Chapter 4

Importing Static Assets

While Unity does have a handful of primitive objects, you will be importing most of the 3D art assets for your games. There are many pre-made assets available for free or for purchase at Unity's Asset Store. If you are mainly a programmer, or are just looking to prototype a game quickly to test functionality before creating assets for it, the Asset Store is invaluable. If you are a 3D artist and are looking forward to creating your own assets, or are in charge of procuring the assets from various and sundry sources, you will inevitably need to learn how to prepare them for use once imported into Unity. In this chapter, you will be exploring the importing, preparation, and management of static assets. Animated assets have different needs and will be covered in Chapter 6.

Supported Formats

Unity supports a large number of formats for imported meshes and textures. It can also read files directly from many popular DCC (digital content creation) applications, such as Max, Maya, Blender, Cinema4D, Modo, Lightwave, and Cheetah3D.

3D Assets

There are two main types of 3D models, or *mesh*, formats that Unity will read: standard export file types and proprietary application file types. The former includes .FBX, .3DS, .dxf, .obj, and .dae (Collada). The latter type can be read directly from many of the popular DCC applications, such as .max, .ma, .mb, etc. Besides containing the 3D mesh itself, these file types can store material, mapping, animation, and even the textures associated with the object's materials.

The important thing to know is that, internally, Unity converts everything to FBX on import. There are advantages and disadvantages to both import types. The export types are the most generic, as they can be used immediately by anyone. The downside is that if you update an asset in the original application, it must be exported into Unity again to be updated. If you are not using SVN or some other versioning software, this helps you by letting you keep versions of the asset to use if needed in the future. Keeping the proprietary files directly in your Unity project allows for quick

updating, but it has a few drawbacks. Most of the proprietary formats require the software to be installed and licensed in order for Unity to be able to use the content. It also is destructive. Unless you are using versioning software (where version changes are backed up and saved), if you want to revert to an earlier version of the asset, you will have no means of doing so unless you are backing up your
files externally.

The safest and most versatile file type for 3D assets is .FBX. It carries very little unneeded overhead and can be read and used by anyone.

Textures

Unity supports all image formats. Internally, most images are converted to .dds for desktop applications. For mobile, they are internally converted to the appropriate format for the selected platform. The bottom line is that you do not have to worry about the texture's original format. This allows you to work in Photoshop or other image-authoring applications, leaving layers intact. Because Photoshop layers are a means of preserving the history of an image's creation, you have the freedom of editing the image while it resides in the Unity project.

In Unity, as with any real-time engine, the optimal image size should be a power of 2. Use 32 x 32, 64 x 64, 128 x 128, 256 x 256, etc. to conserve memory. (See Figure 4-1.) Images that are over the base 2 size will take up the same amount of memory that an image of the next power of two would. The Texture Importer allows you to cap the image size, so you needn't worry about images that are too large unless you prefer to do your own size reduction manually.

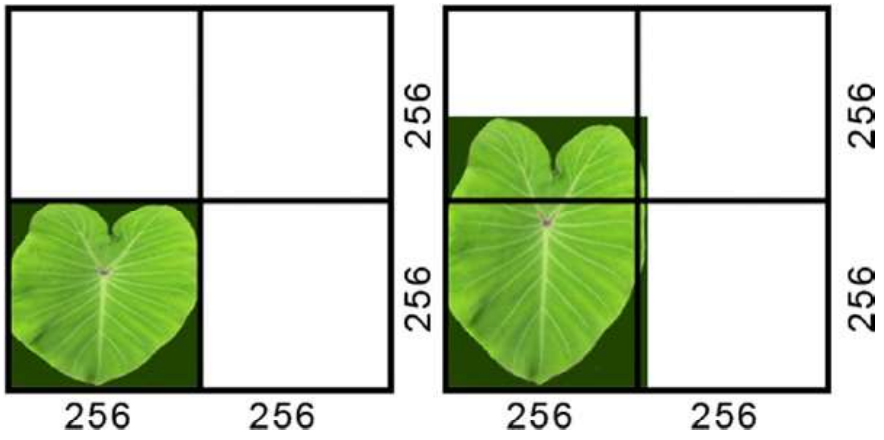

Figure 4-1. *Texture memory usage*

In Figure 4-1, the image on the left uses a 256 x 256 block of memory. The image on the right, retaining its original proportions and size, uses a full 512 x 512 block of memory (four 256 x 256 blocks).

Once an asset is imported, you may need to change the import settings, depending on the function of the imported texture. In Chapter 3, you had to change the import settings on a texture that was going to be used as a hardware cursor. Textures that are going to be used in 2D space only do not have to be MIP mapped. MIP mapping is the process by which an image is reduced in size, base-2 of course, and is smaller and blurrier for several iterations. The smaller, blurrier versions are used farther back in the scene to prevent "artifacting," which is the visual effect of the engine trying to decide

which color from the image should be used when the object in the scene is no more than a few pixels big (Figure 4-2). A perfect example is a black-and-white checker pattern. At some point, the object with the texture is so far away that only a few pixels must represent the check. If the choices are only black and white, the color could vary from frame to frame, causing a sparkling effect. Using a blurred version of the original texture, the pixels will always be a gray color at a distance.

Figure 4-2. The checker map (left). In the three blocks to the right of the checker map, the MIP-mapped texture appears on the left and the non-MIP-mapped texture is on the right, with the quads drawn farther back in space each time

You can see the MIP map in action in the Chapter 4 project file Misc Tests, MIP Mapping scene. If you press Play, you will see the two quads move backward and forward in the Game view.

Audio

Unity reads a limited number of audio file types. Just as with Unity's other assets, they are internally converted on import. Short clips of .AIFF and .WAV are converted to uncompressed audio. MP3 and longer clips are converted to Ogg Vorbis, .ogg, an open source format heavily used in real-time applications. If you are authoring for mobile and have set the platform in the Player Settings, the audio may be compressed to MP3. If the audio clip is not already an ogg file, you can manually set the compression type for higher quality or more efficient memory usage. Ogg is an open source compression similar to MP3 that does not involve license fees. For mobile, the MP3 license fee is covered by the hardware manufacturer.

The Importer

Importing assets is one of the most ubiquitous tasks in Unity. Most of your art assets must be imported. There are a vast number of choices on import, but there are only a few that must be carefully addressed. Fortunately, the defaults are generally quite useful.

From this point on in the book, most of what you will be doing will be aimed toward the book's project, so you will begin by creating a new project.

1. From the File menu, select New Project.
2. Navigate to where you want to keep it, and create a folder named **Garden Defender**.
3. Select it, and click Select Folder.

4. Import the following packages:
   - CharacterControllers
   - Particles
   - Scripts
5. Click Create.

You shouldn't need to save the current project, as you saved it before creating your builds.

Because you will be dealing with assets in this chapter, it will be more useful to switch back to the default layout.

6. From Layers in the top right of the UI, select Default Layout.

Importing Assets into Your Project

There are a few different ways to bring assets into your project. You can drag them directly into Unity's Project view, drop them into the project via the Explorer or Finder, or bring them in individually through the Assets menu's Import New Asset option. When bringing in multiple assets, you will want to use either of the first two methods.

Whichever method you prefer, you will want to make some organization decisions before you go any further. The FBX exporter has the option to export materials with the files. When the file contains all of its textures, they will be grouped in the imported assets folder. This is great for assets such as characters where the texture will be used only for that character, but for more generic textures, it is generally easier to manage them if they are all in one place. To this end, you will begin by adding the texture to your project first. This is always a safer way to make sure the texture can be found when the mesh is imported and the correct material is generated. If the texture cannot be found on import, a default material is made. It is easy to add the texture after the fact, but re-importing will mean rebuilding the material, or if you created a different material, it will require that material to be applied each time the asset is updated.

Importing Textures

Let's begin by importing the textures the 3D assets will require. You used Import New Asset in the previous chapter to import a single asset, a script, but when you have multiple assets, it is easier to load them directly into the project all together.

1. Locate the Game Textures folder from the Chapter 4 Assets folder where you unzipped the book's downloaded assets.
2. Copy and paste it into your project's Assets folder in the Explorer or Finder.
3. Return to the Unity editor, and watch as it imports the textures into the project and processes them.
4. Open the new Game Textures folder, and inspect its contents with the thumbnail slider all the way to the right (Figure 4-3).

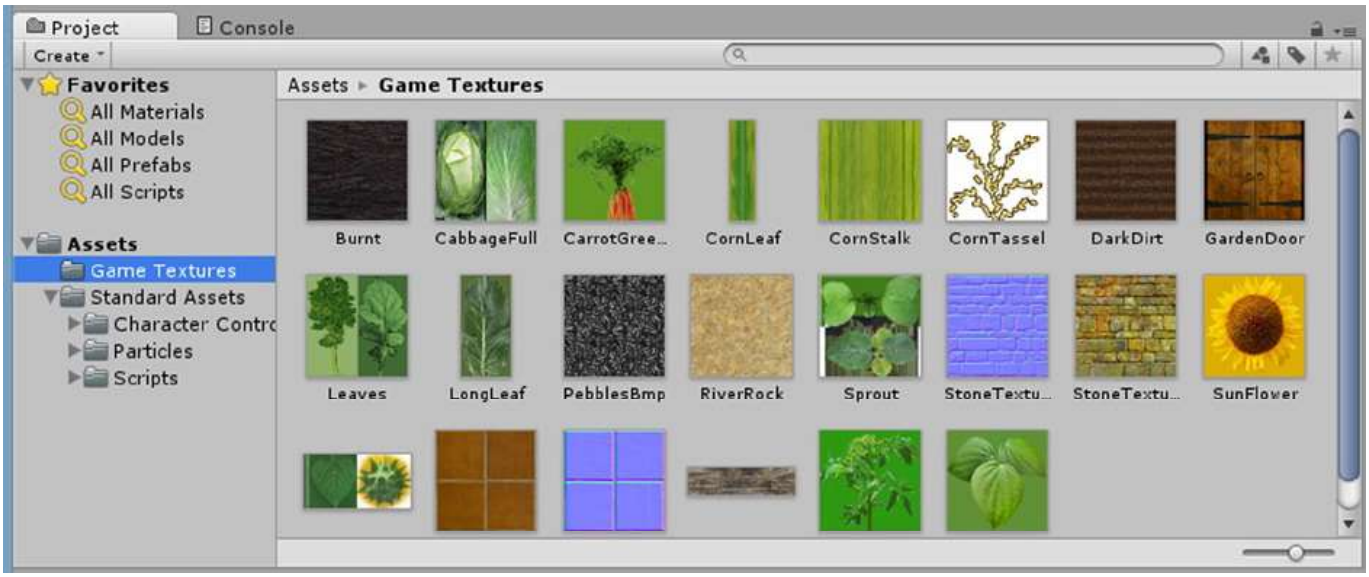

Figure 4-3. The contents of the new Game Textures folder in the Project view

5. Select the Leaves texture, and examine its import settings in the Inspector (Figure 4-4, left).

Figure 4-4. A preview of the Leaves texture's import settings (left), showing RGB colors (center), and the preview toggled to see its alpha channel (right)

6. Examine the Preview.

The color bar icon to the left of the MIP Map slider indicates that the texture contains an alpha channel (Figure 4-4, center).

7. Click the icon to toggle the alpha channel in the preview (Figure 4-4, right).

The Preview shows its size as 512 x 512. Most artists prefer to create their texture maps larger than required so detail is easier to add. The problem comes with having to keep the original for editing and a smaller version for the game itself. In Unity, you can set a max size for a texture in the game. This allows you to keep the original in the project but have it reduced when it is being used. Because these two leaves will never be seen close up, 512 x 512 is probably unnecessary.

1. For Default, click the drop-down menu and select 256 as the Max Size.
2. Click Apply.

You can also leave the default size large and override it for specific platforms. This allows you to let Unity optimize the texture automatically on build.

3. Click the Windows Store Apps button on the toolbar and locate the Override for Windows Store Apps option (Figure 4-5).

Figure 4-5. The Override for Windows Store Apps option for the Windows Store Apps platform

The Texture Type defaults to Texture when you import an image. This is a quick preset that adds compression of the appropriate type for the .dds format—in the case of the Leaves texture, DXT5. If you are a graphics guru and find the preset types to be less than acceptable, you can choose Advanced and tweak the various settings manually.

1. Click the Texture Type drop-down menu, (Figure 4-6) and select Advanced.

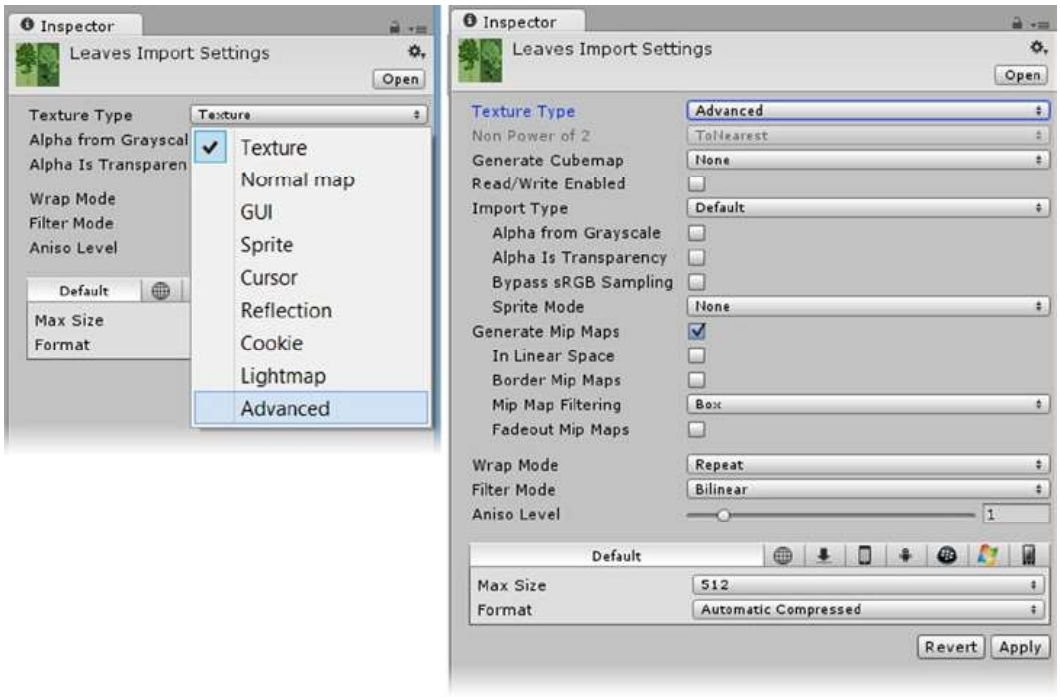

Figure 4-6. *The Advanced option for the texture importer*

2. Reset the Texture Type to Texture.

Let's look next at bump maps. If you are new to the term, *bump maps* add information that is used by the renderer to make 3D meshes look like they have more detail than they actually do have. They use the bump colors to determine where the light and shadows would go if the mesh *did* have the extra triangles or faces. As you may guess, it does add to the frame rate, but far less than if the mesh itself had the detail. Traditionally, a bump map was grayscale. White was bump forward, and black was bump backward.

Modern shaders now use what is called a *normal map*. The "normal" is a direction perpendicular to the face or triangle, pointing the direction that the face will be rendered (Figure 4-7).

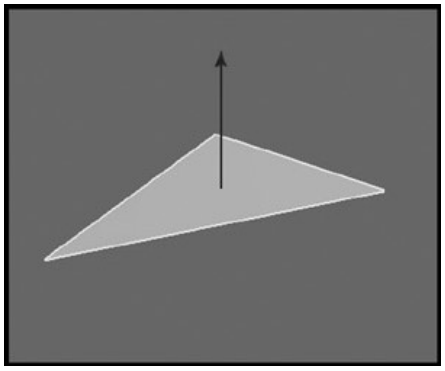

Figure 4-7. *The face normal of a triangle*

Grayscale is limited in the amount of detail it can store (it uses256 bytes) and has only information about height. Normal maps use the three RGB channels (3 x 256) to store information for three times the information and are easily recognizable by their distinctive colors.

3. Select the StoneTextureBump image, and examine it in the Inspector.

The texture displays the distinctive colors of a normal map (Figure 4-8).

Figure 4-8. *The distinctive lavender, cyan, and pinkish colors of a typical normal map*

The terra cotta tile image you will be using in the game comes with a traditional grayscale bump map, TerraCottaBump. You can easily convert it using the Normal Map option from the Texture Type drop-down menu.

4. Select the TerraCotta texture.
5. Set its Max Size to **512**, and click Apply.
6. Select the TerraCottaBump texture.
7. Change its Texture Type to Normal Map in the Inspector, and click Apply.

The default settings are often too harsh, so you will probably want to tone it down a bit. Reducing the Max Size will also help smooth it out.

8. Set its Max Size to **256**, and click Apply.
9. Set the Filtering to Smooth and the Bumpiness to about **0.1**.
10. Click Apply.

The normal map is more appropriate for a tile floor (Figure 4-9, right).

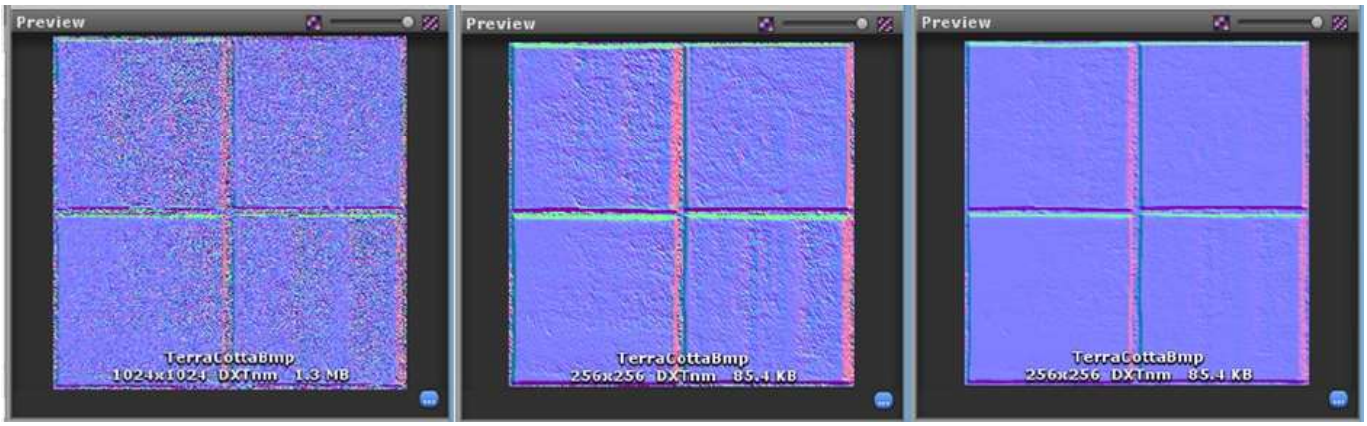

Figure 4-9. *The TerraCottaBump normal map using default settings (left), reduced to 256 Max Size (middle), and adjusted for more subtlety (right)*

Earlier in the book, it was mentioned that Unity shaders quite often commandeer a Texture's alpha channel for other purposes. For textures that have no need of transparency, using the alpha channel for a different purpose saves on resources. RGB uses 256 x 3 bytes. RGBA uses 4 x 256. By storing some other grayscale information in a texture's alpha channel instead of another RGB texture, you save 2 x 256 bytes plus the overhead of keeping track of the second texture. If an asset doesn't come with the extra information, Unity can generate an alpha channel from the texture's grayscale. For the terra cotta tile, it might be nice to have a glossiness map.

1. Select the TerraCotta texture.
2. Click Alpha From Grayscale, and click Apply.

The Preview now shows the color bar icon.

3. Toggle the RBG icon to show the new alpha channel (Figure 4-10).

Figure 4-10. *The TerraCotta texture's new alpha channel*

For glossiness, white is glossy and black is not, so the grout will be glossier than the tiles.

Atlasing is the combining of multiple objects' textures on the same image or texture sheet. It doesn't necessarily mean you will be using less *memory*, but it does mean there will be fewer "draw calls" made when a scene is rendered. There is also less overhead when there are fewer maps to manage. Among the textures you imported for the game, a few have been combined. The Sprout texture contains two different sprout images. The Leaves texture contains a kale leaf and a radish leaf.

To atlas some textures, you may not be able to keep a square texture. With the exception of some mobile platforms, this is not a problem as long as they remain base-2 dimensions. The SunFlowerLeaf texture, for example, is not square, but it does use base-2 sizes. It is not, strictly speaking, an atlased texture because the leaf, stem, and flower head meshes all are part of the same mesh, but it definitely cuts down on the number of textures and materials when they can be combined.

1. Select the SunFlowerLeaf texture.
2. Toggle the RGB/A icon to see its alpha channel (Figure 4-11).

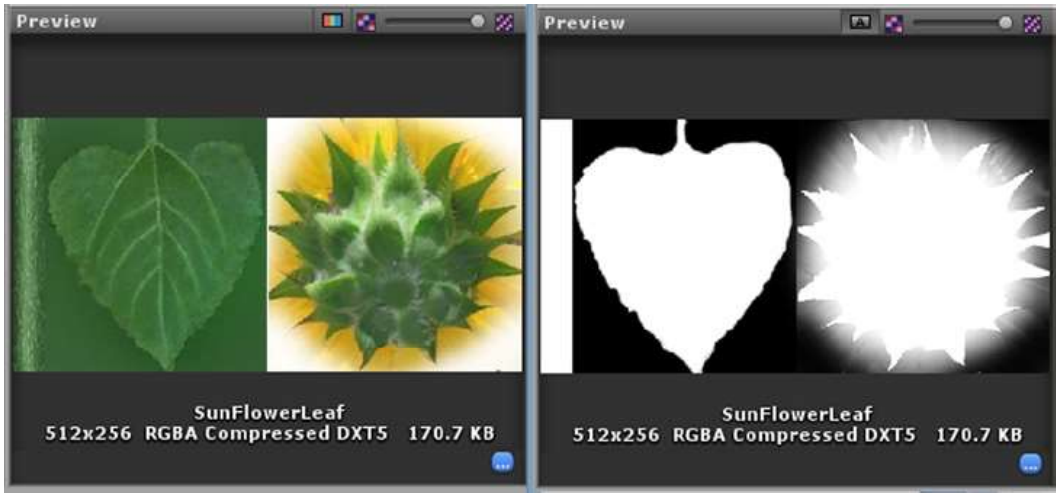

Figure 4-11. *The SunFlowerLeaf texture's alpha channel*

3. In the Inspector, set its Max Size to **256** and click Apply.

The Size in the Preview window now reports 256 x 128, showing that the maximum dimension is affected when limiting Max Size.

Importing Meshes

With the Textures already imported, it is now safe to import the meshes that use them. On import, not only will the meshes come in, but materials will be generated for each object using the main texture that was used in the object's native material.

1. Locate the Imported Assets folder from the Chapter 4 Assets folder where you unzipped the book's downloaded assets.
2. This time, with the Explorer (or Finder) open and Unity windowed on your desktop, drag the Imported Assets folder directly onto the Assets folder in the Project view.

This method only works the first time you import an asset. If you try use it to replace an existing asset, it will import the asset as a copy and increment the name. Replacing an existing asset should be done through the operating system, where you can specify *replace*.

3. Select the folder in the Project view to see what came in (Figure 4-12).

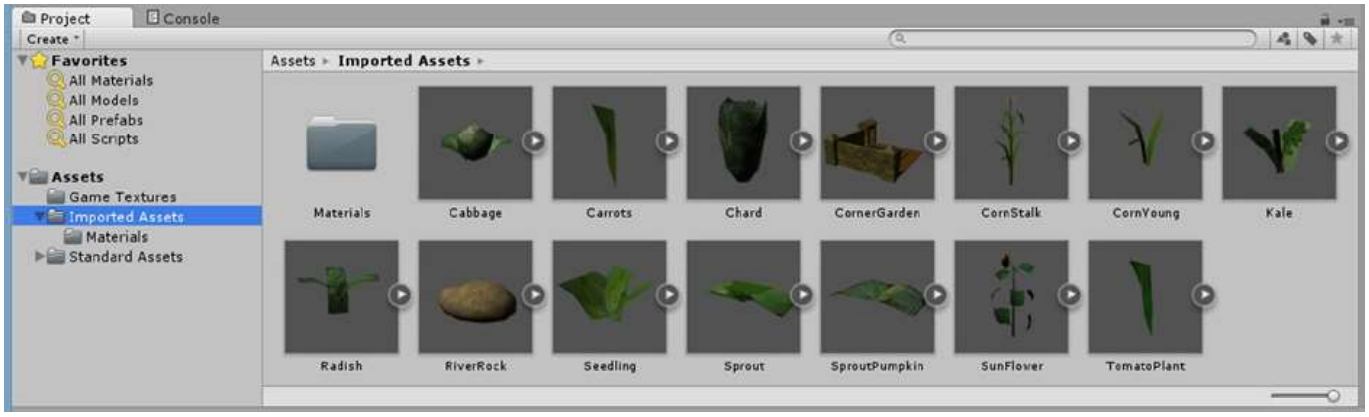

Figure 4-12. The newly imported mesh assets

When the meshes were imported, a material was generated for each using the main texture. As a default, the material is named after the texture (Figure 4-13).

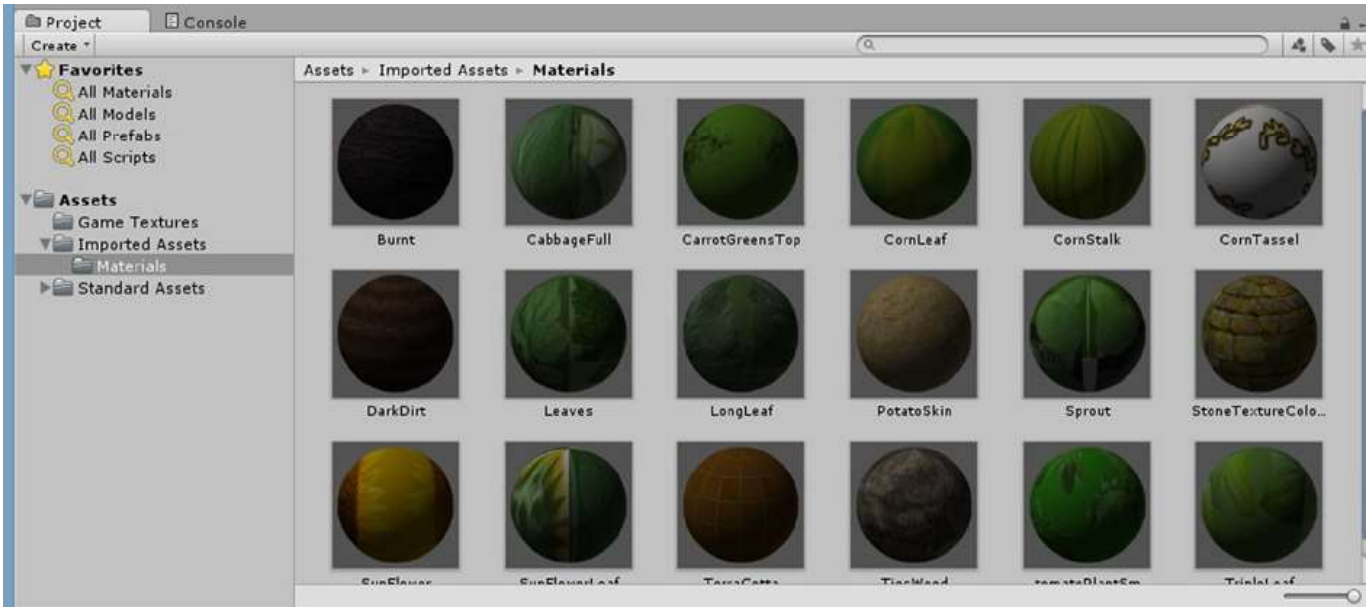

Figure 4-13. The materials generated for the meshes, using their main textures

The first thing you will notice is that the materials are rather dull and lifeless. The next is that most of the plants require both a shader that uses the texture's alpha channel as transparency and that most will also require a two-sided shader. Before dealing with the plants' materials, let's examine the garden structures that also came in.

1. From The Imported Assets folder, drag the CornerGarden asset into the Hierarchy view.
2. Add a Directional light to the scene, and set Shadow Type to Soft Shadows.
3. Set the shadow Strength to about **0.8**.
4. Adjust it so the garden is nicely visible, and toggle on Scene Lights (Figure 4-14).

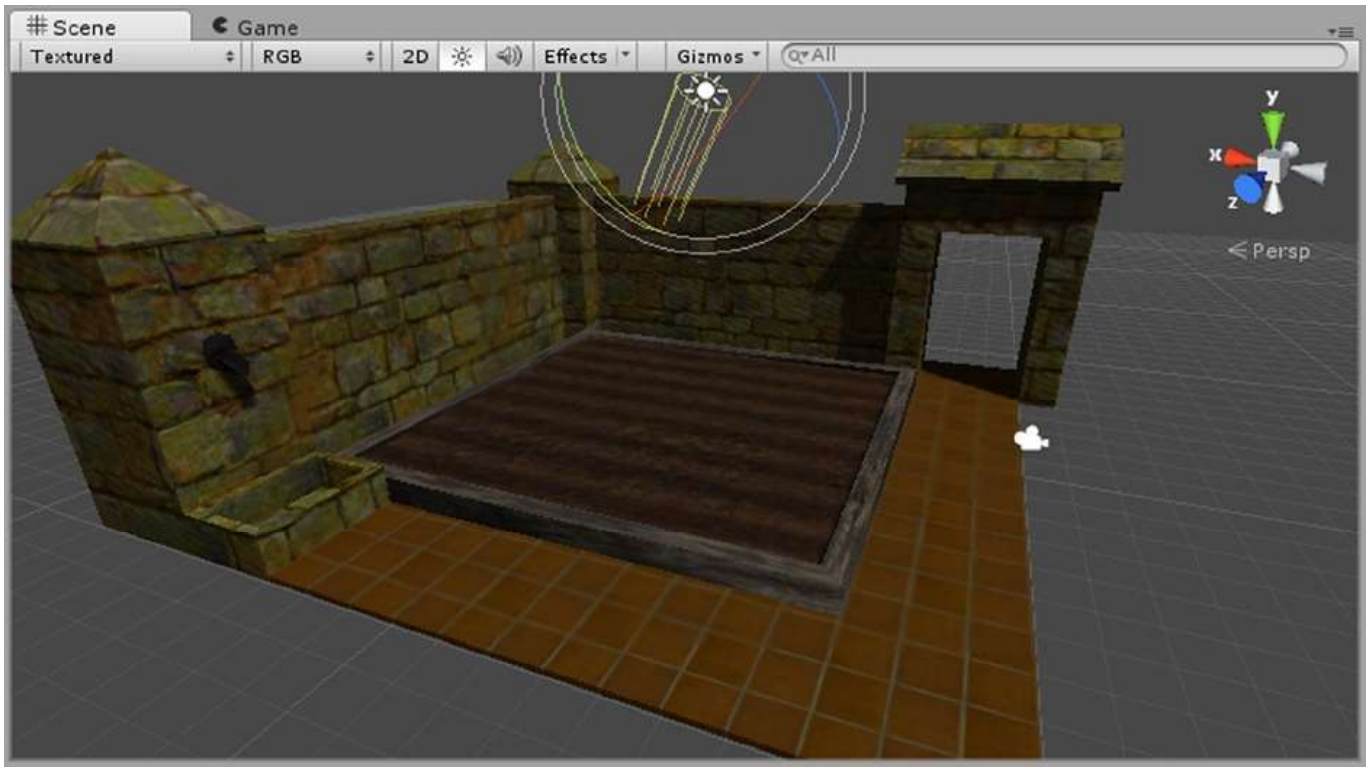

Figure 4-14. *The CornerGarden asset lit by a Directional light*

Scale Factor

The first thing to check when importing a new asset is the scale. Depending on the application it was built in and the purpose for which it was created, it may require an adjustment in its scale. The quickest way to check the scale is to drop a cube into the scene. The Cube primitive is 1 x 1 x 1 meter, or about 3 feet cubed.

1. Create a Cube, and move it onto the tiled path (Figure 4-15).

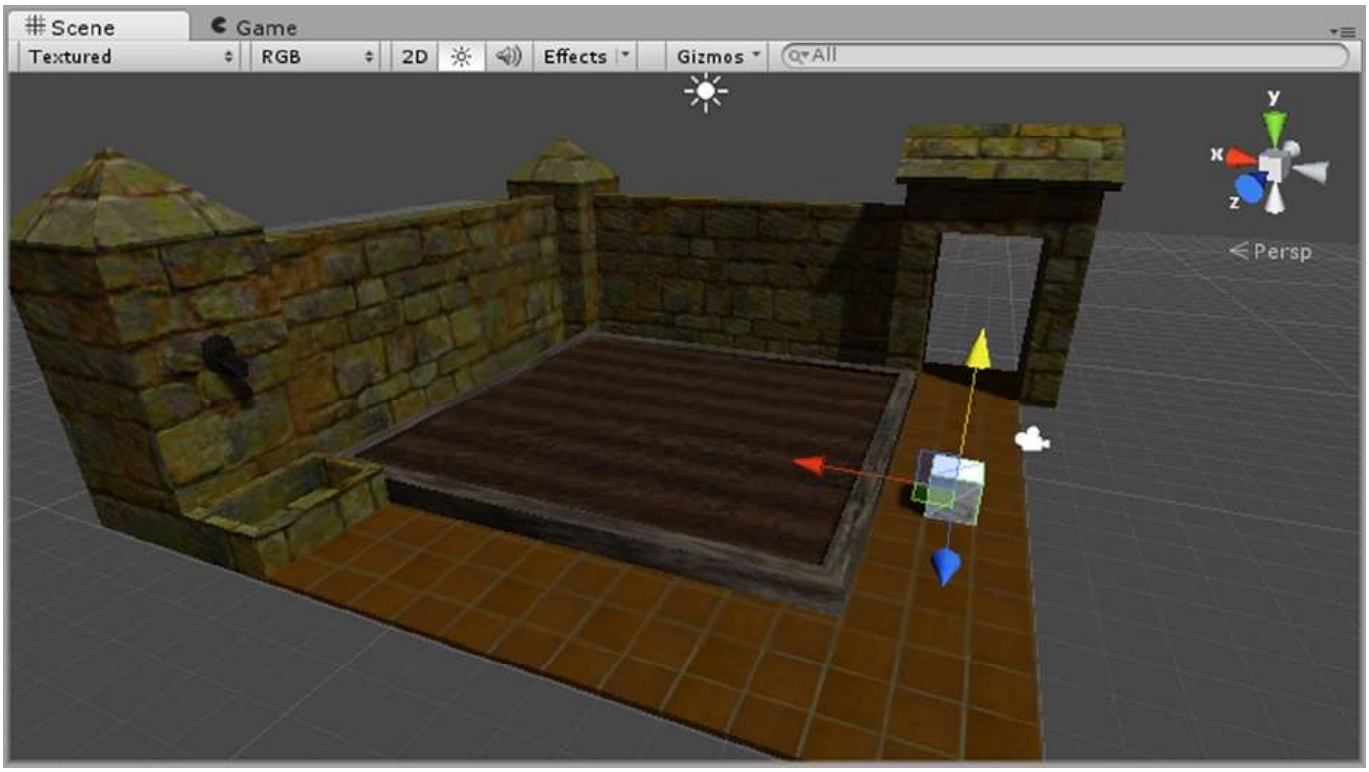

Figure 4-15. *A Cube put in the scene to check for scale*

If the Cube is 1 meter, the garden looks to be a bit on the large size. You could scale it in the scene, but if it needed to be re-imported or used in another scene, you would have to rescale it. Also, changing it in the scene view costs more in performance and can break batching if the scale is not uniform. The best practice is to adjust the scale of imported assets in the Importer itself.

2. Select the CornerGarden asset in the Project view.
3. In the Inspector, Model section, set the Scale Factor to **0.005** and click Apply.

The individual meshes shrink in the Preview window but will recover the next time you select the asset in the Project view. In the Scene view, you can position the Cube near the doorway to see the improvement (Figure 4-16).

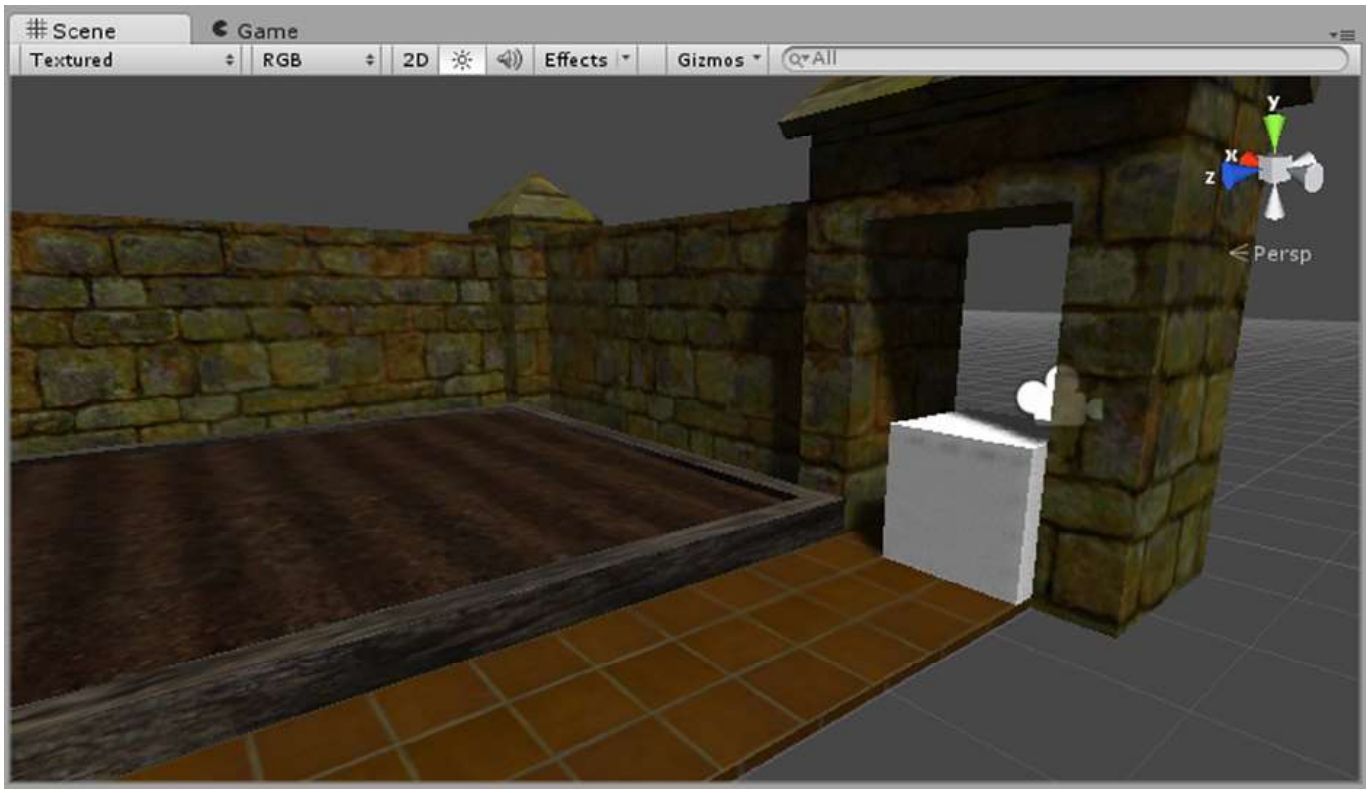

Figure 4-16. The scaled CornerGarden more in keeping with the size of the Cube

4. Deactivate the Cube in the Inspector.

Asset Optimization

No matter what platform or platforms you plan to author for, optimization is always something to be aware of. Besides file size (important for deploying mobile apps), optimization also applies to all types of asset management with the ultimate goal of faster frame rate. Some types of optimization, such as batching, model creation, and mapping must be designed into the assets before import. Other optimizations, such as the type of colliders used on an object are more flexible.

Batching Textures and Objects

The CornerGarden asset is made up of several separate meshes so that you can mix and match them to create many different configurations. In case it occurred to you that several objects will cost extra frame rate, don't worry. Unity "batches" objects that use the same material and have less than 300 vertices each. Another requirement is that stored vertex information is limited to three types. Typically, this is the x, y, and z location, normal (the direction it faces), and UV (its mapping coordinate). This means that each mesh is limited to 900 vertex attributes. If the requirements are met, instead of making separate draw calls for each object at render time, it can combine them into a single draw call.

Although Unity has a script that virtually "combines children" into a single object, batching is preferable because it is done after it is determined which objects are within the viewing frustum. Combining objects in different areas of the scene would cause a slowdown in frustum culling. An object's bounding box is checked first; if any part of the box is within the frustum, each face or triangle is then checked to see if it is within the frustum and must be drawn. An object that has meshes in and out of scene at any one time must always have each face checked, costing frame rate while doing so.

Static batching, available in Unity Pro only, is significantly more efficient because it can batch objects of any size as long as they are static and share the same material.

If you have Unity Pro, you have the option to use Dynamic Batching.

1. In the Project view, open the CornerGarden asset and inspect each of the meshes used to create the garden enclosure (Figure 4-17).

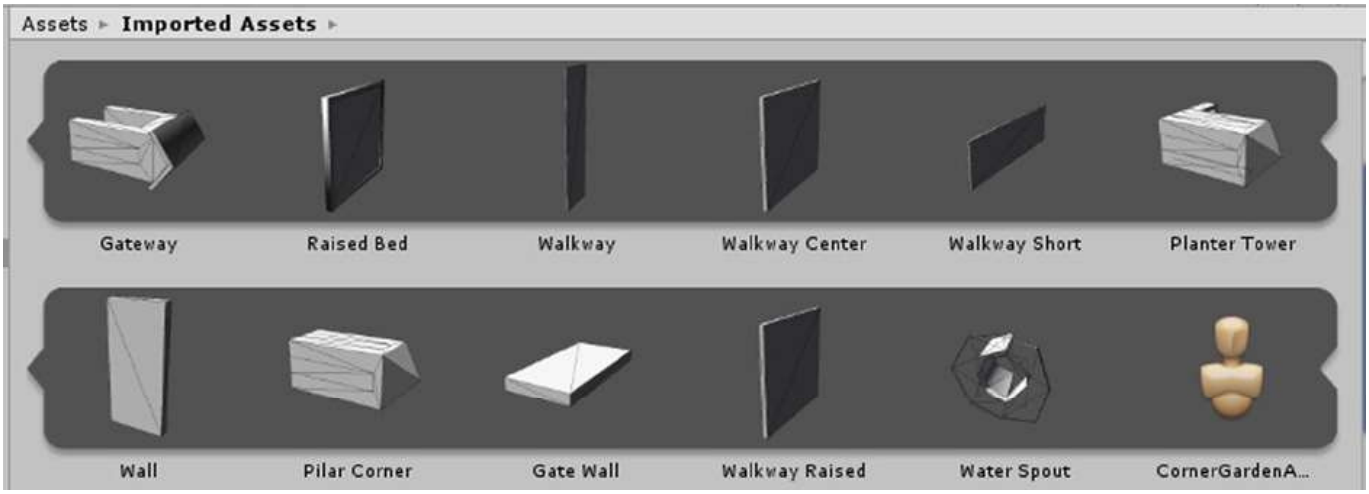

Figure 4-17. *The meshes that make up the CornerGarden asset*

As you select each one, note the vertex count shown in the Preview window of the Inspector.

To get accurate results for the batching, all of the objects must be within the viewing frustum (the screen's bounds) in Play mode. You had a First Person Controller to direct the camera in your terrain test scene. At present, you have a camera but no means of directing it at runtime, so you will have to set it up before you press Play.

2. Arrange the Scene view to be able to see all of the objects.
3. Select the Main Camera.
4. From the GameObject menu, select Align With View.

The camera's view now matches the Scene's view (Figure 4-18).

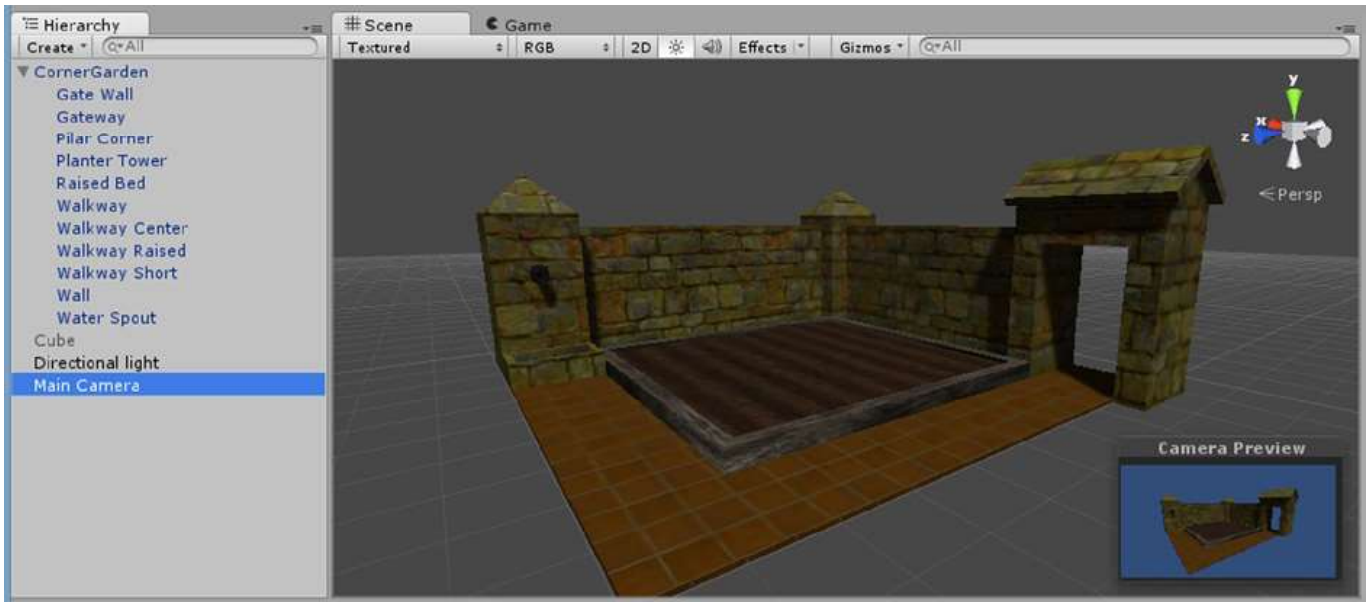

Figure 4-18. The camera aligned to match the Scene view

5. Click Play, and turn on Stats at the top right of the Game view (Figure 4-19).

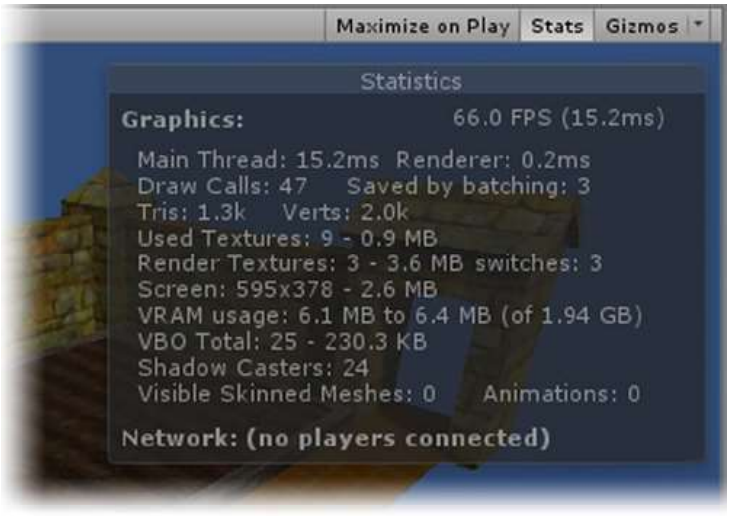

Figure 4-19. The Stats dialog in the Game view

If you have Pro, the Stats report 47 draw calls and only three saved by batching. If you don't have Pro and Dynamic Batching, your numbers may be different.

6. In the Hierarchy view, select the Raised Bed object and deactivate it in the Inspector.

The Stats now report 42 draw calls and 0 saved by batching. The Raised Bed is the only object with two materials, so it appears that Unity can save some draw calls when objects with different materials are combined into one mesh. To test this further, you can drag a couple of the multiple material-using plants into the scene and watch the numbers.

7. Drag the Sunflower into the Hierarchy view. (It should show up in the Game view at the corner of the walkways.)

The draw calls go up to 47, and three are saved by batching. The Sunflower also uses two materials.

8. Delete the Sunflower from the Hierarchy, and drag the CornStalk into it.

The draw calls go up to 48 and six are saved by batching. The CornStalk uses three materials. The tassel at the top requires an alpha channel, but the CornStalk would probably be more efficient by one draw call if the stalk and leaf materials had been atlased onto a single texture sheet.

9. Stop Play mode.

The garden returns to the way it was. The Sunflower is gone, and the Raised Bed it once again active in the scene.

So you've seen a bit of batching done on objects that use multiple materials, but so far, you've yet to see any batching on the objects that share a single material. It turns out you need to mark the objects as *static* before you will see any results. Static objects are objects that will not move in the scene. There are several Unity features that can make use of this setting to improve performance, and each can be handled separately if needed.

1. Select the CornerGarden in the Hierarchy view.
2. At the top right of the Inspector, check Static.
3. Click "Yes, change children" in the dialog.
4. Click the down arrow to see all of the features that were marked as Static (Figure 4-20).

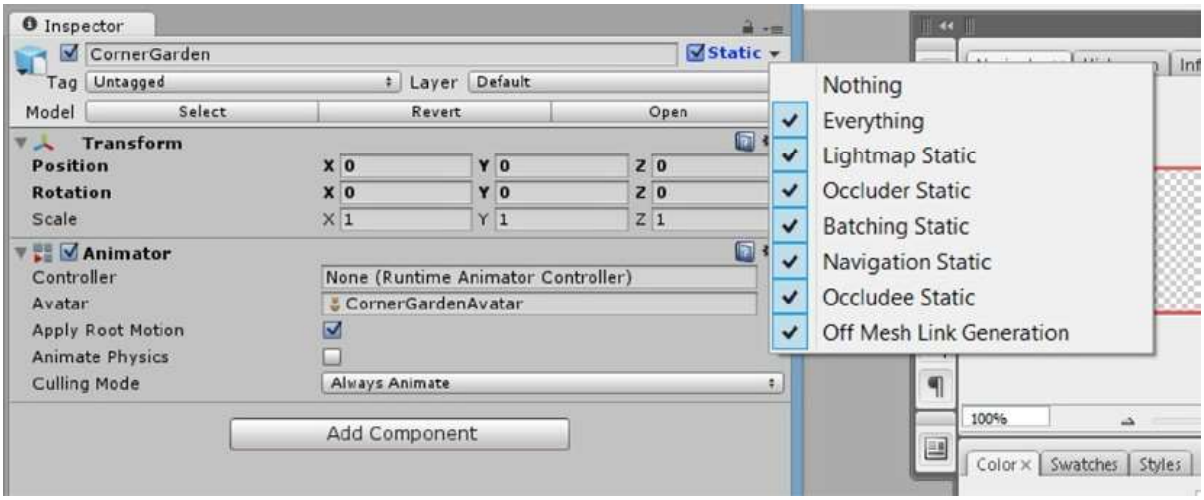

Figure 4-20. Marking the CornerGarden (and its children) as Static

A few of the settings that make use of the Static flag are lightmapping, occlusion culling, batching, and path finding.

5. Click Play, and examine the changes in the Stats dialog (Figure 4-21).

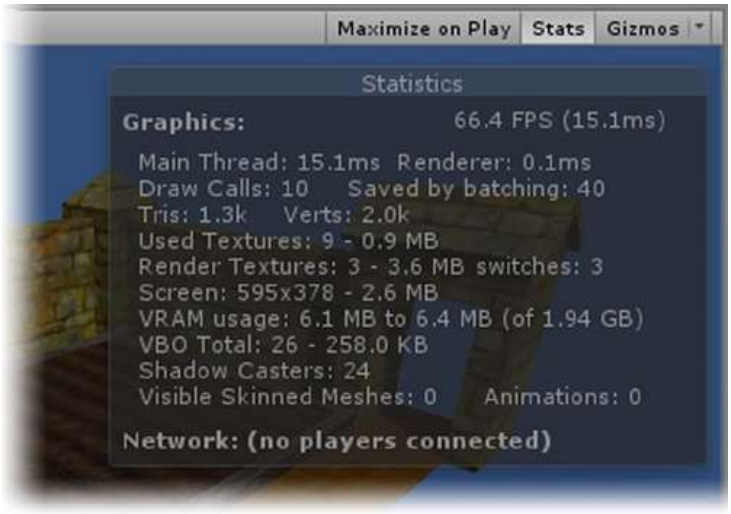

Figure 4-21. *The Stats dialog after setting the CornerGarden objects to Static*

This time the draw calls drop to 10, and 40 are reported saved by batching!

6. Stop Play mode.

There are a few guidelines for successful batching. It is worth keeping them in mind when preparing objects for your scene.

- Objects must share the same Material.
- Objects must be marked as Static.
- Vertex attributes must not exceed 900—that is, three attributes per vertex.
- Objects cannot use separate Lightmaps. (The extra UV map makes a total of four attributes per vertex unless you can do away with one of the standard ones.)
- Objects cannot be scaled in the scene. (Nonuniform scale is apparently allowed.)
- Instantiated objects must not use an instanced material (which they do by default).

Most of the plants contain an extra vertex attribute, alpha, for use as detail meshes in the Terrain. Currently, they are not using the special terrain shaders, so they can also be batched.

Vertex Count

If vertex count seems to be a mystery to you, there's good reason to be confused. There are two main reasons the count may not add up to what it looks like it should be. Lighting and UV mapping are the primary culprits. Besides containing location, diffuse, and alpha color information, vertices also hold lighting and mapping information.

The simple spheres in Figure 4-22 all have 120 tris, or faces. Their vertex number varies greatly. As long as the sphere is smooth and requires no mapping coordinates, what you see is what you get. Sphere 1 has 62 vertices, with the tris sharing a vertex where the edges meet. When the model is not smooth, as in sphere 2, vertices can no longer share the vertex normal (lighting) information unless they are also coplanar. Sphere 2, faceted (nonsmoothed or shared vertices), has a count of 264 vertices.

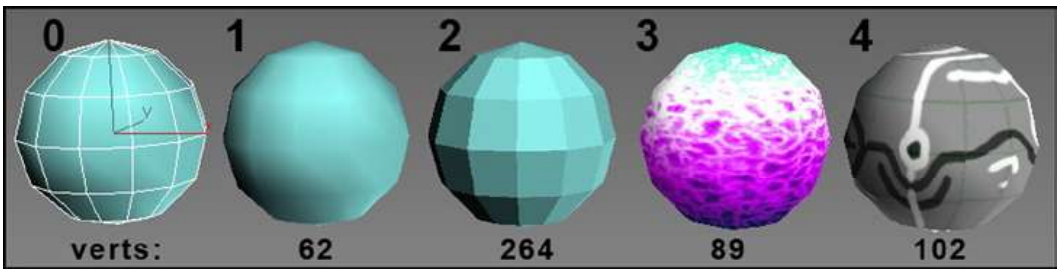

Figure 4-22. *Several spheres with the same triangle count of 120, but vastly different vertex counts*

If it requires simple mapping coordinates, as with sphere 3, it will require more vertices because the same vertex cannot exist at more than one position on the map. Figure 4-23, far left, shows a typical automatic mapping where a texture is wrapped around its circumference. As long as the texture is solid at the top and bottom, it may be okay. If it is Unwrapped, as with sphere 4, it may have an even higher count to accommodate the extra pieces (Figure 4-23, left center). For this reason, unwrapping a model becomes a study in compromise. Bigger pieces are easier to paint and will dictate a lower vertex count. The downside is that they tend to be more distorted on organic models, and they may not make efficient use of the texture map, so you will get less detail per pixel. Smaller pieces give you the opposite pros and cons. The spheres can be inspected in the Chapter 4 assets, the Misc Tests project, Mapping Spheres scene.

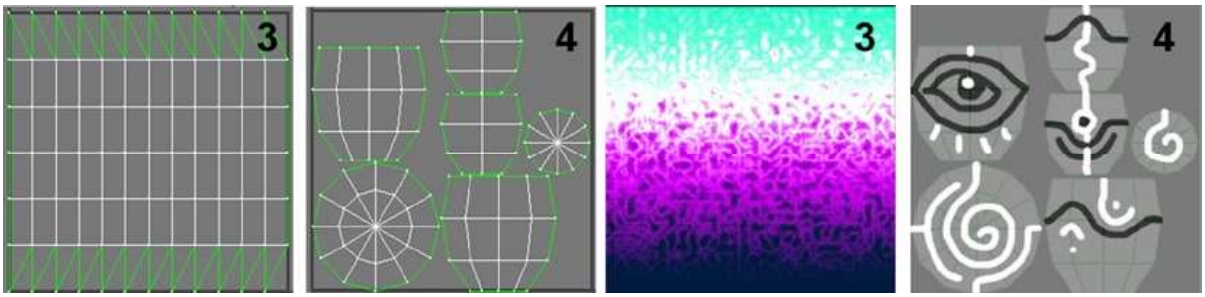

Figure 4-23. *The unwrap UVs of spheres 3 and 4 and their respective texture maps*

Before going any further, you can solve a small importer-based mystery. You've probably noticed a small human torso icon in the Project view (Figure 4-24), and an Animator component on each of the imported assets.

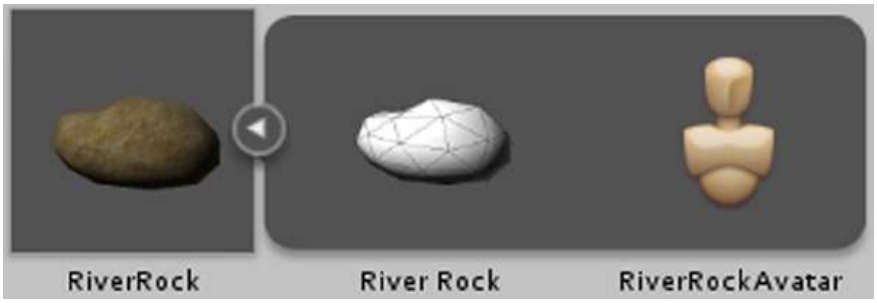

Figure 4-24. The RiverRock Avatar

This component is added automatically on import and is used only if the object is to be animated and, more specifically, is a character or object that will be using Mecanim to control its animations. Objects that are static do not require the component. At one time, Mecanim was only for use with characters and the Animator component was called the *Avatar component*. Unity made the decision to eventually replace its legacy animation system with Mecanim, so you will occasionally find character-centric names related to it.

You will be importing a few characters later in the book, but the current assets should be corrected.

1. Select the CornerGarden asset in the Project View.
2. In the Inspector, go to the Rig tab.
3. Set the Animation Type to None (Figure 4-25).

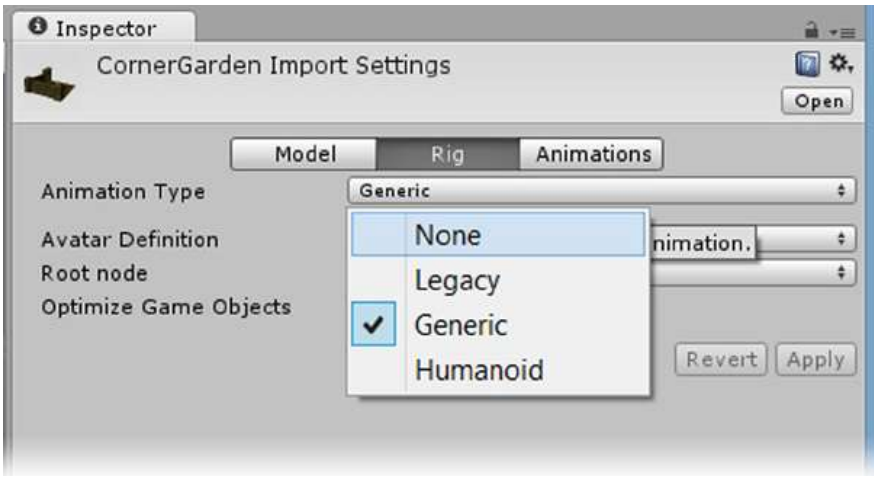

Figure 4-25. Changing the Animation Type to None

4. Click Apply.

The Avatar icon disappears and, if you check the CornerGarden in the Hierarchy view, you will see that the Animator component has been removed.

5. Select each of the other imported assets, and change their Animation Type to None.

6. Select the CornerGarden in the Project view once again.

Adding Colliders to Imported Meshes

Switching back to the Model section of the importer, you will notice an option to Generate Colliders automatically (Figure 4-26). This can be useful for quickly checking objects like structures to see how well a character or First Person Controller can navigate through or around them. The problem is that automatically-generated colliders are Mesh colliders.

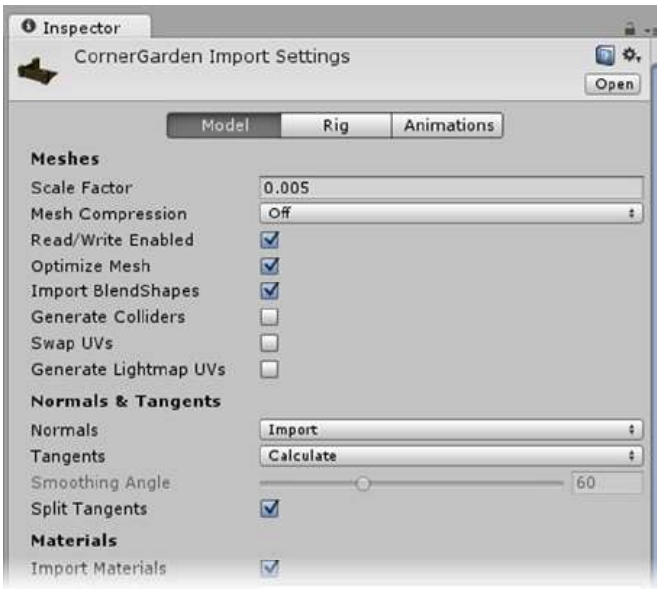

Figure 4-26. *The Generate Colliders option in the Importer's Model section*

These are the least efficient of the colliders because they use the object's actual mesh as the collider. Instead of doing a simple distance check for a primitive shape, each face must be tested. Not only does this take more resources, it will fail if the object has too many tris. Another problem is that most collision detection is performed with no more than one object having a Mesh Collider. If they both have Mesh Colliders, one will be dropped back down to something simpler.

With this in mind, let's look at the CornerGarden objects and decide what sort of collider is most efficient for each. To do that, you will need a bit more information about the game. The character will not be able to go into the Raised Bed and will never be able to get close enough to a wall to interact with it. With that information, you might want to put a simple Box collider on everything but the gate. The game, however, will also have plant-devouring vermin that will be dropped into the scene with physics, so two pieces (the Raised Bed and the Planter Tower) should have Mesh colliders to allow

the NPCs (non-player characters) to land and move around on the visible mesh surfaces. The player will get to shoot projectiles around the garden that shouldn't be able to go through walls, so you can use Box colliders on those.

1. Select the Raised Bed in the Hierarchy view, and from the Components, Physics menu, add a Mesh Collider.
2. Repeat step 1 for the Planter Tower.
3. Select the Gate Wall, and add a Box Collider to it.
4. Repeat for the Wall, Pillar Corner, and all four walkway objects.

The Gateway is a special case. Obviously, the character tasked with eradicating the vermin will have to come through the doorway, but the mesh itself is too complicated to waste a mesh collider on if you consider that most of the tris are up in the roof area. For that object, you will use two simple box colliders. Using multiple, simple colliders is a common and efficient solution in many cases.

5. Select the Gateway.
6. Add a Box Collider to it.

You could change its values in the Inspector until you got a custom fit, but you can use a shortcut key to speed up the process.

7. Hold the Shift key down.

Grips appear on the sides of the collider—they're green, to match the collider's wire color (Figure 4-27).

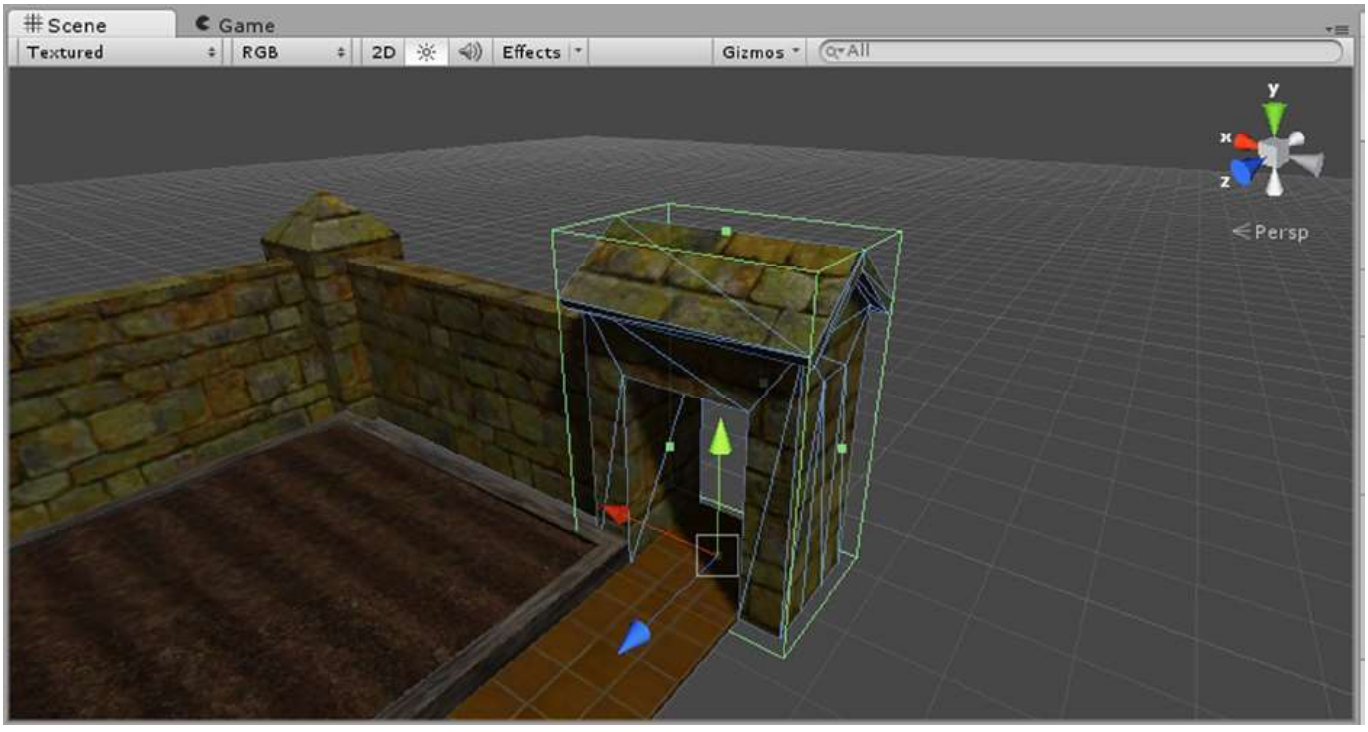

Figure 4-27. Grips appearing on the collider when the Shift key is pressed

8. Tip the view so you can access the grip on the underside of the collider.
9. Select the Rotation button so the gizmo won't get in the way of the bottom grip.
10. Holding the Shift key down, grab the grip and move the bottom of the collider up to the top of the door opening (Figure 4-28).

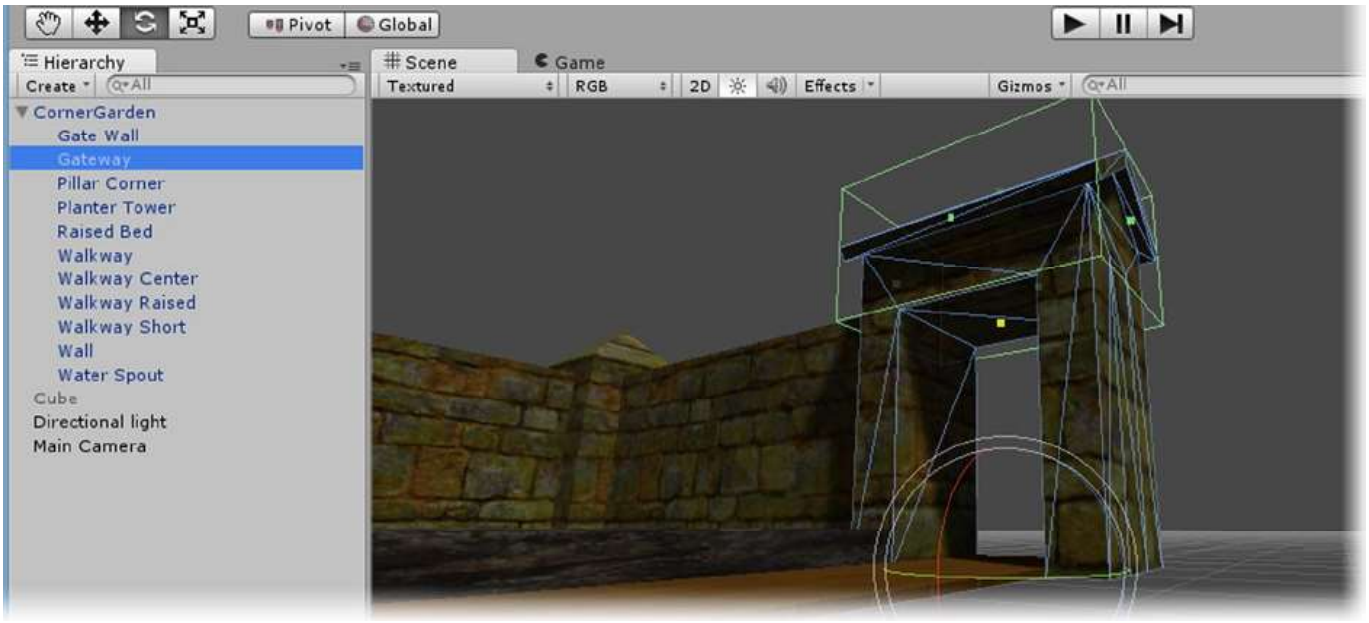

***Figure 4-28.** Moving the collider's base up with the bottom grip*

Even with the grips, the scaling and positioning of collider components is not as easy as manipulating their parent objects. For the side colliders, you will begin with a Cube object. You could add a Box Collider to an empty gameObject, but a Cube will be quicker to set up.

1. Create a Cube, and name it **Gate-Side Collider**.
2. Position and scale it to fit one side of the Gateway (Figure 4-29).

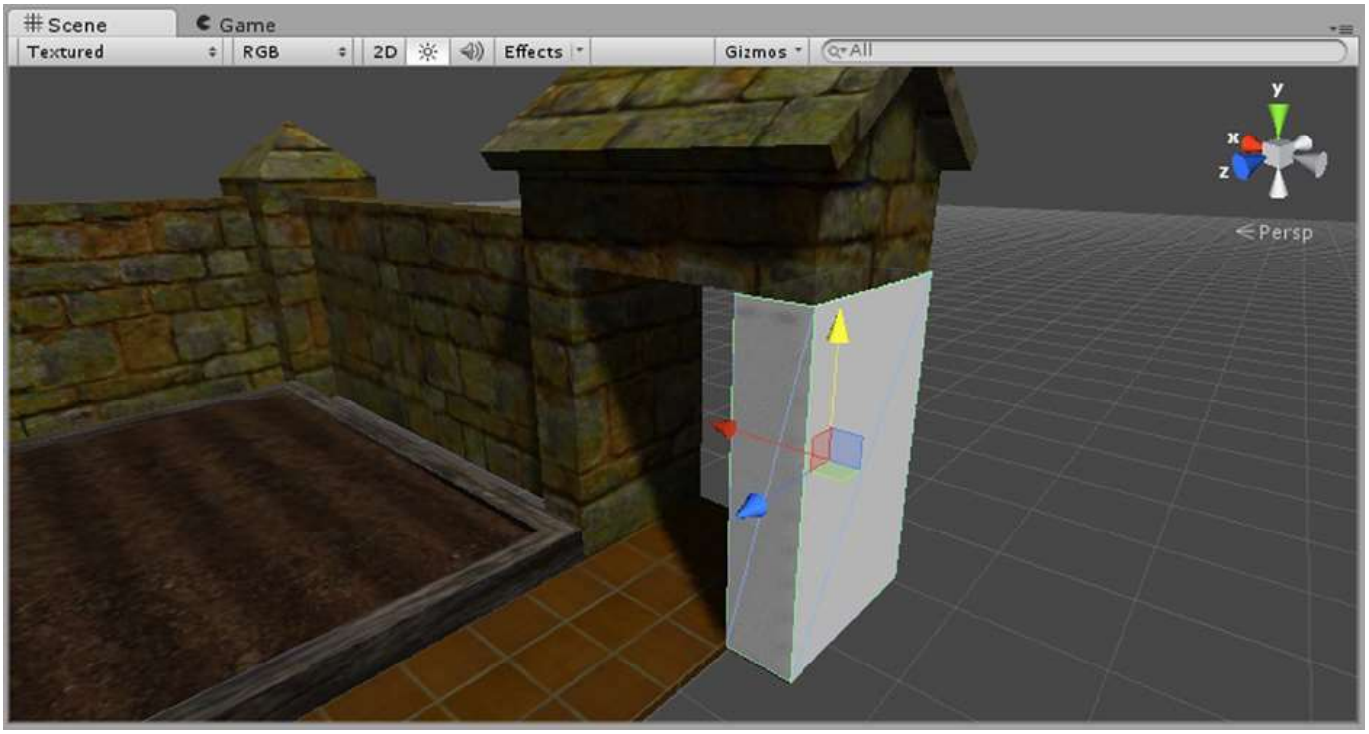

Figure 4-29. Scaling a Cube to the Gateway side

3. Right-click over the Cube's Mesh Renderer, and select Remove Component.
4. In the Hierarchy view, drag the Gate-Side Collider up and drop it onto the Gateway object.
5. Use Ctrl+D (CMD+D) to duplicate the new collider object.
6. Move the clone to the other side of the Gateway.
7. Select the Gateway object (Figure 4-30).

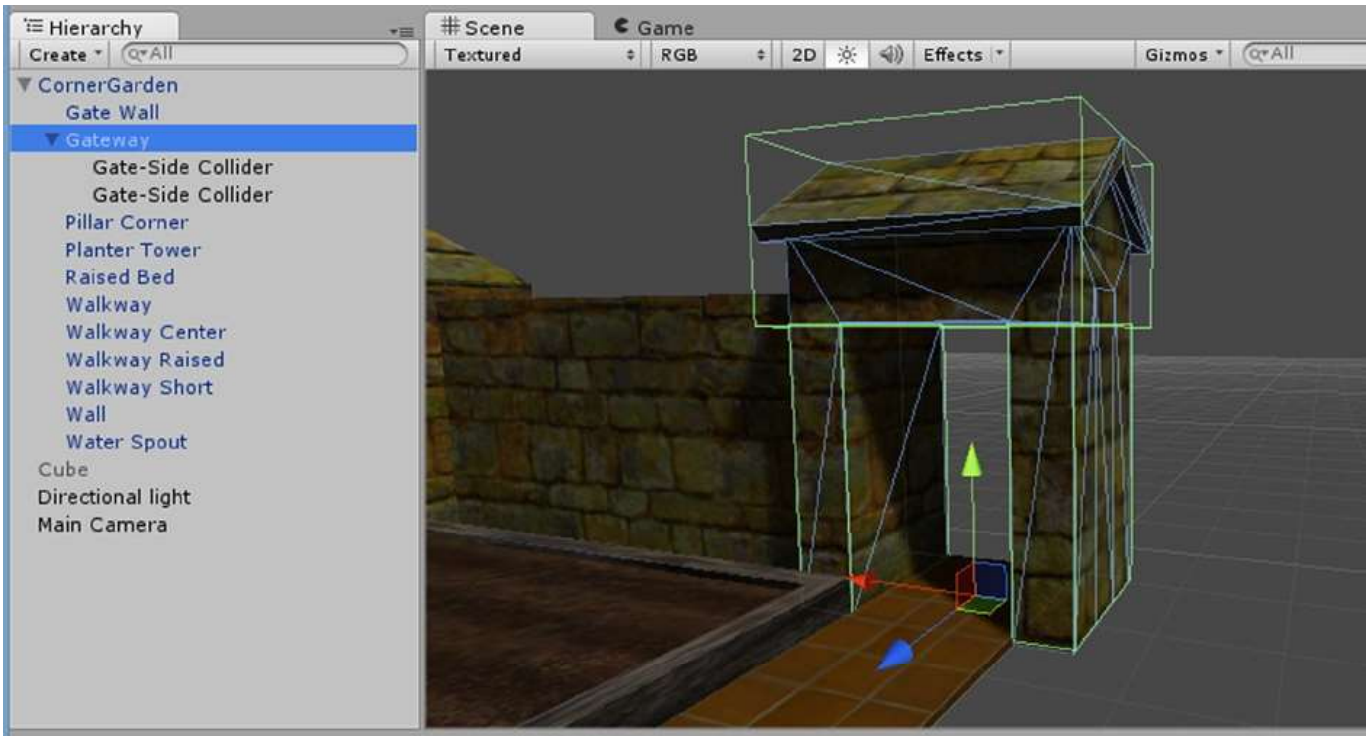

Figure 4-30. *The Gateway's colliders*

If you wanted to get fancy, you could add two more Cubes for their Box Colliders and rotate them for the top section instead of the adjusted Box Collider currently on the Gateway (Figure 4-31).

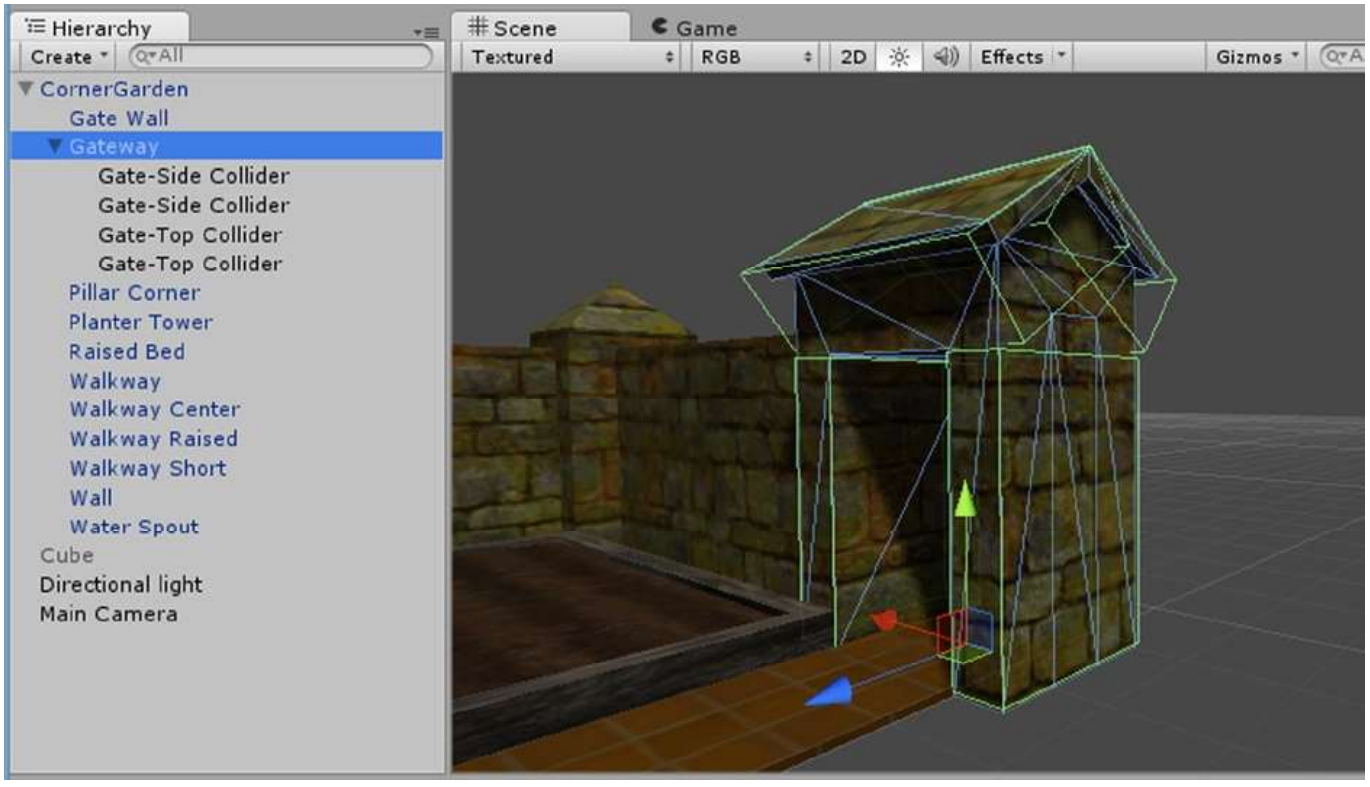

Figure 4-31. *A fancier collection of colliders for the Gateway*

Now that you know how to add multiple colliders to an object, you can probably figure out how you could make more efficient colliders for the Planter Tower and the Raised Bed.

Improving Generated Materials

Earlier in the chapter, you improved and added to the textures that were going to be used by the imported assets. When the assets came in, materials using the most basic (for basic, read *economical*) shader were generated. Let's go ahead and make some improvements to the materials. Be aware that using more complicated shaders may decrease frame rate. As always, use the fancy stuff sparingly.

1. Select one of the wall objects.
2. At the bottom of the Inspector, you will find its material, StoneTextureColored.

Shaders

Shaders are the code that tells the graphics hardware how an object's surface properties will look on screen. Materials in Unity are assigned prebuilt shaders, where the author can only assign textures or adjust parameters that already exist in the shader. Unlike many DCC applications where the artist can "build" a material by adding the attributes he wants to get the affect he has in mind, in Unity, you must look for the shader that best suits your needs. Writing custom shaders requires a different coding language and is beyond the scope of this book. There are custom shaders available at the Asset Store if you can't find what you want in the standard shaders.

The wall's Material, StoneTextureColored, uses the simple Diffuse shader. Fortunately, there are some good shaders available to improve the material generated on import.

1. Click the Shader drop-down list to see the shader choices (Figure 4-32).

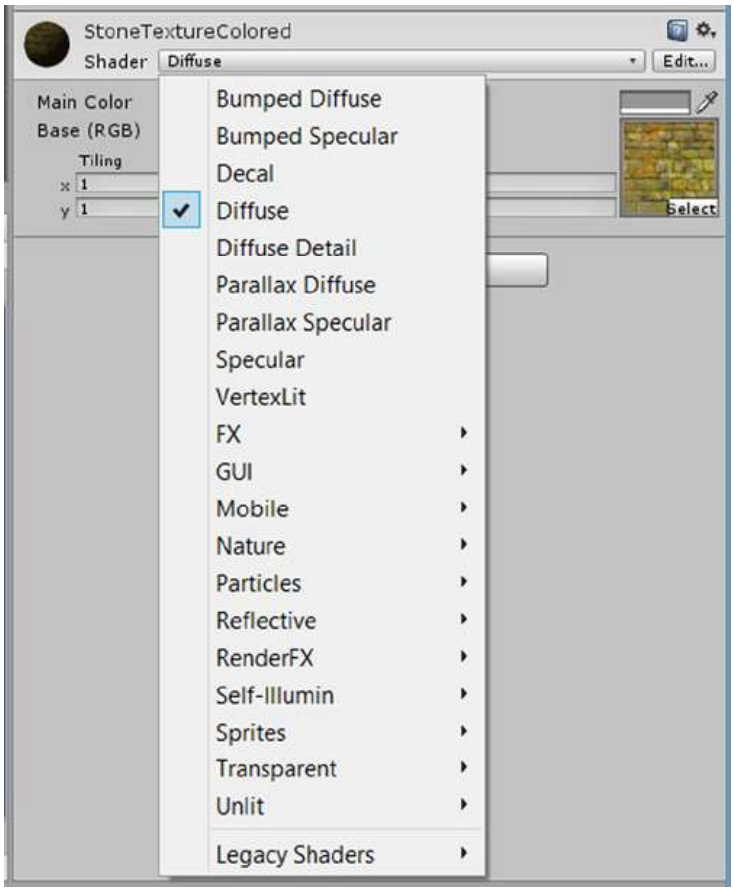

Figure 4-32. *The Shader drop-down list*

2. Select Bumped Specular.
3. Drag the StoneTexturebump texture from the GameTextures folder in the Project view onto the Normal Map thumbnail.

A warning appears telling you that the imported texture is not marked as a normal map.

4. Select the texture by clicking on the thumbnail to locate it in the Project view, and then clicking on it in the Project view.

You will see that it is set to the default import Texture Type, Texture.

5. Select one of the walls again.
6. Click the Fix Now button next to the warning message.

If you check the texture now, you will see that it has been changed to a Normal map texture type. The bump now appears correctly in the viewport (Figure 4-33).

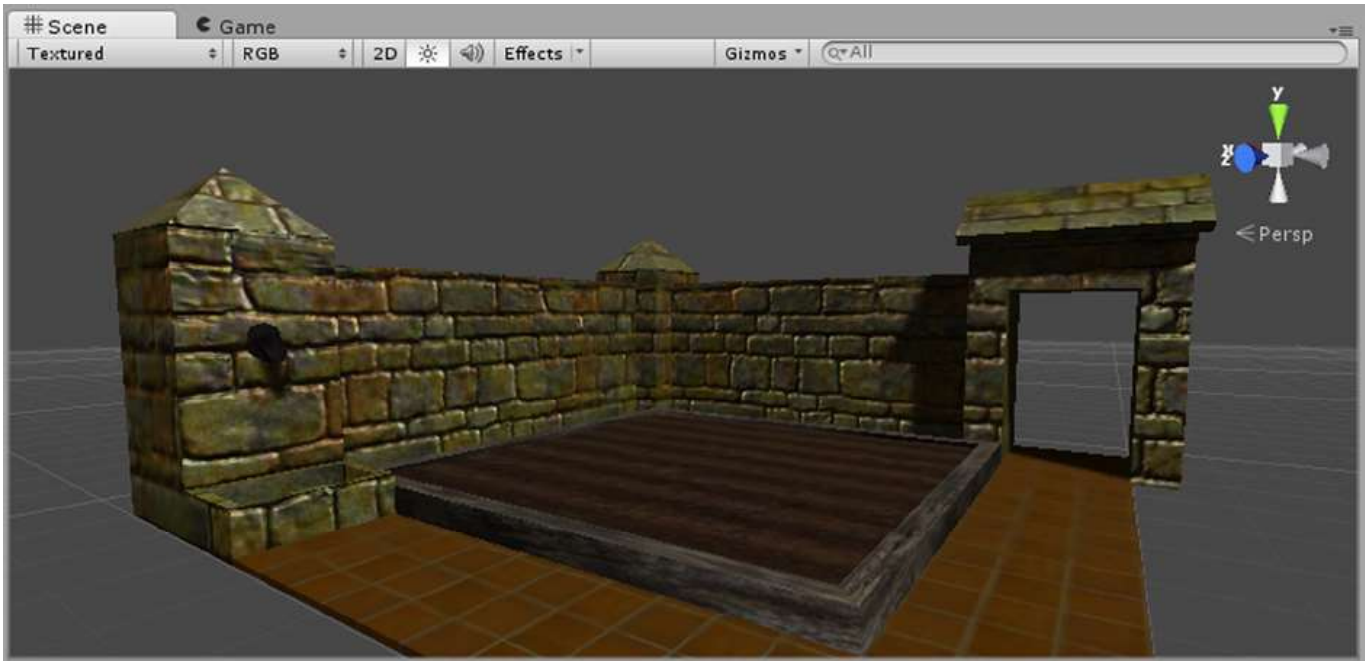

Figure 4-33. The glossy bumpy stone material on the walls

The problem now is that the StoneTextureColored texture has no alpha channel. The shader uses the Alpha channel as a glossiness map (Figure 4-34). No Alpha Channel defaults to white, so the walls are fully glossy. You may remember that you can have Unity generate an alpha channel for you.

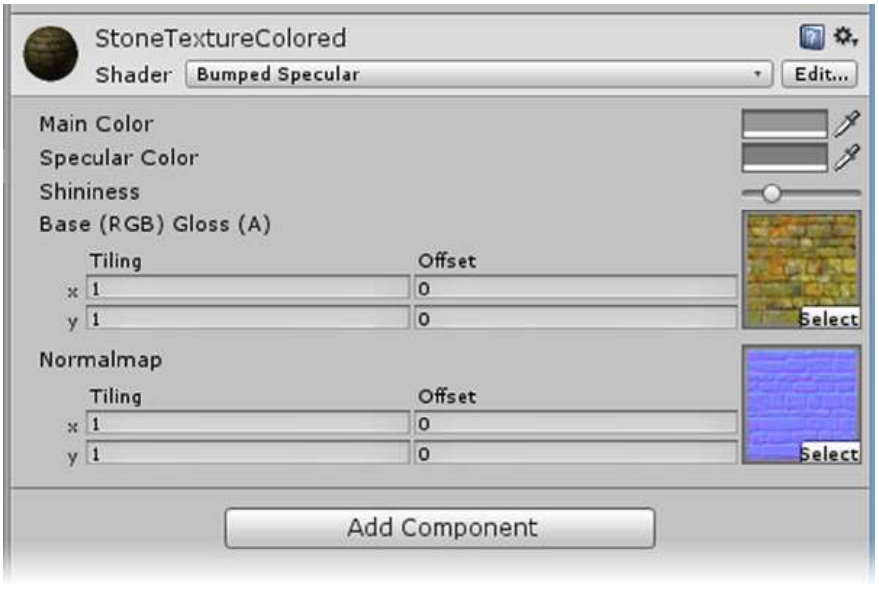

Figure 4-34. The shader using the alpha channel as glossiness

7. Select the StoneTextureColored texture in the Project view, and check Alpha from Grayscale.
8. Click Apply.

The Stone's glossiness is toned down to a more believable amount.

With the gloss reduced, you might decide the mapping on the walls is a bit off. You can adjust the Offset to get rid of the thin ledge at the top of the wall.

9. Select a wall again to gain access to the StoneTextureColored material, or select it from the Project view by filtering for materials to help locate it.
10. Set the y Offset for both the Base and Normal map textures to **0.04**.

The top of the wall looks much better (Figure 4-35).

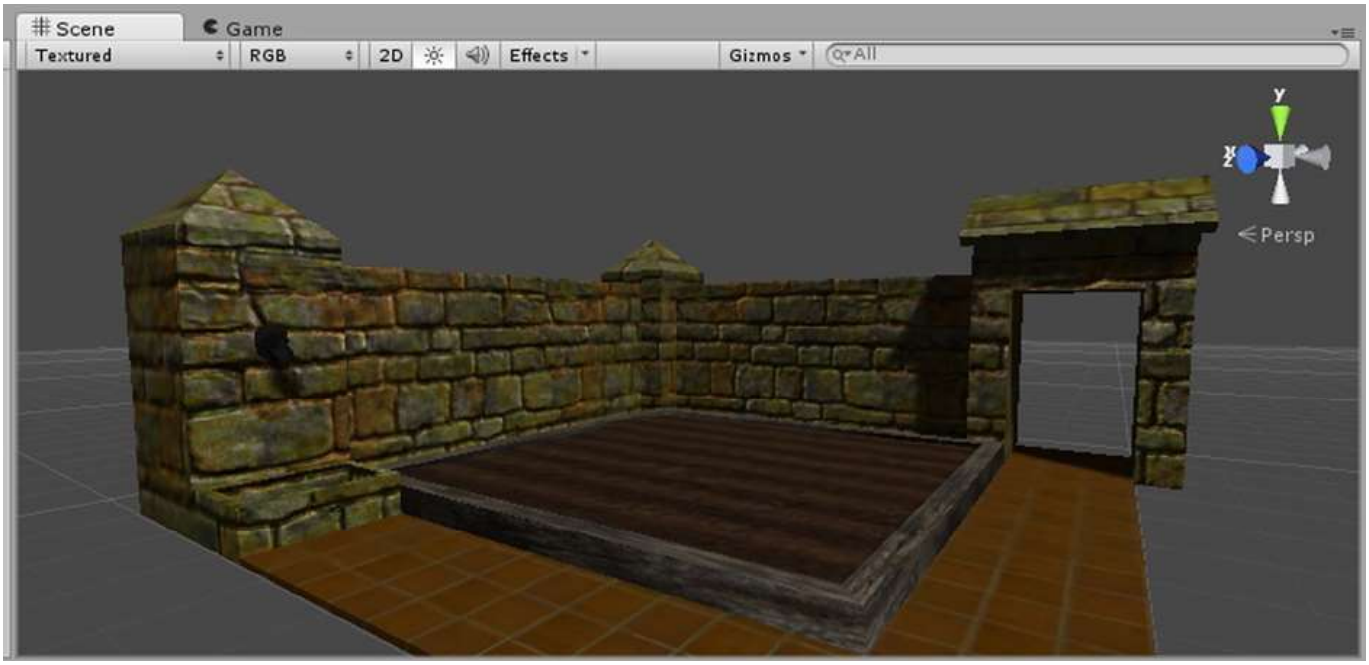

Figure 4-35. *The results of adjusting the two stone textures' y Offset*

Earlier in the chapter, you generated a normal map from a grayscale bump map for the TerraCotta texture. You also created an alpha channel using the TerraCotta's grayscale, so you should be ready to go on the TerraCotta material.

1. Select one of the walkway objects.
2. In its TerraCotta material, change the shader to Bumped Specular.
3. Drag the TerraCottaBump into the Normalmap thumbnail.

The tiling is obviously different (Figure 4-36).

Figure 4-36. Tiling mismatch between Base and Normalmap

4. Set the x and y tiling to **4** to match the Base texture's tiling.
5. Adjust the Shininess until you like the result.

The last two materials belong to the Raised Bed. The wood texture probably has enough detail that it doesn't warrant a normal map. The DarkDirt material could stand to be a little brighter.

1. Select the Raised Bed.
2. In the Inspector, make its Main Color a bit lighter.

You may have noticed that the Main Color is a neutral (150,150,150) gray. In 3ds Max, where these assets were made, a diffuse texture overrides the default Diffuse color, gray. On import, Unity blends the color and the texture by adding them together. The color *could* have been changed before export, but leaving it a light gray can be a time saver. With the light gray darkening the texture slightly, you have a means of "highlighting" an object on mouseover or some other event. By temporarily changing the Main Color to white through scripting, the texture becomes brighter without the more resource intensive use of a second texture. While this is probably not very useful for a first-person shooter, it is a mainstay of adventure or exploration type games.

The Plants

Typical plants require special shaders. If the textures have alpha channels for transparency, they will require a shader that uses the alpha channel for transparency and possibly renders two-sided. For substantially sized plants, you may also want shadows. These three requirements limit the shaders for the plants.

Because the CornerGarden objects now have colliders, you can quickly "plant" the plants by dragging them onto the walkways or Raised Bed. You've also got a directional light in the scene, so you will be able to check on shadows. You may not use all of the plants in the basic game, but you will want to see what they look like in the scene before you make any decisions about which to use.

1. Drag each of the plants from the Project view into the garden in the scene.
2. Create an Empty GameObject, and name it **Garden Plants**.
3. Drag all of the plants onto the new parent object.
4. Select Garden Plants, and check Static in the Inspector.
5. Choose “Yes, change children” in the dialog.
6. Rotate the Sunflower and CornStalk to face the view (Figure 4-37).

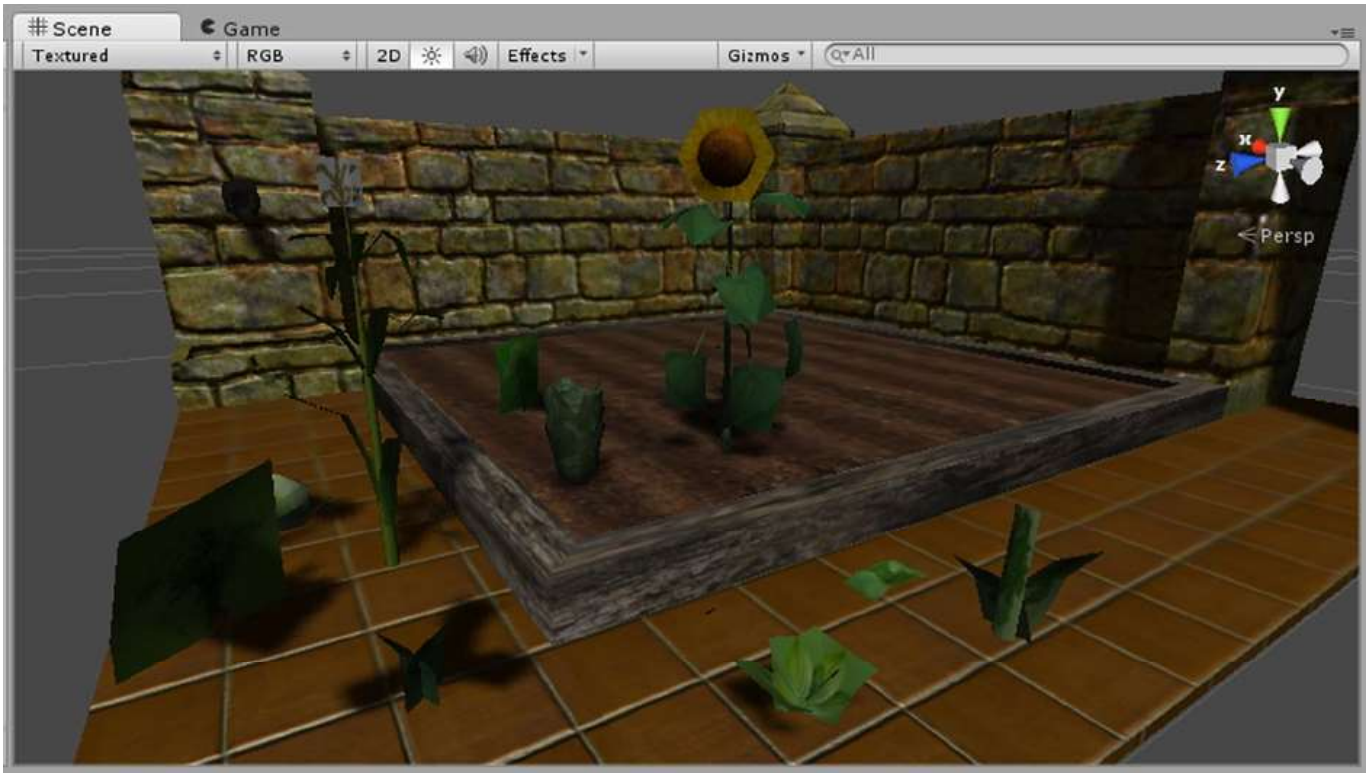

***Figure 4-37.** The plant assets*

Upon examination, you will find three types of plants in the collection. The simplest are little more than crossed planes. Besides the Tomato (which you saw in your terrain test), you have Carrots. Both of these two use images to simulate thick bushy foliage. Because there are basically only two or three crossed planes, you will not want the plants to receive shadows, and they must render the image on both sides of the planes. The Soft Occlusion shaders used for the terrain plants will be a good choice.

1. Move the TomatoPlant near the Carrots so you will be able to compare shaders.
2. Change the shader on the Carrot to Nature/Tree Soft Occlusion Leaves.
3. Adjust its Base Light value to make it slightly brighter.
4. Change the shader on the TomatoPlant to Transparent/Cutout/Soft Edge Unlit.

Both shaders have a slider to control the alpha cutoff in case the texture's alpha map is showing some background bleeding.

5. Try adjusting the Alpha cutoff in both shaders.
6. Orbit the view to make sure both sides of the planes are rendered for both plants.

The Transparent shader does not cast shadows, but both *do* render two-sided (Figure 4-38).

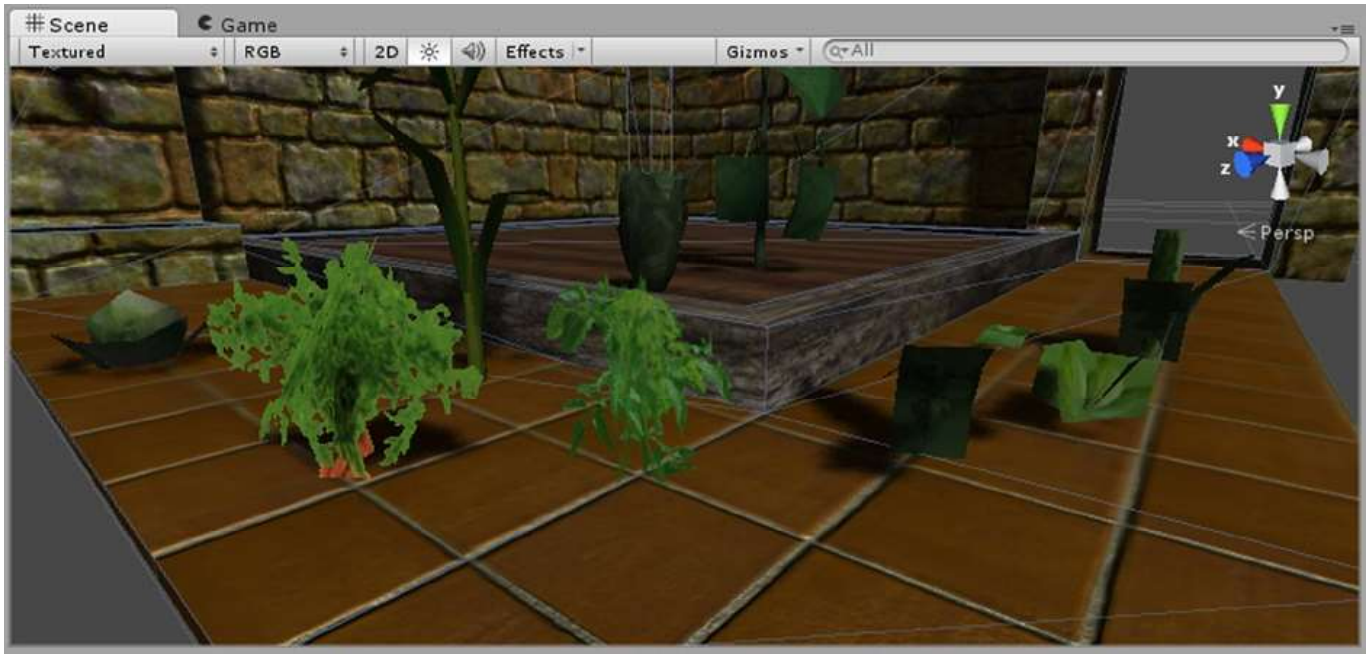

Figure 4-38. *Carrot and TomatoPlant; the Carrot casts a dynamic shadow*

Because shadow casting can be costly, and even if you are using Unity Pro and Deferred lighting, you may decide that minor plants do not warrant dynamic shadows. Let's see what happens with baked lighting. First you will need to set all of the plants as Static so they will be capable of generating shadows during the baking process. You should also save the scene to make sure you don't lose all of the setup you've done.

1. Select all of the plants in the Hierarchy view, and check Static at the top of the Inspector.
2. From Files, choose Save Scene.
3. Name it **Garden**.
4. From the Windows menu, open the Lightmapping window.
5. Click Bake Scene from just above the Preview window.
6. When the baking is finished, turn the Shadow Distance to **0** in the Lightmapping dialog (Figure 4-39) so you will see only the baked shadows.

Figure 4-39. Shadow Distance set to 0 to show only lightmapped shadows

Besides the expected absence of the dynamic shadows, there seems to be very little change. Looking at the generated maps in the Lightmapping, you can see a few pixels' worth of something at the lower left (Figure 4-40). If you do not have Pro, you will only have a far map.

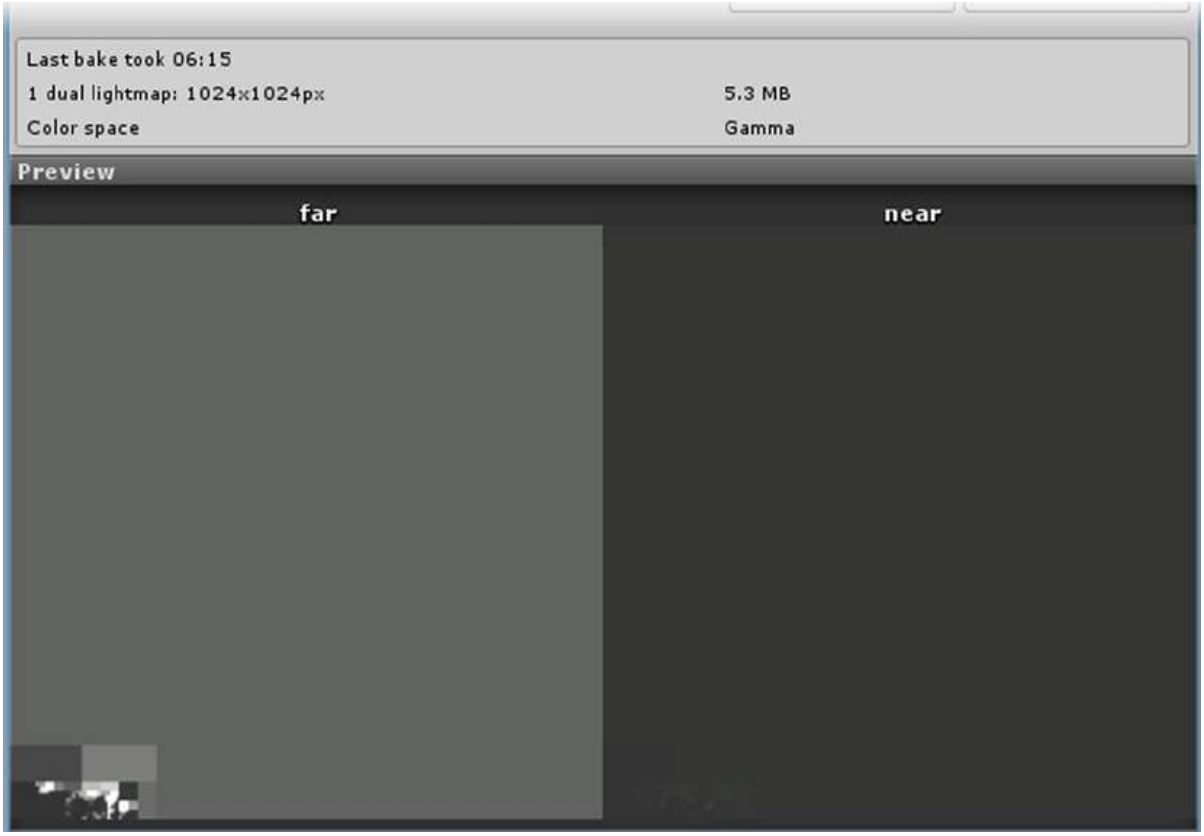

Figure 4-40. The generated maps in Unity Pro

If you toggle the Use Lightmaps option off and on in the Lightmap Display, you might notice a slight change in a few of the plants where they have cast shadows on themselves. None of the structures are receiving shadows, and the Carrots are missing. The status line (below the Project view) reports a problem. This one is just the tip of the iceberg.

7. Click the Console tab (next to the Project tab) to examine the warnings (Figure 4-41).

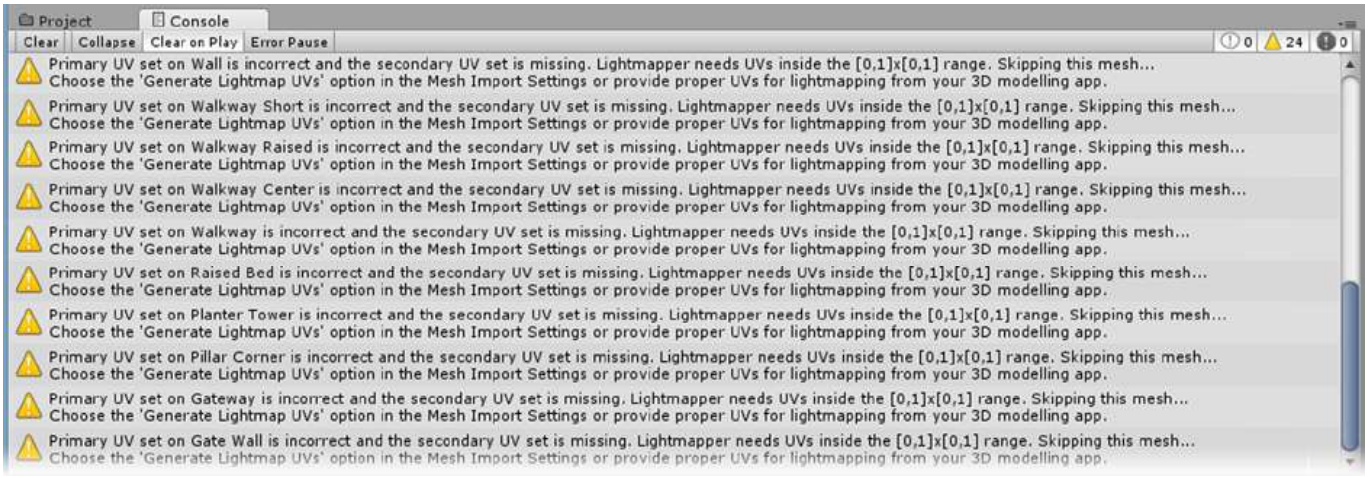

Figure 4-41. *The Lightmapping warnings for many of the objects*

Most of the objects are reported as missing a secondary set of UVs, or mapping coordinates. In the case of the garden structures, the texture has been tiled to increase detail without creating a very large texture map. In Lightmapping, each face must occupy a unique place on the texture map, as each has its lighting calculated individually (Figure 4-42, left). Pieces that overlap or spill over the edge of the map will fail this requirement (Figure 4-42, right).

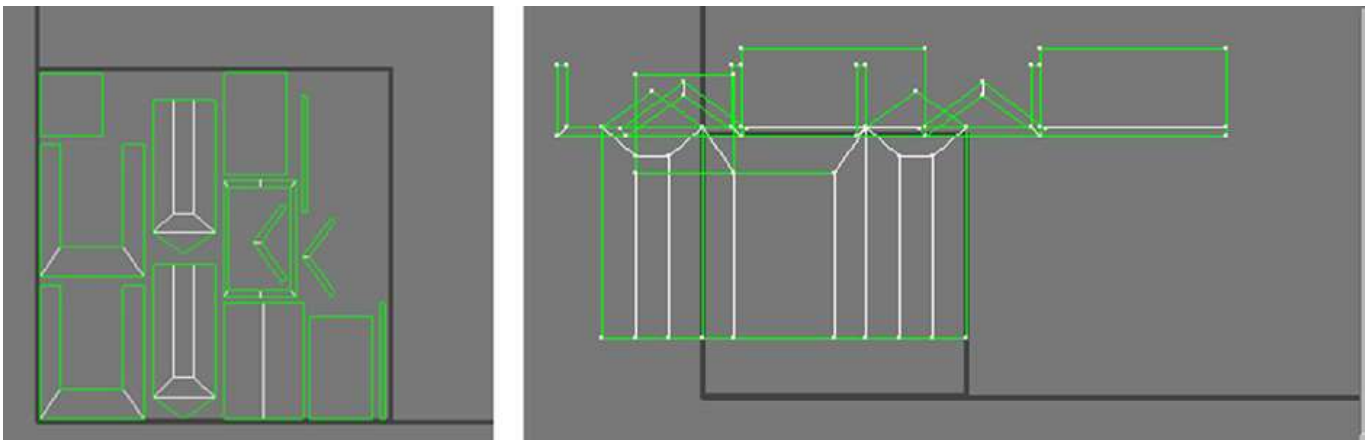

Figure 4-42. *A valid Lightmapping layout (left), and invalid layout (right)*

If you are a 3D artist, you may be familiar with the unwrapping process and UV unwraps in general. If not, don't worry. One of the import options is to generate a second set of UVs. These will automatically be laid out correctly for Lightmapping. Unity, in case you are interested, supports only two sets of mapping coordinates. Let's see about fixing the CornerGarden asset.

1. Click Clear in the Console to get rid of the warnings, and select the Project view again.
2. Select the Corner Garden asset in the Project view.
3. In the Model section, check Generate Lightmap UVs (Figure 4-43).

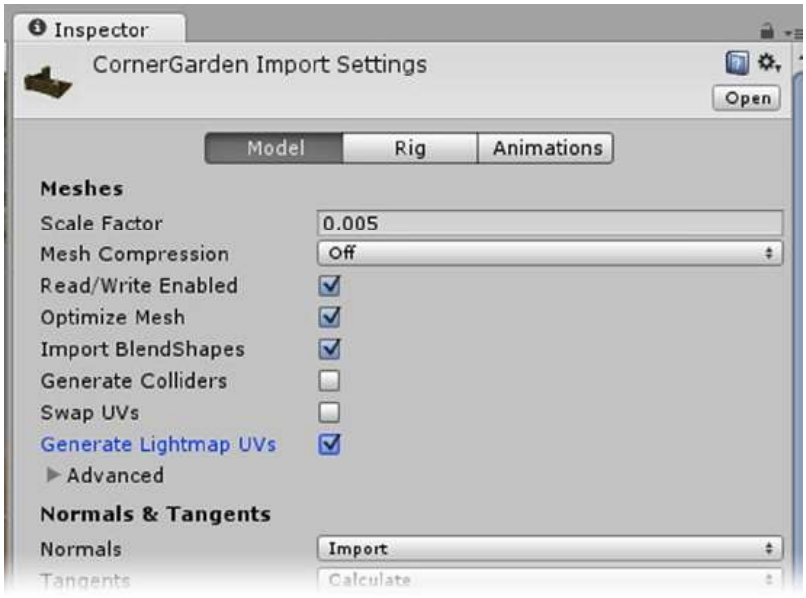

Figure 4-43. *The Generate Lightmap UVs option in the Importer*

4. Click Apply.
5. Rebake the lights in the Lightmapping window.

This time you should see shadows from the Carrots, TomatoPlant, and Radish objects and see several objects atlased in the lightmap (or maps if you have Pro). The Carrots object itself, however, is conspicuously missing (Figure 4-44). It turns out that the Nature shaders use the second UV map for adding wind on terrain plants, so Carrots comes up short one set of UVs . . . in this case, the one used for its regular texture map.

Figure 4-44. *Three shadows for two plants; the Carrots object is missing*

Fortunately, Unity gives you a way to specify whether or not a lightmap should be baked for any given object. While this is a sort of workaround for the Carrots' problem, it allows you to bring in objects with their own Lightmaps already generated. This is especially valuable for architectural models where the lighting is far more complex than you could create in Unity. If you do opt to create Lightmaps outside of Unity, the Lightmaps must be in .ext format. The no-bake flag is also useful for any object that does not have to receive baked shadows. Let's suppress the lightmap on the Carrots.

1. Select the Carrots object in the Hierarchy view.
2. In the Lightmapper, Object section, set the Scale in Lightmap to **0**.
3. Bake the Lights again.

The Carrots object appears along with its shadow (Figure 4-45).

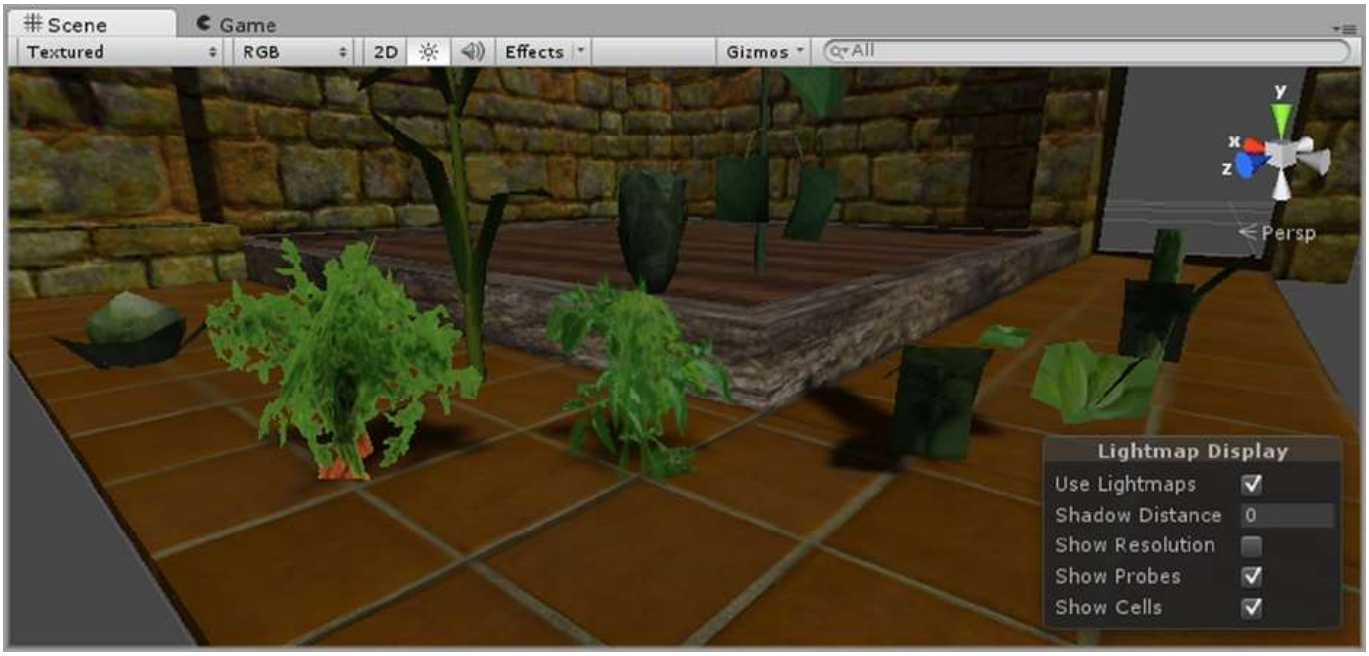

Figure 4-45. The three plants and their baked shadows

4. Set the TomatoPlant's Scale in Lightmap to **0** as well and rebake.

If you are using Pro, it's time for some decisions. With Deferred Lighting, objects within the Shadow Distance will have dynamic shadows. Anything outside of the distance uses the baked shadows. The problem is that Transparent shaders do not cast dynamic shadows.

1. If you have Pro, switch to Deferred Lighting for a moment with Edit, Project Settings, Player, Other Settings, and select Deferred Lighting for Rendering Path.
2. In the Lightmapping Display in the Scene view, turn the Shadow Distance back up to **20** or so until you can see dynamic shadows from the other objects.

The Tomato shadow disappears. If you remember, it is using the Transparent shader. So you have two options. You can change the TomatoPlant's shader to the Nature/Soft Occlusion Leaves shader, or you can use the Forward Rendering Path and give up dynamic shadows. For this very small garden area, the dual lightmaps and transition between dynamic and baked shadows are hardly worthwhile. Because you will have a large population of transient objects, you will do away with baked shadows altogether and rely on dynamic shadows.

3. Return Rendering Path to Forward if you are using Pro.
4. In the Lightmapper, clear the lightmaps to get back to dynamic shadows.
5. Change the TomatoPlant's shader to Nature/Tree Soft Occlusion Leaves.

There are six more plants that must use a two-sided, Transparent shader: Carrots, Kale, Radish, Seedling, Sprout, and the CornStalk's CornTassel material.

6. Change those objects' materials to use the Nature/Tree Soft Occlusion Leaves shader.

When you use any of the the Nature shaders, the objects' assets would have to be added to a folder with "Ambient-Occlusion" in its name in order for shadows to be calculated on them if they were being used as terrain Trees or Detail Meshes. Those shadows are added as vertex color, darkening the object when its vertices lie within a shadow cast by another object. Because the garden plants will be dynamically planted at run-time and are not Detail Meshes or Trees, they will not be receiving shadows, so you can turn the ambient occlusion settings down for each. If the term "ambient occlusion" is unfamiliar to you, think of it as the darkness in cracks, small holes, and other surface areas where light is occluded. The ambient occlusion settings will produce undesirable results on plants that are not being used as terrain Trees or Detail Meshes.

7. Turn the Ambient Occlusion and Direct Occlusion values to 0 on each of the materials using the Nature/Tree Soft Occlusion Leaves shader, and adjust the Base light values until you are happy with them.

The remainder of the objects don't require opacity but a few should have a double-sided shader. At this point, however, you may decide that a bit of specular highlight has a better visual impact than having all of the leaves render on both sides. The corn leaves in particular may be better off. The SunFlower has duplicate geometry for the backside of its foliage to work around the shortcomings of the Specular shader.

8. Assign the Specular shader to the Cabbage, CornStalk, CornYoung, and SunFlower.

9. Adjust the Specular Color and Amount to your taste.

Specular shader helps to give the plants a more realistic look than the Nature shaders and can also receive and cast dynamic shadows. To have a shader that includes double-sided, transparency-respecting, specular highlighting *and* casts shadows requires a lot of resources to render, so it's not surprising that Unity doesn't ship with one.

10. Clear the console.

Creating Prefabs

Having spent a bit of time processing your imported assets, both textures and models, you've probably realized that a lot of the setup, because it was done in the Hierarchy view, will have to be redone if you wish to re-use the objects in a different scene. To avoid doing so, you can create *prefabs* from all of the objects you are likely to use again. Prefabs can contain almost all information an object or objects require to function correctly in your scene. They allow you to quickly re-use assets in multiple scenes or levels and also to instantiate them into a scene at any given time.

1. Switch the layout to the 2 by 3 setting, and change the Project view back to Single Column Layout.

2. In the Project view, create a new folder and name it **Prefabs**.

3. In the Hierarchy view, open the CornerGarden group and drag the Water Spout onto the Planter Tower.
4. Agree to losing the prefab.
5. Now drag the Planter Tower into the Prefab folder in the Project view, and open the Prefabs folder to see the new Planter Tower prefab with its blue cube icon.

In the Inspector, you will see that the object and its child have come in with collider and materials just as they were in the scene (Figure 4-46).

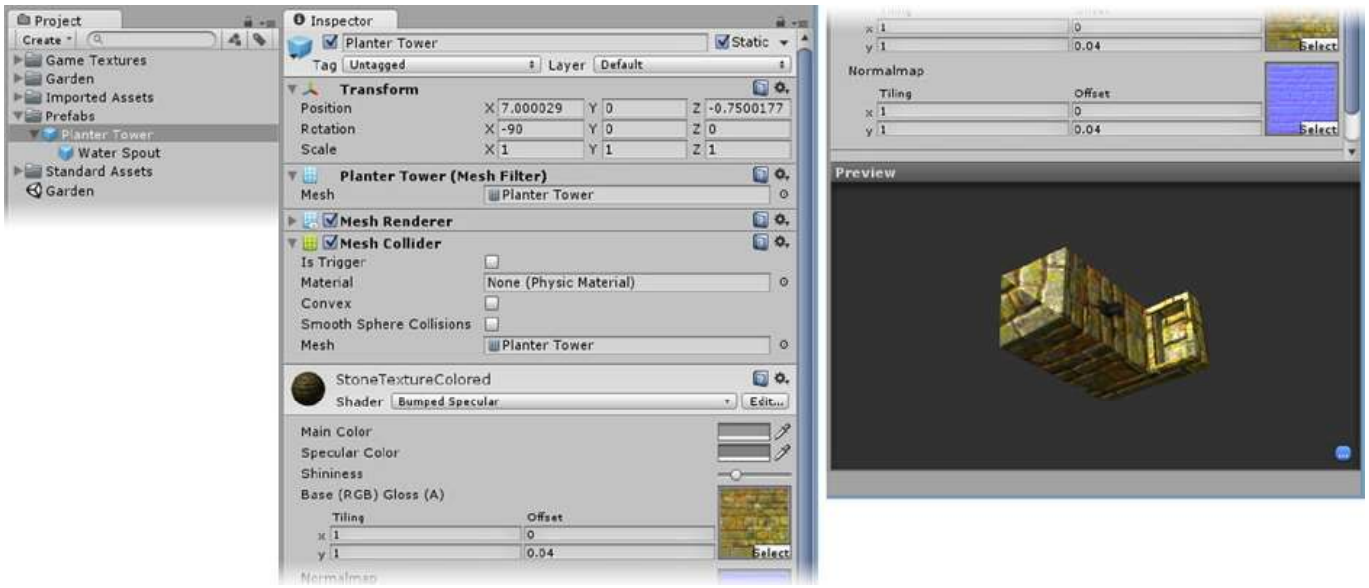

Figure 4-46. The new PlanterTower prefab

6. Make two folders in the Prefabs folder, **Structures** and **Plants**.
7. Drag the Planter Tower into the Structures folder.
8. Drag each of the other CornerGarden objects into the Structures folder.
9. Drag each of the plants into the Plants folder.

Notice that as you drag the objects in, the names go blue in the Hierarchy view, indicating they are prefabs (Figure 4-47).

Figure 4-47. The new prefabs

Unity's Asset Store

These days, no chapter on assets would be complete without a trip to Unity's Asset Store. If you are artistically challenged, you may find a wealth of assets, both free and for purchase. If you are at the prototyping stage of your game, you may find it advantageous to spend a small amount of money for proxy objects to test ideas and functionality. This can help save days of work by artists.

The drawback, of course, to buying assets available to anyone is that your game will not look unique. Worse yet, if the art assets you chose to use were already used on a less-than-stellar game, it could impact the sales and reception of your own game just by association. A prototype using recognizable, existing assets may give the impression that you lack imagination, so always be sure to make it clear where and why they were used.

Scripting assets, on the other hand, can be a big help when there is no need to re-invent the wheel. The caveat to this one is that if you don't know enough scripting to understand how they work in the first place, you may have problems integrating them into your own project.

All warnings aside, the fun part about the Asset Store is that you can shop for assets directly in the Unity editor. Because "free" rarely includes the rights to redistribute other people's work (in any field), you won't be directly using assets from the Asset Store in the book's project. You will, however, have the opportunity to test out the asset acquisition process without worrying about impacting the finished chapters included in the book's resource downloads.

The Asset Acquisition Process

To use the in-editor functionality, you will want to switch to the default layout again.

1. Switch the layout to the Default layout.
2. From Favorites, select All Models and click on the Asset Store text (Figure 4-48).

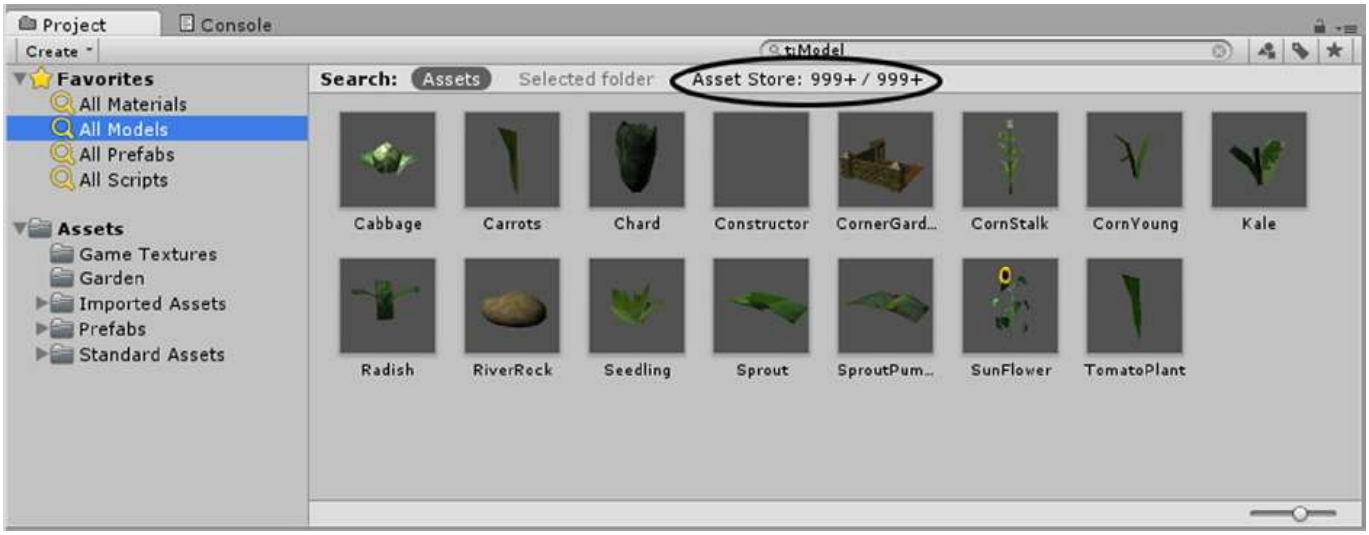

Figure 4-48. *Accessing the Asset Store through the Project view's Favorites*

3. In the search field, type in **'bench'**.

You may or may not see anything useful turn up.

4. Type in **'bench t: Model'**.

This time you should see several models in the Free Assets section and a few hundred in the Paid Assets section.

5. Further refine the search by typing **'stone_bench t:Model'**.
6. In the Paid Assets, click on the thumbnail.

In the inspector, you will see the asset's relevant information (Figure 4-49). If you click the Open Asset Store button, you will find that the curved stone bench is part of a collection that contains several textures and two different configurations of the bench.

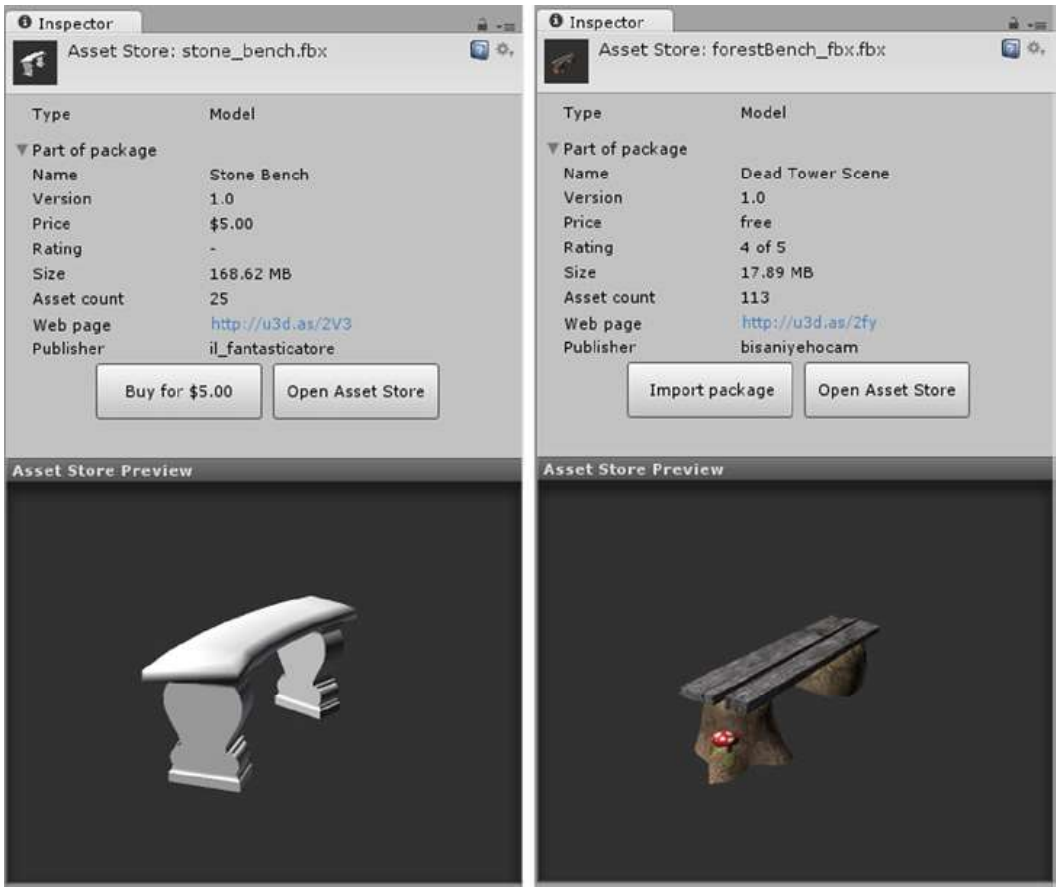

Figure 4-49. *A Paid Asset in the Inspector (left), and a Free Asset (right)*

7. In the Free Assets section, click on a bench that looks promising.

Instead of a "buy" button, you get an "Import package" button.

8. If you wish, click on the "Import package" button to bring the free assets into your scene.

The asset will be downloaded, and you will then get the familiar Import dialog.

9. Click Import if you wish to see how the assets are brought into your package.

Due to Unity's current policy favoring "collections" rather than individual items, you may end up bloating your project file just to use one or two items from a large collection. You may be able to separate them out if they have individual FBX files. The procedure would be to make a new prefab of just the items you want, export it as a Unity package, delete the Asset Store assets, and import the stripped-down package. If you have access to a DCC program, you could open the FBX file and further separate just the assets you want.

Once you've checked them out, feel free to delete the Asset Store assets. To avoid further file bloat, you will use the stone bench provided for you in the Chapter 4 Assets folder.

10. Import the StoneGardenAccessories.unitypackage from the Chapter 4 Assets folder.

You will find a folder for the Stone Garden Accessories that contains a bench and various pots (Figure 4-50).

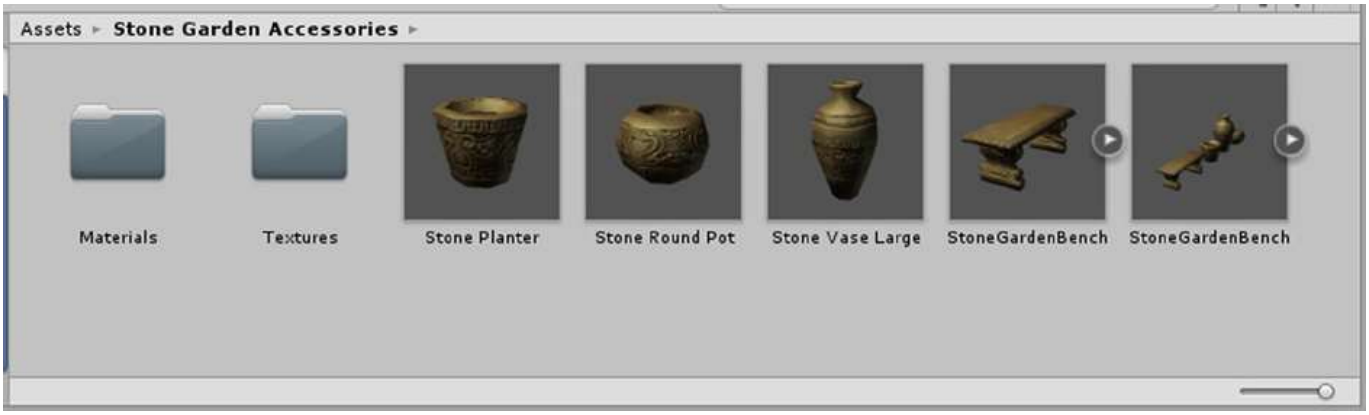

Figure 4-50. The new Stone Garden Accessories in the Project view

11. Drag the Stone Garden Bench prefab into the scene (Figure 4-51).

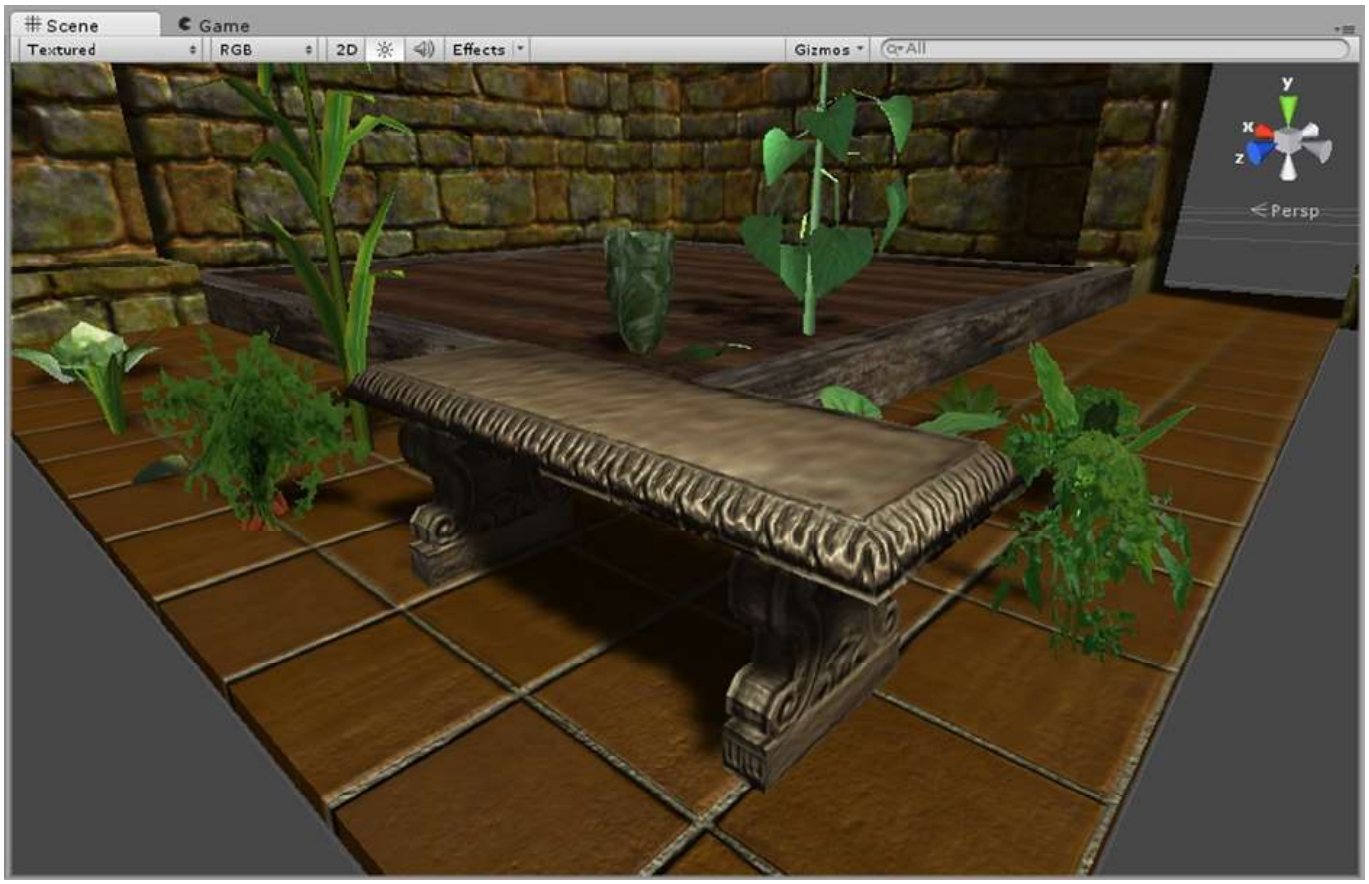

Figure 4-51. The new bench asset in the scene

There will be a few more assets to bring into your scene as the game progresses, but they will come in as Unity packages and will not require extra processing.

Summary

In this chapter, you learned that although Unity supports many file types, both generic and application-specific, the underlying format is .fbx. You found that there are trade-offs to importing directly from your DCC applications such as 3dMax, Maya, or Blender. Saving files directly into Unity is quick, but unless you are using versioning software, you risk not being able to revert to earlier versions of your assets. Files created in proprietary applications may also require licensed versions installed on the machines before Unity can read them. To make sure the assets can be used by anyone for Unity, model assets files should be saved in .fbx format.

With Imported texture assets, you found that Unity can add grayscale alpha channels and generate normal maps. You discovered that it was easy to keep the textures in your favorite format and size because Unity lets you specify a maximum size in scene, and it automatically converts textures to an appropriate .dds format for in-game use.

Once your textures were safely in the project, you brought the model assets in. You found that Unity generated materials using the main texture for the various models. Scale Factor is the first thing to check when importing assets. A quick way to do that is to create a Cube with its default size of 1 meter cubed, for comparison. You discovered that models come in with generic Animator rigs that can be removed by setting the Animation Type to None. Colliders you found, when auto-generated by Unity on import, were always Mesh Colliders. After examining each of the structures included in the CornerGarden asset, you determined that most would be more efficient with standard primitive colliders.

Next you learned how Unity batches" objects to be able to combine draw calls. The main requirement was that the objects be marked as Static. This, you discovered, was used for both batching and Lightmapping. With the structures, you had the chance to see how Unity shaders often use a texture's alpha channel for parameters other than transparency. Upon processing the plants, you had to decide between several shaders, but found that plants that required double-sided shaders and transparency were limited to only two choices. Of the two, only one cast dynamic shadows, and the other was likely to disappear when using Deferred Lighting (a Pro-only feature).

With all of the processing for the imported objects finished, you created prefabs of each of the objects. This, you found out, would insure that you would not have to go through the setup each time you wanted to use them in other scenes or instantiate them during runtime. Finally, you learned how to locate and load an asset directly into your scene via the Unity Asset Store.