User Manual
v1.33

INTRODUCTION

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

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.

Two 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 iceboxstudios.co.uk/adventure-creator/demo and iceboxstudios.co.uk/adventure-creator/2Ddemo.

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

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 tags and layers that will need to be defined before your game will run properly. Before you do anything else, create the following tags in the Tags Manager:

GameEngine, PersistentEngine, **FirstPersonCamera**, **BackgroundCamera**

And the following layers, also in the Tags Manager:

NavMesh, **BackgroundImage**

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

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

Next, the Input Manager will need to be updated. Which axes you will need to define depend on what control method you decide to make use of – refer to 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.

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 it's 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 tags, layers and input axes as described.

If you 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 quick-buttons to creating prefabs like Hotspots and Markers.

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

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

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

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

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

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**.

Alternatively, you can manually create and assign your manager files. Each manager asset is referenced at the top of it's respective tab in the main Adventure Creator. If you remove the reference (by selecting the reference field and pressing backspace), 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 it's "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 it's links to these manager asset files inside a file called **References**. This file MUST be placed in a folder called **Resources** in order to work properly. If such a file is not present, you will receive a prompt to automatically create one when you call up the main Adventure Creator window. Adventure Creator will attempt to create the file inside Assets → AdventureCreator → Resources, so ensure that this directory exists.

1.4 - PREPARING A 3D SCENE

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

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

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

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

Now we will want a Navigation Mesh, or NavMesh. NavMeshes are 3D geometry that dictate where the player can navigate to inside the set. 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 2.8 to 2.10 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.

We also need to define our initial camera, or GameCamera. GameCameras are separate from our MainCamera, in that they dictate the available positions that our MainCamera can take. Only the MainCamera is active throughout the scene, but it will copy it's associated GameCamera's position, rotation and field of view. Click **GameCamera** in the Scene Manager, and position the

newly-created camera to a point you're happy with. Next, find your default PlayerStart object - you can do so quickly by clicking on the **Default PlayerStart** field to highlight it. In the PlayerStart inspector, click the **Camera On Start** field and select your new GameCamera. When the scene begins, the player will appear at this PlayerStart, and the MainCamera will appear at this GameCamera.

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** 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 it's 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 it's 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.

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.

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 regulard 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 apprarant 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.

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. Additionally, the Settings Manager will list your game's required Input Axes based on the settings chosen.

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 2.8 to 2.10.

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. Double-clicking will make the player run, if they are able to.

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

This style makes heavy use of pathfinding to move the player around the scene, so it's essential that your Player prefab has the **Paths** component attached. You will need to define a NavMesh for every scene – see sections 2.8 to 2.10 for more information.

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

You can optionally supply a “Click marker” prefab via 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. Unity should have already defined the first two by default upon starting a new project. If not, refer to Unity's documentation on how to set them up. The Run axis should be a keystroke, such as left shift, that you can hold down to make the player run.

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").

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.

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

Touch Screen input works with First Person control in the same way as Direct control. The drag distances required to make the Player character walk and run can be set in the Setting Manager. You can also set touch-dragging to affect just the Player's direction, rather than position, in the **Movement settings** panel.

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.

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

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

The **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.

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 - 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 it's 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**.

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

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.

2.7 - 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, objects are examining by placing a second finger down on the screen while the first finger is still touching. You can simulate this effect in the Unity Editor by right-clicking on a Hotspot while the left mouse button is held down.

2.8 - 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 it's bare minimum, since Adventure Creator's pathfinding algorithm refers to these vertices when calculating a path.

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

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

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.

2.9 - 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.

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.

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.

2.10 - POLYGON COLLIDER PATHFINDING

Polygon Coller-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 it's 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.

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.

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.

NPCs require the NPC script, while the Player character requires the Player script attached to it's 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 it's **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.

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 default subtitles menu is turned on, 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

A game cannot run until a Player is defined. Even if the game is purely first-person, and the player is never seen, a prefab still needs to be created. 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 run the game. So that it can be instantiated, the Player prefab is required to be placed in a folder called **Resources**.

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

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

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 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 2.8 to 2.10 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.

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

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

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

Once you have set up your pre-determined path, you can use the **Character: Move along path** Action to move either the player or an NPC along it. Again, see section 5.2 for more on Actions.

Predetermined paths can also be used to restrict player movement during gameplay. You can use the **Player: Constrain** Action to assign a Paths object to the player, which will mean the player can only move along that path. Note that this feature only works with the Direct and First Person 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 it's 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. Note that it is also possible to animate a character's jaw bone automatically, according to the sound they are making: see section 9.1 for more.

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

3.6 - CHARACTER ANIMATION (UNITY SPRITES)

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

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.

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 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.

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.

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 **Character engine** field in the Settings Manager. When you select it for the first time, a message regarding a preprocessor define will appear. The "tk2DIsPresent" preprocessor must be defined for 2DToolkit integration, and this can be done in two ways:

- Uncommenting the line `#define tk2DIsPresent` at the top of the tk2DIntegration.cs script file, found in AdventureCreator → Scripts → Static, by removing the two preceding slashes.
- 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.

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

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.

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 it's 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 merely affects which type of GameCamera prefab is available within the Scene Manager, to avoid confusion since most games will only require GameCameras of one type. 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 to use them.

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. They have three axes that can be controlled: X-axis position, Y-axis position, Y-axis rotation, and Z-axis position. Each axis can be locked or unlocked separately.

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 Y-axis 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.

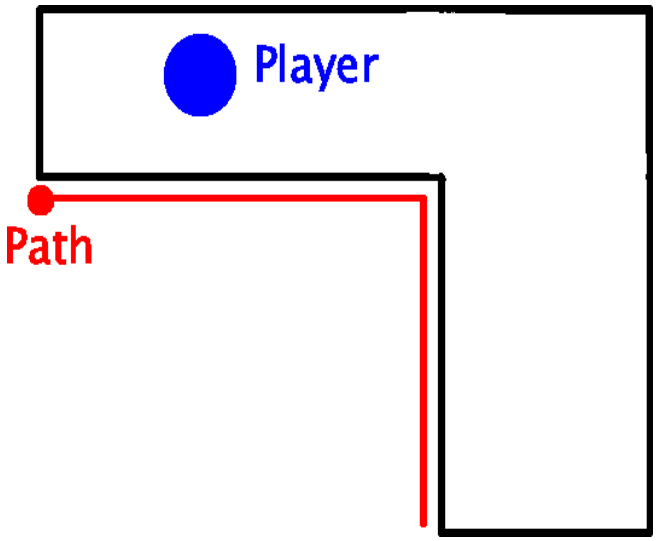
Though their default projection is Perspective, GameCameras can also be set to Othographic. It is important, however, that the scene's Navigation Mesh is always visible to the camera if you are making a Point And Click game – if you are making one with Orthographic camera, be sure to rotate them downward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

4.3 - ANIMATED CAMERAS

The GameCamerAnimated 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 it's 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 it's 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 it's inspector to correct it's 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 it's panning speed – background objects should have positive values, while foreground objects should have negative values.

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 it's inspector. Clicking this alters the prefab's layer, as well as parents the MainCamera, so that you can have a live preview of the scene – with background – within the Game window. The MainCamera will be automatically un-parented when the game begins.

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

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.

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 "Wait" cursor, as well as store a few other cursor-related settings.

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 it's 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, or select one via a list in a Menu (see section 10.1 for more on Menus). This allows for an interface similar to those used in early 90s Sierra games, such as Kings Quest V.

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. Clicking on a Hotspot causes a Menu to appear, on which the available interactions are available 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.

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, the Input axes **CycleHotspotsLeft** and **CycleHotspotsRight** must also be declared.

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** and **Detect Hotspots** components, and position the sphere collider such that it's 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.

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. The standard actions available are:

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

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

Camera: 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 or sprite direction. This is intended mainly for 2D games.

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.

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

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

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: Switch portrait
Changes the "speaking" graphic used by Characters. To display this graphic in a Menu, place a Label 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.

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

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

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

Engine: Change scene setting
Changes any of the following scene parameters: NavMesh, Default PlayerStart, Sorting Map, Cutscene On Load, and Cutscene On Start.

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

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.

Engine: Pause game
Waits a set time before continuing.

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

Engine: Run ActionList
Runs any ActionList (either scene-based like Cutscenes, Triggers and Interactions, or asset-based like Inventory ActionLists). Can optionally run the new list concurrently.

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

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

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

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

Inventory: 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.

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: 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. When this is used by an Inventory or Menu ActionList (see sections 6.1 and 10.3 respectively), the GameObject field becomes a Constant ID number. Attach either the ConstantID script or a Remember script (see section 8.1) to give a GameObject a Constant ID number that can be used to reference objects.

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

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

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

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

Player: 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.

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.

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: 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 10.2), 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 forumla (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 forumlas do not currently work on the Windows Phone 8 platform.

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

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. 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, and manipulate them in bulk by right-clicking on empty space.

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 Interaction for them that run when clicked on. Hotspots need to be on the **Default** layer to be recognised by the control scripts.

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

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

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

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

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

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 **Object: Send message** Action. A Hotspot is declared "on" only if it is on the Default layer.

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

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. Cutscene objects are invisible and cannot be interacted with directly by the player – their position is unimportant.

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

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

Nearly all Actions can call Cutscenes after running. The Scene Manager contains 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 8.3. The **Start delay** determines the time (in seconds) that the Cutscene will wait before running it's Actions. If a **Kill** message is sent to a delayed Cutscene (using the **Object: Send message** Action), the cutscene will not run when the time runs out. This feature can be used to create timed sequences in your game.

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.

5.6 - TRIGGERS

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

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

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

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

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.

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

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

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.

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

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.

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.

5.8 - 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. As with Conversations and Cutscenes, Arrow Prompt objects are invisible and their transforms are unimportant.

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

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

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

5.9 - SOUNDS

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

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

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

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

By default, sounds do not carry over when changing scene, but you may wish to have background music continue playing as you navigate the game. When a Sound prefab's type is set to Music, check the **Play music 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.

5.10 - 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, that 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.

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.

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. If it is the former, called "Context Sensitive" by Adventure Creator (see section 5.1), you can also make use of an **Active** texture for each Item. An Active texture is displayed when an Inventory cursor is hovering over a Hotspot, in place of the Item's regualar texture. The cursor effect that is employed, be it simple or pulsing, can be modified in the Settings Manager.

Inventory items can also be sorted into categories. Categories can be created, renamed and removed from the top of the Inventory Manager. Once one more 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.

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

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

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

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

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.

6.2 - SCENE-INDEPENDENT INVENTORY HANDLING

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

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

Inventory ActionLists (along with Menu ActionLists – see section 10.3) cannot refer to scene objects by dragging them into appropriate fields, because they won't be able to find them when another scene is loaded. Instead, references to scene objects are made by way of Constant ID numbers. A GameObject can be given a Constant ID number by attaching the **ConstantID** script component to it – *Remember* scripts can also be used in this way (see section 8.2). The inspector will display the number, which can be entered into an Inventory ActionLists's Action to make a reference to that object. Character-based Actions work slightly differently – while they don't need a Character's Constant ID number to be entered directly, any Character referenced in an Inventory ActionList must have a Constant ID number associated with it.

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

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.

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.

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.

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.

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 10.2), 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 it's 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 10.2) to a String Variable, and Mecanim parameter values to Boolean and Integeer Variables.

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.

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 it's **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, it's value is "downloaded" to Adventure Creator only when it is needed – ie. when the Variable: Check Action is used to determine it's 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 8.4 on how to do this.

CHAPTER II: ADVANCED FEATURES

8.0 - SAVING AND LOADING

8.1 - 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.

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**.

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, it's data will be saved and loaded. Adventure Creator will also handle the storage and return of room data as the player navigates different scenes.

Saved game files are stored in Unity's **Application.persistentDataPath**. You can display this path in the Unity console by calling `print(Application.persistentDataPath);` in a script.

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 8.5.

The save system will also not save dynamic paths, i.e. those generated by characters when

pathfinding. Thus, if the Player character is pathfinding do a point as the game is saved, they will no longer be moving when that game is loaded back.

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

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

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

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

8.2 - SAVING INDIVIDUAL OBJECTS

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

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

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

RememberCollider – Attaching this script to any Collider or Trigger will make sure the enabled state of it's 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 it's DialogueOptions will be saved. This is a part of the Conversation prefab by default.

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

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

RememberNPC – Attaching this script to any NPC will record that NPC's transform, active path, and visibility to the player's cursor. Note that in order to also save the active path, a *ConstantID* script must also be present on the Path object. If your NPC uses sprite-based animation, the standard animations are also saved.

RememberTransform – Attach this script to any GameObject to save it's transform data. Can also be used to save which GameObject is it's parent, provided that GameObject has either a *ConstantID* script of it's own, or is an assigned Hand Bone of a Character.

RememberVisibility – Attach this script to any GameObject to save it's renderer component's "enabled" state, and optionally that of it's children. This works for both mesh and sprite renderers.

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

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

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

8.3 - AUTOSAVING

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

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

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

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

8.4 - OPTIONS DATA

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

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.

8.5 - HOW THE DEMO DOES IT

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

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

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

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

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

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

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

9.0 - SPEECH MANAGEMENT

9.1 - AUDIO FILES

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

You can access the Speech Manager from the final tab in the main Adventure Creator window (Window → Adventure Creator → 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. It will also do the same to lines contained in InvActionLists referenced in the Inventory Manager – see section 6.2.

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

Audio files can now be made for the game. Clicking **Create script sheet** will let you save an file containing your game's script, which voice actors can be given. This script file also contains the filename that each line is expected to have. The filename syntax is "Character name" + "ID number", while the audio format and extension can be any that Unity recognises. For example, Brain's line, "Hey, little robot!" has an ID number of 21, meaning the associated audio file is named *Brain21.mp3*. Player lines are always named as *Player*, rather than the Player prefab's name. If a speech line has no associated Character, it is considered a narration line, and the audio file is named *Narration*.

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

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 method of moving a character's mouth automatically, based on the volume of their speech audio. This is used by the demo game to animate the robot's jaw without a need for line-specific animation clips.

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

9.2 - MANAGING TRANSLATIONS

Adventure Creator comes with full translation support for speech lines, Hotspots and Dialogue Option labels, Journal pages, and Menu element and Inventory Item names. A game's active language is stored in it's Options Data (see section 8.4) and is controlled by the player in the Options menu.

Within the Speech Manager, you can add and remove supported translations underneath the Languages panel. When you are ready to create a translation, click **Export CSV** to generate a CSV file which contains your game's text. Be sure to click **Gather lines** 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 CSV** within the Speech Manager to update the translation. Note that the CSV delimeter is a pipe (|) character. As an example, the demo game comes with a French 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*.

10.0 - MENUS

10.1 - 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.

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 axis 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, it's 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 it's orientation, position, size, and more. The main property to define is it's **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 it's defined area. Provides an additional option to lock it when access to the Inventory is also locked.

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.

On Container – Activeates 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 it's defined area. 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. Can additionally be filtered to only display for Character or Narration lines (Narration lines being those without an associated Character).

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.

10.2 - 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, it's 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 it's 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, it's 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.

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 it's link to other data, and can be used to affect integer Variables.

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. It's click functionality is handled automatically.

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. It's 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. If it's **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 Inventory** 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**. More pages can be added during gameplay via the **Menu: Change state** Action.

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.

SavesList – Displays a list of saved games found in the file system. The **Click action** is used to determine if clicking loads them or overwrites them. If set to **Save**, an option to save a new game will show if the number of save files does not exceed the maximum set.
Slider - Draws a horizontal bar whose width represents a value between 1 and 10. The **Slider type** field can be used to link it with one of the three sound types in the game, link to an integer Variable, or allow it's value to be defined via a custom script.

Timer - Draws a horizontal bar whose width represents the time remaining for the player to select a Dialogue option, if the active Conversation is set to be timed.

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 it's value to be defined via a custom script.

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**.

10.3 - 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 a Menu ActionList asset.

Menu ActionList assets are similar to regular ActionLists (such as Cutscenes and Interactions), only they can be run independently of a scene. They are identical in function to Inventory ActionLists (see section 6.2), and are created as asset files inside your Assets folder. You can create them by choosing Assets -> Create -> Adventure Creator -> Menu ActionList from the toolbar.

The demo Menu Manager relies on a Menu ActionList 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 is run, with click data sent as parameters.

10.4 - 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, it's appearance is not important. Make sure it's **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 it's associated Menu and Element. If a GUIText component is also attached to the object, it's 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 a Menu ActionList (see section 10.3) to manipulate objects in your scene, for example moving the Camera to focus on another Element. Note that if you use a Menu ActionList to reference a scene-based object, you must do so using that object's ConstantID number (see section 6.2).

CHAPTER III: EXTENDING FUNCTIONALITY

11.0 - INTEGRATING NEW CODE

11.1 - CUSTOM SCRIPTS

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

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

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 it's **TurnOn** and **TurnOff** functions to enable and disable it respectively.

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.

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

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

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

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

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

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

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

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

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

11.3 - CUSTOM PATHFINDING

Adventure Creator provides two methods of pathfinding – Mesh Collider and Unity Navigation (see sections 2.8 to 2.10). Both methods are written in separate scripts, which are both subclasses of the **NavigationEngine** ScriptableObject class.

The correct subclass is instantiated at runtime by the NavigationManager component of the GameEngine prefab, which uses the **AC_NavigationMethod** enum to determine the class's name. For example, AC_NavigationMethod.UnityNavigation enum instantiates the NavigationEngine_UnityNavigation class.

To integrate a new pathfinding method into Adventure Creator, simply add a method name to the NavigationMethod enum (found in AdventureCreator → Scripts → Static → Enums.cs), and create a NavigationEngine subclass with the name syntax NavigationEngine_NewMethodName. You can refer to the NavigationEngine script (found in AdventureCreator → Scripts → Navigation) to see what functions can be overridden.

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.

11.4 - CUSTOM ANIMATION ENGINE

Adventure Creator provides three methods of animation – Legacy 3D, Unity's built-in 2D, and 2D Toolkit. These methods are written in separate scripts, which are subclasses of the **AnimEngine** ScriptableObject class.

The correct subclass is instantiated at runtime for each Character and Object: Animate Action, which use the **AC_AnimationEngine** enum to determine the class's name. For example, AC_AnimationEngine.Lgegacy instantiates the AnimEngine_Legacy class.

To integrate a new animation engine into Adventure Creator, simply add a method name to the AC_AnimationEngine enum (found in AdventureCreator → Scripts → Static → Enums.cs), and create a AnimEngine subclass with the name syntax AnimEngine_NewMethodName. You can refer to the AnimEngine script (found in AdventureCreator → Scripts → Animation) to see what functions can be overridden.

12.0 – HOW IT WORKS

12.1 - 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 12.4).

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 it's data survives scene loading, and **GameEngine** is scene-specific, and does not survive scene loading.

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

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

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

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

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

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

12.2 - MANAGERS

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

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

Changes made in the managers, except those made in the Scene Manager, survive assembly reloads. Because most scripts that refer to them do so only when needed, you can make changes to your game while the game is running (such as changing the 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.

12.3 - PATHS

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

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

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

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

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

12.4 - PLAYER CONTROL

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

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

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

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 intefering 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:

```
GameObject.FindWithTag (Tags.gameEngine).GetComponent
<KickStarter>().TurnOffAC ();

GameObject.FindWithTag (Tags.gameEngine).GetComponent
<KickStarter>().TurnOnAC ();
```