

## Unity 2D Tutorial

It used to be quite hard making a 2D game in Unity 3D. The basic idea was to apply textures to 2D four-sided polygons called "quads," adjusting their offsets with a script to create animations. The older Unity physics engine works in 3D, so you had to make sure the sprite objects had sufficient depth to interact with each other while ensuring they didn't accidentally rotate around their x- or y-axes.

The only alternative was to use one of the various add-ons available on Unity's Asset Store, (e.g., 2D Toolkit or Orthello 2D Framework), any of which include great features but restricts a game to work within its own set of constraints. And most of these were not cheap if you wanted the full feature set.

While all of these options are still available, Unity 4.3 now has native tools that add a new dimension to your workflow options: the 2nd dimension!

Note: This tutorial assumes you have at least some experience with Unity. You should know the basics of working with Unity's interface, GameObjects and Components, and you should understand an instruction like, "Add a new *droid* to your scene by dragging *droid* from the *Project* browser into the *Hierarchy*."

If you think that sounds like [C-3PO](#) on a bad day, or if you'd like a moment to get yourself into the right mindset for dragging *droids*, you may want to go through a tutorial that gives a more basic introduction to Unity, such as this [one](#).

Finally, note that the screen captures in this tutorial show the Unity OS X interface. However, if you're running on Windows don't worry - since Unity works the same on Windows most of these instructions will still work just fine. There will be a few minor differences (such as using Windows Explorer instead of Finder) but you'll get through it.

### 1) Downloading and Installing Unity 4.3.x

You can download Unity 4.3.x from [here](#). Follow the online Unity instructions to install it. If this is your first time installing Unity, you get a free 30-day trial of Unity Pro! See the Unity webpage for more info.

### 2) Game Resources

You'll also need some art to make a 2D game. Fortunately, I made some Android images for "droid vs apples". They are in the same zip file as this tutorial in a folder called "tutorial\_resources".

### 3) Starting the project

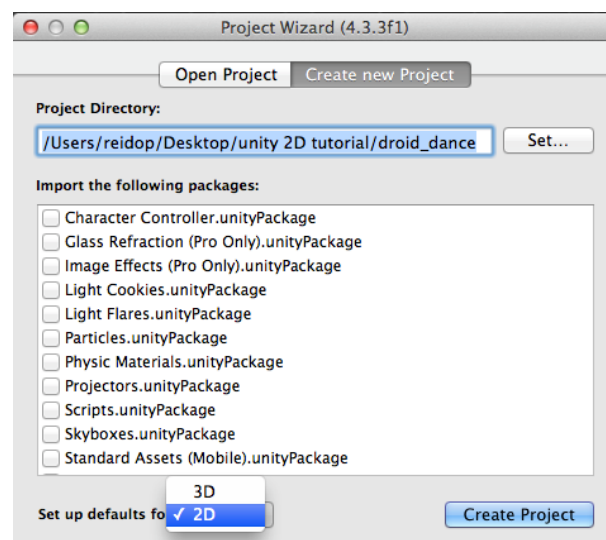
Open Unity and create a new project by choosing

**File > New Project...**

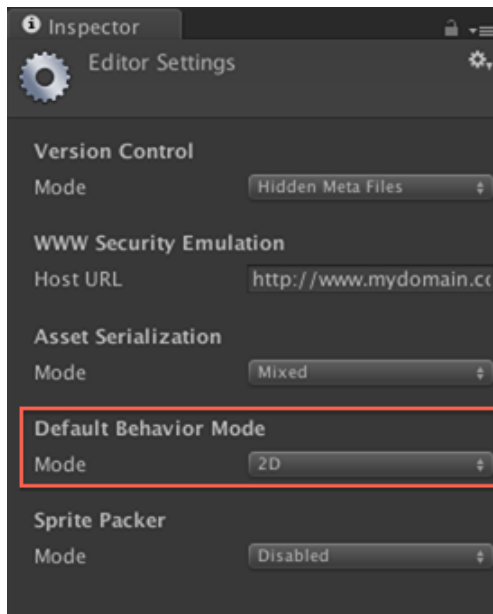
Click **Set...** in the Create new Project tab of the Project Wizard dialog that appears.

Name the project *droid\_dance*, choose a folder in which to create it, and click Save.

Finally, choose 2D in the combo box labeled **Set up defaults for:**, as shown below, and click **Create Project**:



The above-mentioned combo box is the first 2D-related feature you'll come across in Unity. It's supposed to change the default import settings for your project's art assets, but so far I haven't seen it work properly. Fortunately, this isn't a problem because you can change this setting in your project at any time, and doing so works fine.



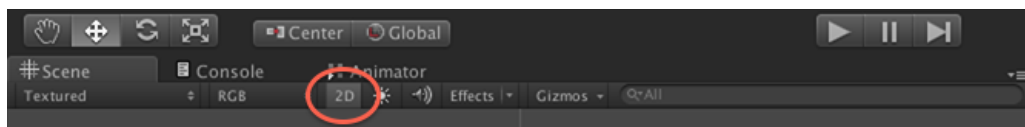
To ensure it's set properly, and so you know how to change it if you ever want to, choose

**Edit > Project Settings > Editor**

to open the Editor Settings in the Inspector. In the **Default Behavior Mode** section, choose 2D for the **Mode** value, as shown below:

The **Default Behavior Mode** defines the default import settings for your project's art assets. When set to 3D, Unity assumes you want to create a Texture asset from an imported image file (e.g. a .PNG file); when set to 2D, Unity assumes you want an asset of type Sprite. You'll read more details about Sprite assets and import settings throughout this tutorial.

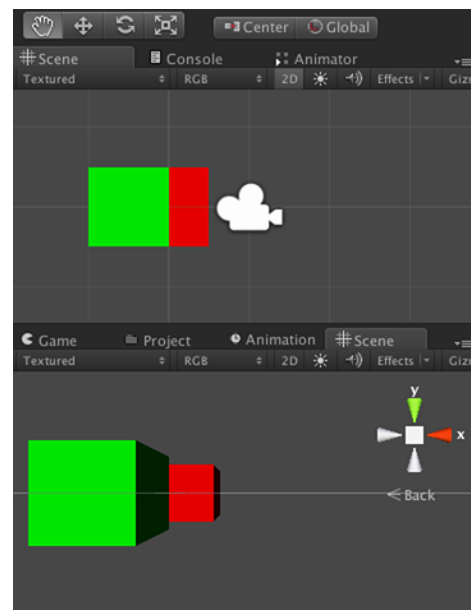
The next 2D feature you're faced with is the 2D toggle button in the Scene view's control bar. Click the 2D toggle button to enable 2D mode, as shown below:



This button toggles the Scene view's camera between perspective and orthographic projections. What's the difference?

When viewed with a perspective projection, objects appear smaller as they move further away from the camera, just like objects in the real world look when you see them with your eyes. However, when viewed with an orthographic projection, an object's distance from the camera doesn't affect its size. Therefore, in 2D mode, an object that is further away from the camera will appear behind any closer objects, but its size will remain unchanged regardless of its position.

The following image shows two Scene views, each looking at the same two cubes from the same location. The top view is in 2D mode while the bottom one is not.



The previous screenshot also shows how 2D mode hides the Scene Gizmo that lets you change the orientation of the Scene view's camera. With 2D mode enabled, the orientation is fixed so the positive y-axis points up and the positive x-axis points to the right.

Important: Toggling this setting has no effect on how your game finally appears when played - that's determined by the camera(s) you set up in your scene - but it can be helpful when arranging objects. You'll probably move back and forth between these two modes while creating your own 2D games, and even sometimes while creating 3D games, but this tutorial's screenshots all show the Scene view in 2D mode.

#### 4) Sprites Made Easily!

How easy is it to add a sprite to your scene using Unity's new features? Try the following experiment to find out.

**Step 1:** Drag *rat.png* from your **Game Resources** folder into the Scene view.

**Step 2:** Use some of the time you save making your game to send a thank you note to the Unity devs.

That was pretty easy! If you got lost, just re-read those instructions and try again.

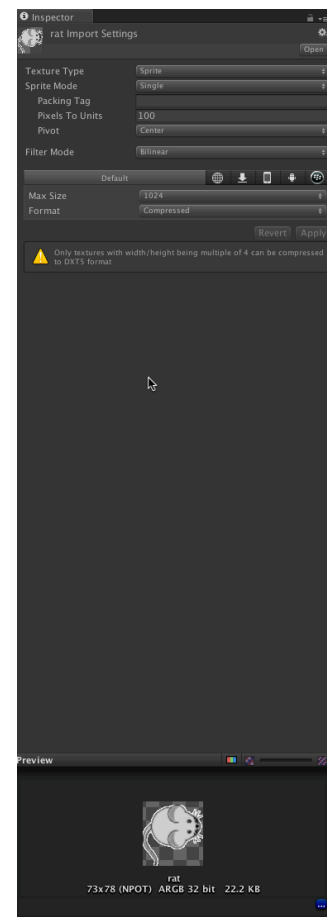
This demonstration was simplified by relying on Unity's default import settings, which oftentimes won't be correct for your images. However, this serves to illustrate a point - Unity's new features make working in 2D amazingly easy! The rest of this tutorial covers everything you'll need to know to really get started working with 2D graphics in Unity.

#### 5) Sprite Assets

Select *rat* in the Hierarchy and look in the Inspector. Your Inspector most likely won't show the same position that you see in the following screenshot, but don't worry about that right now. What's important to note here is that, in order to display the *rat* in the scene, Unity attached a **Sprite Renderer** component to a **GameObject**.

It's not obvious, but Unity created geometry for the object, too. For each Sprite, Unity creates a mesh that basically fits the non-clear pixels in your image. Notice the blue mesh in the following image of the

By creating a mesh like this rather than applying your sprites as textures on a quad, Unity can improve your scene's fill rate at render-time. It also makes creating polygon colliders easy, but that will have to wait for another tutorial.



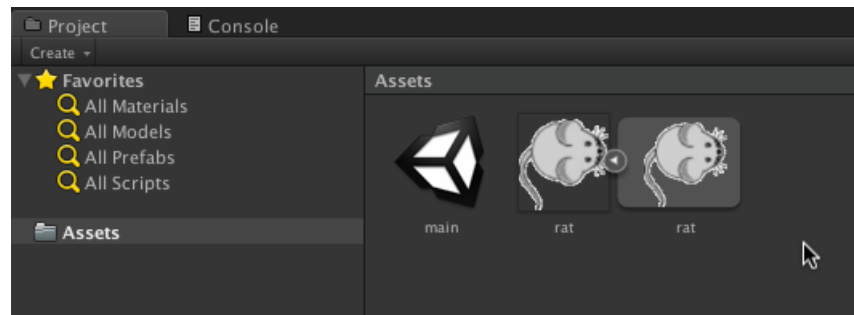


We'll learn about the Sprite Renderer's properties throughout this tutorial, but for now, look at the field labeled **Sprite**. This shows the name of the Sprite asset assigned to this renderer.

You can only assign one **Sprite** at a time, but later you'll learn how to update this field at runtime to create animations.

As you can see in the following image, the **rat** GameObject has a Sprite named **rat** assigned to its renderer.

Be sure the **Project** browser is visible. Then click inside the **Sprite** field in the Inspector to locate and highlight the **Sprite** asset in the **Project** browser, as shown here:



Note: The highlighted border fades away after a few seconds, so if you don't notice it, click the Sprite field again. Of course, with only one asset in your project, it's unlikely you'll miss it.

As you can see in the previous screenshot, Unity highlighted an item named **rat** inside the Project browser, which is a child of another object, also named **cat**. Two cats in the Project browser? Yeah, that could be confusing. Here's what's going on:

- The parent **rat** is the Texture asset. It's a reference to the original art file you imported, **cat.png**, and controls the import settings used to create the Sprites from your artwork. As you can see, it shows a nice thumbnail of the file's contents.
- The child **rat** is a Sprite asset that Unity created when it imported **rat.png**. In this case, there is only one child because Unity only created a single Sprite from the file, but later in the section on slicing sprite sheets you'll see how to create multiple Sprites from a single image.

**Note:** Unity's **Sprite** class actually only contains the information needed to access a **Texture2D** object, which is what stores the real image data. You can create your own **Texture2D** objects dynamically if you want to generate **Sprites** at runtime, but that requires much more coding than we can concern ourselves with now.

As you saw with **rat.png**, you can add Sprites to your scene by dragging art assets from the Finder directly into the Scene view (or the Hierarchy, if you'd like). But more commonly, you'll add assets to your project prior to adding objects to your scene.

Add to your project the remaining image files you downloaded: **background\_CB.png**, **enemy\_Apple.png**, and **kitkat\_andy.png**.

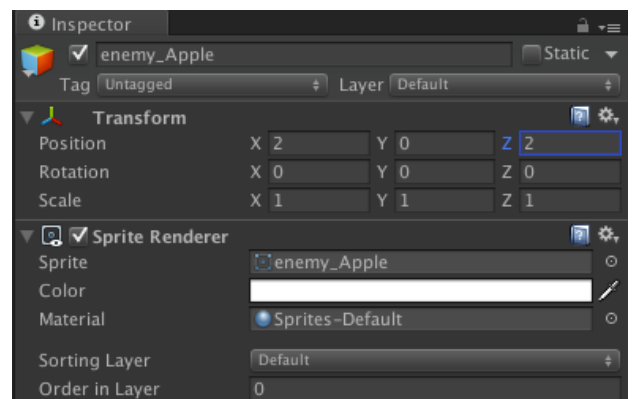
Unity gives you the following five options to get assets into your project:

1. Drag files from your Finder window into the Project browser.
2. Go to Assets > Import New Asset..., select your files and click Import.
3. Right-click within the Project browser, choose Assets > Import New Asset..., select your files and click Import.
4. Within your OS, add the files directly to your project's Assets directory, or one of its subdirectories. Unity refreshes your project automatically to keep assets up to date. Warning: Although it's ok to add assets this way, you should never delete assets directly from your file system. Instead, always delete assets from within Unity, because Unity maintains metadata about your project's assets and modifying the file system directly could corrupt it.
5. Of course, you can also drag files directly into the Hierarchy or the Scene view, but doing so has the additional effect of creating a GameObject in the current scene.

Add an enemy to your scene by dragging **enemy\_Apple** from the Project browser to the Hierarchy.

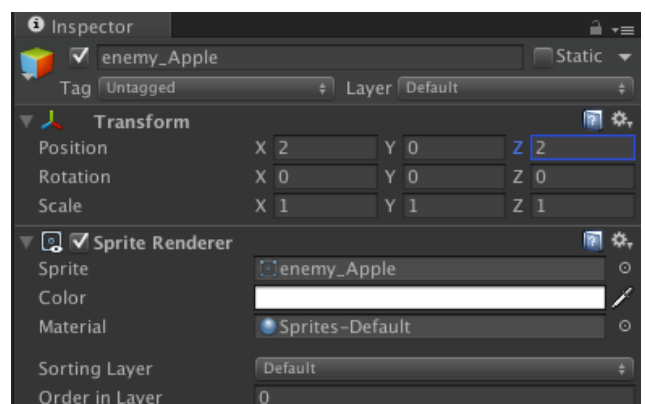
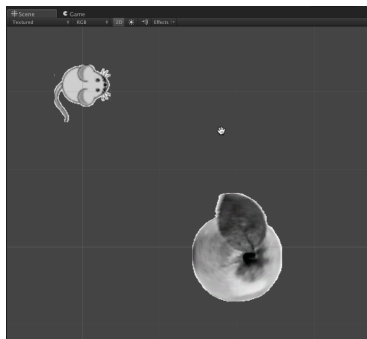
Just like with **rat**, there are two items named **enemy\_Apple** in the Project browser, but it doesn't matter which one you choose. That's because dragging a **Sprite** asset (the child) always uses that specific **Sprite**, whereas dragging a **Texture** asset (the parent) uses the first child Sprite, which is the same thing in a case like this where there is only one child.

Select **enemy\_Apple** in the Hierarchy and set its **Transform** component's **Position** to (2, 0, 0), as shown here.

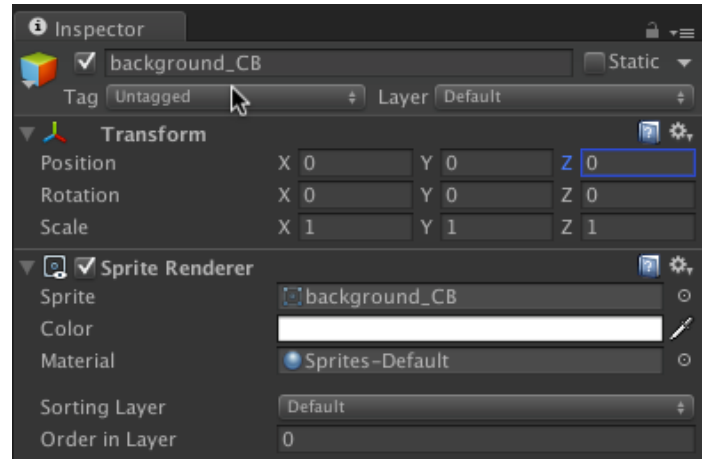


Before your scene starts getting sloppy, select **rat** in the Hierarchy and set its Position to (0, 2, 0), like so.

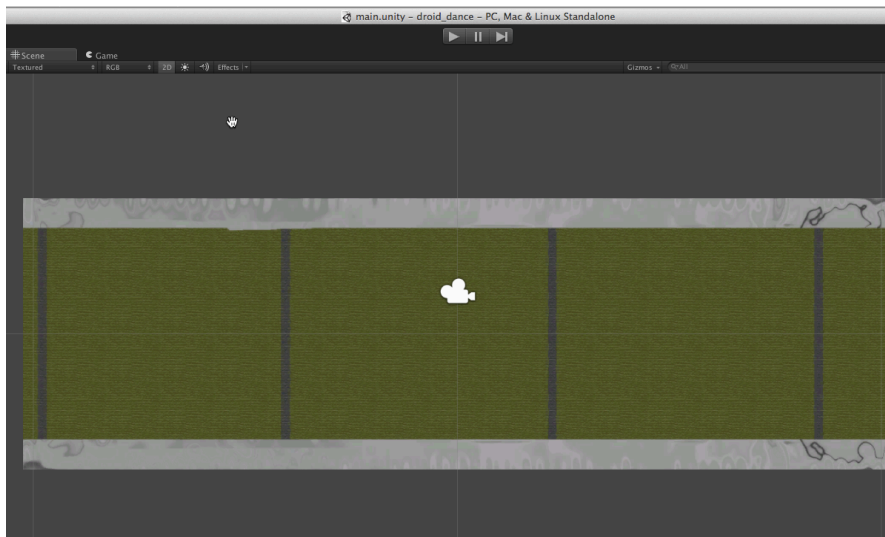
Your scene should now be arranged like the following image:



Finally, drag **background\_CB** from the **Project** browser to the **Hierarchy**, and set its **Position** to (0,0,0), as shown here:



You'll improve the **background\_CB** image quality a bit later, so don't worry if it doesn't look quite right. (Hint: Importing **background\_CB.png** is one of those times where Unity's default settings aren't correct.) Your Scene view will now look something like this:



Don't be alarmed by the fact that you can no longer see the **rat** or the **enemy\_Apple** in your Scene view.

They're simply behind the background.

Next we need to prepare the sprites for **kitkat\_andy**.

## 5) Slicing Sprite Sheet Assets

We already imported **kitkat\_andy.png** into the project, but the file is different from the other ones. Instead of a single image, it contains several, as shown below:

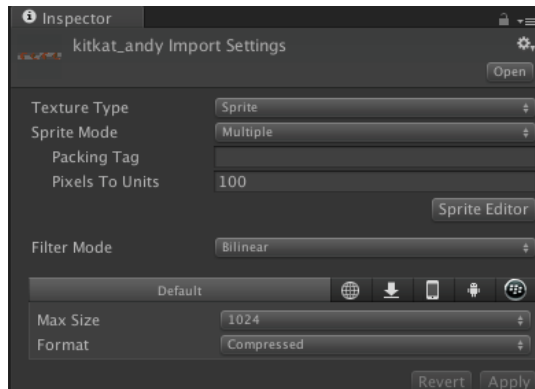




Such a file is usually referred to as a **sprite sheet**, and you'll want Unity to create a separate **Sprite** asset for each of the sheet's individual images.

Expand **kitkat\_andy** in the **Project** browser. As we can see, Unity created a single child - a **Sprite** containing the entire image.

Unity offers a simple way to treat this image as a sprite sheet. Select the top-level **kitkat\_andy** in the **Project** browser to open its Import Settings in the **Inspector**.



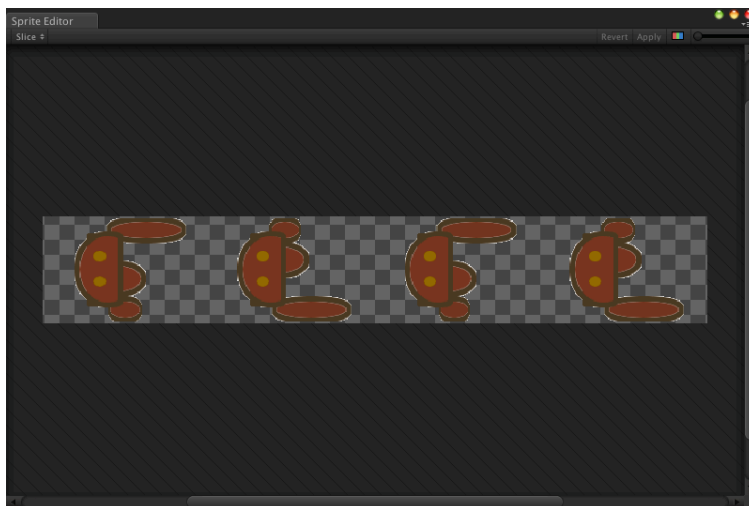
Set **Sprite Mode** to **Multiple** (see the image) and click **Apply**.

Choosing this option caused a new button labeled **Sprite Editor** to appear. It also removed the **Pivot** property, because each individual sprite will define its pivot point elsewhere.

Notice in the **Project** browser that **kitkat\_andy** texture asset no longer has children, as indicated by the lack of a small arrow on its right side.

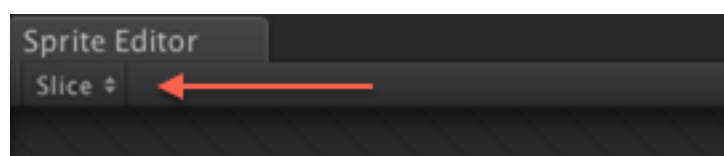
In this state, the **kitkat\_andy** texture is unusable. If you tried to drag it into the Hierarchy, you would get a message indicating it has no **Sprites**. That's because you need to tell Unity how you want to **slice** the sprite sheet.

With **kitkat\_andy** selected in the **Project** browser, click **Sprite Editor** in the **Inspector** to open the following window:



The **Sprite Editor** lets you define which portions of an image contain its individual sprites.

Click the **Slice** button in the upper left of the window to start defining sprites.

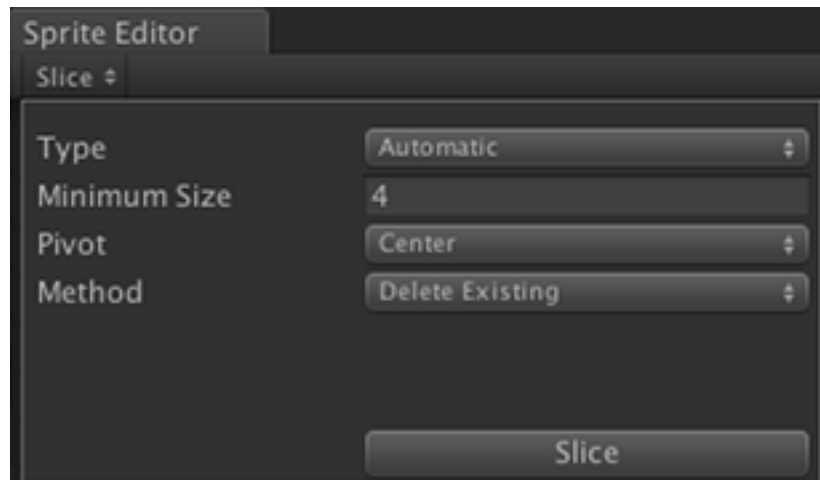


Unity can find your sprites automatically, but you can adjust its results.

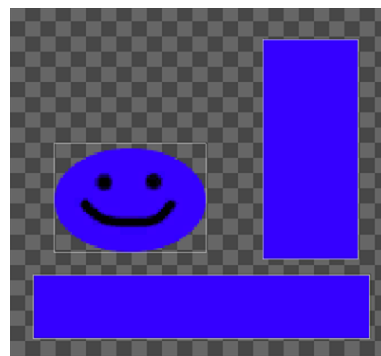
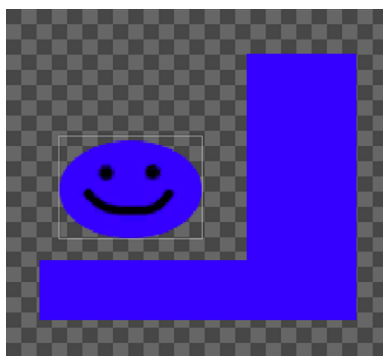
Start with the default settings shown and click **Slice**.

Unity uses the transparency in the texture to identify possible sprites and displays a bounding box around each one.

In this case, it found the following four sprites:



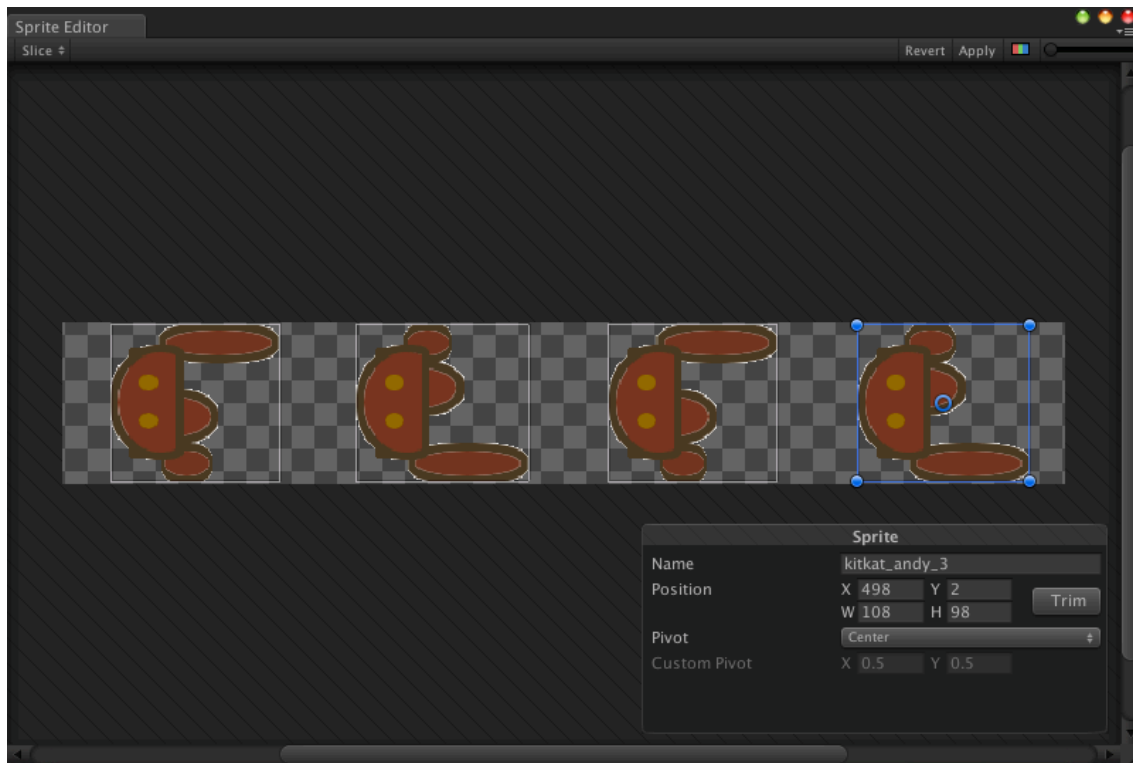
Unity's automatic slicing works best when images are laid out with unambiguous empty space between each item. Notice how Unity only finds the smiley face in the left image, but finds three sprites in the image on the right.



The above images point out that you should arrange the images in your sprite sheets carefully.



Click on any of the sprites that Unity identified to edit the details of that sprite, including its name, position, bounds, and pivot point.



You can make changes in the window's fields, and you can adjust the bounds and pivot point directly within the image.

Normally, after you've made changes, you would hit **Apply** or **Revert** in the upper right of the **Sprite Editor** to save or discard them, respectively.

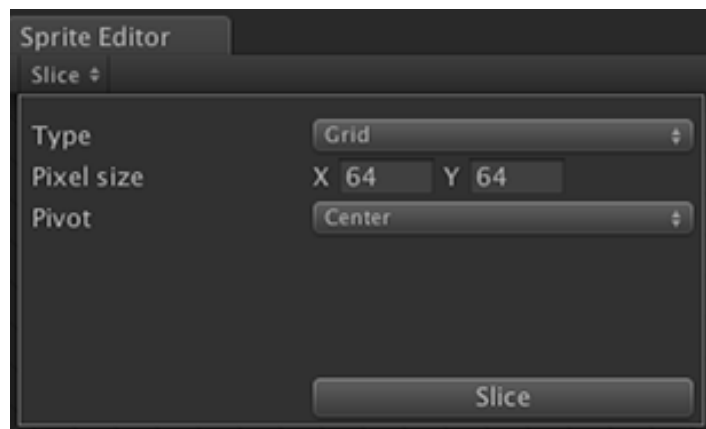
However, while the option to tweak Unity's findings is great, we won't need to do that here because we aren't going to use the sprites it found. The images in *kitkat\_andy.png* are arranged in four equally sized rectangles, and Unity has a separate option to handle cases like this one.

Click **Slice** in the upper left of the **Sprite Editor** to open the slice settings again, but this time, set **Type** to **Grid**.

The splice settings change to those shown here.

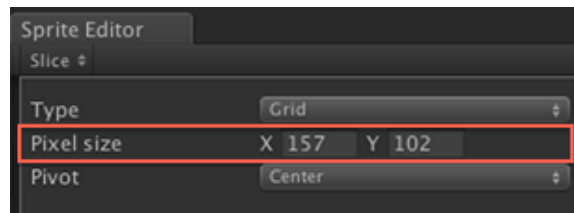
The Pixel size fields allow you to specify the size of your grid's cells.

*X* defines the width of each cell; *Y* defines the height.

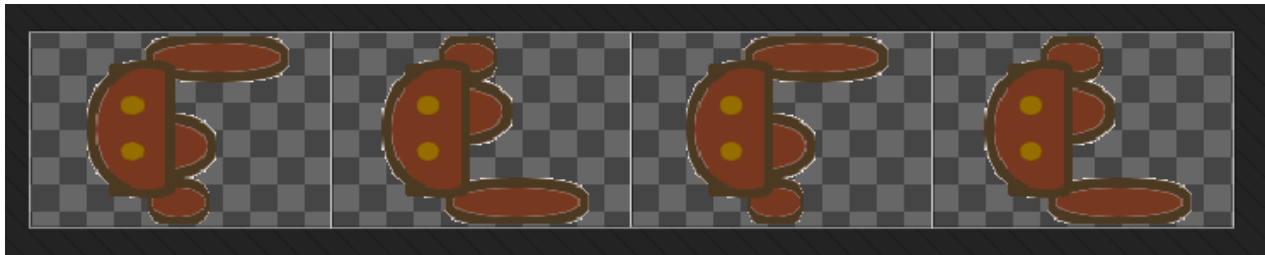


Unity will use those values to divide the image up equally, starting in the upper left corner of the image.

Set *X* to 157, and *Y* to 102, as shown.



Click *Slice* and Unity finds the following four sprites:

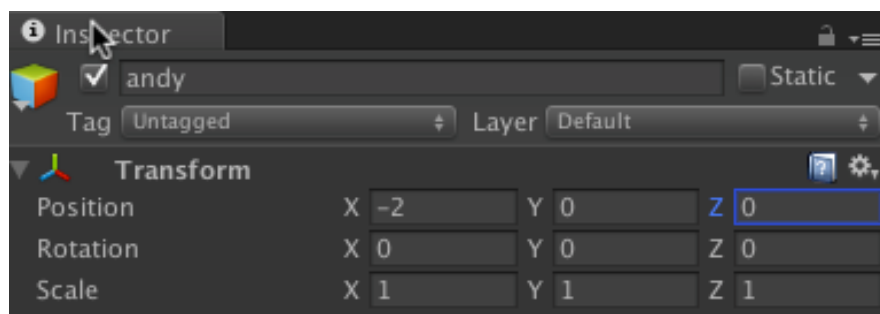


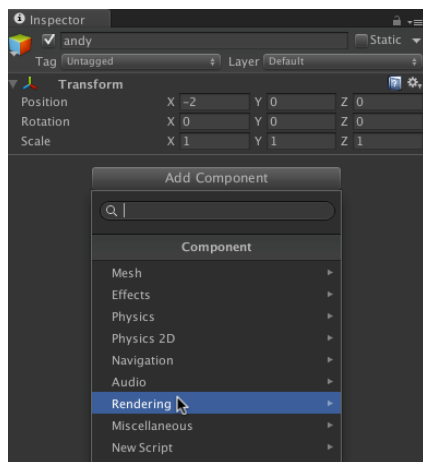
You can still select individual cells in the grid and tweak their settings like you could when using Unity's Automatic slicing option, but that's unnecessary for these sprites.

Click **Apply** in the upper-right of the *Sprite Editor* to commit your changes. Notice how Unity updates the *Project* browser so that the top-level *kitkat\_andy* texture asset now contains four child Sprites, named *kitkat\_andy\_0*, *kitkat\_andy\_1*, and so on, as shown below:



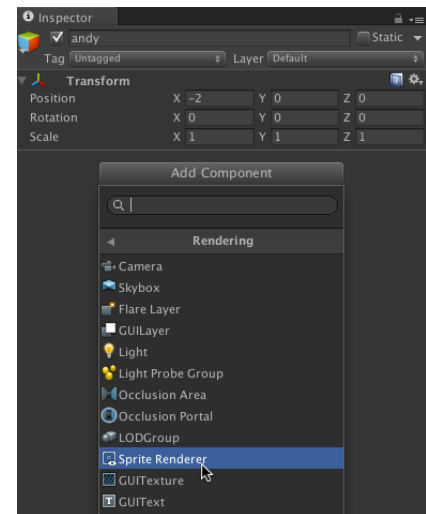
Create a new empty *GameObject* by choosing *GameObject > CreateEmpty*. Click on the *GameObject* in the *Hierarchy* browser and rename the object *andy*, and set its *Position* to (-2, 0, 0) in the *Inspector*.



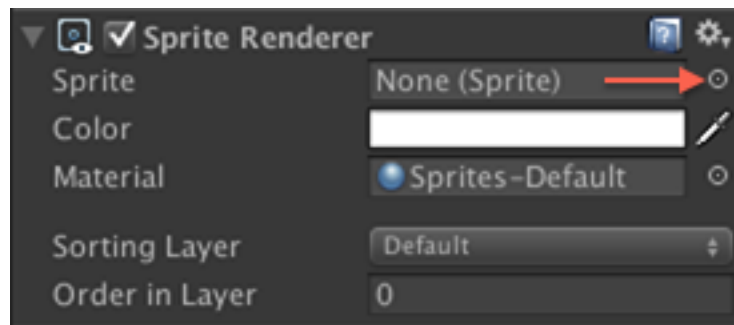


With *andy* selected in the *Hierarchy*, add a *Sprite Renderer* component by clicking *Add Component* in the *Inspector*. See the left image.

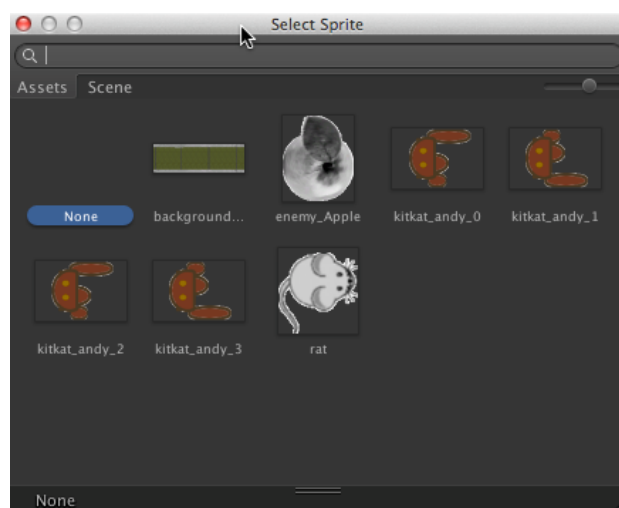
In the menu that appears, choose *Rendering* and then choose *Sprite Renderer*, as shown on the right.



Click the small circle/target icon on the right of the *Sprite Renderer's* *Sprite* field to open the *Select Sprite* dialog. The icon is shown below:



The dialog that appears contains two tabs, *Assets* and *Scene*. These show you all the Sprites you have in your project and in the current scene, respectively.



Choose the *Assets* tab and then click on *kitkat\_andy\_0* to assign that *Sprite* to the renderer.

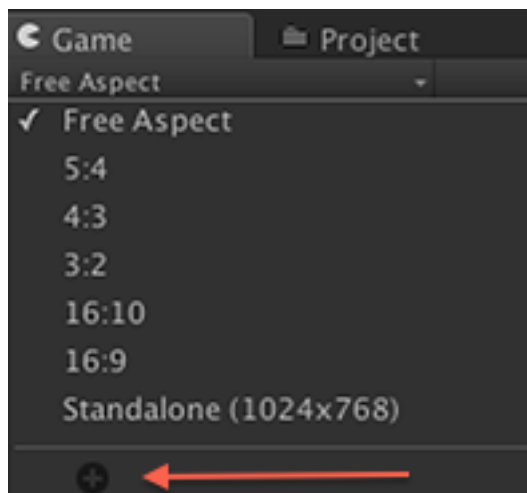
You should see something like this in the **Scene** viewer:



## 6) Setting up the “Look,” etc.

We now have all the Sprites in the game, we can configure how the game should look for the players. First, we are designing this game for an Android system, so we will set our Game view size to 960 × 540 pixels, which is a standard screen size for many Android devices.

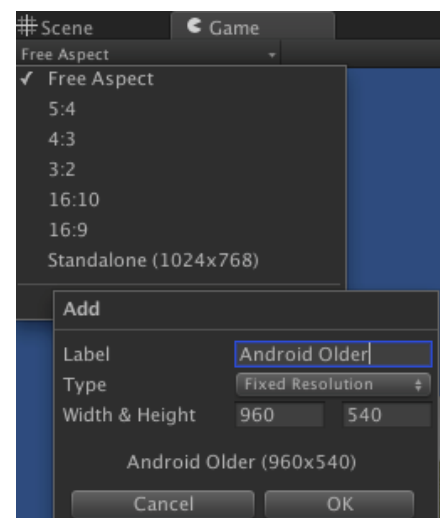
We change the **Game** view’s aspect ratio or fixed resolution by using the drop down menu in the view’s control bar.



Clicking the menu reveals several default options that differ based on the editor’s current player settings. If you happen to have an option for an 960 × 540 resolution, choose it and you’re done. Otherwise, click the + button at the bottom of the menu, as shown.

Create a new size option with a Type of Fixed Resolution, and Width and Height values of 960 and 540, respectively, as shown to the right.

Click OK and then select the new setting in the menu.



The **Game** view now looks something like this:



It may not look exactly like this, because Unity resizes the **Game** view to maintain your chosen aspect ratio within the available space. Regardless of its scale, you should see the same amount of the scene in your view.

Obviously, that isn't quite right. You're seeing the results of three different problems here, and you'll correct each one in turn:

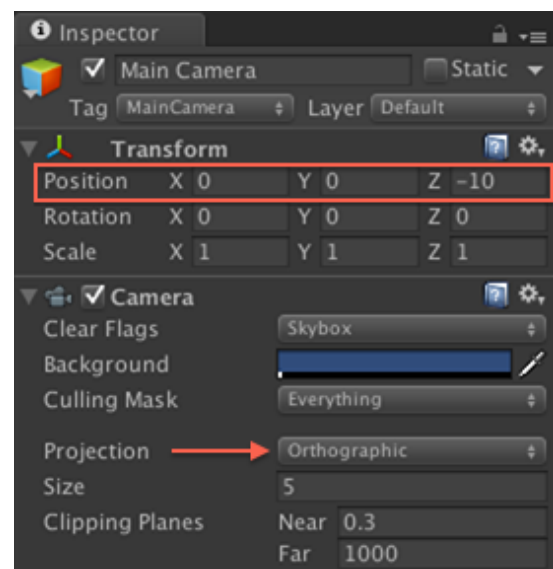
- The scene's camera is not set up properly, so the background doesn't fill the view properly.
- The scene is rendering your game objects in the wrong order, so the rat and Apple are both behind the background.
- The image quality is not very good. This one might be hard to detect with the current camera settings. But trust me, it can be better!

## 7) Fixing the camera.

In 2D games, you'll usually want the camera to use an **orthographic** projection rather than a perspective one. You already read about these two projections earlier regarding the **Scene** view's 2D mode, but what you may not have realized is that Unity may default your game's cameras to use a perspective projection. Not nice!

Select **Main Camera** in the **Hierarchy**. Then, inside its **Camera** component, make sure the **Projection** is set to **Orthographic**.

Center the camera vertically on the scene by setting its **Transform's Position** to (0, 0, -10). Your **Inspector** looks like:

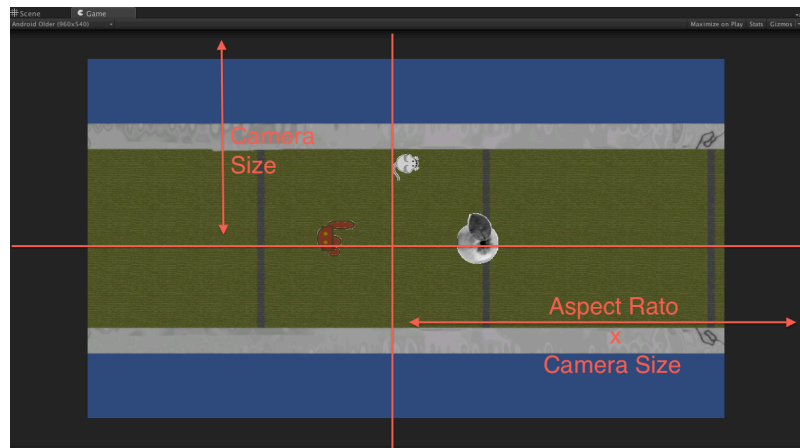


The **Game** view now looks like:



Not much different from the perspective projection! In 3 dimensional games, sprites change size based on their distance from the camera. In 2D games, how do you zoom in so that the background fills the screen? You could try scaling your GameObjects, but using the camera's **Size** property is the more robust approach.

The camera's **Size** defines the dimensions of its viewport. It's the number of units from the center of the view to the top of it. In other words, the camera's **Size** is half the **height** of the view. The **width** of the view is calculated at run time based on the view's aspect ratio.

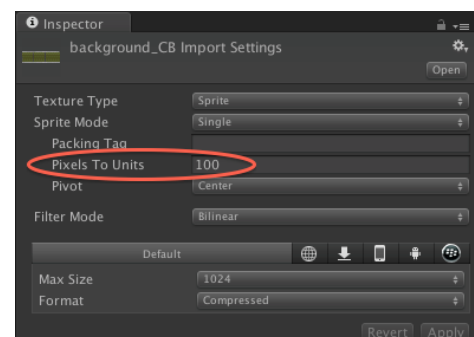


But there's one further complexification: Unity does not measure the Sprites directly in pixels, but in its own **units**. The scale factor determining the number of Unity units for a particular **Sprite** is listed for that **Sprite** in the **Project** browser, under the property **Pixel to Units**.

If we select the **background\_CB** sprite in the **Project** browser, the look in the inspector, we see that its **Pixel to Units** is set to 100. See the image to the right.

So our **background\_CB.png** image was 2048 pixels wide x 640 pixels high, and the **background\_CB** sprite has a **Pixel to Units** of 100, therefore the height in **Unity units** of the **background\_CB** sprite is

$$640 \div 100 = 6.4 \text{ Unity units}$$

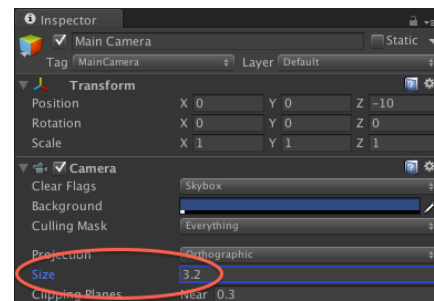




The appropriate camera *Size* is half of that, so we should set it to

$$6.4 \div 2 = 3.2 \text{ Unity units}$$

Now *background\_CB* fills the Game view properly. See the next image.

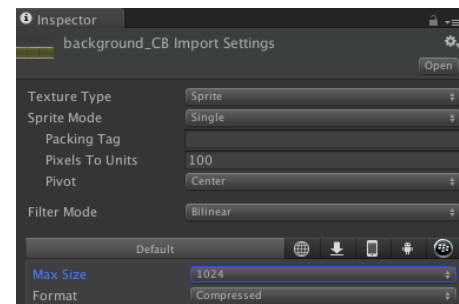


No blue filler area!

## 8) Improving the appearance of the Sprites.

When we imported our Sprites into Unity we simply accepted the default import settings for them. Many times this is a good choice, but it never hurts to review them, especially when you are using Sprite Sheets or larger textures for the background. Sometimes Unity's "one size fits all" approach doesn't work in these cases. **Having inappropriate import settings for your sprite images is the most common cause of a weaker appearance for a game.**

In particular, let's look at the *background\_CB* sprite's import settings. Select *background\_CB* in the *Project* browser. Note that the *Max Size* property is set to 1024 pixels. But *background\_CB* is 2048 pixels wide, so Unity is shrinking it by 50%. Not good for the appearance. **Change this setting to 2048.** Don't forget to click **Apply!**



Also note that the format is set to *Compressed*. Clicking on the drop-down menu shows three settings: *Compressed*, *16-bit* and *TrueColor*. These correspond to 8-bit, 16-bit and 32-bit per pixel formats, respectively. The more bits per pixel, the higher the quality, but also the greater the memory required and the more processing power it takes to render the sprite. So it's a trade-off.



For our purposes, the **background\_CB** sprite has little detail, so the **Compressed** setting is best. For an image with a plethora of detail or great significance in the game, use a less compressed setting.

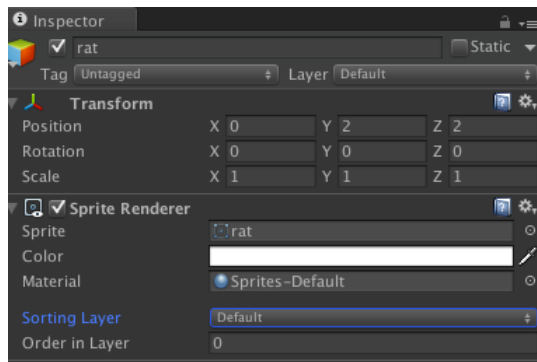
Checking the import settings for the rest of our sprites, we can see that since all of them are small compared to the default **Max Size**, we can leave that alone. None of them have much significant detail, so leave them at **Compressed** for the format.

## 9) Controlling the Draw Order

Unity has an **order** in which it tells the sprites to be drawn in the graphics buffer. This is called the **Draw Order** and it ensures that sprites meant to be in front of other sprites, end up there. We saw that both Starling and Cocos2D have a way of handling this as well. It is a common task for any game engine. Generally, in a 2D game, sprites are stacked in the draw order as they are introduced or given a z-index as they are added to the game play. This z-index number provides the Draw Order for many game development environments.

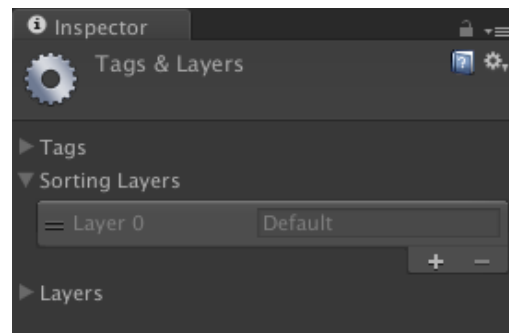
Select the **rat** in the **Hierarchy** and in the **Inspector**, set its **Position** property z-coordinate to 2. It disappears behind the **background\_CB** sprite. Select **enemy\_Apple** in the **Hierarchy** and set its z-coordinate to 3. It also disappears behind the **background\_CB** sprite. So adjusting the z-coordinate would provide a simple way of controlling the draw order.

Although we can use this approach in Unity, it also has a unique feature called a **Sorting Layer** which provides some additional flexibility for managing the draw order.

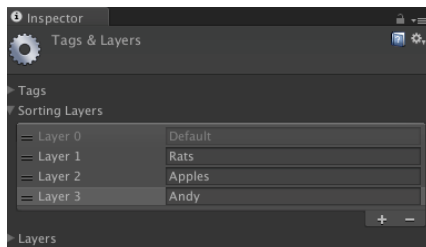


Select **rat** in the **Hierarchy**. Notice its **Sprite Renderer's Sorting Layer** value is set to **Default**. Click the **Sorting Layer** drop down and you'll see a list of all the sorting layers defined in your project, which right now is only **Default**.

You'll also see an option called **Add Sorting Layer....** Click it.



This brings up the **Tags & Layers** editor that you can get to from various other places in Unity, but with the **Sorting Layers** group open while the **Tags** and **Layers** tabs remain closed. See the image on the right.



Click **+** in the **Sorting Layers** group to create a new sorting layer and name it **Rats**. Do that two more times to create a sorting layer named **Apples** and one named **Andy**. Your editor should now look like the image on the left.

These layers define the draw order - Layer 0, named **Default**, is the furthest in the back, with Layer 1, named **Rats**, in front of it, and so on.

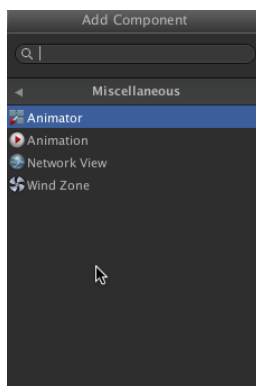
Right now, each of the **GameObjects** you've added is using the **Default Sorting Layer**. For the background sprite, that's fine because you want it in the back anyway, but you need to change the **Sorting Layer** for the other sprites.

Select **rat** in the Hierarchy and set its **Sorting Layer** to **Rats**. You'll immediately notice that the **rat** is now visible in both the **Scene** and **Game** views.

Select **enemy\_Apple** in the **Hierarchy** and set its **Sorting Layer** to **Apples**. Finally, select **andy** in the **Hierarchy** and set its **Sorting Layer** to **Andy** to ensure your player renders on top of all the other sprites. Your **Game** view now looks like this:



## 10) Animating the Sprites



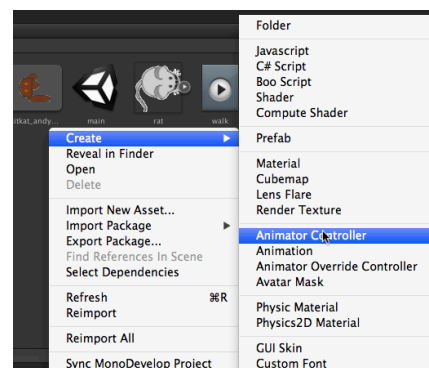
Unity has a built-in **Animation** time-line window that is new in the 2D subsystem. This makes Unity a bit more like Flash. It allows us to automatically animate our Sprites without having to write a bit of code, like we can in Flash.

Select **andy** in the **Hierarchy**. In the **Inspector**, click **Add Component** and in the **Miscellaneous** section, select **Animator**. An **Animator** component will be added to your Sprite.

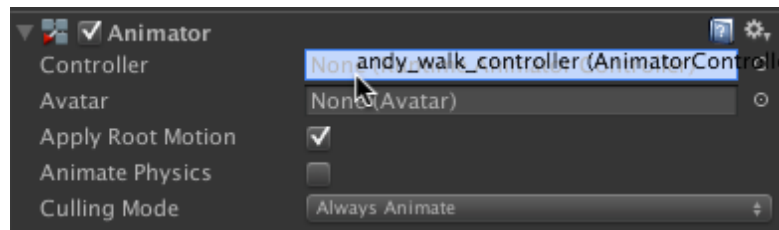
Next in the **Project** browser, right-click to show the context-sensitive menu.



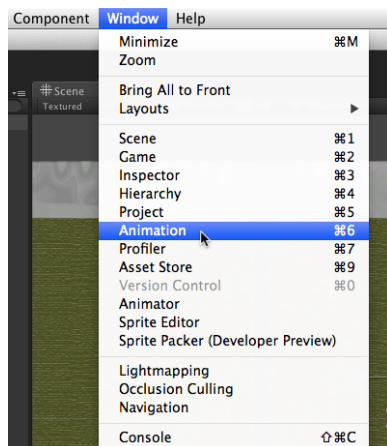
Select **Animation Controller**. An new icon appears in the **Project** browser with the name, **New Animation Controller**.



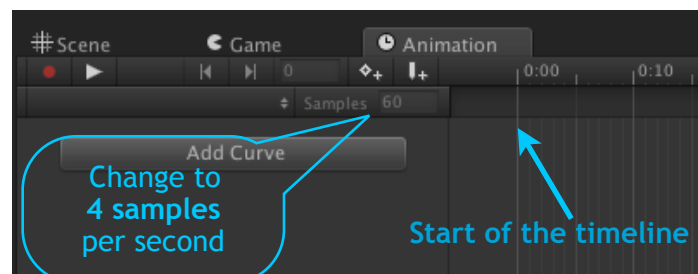
Change its name to **andy\_walk\_controller**. Select **andy** in the **Hierarchy** browser. In the Inspector, look in the **Animator** component for the **Controller** section. Drag the **andy\_walk\_controller** to the **Controller** section and drop it in the box that has the text **None (Runtime Animator Controller)**.



Uncheck the **Apply Root Motion** (this doesn't apply to our type of Sprites). If we were using gravity or other physical forces in our game, we would check the **Animate Physics** box.

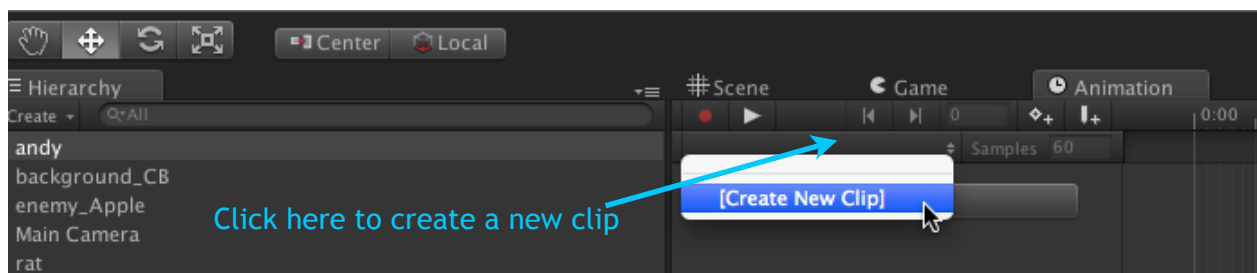


Next, open the **Animation** window as in the image to the left. It will open in a small floating window. Grab it by the tab and dock it to the **Scene/Game** viewer like so:



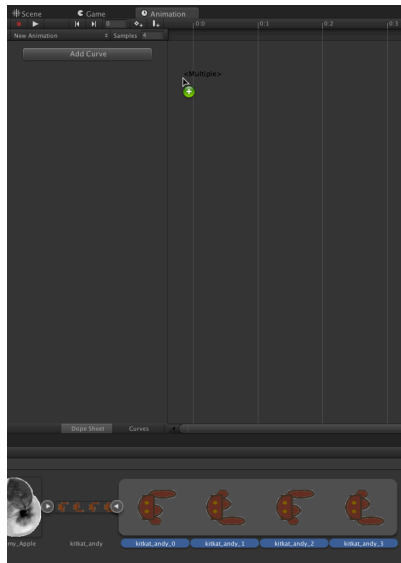
You can see the timeline in the **Animation** window. It is divided into samples which correspond to frames in the Sprite animation.

Now select **andy** in the **Hierarchy** browser. It is **crucial** that the appropriate **Game Object** be selected in the **Hierarchy** before we do the next step. Otherwise no animation will be attached to the **Game Object** (in this case **andy**). Click to create a new clip as in the image:



A new window appears titled **Create New Animation**. Call it **andy\_walks** and click **Save**. It will be saved in the Assets folder of our project folder by default. If we were going to have many animations we should create a new folder inside of the Assets folder and save them all in it for the sake of being organized.

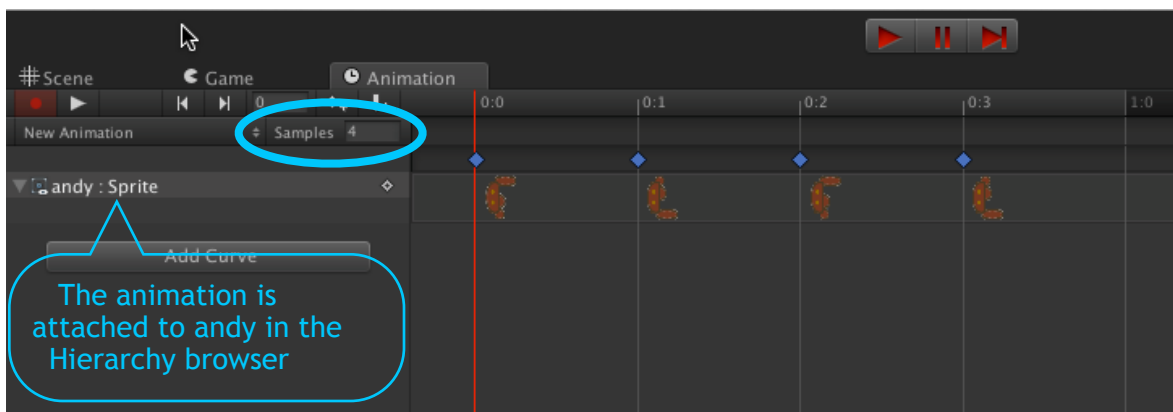
In the Project browser, find the **kitkat\_andy** asset and open it so you can see all the Sprite images inside it. Select them all by using shift-click. You should have them all selected as in the image to the right:



Now drag all the selected Sprite images up to the start of the timeline in the **Animation** window as in the image to the left.

The animation default sample rate is the game play rate of 60 samples per second. This is far too fast for most sprites in a 2D game, so change it to **4 samples per second**.

We should end up with an Animation window that looks like this:



Switch to the **Game** tab in the viewer and click on the red play button. We should now see **andy** going through a walk cycle in the **Game** viewer tab.

We did all this without a line of C# or Javascript!

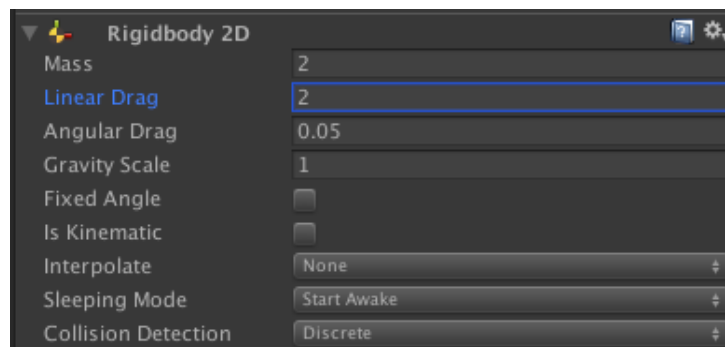
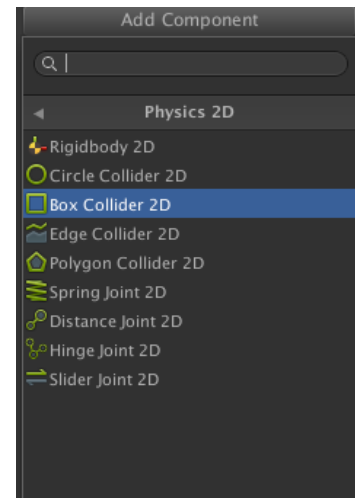
## 11) Setting up and using colliders

Unity has a new system of 2D colliders for the new 2D sprites. Select **andy** in the **Hierarchy** browser, then in the **Inspector**, click **Add Component > Physics 2D**. You will see all the possible colliders that Unity has for 2D sprites.

Choose a Box Collider for **andy**.

Select the **enemy\_Apple** in the **Hierarchy** browser. Add a **Circle Collider** to it. Do the same for the **rat** in the **Hierarchy** browser. Now all our Game Objects have colliders and will work with the Unity 2D physics engine.

One last thing in this section. Select **andy** in the **Hierarchy** browser again. in the **Inspector**, click **Add Component > Physics 2D > Rigidbody 2D**. This places a Rigidbody 2D component on the sprite so that it can interact with the colliders with sophisticated physics settings. Look at the **Rigidbody 2D** component for **andy** in the Inspector. Set the **Mass** to 2 and **Linear Drag** to 2. Leave everything else as is.



## 12) Creating the Prefabs for the game

We need to be able to “spawn” **rat** and **enemy\_Apple** sprites through code as we need them. In order to do that we need to turn the Game Object sprites **rat** and **enemy\_Apple** in the **Hierarchy** into what Unity calls **Prefabs**. A **Prefab** is a **Game Object** that is pre-loaded with all the components it needs to run in the game, but is kept in the **Project** browser until it is needed. Then it will be created in the **Hierarchy** by a game script.

First right click in the **Project** browser and select **Create > Folder**. Name the folder **Prefabs**.

From the **Hierarchy** browser, drag the **enemy\_Apple** and the **rat** into the **Prefabs** folder. Check to see if they are in the folder, and if they have been successful placed into the **Prefabs** folder, delete the **enemy\_Apple** and the **rat** from the **Hierarchy** browser (right-click on each one in the **Hierarchy** browser, then choose **Delete**).

These **Prefabs** should not be confused with the images of the **rat** and **enemy\_Apple** in the **images** folder. The **Prefabs** are **Game Objects** pre-loaded with all the components for the game, whereas the images are only Unity’s stored image.

## 12) Coding the C# scripts for the game

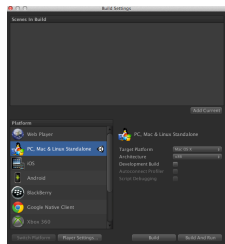
We are now in a position to write the scripts for the game. However, writing detailed instructions on coding C# would take another 20 pages, and is really beyond the scope of this introductory tutorial. All of the scripts are included in the completed version of the game (which is part of the zip archive which includes this tutorial). They have extensive comments in them explaining the sections of the code. *If you would like to gain a more complete understanding of the mechanics of the game's code, please examine the comments in those scripts.*

## 13) Adding Scenes to the game

As in the Starling and Cocos2D tutorials, we are using the notion of a finite state machine to guide the development of this game. With that in mind, we need to add two more states to the game, a GameStart scene and a GameOver scene. This will give us a complete finite state machine of three states: **Start**, **Main**, and **Over**.

To create new scene in Unity, simply select **File > New Scene**. It may then be fashioned into either the **Start** or **Over** state. The finished game has the three scenes for our finite state machine already built. Take a look at them to see how they were done.

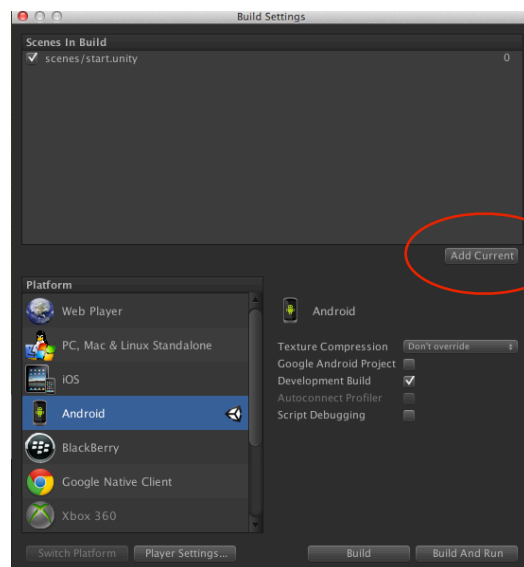
## 14) Building the App for Android



The last step is to build the app for an Android device. We do this in the **Build Settings Panel**.

Go **File > Build Settings ...** to open this panel in a floating window. First, select **Android** from the **Platform** list in the lower left. Next, since we are just trying out the game, click the **Development Build** check box. Leave everything else as is for now.

We now need to add the scenes we wish to include in the build into the upper **Scenes in Build** area of the panel. In the completed version of the game, we have three scenes we wish to include: **Start**, **Main**, and **Over**.

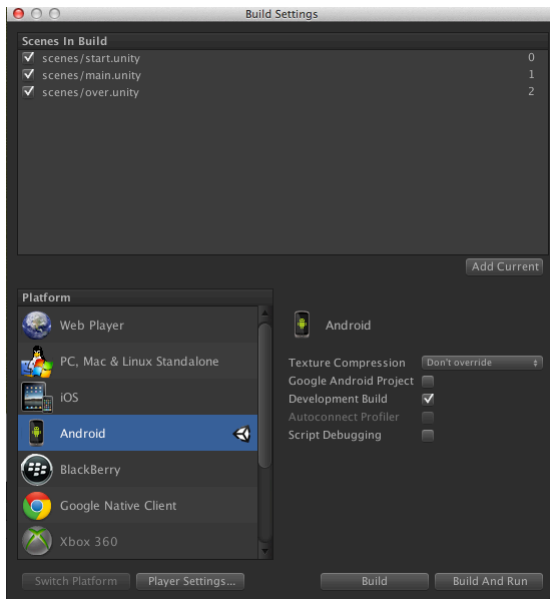


In the **Project** browser find the **Start** scene (that is our first state). Double-click it to open it up in the **Scene** window. In the **Build Settings ...** panel, click the **Add Current** button. This will add the **Start** scene to our planned build.

If the **Build Settings ...** panel has disappeared when you opened the **Start** scene, just go **File > Build Settings ...** to reopen it.

Note the small number 0 (zero) in the upper right of the **Scenes in Build** area of the panel. That equates to the load number of the scene. Scenes with lower load numbers load before scenes with higher load numbers.





Now open the **Main** scene and load it using the **Add Current** button in the **Build Settings ...** panel. Finally, open the **Over** scene and load it using the **Add Current** button in the **Build Settings ...** panel. You should see something like the image to the left.

Next we need to make sure our **Unity Player** settings are correct. The **Unity Player** is the engine that Unity attaches to your own code and assets to make your game work. We need to set it correctly so that it will accurately play the game on the platform we have chosen.

Click on the **Player Settings** button at the bottom of the panel. The **PlayerSettings** should appear in the **Inspector**.

The **Icon** section should open by default. Click the **Select** text for each icon size and load the **android-kitkat-full** image for each of the icons. Unity will take care of generating all the required Android icons as long as you set this up correctly.

Next select the **Other Settings**. Set the **Bundle Identifier** to a reverse URL identifier like so:

***com.mycompany.product\_name***

Set the **Bundle Version** to the version number you've need. In our case leave it at 1.0. Set the **Minimum API Level** to the appropriate level for the app. Generally the lower you can set it the better, since then it will work on a broader range of Android devices. However the more complex a game becomes, and the more it relies on advanced graphics features like particle effects, the less likely it will run on a low end device. Leave everything else at the default settings.

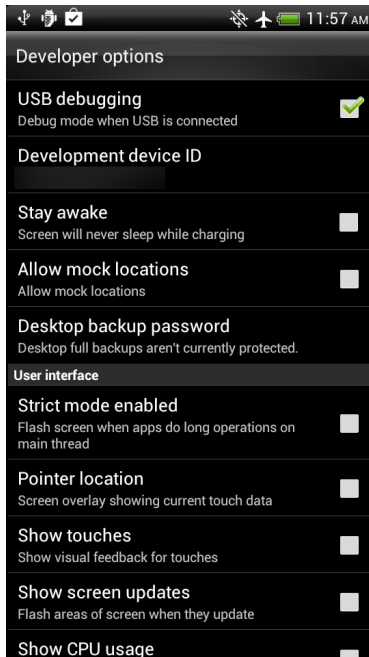
Now click the **Publishing Settings**. This where you set up the keystore which Google Play requires to distribute a game. The keystore locks the app cryptographically which secures it against others hacking the app. The Android documentation states:

The Android system requires that all installed applications be digitally signed with a certificate whose private key is held by the application's developer. The Android system uses the certificate as a means of identifying the author of an application and establishing trust relationships between applications. The certificate is not used to control which applications the user can install. The certificate does not need to be signed by a certificate authority: it is perfectly allowable, and typical, for Android applications to use self-signed certificates. <http://developer.android.com/tools/publishing/app-signing.html>

If this is the first time building this app, check the **Create New Keystore** checkbox. Otherwise click **Browse Keystore** to find the keystore you already created. In the **Keystore Password** text box, type a password you wish to use, then confirm the same password in the **Confirm Password** text box.

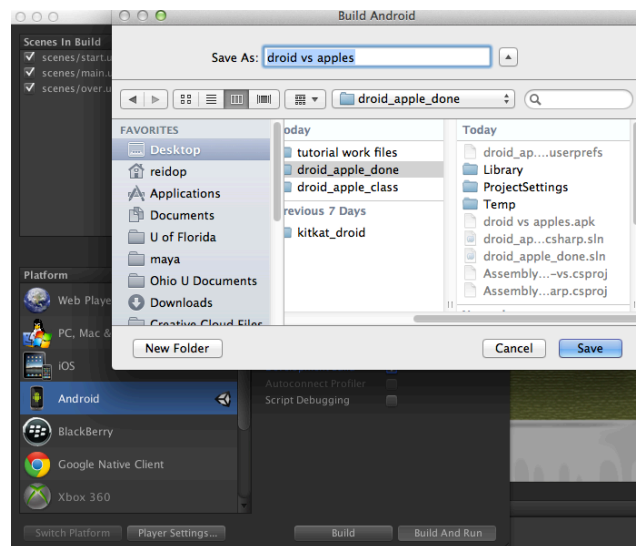
Just below the password text boxes is an area to specify the Key alias for our app. Since we are doing a Development Build, we can leave the **Key alias** at **unsigned**.





We can now **Build and Run** our finished game.

On your Android device make sure the **Settings > Developer Options > USB Debugging** option is selected. See the image to the left for an example. Your settings may look different, but will have the same Developer option for USB Debugging.



Connect the Android device to your computer. Go back to the **Build Settings ...** panel. Click **Build and Run**.

Enter a name for the saved game APK file.

On my set-up it takes about 45 to 60 seconds to complete the build, push the APK file to the Android device and start it running.

This completes the Unity 2D tutorial.

