

2

Importing and Configuring Props

Before we start creating new, wonderful and exciting worlds, we need to import all our assets and get them ready to go. This chapter assumes that you have a basic understanding of modeling and texturing, and that you are also familiar with common terms such as UVs, normal maps, alpha channel, and so on. It doesn't matter which apps you are using for asset creation as long as they support exporting common file types. That is why you won't find any step-by-step export processes described for every single program, but only general recommendations and which things to avoid that are found applicable, regardless of you app choice.

In this chapter, we will look into the following topics:

- Object manipulation
- Working with components
- Importing props into Unity
- Configuring meshes
- Configuring textures
- Learning how to use Unity materials
- Setting up LODs for various objects
- Basics of collision

By the end of this chapter, you should be able to import all your props and prepare them to be used in the level.

Object manipulation

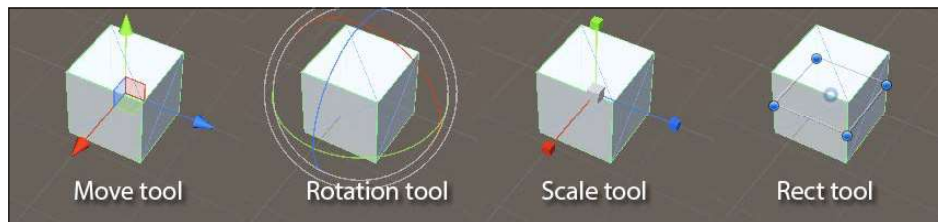
Learning to use the following tools will allow you to place objects in a **Scene** window with ease.

To test out the object manipulation tools, make sure you create one of the available 3D primitives in Unity via **GameObject | 3D Object** (**Cube** is recommended).



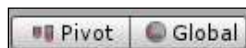
Object manipulation is done via five tools available on the left side of the toolbar:

- **Hand tool** (used for panning), **Move tool**, **Rotation tool**, **Scale tool**, and **Rect tool** can be accessed via the toolbar or by using the hotkeys *Q, W, E, R, T* respectively
- To duplicate an object, you can use *Ctrl+D* or a combination of *Ctrl+C* and *Ctrl+V* to copy and paste
- You can undo the last action via the *Ctrl+Z* combination



Rect tool is a recent addition that made an appearance in the later versions of Unity 4. The circle in the center of the object acts as the **Move tool** and a quad created with four dots on the outside works similar to the **Scale tool**. Dot location will change position based on the camera angle.

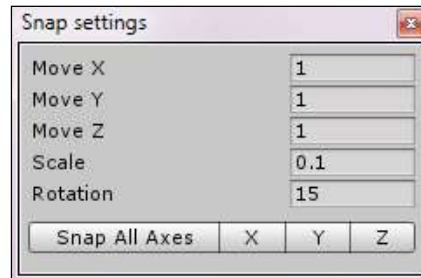
In addition, the toolbar contains tools that allow changing the pivot point of the object to its center or default pivot (useful if you imported the object with specific pivot point placement in mind). Objects can also be rotated, scaled, or moved relative to the **Local** or **Global** space. Settings are defaulted to **Pivot** translation in **Global** space



Snapping

Moving, rotating or scaling objects at equal intervals can be done via snapping. This is done by holding the *Ctrl* button.

To set the snapping intervals, go to **Edit | Snap settings** of the top menu and enter the following settings:

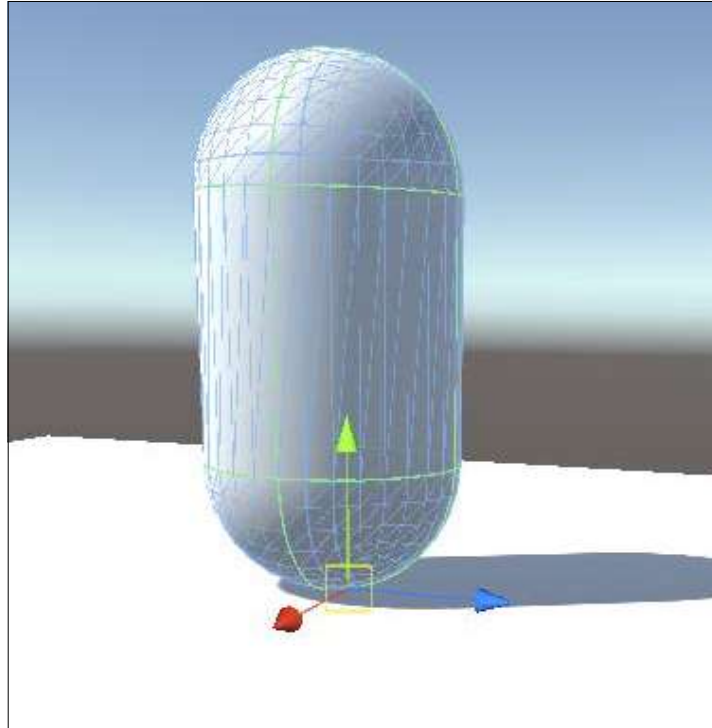


The buttons at the bottom of the **Snap settings** window allow you to round up to get rid of decimal values; it's great for if you wish objects to be perfectly aligned on the grid.

To move objects freely, press and hold *Shift* and use the **Move tool**. To move along the surface with a custom pivot point:

1. Select an object.
2. Hold *V*.
3. Position the pivot point with a mouse.
4. Hold and drag the left mouse button along the surface.

This is perfect for prop placement. As shown in the following screenshot:



Surface snapping may sound like a lot of fun, but in reality, its results may vary based on surface and object topology; it's also very tricky to do in **Perspective** mode.



Remember: Snapping works only in the Scene window.

Greyblocking

Knowing this much about Unity, we can utilize our knowledge of creating primitives by constructing a greyblock of our future level. Using Unity primitives, block out the level to solidify your knowledge of object manipulation.

Components

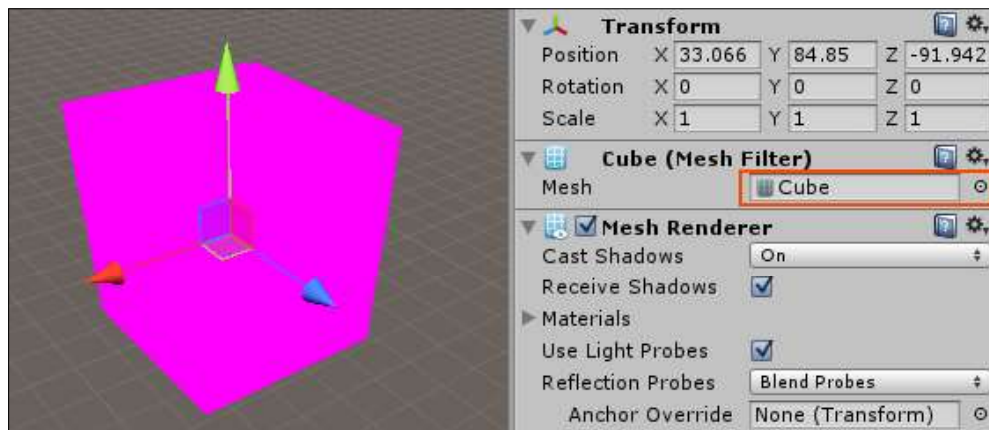
All objects in the level are GameObjects: entities that don't do much by themselves but serve as containers for parameters. These parameters are added with the help of components. If you create an empty GameObject via **GameObject | Create Empty** and look at the **Inspector** window, you will find that it contains one mandatory component called **Transform**, which houses properties mandatory for a 3D space (**Position, Rotation, Scale**). But that is all there is to it; GameObject is just an abstract entity that occupies a location in the Scene. In order to transform it to something more meaningful, more properties, and therefore, more components need to be added.

Adding components

By adding more components, we will add more properties to our GameObjects, so let's do just that and try to reconstruct a primitive:

1. Create an empty GameObject by going to **GameObject | Create Empty** in the top menu.
2. Click **Add Component** in the **Inspector** window or **Component | Add** in the top menu.
3. Go to **Mesh | Mesh Filter**.
4. Assign a **Cube** model reference to a **Mesh** property of a component through the object picker.
5. Add a new component via **Mesh | Mesh Renderer**.

At this point a purple cube should have appeared in place of your empty GameObject.



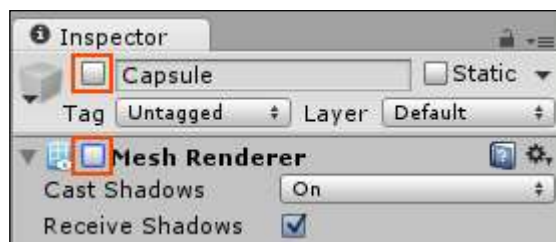
The **Mesh Filter** component allows us to set a mesh to represent the object in the level. **Mesh Renderer** adds rendering properties for this mesh, allowing it to be rendered in the **Scene** and **Game** window.



If you see an object rendered purple, it's most likely because it doesn't have a material selected for it. We will look into **Materials** later in this chapter.

Component deactivation

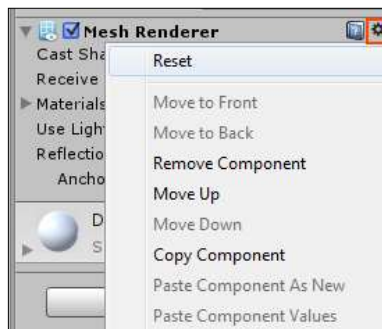
One way to get rid of the GameObjects is by deleting them. However, that's very inefficient to do during runtime, and sometimes you'll find yourself needing to bring the removed object back. The best way to do that is by activating/deactivating the GameObject. To do that, toggle the check box next to the **Name** field at the top of the **Inspector** window. Some components can also be deactivated in a similar manner. That way, deactivation of the **Mesh Renderer** attached to our cube will cause it to disappear in the **Scene** window.



Attached components are dependent on the active status of the GameObject and will be automatically deactivated if the GameObject housing them is deactivated.

Component options

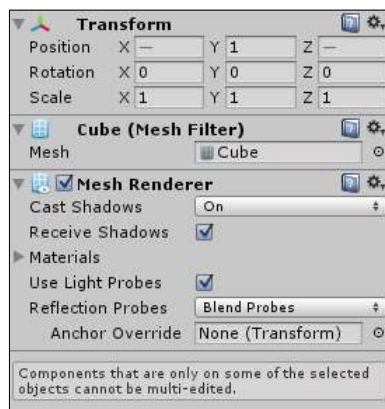
Right-clicking on the component or left clicking on the drop-down menu with a cog icon at the top-right corner of the component will bring up an options menu as seen here:



The most notable options are:

- **Reset:** This returns parameters to their default values
- **Remove Component:** This removes components from the GameObject
- **Copy Component:** This copies the component and its values
- **Paste Component As New:** This adds the copied component with copied values
- **Paste Component Values:** This transfers copied values to another component (only if the component is of the same type)

Objects that share the same components can be edited simultaneously. Selecting multiple objects will result in the **Inspector** window showing only shared components. Parameters will be left as they are if their values are the same for all selected objects, or they will be substituted with a dash if they don't match. Modifying a parameter will change it for all of the objects, as shown in the following screenshot:



There are various components available, and we will look into most of them in future chapters.

Importing props into Unity

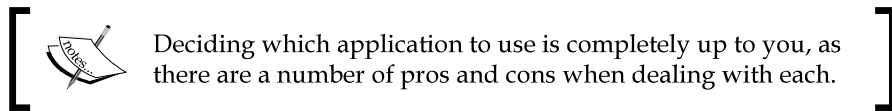
Unity handles the importing of assets quite smoothly, and there aren't that many issues that you will encounter, as long as the actual assets are without issues themselves.

Supported formats

Firstly, we will talk about 3D formats that are recognized by Unity.

Unity supports a number of common 3D file formats, namely: `.FBX`, `.dae`, `.dxf`, and `.obj`.

Among 3D application files, Unity supports 3Ds Max, Maya, Blender, Cinema4D, Modo, Lightwave, Cheetah3D, and many more (through conversion). Chances are that your preferred modeling tool is among them.



When exporting into common 3D file formats (`.obj` for example), you will gain the advantage of:

- Having smaller file sizes.
- Being able to export from 3D apps that aren't supported by Unity.
- Being able to handpick the data you want to export.

However, there are few cons to that approach:

- They are harder to iterate.
- It is easy to lose track of versions between source and game data.

When using file formats native to your 3D app, you will benefit from the following:

- You will be able to iterate on the assets quickly (editing the imported asset will cause Unity to reimport it upon committing changes and returning it to the Editor).
- It's simple (there are no prerequisites to it, just drag and drop the file you were working on and open it from Unity for quick modification).

But there are few things that you will have to keep in mind:

- A licensed copy of a 3D app is required to open them.
- Along with assets, you may import unnecessary data.
- Files are generally bigger and may slow down Unity.
- Less validation will make it harder to troubleshoot errors.



Exporting in a 3D app file format is justified during the prototyping, when you are constantly iterating, and all members of your team have a licensed version of the app installed.



Exporting from Blender

As mentioned before, Unity supports import from popular 3D apps, and Blender is no exception. There is one thing that you need to be wary of when importing a Blender file into Unity; your Blender version has to be 2.45 or higher. The reason for this is that Unity uses the Blender FBX exporter added to Blender in version 2.45. If you are using an older version of Blender, you need to manually convert your files to a common 3D file format before importing.



This is not just the case with Blender. If your 3D app doesn't have a built-in exporter, Unity will not be able to read the native file format. That is also the reason why you need to have that 3D software installed on a computer in order to access imported assets.

Object setup before exporting

In terms of the mesh, there aren't any Unity specific requirements that need to be met to import successfully.

If the asset that you are importing has multiple components and/or groups, make sure to set them up properly in a 3D app, as it will have the same hierarchy when imported into Unity.



You cannot modify an asset or its hierarchy directly in Unity. Say you've imported a .blend file into Unity and wish to edit it:

1. Double-click on the imported file in the **Project** window.
2. If you have Blender installed on your computer, Unity will opt to run it.
3. Edit the file and save it.
4. Return to the editor.

Once you've returned to the Editor, you might experience a small lag; that's Unity reimporting the file you've just edited.

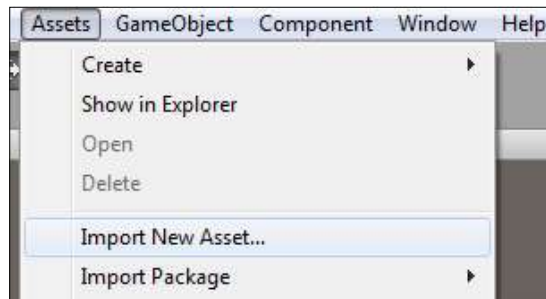
Make sure that there is a pivot point where you want it to be. Remember, in Unity, the default is a center pivot point, but you can switch it to the one set in the 3D app, by going into Pivot mode.

Make sure that you remove construction history – Nurbs, Norms, and Subdiv surfaces must be converted into polygons. During the final export, get rid of your scene lighting; it will not be exported.

There are a few tips and tricks that you can apply in order to optimize your models, however, we will touch upon them later, after covering all related material.

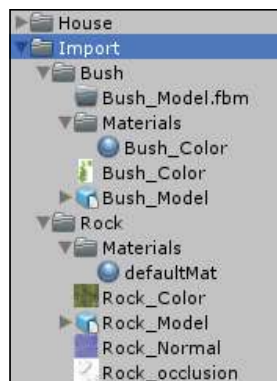
The importing process

The easiest way to get assets, is simply by dragging and dropping the file into the **Project** window. It will create a copy of the file within the Unity directory. Alternatively, you can go to **Assets | Import New Asset...** in the top menu and import it that way.



Files will be automatically converted, so you don't have to change anything else. Your asset is now in Unity and is ready to be used. To start things off, we need to do the following:

1. Open the **Chapter 2** folder of complimentary files.
2. Drag and drop the **Import** folder into the **Chapter 2** folder in the **Project** tab of Unity.



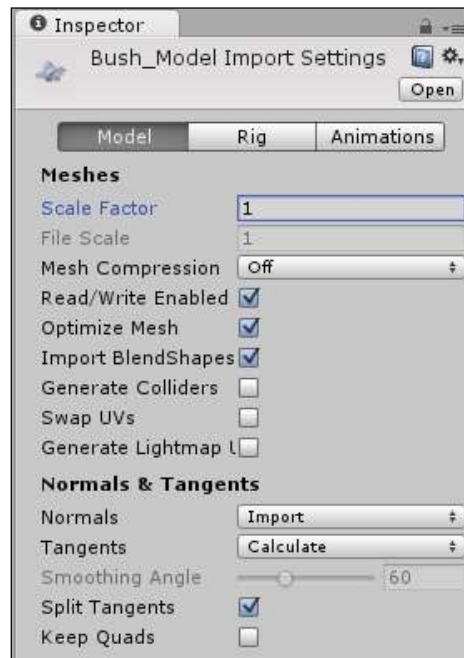
Congratulations! You've just imported OBJ and FBX models as well as PNG textures into Unity.

Configuring meshes

Now that we have our assets in, it's about time we started configuring them in Unity.


Model-meshes options

Selecting asset in **Project** window will display **Import settings** in the **Inspector** window.



Import settings have three tabs available corresponding to **Model**, **Rig**, and **Animation** settings of the asset. Right now, we will focus on the **Model** tab and look into available settings to prepare our asset to be used:

- **Scale factor:** This scales an asset in comparison to the original model. This is a great way to adjust a model's scale outside of a 3D modeling app. Using the **Scale Factor** parameter over **Scale tool** is also beneficial as it ensures that the object will correctly respond to physical interaction within Unity by using uniform scaling (developers report fixing this in Unity 5).
- **Mesh Compression:** Unity will attempt to reduce the size of the **Mesh**. **Compression** has four states: **Off**, **Low**, **Medium**, and **High**. To minimize your build size, try to use maximum compression until you see any irregularities appearing on the model.

 In case you are wondering how compression works, the numerical accuracy of the mesh is reduced: instead of 32-bit floats, mesh data will be represented with a fixed number.

- **Read/Write Enabled:** This enables the meshes to be written at runtime. Enabling this option will allow mesh modifications through code. This is another optimization option that allows saving memory by turning it off for all meshes that aren't intended to be scaled or instantiated at runtime with a non-uniform scale.
- **Optimize Mesh:** This optimizes the order in which triangles will be listed in the mesh. Check it if you wish to trade loading time for a better runtime performance and vice versa.
- **Import BlendShapes:** Unity will import blend shapes for the model if this is enabled.
- **Generate Colliders:** This automatically generates a **Mesh Collider** for the model. It is useful for environment geometry. However, it should be avoided for any dynamic objects that will be moving.
- **Swap UVs:** This swaps primary and secondary UV channels.
- **Generate Lightmap UVs:** This creates a secondary UV channel for Lightmapping.
- **Normals & Tangents** are very straightforward options: you either **Import** info on **Normals & Tangents** from the source file, allow Unity to **Calculate** them for you (**Normals** are calculated based on the **Smoothing Angle** slider that sets how sharp the edge has to be in order to be treated as a Hard Edge) or choose **None** to disable them.

Make sure to use the **Calculate** option on **Normals** if you need a quick fix for a model. Unity does a pretty good job of it.

 The import option on **Tangents** is available only for FBX, Maya, and 3Ds Max files.

- **Split Tangents:** Enable this if Normal Map lighting is broken by seams on your mesh.
- **Keep Quads:** This preserves quads on model topology for DirectX 11 tessellation.

Double-sided normals

There is one particular problem that you might run into when importing your models and try to view them in Unity. The problem lies in the fact that Unity, by default, doesn't support double-sided normals, making one side invisible. This is quite a regular problem, and there are a couple of ways to approach it. Have a programmer write a double-sided shader, use duplicate faces or reverse normals on your model in the 3D app. The latter is a lot easier, and a quicker way to solve this problem.



Labels

Now to the most important part of handling assets: **Labels**.

Labels are used for filtering and search queries. You can assign as many labels as you want to any asset, imported or internal. To do that:

1. Select the asset in the **Project** window.
2. Open the **Inspector** window.
3. Click on the **Label** button in the bottom-right corner of the **Preview** window.
4. Type in the label name.
5. Press *Enter* to create a new label and assign it to the object.



Now you can use the search field of the project window to find the asset with the newly created label as shown in the following screenshot.



That's it. Labels are easy to understand. However, getting used to working with them and figuring out a system that you will use to filter through assets quickly will take some time.

Prefabs

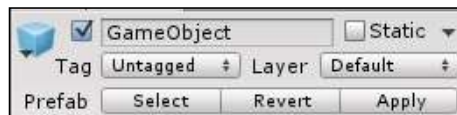
Let's say you've configured a **GameObject** and its components, and you now want to reuse it multiple times throughout the project. Duplication is an option; however, there is a much better way, in the form of prefabs.

Think about prefabs as templates—they are generally used to create instances of certain objects and quickly modify them by applying changes made to the templates.

Creating prefabs is easy:

1. Click on the **Create** button of the **Project** window.
2. Select **Prefab**.
3. Drag and drop a primitive created earlier onto the prefab.

Done! Now you can observe how the prefab works. Notice that our original primitive in the Scene window now gained a new section called **Prefab** at the top of the **Inspector** window with three options available:



- **Select** will select a prefab that this object belongs to in the **Project** window.
- **Revert** will reset all changes done to the object and make it identical to the prefab.
- **Apply** will update the prefab with changes done to the object.

Try creating more instances of the prefab in the Scene, manipulate the prefab and use object transform, add and remove components from the prefab and objects, and see how they affect each other.

You should figure out that all objects are immediately updated whenever the prefab is changed, but a change done to a single object doesn't affect the prefab itself, unless changes are committed via the **Apply** button.



If by any chance you forgot what you've changed in the object, there is no need to compare it to the prefab to find out; all changes will be highlighted in the Inspector window.

If you want to reset a particular component of an object from a prefab, you can do that by right-clicking on the component you wish to reset and select **Revert To Prefab**. This will reset that particular component only and leave other components untouched.

Objects that are connected to their prefabs will also be highlighted in the **Hierarchy** window with a blue color.

One thing to be cautious about is dragging and dropping an object onto a non-empty prefab, which will result in the replacement of all of its instances with the object you've dropped. This cannot be reverted with the *Ctrl+Z* command, but you will receive a warning message from Unity and can still cancel the action; so be wary.



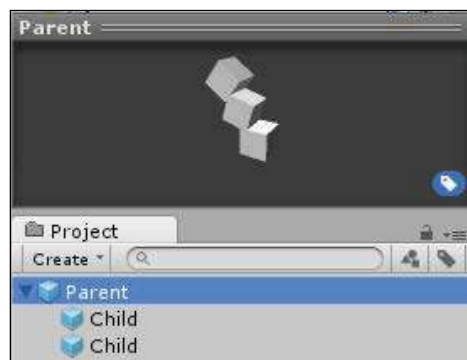
As a rule of thumb, it's a good precaution to set up all the objects to have their own prefabs, even if you aren't planning to mass modify them. There are two major reasons to do this:

1. Meshes cannot be modified, nor do they have components attached to them in the **Project** window. You can only do that by dragging the mesh into the Scene, which will automatically create a GameObject that you can then modify. Or, skip that and create a prefab editable in the **Project** window.
2. Programmers will have a much easier time working with cooked prefabs rather than manually assembling the required GameObject via code.

That being said, you should probably go ahead and create prefabs for the **Bush_Model** and **Rock_Model** assets we've recently imported. We'll need them very soon.

Object parenting

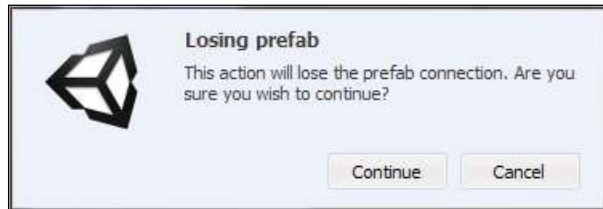
You can group objects by parenting them to other GameObjects, empty or not. In Unity, it can be done by dragging and dropping the GameObjects on top of each other in the Project or Hierarchy window.



There is one thing to be aware of: it is important to understand that the transformation of the child is no longer relative to the global space, but to its parent object. So whenever the parent object moves, it will move all its children with it. This is also the case for scale and rotation.

Parenting and prefabs

When it comes to prefabs, parenting is quite simple: You freely modify and add any children to the parent object in the Hierarchy view and then hit **Apply** for them to be saved to the prefab. However, unparenting a child from a GameObject will cause the parent object to lose connection to the prefab.



If you need to use unparenting you will have to lose connection to the prefab for that object, unparent, and then drag and drop it back onto the prefab. That will commit changes to all other references as well.

The pivot point

Another important thing to know is how the pivot point of the parent object is going to react to the addition of child objects. This is where the **Pivot tool** that we've discussed earlier comes in handy. In the **Center mode**, it is located exactly in the middle of the group. However, switching to the **Pivot mode** will result in the pivot point snapping to the original position, where it was back when the parent object was exported (if it was).

Configuring textures

It's about time we talk about textures and how they can be handled in Unity. We will start by listing supported formats, discuss useful tips to consider during exporting, and take a close look at options available upon importing.

Supported formats

Unity has a wide range of supported formats for textures; they include PSD, TIFF, JPG, TGA, PNG, GIF, BMP, IFF, and PICT. Yes, multilayered files are supported, and you can easily use them without any fear of memory increase or performance loss. That being said, don't expect to be able to take advantage of a multilayered format, because Unity will convert the files and flatten all layers. This conversion is purely internal and your source files will not be altered in any way, allowing you to continue to iterate on them.

Preparing textures for export

There aren't any specific requirements for textures to be imported into Unity. If the image format is supported, drag and drop it into the **Project** window—just like we did earlier with the `Import` folder—and you are good to go.

From a performance standpoint, it is strongly recommended that you use the power of two sized textures (2, 4, 8, 16, 32, 64, 128, 256, 512...). The **non power of two textures (NPOT)** can be used in Unity at the cost of extra memory and a small performance loss. However, if a platform or a GPU doesn't support NPOT textures, Unity will automatically scale them to the nearest power of 2, causing a dramatic loss in performance.

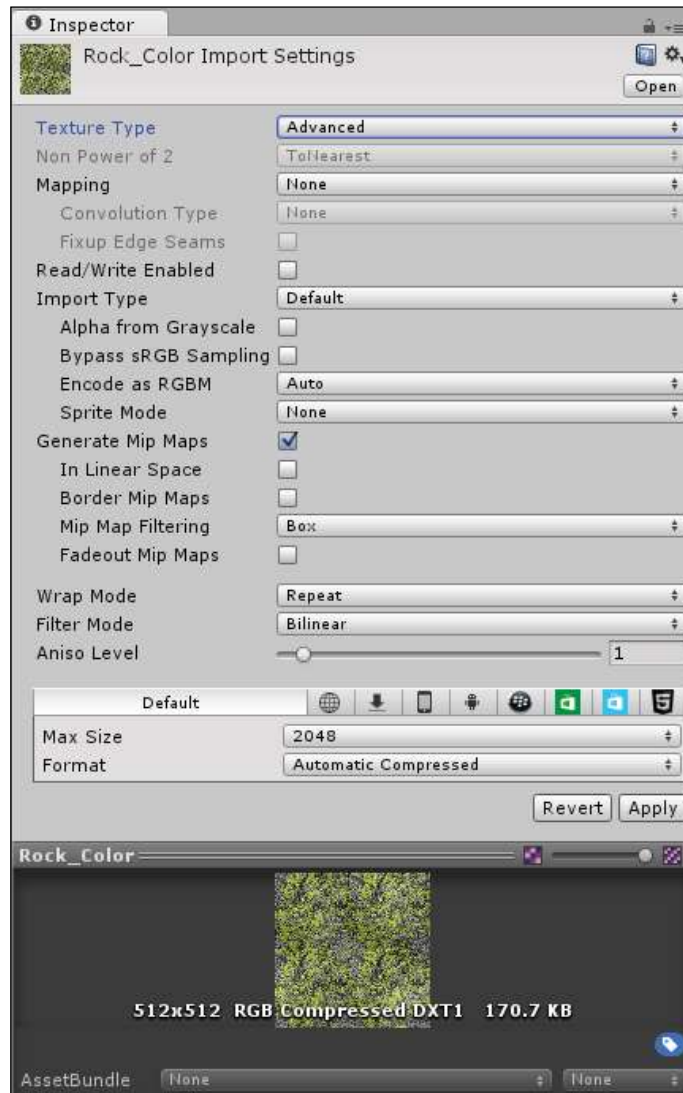


You can still use NPOT textures for something like UI.



Settings for various texture types

By selecting one of the imported textures, you will see import options in the **Inspector** window.



- Most of the settings of a texture are decided based on the value selected in the **Texture Type** drop-down menu. Here are the available options:
 - **Texture**: This is the most common setting for textures (this is your default go-to option).
 - **Normal Map**: This turns color channels into a format suitable for real-time normal mapping.
 - **Editor GUI and Legacy GUI**: This is used for the GUI.
 - **Sprite (2D and UI)**: Select this to use the texture as a sprite in a 2D game and UI.
 - **Cursor**: This is useful for cursor sprites.
 - **Cubemap**: This is used to create Cubemaps.
 - **Cookie**: This is used for light cookies.
 - **Lightmap**: This is used to identify lightmaps.
 - **Advanced**: This reveals all parameters of the texture.

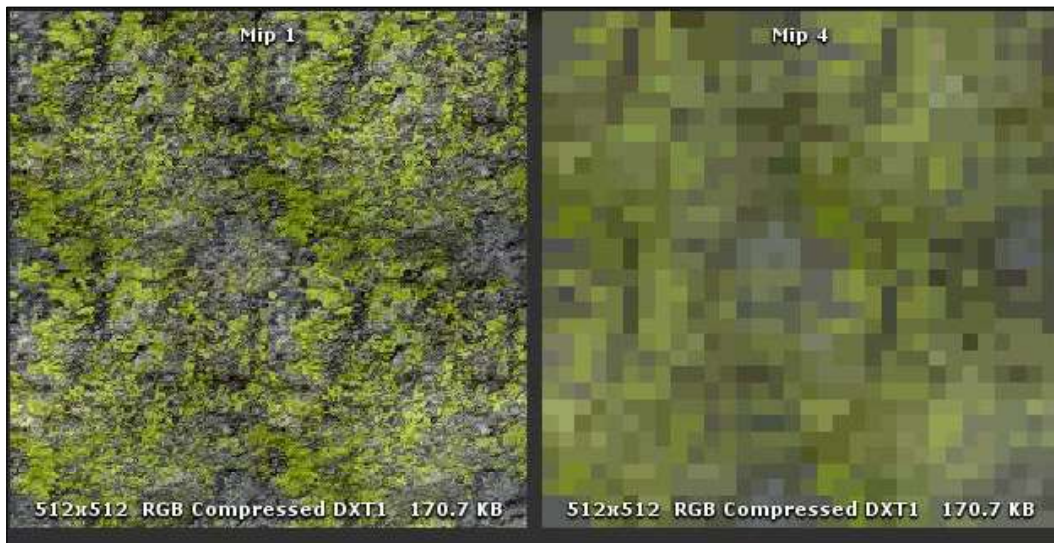
Since the **Advanced** option gives us all the necessary parameters, we will stick to it from now on.

- **Non Power of 2** will define scaling behavior in case our texture has non-power-of-two size. Four options are available here:
 - **None**: Texture won't be scaled
 - **To nearest**: Texture will scale to the nearest power of 2 (130x250 to 128x256)
 - **To larger**: Texture will scale to the next larger power of 2 (130x200 to 256x256)
 - **To smaller**: Texture will scale to the next smaller power of 2 (130x200 to 128x128)

All the changes will be applied upon import internally. After changing the parameter, Unity will re-import the texture.

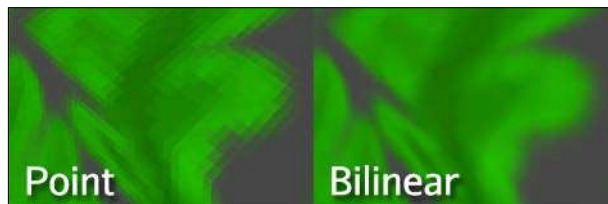
- **Mapping** specifies layout options for custom Cubemaps and enables additional options associated with them. We will omit talking about Cubemaps as they are outside of the introductory scope.
- **Read/Write Enabled** allows coders to get access to texture data. Be careful with this one and keep it disabled by default unless it's absolutely required. This doubles texture memory usage due to the necessity of storing the edited and original version of the texture. It is only valid for uncompressed and DXT compressed textures.

- **Import Type** is a simplified version of a **Texture Type** parameter. It allows the texture's purpose to be interpreted properly and opens several options based on the type selected: **Default**, **Normal Map**, or **Lightmap**.
 - **Alpha from grayscale** is available for the **Default** texture type. It creates an alpha channel from the luminance information in the image.
 - **Create from grayscale** is available for the **Normal Map** texture type. It creates a Normal Map from luminance information in the image.
 - **Bypass RGB sampling** is for the **Default** texture type. It allows the use of direct color values from the image without gamma compensation applied by Unity.
 - **Encode as RGBM** does just that. It is useful for the HDR textures.
 - **Sprite Mode** allows you to configure your sprites as a **Single** image or a sprite sheet (**Multiple**).
- **Generate Mip Maps** allows the creation of smaller versions of the texture to be used when the texture appears small on the screen.

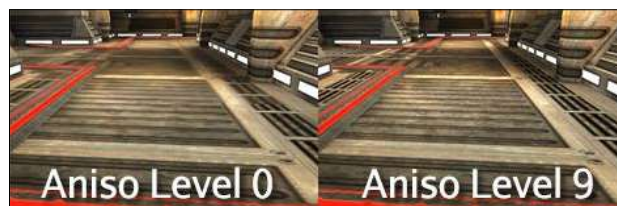


- **In Linear Space** generates mip maps in linear color space.
- **Border Mip Maps** prevents color from seeping out to the edge of the lower **Mip** levels.

- **Mip Map Filtering** is used to optimize mip map quality. The **Box** parameter makes mip levels gradually smoother, while **Kaiser** applies a sharpening algorithm to avoid blurry textures.
- **Fade Out Mip maps** makes mip maps fade to gray with mip level progression. The Fade Range scroller that appears after enabling this option defines the first level to start graying out and the last level when texture completely grays out.
- **Wrap Mode** defines the behavior of a tileable texture: you can choose to either, Repeat which will make texture repeat itself, or Clamp, to stretch edges (used by default for non-tileable textures).
- **Filter Mode** defines filtering options when the texture is stretched by transformation. **Point** will make the texture blocky, **Bilinear** will make it blurry, and **Trilinear** will blur the texture between different mip levels.



- The **Aniso Level** slider is available for Bilinear and Trilinear filters. It improves texture quality when viewed at a steep angle. It is most commonly used for a floor.



Getting your textures into Unity is as easy as it gets, plus you'll have a lot of options to tune them the way you like in the future. The best, and really the only, way to avoid being overwhelmed by the variety of options is to use **Texture Type** to filter them and base your decision on what's left.

Having textures imported is good and all, but it's not enough to use them with your GameObjects. For that we need **Shaders**, but more specifically their holders: **Materials**.

What are Materials?

Objects are rendered in Unity with the help of Shaders: chunks of complex code that can be created in Unity's MonoDevelop. However, there is a much easier way to work with Shaders, and that's through Materials. Materials allow the adjustment of properties and assignment of assets to Shaders without any programming knowledge.

Materials in Unity

Upon importing assets earlier, you probably noticed that Unity also created a folder called Materials in each asset folder. By default, each GameObject should have a Material assigned to it—if it doesn't it will be rendered pink, just like we've witnessed at the beginning of this chapter when tried to recreate primitives with components. During asset import, Unity used the name of the Shader assigned to the assets and gave it to the newly generated Material, which was automatically assigned to imported object; however, it didn't assign textures to both of the materials but only to the **Bush_Color**. There are three reasons why this could happen to any model:

1. Textures were not assigned to the Shader upon exporting. This is true in our case; **Rock** did not have textures assigned to it prior to exporting.
2. The file format was incorrect. That is also true. **Rock** was exported as OBJ and **Bush**, which has a color map assigned to its material was exported as FBX.
3. Unity couldn't find the assigned texture. That is not the case for our models, but it could often be the reason for others. To avoid that, I would recommend keeping models and assigned textures in the same folder upon import (you can always reallocate them afterwards).

Reassigning textures upon import can be a real pain. To avoid that issue, I usually:

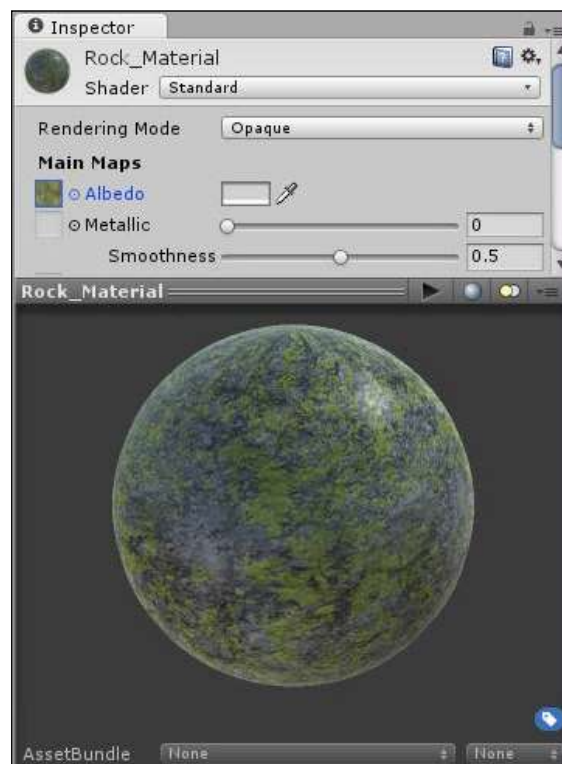
1. Assign textures to the model in the 3D app.
2. Keep assigned textures in the same directory as the mesh file.
3. Export the model in FBX format.
4. Import the model and textures into Unity together.

This is not necessarily the only way to do it—you might not need to assign the textures in a 3D app in the first place for your pipeline, but this is just something I can recommend from personal experience.


In order to assign the missing textures to materials, we need to do the following:

1. Rename imported materials to **Rock_Material** and **Bush_Material** in the **Project** window.
2. Select **Rock_Material** and go to the **Inspector** window.
3. Click on the little circle on the left of the **Albedo** parameter.
4. Select the **Rock_Color** texture with the object picker.

Now all GameObjects that share this material will be automatically updated. You can use the **Preview** section at the bottom of the **Inspector** window to see how your material will look on the object:



This color map will now be automatically assigned to all GameObjects that share this Material.

 You can't assign materials to imported models; however, you can do it for prefabs and objects in the Scene/Hierarchy window.

Creating Materials

Materials are very useful in a way that they are automatically updated on all assigned GameObjects. But at the same time, that is their biggest limitation, as we cannot create any variation this way. In our case, if we want to make some rocks darker and others greener, we'll have to create unique materials.

To create new materials, use the **Create** drop-down menu of the **Project** window.

Newly created material can be assigned to GameObjects with the **Mesh Renderer** component:

1. Select a **GameObject**.
2. Go to the **Mesh Renderer** component of the **Inspector** window.
3. Assign the material to one of the elements of the **Materials** array.



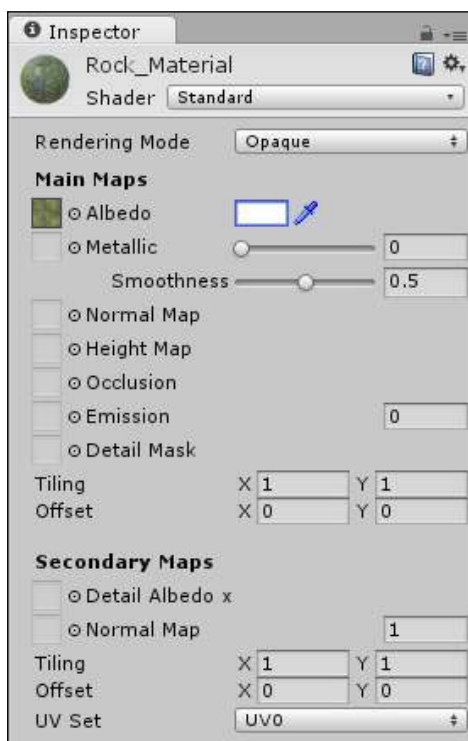
Shader types

The most important part of each material is to select the correct shader to render it. There is an abundance of shaders available for Unity, both built-in and user-created, available in the **Asset Store**. You can even write your own using ShaderLab coding language; however, that will require extensive knowledge of the subject. Thankfully, with the release of Unity 5, most of them were moved to the backlines for backward compatibility with project upgrades and were labeled Legacy. All of them were replaced with a single default Shader dubbed Standard.

Material parameters

Standard is a very powerful and versatile material with a lot of customization options that widen its application range.

Let's take a closer look at the options available for it:



- **Main Maps** is a set of primary textures that are utilized by the Material:
- **Albedo** parameter defines a diffuse color produced from the assigned color map and a surface color defined by the color picker on the right.
- **Metallic** defines how smooth and reflective the surface is. You can import a custom texture or use a slider to control how metallic you want your material to look. This parameter is further enhanced by the **Smoothness** slider.
- **Normal**, **Height**, **Occlusion**, and **Emission** are your standard maps for 3D objects. In this book, we will work with **Normal Map** and **Occlusion Map** for props and characters and we will use the **Height** map to generate terrain in the next chapter.

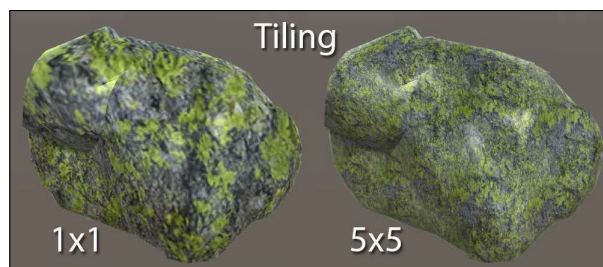
- **Detail Mask** uses the Alpha channel of the imported texture to create a mask for **Secondary Maps**. **Secondary Maps** allow us to create more details on the surface by importing in additional Color and Normal maps. Controlling the areas in which they will overlap is the purpose of the **Detail Mask**.
- **UV Set** allows you to toggle multiple UV sets if they are available for the model.



Here is a general idea of how the Detail Mask works

- **Tiling** allows us to control how many times our texture is going to be repeated across the x and y axes.
- **Offset** slides the texture across the x and y axes.

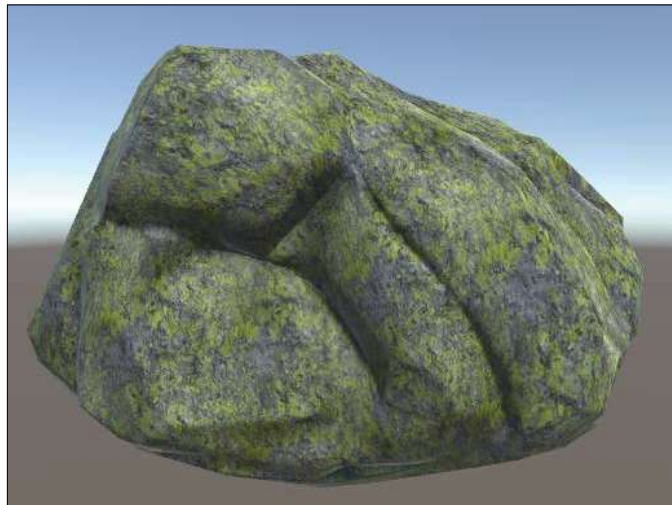
The last two parameters are best suited for tileable textures, like our imported **Rock_Color**. Increasing the x and y of the **Tiling** parameter to 5 will significantly improve the quality of our rock. Tiling has no effect on performance, nor does it require more memory to render the texture.



However, doing that will also create a problem when we try to apply non-tileable Occlusion and Normal maps to our model. To solve this dilemma, we will make use of both map-sets:

1. Remove the **Rock_Color** texture from the **Main Map Albedo** parameter and assign it to the **Detail Albedo** of the **Secondary Map**.
2. Set the surface color to a dark grey using the Color Picker next to the **Albedo** parameter.
3. Set the **Smoothness** parameter to 0.25.
4. Set the *x* and *y* **Tiling** of the **Secondary Map** to 5 (**Main Map Tiling** should be reverted to 1).
5. Assign the **Rock_Normal** texture to the **Normal Map** parameter of the **Main Map** and click the **Fix Now** button.
6. Assign the **Rock_Occlusion** texture to the **Occlusion** parameter.

With Normal and Occlusion Map assigned, our rock looks much better.



There is but one problem with this approach: secondary maps are much more expensive than main maps; therefore, Unity developers do not recommend utilizing Secondary Maps if Main Maps aren't being utilized.

While assigning the **Normal Map**, you've seen the **Fix Now** button pop up. This happens whenever you are trying to plug in textures to the **Normal Map** channel that weren't marked as such. Clicking on the **Fix Now** button automatically changes the **Texture Type** of **Rock_Normal** to the **Normal Map** in the **Import Settings**.

Rendering modes

As mentioned previously, the Standard material has four rendering modes that serve to fill the roles of different materials:

1. **Opaque:** This is used for solid or opaque objects. This is the Default go-to mode for most objects.
2. **Cutout:** This uses the Albedo alpha channel as a mask to isolate parts of the texture.
3. **Transparent:** This is used for objects with transparency, such as glass. The transparency parameter is transformed by the Albedo alpha channel.
4. **Fade:** This is very similar to the **Transparent** mode, with the only difference being that it also affects the specularity of the object, allowing it to gradually fade away by controlling the Albedo alpha channel.

To prepare our second model, we are going to rely on the Cutout Rendering mode of its material:

1. In the **Import** settings of the **Bush_Color** texture, check the **Alpha is Transparency** box. This will allow us to utilize the alpha channel to render leaves.
2. Change **Rendering** mode of the **Bush_Material** to **Cutout**.
3. The **Exposed Alpha Cutoff** slider will allow us to control the amount of cutout.



Using LODs in Unity

LOD stands for **Level of Details**. This is an extremely useful functionality that allows you to optimize your game by switching highly detailed objects with those of a simpler geometry, based on their screen space.



LODs are toggled based on the percentage of game screen that is being occupied by the object, not just the camera distance, as is the case in many other programs.

In Unity, LOD is represented by an **LOD Group** component.

How to prepare LODs

In order to make LODs work, you need to have the actual models — more precisely, multiple versions of the same model that scale down in polycount. Here are a few useful tips to follow when creating LOD models:

- The number of versions is completely up to you as Unity will allow the creation of as many as you need.
- Keep object silhouettes relatively close so that players won't notice when models are being swapped.

Setting up LODs in Unity

To demonstrate how LODs work, we are going to utilize meshes that were imported with external package in the previous chapter. So make sure that you've successfully imported the package and have the **Chapter 2 | Ruin** folder in the **Project** window. To start up, let's do the following:

1. Create an empty GameObject via **GameObject | Create Empty** in the top menu.
2. Name it **LODParent** (not mandatory).
3. Attach the **LOD Group** component via **AddComponent | Rendering | LOD Group**.

As a rule of thumb, you don't attach the **LOD Group** component to the actual GameObjects you will be using as LODs; instead, you attach it to the empty GameObject.

Let's take a look at the properties that are available for the LOD Group component in the **Inspector** view.



The LOD Groups at the top determine the number of LODs that this object has, and transition thresholds between them. As we've talked before, the switching is based on screen space and the percentages under the group names that represent the max point at which that particular LOD Group will be used.

You can create more LOD Groups by:

1. Right-clicking on the **LOD Group**.
2. Selecting **Insert before**.

Or, you can delete LOD Groups that you don't need by:

1. Right-clicking on the specific **LOD Group**.
2. Selecting **Delete**.

Thresholds are not fixed, and you can adjust them by dragging the LOD Group border and adjusting it to your preferences.

The camera icon above LOD Groups is a slider that allows you to manually adjust your camera to see the transition between LOD Groups.

Renderers is a list of models that become visible when **LOD Group** is active. To see them, select any LOD Group. You can add models there by clicking the **Add** button, or just drag and drop the model you want in there. To remove models, simply click the minus icon at the bottom-left corner. Another way to assign models to LOD Groups is to drop them into the LOD Groups directly. Every time you add new models, you will get a message from Unity about parenting the selected object to the object with the LOD Group. This is not mandatory; however, it's recommended that you do so.

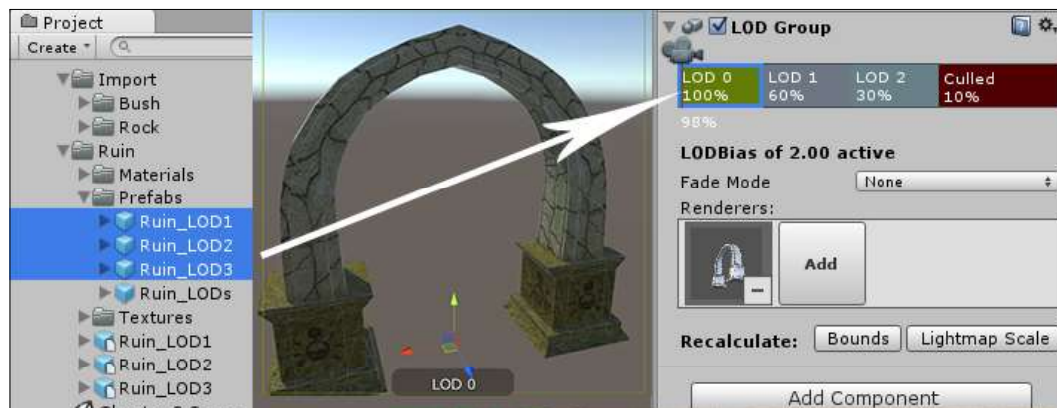


Bounds allow you to recalculate the bound volume of the object after the addition of a new LOD level.

Lightmap Scale updates **Scale** in the **Lightmap** property of the lightmap whenever LOD bounds are recalculated.

Drag and drop **Ruin** prefabs onto the respective **LOD Group**—all except for the **Culled** group. Culled is a point at which your model will be culled by the camera:

1. **Ruin_LOD1** to **LOD0**.
2. **Ruin_LOD2** to **LOD1**.
3. **Ruin_LOD3** to **LOD2**.



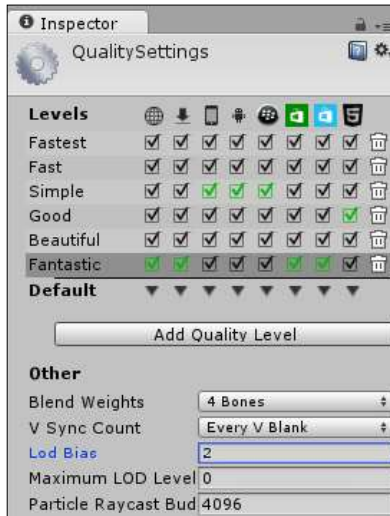
This will place the **Ruin** GameObjects in place of **LODParent** and indicate the **LOD** Group that is currently being rendered. You can now test the LODs with the camera slider and see models toggle as you pass the threshold.

LODBias

A lot of people are confused when they use camera slider because models don't seem to change at their thresholds, but at random spots. If that's the case with you, don't panic; that only means that it's working as intended and is being affected by **LODBias** parameter specified under **LOD Groups**. **LODBias** is used to adjust LOD Group thresholds to the quality settings of the game: the higher number will scale the amount of screen space required for LOD Group to change.

1. To adjust **LODBias**, go to **Edit | Project settings | Quality settings**.

2. Now, select your current quality level and change the **LODBias** parameter to 1.

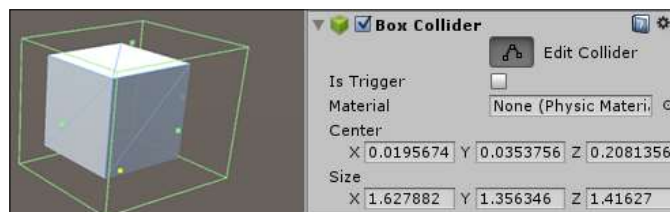


Now you should be good to go, and your GameObject swapping models should be at their designated places.

That's it for the LODs in Unity. You don't have to use them in your game, but they will definitely help to improve performance when used properly.

Collider

Collider is a shell that is used to register physical interactions between objects; however, this will not prevent objects from moving into each other. Colliders are there to register collision, not to prevent it. To add a collider to an object, we need to add yet another component called **Box Collider** under **Physics | Box Collider**. The purpose of the collider is to register the collision between itself and objects controlled by physics and essentially to prevent your character from walking through walls and falling through the floor. Collider can be transformed manually by clicking **Edit Collider** button and dragging its boundaries



You might have noticed that there are a lot more colliders, but the **Box Collider** is the most commonly used one because of its simple geometry. Based on object topology and collision precision requirements, you might have to use multiple colliders by adding more collider components. The best results can be achieved by using **Mesh Collider**, which will copy the topology of the assigned mesh reference. But it is also the most performance heavy Collider and might cause issues if overused on dynamic objects.

Summary

At this point, you should feel capable of manipulating GameObjects, adding components, importing models and textures, and configuring materials in the Editor. Hope you've come to appreciate Unity even more after you've witnessed how simple this engine actually is. If you need more information on how to handle importing objects from any specific 3D app, you can always reference the official documentation available at <http://docs.unity3d.com/Manual/HOWTO-importObject.html>.

If you wish to learn more about materials and physical-based shading, I would direct you to the official tutorial at www.youtube.com/watch?v=fD_ho_ofY6A.

In the next chapter, we will look into Unity's built-in **Terrain** system and how to use it.

