JASON BOWEY

LEARNING TO USE THE UNITY 2D GAME ENGINE

Contents

An Introduction to The Unity Game Engine

Contents

Learning the Interface

The Unity3D Game Engine is a powerful tool, and in these tutorials we will explore many of its powerful features. To have a firm understanding of these features, however, it is important to become intimately familiar with the interface before jumping in and starting with anything interesting.

Creating, Saving, and Opening a Project

Let's start with the most basic operations in any computer programs: saving and loading. Since Unity is a complex system, saving and loading is a little different than other applications you may be used to using, so let's look at it step-by-step. Please note that for the creation of these tutorials I am using OS X V10.11.15 and Unity V5.3.4f1 Personal. There will likely be slight differences between my screenshots and what you actually see, but the main idea will almost always be the same.

Likely the first thing you will want to do with Unity is create a new, blank project. As of Unity 5.0 the default for when you start the Unity3D Editor is to present you with a list of previously opened projects. (See Figure 1)

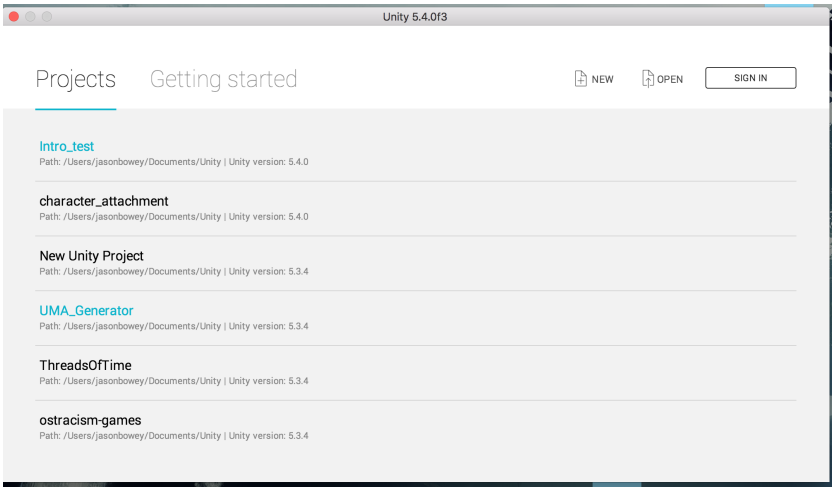

Figure 1: Unity startup window. Lists all recently used projects, with options to create a new project and open one.

This is very useful when you are reopening a project, but along the top there are also options to load a project and to create a new project if your project isn't in the list or you are creating a new project. This time click the 'NEW' button.

This should bring up a new window with a few options (See Figure 2). Under 'Project name' enter the name of this project. Please

please please put a name in this field. The marker will not appreciate everyone handing in a project called 'New Unity Project', it makes it very difficult to download and keep track of. It also gets confusing for you to remember if assignment 3 was in the project 'New Unity Project 4' or 'New Unity Project 5'.

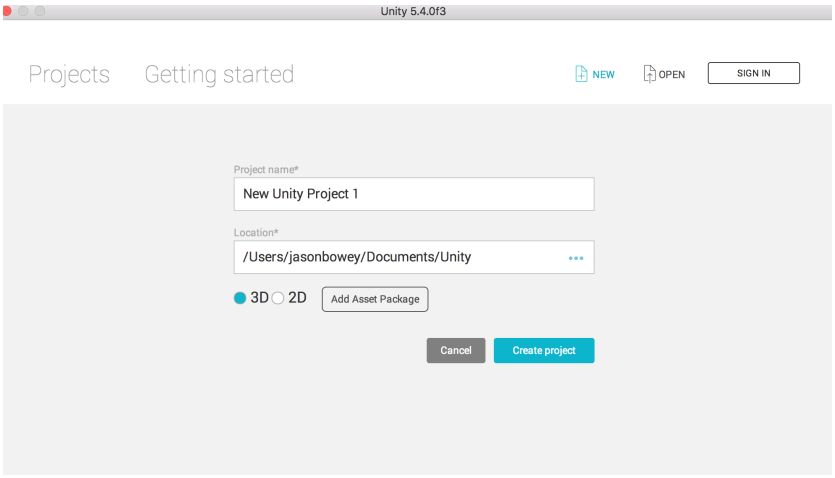

Figure 2: Unity Create New Project Window. .

Under location you can set where the project will be saved. Note that the project will be saved in that location as a directory, so you don't need to create a new blank directory to save the project inside. If you are using the Mac computers in third-floor Spinks, the saving procedure is a little different. Speak with either myself or the TA if you plan on using the Mac computers.

You can set the default 2D/3D mode, but this isn't important since it can easily be changed at anytime when building and running the game.

The 'Asset packages' button allows you to important external libraries other people have made and distributed. By default there are several available libraries created by Unity, but it is possible to import your own. This will be covered in a later tutorial. For now, let's not import any packages and just create a plain, blank project by clicking 'Create Project'.

This should bring up the editor which will look something like Figure 3 (probably not identical).

If you open up the 'File' menu, you will see several options for loading and saving. These should be fairly straightforward, but one distinction that should be made is the difference between a Project and a Scene. The project is the entire collection of pieces that will

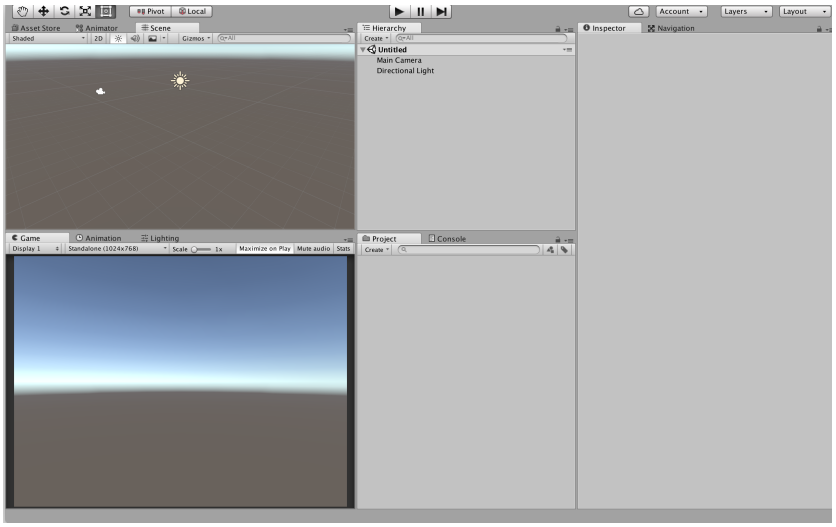

Figure 3: New empty Unity project.

be used to create the game. This includes assets (like images and sounds), GameObjects you have created (called prefabs), temporary build files, editor customizations, and general project settings. A scene is a single individual level in the game you are building. So when you save the project you are saving the state of the entire project. When you are working on a single scene you will also need to frequently save that particular scene. Since the Unity project is just a file system most changes you make (such as importing or changing an image) are automatically saved. Save your project and exit Unity (it will prompt you to save the current scene. You don't need to do that, since you haven't actually done anything yet).

Now if you open Unity you should see the same Project Overview window as before, but now the project you just created should be at the top of the list (likely the only project there at the moment). If you click on it it will open up, but just for fun let's look at what the project structure actually looks like on the file system and how you can open a project that doesn't appear in the recent list.

Click OPEN and a file picker will come up. If you navigate to the place where you saved your project you will see a directory with the name of your project and at least three directories inside (also possibly some other files and directories depending on whether you have started working on the project yet). The main directories are Assets/, Library/, and ProjectSettings/.

- **Assets/** This is where all of the things you import will be saved to and is the root directory for the Unity Editor 'Project Panel' which we will see soon. If you don't want to import things through the Unity Editor you can manipulate, add, and remove files and di-

rectories within Assets/ and the changes will automatically be updated in the Unity Editor (and vice versa). This directory is the meat of your game development, so it will definitely need to be tracked under version control.

- **Library/** This folder will contain all of the temporary build files Unity Creates when you play a game within the editor. None of these are useful to us so just don't touch this folder. If you accidentally delete it no harm done since Unity regenerates these files when you press play anyway. As a result, you should not include this file as part of your files tracked in version control.

- **ProjectSettings/** This directory contains all of the settings you have customized for your project. So, for example, if you set the global gravity setting to be -20 instead of -9.81, that change will be saved in a file called Physics2DSettings.asset. This directory needs to be included in version control, since these settings will be often changed and tweaked throughout development.

All other files are user specific settings and are only important for an individual developer (for example your customized editor layouts, plugins to integrate with monodevelop or visual studio IDEs, etc).

To Open the project you are looking at, make sure the project root is selected in the file picker (this will be the directory with the Project name) and click Open. You should be back at the new blank project you saved earlier before our little save and load detour. Now that you (hopefully) understand the underlying project structure let's look at the pieces of the Editor, which I will call Editor Windows.

Hierarchy Panel

Let's start with the Hierarchy Panel. This window depends on the current scene you have open, so its contents will look different for different scenes. This is important because the Hierarchy Panel is a list of all of the GameObjects (things) currently in this level. More on GameObjects later, but just know that when you add something to the game it will appear in the Hierarchy Panel. Even when you are running the game, spawned objects will appear in the Hierarchy and deleted objects will disappear.

The Create button at the top of the Window allows you to create new GameObjects to add (surprise!) and the search bar filters through the list of GameObjects.

If you drag and drop one GameObject onto another GameObject, the moved GameObject becomes a child of the other GameObject, inheriting its Transform (Position, Rotation, Scale). This is why it's

called the Hierarchy, it is a list of GameObjects with a structure of Parent/Child relationships.

Project Panel

Next is the Project Panel. This is kind of like the hierarchy, only it is the list of Assets for the entire Project, instead of for a particular scene. Specifically, the Project Panel reflects the Assets/ directory we saw earlier. Anything you add to that directory automatically appears in the Project Panel, and any changes you make to the Project Panel changes the Assets/ directory directly.

There are several ways to important an asset to the Project Panel. The first, is adding the asset directly to the project file system. The second way is to RIGHT CLICK in the Project Panel > Import New Asset. The third way is to go through the menu at the top: Assets » Import New Asset.

The easiest way to add something from the Project Panel into the current scene is to drag it into the Hierarchy (or the Scene View), but more on that later.

Scene View

There are two different windows that look at the game world. The first of these is the Scene View. The Scene Views is a window that lets you explore and edit the game world. You can move objects around to place them, scale them, rotate them, or even pan your own view through the world. There are keyboard shortcuts for each of these commands, but there are also mode buttons along the top left of the Unity Editor to toggle between panning, move, rotate, scale, and sprite mode.

Figure 4: Mode buttons for manipulating GameObjects in the Scene View (from left to right): Pan, Translate, Rotate, Scale, Sprite.

Game View

Unlike the Scene View, which allows you to explore the game world, the Game View is a window that previews what the game would look like right now, if you were to build the game. The Game View

is controlled by the Play/Pause/Skip buttons along the top of the Editor.

It is important to warn that most changes you make to the hierarchy while in PLAY MODE will NOT be saved once you leave play mode. So if you play the game and move things around, when you unclick PLAY things will revert to their original positions. This can be incredibly useful but will also result in lost work if you aren't careful.

Figure 5: Player controls for when previewing in the Game Window. PLAY, PAUSE, SKIP FRAME

Perhaps the most useful Editor Window is the inspector panel. This panel will display all of the information of the thing you currently have highlighted. So, when you click on a GameObject you will be shown all of the values and information about that GameObject. If you need to make a change of some kind you will likely need to go to the Inspector Panel to perform the change.

Editor Windows

There are many other editor windows available (Found under the Window menu option along the top menu bar) and you can find or purchase others online/in the Asset Store. You can even design and create your own (See tutorial here for details)

Also, the Unity interface is completely customizable. If you don't like the layout and want 'Window X' where it is, just click on the window's tab and drag it somewhere else. You can stack several Windows in the same section and tab between them, or even split an existing section, either horizontally or vertically. The possibilities are endless so feel free to customize to your preference and personal workflow. For the purposes of this tutorial I will keep the windows as the default, and in a few cases I will explicitly move certain windows around if needed.

GameObjects

Now that the boring interface stuff is out of the way, let's get into the... slightly less boring stuff: how to construct a GameObject.

Tags, Names, and Layers

For now, let's just create a very simple, blank GameObject. In the Hierarchy right click and select 'Create Empty'. Now you should have a second Object in the Hierarchy named 'GameObject'. We will use this as an example for the remainder of this section. Make sure the new GameObject is selected and look at the Inspector.

At the very top of the Inspector there are several toggles, text fields, and dropdown menus. The leftmost toggle enables/disables the GameObject, effectively turning it on or off in the Game. The text field is the name of the GameObject, currently called 'GameObject'. Let's change the name to 'Test' and press ENTER. Now the name in the Hierarchy should be changed.

There are two dropdown menus left: Tag and Layer. These are two options that every GameObject has, which allows you as a developer to assign that GameObject to two different categories. These can be anything you want, but generally Tags are used to label a GameObject with a specific property or assign it to a category, and Layers are used in rendering, raycasting, and linecasting. For example, I could assign a GameObject with a 'Player' tag, which would indicate that this GameObject is the current Playable character. I could then create a 'Character' layer, defining a broad group of GameObjects which share a similar rendering layer (all characters are drawn on top of the background, as an example). How you use tags and layers is up to you, but they are both extremely useful and powerful.

Components

Contrary to what you may have studied in other courses and languages which likely use some form of Model View Controller architecture (MVC) Unity works best when using a Component-Based Architecture. Broadly speaking, rather than separating objects into Data, View, and Controller classes, GameObject's behaviour is defined through several individual pieces called Components. In many cases this architecture seems similar to MVC but it can be fundamentally different.

A component is an individual piece of code that does a single thing (if designed well). Unity has several pre-defined components that handle rendering, physics, even basic character controls. As developers we can also define custom components, called scripts,

which allow us to create unique components that do a specific thing.

For example, as we will see in the next section, every GameObject has a component called Transform. This component determines where the GameObject is in the game world. To move the game Object, we will need to change the Transform component. Components are a very complex idea and it is difficult to get your head wrapped around the architecture. We will go through a more detailed example in a later tutorial, but for now just remember that a component makes a GameObject behave or appear a certain way, and the combination of components attached to a GameObject determines how it looks and acts in your game.

The Transform Component

There is one more portion of our 'Test' GameObject that has yet to be explained: A collapsable section with the title 'Transform'. This is a component, and it is the one component that every single GameObject must have, as it determines the GameObject's position, rotation, and scale in the game world. Changing these values are the same as using the movement tools in the Scene View.

Adding Components

At the very bottom of the Inspector is a button that says 'Add Component'. If you click it, you can browse through the components pre-defined in the Unity Engine as well as any custom scripts you have written in this project. If you know the name of the component (which matches the class name of the script) you can use the search bar to find it faster.

Creating Custom Components

We will get into writing scripts in another tutorial but for now just remember that creating a script will allow us to have a custom component to attach to GameObjects.

Sprites

In the context of games, a sprite is simply a 2D image to be used as GameObjects such as characters, or interface components such as buttons. Unity has a sprite rendering system that displays images and supports sprite animation: flipping between different 2D images to give the player the perception of animation, similar to how it is done in cartoons.

Let's start exploring the Unity Sprite system. In this example we will go over how to import sprites, add them to the game, look closely at how to construct a Sprite GameObject, and a simple way of creating sprite animations.

Before we do anything let's save the current (and blank) scene (File » Save Scene As...). Organizing your project is very important, but for now just save it somewhere in your Assets/ folder. Name the Scene 'SpriteTest'. Remember to periodically save your scene, as well as your project, so you don't lose hours of work if (and likely when) Unity decides to crash.

The Sprite GameObject

If you haven't done so already, download the image from the course Moodle site, called 'knight.png'. Import the image into Unity. The image should now be in the Project Panel (if not, make sure you imported it using one of the methods described in previous sections).

Now that you have the image imported as part of the project we can import it to the game. The first thing to check is to make sure that it is in the correct format to be used as a sprite. Click on the image in the Project Panel so its information appears in the Inspector. The only thing we will change now is the Texture Type. The Texture Type should be set to 'Sprite (2D and UI)'. Once this is changed click the Apply button.

Now that the image is in the correct sprite format, drag the sprite from the Project Panel into the Hierarchy to import it into the current scene. Click on the newly added entry to the Hierarchy and examine its construction in the Inspector. What do you notice about the fields in the Inspector?

The Sprite GameObject should look almost the same as our previous blank GameObject, except it should have a second component that was automatically added: something called a SpriteRenderer. By adding this component to a GameObject, it tells the GameObject to display a 2D sprite image. You should see that the Sprite Renderer has a field called Sprite with a box next to it that matches the name of the image you added to the scene. If you click on the box it should highlight the sprite image you had imported and added to the scene. By dragging the sprite into the Hierarchy Unity automatically constructed a Sprite GameObject to you by adding the renderer and assigning the Sprite field, but there is nothing stopping you from changing how the object appears by changing the Sprite in that field (you can drag any sprite image onto the Sprite Renderer's Sprite field to change the Sprite associated with that GameObject). This is an important observation to make about how Unity works: dragging assets

into the Hierarchy will create a simple GameObject, but all Unity is doing is adding the components it thinks you will need. It will always be possible (and in some cases easier or cleaner) to construct a GameObject from scratch yourself to have complete control over how it is constructed to match your design. We will see an example where this is the case when we look at animation construction.

Sprite Sheets

A sprite sheet is a single image file that contains multiple images, often arranged in rows and columns. When using the sprite sheet only one image is displayed at a time, the remainder of the sheet is hidden by the system. This allows the system to use the same image file for multiple images, saving on memory.

Unity has an effective system for handling sprite sheets, easily allowing developers to import sprite sheets and cutting them into individual pieces. Import a second image into your project, the spritesheet called elf.png. If you look at the image, you will see it is several rows of elf images and each row contains the separate poses of an animation. When we animate the elf sprite the GameObject will cycle through each of these individual images very quickly to give the perception of animation to the player. In the Inspector for the spritesheet make sure Texture Type is set to 'Sprite (2D and UI)' and this type set the Sprite Mode from 'Single' to 'Multiple'. This is telling Unity that this image is a sprite sheet and needs to be cut up into more than one piece.

To do the actual cutting, click the Apply button to save the settings and click the 'Sprite Editor' button in the Inspector. This will open the Sprite Editor window (see Figure 6).

Click on the 'Slice' button in the top left corner to open the Slice menu. Set the Type to 'Grid By Cell Count' which will bring up several new fields. Set the 'Columns and Rows' to 4 and 4 (specifying the sheet has 4 rows and 4 columns). All of the other options are fine for now, so press slice to slice up the sheet, press apply and close the sprite editor.

Now, you should see an arrow by your knight spritesheet in the Project Panel. Click this to expand and show all of the individual images you just cut up in the sprite editor. Now when you use any of these individual images it will actually be using the sprite sheet under the hood, saving memory and complexity without you noticing!

The sprite sheet isn't perfect, so there may be some cleaning up to do later on, but all of the individual images should now be available and for our purposes that is the important part.

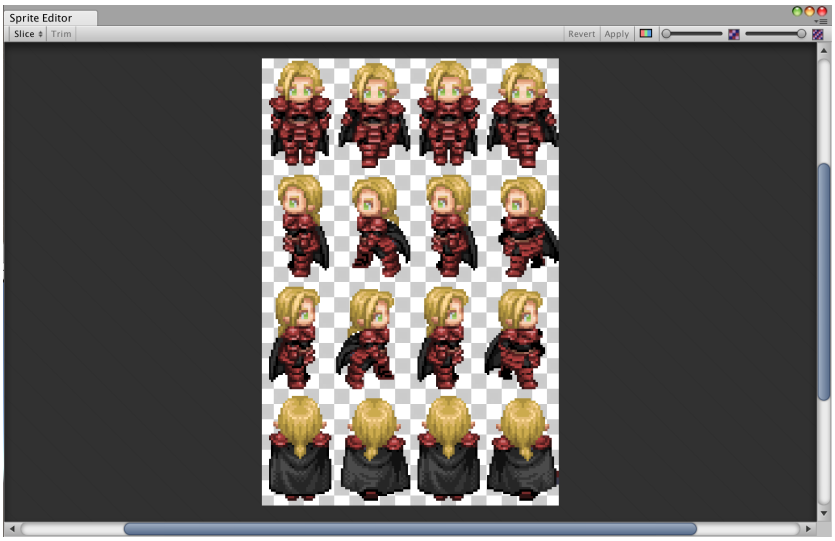

Figure 6: Sprite Editor window.

Look through the sprites and see that as you go through each row the elf appears to animate as you go from one to the next. Pick a series of sprites that form one of the animations and SHIFT SELECT all of them at once by holding the SHIFT KEY as you click. Once you have all of the sprites highlighted drag them all into the Hierarchy at once and you will be prompted with a save dialogue asking where to save your new animation. Again, it doesn't matter where you save it at the moment, so save it anywhere in your Assets/ folder and name it whatever you want. If you look in the Hierarchy there should be a new GameObject added. Select the GameObject and rename it 'elf'. How does the construction of this elf GameObject different from the knight?

The difference is a new component that was added, called Animator. In a later tutorial we will look at the details of Animators but for now suffice it to say the Animator is a component that determines which animation to play and when to change to a different one.

In the Project Panel there are two new assets that were automatically created, one that matches the name and number of the first sprite you dragged into the Hierarchy (this is called an Animator Controller and was automatically linked to the Controller field of the Animator component) and the new animation you saved. More on these later, but just like when we added the single sprite we will be able to create each of these things ourselves to have more control over how they are organized and behave.

Now press the 'PLAY' button along the top to play what we have so far in the Game View. Now you should see one of your elves cycling through the images you combined to create our first, rough

animation, congratulations!

Physics 2D

Now that you have a sprite and a basic animation, let's look at the 2D physics system. First of all, I would like to make a simple distinction between 2D and 3D mode in Unity. Really, the only difference between the two modes is that 2D mode completely ignored the z-axis (depth). There is a toggle along the top of the Scene View that switches between the two modes, but that is literally all it does. In both modes your game world exists in a 3D space, but 2D mode compresses everything together into the same depth. As a result, you as a developer only have two dimensions to work with, which makes programming the game much much simpler.

In terms of the Physics systems, Unity has a 2D and a 3D version. They are similar, with most of the major components having analogous components in both versions, but the two systems are not compatible with one another. For the purposes of this course we will only be dealing with the 2D versions for simplicity. If, for example, you accidentally added a 3D Collider onto an object it will not interact with a 2D Collider at all, since these are completely separate systems. Let's look at the three main components in the Physics2D system: Rigidbody2D, Collider2D, and PhysicsMaterial2D.

Rigidbody2D

Let's start with the Rigidbody2D component. To describe the Rigidbody2D in one line would be to say: adding a Ridibody2D component to a GameObject forces that GameObject to obey the laws of 2D physics. Adding a Ridibody2D component causes a GameObject to obey gravity as well as Newton's Laws of Motion (Forces, acceleration, air resistance, etc) and similar physics of the natural world. Add a Rigidbody2D to one of your GameObjects and see what happens if you press play.

There are many values to change in the Rigidbody2D that affects how the GameObject's physics behave. Take a few minutes to play around with the values and try to figure out what they do. See here here for a complete description of the Rigidbody2D component.

Collider2D

Rigidbody2Ds control a GameObject's adherence to the laws of physics, but Collider2Ds are required to give a rigidity to GameObjects. To actually register a collision between two GameObjects both

GameObjects must have a Collider2D attached. There are four different types of Collider2Ds: BoxCollider2D, CircleCollider2D, EdgeCollider2D, and PolygonCollider2D. Each of these are functionally the same, but they define a different shaped region of collision.

PhysicsMaterial2D

A PhysicsMaterial2D is an asset that can be added to the 'material' field of Collider2Ds. PhysicsMaterial2D has two values that manipulate the collider's surface material: Friction and Bounciness. Each of these values range from 0 to 1.

To create a PhysicsMaterial2D right click in the ProjectPanel » Create » Physics Material 2D. To add the created asset to a GameObject, simply click and drag the asset from the Project Panel onto the Material field of a GameObject's Collider2D.

Project Settings

Unity allows you to set a wide range of global project settings. These can be found in the top menu at Edit » Project Settings. The most commonly used Project Settings are: Input, Tags and Layers, and Physics2D. There will be several instances in these tutorials where Project Settings will be manipulated, but I encourage you to look through the different options to see what Unity allows you to change.

Prefabs

In a 'real' game, especially one with multiple levels it is likely the same GameObjects will be used again and again. For example, the player character may behave the exact same in every level and the same enemy types may be spawned again and again. In these cases, it would be useful to have a construct that allows developers to build a GameObject once and simply copy it everytime it is used. Fortunately, Unity provides such a construct: prefabs.

A prefab is to a GameObject the same way a class is related to an Object in classical Object Oriented programming. A prefab is simply a blueprint of instructions that can be used to create a GameObject, or many GameObjects, within a game. The steps of creating a prefab are very simple: design a GameObject inside the Hierarchy and then drag it back to the Project Panel which creates a copy of the GameObject, a prefab. Once a prefab is created you can then delete the GameObject in the Hierarchy. Try it now with one of your GameObjects.

You can now drag your prefab into the Hierarchy as many times as you want, adding a new copy to the game everytime.

You can edit a prefab in both the Hierarchy and directly in the Project Panel. If you edit it in the Hierarchy, however, you need to press the Apply button at the top of the inspector for the changes to be applied to all other prefabs. Changing the prefab directly in the Project Panel does not require changes to be saved, they are automatically applied.

Asset Packages

The final introductory concept in this tutorial is Asset Packages. Asset Packages allow you to export a subset of the assets in your Project Panel as a single entity that can then be imported into other projects. This is useful for sharing common assets or creating custom libraries to be used in multiple projects. Some of the tutorials in this course will provide Asset Packages that contain the necessary assets to be used in a particular tutorial.

Both export and import options can be found in Assets » Import Package (or Export Package). Most assets purchased on the Unity Asset Store will come as a package to be imported.

Using Version Control with Unity

Contents

A Very Brief overview of GIT and Why You Should Use it

By far the most annoying, and (with the exception of good group communication) most important part of a large software project is Version Control. Every aspect of it can be annoying but it is the one key piece of your project that MUST be in place early on and MUST be used by every group member. Besides the fact that all of the project milestones will be marked through your Repository (the TA will have access to your code base) it is the one thing between you and accidentally deleting your entire project at four in the morning on the night before the project is due.

In theory, version control is simple: it is a system that allows you as the developer to make changes to your code while also saving older versions of your code at various stages over time. So, if you mess up or delete everything it is possible to revert back to a recent (ish) version where stuff still worked.

For this course it's recommended you use a style of version control called GIT, simply because it is widely used and both the instructor and TA are familiar with it. Other versions such as SVN are perfectly fine to use, but if you do don't count on having anyone who is able to help if the version control blows up (and chances are, at some point, it will). All references and suggestions in this tutorial are git-specific, but many of the general guidelines have analogues to other types, like SVN.

There is an excellent reference to git here including everything from what git is and how it works (Chapter 1), to actually using it in practice (Chapter 2), to more advanced features (everything else). For the purposes of this course, I would recommend reading Chapter 2 for sure (likely more than once) and Chapter 1 is pretty interesting (to me at least) but not required to be able to use git. Since it goes through the different steps and commands, I am not going to waste time by repeating them here. Instead, this tutorial will focus on the Unity-specific part of version control, and tips and suggestions for managing your project with a minimal number of repo explosions.

Setting up a Repository

There are several free cloud-based repositories that support git (some support others like SVN as well). The two main and most popular are bitbucket and github. Both support git and both have (for students) free private repositories (only members in the project can see or access the code and clone the repository). The steps on both of these are fairly straightforward: everyone creates an account, one group member creates a new project and adds everyone to the group (plus the course TA). If you are unfamiliar with either of these, I recommend using the deparement provided service, found at `git.cs.usask.ca`.

Creating a .gitignore

The most important different between Unity projects and other projects (like a java application) is the sheer number of temporary files created by the program. Everytime your press the play button (or build the game) unity builds and compiles many temperary files to be able to run the game and these get replaced again and again. As such, you do not want to have these in your version control (imagine everyone having hundreds of completely different binary files every couple minutes and having git try to merge and resolve conflicts between them everytime you try to push your changes to the repository). Not only that, but it is unnecessary to have these files tracked since they are not required to build the project and it just wastes space in your repository. So, git allows us to explicitly tell it what to track and what to ignore in a .gitignore file.

A .gitignore file is just a text file named .gitignore. Inside, you list all of the files and directories you do not want included in the version control. If you create a project in github (maybe in bitbucket as well) you have an option to use a pre-written .gitignore file designed specifically for Unity projects. There is a sample starter .gitignore file

on Moodle, included with the First tutorial assets. You may need to add things depending on what your project looks like, but as a general rule you should ignore everything except the Assets/ directory and the ProjectSettings/ directory, as mentioned briefly in the first chapter. The idea is you need to include all of the images, scripts, sounds, settings, etc (all of the things required to build and run your game) but don't include any temporary or auto generated files which would add confusion and waste space.

GIT Workflow

As you would have seen in the GIT book linked above, the general workflow is as follows:

1. Make changes to stuff
2. When the project is free of errors and a major change has happened, add your files and commit to your local copy of the repository
3. Periodically use pull to take the central repository and merge it with your local repository (and hope nothing breaks)
4. Periodically push your local repository to the central repository for the rest of the group to then pull (and hope that YOUR changes don't break anything now)
5. make more changes
6. repeat

   In principle, it's simple. In practice, not so much.

Tips and Tricks for working with Unity and GIT

I'll end this tutorial with a few tips and tricks for group management and avoid issues with GIT.

1. Communicate with your group ALL THE TIME

   Use email if you must, but group chat applications such as Facebook, Google Hangouts, Skype, etc are much better because they allow everyone to quickly and easily update and broadcast to everyone. Email is too clunky and cumbersome to send to everyone with messages like: 'I am working on the player character's animations' ten times a day. Also, email easily gets lost and is too easy to ignore.

Personally, I recommend using Slack which allows you to create a group and individual channels to separate messages. It also allows you to message individual users directly and privately. So, for example, you could create channels for AI, Animations, Art, User Input, etc and have everyone subscribe to each of these. Then if you are going to be working on the GoblinAI script just send a message in the AI channel so everyone sees that they shouldn't touch the GoblinAI script. Then once you have finished and pushed back to the repository you can inform your group that someone else can work on the script if they want. As long as you check the channel and make sure nobody else touches the code while you are working on it, you will minimize most of the issues that may arise.

2. Only allow one person to work on something at a time

   It was mentioned in the last point, but it really can't be mentioned too many times. Git does do its best to auto merge files that are different when pushing or pulling, but since Unity uses so many complex binary files (like the Project Settings) merging rarely works and usually requires manual merging. If only one person works on a particular part of the game at once it should be perfectly safe to have multiple people working on the project without too many issues.

3. Enable meta-data

   It is recommended to enable meta-data files when using version control with Unity. This basically creates a text file of metadata for every asset file you have in your project. It is supposed to help make the whole process of pushing, pulling, and merging easier. It clutters your project a little but it's useful.

   ```
   The setting can be found at Edit >> Project Settings >> Editor >> Version Control-Mode.
   ```

4. That script you are editing isn't the only thing you are changing

   This last tip is to point out that while creating scenes are prefabs are very handy, it is a double edged sword. Sometimes if you are working on a script or editing an asset, the changes you make will alter the structure of an entire prefab or scene. As such, be careful which files you add to be committed and be aware of which files you are actually making changes to. For example, if you and someone else are both working on different scripts, but they are both part of the same prefab, you may both be making changes to the prefab without intending to which will lead to merge conflicts.

Scripting

Contents

As mentioned in previous tutorials, Scripting in Unity is the way you as a developer can create custom components for GameObjects. A few common examples of scripts you may create are: PlayerController, EnemyAI, and EnemySpawner. Note that each of these scripts would do a single thing, namely control the player, execute the enemy AI, and spawn enemies. This idea of a single script doing a single thing is best practice for Unity Project Architecture and will be examined in detail in a later tutorial.

For simplicity of learning, this tutorial will violate some of the architecture best practices discussed later in the course. The important part of this tutorial is to look at the different pieces of writing a script, and it is easier to follow along at the beginning with everything in a single script. In later tutorials we will look at breaking a complex script like this into smaller, more specific scripts.

Creating a new Script

Let's get into writing some code! Right click in the Project Panel and select Create » C# Script. This will create a new script in the Project Panel and highlight the filename. Name the Script and press

ENTER to save the script. Double click on the script to open it for editing. By default the script will open in the IDE MonoDevelop, but it is possible to configure Unity Scripts to work with Visual Studio instead. For these tutorials I will be using MonoDevelop.

Once the script opens you should have a basic skeleton script. This will be the default script everytime you create a new script. One important thing to check right away is to make sure the class name of your script matches the filename in Unity. For example, if your script's class is Test then the filename should be Test.cs (in Unity the .cs will be omitted from the name). If these don't match, change one or both so they do match, otherwise you will get an error about the class name not being found.

Go back to Unity and drag the script onto a GameObject in your Hierarchy, it doesn't really matter which one you choose.

Using Public/Editor Variables

In most languages declaring a variable as Public/Private/Protected determines the visibility of that variable by other scripts. This is true in Unity, but it has a second use as well. Let's find out what that use is. Class variables are declared inside the class definition and outside of any defined methods, often before any methods is preferred for the best code readability. Create two class variables like so:

```
using UnityEngine;
using System.Collections;

public class Test : MonoBehaviour {

   public int publicNum;
   private int privateNum;

   // Use this for initialization
   void Start () {

   }

   // Update is called once per frame
   void Update () {

   }
}
```

Save the script and go back to Unity. Wait a few seconds for the file to update. Check the console window for errors (if you can't find

the console window you can open it at Window » Console in the top menu bar). If you have any errors make sure the code is identical to the above code, and make sure the script was saved. With no errors, click on the GameObject your script is attached to. You should see your script attached as a component to the GameObject. What do you notice about the component fields?

You should see that the component has a field called 'Public Num', which matches the public variable we just created in the script. This is the second use of declaring a variable as public: you can see and edit it directly from the Unity Editor, without having to go back to the script to change it. It even works when testing the game!

It is important to note the order that variables are written when a script starts. If you give the variable a value when you declare it in the script like so:

```
public int publicNum = 5;
```

The value of publicNum is set to 5 first. If you change the number in the component's field in the Unity Editor, the number you enter will overwrite the value 5. Finally, any game logic that changes the variable will overwrite the value in the Editor.

Awake(), Start(), Update(),and FixedUpdate()

In scripts, Unity has many built-in methods and variables that have special meanings. The variables are often shortcuts to commonly used components, such as the Transform component, and most of the methods will be automatically called when appropriate, if the particular method is added to a script. All of these seemingly magical methods and variables are available because, by default, scripts inherit from the superclass 'MonoBehaviour'. For more on inheritance see the Architecture tutorial.

Looking at our test script, notice that there are two special methods automatically included: void Start() and void Update(). We will get into what they do in a minute, but first add two more methods (the order you have them in the script does not matter in the slightest): void Awake() and void FixedUpdate(). Finally, let's add one Log statement in each of the four methods. your final script should look something like this:

```
using UnityEngine;
using System.Collections;

public class Test : MonoBehaviour {
```

```
    public int publicNum;
    private int privateNum;

    void Awake(){
        Debug.Log("Awake");
    }

    void Start () {
        Debug.Log("Start");
    }

    void Update () {
        Debug.Log("Update");
    }

    void FixedUpdate(){
        Debug.Log("Fixed Update");
    }
}
```

Save the script and go back to the Unity Editor. Check for errors (and try to decipher what the problem is if any errors do occur. I find Google is a good friend for this) and if your console is clear press play. As the game is playing, what do you notice about the logs coming into the console?

Look at the first two log messages: "Awake", and "Start". Try running/stopping the game a few more times and see if the order of these two logs ever change. No matter how many times you run it, the order of these two won't change. This is because of the order of execution for these two methods: when a GameObject is created (for example, when the game starts) it will execute its Awake method and then its Start method, always in that order. In fact, if you are creating multiple GameObjects at the same time (like when the game starts) Unity will execute each of the Awake methods for all of these GameObjects and then all of the Start methods. So, Start and Awake are initialization methods. In particular, Awake is used to guarantee a value is initialized before another GameObject's Start method uses it. Generally, both are not needed except for specific cases where order is crucial.

After the initialization methods, the game loops execute. The two main game loops you have control over are Update() and FixedUpdate(). What did you notice about these two methods in the console logs? You probably didn't notice much, except they didn't alternate, sometimes you may have seen multiple Update calls before

a FixedUpdate call, or vice versa. That is because the frequency of these two methods differ.

Let's use string concatenation to add a timer value after each of the log calls:

```
void Update(){
   Debug.Log("Update " + Time.time);
}

void FixedUpdate(){
   Debug.Log("Fixed Update " + Time.time);
}
```

Now what do you see when you run the game and look at the game logs? What do the timing frequencies look like?

It looks like the FixedUpdate() is running very periodically (t=0.02, t=0.04, t=0.06, etc) and Update seems to be all over the place (for me I have 0.02, 0.119, 0.184, etc). As the name suggests, FixedUpdate() is called at regular, fixed intervals, and Update is called whenever it is ready. FixedUpdate() is closely linked to the physics system, and is called in sync with the Physics calculations. Update is closely linked to the rendering system, and is called in sync with when the screen is being redrawn.

Go to Edit » Project Settings » Time. This should open up the Time settings in the Inspector. Look at the three values: "Fixed Timestep", "Maximum Allowed Timestep", and "TimeScale". Simply put, The Fixed Timestep determines how often the FixedUpdate() call (and the physics calculations) are performed. Maximum Allowed Timestep is the longest a frame can take before moving on to the next frame (if you are performing a complex calculation in Update that takes longer than this time your calculation will be stopped where it is and Update will be called again). If all of the Update() calls finish early then the next frame begins early. This makes the time between Update() calls very inconsistent. The final value in the Time Settings is TimeScale, which makes time flow faster or slower than normal.

As a general rule: use Awake() to set values that need to be set before other objects need them, use Start() for basic initialization, use Update() for periodic calculations that do not require fixed intervals, and use FixedUpdate() for periodic calculations that need to be regular (for example, object motion).

Finding a GameObject Through Code, Sending Messages

Likely, your game will have more than one GameObject, and there will be times where one GameObject will need to communicate with

another GameObject. For good Software Architecture, it should be your goal to minimize the amount of communication and reliance between GameObjects but it is impossible to completely get rid of it.

There are two approaches to communicating between GameObjects:

1. Get a reference to the gameObject(s), access the script, then access a method or variable
2. Get a reference to the gameObject(s), send a message to all of the scripts on that GameObject

It should be clear that the second method seems to be much better from the point of view of Good Software architecture but the method can be limited and isn't applicable in every situation. For this reason, it is a good idea to know both ways and pick the best one when needed.

There are several static methods (more on those a bit later) in the GameObject class that are useful for finding GameObject(s) currently part of the game:

- GameObject.Find(string name) - returns a reference to the first GameObject in the Hierarchy with a matching name.
- GameObject.FindGameObjectWithTag(string tag) - returns a reference to the first GameObject in the Hierarchy with a matching tag.
- GameObject.FindGameObjectsWithTag(string tag) - returns a reference to ALL of the GameObjects that match the provided tag (returns an array of GameObject, GameObject[]).

There are other methods that are similar (such as GameObject.FindObjectOfType...) but let's just stick with these for now, since they are the most useful.

I'll go through a couple common situations where these methods are useful, but we will use these all the time throughout the tutorials, assignments, and in your own projects.

For these examples, let's say you have a level with a tiled background, several enemy GameObjects, and a player GameObject. Each of the tiles in the background is a Sprite GameObject, tagged as 'Background', each of the enemies are tagged as 'Enemy', and the player is tagged as 'Player'.

Say you have an AI script attached to the enemies that needs to find a reference to the player GameObject and if the player is within a certain radius of the enemy and within line of sight the enemy walks towards the player. In this case, it would be easiest to find the player using the 'FindGameObjectWithTag' method:

```
GameObject player = GameObject.FindGameObjectWithTag("Player");
```

Say your player has a special ability that instantly kills all existing enemies with a bolt of lightning. In this case it would be tedious and difficult to grab a reference to each enemy individually, so you could use the 'FindGameObjectsWithTag' method to assign an entire array of GameObjects all at once, which you can then iterate through:

```
GameObject[] player = GameObject.FindGameObjectsWithTag("Enemy");
```

Notice it is only one line, just like the other version of the method, but there is a lot more going on behind the scenes so be aware that if you have a large number of enemies (on the order of hundreds or thousands) it will take significantly longer to execute than when grabbing a single GameObject.

Finally, imagine a situation where you have a unique enemy (maybe a boss or miniboss). Since it is an enemy it is tagged as 'Enemy', just like the lesser, player-fodder, goblins and skeletons, you can't get this unique enemy by through its tag (sure you could just give it a unique tag, but then you couldn't group it with the other enemies without adding a second condition to check this other tag everytime you need all of the enemies). The solution to this problem is to change the NAME of the boss as "Boss", (or something more creative, likely) and access the object:

```
GameObject boss = GameObject.Find("Boss");
```

Be careful using the Find function, since it will only give you one GameObject, even if you have several GameObjects that share the name. If there is more than one it will just return the first one it finds.

Invoke and InvokeRepeating

Games, and Unity specifically, are based on the concept of periodic loops. As we saw, there are at least two implicit loops being called throughout the duration of the game (Update and FixedUpdate; there are actually others but they don't really matter yet), but what happens if you want a FixedUpdate loop at a different frequency (for example, every 3 seconds)? You could add a check in FixedUpdate to perform a modulus on the time or frame count to skip frames to approximate the frame count, but that's unnecessarily complicated and messy. Luckily for us, Unity has two built-in methods that handle function scheduling (both single calls, and periodic calls). These methods are called Invoke() and InvokeRepeating():

- Invoke(string MethodName, float time) - schedules method with matching name to execute in time seconds
- InvokeRepeating(string MethodName, float time, float repeatRate) - schedules method with matching name to execute in time seconds, and then repeat every repeatRate seconds until either cancelled or the current level ends.
- CancelInvoke() - cancels all Invoke calls in the script

So, say you want to schedule something to happen 3 seconds in the future (maybe a game event, such as open a door). You would need to define a method with the following function signature:

```
void METHODNAME(){

    }
```

and place this line of code where you want to schedule the method:

```
Invoke("METHODNAME", 3.0f);
```

Try this now in your your testing script. Try changing where the Invoke call is placed. What happens if you place it in Start vs. in Update? Why is this different? (hint: think about the lines of code from the perspective of the game loop executing)

Similarly, if you wanted to schedule something like an enemy spawner to start in 3 seconds, but then spawn an enemy every 5 seconds after, you could use InvokeRepeating(), like so:

```
InvokeRepeating("METHODNAME", 3.0f, 5.0f);
```

Finally, say you needed to cancel all of your enemy spawners, you could use 'CancelInvoke()' to cancel all Invoke calls you called from *THIS SCRIPT*.

Getting User Input

A game without user input is nothing but a movie, so an important part of writing game scripts is to get user input and use it do interesting things in our game, like move the player around, attack enemies, etc. For our purposes, user input will include any events triggered by a mouse or keyboard (this can easily be extended to things like an XBox controller).

To get input, there are several static methods built into the predefined Input class (just like we saw above with the GameObject class),

and these will be called in the same fashion (Input.SOMETHING). Before we see specific input calls, let's look at the important difference between a 'Key' and a 'Button', as Unity defines them.

- A **Key** refers to the raw input (for example the 'H' key, or the spacebar)
- A **Button** refers to a predefined mapping between a key (or several keys) and a string name.

The difference between these two concepts is that of good software engineering and reusable code. A key refers to a particular key, whereas a button is simply a name that is associated with a particular key in the project settings somewhere.

Let's say you designed a game to be used with the WASD keys for movement, and all of your input scripts are looking for the WASD keys specifically by their keyCode. Then let's say you decide to switch to the Arrow Keys (or even worse, you want to port your game to PlayStation and need to remap everything to controller analogues). Now you would have to go through all of your code looking for every instance where you got key input and replace it with your new mapping code. Even worse, imagine you need to have both versions, depending on the platform the game is running on. Luckily, defining Buttons will avoid all of this work. You call a button by a particular name in your code, and then in the project settings you can switch what those buttons mean in a single place, neat!

If you go to 'edit » ProjectSettings » Input' the InputManager should appear in the Inspector. You will see an expandable list called Axes. Click on the arrow to expand the list and you will see all of the predefined Buttons currently in the project. If you open one, let's say 'Jump', you will see several properties that define what 'Jump' means when you call it in a script. There are lots of properties here, but notice the name ("Jump") and the Positive Button ("space"). This is what defines the mapping. The button name "Jump" in the game actually means spacebar. If you later decided to use the mouse button to jump, all you would have to do is change that in the Positive Button field and the change would apply to everything in your game.

So now let's see how to actually call these things in code. There are two main types of buttons that are predefined:

- **Button** - a single button which is either pressed (TRUE/FALSE)
- **Axis** - a pair of buttons that represents a positive and negative direction [-1,1]

So a single button, like the spacebar (or "Jump" like we saw above) would return either true or false, it's either being pressed or not:

```
if(Input.GetButtonDown("Jump")) {
   // do something to handle the input
    }
```

Conversely, an axis returns a float (instead of a boolean), by default between -1 and 1, depending on which direction it is being pressed. The two predefined axes are "Horizontal" (both 'a'/'d' and 'LEFTARROW'/'RIGHTARROW') and "Vertical" (both 'w'/'s' and 'UPARROW'/'DOWNARROW'):

```
float h = Input.GetAxis("Horizontal");
float v = Input.GetAxis("Vertical");

if(h<0) {
   // move character left...
}
else if(h>0) {
   // move character right...
}
... etc ...
```

If you absolutely NEED to get a key directly:

```
if(Input.GetKeyDown(KeyCode.A)) {
   // do something
}
```

but I recommend you avoid this whenever possible.

Finally, if you look at the Input documentation you will notice there are variations of the functions (GetButton, GetButtonDown, GetButtonUp):

- **GetButton** returns true if the button is currently being pressed down, false otherwise
- **GetButtonDown** returns true if the button was pressed down THIS FRAME (only called once each time the button is pressed), false otherwise
- **GetButtonUp** returns true if the button was released THIS FRAME (also once each time the button is pressed), false otherwise

Instantiating Prefabs

A common operation to perform in games is spawning new GameObjects into the Scene (For example, spawning enemies as the player

walked across the level). To do this, use the Instantiate() method. There are two versions of Instantiate which take different parameters, and the syntax for calling Instantiate is slightly different depending on whether you need to assign the created GameObject to a variable or not.

Instantie can either have one or three parameters:

- Instantiate(prefab);
- Instantiate(prefab, position, rotation);

Where prefab is a reference to the prefab you want to spawn, and (optionally) the world position and rotation where the GameObject should spawn. In the one argument form of the method, the GameObject is spawned at the origin (0,0,0) with zero rotation.

The first argument, the prefab that is to be spawned, is generally a public GameObject variable. Once the variable is declared as public, a slot will appear in the component in the Inspector, and you can drag in a prefab that you have saved (make sure you are dragging the prefab that is saved in the Project Panel) which is then connected to the variable and can be instantiated.

Generally, when working with 2D, there is no need to specify a particular rotation, but you still need to specify a position. Since Unity uses something called Quaternions to specify rotation in 3D space, there is a placeholder rotation that essentially evaluates to no rotation (Quaternion.identity).

The position argument must be expressed as a Vector2 (if in 2D mode) which is basically a data type that is a fancy kind of array which is limited to two positions, and can be accessed by naming x and y as properties:

```
Vector2 myVector = new Vector2(myX, myY);
    float x = myVector.x;
    float y = myVector.y;
```

The final thing to note about Instantiate is the two variations in syntax, depending on whether or not you need to assign the resulting GameObject to a variable. If you do, then the normal method call needs to be followed with 'as GameObject' which is need to specify what type the Instantiate function needs to return. I'm not sure exactly why this is necessary, but it's just something that is part of C# and needs to be done.

Here is an example of using a basic Instantiate call:

```
    public GameObject enemyPrefab;
```

```
void Start() {
   InvokeRepeating("Spawn", 0.5f, 2.0f);
}

void Spawn() {
   Instantiate(enemyPrefab);
}
```

Assigning the resulting GameObject to a variable:

```
public GameObject enemyPrefab;

void Start() {
   InvokeRepeating("Spawn", 0.5f, 2.0f);
}

void Spawn() {
   GameObject ob = Instantiate(enemyPrefab) as GameObject;
}
```

Finally, adding in a position and rotation:

```
public GameObject enemyPrefab;

void Start() {
   InvokeRepeating("Spawn", 0.5f, 2.0f);
}

void Spawn() {
   GameObject ob = Instantiate(enemyPrefab, new Vector2(5, 10), Quaternion.identity) as GameObject;
}
```

Debugging

While MonoDevelop (and Visual Studio) do have built in debuggers that automatically interact with Unity and allow you to set breakpoints, watch variables, and most things that your favourite debugger can do, there are also a few strategies that can be run directly through Unity without having to set up the debugger and breakpoints for a simple bug. Namely, these strategies mostly consist of logging and print statements, as we used above when talking about Update and FixedUpdate.

- **Debug.Log(string s)** - there are various versions of Debug.Log (for example, Debug.LogError) but the main idea with all of them is they print a string to the Console. If you are having a problem where something isn't working quite right, put in a quick Debug.Log and print out the value(s) of variable(s) that you think would be contributing to the problem. If you have a problem where you don't think a function is getting called (or it is being called too frequent) you can use a Debug.Log inside of that function and see how often the Log is printed in the Console (just like we did with Update and FixedUpdate).
- **Debug.DrawLine / Debug.DrawRay** - These are useful when doing Linecasts and Raycasts (see next section). If your linecast doesn't seem to be behaving properly you can add a Debug.DrawLine to display the Line and see if the line is being drawn in the place you think it is.
- There are many other useful functions in the Debug class. For full documentation, see here.

Linecasting and Raycasting

Finally, the last scripting trick we will look at in this tutorial is Linecasting and Raycasting, which are basically the same thing, or they at least achieve the same outcome.

In essence, linecasting and raycasting draw an invisible line from a starting point in a particular direction (or towards a specific point in space) and tell you whether or not they hit any colliders along the way. It is a way of doing a check for line of sight, whether or not there is an unobstructed view between two GameObjects.

Linecasting and raycasting will be used a lot in AI, pathplanning, and Procedural Generation, but we will also see it in other mechanics, such as movement, in later tutorials.

For now, take a look at the documentation for linecasting and raycasting. This tutorial won't look at how to use them, but just know that they exist and they will be an important tool at your disposal throughout the course.

Animations

Contents

Animations Asset Pack

For this tutorial you will need a character sprite sheet to use in constructing an animated character. Feel free to use the spritesheet from the first tutorial, or the provided spritesheet for this tutorial, or find a free spritesheet online (a sidescrolling character with idle, walking, and jumping animations is recommended; this will make the first assignment much easier).

The Animator Component

Before we get into anything too crazy with animations take a second to review the simple animation we made in the first tutorial and familiarize yourself with the process. We will be doing a very similar thing in this tutorial, but since we will be creating more complex animations we will be constructing them entirely from scratch so the project remains organized and we have total control over what happens.

In tutorial 1 we simply selected a series of sprite images and dragged them all into the game at once. Unity interpretted this action as wanting to create a new GameObject with the selected sprites combined into the GameObject's default animation. This is a perfectly fine way to do it, but in this tutorial we are going to go through

a detailed explanation on how to do it, as well as how to combine multiple animations on the same GameObject and transition between them.

Start with a scene with an empty GameObject, let's call it Character for now. This will be the character with all of our animations attached to, and the basis for our first playable character in future tutorials.

In general, these are the steps we will go through for creating an animated character:

1. Add Animator Component
2. Create Animator Controller, and add to Animator
3. Create sprite Animations
4. Add Animations to Animator
5. Set up transitions between Animations
6. Write code to trigger Animation transitions in Animator

Add Animator Component

Select the GameObject, and click on the Add Component button. Search for Animator and select it.

Create Animator Controller

Like creating any asset inside Unity, RIGHT CLICK in the Project Panel, and navigate Create » Animator Controller. This will create a new, empty Animator Controller. Select your Character GameObject, and drag the Animator Controller in the controller slot in the Animator.

The Animator is the component that handles all the transitioning and manipulating of Animations. The Animator Controller we created is the device in which we actually save our animation structure (as we will create in a few minutes). The controller is independent of the Animator Component, which means you can save the controller as a variable, change animator controllers in GameObjects, and even apply the same Animator Controller to more than one GameObject. In our case, that would make every GameObject display the exact same sequences of Sprites, but in 3D development the animations are made up of changes in bone positions which can be applied to any 3D character that has a similar skeleton structure.

For sprite animations, Unity has a neat variation called an 'Animator Override Controller' which allows you to create many characters

with identical Animation structures, only having to define which animations are different in each subsequent GameObject. I won't go into detail in the tutorials, but I would recommend to consider using them in your project, where applicable. Find more information about them here.

Creating an Animation

Now that our GameObject has an Animator component, and is tied to the newly created Animator Controller, it's time to create our animations. I will only go through the process for one Animation, as it is the exact same steps for each subsequent Animation.

Creating an Animation can be broken into several simple steps:

1. Create Animation(s)

   In Project Panel, RIGHT CLICK » Create » Animation. This will create a blank Animation. Name it, and move it to the appropriate folder location in your project structure (for now just create an Animation folder, or something that makes sense to you. We will go over recommendations for project structures in the tutorial on Architecture.

   Once your animation(s) are created, drag them onto your GameObject. Repeat the Animation Creation process for every animation you want to add (you can add more later, if needed). Open the Animation window (Window » Animation) and select your GameObject. At the top of the Animation Window you should see a small drop-down menu. Click on the menu and you should see all of the Animations you just created and added to the GameObject.

2. Drag Sprites into the Animation Window

   In the Animation Window, select each of these entries in the drop-down menu one at a time. For each entry, drag in all of the sprites that are part of that animation (either one at a time, evenly spaced, or all at once). This will define which sprites will be played during that particular animation.

3. Check frame rate of Animation

   For each of the animations you created, you can press the play button in the Animation Window to see the Animation playing. It is likely the Animation will be too fast, because it is set to run too many samples per second by default. There is a box in the Animation Window right next to the drop-down menu labelled 'Samples'. Reducing this number will make the Animation play slower, increasing it will make the Animation play faster.

4. Uncheck Record Button

   Finally, notice the record button in the top left of the Animation Window, to the left of the play button. When you add sprites to the Animation, or make other changes, Unity will turn on the record button. The record button, when on, will take almost any change you make to the game and save it as part of the Animation. For example, if you change the position of a GameObject that new position will be baked into the Animation (which is usually the cause if you have a GameObject jumping around the screen weirdly). Another, more useful case, is tinting the Sprite by changing the colour in the Sprite Renderer. You can Change the colour of a Sprite to red in an 'injured' Animation, for example, to indicate a character getting hurt. By recording animation states (such as different colours at different points of the animation) the animation system will automatically interpolate between the two states (slowly transition from white to red, for example). The interpolation is linear by default, but you can define a non-linear interpolation if your animation needs it.

   If you aren't recording anything, it's best to uncheck the button right away, to avoid weird, unexplainable errors.

Creating the Animation Tree

Now that you have all of your Animations created (or at least some of them) we can create transitions between Animations. The Animations will be set up as a state machine, which each Animation acting as a single state. We will set up transitions between states, and these changes will be triggered by certain variable conditions between states. As long as the Animator is on, the Character will be in one state, and will play the corresponding Animation. If the requirements to move to another state (all of the conditions set in the transition) are true, then the GameObject will move to that other state, and start playing a different Animation.

To see and edit all of this, open the Animator window (Window » Animator). Again, make sure the Character GameObject is selected and you should see a window with all of the Animations you created/added to the Character GameObject, as well as an 'Entry' node, and an 'Any State' node. One of your Animations will be orange (likely the first one you added). This one is the default Animation, so the one that will play when the game first starts. When designing a character, it is usually an Idle animation of some kind, since the player isn't controlling the character right off the bat.

On the left portion of the Animator Window there are two tabs:

Layers and Parameters. We are only going to deal with Parameters. These parameters are variables that are declared within the scope of the Animator. They can be accessed and changed through scripts, but must be explicitly changed. So, if you name these variables exactly the same as in a script, that doesn't mean changing the variable in script will update the Animator variables.

To create a new variable, click the little plus sign and select the variable type. The options are Float, Int, Bool, and Trigger. The first three should make sense, and it should be easy to picture the kinds of things you would represent with them (floats can represent continuous variables like the speed a character is currently moving, an Int can represent states, a Bool can represent a binary state like whether or not an enemy is hostile and should attack). The one different variable is Trigger. A trigger is a special type of boolean, one that is always false, except for the moment you invoke it (or trigger it) which causes it to be true for a single frame, just long enough to cause a transition to happen, before it is flipped back to false. So, it's a bool you can flip and don't have to manually flip back. Triggers are useful for events, like attack or jump, which may never do anything with a second value, so why bother setting it back manually everytime?

You can create as many variables as you need (name them descriptively and appropriately, obviously). You can also set the default value the variable should have when the game starts (speed=0, for example).

With variables created they can be applied to transition between states. Right click on the initial state and select 'Make Transition'. This will create a little arrow that follows the mouse cursor around. Click on the state you need to transition to to complete the transition. You can click on the transition arrow to specify the properties of the transition, and define the conditions to cause that transition to happen.

For simplicity, let's only look at Conditions which will give the basic mechanics of the transitions. For a real game, you may need to look into the settings and exit time options to smooth the transition but that's for you to play around with and figure out when needed.

You should see an empty list called Conditions. This is where you can define the transition conditions that must ALL be true for the transition to happen. Click the plus sign to add a new condition. Select the variable you want to check, and select the value or relationship that needs to be true to cause this transition to happen. Add as many conditions as necessary.

As an example, let's say you are creating a transition between an Idle (stationary) state and a walking state. The transition would likely rely on a speed variable (how fast the character is currently

moving). You would have one condition that uses this 'speed' variable, and checks that it is greater than zero (or some small threshold number). This means that when your player is standing still and you press a key to make the player run the speed variable will be set and the walking animation should now be played because the character is moving (or should be).

Finally, let's look at how to set that variable in code to make the transition happen.

Triggering Animation Events Through Code

Incorporating Animation transitions in code requires a few things:

1. Create a Class variable to store a reference to the Animator component
2. Write game logic that requires a change in animations
3. Change the Animator variable when the change happens in code

I will show a script fragment first, and then explain the pieces of it:

```
public class MyAnimations : MonoBehaviour {

   private Animator anim;

   void Awake() {
      anim = GetComponent<Animator>();
   }

   //triggered from somewhere else, when this character needs to attack
   void Attack() {
      anim.SetTrigger("Attack");
   }
}
```

Let's start by looking at the first step, getting a reference to the Animator component. At the beginning of the script, a class variable of type Animator is declared. A class variable is a variable associated with this script, and is global to all functions in this script. So as long as this script is attached to a GameObject, once the variable has a value it will keep that same value.

In the Awake method the GetComponent method is called to look for the Animator component and assign it to the variable, to presumably be used later. Notice what GameObject the GetComponent

method is being invoke upon: seemingly nothing (there is no OBJECT.GetComponent<Animator>()). This is a special shortcut Unity let's you do. If you leave out the target (OBJECT.) at the beginning of the method called, Unity assumes you mean the GameObject you (this script) is currently attached to. In this case the character we are building and trying to animate.

So in simple terms, that line is looking at the GameObject THIS SCRIPT is attached to (this script could be attached to many different GameObjects) and finds the first component of type ANIMATOR. Assign a reference to that component to the anim variable. Now we have a variable that points to the Animator component attached to the same GameObject as this script. If you needed to get the component of a different GameObject, it would be identical, except you need to put an OBJECT. in front, to declare what GameObject you are targeting (perhaps player.GetComponent..., in which player is a GameOject variable storing a reference to a different GameObject). This can be very confusing (but it is incredibly important) to understand. It is essential to know which GameObject the script you are writing is or will be attached to, and know which GameObject you need to communicate with, and which GameObjects you have references to. More guidelines and suggestions will be provided in the Architecture Tutorial for common patterns and ways to minimize and organize inter-GameObject communication.

Finally, in the Attack method the SetTrigger method is called (on the Animator). In this case, it triggers a trigger variable named "Attack". If it were a Boolean variable it would be something like:

```
anim.SetBool("Attack", true);
```

For a full list of available methods, see the Animator documentation.

Putting it All Together

Now that we have gone through the basic process of setting up Animations (though with enough empty spaces, vagueness, and muddy explanations to have even more questions that need to be answered) take some time to practice what you have learned. Go online and find a sprite sheet (just google 'free sprite sheet character' or something similar) and go through the processes learned in the last couple tutorials. Go through cutting up the spritesheet, creating the GameObject, creating the individual Animations, and create transitions between them. Try to find a sidescrolling character (or use

the spritesheet provided on Moodle which will put you in a good position for the sidescroller tutorial and for Assignment 1.

Building a Simple Platforming Game

Contents

This tutorial is a broad overview of how to build a platformer-styled game. Many parts of the tutorial will refer to previous concepts, or require previously learned concepts to understand what to do. Please refer to previous tutorials if you unfamiliar with something. For specific requirements for the assignment, please see the Assignment description on the course webpage.

For the most part, from here on in, the tutorials will describe steps and concepts at a much higher level. We will be dealing with increasingly complex algorithms and structures, so the specific details won't be provided. Instead, the remainder of the tutorials are designed to give you an understanding of how the problem or mechanic should be implemented, and giving you a chance to implement it yourself.

So far in the course we have covered the majority of Unity basics, but there are still a few physics-related topics that have not yet been covered. Specifically, detecting collisions between GameObjects and using forces to push GameObjects around and make them move.

Detecting Collisions

In the first tutorial you likely saw the simplest form of detecting collisions: having the GameObjects actually collide and bounce away from each other (or push each other, depending on the Rigidbody2D settings). In some cases you not only want the GameObjects to collide, but to also make something happen in code (for example, decrease an enemy's health when the player's sword makes contact). In

still other cases, you want to avoid the physical collision entirely and just trigger the code when the GameObjects 'collide'. These are the three situations this section will look at.

For a platformer game, there really aren't any collisions that need to trigger code. It's mostly adding colliders to the platforms and the player, and using forces to push the Character around and jump. Nonetheless, we will look at the other two types of collision detection in detail here, which will be necessary for the second assignment.

As was mentioned in the Animation tutorial, it is useful to look at Unity from the perspective of "who am I?" when you are writing a script. It is useful to think of 'me' as the GameObject the script is attached to. This allows you to access sibling components on yourself, and many special callbacks and methods only apply to other components also attached to 'you'. An example of this is the Collider2D callbacks. When a collision happens, any script attached to the same GameObject as the Collider2Ds involved in the collision can receive the Collision event, using the 'OnCollisionEnter' method.

```
void OnCollisionEnter2D(Collision2D col)
{
   // code to be executed everytime THIS GameObject collides with something else
}
```

So, to register a collision callback, just put the above method in the script where you want the code to be (it must be attached as a component to the same GameObject as the Collider2D) and any code you write inside the method will be called everytime another Collider2D comes into contact with THIS GameObject's Collider2D.

Related to OnCollisionEnter2D are the methods OnCollisionExit2D and OnCollisionStay2D, which are used the exact same way but measure different moments in the collision event (after and during).

The above is used to detect a physical collision between two GameObjects with Collider2Ds attached. You probably noticed a checkbox on Collider2D components which says 'On Trigger'. If this box is checked, the Collider2D does not behave as a collidable GameObject (other Collider2Ds will not bounce off of it, they will pass right through). The purpose of a trigger Collider2D is to Register an event when something passes within the field of the defined Collider2D without the physical interactions to happen. Triggers are useful for things like moving the game forward (i.e. player walks through a trigger which causes enemies to spawn) and melee attacks (turn on a trigger when the character attacks and check if it is hitting an enemy or not). There are many other uses as well.

Detecting a Trigger Collision is the exact same as the previous

explanation, except the method signature is slightly different, to distinguish it as a Trigger Collision:

```
void OnTriggerEnter2D(Collider2D col)
    {
       // code to be executed everytime another GameObject enters THIS GameObject's trigger field
    }
```

Finally, notice that for each of the above method signatures a variable is passed in, either of type Collider2D or Collision2D. These are two different Object types in Unity, but it is possible to use either type of variable to get the 'other' GameObject involved in the Collision/Trigger. Since it is so easy to get the other GameObject, it doesn't really matter which of the GameObjects you put the callback on (Except the trigger callback has to be on the TriggerCollider GameObject) so pick whichever GameObject works best for the design of your game.

Adding Forces

Before we get to how to put together a platformer, let's look at how to add forces to a GameObject. Specifically, we will be adding a force through the GameObject's Rigidbody2D component (so it needs to have a Rigidbody2D component).

The first thing you should do is declare a class variable in a script to store a reference to the Rigidbody2D, similar to what we did with Animations, and assign that variable in Awake:

```
public static class myClass : MonoBehavior {
   Rigidbody2D rbody;

   void Awake() {
      rbody = GetComponent<Rigidbody2D>();
   }
}
```

From now on in this script, the variable rbody will reference the Rigidbody2D component (unless the variable is assigned a new value or the Rigidbody2D component is removed). This will just save us from having to call 'GetComponent<Rigidbody2D>()' everytime, which makes the code a little cleaner and simpler to read and comprehend.

Rigidbodies have an 'AddForce' method, which applies a physical force onto the Rigidbody2D, pushing it in the desired direction. This

force will interact with the built in Physics2D system, fighting against gravity and other competing forces, friction, etc. The signature for the AddForce method is as follows:

```
public void AddForce(Vector2 direction, ForceMode mode=ForceMode2D.Force);
```

The first argument is required (it specifies the direction and magnitude of the force to be applied; the X and Y components of the force). The second argument is optional, and specifies the type of force to be applied. Let's look at a few examples:

```
rbody.AddForce(new Vector2(0,0));
```

This one would add no Force to the GameObject, because it is pushing 0 in the x direction and 0 in the y direction.

```
rbody.AddForce(new Vector2(100,0));
```

This one would push 100 units of force to the right (when applying forces the forces often need to be in the hundreds to notice the force; applying 2 units of force won't do anything).

```
rbody.AddForce(new Vector2(0, -250));
```

This one would pull the GameObject downwards (-250 in the y direction). Finally,

```
rbody.AddForce(new Vector2(200,200));
```

will push the GameObject up-right at a 45 degree angle (200 right and 200 up). This will be the basis for the platformer movement described below.

Designing the Level

Designing a simple platformer level is simple: find rectangular sprites (bricks, platforms, whetever style(s) you want to use) and apply a BoxCollider2D. Move the bounds of the BoxCollider2D to fit snuggly around the edges of the platform sprites. Finally, check 'isKinematic' on the Rigidbody2D. Making it Kinematic basically means the Object will have infinite mass, so you won't have the player pushing the platforms around when she jumps on and off of them.

Writing the Player Script

Finally, use the information from this tutorial and previous weeks to put everything together in a player script. See the Assignment Description for specific details and requirements, but the basics to assemble a platformer are as follows:

- Build Level with platforms
- Create walking, idle, and jumping animations for player character
- Add Animator to player, design animation transitions
- Get input for moving left/right and jumping (horizontal axis and some button, such as spacebar)
- Use input to add forces left,right and up to move character around the level
- Use Input to trigger appropriate animation transitions

Changing the Settings

If you have completed everything up to this point you should have some sort of working version of a platformer but there are likely several things wrong with it, or at least not as smooth and clean as you would like. For example, maybe the character jumps too high, or not high enough, or maybe it doesn't fall fast enough. All of these types of things can be fixed by tweaking the values of the forces until if feels just right. This takes time and it isn't an easy step. This is why making games can be difficult and time consuming. Don't worry too much about getting things perfect for the assignment, but for your projects it is important to get the input to feel just right.

Designing a Good Project Architecture

Contents

Understanding a Component-Based Architecture

One of the trickiest part of understanding Unity is wrapping your head around the project architecture, which is likely different from programs and languages you have previously used. As was mentioned in the first tutorial, Unity uses a component-based architecture, rather than Model View Controller (MVC) which is the common structure for Object Oriented programming.

It's important to understand that when you are writing a script, you are writing a class, which means it can exist in multiple places simulataneously (when instantiated as an object). So, as a simple example, if you write a class (or script) that moves a character around, keep in mind that the same script could be attached to ANY GameObject and it would allow you to move that GameObject around just like the character did. The majority of this tutorial will look at this idea: If you make a script simple and general enough, you can reuse those scripts on all kinds of different types of GameObjects, rather than writing almost duplicate code. The question is, how do we design a script to make it simple and general?

Organizing Your Project

Organization is key in developing a great game (or even in making a working game...). In this tutorial we will look at tips and tricks for designing a good project architecture, but the first step is to create an organizational structure early and stick to it. It is very easy to create a new animation and not put it in the Animations directory, but that one Animation adds up and soon becomes 100 Animations, all named things like "New_Animation_1", "MyAnimation_0", "Idle", etc. Now try figuring out which Animation goes with which character.

When it comes down to it, it doesn't matter how you organize your project, as long as you organize it in whatever way makes sense for your group. Generally, I organize my projects like this:

```
Assets/
   Animations/
      Clips/
      Controllers/
   Prefabs/
      Characters/
      Objects/
   Scenes/
   Scripts/
   Sprites/
```

Each of these directories would also be subdivided as more assets are created or added. Generally, I find that once I have more than ten things in a directory it helps to subdivide it to make it cleaner. Again, do what works for you, but if I ask you where your Enemy's Idle Animation clip is stored and you don't know right away, you have a problem with your organization.

Inheritance and Interfaces

While this tutorial will focus on making simple, general scripts, keep in mind that the rules of class Inheritance exist in C# (and therefore you can use them in Unity). I'm not going to get into talking about inheritance in these tutorials, but it works basically identical to java or regular C#, if you are more familiar with that. Inheritance can still be quite useful when working with Unity, and there are times when it is the best solution to a problem, but I find that if you focus on making simple and reusable components you can often avoid having to deal with object inheritance in many cases, so we will focus on that for these tutorials.

Reusable Components

When building Reusable Components, there are two keys to keep in mind when designing them:

1. Keep the scripts general

   Keeping a script general can mean obvious things like keeping everything as a visible class variable (for example, a CharacterMover script would have a speed variable, so both fast and slow characters could use the same movement script), keep the code simple, and break everything into small methods. The generality can also refer to the design itself. As mentioned above, it would be great to design a movement script that is simple and general enough to be attached to every single GameObject that moves: The Playable Character, Enemies, NPCs, Bullets, etc. This is very possible, and it is the sort of architecture you should strive to have in your game, since it allows quick and easy debugging, and expanding your game becomes incredibly quick and easy. Building a good architecture takes time and it is usually a tradeoff between more work early on but usually pays off with more results at the end of the project.

2. The script should only do ONE thing

   A related point is that every script should do one thing. Up until now (and moving forward) these tutorials, and likely your projects/assignments will not always follow this rule, and that is OK. It is sometimes unavoidable, or much easier, to not follow the rule, but always be aware of what you *should* be doing. In bigger games it becomes necessary to have better architecture, but starting out it is often helpful to just try things and see what they do and how they work. Once you get it working, though, refactoring and redesigning is usually a very very good idea.

Some assignments will require you to follow a particular architecture, and your project milestones will be marked on the quality of the project architecture and organization, so it is important to at least try a few of the ideas in this tutorial and start thinking about what a good way of organizing your project will be.

Interaction Between GameObjects

Just as a quick review, we saw in the scripting tutorial a brief introduction to communicating between GameObjects. Two ways of communicatng were discussed, and the first one was explained in

somewhat detail. We looked at how to get a reference to a GameObject that exists in the game and how to then access methods and variables through the components attached to that GameObject. The tutorial also mentioned the second way, which is sending messages, but it wasn't looked at in detail. We will look at that now.

Sending Messages

In our quest to write and design scripts that are as general as possible, a key part will be sending messages, as opposed to accessing GameObjects directly. In the previously seen method, you need to not only find the GameObject, but you also need to know which component(s) it has currently attached and know which variables and methods are available to be used. This works for a simple game, but to do more interesting and complex things it becomes necessary to come up with a better architecture.

Unlike the direct-access solution, message passing does not require you to know what components a GameObject has. Instead, you send a message to a GameObject (you obviously still need to find the GameObject, how else would you specify the target?), but instead of passing the message directly to a particular component (for example, the Attack() method we made in the Animation tutorial) you simply pass a message to ALL scripts and components attached to that GameObject. If the Component has the corresponding method, it is executed, if not then the message is ignored. This is the key to designing a general architecture and minimizing dependencies between GameObjects.

Here is what the SendMessage signature looks like:

```
public void SendMessage(string methodName, object value, SendMessageOptions options);
```

Its use is just like any other method, except you call it directly on a gameObject, without needing to get a reference to a particular script or component. For example:

```
public class ExampleClass : MonoBehaviour {

    GameObject enemy;

    public void DealDamage(float damage) {
        enemy.SendMessage("Damage", 10.0f);
    }
}
```

In this example, the script is likely on a player GameObject. The DealDamage method would be called when the player attacks an enemy and a collision has been detected, so the player needs to send the damage message to the enemy. The first parameter is a string that is the message that needs to be sent (and the receiver needs to have a public Damage method in order to catch the message). The second parameter is an optional argument to pass with the message, in this case the amount of damage that needs to be done.

Note there is a third parameter to SendMessage, which allows you to specifiy whether or not the target GameObject needs to have at least one public method that will receive the message. If your design requires all potential targets to have at least one message catcher, then it is a good idea to pass 'SendMessageOptions.RequireReceiver' which will throw an error if there is nothing to catch the message. On the other hand, if you will be passing lots of messages and due to your design some targets may not be handling the message, you can pass 'SendMessageOptions.DontRequireReceiver' instead, which will just ignore invalid messages.

Let's look at a brief example of how message passing could work in a game. This is not an exhaustive or definitive example, but should instead be used to get you thinking about how a good architecture could look and how you would build it in your own projects/assignments.

- We have a game with a player and several enemies
- The Player has an Input script which records all of the User Input in Update()
- The Input script broadcasts an 'InputUpdated' method to the Player GameObject which contains parameters for all input values (this could be separated into individual methods, as needed).
- The Player has an Attack script, which is triggered by a particular key pressed (let's say SPACEBAR)
- The Attack script reacts to the SPACEBAR being pressed, and broadcasts an 'Attack' message to the Player
- The Attack message is caught by an Animation Script (which triggers the attack animation), a Collision script which determines whether an enemy is within range to be damaged, as well as anything else in the player which would respond to an attack (perhaps a cooldown or combo script)
- If the Collision script detects an enemy to damage, it would send a 'Damage' message to the Enemy

- The Damage message would be caught by the Enemy's Animation script, Health script, etc.

Notice how elegant this method is. All the Player had to know about the enemy was that it could (and should) be damaged when the player's sword hits it. Beyond that, the Player didn't have to know how the enemy was constructed, what it's animation tree looks like, or how to reduce the enemy's health. The Player just said 'Damage' (probably also would need to pass a number with how much damage to do) and the Enemy responded accordingly. You could completely redesign every aspect of the enemy, but as long as it still caught the Damage message the game would work perfectly.

Finally, you may be wondering why I included the Input script at the beginning. It's true, it is somewhat unnecessary, since it's basically doing the same thing as Unity does to broadcast the Input callbacks to all MonoBehaviours. It may seem like adding an extra layer to do nothing, but there is at least one situation where this would be extremely useful to do:

- Imagine you are building a fighting game (like Smash Bros., for example)
- Players press buttons to perform attacks and damage each other
- Now say you want to add a 'replay' feature which replays the battle (maybe allowing players to move around the arena, slow down/speed up time, etc.)
- This may seem like a daunting task at first, but wait. You already have a script which captures all input for every player and broadcasts it to the player GameObject.
- it wouldn't be too much of a stretch to imagine adding a 'record' script to every player which receives the Input message and saves the input data in a file or database.
- In replay mode, it would simply be a matter of turning off the Input script, and adding a script which reads from the data file (or database) and use that data in the Input broadcast.
- It would still take a bit of work to make it perfect, but that is a great start to implementing a record function, all from our organized architecture

ACTIVITY: A Sample Character Architecture

- Now that we've gone through the basic idea behind setting up a good architecture, take some time to go through starting to actually build it. Create the framework for a Playable Character.

- Start by making a list of the kinds of things a player character needs to do (as you have seen so far in the course). Also consider what an Enemy character needs to do. Try to design both characters as similar as possible (the only real difference should be the Player has an Input collecting script which sends messages and the Enemy has an AI script which can send the exact same messages). Each item on your list should be a separate script in your eventual character
- Make a second list of the different interactions that will need to happen between components (and between player and enemy)
- Look at your lists of scripts and interactions and decide which messages will need to be passed and when.
- Finally, create a blank skeleton script for each item in your first list. Put in the methods to catch the messages and start fleshing out code to match your lists above. You likely won't get everything finished and working, it is quite a bit of work and we haven't covered everything yet. When you get to something you don't know how to do, put a comment where it should go with a note of what it will eventually do.
- Try to integrate your Animations from the previous tutorial with your character (or rebuild them)

NOTE: This ACTIVITY will be the starting point for the second assignment, so take the time now to get started early, if you leave the entire assignment for the last minute you will not get it finished.

Building a Simple Top Down Game

Contents

This tutorial will introduce several new concepts (A new type of character movement and shooting projectiles)

Movement: Manually Changing Position

In the previous tutorial on how to build a simple platformer the Applying Forces method of movement was described and expected for that assignment. For our top-down shooter game, we will look at a second type of movement: manually changing position.

This new type of movement has several downsides when compared to the applying forces method, but it has one or two situations where it significantly improves performance and makes development simpler.

The downside:

- Can't use built-in physics system to manage Velocity, Acceleration, and Collisions (it doesn't work very well, at least)
- For some types of games, can feel unnatural

Where the method wins:

- Makes implementing networking a breeze (ha, not really, but much much more straightforward)

- much cleaner collisions with less jitter

At the end of the day, both methods work, but the big issues here are collision jitter and networking. You may have seen that sometimes if you move a GameObject against a collider (like a wall, for example) the GameObject will bounce against the wall and jump back and forth as the forces pushing the GameObject fight against the collision detection. This is an artifact of the input not matching up with the collision detection frequency and leads to weird things happening. If this happens, trying this other movement method may help.

The second benefit for using this method is networking. By maintaining the position manually, you have complete control over every step in moving the GameObject, and so you can send a networked message everytime you update position. Using the forces method, while it feels more natural, make it difficult to be able to send an accurate enough message across the network to maintain the correct position of every GameObject. This requires you to periodically check the position error and implement extra code to correct position when the error gets too large. It's not unheard of to use this method for networking, but it adds complexity and much more work to make it smooth and usable.

When moving a character (or any GameObject) using this method, you will usually do the following steps:

1. Calculate where the GameObject should be based on the input/ai
2. Use a raycast (or linecast) in the direction where character should move
3. if the cast detects something in the way, do not move
4. else, move the character to the desired position

This way, the moving GameObject will never stutter against a collider, it will predict if it is going to hit it and never actually move against it. The consequence of this cleanliness, however, is that 'OnCollisionEnter2D' methods won't actually be called. Instead, you will need to use the raycast to detect the potential collisions and include collision-triggered code there (such as destroying enemy when bullet collides).

Linecasts and Raycasts

Unity has two commonly used methods for performing line of sight calculations: Linecasts and Raycasts. Both of these do the same thing, more or less, but the method of using them is slightly different.

From a high level, you will use one of these two methods if you want to see if a GameObject is between two points (such as the player and the enemy), or you want to detect if you are close to a collider (such as performing a ground check in a platformer game).

In a linecast, you define two points (Vector2s): the start point, and the end point (you can also provide a layermask which determines which layers are affected by the cast). The linecast will draw a line between the two points and returns a RaycastHit2D object which contains all of the information about the **first** collider2D the linecast hit, going from start to end point.

Almost identical is the raycast, which does essentially the same thing except represents the line using a start point and a direction, instead of two points like the linecast uses.

While both can be used interchangeably in most situations, it is generally easier to use linecasts to perform line of sight checks between two things (such as an enemy AI checking if the player is in an unobstructed path which will trigger the enemy to become hostile. Raycasts, on the other hand, would be useful in something like a stealth game where the enemy would use a raycast to determine if the player is in the direction they are facing, rather than just plain line of sight. This allows the player to sneak around behind the enemy without being detected.

In terms of the movement described in the previous section, linecasts could be used to detect if something is in the way when the player goes to move. Since the player movement script calculates the exact position the player should move to, the script can perform a quick linecast between where it currently is and where it will be after moving. If the linecast finds something, then that means there is something obstructing the player's path and shouldn't move. A more sophisticated version of this could move the player right up to the edge of the collider2D, but not push the player completely against it.

Shooting Projectiles

Shooting projectiles combines many of the previously mentioned topics together into one: Getting user input, Instantiating Prefabs, moving a GameObject, and Collision detection. It is assumed you already know each of these pieces (see previous tutorials if unsure) so the remainder of this tutorial will focus on designing the architecture at a higher level, using the pieces learned in previous tutorials (and from online research/looking on forums).

The steps for shooting a projectile are as follows:

1. Get user input (for example, SPACEBAR pressed)

2. Instantiate projectile prefab (at a position on the player that makes sense, such as the barrel of a gun)
3. Set the direction the projectile should move (depends on how you design a movement script)
4. Projectile should move in that direction until it hits something (enemy, wall, etc)
5. Collision detection should behave according to game design or assignment specifications

Designing the Level

- Select an image or tile images to make the background (set the sorting layer in the sprite renderer for these image to a negative number, so they are behind the players and enemies).
- Create four rectangle sprites with box colliders, and place around the edge of the gameplay area (same as the platforms in the platformer (Assignment 1))

Designing the Player

- Get user input to move and rotate player (horizontal axis should rotate left/right, and vertical axis should move forward/backward)
- Write code to move/rotate player
- Create script to shoot projectile straight forward in direction player is facing

Designing the Enemy

- Simple AI (literally just make it move towards the player at a constant speed) (**HINT: look at the Vector3.MoveTowards() method**
- Write code to move/rotate enemy
- Enemy should die when projectile collides with it

Maintaining Character Health

The character may or may not have health, depending on project or assignment requires. If the health is a one-hit, then the player should die just like the enemy when hit by a projectile, and restart the level.

Incorporating health at a more complicated level (more than one hit) requires a health variable. The best way to incorporate this is to create a health.cs script which maintains a health value, has public functions to increase/decrease health, and checks for health <= 0 and destroys GameObject. By doing this, you can use the same health script on Players, enemies, even on the projectiles.

Creating an Enemy Spawner

The final piece of designing a top-down shooter is spawning enemies. This is very similar to spawning a projectile, but instead of spawning an enemy on keyboard input, it will spawn enemies periodically. You can create an empty GameObject(s) and add a 'spawner' script to it. This script should have a public variable to put an enemy prefab in (or even a list of possible enemies) and use InvokeRepeating to periodically spawn another one).