CHAPTER 6

Using Spatial Mapping

In this chapter, you'll learn how to use one of the most defining features of Windows Mixed Reality headsets like the HoloLens: spatial mapping. You'll learn how to apply spatial mapping in Unity using the HoloToolkit and unwrap some neat tricks that you can do with spatial mapping. You'll learn how to identify walls, floors, ceilings, and chairs. You'll also learn how to anchor digital objects to your physical environment, and how to save those anchors so that your digital objects will persist where you left them, even after closing and re-opening your app.

What Is Spatial Mapping?

Devices like the HoloLens are constantly tracking their environment and building a 3D model of the area that they're in. This is called *spatial mapping*. Without spatial mapping, holograms wouldn't be able to be set on floors and tables, or be pinned to walls. Objects in other rooms would still be visible, degrading the user's experience.

Spatial mapping is important for several reasons:

- *Occlusion*: This tells the HoloLens which holograms to hide from view. For example, if you place a hologram in your hallway and then walk into another room, the spatial map of that room's walls will prevent you from seeing the hologram in your hallway. If there were no spatial map, you'd see the hologram as if it were visible through your walls, causing an unrealistic experience.
- *Placement*: This allows users to interact with the spatial map—for example, to pin items to your walls, allow characters to sit on your sofa (as seen in Microsoft's Fragments app), or automatically decorate your surroundings.
- *Persistence*: This allows for *holographic persistence*, which is the ability for holograms to stay where the user left them, even after turning off the device. Your HoloLens will (remarkably) be able to remember your space and restore any holograms you had placed in that space.

© Sean Ong 2017
S. Ong, *Beginning Windows Mixed Reality Programming*, DOI 10.1007/978-1-4842-2769-5_6

- *Physics*: This allows objects to collide with or bounce off your walls, furniture, and floors, resulting in a more realistic experience.
- *Navigation*: Use gaze to allow game characters and other holograms to follow along mapped surfaces.

For more information on spatial mapping and the sensors involved, see Chapter 1.

Spatial Mapping Tutorial

In this section, I walk you through setting up some basic spatial mapping capabilities. I show which elements from the HoloToolkit are needed to enable spatial mapping and provide some tips for a good experience.

Step 1: Set Up Unity Scene

This tutorial uses a test scene from the HoloToolkit. If you haven't done so already, be sure to set up Unity for Mixed Reality development as described in Chapter 4. Refer to Chapter 4 for a refresher on how to run HoloToolkit test scenes in Unity.

Find the TapToPlace test scene (or TapToPlace.unity) in your Project panel by using the search bar or find it within the folder structure. Drag the test scene into your Hierarchy, as shown in Figure 6-1. Be sure to unload (disable) all other scenes that you might have open.

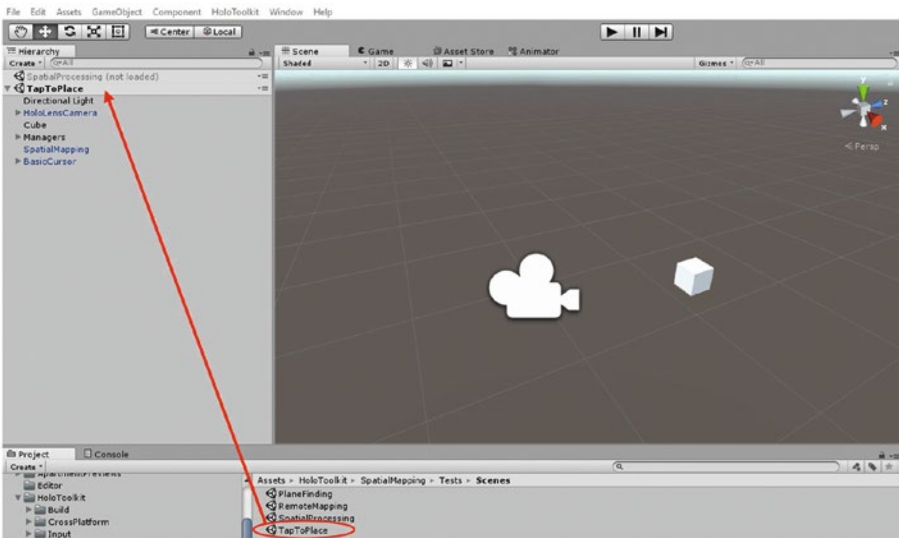

Figure 6-1. *Open the TapTopPlace scene from the HoloToolkit to explore a basic implementation of spatial mapping*

Step 2: Try It Out

The next step is to try it out by clicking the play button.

If you have a HoloLens or similar Windows Mixed Reality device, I highly recommend using Unity's holographic remote to device feature (see Chapter 3 for a discussion and tutorial on holographic remoting) in order to experience spatial mapping of your physical environment. You may also deploy the app to your HoloLens.

If you don't have a device, or prefer not to use it for this test, be sure to use Simulate in Editor with Unity's holographic emulation (again, see Chapter 3 for more information on this) in order for spatial mapping to work.

■ **Tip** When using the Simulate in Editor mode of Unity's holographic emulation, Unity will load in a 3D model of a room or area that you can use to test your spatial mapping capabilities without using a headset. Unity provides several different rooms and spaces that you can use. To chose a 3D space, use the Room drop-down menu in the holographic emulation window.

After clicking the play button, you'll see the scene's cube in your area, but you won't be able to see the spatial map. After tapping the cube, the spatial map will appear, and the cube will follow your gaze, as shown in Figure 6-2. If wearing the HoloLens, you'll see the spatial map well aligned to your physical surroundings, as shown in Figure 6-3. When you tap a second time, the spatial map will become invisible again.

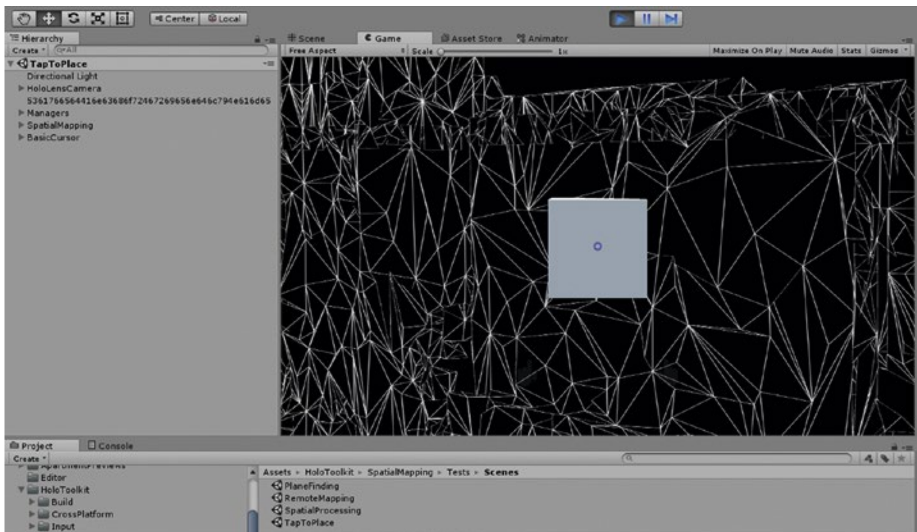

***Figure 6-2.** View of the spatial map, as seen through the Unity Editor*

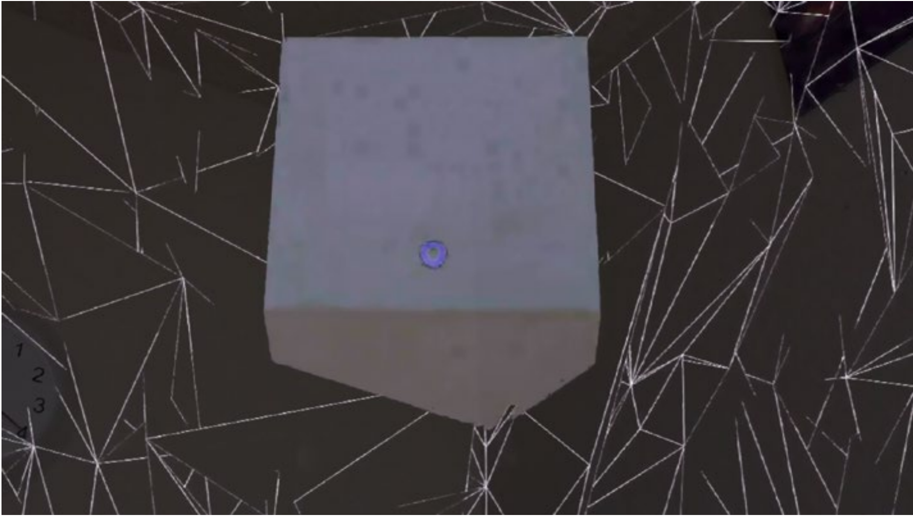

Figure 6-3. *When viewed through the HoloLens, the spatial map will align well with your physical surroundings*

As you can see, the rendering of the spatial map is a collection of vertices, edges, and faces. It looks like a net covering your surroundings (later we'll see how to change the spatial mapping appearance). The 3D object generated by spatial mapping is often called the *spatial mapping mesh*.

Step 3: Understand the Scene

Now that you've had the opportunity to experience spatial mapping, let's dig into our scene to learn about the key components that make spatial mapping possible.

Looking at the scene's Hierarchy, we see some familiar items that we've already learned about in Chapter 5, including the InputManager prefab and the BasicCursor prefab. There is one unfamiliar item in our Hierarchy: the SpatialMapping prefab, as shown in Figure 6-4.

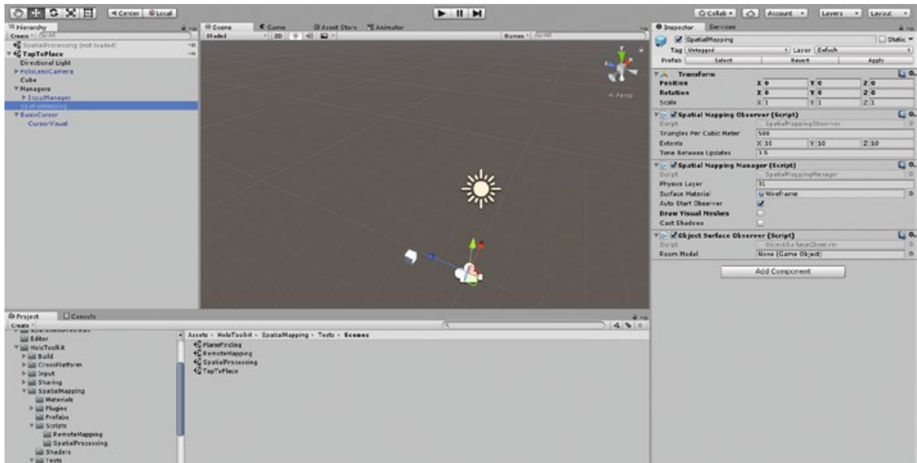

Figure 6-4. *The SpatialMapping prefab is all that's needed to enable spatial mapping in your project*

This small prefab, containing only three scripts, is all that's responsible for spatial mapping. You can easily find this prefab in the HoloToolkit and drag it into your project to enable spatial mapping. This is yet another example of how the HoloToolkit makes it easy for developers to quickly and efficiently set up a Windows Mixed Reality project.

Let's walk through each of the three scripts in the SpatialMapping prefab.

- *SpatialMappingObserver.cs*: This script is responsible for managing the surfaces observed on the HoloLens and renders them so they can be displayed in the scene. You can adjust the resolution of the spatial map in the Inspector panel using the Triangles Per Cubic Meter field. You can also adjust how far out from the HoloLens you want to observe by adjusting the Extents variables, and you can specify how often to process spatial mapping updates using the Time Between Updates field.

- *SpatialMappingManager.cs*: This script allows you to choose to load a saved spatial mapping mesh or collect data in real time from the HoloLens. To help with performance and avoid the processor-intensive task of constantly scanning a room, it can be beneficial to save the current room to memory and only scan occasionally or as needed. In the Inspector panel, you may also select the material to use for rendering the spatial mapping data.

- *ObjectSurfaceObserver.cs*: This script is used when you're not using a HoloLens device for spatial mapping but instead are using a pre-existing 3D model of a room or area within the Unity Editor. You can specify a custom 3D model in the Inspector panel.

In addition to the scripts in the SpatialMapping prefab, the Cube game object also has a script attached to it called TapToPlace.cs, which is responsible for making the Cube interactive and placeable on the spatial mapping mesh. There's also a new script called WorldAnchorManager.cs. If you click the Managers item in the Hierarchy, you'll see this script. I discuss world anchors and spatial anchors in greater depth later in this chapter.

Step 4: Use Spatial Mapping in Your Application

As mentioned in the previous step, enabling spatial mapping in your application is as easy as dragging the SpatialMapping prefab from the HoloToolkit to your project Hierarchy. Simply use the Project panel's search bar to find the SpatialMapping prefab or navigate to it in the directory, as shown in Figure 6-5.

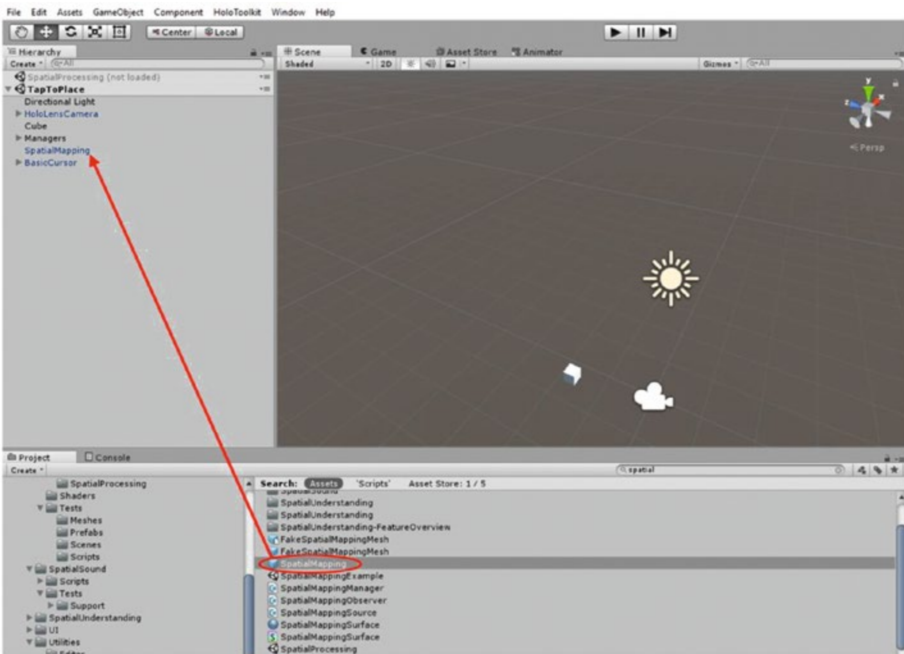

Figure 6-5. *To apply spatial mapping to your application, simply apply the SpatialMapping prefab from the HoloToolkit to your scene's Hierarchy*

You must also enable SpatialPerception to your Unity application by going to Edit ➤ Project Settings ➤ Player ➤ Settings for Windows Store ➤ Publishing Settings ➤ Capabilities. See Figure 6-6 for an illustration of this setting.

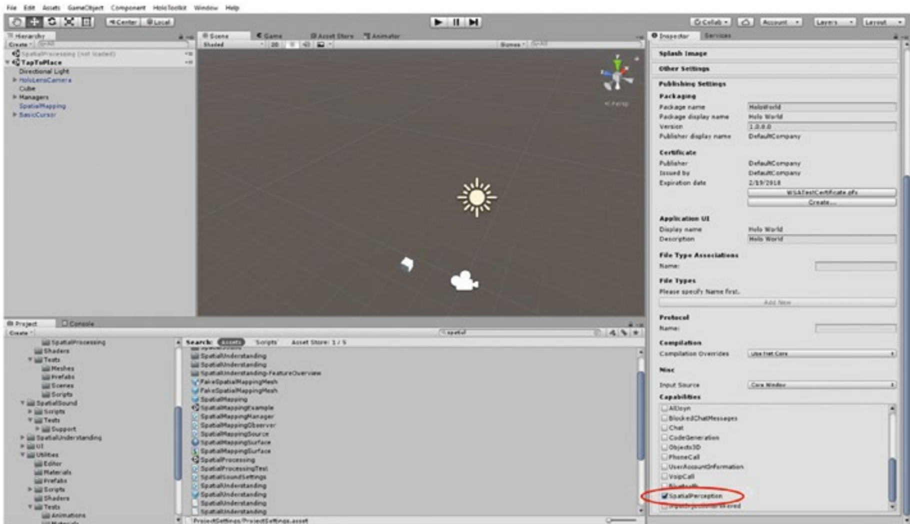

Figure 6-6. *Be sure to enable SpatialPerception in Unity's Publishing settings for spatial mapping to work*

Spatial Plane Finding Tutorial

Rather than just applying a digital mesh "blanket" over physical surfaces, we can leverage the HoloLens' computational power to find planes in our environment. In this section, I walk you through enabling *plane finding* in your application and discuss why plane finding is important.

Step 1: Set Up the Unity Scene

This tutorial uses a test scene from the HoloToolkit. If you haven't already done so, be sure to set up Unity for Mixed Reality development as described in Chapter 4. Refer to Chapter 4 for a refresher on how to run HoloToolkit test scenes in Unity.

Find the PlaneFinding test scene (or PlaneFinding.unity) in your Project panel by using the search bar or finding it within the folder structure. Drag the test scene into your Hierarchy, as shown in Figure 6-7. Be sure to unload (disable) all other scenes that you might have open.

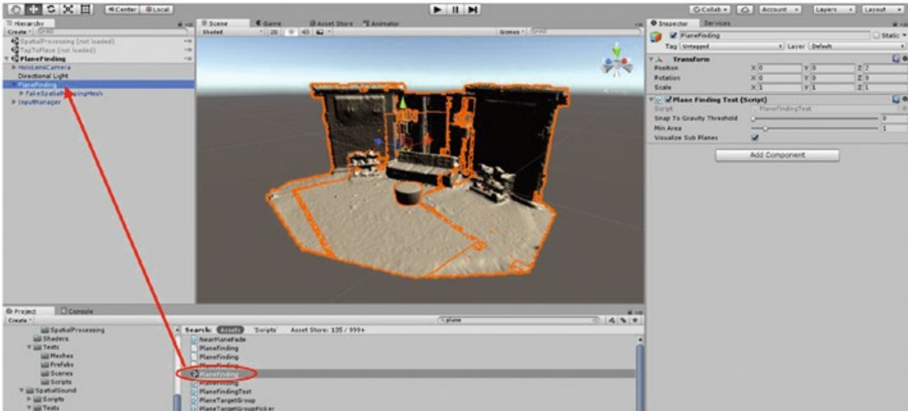

Figure 6-7. *Open the PlaneFinding scene to explore the HoloToolkit's spatial mapping plane finding feature*

Upon loading the scene, you should see a 3D model of a room. You may notice that this scene doesn't use spatial mapping but rather the pre-existing room model. We'll try out plane finding on a real spatial mapping mesh later in this section.

Step 2: Try It Out

Go ahead and click the play button to start exploring the scene. This scene is intended to be experienced within the Unity Editor, so we won't be using the headset.

To see the identified planes, *switch to the scene view while in Play mode*. You will be able to visualize the planes, as shown in Figure 6-8.

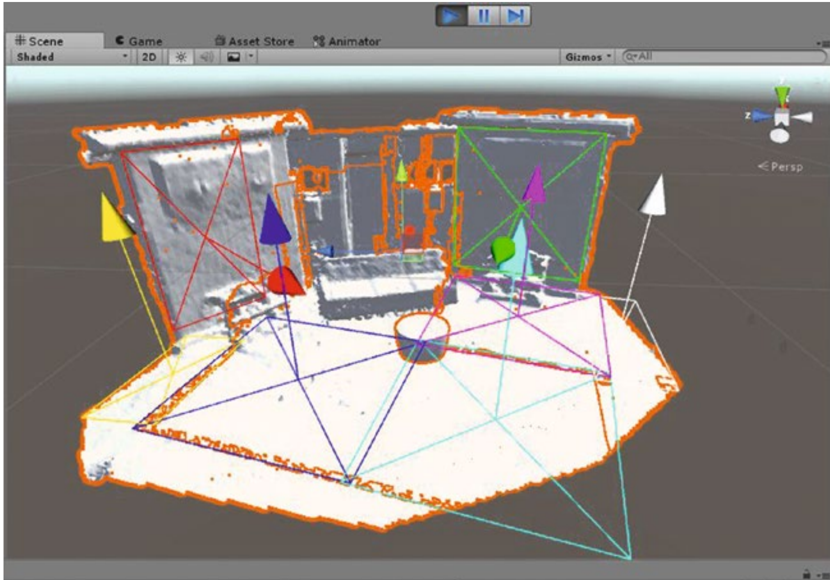

Figure 6-8. *Identified planes can be seen while the scene is running and you're in the scene view*

While in the scene view, feel free to adjust the parameters for PlaneFindingTest.cs in the Inspector panel (you'll need to select the PlaneFinding item in the Hierarchy to see the script).

This scene provides a controlled environment to test plane finding and allows you to carefully explore parameters.

Step 3: Load the Spatial Processing Scene

Now that you've had the opportunity to explore a basic plan finding scene and implementation, let's expand this capability to an actual spatial mapping mesh. Find and load the SpatialProcessing scene (SpatialProcessing.unity), as shown in Figure 6-9, and unload the PlaneFinding scene.

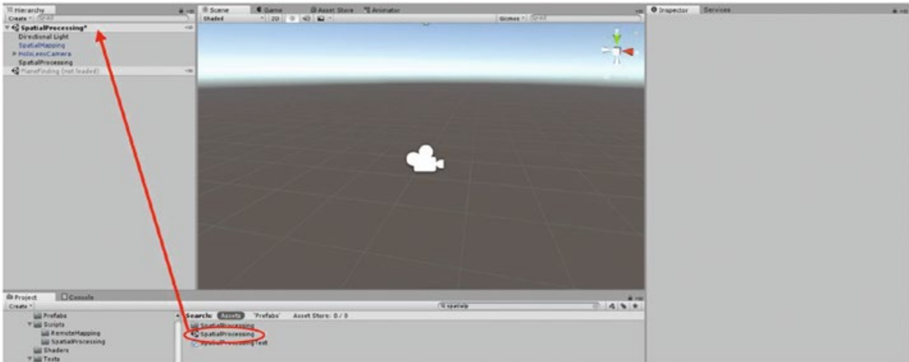

Figure 6-9. *Load the SpatialProcessing scene and disable the PlaneFinding scene*

Step 5: Try Out the SpatialProcessing Scene

Try out the spatial processing scene by clicking the blue play button while using Unity's holographic emulation or deploying to your device. At first you'll see the regular spatial mapping mesh, but after a few seconds the spatial mapping mesh will be replaced with large white rectangles representing the identified planes of the room, as shown in Figure 6-10.

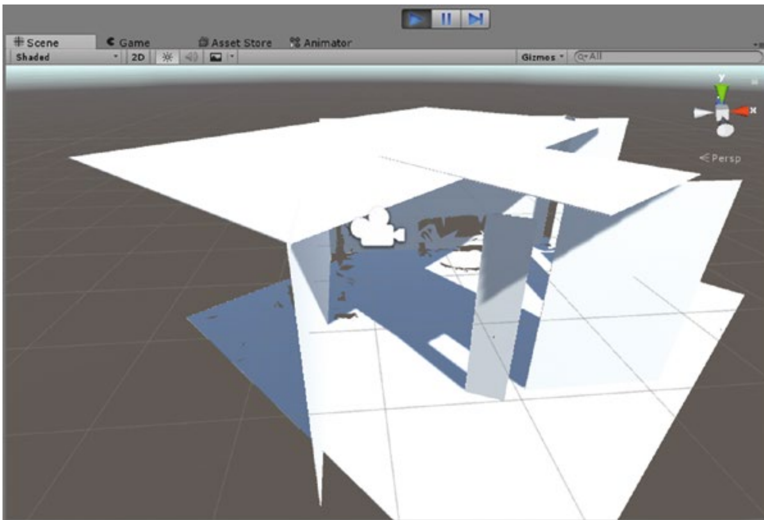

Figure 6-10. *After a few seconds, the spatial processing feature will identify planes and replace the spatial mapping mesh with white planes*

Step 6: Understand the SpatialProcessing Scene

When I first tried the SpatialProcessing scene while wearing the HoloLens, I was overjoyed at seeing how perfectly aligned the planes were to my walls. SpatialProcessing is made possible by the SpatialProcessing item in the Hierarchy, which includes three important scripts, as shown in Figure 6-11.

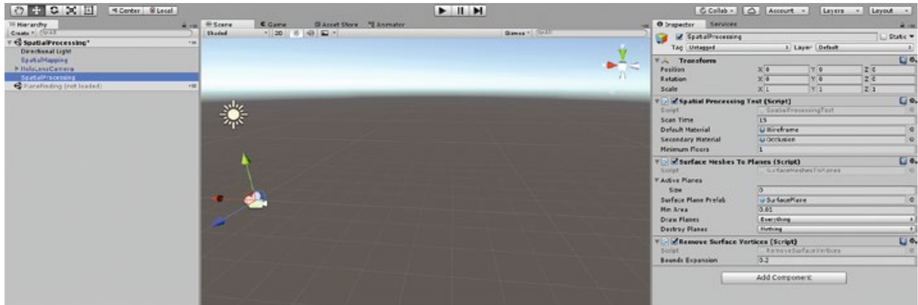

Figure 6-11. *The SpatialProcessing object (highlighted in blue) contains three important scripts, as can be seen in the Inspector panel*

Let's walk through each of these three scripts to see how they come together for visualizing planes in our environment:

- *SpatialProcessingTest.cs*: This script allows you to control a variety of settings related to spatial processing. You'll notice that you can adjust these settings directly from the Inspector panel. The first field is the Scan Time, which is the number of seconds to allow the SurfaceObserver to scan the environment. The longer the scan time, the more time you'll have to build out a good spatial mapping mesh of your environment. After the time expires, this script will stop the SurfaceObserver (that is, stop scanning the environment) and start the mesh processing. This script also allows you to set the default material, which lets you visualize the spatial mapping mesh during scanning. The secondary material allows you to visualize the spatial mapping mesh after scanning is complete. The Minimum Floors field is the minimum number of floor planes needed to exit processing. If you set the minimum number of floors to 1, but no floors are detected, the script will continue trying to find a floor plane.

- *SurfaceMeshesToPlanes.cs*: This script is responsible for finding planes from the mesh and generating planes. In the Inspector, you may set the MinArea, which is the minimum area required before a plane will be created. A larger number means you'll have larger but fewer planes in your scene, whereas a smaller number means you'll create plans for smaller surfaces in your scene. There are also two drop-down lists in the Inspector where you can specify which types of surfaces to draw planes for and which types of surfaces to destroy. The `snapToGravityThreshhold variable` (in the script but not shown in the Inspector) is used to align planes with gravity so that they appear more level. The Surface Plane prefab determines the appearance of the planes. Feel free to edit this prefab if you want to modify the plane appearances—for example, if you want to have the floor planes be a different color from the wall planes.
- *RemoveSurfaceVertices.cs*: This script is responsible for removing vertices (removing parts of the spatial mapping mesh) that fall within boundaries that you specify. You may want to remove vertices if you need holes in your spatial mapping mesh or if you want to reduce polygon count in order to improve the performance of your application. In our SpatialProcessing scene, the SpatialProcessingTest.cs script uses the planes generated by SurfaceMeshesToPlanes.cs as the boundaries. It sends these boundaries to the RemoveSurfaceVertices.cs script, which then removes the spatial mapping mesh near the generated planes. The Bounds Expansion parameter that is visible in the Inspector allows you to expand the boundaries you provide, to remove more vertices around the boundaries.

■ **Tip** Because the SurfaceMeshesToPlanes script allows you to specify plane types (walls, floor, celling, table, and so on), you can selectively send some of these planes to the RemoveSurfaceVertices script. This is useful if you'd like to selectively remove parts of your spatial mapping mesh—for example, if you'd like to remove only your ceiling.

Step 7: Use Spatial Processing in Your Application

To enable spatial processing in your application, you need to do the following:

- Make sure to enable spatial mapping, as described in the previous tutorial.
- Make sure to enable spatial perception, as described in the previous tutorial.

- Find and add the SpatialProcessingTest.cs script to your Hierarchy to control scan times and mesh visualization.
- Find and add the SurfaceMeshesToPlanes.cs script to your Hierarchy. Adjust settings in the Inspector panel as needed by your project.
- Optionally, you may also find and add the RemoveSurfaceVertices.cs script to your Hierarchy to remove parts of your meshes that you replace with a plane.

Spatial processing is excellent for scenarios where you need to identify planes without necessarily visualizing them. For example, if I need to place a hologram on the floor, I can use spatial processing to identify and create an invisible floor plane and then proceed to place holograms on the floor. Other examples include placing objects on walls, hanging holograms from the ceiling, calculating headset height above the floor, and more.

Occlusion Tutorial

In this section, I walk you through implementing occlusion to your spatial mapping mesh. As mentioned, occlusion allows parts of objects that are fully or partially behind walls and other surfaces to become invisible, just as they would in the physical world. This increases your holograms' realism and improves your user's experience.

Step 1: Load the TapToPlace Scene

Let's reload the TapToPlace scene that we started with at the beginning of this chapter, shown back in Figure 6-1. Feel free to try out the application again. You will notice that the cube is visible, regardless of whether it's in your space or behind a physical wall.

Step 2: Apply Occlusion

Next, we want to apply a new material that will allow our spatial mapping mesh to block objects behind it while appearing invisible when viewed through the HoloLens or other device. The HoloToolkit provides a useful item to achieve this. Browse or search for the Occlusion material in your HoloToolkit folders within the Project panel. Once you locate it, drag it to the Surface Material field of the SpatialMappingManager.cs script within the Inspector panel, as shown in Figure 6-12.

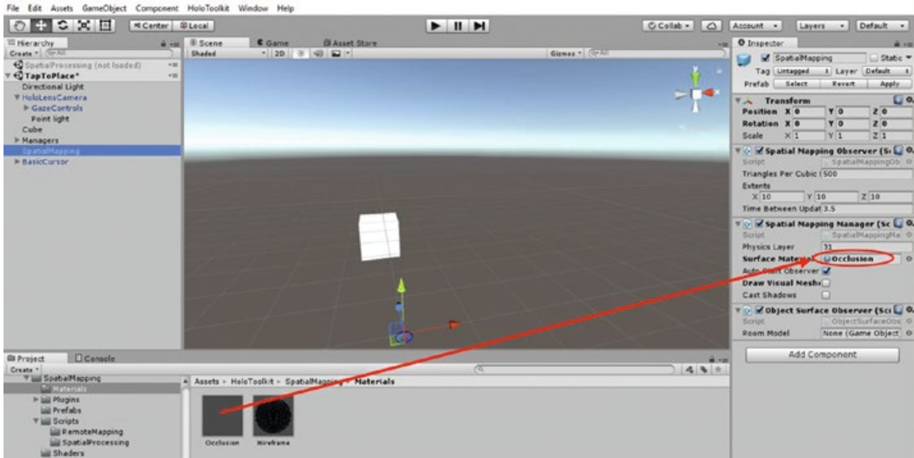

Figure 6-12. *Apply the Occlusion material to the Surface Material field of the Spatial Mapping Manager script*

Step 3: Try It Out

As before, use holographic remoting to test the app with the new Occlusion material. Initially, the occlusion material won't be rendered until you tap the cube. While in Placing mode (with the cube following your gaze), the occlusion material will be rendered. Because the material is transparent, you won't directly see the spatial mapping mesh, but you will see that part of the cube will be occluded by its environment.

As soon as you release the cube (by tapping it again), the spatial mapping mesh will no longer be rendered, and the cube will no longer be occluded by your surroundings. Go ahead and manually turn on occlusion while the application is running in Unity by checking the Draw Visual Meshes box for the SpatialMappingManager.cs script in the Inspector panel, as shown in Figure 6-13.

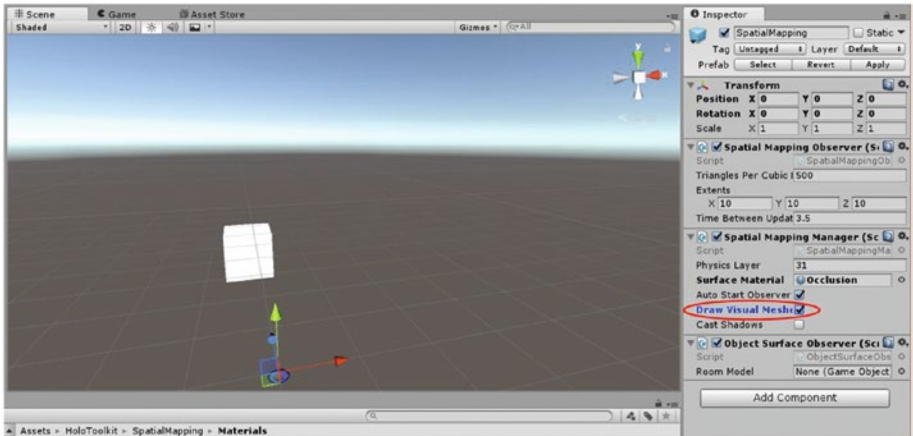

Figure 6-13. *While the application is being simulated in Unity, check the Draw Visual Meshes checkbox to force the occlusion material to be rendered*

As you can see from my tests, the cube was fully visible when there were no obstructions between me and the cube (Figure 6-14), but the cube was partially visible when obstructions were present (Figure 6-15).

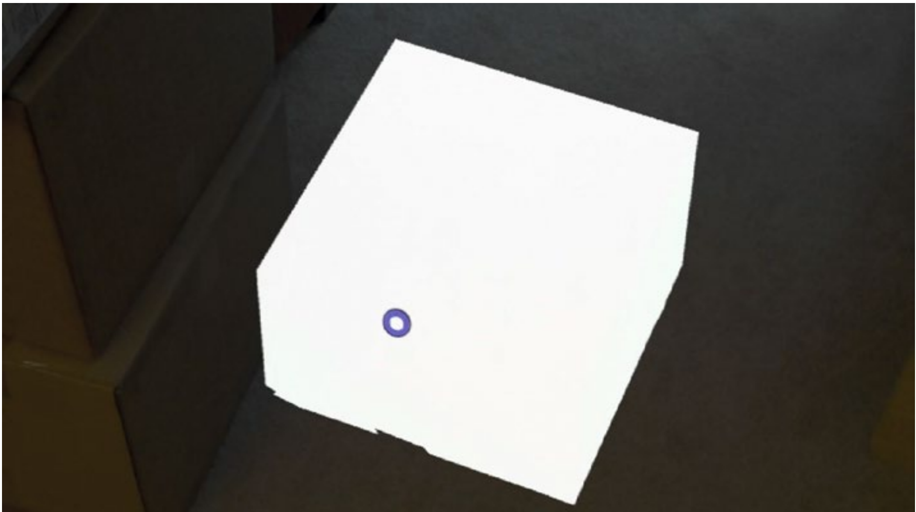

Figure 6-14. *The white cube appears full when there are no obstructions*

Figure 6-15. *When obstructed (in this case by boxes), only the unobstructed part of the cube is visible*

Because no processing or smoothing of the spatial mapping mesh occurs in this scene, you'll notice some imperfections in the spatial mapping mesh, as can be seen in Figure 6-15, with the sharp artifacts. Replacing the spatial mapping mesh with smooth planes and other smoothing techniques can help with occlusion and visualization.

Step 4: Use Occlusion in Your Application

Occlusion is an essential part of Mixed Reality development, especially for projects that utilize spatial mapping. Without occlusion, distant holograms in other rooms or behind objects would still be visible, causing the experience to be confusing and unnatural.

To apply occlusion to your project, you need to render your spatial mapping mesh using the Occlusion material found in the HoloToolkit, as shown in Step 2 of this section.

A more advanced treatment of the spatial mapping mesh may utilize multiple materials throughout your app. For example, your application may start out using one material that's visible (such as during a room-scanning phase of your app) and then later switch to the invisible occlusion material.

You may actively switch between materials for spatial mapping when scripting by using the `SpatialMappingManager.SetSurfaceMaterial()` function. See the following code for an example of this implementation:

```
if (condition == true)
{
    SpatialMappingManager.SetSurfaceMaterial(surfaceMaterial);
}
```

In the preceding code, `surfaceMaterial` is a previously assigned material. You may, for example, declare it as `public Material SurfaceMaterial` and drag and drop a material using the Unity Editor's Inspector panel, as we did earlier in this tutorial.

Spatial Understanding Tutorial

In this section, I walk through how to enable spatial understanding for your application. One of the most powerful examples that the HoloToolkit has to offer is the Spatial Understanding example. *Spatial understanding* can be thought of as a much more powerful version of the plane finding feature covered earlier in this chapter.

Here are a few things that spatial understanding allows you to do:

- Place objects on floors, ceilings, and walls
- Place objects in the air away from you or near you, without touching walls
- Place objects on the floor away from you or near you
- Find the largest wall and place objects on it
- Find sittable surfaces (so you can have characters sit on anyone's chair)
- Identify chairs and couches
- Identify large empty surfaces
- Allows users to "paint" their spatial mesh to limit the scanned area
- Smoothe the spatial mapping mesh

Step 1: Set Up the Unity Scene

This tutorial uses a test scene from the HoloToolkit. If you haven't done so already, be sure to set up Unity for Mixed Reality development as described in Chapter 4. Refer to Chapter 4 for a refresher on how to run HoloToolkit test scenes in Unity.

Find the SpatialUnderstandingExample test scene (or SpatialUnderstandingExample.unity) in your Project panel by using the search bar or find it within the folder structure. Drag the test scene into your Hierarchy, as shown in Figure 6-16. Be sure to unload (disable) all other scenes that you might have open.

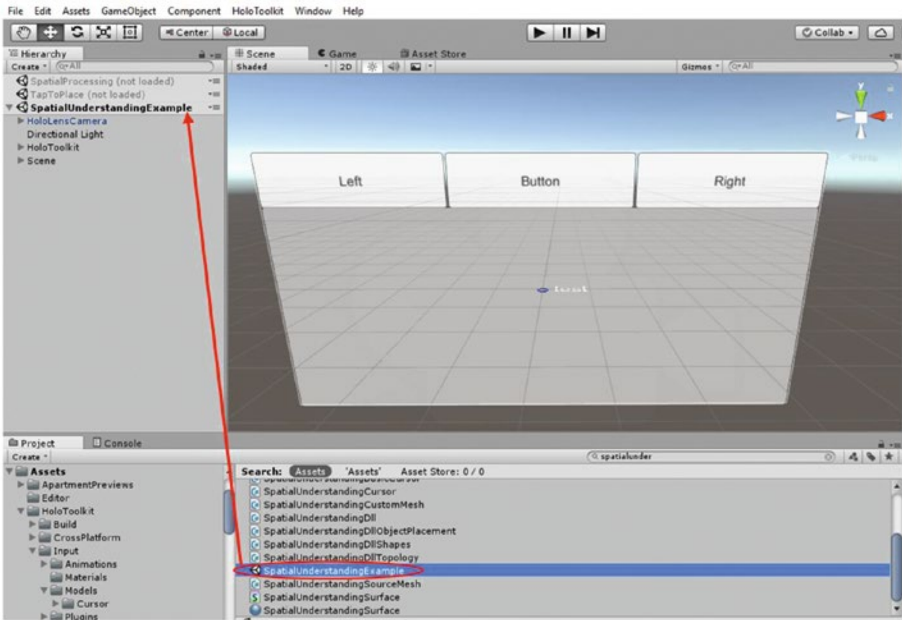

Figure 6-16. *Open the SpatialUnderstandingExample scene to explore the HoloToolkit's spatial understanding features*

Step 2: Try It Out

For this example scene, I highly recommend taking the time to build and deploy to your HoloLens or similar headset. Spatial understanding is an awe-inspiring example provided by the HoloToolkit and is best experienced on your device, without lag or limitation, and in a larger area of your home or office with interesting features nearby (table, chair, open area, walls, ceiling). If you prefer, you may still try out this scene from within the Unity Editor using holographic emulation or remoting to your device.

The application will first ask you to scan your environment. You'll immediately notice a very smooth implementation of the spatial mapping mesh, with well-leveled meshes on walls, floors, and ceilings. See Figure 6-17 for an example of the spatial mapping mesh used by spatial understanding.

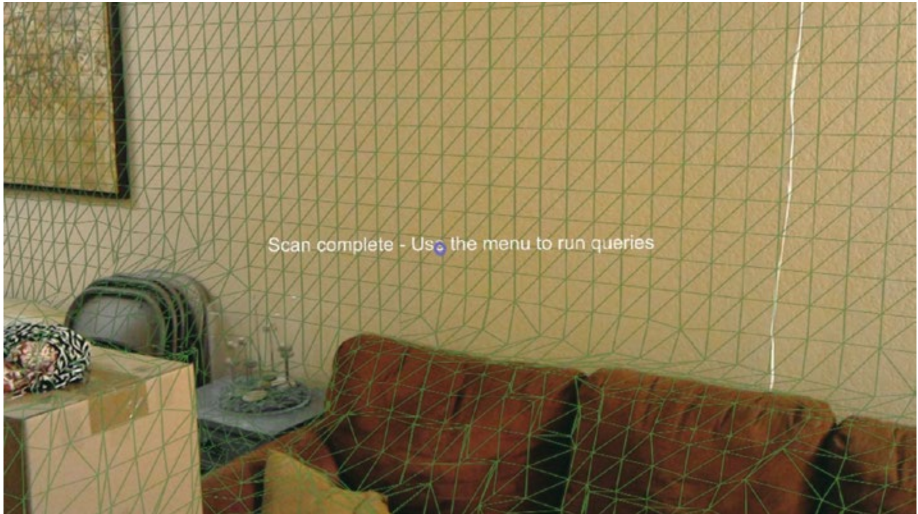

Figure 6-17. *The spatial mapping mesh processing by spatial understanding is remarkably smooth*

Next, you may notice that a menu item has been placed on a wall near you, as shown in Figure 6-18. I recommend exploring all buttons and tabs in this menu to gain familiarity with what spatial understanding has to offer.

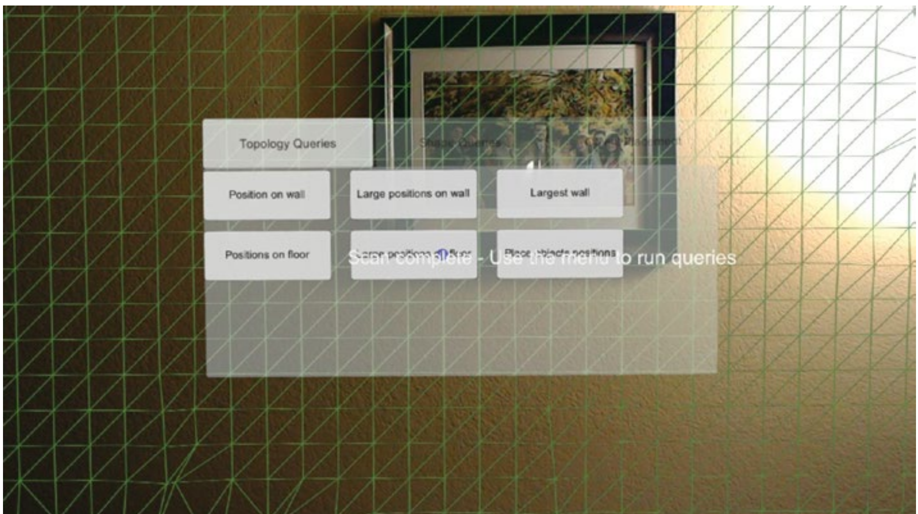

Figure 6-18. *Explore all buttons and tabs offered in the spatial understanding menu. Prepare to be amazed!*

When selecting buttons, a typical experience will involve rectangular objects flying from the menu to recognized surfaces around your mapped room.

Step 3: Use Spatial Understanding in Your Application

Entire chapters could be written about the inner workings of spatial understanding and all the ways developers could leverage it in their applications. In this step, I point out the key components needed to enable spatial understanding in your application and how to get started with it.

Enabling spatial understanding involves the following:

- Making sure spatial mapping is enabled, as shown previously in this chapter
- Locating the SpatialUnderstanding prefab and loading it into your scene's Hierarchy

The spatial understanding prefab contains three scripts, as shown in Figure 6-19.

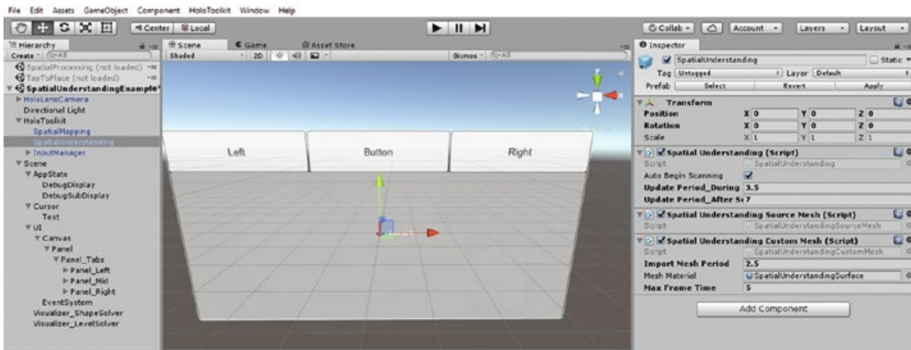

Figure 6-19. *The SpatialUnderstanding prefab contains three important scripts*

In simple terms, here is what each of the scripts are responsible for:

- *SpatialUnderstanding.cs*: This is responsible for managing the spatial understanding scanning process.
- *SpatialUnderstandingSourceMesh.cs*: This is responsible for providing the source (raw) spatial mapping mesh to the spatial understanding feature.
- *SpatialUnderstandingCustomMesh.cs*: This is responsible for generating the custom spatial mapping mesh, during spatial understanding processing and after it's complete.

To learn how to utilize all the data generated for the spatial understanding process, you'll want to dig into the following scripts: SpaceVisualizer.cs and LevelSolver.cs. These scripts are not part of the Spatial Understanding prefab or module, but they are part of the

Spatial Understanding Example scene, provided to show developers how to utilize spatial understanding data.

Let's look at one example code snippet from SpaceVisualizer.cs:

```
public void Query_Topology_FindLargeWall()
{
    ClearGeometry();

    // Only if we're enabled
    if (!SpatialUnderstanding.Instance.AllowSpatialUnderstanding)
    {
        return;
    }

    // Query
    IntPtr wallPtr = SpatialUnderstanding.Instance.UnderstandingDLL.
    PinObject(resultsTopology);
    int wallCount = SpatialUnderstandingDllTopology.QueryTopology_
    FindLargestWall(
        wallPtr);
    if (wallCount == 0)
    {
        AppState.Instance.SpaceQueryDescription = "Find Largest Wall (0)";
        return;
    }

    // Add the line boxes
    float timeDelay = (float)lineBoxList.Count * AnimatedBox.DelayPerItem;
    lineBoxList.Add(
        new AnimatedBox(
            timeDelay,
            resultsTopology[0].position,
            Quaternion.LookRotation(resultsTopology[0].normal, Vector3.up),
            Color.magenta,
            new Vector3(resultsTopology[0].width, resultsTopology[0].length,
            0.05f) * 0.5f)
    );
    AppState.Instance.SpaceQueryDescription = "Find Largest Wall (1)";
}
```

This function helps to find the largest wall in the room and places a pink rectangle over the position of the largest wall. The key pieces of code here are `resultsTopology[0].position` and `resultsTopology[0].normal`, where we can directly get the information about the largest wall and place objects on or near it. I encourage you to look through these scripts and use the code as inspiration for use in your own projects.

Interesting things to try:

- Have an avatar sit on a chair or sofa at your friend's house.
- Put a holographic clock on your wall.
- Place a holographic dinner on your dining room table, complete with food, plates, silverware, and more.
- Create Roman pillars that extend from your floor to the ceiling.

Spatial Anchors and Persistence

This section briefly discusses the importance of spatial anchors. I walk you through how to use spatial anchors in your Unity project and provide some best practices when using spatial anchors.

Spatial anchors are locations in your application that are anchored to the real world. This is part of what makes Mixed Reality possible. Without spatial anchors, the virtual experience viewed through your device would become increasingly disconnected with reality over time. The HoloLens and other similar Mixed Reality devices do their best to scan and recreate your physical world using spatial mapping. However, this process isn't perfect, and the spatial map may be improving and adjusting over time. These adjustments would cause your holograms to appear to be shifting away from where you placed them, unless you "anchored" the holograms to the spatial map using a spatial anchor.

How to Use Spatial Anchors

Attaching and removing spatial anchors is a relatively simple process. A spatial anchor is called a *world anchor* in Unity. To attach an anchor to your game object, use the following method:

```
WorldAnchor anchor = gameObject.AddComponent<WorldAnchor>();
```

To remove an anchor from a game object that you don't intend to move, use `Destroy`:

```
Destroy(gameObject.GetComponent<WorldAnchor>());
```

To remove an anchor from a game object that you intend to move, use `DestroyImmediate`:

```
DestroyImmediate(gameObject.GetComponent<WorldAnchor>());
```

We need to destroy anchors on game objects because (as the name implies) anchors prevent us from moving objects. To move an anchored object in your scene, you first need to immediately destroy the anchor, move it, then create the anchor again. See the following code example:

```
DestroyImmediate(gameObject.GetComponent<WorldAnchor>());
gameObject.transform.position = new Vector3(2, 2, 2);
WorldAnchor anchor = gameObject.AddComponent<WorldAnchor>();
```

Hologram Persistence

What if you could save a spatial anchor to your device's memory and load it the next time your application starts? If you did that, your holograms and objects would be exactly where you placed them in the physical world, even after closing and re-opening your application. The good news is that this feature, called *persistence*, is available and widely used with spatial anchors.

You can save spatial anchors to your device in a place called the WorldAnchorStore. Most of the hard work is done for us using a script in the HoloToolkit called WorldAnchorManager.cs.

Include the WorldAnchorManager in your scene by finding it in the Project panel and dragging it to an item in your Hierarchy, such as a Managers object, as shown in Figure 6-20.

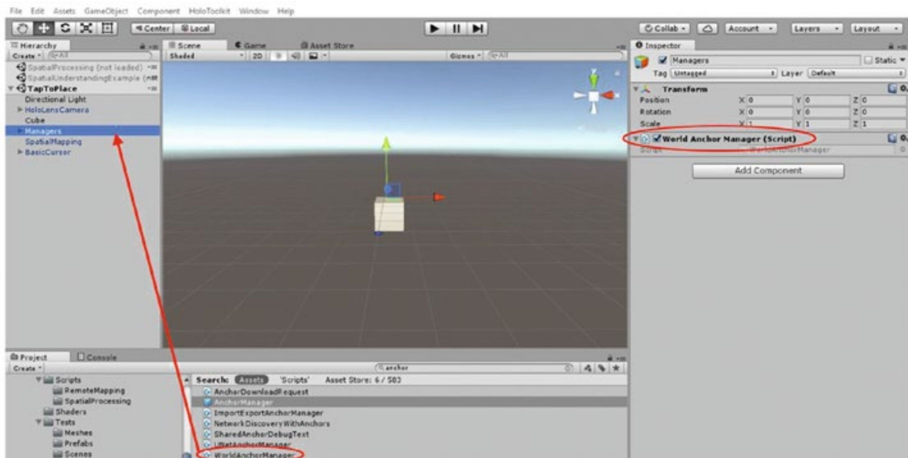

Figure 6-20. *Include the WorldAnchorManager.cs script in your scene for a simplified spatial anchor and persistence experience*

In Figure 6-20, you'll notice that I'm in the TapToPlace test scene from the HoloToolkit that we've been working with in this chapter. The TapToPlace scene already utilizes the WorldAnchorManager.cs script and the TapToPlace.cs script (attached to the cube). Feel free to dive into the scripts in this scene to explore how hologram persistence is set up in a working scene.

■ **Note** To experience persistence across app sessions, you need to deploy the application to your device or run your application in the HoloLens emulator outside of Unity. Persistence won't work when using the Unity Editor or holographic emulation within Unity.

With the WorldAnchorManager.cs script attached to your project, you'll be able to call its functions from other scripts. I include a few useful examples here. In the following examples, `anchorManager` refers to the WorldAnchorManager script:

```
anchorManager = WorldAnchorManager.Instance;
```

Use the following code to attach an anchor to your game object and save it to the WorldAnchorStore using a custom anchor name SavedAnchorFriendlyName that you can use to later retrieve the anchor in another app session:

```
anchorManager.AttachAnchor(gameObject, SavedAnchorFriendlyName);
```

To remove the anchor (perhaps while moving the object), you can use the following code:

```
anchorManager.RemoveAnchor(gameObject);
```

To load an existing anchor and attach it to a game object, use the same `AttachAnchor` code that we use to create anchors:

```
anchorManager.AttachAnchor(gameObject, SavedAnchorFriendlyName);
```

If the anchor store already has an anchor with the custom name you provided, it will load the anchor instead of creating a new one.

If you want to get all the existing anchor names in the anchor store and iterate through them, use the following code:

```
var ids = anchorManager.AnchorStore.GetAllIds();

foreach (var id in ids)
      {
        anchorManager.AttachAnchor(gameObject, id);
      }
```

A Note on Sharing Anchors

Not only do you have the ability to save anchors to your device, you can also transfer anchors to other devices. When two or more devices that are in the same physical room share anchors (and associated data), they see holograms and objects in the same place as everyone else. This allows for truly awe-inspiring shared sessions where multiple people can collaborate on the same project or view the same experience together.

Summary

Congratulations! You're now equipped with core knowledge about spatial mapping and can start taking advantage of some cool spatial mapping tools. Let's review what you learned in this chapter:

- What spatial mapping is
- All about the SpatialMapping prefab and all associated scripts
- How to enable spatial mapping in your application
- How to use spatial mapping to find and identify planes in your environment
- How to occlude objects using spatial mapping for a more realistic effect
- How to use spatial understanding to unleash the power of spatial mapping, identify objects and surfaces in your environment, and place objects on key surfaces in your environment
- All about spatial anchors and how to use them in your application
- How to persist objects and holograms across app sessions

You may not have thought there was so much to cover in a chapter about spatial mapping. Spatial mapping is extremely important for Mixed Reality. In fact, it's the headset's understanding of the physical environment that warrants the *mixed* in Mixed Reality, allowing our applications to mix the virtual and physical worlds together.

We've only touched the tip of the iceberg with regard to spatial mapping. Many untapped opportunities are waiting to be explored and implemented. Here are just a few examples of spatial mapping ideas I've heard mentioned:

- Expanding the spatial mapping mesh to make your room or area appear larger than it actually is
- Virtually painting your walls and furniture to see what various color options would look like
- Making holes in your walls to give the sensation that you can see through them

As you continue your developer journey, think about creative ways you can leverage spatial mapping and all associated tools. Be sure to think outside the box. Keep in mind that your spatial mapping mesh doesn't need to obey the laws of physics like your real walls and furniture do. The sky is the limit to what you can achieve.