take screenshots when saving, you can also change the Display type to show screenshots – with or without an accompanying label. By default, the saving/loading will be handled automatically when clicked on, but this can be prevented to allow for different click-handling – such as deleting or renaming save files.

Slider - Draws a horizontal bar whose width represents a value between a lower and upper value (by default, this is 0 and 1). The **Slider affects** field can be used to link it with one of the three sound types in the game, link to a float Global Variable, or allow its value to be defined via a custom script. A Slider can optionally be made to ignore mouse clicks.

Timer - Draws a horizontal bar used to represent the time remaining in a timed Conversation or Quick Time Event, the progress made in a Quick Time Event, or the progress made while loading a new scene (if **Load scene asynchronously?** Is checked within the Settings Manager – see section 9.6).

Toggle - Similar to a Cycle, only the available values are either On or Off. The **Toggle type** field can be used to link it with the visibility state of subtitles, affect boolean Variables, or allow its value to be defined via a custom script. An ActionList can be assigned to run when clicked.

Elements that can be clicked on can also be assigned two sound clips: one for when the Element is made active (i.e., when the mouse hovers over it), and another for when the Element is clicked on. Before such sound clips will play, however, a **Default Sound prefab** must be defined in your scene, as it is through this prefab that menu sounds will play. To create a prefab, click **Sound** from the list of objects in the Scene Manager, and then drag it into the Default Sound prefab box underneath **Scene Settings**.

11.3 - UNITY UI INTEGRATION

Unity 4.6 brings a new UI system, and Adventure Creator can link them with Menus - allowing for things like button clicks and label text to be controlled by Adventure Creator automatically. The **Menu: Change state** and **Menu: Check state** Actions can also be used with them. Adventure Creator will only control exactly what it's told to: if a UI component is not "linked" to Adventure Creator, then the user has full control over it.

To see Adventure Creator's Unity UI integration quickly, you can use the **New Game Wizard** to create a UI-linked Menu Manager by choosing Default Unity UI as your starting GUI type. The New Game Wizard can be accessed by choosing Adventure Creator → Gettings started → New Game Wizard from the top toolbar.

To integrate an existing UI into Adventure Creator, create a Menu in the Menu Manager, and change its Source to either **Unity Ui Prefab** or **Unity Ui in Scene**. All visual controls will then disappear from the properties list. In its place, **Linked Canvas** and **RectTransform boundary** fields will appear. The Linked Canvas should be the root object of the UI, and the RectTransform boundary should reference the "bounding box" - this can be invisible, but is necessary for Adventure Creator to determine whether of not the mouse is currently "over" a UI.

If the source is set to **Unity Ui in Scene**, Adventure Creator will only be able to manipulate the UI if it's present in the scene when the scene loads. This is useful for UIs rendered in World Space.

If the source is set to **Unity Ui Prefab**, then the Linked Canvas field must reference the prefab instead.

For Unity UI objects to function properly, an Event system must be present in the scene. At the top of the Menu Manager, you can supply an **Event system prefab** that will be automatically added to the scene at runtime. If none is supplied, and no such system is present in the scene, Adventure Creator will create one.

The Event system can be used to handle keyboard and touch input easily. If you enter an Element's name into the Menu's **First selected Element** property, then that Element will be selected automatically when the Menu is turned on – ideal for Menus that are keyboard-controlled.

By default, Button and Interaction elements linked to Unity UI will respond to Unity's “PointerClick” event (i.e. a full click-and-release). However, this can be set to “PointerDown” (i.e. no release needed) within the element's properties box.

UI-linked Menus have different **Transition type** and **Position type** options to regular Menus. A simple "fade" transition can be easily implemented by setting Transition type to **Canvas Group Fade** - just place a Canvas Group component on your Canvas object. For more complex animation, select **Custom Animation**. You must then place an Animator component on your Canvas object, with four states: **On**, **Off**, **OnInstant** and **OffInstant**, which will be called automatically when the UI is turned on, both instantly

and over time. An example controller can be found in your Assets folder, under AdventureCreator -> UI -> AnimatedMenu_Example.

Each UI component you wish to manipulate must be linked to an appropriate Menu Element. Each Element will present a field for its expected component - for example, a Label element will expect a UI Text component. When one is assigned, a Constant ID number will be generated for it and displayed underneath. It is this ID number that links the two together.

Elements that make use of multiple slots (for example, InventoryBox elements) will require one UI component per slot. When the number of available slots exceeds the number of used slots (for example, when the player carries fewer inventory items that the InventoryBox element can display), you can choose to either disable the slot GameObject, or clear its contents. The latter is necessary if you wish to allow item re-ordering in your Inventory menu, since unused slots must still be active in the scene for them to receive items.

When viewing a prefab in the Project window, Unity won't let you select sub-children - making it impossible to find the UI component needed to link to an element. If this is the case, simply drag the UI into a scene, and assign the UI component from there. Adventure Creator will record the Constant ID number - this is what's important, so click "Apply" at the top of the UI's Inspector to save the new Constant ID number back into the prefab, and remove it from the scene.

11.4 - MENU SCRIPTING

Aside from simple commands such as turning Menus off or displaying labels, Adventure Creator's Menus require extra steps to give them functionality. One of the available options for a Button's **Click type** is to **Run Action List** - specifically an ActionList asset (see section 5.8). Additionally, ActionLists can also be run when a Menu is turned on or off.

The Demo game's Menu Manager relies on an ActionList asset when the player clicks the Quit Button in the Pause Menu - the ActionList runs the **Engine: End game** Action to exit the game.

Menu functionality can be customised much more heavily through scripting. Whenever a Menu is activated, a function called **OnMenuEnable** is called within the **MenuSystem** script (found within Assets -> Adventure Creator -> Scripts -> Menu). The activated Menu is sent to the function as a parameter, so that specific commands can be issued to it.

Most clickable Elements, such as Buttons, can also call the **MenuSystem** script when clicked on by the player. Changing a clickable Element's "type" (for example, a Button's Click type) to Custom Script will cause the OnElementClick function to be run, with click data sent as parameters.

11.5 - SCENE-BASED MENUS

The **MenuLink** script links Menus to scene objects, making it possible for your game to have 3D menus. To make a scene-based Menu, the Menu must first be created as normal within the Menu Manager. Since it won't be directly seen by the player, its appearance is not important. Make sure its **Appear type** is set to **Manual**, and that **Enabled on start** is left unchecked.

A scene-based Menu relies on a separate GameObject for each Menu Element. If an Element has more than one “slot” (for example, a SaveList), each slot must also be a separate GameObject. Attach the MenuLink script to each GameObject, and then use the Inspector to enter the name of its associated Menu and Element. If a GUIText component is also attached to the object, its Text field will be set to the linked Element's label. Otherwise, you can access the Element's label from the MenuLink script's **GetLabel** function.

When the MenuLink script's **Interact** function is called, either through custom scripting or by using the **Object: Send message** Action, the game will react as though the Element itself was clicked on. If the Element is a Button, you can use an ActionList asset (see section 5.8) to manipulate objects in your scene, for example moving the Camera to focus on another Element.

CHAPTER III: EXTENDING FUNCTIONALITY

12.0 - INTEGRATING NEW CODE

12.1 - SUPPORTED THIRD-PARTY ASSETS

Adventure Creator can be integrated with a number of other Unity assets out-of-the-box. To allow Adventure Creator to "talk to" third-party assets, you may need to add a particular Scripting Define Symbol to your Player settings – this is just a piece of text that tells Adventure Creator that it can safely run certain lines of code without causing errors.

To add a Scripting Define Symbol, simply choose Edit → Project settings → Player from the top toolbar, and add the symbol in the text box underneath **Scripting Define Symbols**. Multiple symbols can be placed in this box – just separate them with a semicolon (**;**).

Below is a list of third-party Unity assets that can be integrated with Adventure Creator, together with their respective symbols.

2D Toolkit

Description:
2D Toolkit is an alternative sprite animation system to Unity's built-in 2D system, and can be used to animate Characters. To animate a Character with 2D Toolkit, see section 3.8.

Web address: **www.unikronsoftware.com/2dtoolkit**
Scripting define symbol: tk2DIsPresent

Cinema Director

Description:
Cinema Director is a timeline-based cutscene tool that can be used to make complex cinematics. The **Third party: Cinema Director** Action can be used to trigger Cinema Director cutscenes within Adventure Creator ActionLists.

Web address: **www.cinema-suite.com**
Scripting define symbol: CinemaDirectorIsPresent

FaceFX

Description:
Face FX is an industry standard application that produces automatic facial animation based on audio. To add FaceFX lipsyncing to your Characters, see section 10.5.

Web address:
http://unitydemos.facefx.com.s3.amazonaws.com/FaceFXBonesMorph.unitypackage
Scripting define symbol: FaceFXIsPresent

OUYA

Description:
The OUYA is an Android-based console that Unity games can easily be published to. To link Adventure Creator's input settings to the OUYA's, see section 2.10.

Web address: www.ouya.tv
Scripting define symbol: OUYAIsPresent

PlayMaker

Description:
PlayMaker is a popular visual scripting system for Unity. Adventure Creator can call PlayMaker Events with the **Third Party: PlayMaker** Action. Global Variables can also be linked to PlayMaker's Variables – see section 7.3 for more.

Web address: www.hutonggames.com
Scripting define symbol: PlayMakerIsPresent

Rogo Digital LipSync

Description:
Rogo Digital LipSync 3D lip-syncing Unity asset. Adventure Creator can be used to play lipsync animations generated by this asset when characters speak, as described in section 10.6.

Web address: https://www.assetstore.unity3d.com/en/#!/content/32117
Scripting define symbol: RogoLipSyncIsPresent

SALSA With RandomEyes

Description:
SALSA With RandomEyes is a 2D and 3D lip-syncing Unity asset. While 3D characters made in Adventure Creator can make use of Salsa's 3D component without conflict, 2D

characters require special set up – as described in section 10.6.

Web address: crazyminnowstudio.com/projects/salsa-with-randomeyes-lipsync/
Scripting define symbol: SalsaIsPresent

Ultimate FPS

Description:
Ultimate FPS is a popular first-person control system for Unity. Characters made with Ultimate FPS can be used as Adventure Creator players – see section 2.7.

Web address: www.visionpunk.com
Scripting define symbol: UltimateFPSIsPresent

12.2 - CUSTOM SCRIPTS

A script reference for all of Adventure Creator's public variables and functions can be found at adventutrecreator.org/scripting-guide. The reference page for any AC component can also be accessed by clicking on the “Help” icon in the upper-right corner of its Inspector (in Unity 5.1 and above).

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.

The ParticleSwitch script (found in AdventureCreator → Scripts → Object) can be used in this manner to turn particle systems on and off. Simply attach the script to a Particle System, and then use the **Object: Send message** Action to call its **TurnOn** and **TurnOff** functions to enable and disable it respectively. Similarly, the LightSwitch script can be used to enable and disable Light sources.

As another 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.

If you want to avoid Actions and hard-code a gameplay-blocking Cutscene, you can use the StartCutscene function to place the game in Cutscene mode (see this tutorial).

It may also be the case that you need to check Adventure Creator's “game state”, which determines if you're in normal gameplay, a cutscene, the game is paused, or if dialogue options are presented. This can be done every frame by checking the variable **AC.KickStarter.stateHandler.gameState** (see section 12.7). However, a more efficient way to check for changes is to write a script that implements the **IStateChange** interface, and place it on the GameEngine prefab in the scene. This interface provides an **OnStateChange** function that is called whenever the gameState variable changes value.

12.3 - CUSTOM ACTIONS

For an introduction to Actions and their functions, refer to the section 5.2.

Each Action used by Adventure Creator is a self-contained script file. When **Set directory** is clicked, the Actions Manager searches a custom folder for scripts with the ".cs" file extension, and includes them in the list of available Actions. Only one folder can be defined at a time, so all custom action files must be placed together.

Each Action is a subclass of the Action base class, and its implementation and inspector GUI are written together. By writing a new Action subclass script and "registering" it in the Actions Manager, it can be included for use in Cutscenes, ActionLists and other game logic.

To be properly visible inside the Actions Manager, a new Action must have its *title* text field and category enum field defined within its constructor. An override function called *ShowGUI* is used to display variables inside its inspector.

An ActionLists' Actions are stored inside a list variable. When an ActionList runs, it calls upon its 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 its 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.

A step-by-step tutorial on writing Actions can be found online here.

12.4 - CUSTOM PATHFINDING

Adventure Creator provides three methods of pathfinding – Mesh Collider, Unity Navigation, and Polygon Collider (see sections 2.11 to 2.13). Each method is written in a separate script, which are all subclasses of the **NavigationEngine** ScriptableObject class. Which script is used in a scene is determined by the **Pathfinding method** option in the Scene Manager.

To integrate a new pathfinding script, set the **Pathfinding method** to **Custom**, and then enter the name of your NavigationEngine subclass into the box that appears beneath.

You can refer to the Scripting Guide to see what overridable functions are in the NavigationEngine script.

The only essential function is GetPointsArray, which takes two Vector3s as inputs and returns a Vector3 array that describes the path. Other functions, such as SetVisibility and SceneSettingsGUI can be used to better integrate the method into your workflow, but are not necessary.

For the script to be useable when working with Unity's 2D view (i.e. make use of Physics2D raycasts), the is2D boolean must be set to **true**. This can be done within the OnReset function, which is called when the scene begins.

12.5 - CUSTOM ANIMATION ENGINES

As explained in section 3.1, characters in Adventure Creator can be animated with Legacy, Mecanim, Sprites, and 2D Toolkit. The animation engine used is selected at the top of the character's NPC or Player component Inspector. However, it is also possible to animate characters with other animation engines: the option "Custom" can also be chosen.

When the **Animation engine** pop-up is set to **Custom**, you then need to supply a **Script name**. The referenced script must already exist, and needs to be a subclass of the **AnimEngine** ScriptableObject class.

Each of the default animation engines have their own such script, found in Assets -> AdventureCreator -> Scripts -> Animation. They work by overriding functions that are called when a character must be animated. For example, when a character walks, the script's PlayWalk() function is called every frame. The script can also be used to change the behaviour of the "Character: Animate" Action when affecting characters with a particular animation engine.

The functions below can be overridden in a custom animation script. Its **character** variable can be used to access the character's NPC/Player script properties.

The following are called every frame, depending on what the character is doing:

PlayIdle ()
PlayWalk ()
PlayRun ()
PlayTalk ()
PlayJump ()
PlayTurnLeft ()
PlayTurnRight ()

The following can also be overridden:

CharSettingsGUI ()
: Used to display any additional GUI settings the character's Inspector may require

ActionCharAnimGUI (ActionCharAnim action)
: Used to display the "Character: Animate" Action's GUI

ActionCharAnimRun (ActionCharAnim action)
: Called when the "Character: Animate" Action is run

ActionCharAnimSkip (ActionCharAnim action)
: Called when the "Character: Animate" Action is skipped

A full list of the variables and functions available in NPC and Player scripts can be found online here.

12.6 - CUSTOM MOTION CONTROLLERS

By default, a character's motion is handled automatically. However, you can also set their Motion control field to **Just Turning** or **Manual**.

When set to either, Adventure Creator will leave the character's positioning to a separate motion controller. When set to **Just Turning**, Adventure Creator will rotate the character when idle. In either case, Adventure Creator will still calculate what the character's position and rotation "should" be - which custom animation controllers can make use of.

This feature is made use of by the **NavMeshAgentIntegration** script, which is an example of how an Adventure Creator character can move using a NavMeshAgent component instead.

A tutorial on writing a "bridge script" to another motion control system can be found online here. The tutorial also lists commonly-used functions and variables in the Player / NPC scripts that are useful when writing such a script.

A full list of the variables and functions available in NPC and Player scripts can be found online here.

12.7 - COMMON FUNCTIONS AND VARIABLES

A script reference for all of Adventure Creator's public variables and functions can be found here. A tutorial on writing an Interaction through script can be found here.

Adventure Creator has a number of public functions and variables that are useful when writing custom code. All scripts make use of the "AC" namespace. The most important thing to note before starting is that all managers and engine scripts can be accessed in-game by referencing their associated static variable in the KickStarter script. For example, the SceneChanger script can be accessed with:

AC.KickStarter.sceneChanger

Below is a list of the most fundamental:

Variables

AC.KickStarter.player
The current Player prefab

AC.KickStarter.mainCamera
The MainCamera prefab

AC.KickStarter.stateHandler.gameState
The GameState enumeration that dictates whether the game is Normal (regular gameplay), Cutscene, DialogOptions, or Paused.

AC.KickStarter.mainCamera.attachedCamera
The camera that the MainCamera is currently attached to

AC.KickStarter.settingsManager
The Settings Manager. **All Managers and GameEngine / PersistentEngine scripts can be accessed this way**. Additionally, a Manager asset file can be temporarily assigned by setting it to this variable.

AC.KickStarter.playerInput
The PlayerInput script component. Any component attached to the GameEngine or PersistentEngine GameObjects can be accessed this way.

AC.KickStarter.runtimeInventory.selectedItem
The currently selected inventory item.

AC.KickStarter.runtimeInventory.selectedItem.id
The ID number of the currently selected inventory item.

AC.KickStarter.runtimeInventory.hoverItem
The unselected inventory item underneath the mouse cursor

Functions

AC.KickStarter.stateHandler.StartCutscene ()
Places the game in Cutscene mode.

AC.KickStarter.stateHandler.EndCutscene ()
Takes the game out of Cutscene mode.

AC.KickStarter.playerInteraction.GetActiveHotspot ();
Returns the active Hotspot.

AC.KickStarter.playerInput.GetMousePosition ();
Returns the cursor (or last touch) position in absolute pixels.

AC.KickStarter.playerInput.SetTimeScale (float _timeScale);
Sets the speed of the game whenever menus are not pausing gameplay.

SaveSystem.SaveGame (int saveID);
Attempts to load the save game of number saveID.

bool SaveSystem.LoadGame (int saveID);
Attempts to load the save game of number saveID. Returns false if unsuccessful.

SaveSystem.LoadAutoSave ();
Attempts to load the last-made autosave.

AC.KickStarter.mainCamera.GetFocalDistance ();
Returns the MainCamera's focal distance, as set by its current GameCamera

AC.KickStarter.playerInput.SimulateInputButton (string button);
Simulates the pressing of an AC-recognised button, such as FlashHotspots

AC.KickStarter.playerInput.SimulateInputAxis (string axis, float val);
Simulates the pressing of an AC-recognised axis, such as CursorHorizontal

AC.PlayerMenus.GetMenus ();
Returns a List of Menu classes as set by the Menu Manager

AC.PlayerMenus.GetMenuWithName (string menuName);
Returns a Menu named *menuName*

AC.PlayerMenus.GetElementWithName (string menuName, string menuElementName);
Returns a Menu Element named *menuElementName* inside a Menu named *menuName*

AC.PlayerMenus.SimulateClick (string menuName, string menuElementName, int slot);
Simulates a left-click on slot *slot* of Menu Element *menuElement* in Menu *menuName*.

AC.KickStarter.playerMenus.RecalculateAll ();
Recalculates the size and position of all Menus. Call this after a change in screen resolution.

AC.KickStarter.playerMenus.RebuildMenus (MenuManager menuManager)
Rebuilds all Menus according to those defined in the "menuManager" asset. This allows you to completely change your Menus mid-game.

AC.PlayerQTE.GetRemainingTimeFactor ();
If a QTE is active, returns the time remaining as a decimal

AC.PlayerQTE.GetProgress ();
If a QTE is active, returns the progress made by the player

AC.ActionListManager.KillAll ();
Abruptly ends all currently-running ActionLists

AC.ActionListManager.EndAssetList (ActionListAsset actionListAsset);
Abruptly ends an ActionList asset file

AC.ActionListAssetManager.EndList (ActionList actionList);
Abruptly ends a scene-based ActionList

AC.AdvGame.RunActionListAsset (ActionListAsset actionListAsset);
Runs the ActionList asset file *actionListAsset*

AC.Options.GetLanguage ();
Returns the index number of the currently-selected language

AC.Options.SetLanguage (int i);
Sets the index number of the currently-selected language to "i"

AC.Options.GetLanguageName ();
Returns the name of the currently-selected language

AC.KickStarter.runtimeVariables.GetSpeechLog ();
Returns all game text spoken since the last time the game was loaded, in an array of SpeechLog classes. The SpeechLog contains a string of the dialogue text (fullText), a string of the character who spoke it (speakerName) and the ID of the line if set by the Speech Manager (lineID).

AC.KickStarter.stateHandler.GatherObjects ();
Rebuilds the various arrays of known scene objects. This should be called after any object is removed or added to the scene through script.

The following functions are used for retrieving and setting Variable values. "_id" refers to the Variable's ID number, which is displayed to the left of its name in the Variables Manager.

AC.GlobalVariables.GetVariable (int _id);

AC.GlobalVariables.GetIntegerValue (int _id);
AC.GlobalVariables.GetBooleanValue (int _id);
AC.GlobalVariables.GetStringValue (int _id);
AC.GlobalVariables.GetFloatValue (int _id);
AC.GlobalVariables.GetPopupValue (int _id);

AC.GlobalVariables.SetIntegerValue (int _id, int _value);
AC.GlobalVariables.SetBooleanValue (int _id, bool _value);
AC.GlobalVariables.SetStringValue (int _id, string _value);
AC.GlobalVariables.SetFloatValue (int _id, float _value);
AC.GlobalVariables.SetPopupValue (int _id, int _value);

AC.LocalVariables.GetIntegerValue (int _id);
AC.LocalVariables.GetBooleanValue (int _id);
AC.LocalVariables.GetStringValue (int _id);
AC.LocalVariables.GetFloatValue (int _id);
AC.LocalVariables.GetPopupValue (int _id);

AC.LocalVariables.SetIntegerValue (int _id, int _value);
AC.LocalVariables.SetBooleanValue (int _id, bool _value);
AC.LocalVariables.SetStringValue (int _id, string _value);
AC.LocalVariables.SetFloatValue (int _id, float _value);
AC.LocalVariables.SetPopupValue (int _id, int _value);

12.8 - CUSTOM SAVE DATA

Adventure Creator makes it possible to write scripts that extend its saving capabilities – allowing you to save both scene and global data.

It is possible to write your own Remember scripts (see section 9.2) and use them to save your own data within a scene. Such a script must derive from the **Remember** class (of **AC** namespace) and contain SaveData and LoadData functions. A tutorial for writing a custom Remember script can be found online here.

It is also possible to save and load custom global data using script hooks. A global save script must implement the **ISave** interface, be attached to the **PersistentEngine** prefab, and contain both PreSave and PostLoad functions. A tutorial is available here.

12.9 - REMAPPING INPUTS

The **PlayerInput.cs** script uses custom functions to detect input, which are called throughout Adventure Creator in place of Unity's standard functions, such as Input.GetButtonDown. These functions can also be overridden using delegates, meaning your game's control scheme can be changed on the fly, or integrated with a third-party input manager asset.

The following table shows the available functions that can be overridden using delegates:

Unity function	PlayerInput function	Delegate override
bool Input.GetButtonDown (string name)	bool InputGetButtonDown (string name)	bool InputGetButtonDownDelegate (string name)
bool Input.GetButtonUp (string name)	bool InputGetButtonUp (string name)	bool InputGetButtonUpDelegate (string name)
bool Input.GetButton (string name)	bool InputGetButton (string name)	bool InputGetButtonDelegate (string name)
float Input.GetButtonAxis (string name)	float InputGetAxis (string name)	float InputGetAxisDelegate (string name)
Vector2 Input.mousePosition	Vector2 InputMousePosition (bool cursorIsLocked)	Vector2 InputMousePositionDelegate (bool cursorIsLocked)
bool Input.GetMouseButtonDown (int button)	bool InputGetMouseButtonDown (int button)	bool InputGetMouseButtonDownDelegate (int button)
bool Input.GetMouseButton (int button)	bool InputGetMouseButton (int button)	bool InputGetMouseButton (int button)

A tutorial on using these delegates in practice can be found online here.

12.10 - CUSTOM EVENTS

In addition to the input delegates listed in the previous section, a number of general-purpose events exist at specific times during gameplay – for example whenever a character speaks.

These events can be listened to by custom scripts, to aid in the integration of third-party assets or added-on features. Unlike the input delegates, these events to not override regular functionality – they are simply called in addition.

These static events are defined in EventManager script. A list of them, together with descriptions, can be found in the Scripting guide.

A tutorial on working with custom events can be found here.

13.0 – HOW IT WORKS

13.1 - 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 also contains two functions that can disable and re-enable all of Adventure Creator's systems (see section 13.4).

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 its 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.

13.2 - MANAGERS

Each Manager is its own asset file. A project can have multiple Managers of the same type in its 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 movement method). The global variables defined in the Variables Manager, however, are kept separate at runtime. The GlobalVariables script, attached to the PersistentEngine prefab, store a local copy of the Global 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.

13.3 - 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 its 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 its 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.

13.4 - PLAYER CONTROL

The Player is controlled indirectly, via six 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
- **PlayerQTE** – Handles the state of any active quick-time event.

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. You can disable individual scripts by using the **Engine: Manage systems** Action.

When you wish to add your own custom Player control without affecting the other scripts, it's recommended that your Player prefab uses the Mecanim engine for animation, so that you can script parameter changes without interfering with the other systems. If you wish to disable Adventure Creator at any time, even if it is for a brief time such as while firing a weapon, you can easily do so by accessing the KickStarter script. Since it relies on objects created at runtime, calls to it in *Awake ()* will not work, but *Start ()* will work just fine. The code to disable and re-enable Adventure Creator is as follows:

```
AC.KickStarter.TurnOffAC ();

AC.KickStarter.TurnOnAC ();
```

13.5 - GAME DEBUGGING

Adventure Creator has a few features that aid in debugging:

A list of all currently-active ActionLists can be displayed in the Game window, along with the game's current state. This list can be shown by clicking **Show active ActionLists in Game window?** in the Settings Manager.

Actions can be marked as breakpoints, causing the Unity Editor to pause just before they are run – allowing the user to check the state of a scene at that point in time. Actions can be toggled as breakpoints via their context menu to the top-right of their node.

The **ActionList: Comment** Action is mainly used to help keep track of what Action chains do within the ActionList Editor window, but they also have the ability to print their contents to the Console window when run.