

Board Game Making in Unity

Part 3: Importing Assets and Making the Board

Overview

This is part three of the board making in Unity tutorials and will build off the project created in tutorial two. In part one, we created a rolling six sided die, and in part two we added some scripts to read the value of the die and display on screen. If you haven't looked at part one and two, you should probably go back and do that now. In this third part we will looking at importing assets from Adobe Illustrator and Maya.

This tutorial was made for Unity 3.0 and assumes that the user has at least watched the first few Unity tutorial videos found here: <http://unity3d.com/support/documentation/video/> This tutorial starts with the Unity project created during tutorial two.

This tutorial attempts to be more than a list of steps to follow, I tried to record some of the reasons for why I make choices. If you don't want to follow my logic and just want steps, look at the stuff in bold. At the end of each section is a list of the new Game Objects, Components or Functions covered.

We left off the last tutorial with a roll-able die that displayed its value on the screen. For this tutorial, we will look at two different board designs in Adobe Illustrator, importing our board design to Maya, importing the Maya board mesh to Unity, and scripting the board to change colors as a player piece moves across the board.

Drawing Board Designs in Adobe Illustrator

Our board game needs a board. We could go through the trouble making a board from Unity primitive Game Objects, but sometimes we want something a little fancier. In the next section we look at some Adobe Illustrator drawing tools for designing a board, and we will design a small pond board for a game piece to move along. In the following section, we will create a 3D mesh model in Maya from the design. When this is imported to Unity, we can use the mesh not only for visuals, but for collision as well.

To get warmed up in Adobe Illustrator, we can quickly look at how to design a board using basic geometric shapes. Open up **Adobe Illustrator and make a new document**. I set my document Units to Inches and **make the width and height ten inches**. Using a Illustrator -> Maya -> Unity work flow causes one inch in Illustrator to translate to one unit in Unity.

We don't want to use just any Illustrator paths to create our game board, we want each tile to be a closed path. That means rectangles, circles, triangles, or any curvy shape, as long as it has no endpoints. In the tool bar, find the rectangle tool and left click to expose the other shape tools, choose the Polygon Tool. Normally you could left click and drag to draw a polygon, but to have more control, single left click on the artboard to bring up the polygon dialog window. Make a polygon with six sides (hexagon) and a radius of one inch.

In the toolbar choose the selection tool, it looks like a black arrow. Now you can select the polygon and left click drag to move it around the artboard. Try not to resize the object by clicking and dragging an empty box on the edge of the object. Holding down the Alt key will allow a left click to copy the object, and you can drag it to the correct location. Alt-left clicking over a solid corner box will allow you to drag to another point, to snap the hexagons in place. I used this method to create a horse shaped board.

We can use Illustrator to create some more complicated, organic shapes. How about a jellybean shaped pond? We want the board to have an underlying grid structure, but we want the edges to look like a pond from above, so we can draw the grid in Illustrator, and cut the pond shape out of the grid.

Make a clean Illustrator document and draw a grid of rectangles, with each rectangle 2 inches by 2 inches. Make the grid be two rectangles wide and three rectangles high; each edge should be overlapping.

Find the pen tool in the Toolbar and left click to select. Use the pen tool by left clicking in the general shape of pond on top of the grid. Remember to close the path by left clicking on the initial point at the end of the sequence.

Left click and hold over the pen tool in the toolbar to reveal the pen tool palette. Choose the "Convert Anchor Point Tool". This tool will allow you to pull the smooth shape of the pond out from the points along the pond path. At each point, left click with Convert tool until the path takes the desired shape.

Your shape should be sitting above the grid of rectangles; if it is not, select it and choose with the Selection tool and click Object -> Arrange -> bring to front. The pond shape is now ready to be cut into

the grid. With the pond shape selected, choose Object -> Path -> Divide Objects Below. That should cut the pond shape into the grid of rectangles. Use the Selection tool to delete the extra edge pieces. If you look in the layers window, Window -> Layers, and click the little triangle in Layer 1, you should see six paths listed.

Now the drawing is ready to save to format that Maya can read. Save the drawing as an Illustrator 10 document, and make sure you don't enable compression, also uncheck the compatibility for PDF(it will cause extra paths in Maya). Using compression will cause Maya to have an error during import. When we import into Maya, this entire shape will import as a single mesh. If you want multiple objects, use multiple files. Delete all guides you have created, they also import into Maya as edges.

Importing the drawing from Illustrator to Maya

Open up Autodesk Maya and make a new scene. Set up the Maya environment to work for working with meshes, normally Maya opens with Animation menus, we want mesh and polygon menus. Just under the File menu is a button that says Animation or Polygons, click and change it to Polygons.

Import your Illustrator 10 document by clicking Create->Create Adobe Illustrator Object, but make sure to click the square on the right side of the text to see the import options. Using the default options will nicely extrude and bevel your object, but won't offer you much control. I chose "Bevel" import, and set my extrude distance to 0.25 and the bevel width to 0.0 and bevel depth to 0.0. This just gives me flat object, but it has the depth I want.

Notice that the imported mesh in Maya is oriented the wrong direction, just rotate it to fix that problem. Remember that the Y-Axis in Unity is up. You need to "freeze" the transformation to make it import into Unity properly, so with the mesh selected in Perspective view, click Modify->Freeze Transformations. Feel free to UV map and texture the new mesh, Unity will pick up the UV and import it along with the mesh. For this tutorial, we will skip UV mapping and assign a new material to the mesh. I like the pivot(little arrow thingy) to be in the center, so Modify->Center Pivot.

We want all our tiles to be separate and we can do that in Maya with the separate tool. Select the mesh in the perspective view and, choose the Polygons tab and then the separate tool, or Mesh -> Separate from the top menus. The faces will turn to separate meshes.

The meshes are now separate, but they are still parented to their original location. Open the Outliner from Window -> Outliner to see the scene hierarchy. Select the meshes under the original transform, and unparent with Edit -> Unparent.

Even though the meshes aren't under the original transform, they still remember their parent transform, so we need to fix that. Select all the un-parented meshes, and clear their history with Edit -> Delete by Type -> History. You should be able to safely delete the original transform.

The mesh is nearly ready to import into Unity. At this stage we can do one more thing to make life easier on the Unity side: rename our meshes. To rename a mesh, select it in the outliner or in the perspective window, and find the first attribute tab. It should say "Transform" and have the name of the object next to it. Click the name and rename the object to something like tile1.

The scene will need to be saved twice, once for backup purposes, and once for Unity. First, save the Maya scene in a location other than the project folder. To do this we'll simply go File -> Save and choose a safe location, like the Desktop.

Importing the mesh from Maya to Unity

As you may have guessed, in order to import the scene to Unity we need to drop a file into Unity, but instead of using the standard Maya binary file (.mb), we'll export the scene as an FBX file. Save the Maya scene with a descriptive name, something like GameBoard.fbx, then drag the file into the Assets folder of your Unity project. When you return the Unity IDE window, the Maya scene will begin importing. You could also drop the fbx file into the Projects tab in the Unity IDE.

Select the new object in the Project tab and examine the information in the Inspector tab. The second Component listed in the Inspector should be (FBXImporter), that's the Component that handles importing the Maya scene. There are many options in the FBXImporter, but the first one we're interested in is the scale. Make sure the scale is set to 1.0; for some reason, the scale often is changed to 0.01, possibly expecting much bigger Maya units.

The second FBXImporter option of interest is the Generate Colliders checkbox, select this box to make the object interact physically with rigid bodies in the physics engine. If this box is left unchecked, the mesh will render properly, but will not interact physically with any other objects in Unity.

The mesh should be ready to view in our scene. Drag the GameBoard from Project tab into the scene and hopefully the mesh will show up in the Scene view. If it doesn't, check again that the scale of the importer is 1.0.

Arrange the mesh so that it sits in the proper place, just above the GroundPlane. This will be the base for our spaces to move a game piece across. Sometimes materials import into Unity from Maya, sometimes not; update the material in the Inspector if it is wrong. Perhaps even make a new material called PondWaterMaterial with ProjectTab ->Create -> Material . Make the new material Reflective/Specular, like shiny plastic. Set main color to dark blue, specular to light blue and reflection to grey-ish. Apply texture to water tiles.

Run the project to make sure the collider is working for the imported mesh. Test with the collide-able die, look for the die hitting the newly imported mesh.

Giving the board behaviors

To give our board some visual feedback when a piece moves onto a tile, we'll make our tiles change color. We need to again use triggers. Because we have a nice mesh for each game tile already, we'll use an invisible duplicate mesh sitting just above the visible tiles as triggers. Duplicate the pondTiles and rename to pondTilesTriggers. Now just nudge those up so they float just above the original tile meshes. Select IsTrigger in the collider for each tile, and turn off the mesh renderer. Rename each tile trigger game object to tile1trigger, tile2trigger, etc., so we know which trigger goes with which tile.

To handle the changing of color, make an new Javascript, called TileTrigger. Our script needs to store two pieces of information: the target tile that needs to change color, and the original color of the tile. Here's the contents of that script:

```
public var targetTileName = "";
```

```
private var originalColor;
```

```
function Start() {
```

```
tileGameObject = GameObject.Find(targetTileName);

originalColor = tileGameObject.renderer.material.color;

}
```

```
function OnTriggerEnter (other : Collider) {

    Debug.Log(other.name + " entered");

}
```

```
function OnTriggerStay (other : Collider) {

    tileGameObject = GameObject.Find(targetTileName);

    tileGameObject.renderer.material.color = Color.green;

}
```

```
function OnTriggerExit( other : Collider ) {

    Debug.Log(other.name + " exited");

    tileGameObject = GameObject.Find(targetTileName);

    tileGameObject.renderer.material.color = originalColor;

}
```

We have the two values to store at the top, the original color and the tile target. Since this script will be added to every tile trigger, we leave the targetTileName to be filled in the Unity IDE. This script has some new overloaded Unity functions: Start(), OnTriggerStay(), and OnTriggerExit(). These three functions are overloaded functions similar to the Update() function.

The Start() function is called once before the Update() function starts to get called. The Start() function is a great place to store values that get used repeatedly in your script. In this example, the original color of the tile needs to be stored, and instead of using a GameObject.Find() function call each time we need to get the color, we'll store that value of the Component for use later.

OnTriggerStay() and OnTriggerExit() are similar to the OnTriggerEnter() function, but are called when another game objects stays in contact with the trigger and when another game object leaves the trigger. The script finds the target Object by name in the Start function and stores that for later, we wouldn't want to lose track of the original color.

Each tile trigger needs to have a copy of this script added as a Component. As you add the script to each trigger, change the name of targetTileName in the Inspector to match the corresponding tile, like tile1 or tile2.

Functions Used in this Section:

Start() - <http://unity3d.com/support/documentation/ScriptReference/MonoBehaviour.Start.html>

OnTriggerExit() - <http://unity3d.com/support/documentation/ScriptReference/Collider.OnTriggerExit.html>

OnTriggerStay() - <http://unity3d.com/support/documentation/ScriptReference/Collider.OnTriggerStay.html>

Making a moving player piece

Now is a good time to add a game bit for our player. We could draw something in Illustrator, and extrude in Maya, but for now I'll just use a Cube and rename it to "PlayerPiece". It would be nice for the piece to move automatically, so we'll create a script to do that. Add another Javascript called "MovePlayer", and insert this code:

```
public var dieName = "SixSidedDie";
```

```
public var allowedDistance = 0.5;
```

```
private var lastDieValue = 0;
```

```
private var target : Transform;
```

```
public var damping = 3.0;
```

```
private var dieValueComponent;
```

```
function Start() {
```

```
    dieGameObject = GameObject.Find(dieName);
```

```
    dieValueComponent = dieGameObject.GetComponent("DieValue");
```

```
}
```

```
function Update ()
```

```
{
```

```
    // get the new transform if necessary
```

```
    if (lastDieValue != dieValueComponent.currentValue) {
```

```
        Debug.Log("tile" + dieValueComponent.currentValue.ToString() + "target");
```

```
        tileGameObject = GameObject.Find("tile" + dieValueComponent.currentValue.ToString() + "target");
```

```
        target = tileGameObject.transform;
```

```
}
```

```
    if (lastDieValue != 0) {
```

```

    dist = Vector3.Distance(transform.position, target.position);

    if (dist > allowedDistance) {

        transform.position = Vector3.Lerp(transform.position, target.position, Time.deltaTime * damping);

        transform.rotation = Quaternion.Slerp(transform.rotation, target.rotation, Time.deltaTime * damping);

    }

}

lastDieValue = dieValueComponent.currentValue;

}

```

In order for this script to work correctly, we need to add some targets for the PlayerPiece to move toward. Make an empty Game Object and parent it under each tile trigger. Change each of the empty GameObject's names to tile1target, tile2target, etc. Each should be arranged so that it sits above the tile, near the center of where you want the PlayerPiece to stop moving.

To summarize how this script works: on Start, the script caches the DieValue Component so it can determine value later. On Update, the script checks if the lastDieValue is different from the current die value, and if so, it grabs the Transform component from the tile that corresponds to the number on the die. Then, if there is a value to move to, meaning the last value is not 0, remember that 0 is the die starting value, we calculate the distance between the current position of the target position. If that distance is greater than a specified allowed distance, we move the piece toward the target by altering the Transform Component of the PlayerPiece using the Lerp and Slerp functions.

For it to move around, the PlayerPiece needs a rigid body, so add that component. Attach the MovePlayer script to the PlayerPiece and test the project. You may notice that as the piece moves toward the target, it jumps around a little. This is because the script is fighting with the physics engine; the physics calculations are trying to apply gravity and other forces, and we're bypassing that with the script. To solve this we can check "Is Kinematic" in the Rigidbody component of the PlayerPiece.

Components Used in this Section:

Transform.position - <http://unity3d.com/support/documentation/ScriptReference/Transform-position.html>

Transform.rotation - <http://unity3d.com/support/documentation/ScriptReference/Transform-rotation.html>

Functions Used in this Section:

Vector3.Distance() -

<http://unity3d.com/support/documentation/ScriptReference/Vector3.Distance.html>

Vector3.Lerp() - <http://unity3d.com/support/documentation/ScriptReference/Vector3.Lerp.html>

Quaternion.Slerp() -

<http://unity3d.com/support/documentation/ScriptReference/Quaternion.Slerp.html>

End

If all went well, you should have a cube that moves automatically across a game board surface. This simple example should give you some idea of how to start generating behaviors in your game.

initial part 3 - 1/17/2011 - Jonathan Cecil . jonathancecil@ucla.edu

1/24/2011 - jc - edits, script explanations