Ayrshire College

Nightmare Island – Going Inside

Interior Mapping & Transitions

Contents

Interior Mapping & Transistions 1
New Scene – Setup Editor 3
Snap Settings 6
Empty – Grouping Objects 6
Corridors – Laying out the level plan 7
Create Empty Group 7
Create Quads 7
Test Textures – Show Unit Grids 8
Rotate Walls 8
Add Roof 10
Create Re-usable Pieces – Prefab (ulous) Duplication 11
Prefabs to make prefabs to make prefabs… 14
So we have Reusability… Now what? 15
Layout your level 15
Investigation – Project Relevance 18
Portal Box – Loading a Scene on Proximity 19
Setup Build Settings 19
Setup Scene 19
Add Cube 19
Add Spotlight 19
Add Script – Evaluate, Develop, Test 20
Make Prefab 20
Add to other Scene from Prefab 21

Sequence is Set – Add the Scripting Magic 21

Locate Shader Variables 22

Update Renderer 23

Detect FPSController Proximity 23

Update Shader Values 24

Update Spotlight Values 24

Add GUI Text 25

Load Level 26

Playtest – Debug – Experiment – Extend 28

Key Lessons:

- Prototyping interiors with quads and unit grid textures
- Creating Prefabs
- Proximity Triggers – Load a scene, activate children objects and components
- Text Prompts – UI Text

Periodically save your work and back it up!

Key shortcuts:

- Duplicate Object - Ctrl+D
- Snap Move/Rotate/Scale – Hold Ctrl+Drag Widget
- Rename Objects – F2 (Also a Windows shortcut)
- New Empty - Ctrl+Shft+N
- Transform Widget – W
- Rotate Widget – E
- Scale Widget - R

Throughout this tutorial it will make references to your Project view and Scene Hierarchy view at different stages, please ensure you double check all notes carefully if you encounter errors, re-read the directions. If you're stuck, ASK FOR HELP!

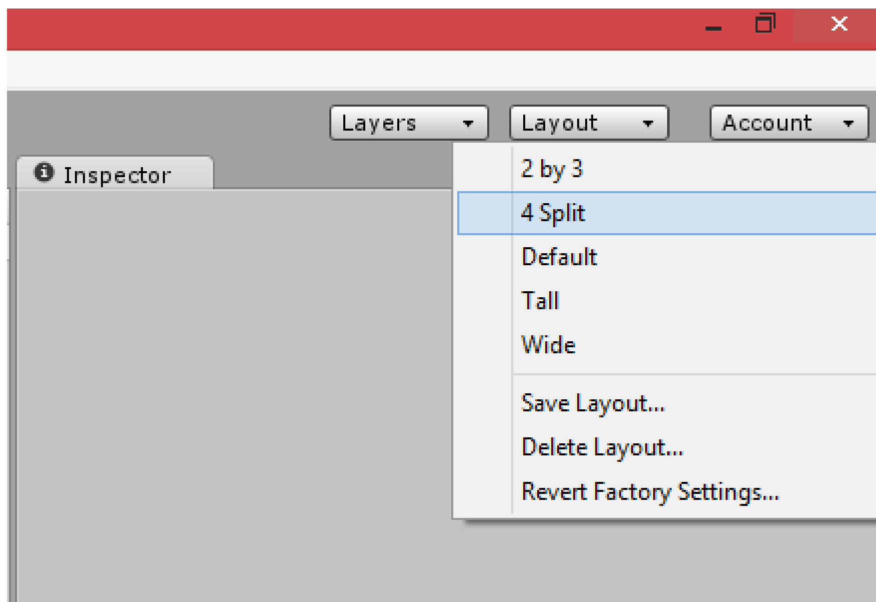
Now that we have had ample time to explore and investigate exterior environments we'll now utilise all that visual styling knowledge to rapidly prototype an interior layout.

For this tutorial we're going to look at quickly prototyping an interior space consisting of corridors and varying spaced rooms. Then we'll look at setting up some trigger objects to load scenes and change between your interior and exterior maps.

New Scene – Setup Editor

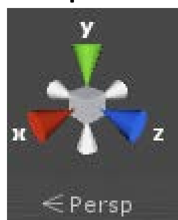
Create a new Scene – Save as **02_Level_002** (if you have adopted a different naming convention please adhere to that instead)

Switch Layout view to use the 4-Split layout (top right of editor window)

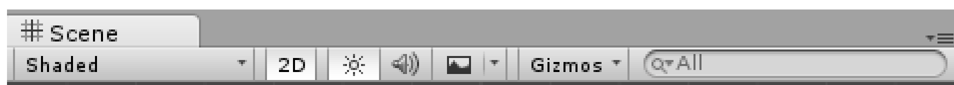


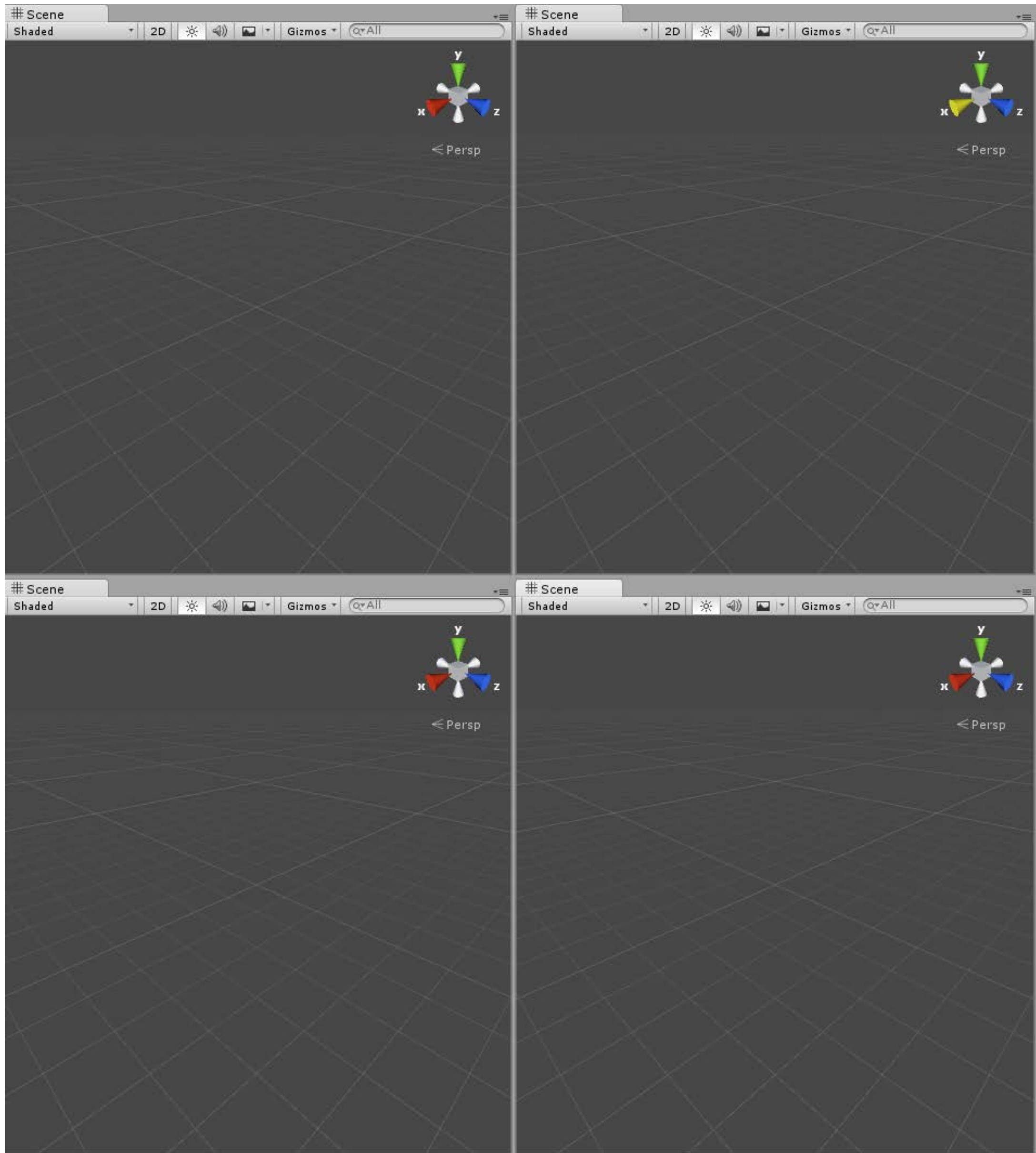
When modelling interiors, objects can get lost quickly if only using one camera (culled objects) and have multiple angles can really help prototype the level more effectively.

Viewport Axes Selection



If you can't see the axes selection, be sure to check if **2D** button is enabled on the viewports toolbar





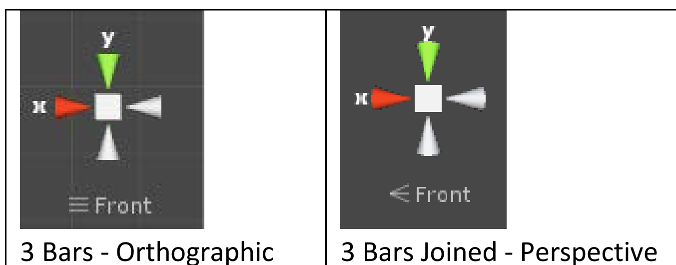
Bottom Right – Perspective (Click the corner of the grey cube if it's not showing)

Top Right – Front – Click the Z Cone on the views axes selector

Top Left – Top – Click the Y Cone on the views axes selector

Bottom Left – Right – Click the X Cone on the views axes selector

By clicking the selector, you're telling Unity's viewport to look along that particular axes. To change between perspective (3D Projection) and orthographic (2D Projection) simply click the cube in the centre of the selector. This will change the UI element in front of the view name to indicate which mode.



3 Bars - Orthographic	3 Bars Joined - Perspective

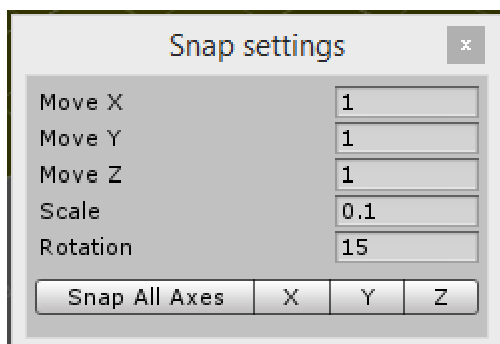
For editing floor layouts and interior scenes it is best to change the Front, Top and Right views to orthographic, leaving the 4th viewport to display the fully rendered scene.

Snap Settings

Now let’s check our snap settings go to **Edit | Snap Settings…** (All the way at the bottom of the menu)

I like to keep my move set to one unit (one metre if you’re working in default scaling – didn’t change it? Then you’re on default), because even with snapping enabled via holding Ctrl it is fairly quick to re-align with little to no re-adjustment. Rotation, keep at increments of 15 as these denominations are by far the most common (30, 45, 60, 90 etc.).

For now, leave the values as is, but if you feel that the increments aren’t to your liking, you now know where to find them!



Empty – Grouping Objects

In terms of project organisation we have currently got folder structures set up but our hierarchy in our scene (if you painted a lot of trees in your exterior you’ll know!) can quickly get messy.

So for this scene we’re going to organise our hierarchy by grouping objects using Empties. What this means is that any object within the empty group (shown by the arrow head to indicate sub-objects) will be classified as its **children** as will all their components and relationships.

These can also have their own components to manipulate the child objects below as a group or individually. One rule of thumb here if objects in a group are “misbehaving”, you may want to check there’s parent contributing changes. That all said let’s make some corridors!

Also grouping objects together like this and forward planning means setting up features like occlusion culling later is far less troublesome as your scene is already well organised!

Corridors – Laying out the level plan

Create Empty Group

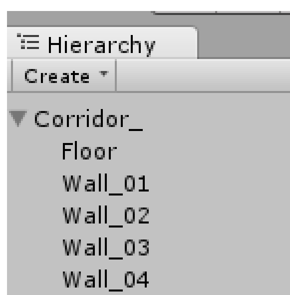
Right click Hierarchy Pane, Select **Create Empty** – Position at the origin (0,0,0)

Rename to **Corridor_** (Select empty, press F2 or **right click | rename**)

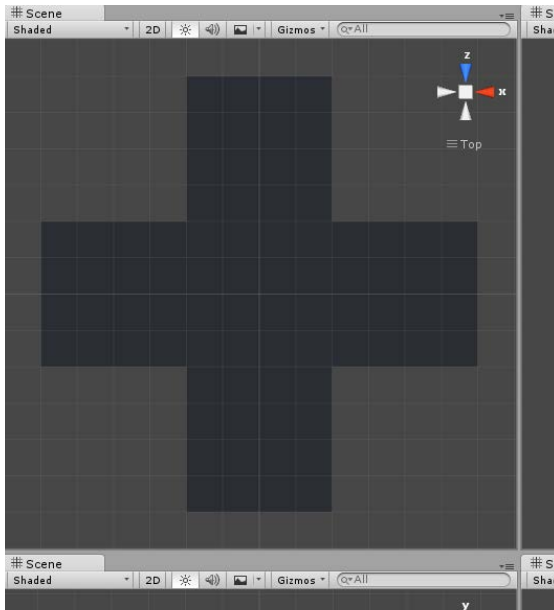
Create Quads

- With the Empty selected, right click and Select **3D Object | Quad –** Also position at the origin (good form to maintain group consistency when you start blending group and local transforms)
- Rename this quad **Floor**
- Change **Rotation** on the quad's **X axis** to **90** to see it as the floor and ensure the normals face up (don't freak out if you don't see it, it only renders one side to reduces calculations i.e. Culling)
- Change **Scaling** on **X** and **Y** to **4** – You can change the scaling to suit later, for the purposes of the tutorial we'll scale to 4x4 metre to provide a sufficient space for our 1.84-2.2 metre tall character (FPSController Y value).
- Press **Ctrl+D** to duplicate the quad
- Using, **Snap drag** (Hold **Ctrl** when moving/scaling/rotating) in the **Top Viewport**, move the duplicated quad to the top most edge from your viewport.
- Rename the quad to **Wall_01**
- Select the floor again, duplicate, this time snap dragging to the left edge of the floor
- Rename **Wall_02** (Seeing a pattern here?)
- Duplicate floor again, snap drag right, rename **Wall_03**
- The same process to add the bottom wall, called, you guessed it, **Wall_04**.

Your hierarchy should now look like this:



Now your net shape should look like this from the Top Viewport



Some of you may wonder why we've kept the generic numbering rather than say, Top, Left, Right and Bottom. Put simply, under rotation there's no guarantee that the wall will be on the top, left, right and bottom in the viewport so it could confuse the user as to what direction the piece is facing.

Test Textures – Show Unit Grids

Now we're going to allocate some test textures, I suggest using grid formatted textures as they can greatly help with visually representing scale of objects. Some can be found in the standard assets prototyping package (Available on **Moodle page 3D Level Editing** if not included in your current project) under **Standard Assets/Prototyping/Materials**.

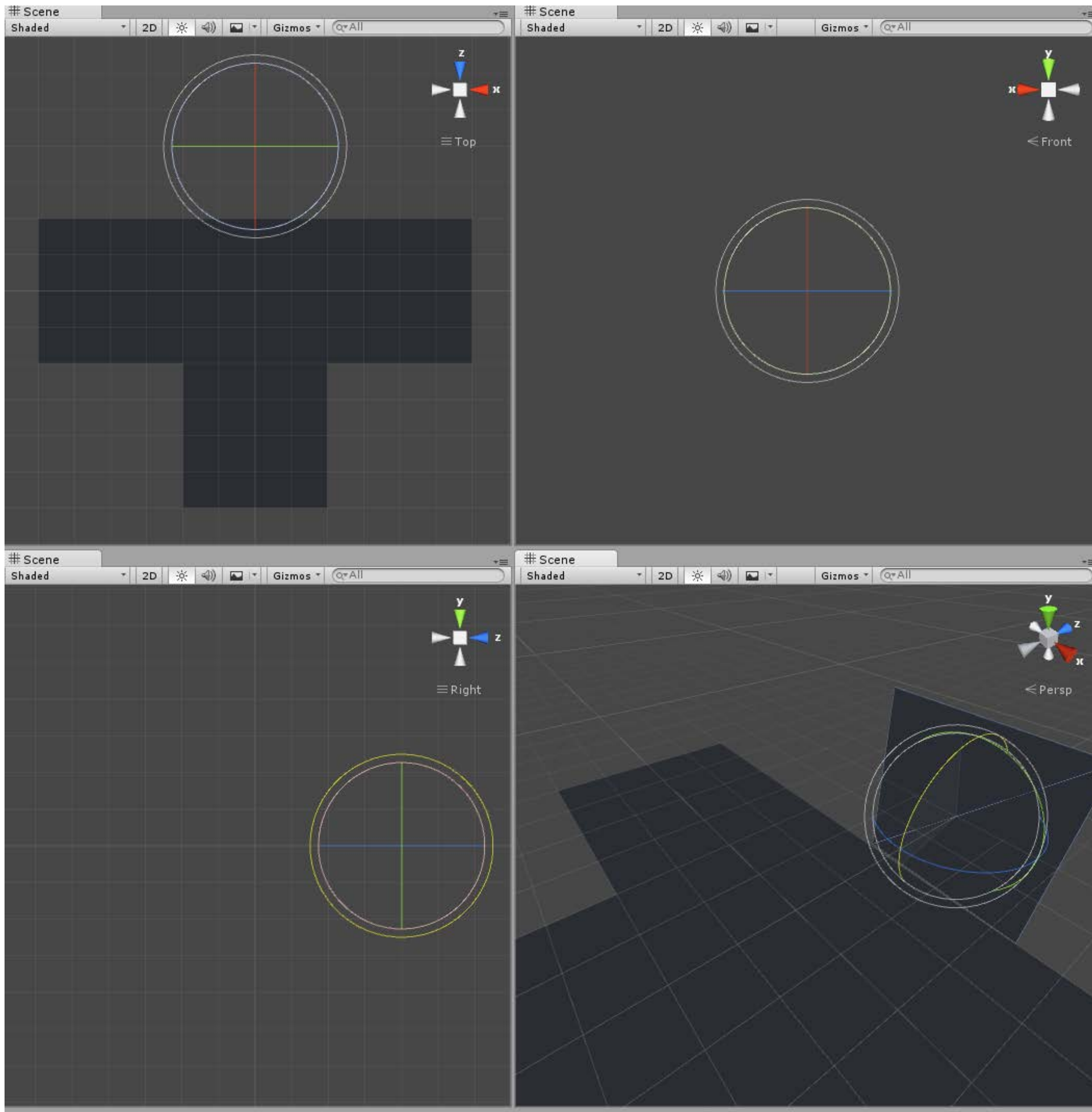
For the floor I picked **Pink Grid** and the walls I picked **Yellow Grid**, for the roof (which we'll add from one of the wall pieces later) will be the **Turquoise** colour.

For your finished prototype you'll have to texture your areas to suit the theme (created or sourced), but for now we'll use these to give us the unit grid on the surface of our quads.

Rotate Walls

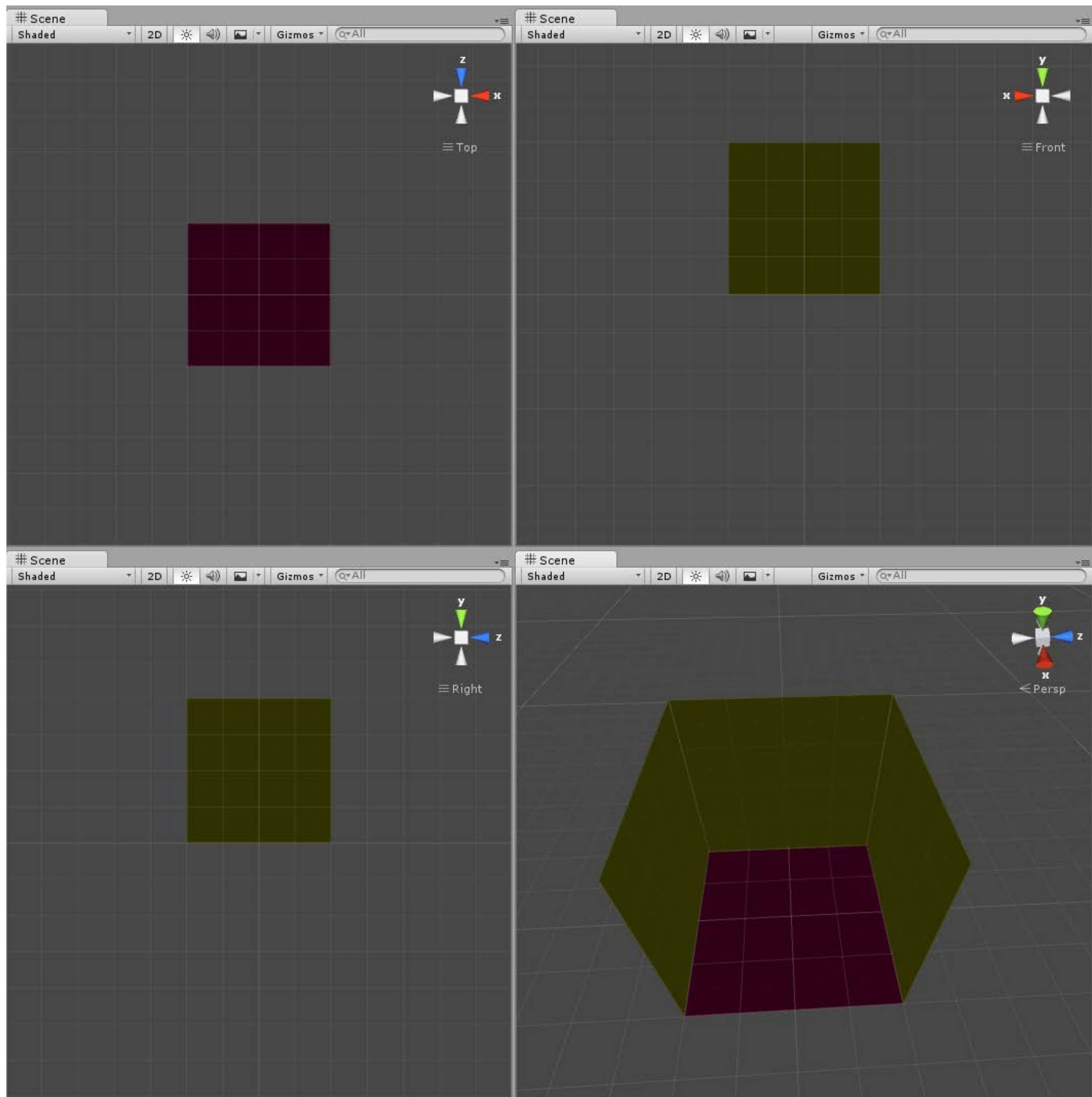
From the perspective viewport, select **Wall_01** and **press E** (change to rotate widget)

Now use snap drag (hold **Ctrl**) on the X axis rotate line to bring the wall piece to face upright – it will disappear from the top view net shape but the widgets should still show its position.



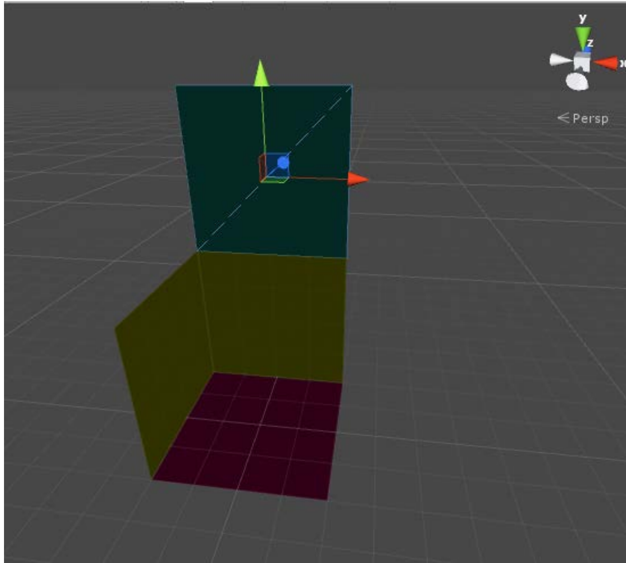
Position **Wall_01** using snap drag position (**Press W** to switch widgets).

Now do the rotation and positioning for **Wall_02** (Z Axis rotation), **Wall_03** (X Axis rotation) and **Wall_04** (Z Axis rotation). When you're finished it should look like the following:

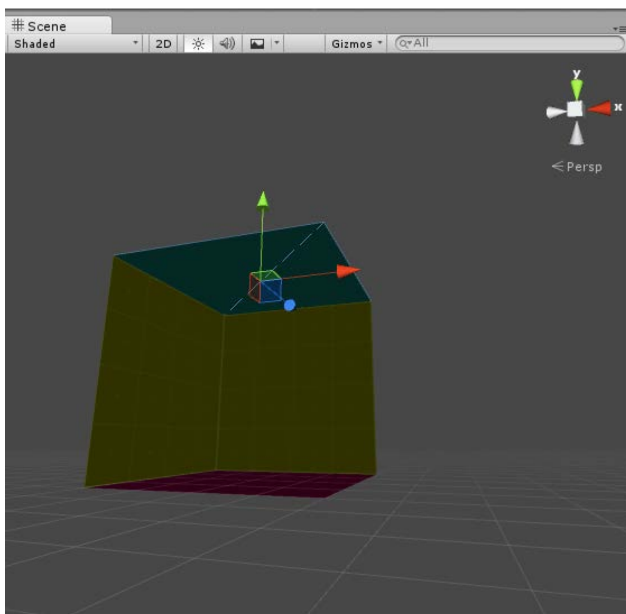


Add Roof

- Select **Wall_01** and duplicate it
- drag snap position it above the wall piece
- Rename **Roof**
- Add the texture from the **prototype** set for **turquoise**.



Now **rotate** and **position** it so that it is **facing down** into the room and directly above the **floor** area. You will have to move your editor viewport camera down to see underneath the roof surface.



Create Re-usable Pieces – Prefab (ulous) Duplication

Lastly, in setting up our level areas we're going to now duplicate the groups and make level layout shapes (Now that name of **Corridor_** doesn't look so dumb ;)). Now you might be tempted to just jump right in there and start duplicating the main **Corridor_** group in your scene hierarchy and editing away. However, for the purpose of making our future efforts that bit simpler, we're going to make the corridor piece a prefab for project.

The Benefits

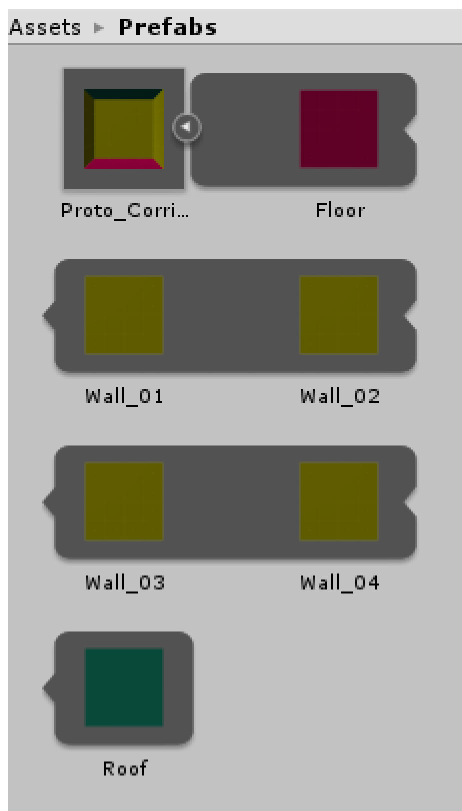
- Great for maintaining object groups and components as a template
- Duplication editing made easier, you can edit the prefab data directly and it will update all instances in your scene automatically!
- Allows a formal structure that can be coupled with scripts more closely
- Can be used cross project and so long as you maintain additional local resources (textures, 3D models from external packages etc) in a good, logical order (i.e. parent>children>leaf data) then they can be a very versatile tool.
- Can be used cross scene, by decoupling the objects from the hierarchy into the project, it can be used in many places

Walkthrough

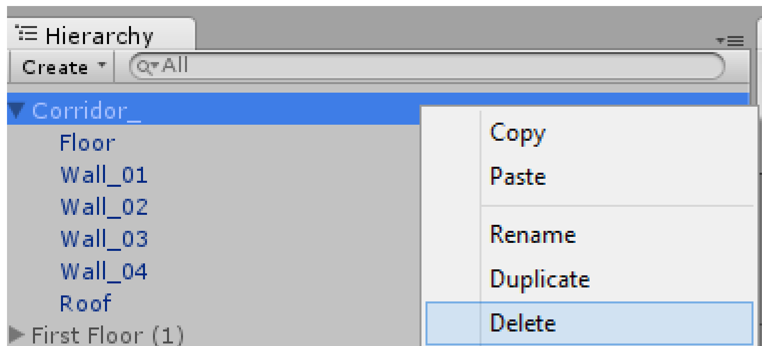
To create a new Prefab is very straight forward, in your project view, select your root Assets directory and create a new folder for Prefabs.

After which, open the folder, **right click | create | Prefab** rename to **Proto_Corridor_MASTER** this will make a blank in the area. From Hierachy view, drag the **Corridor_** group directly into the Prefab box.

This will update the prefabs preview icon to show the data has been attached similar to below.



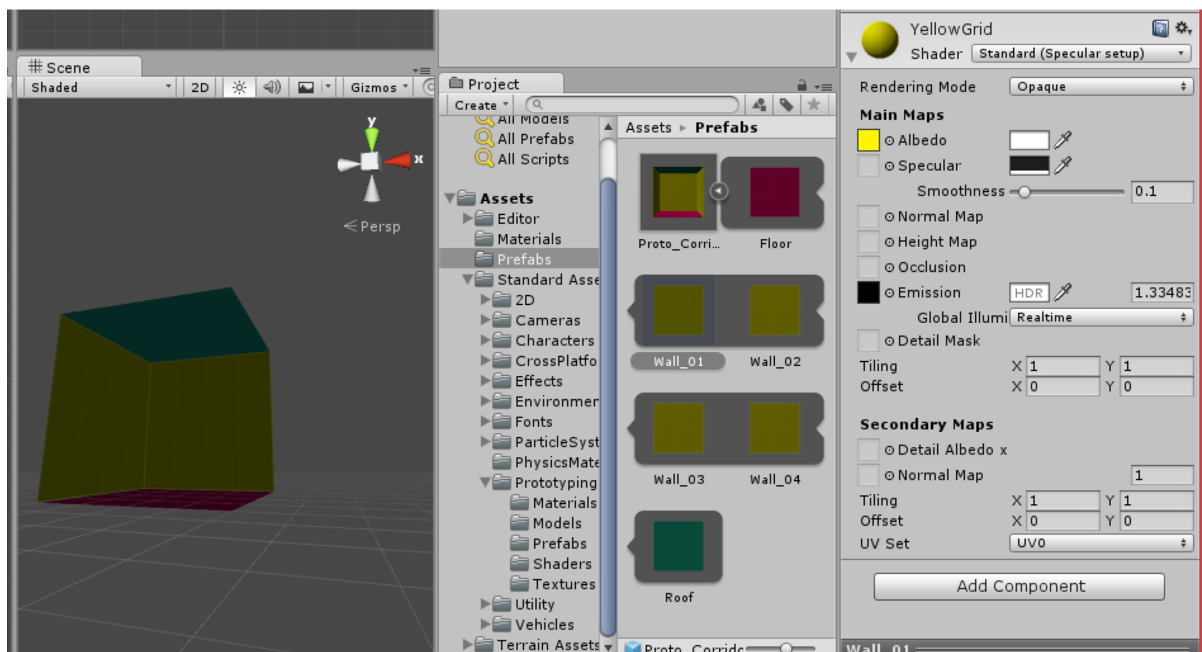
We can now safely delete the group **Corridor_** from our Scene Hierarchy as it's served its purpose.



Now that we have a prefab, we should note the biggest key point, ***any changes to the prefab data or components will be reflected in all instances within your scene hierarchy***. You can however break the link to the prefab by altering the object/s while they're in the scene (we're' going to do just that now).

You can transform the objects as you please (why else would need instancing?) but core components will break the link, don't worry Unity will alert you when it is going to result in a change.

To see a quick example of this, select **Wall_01** in the Prefab data



- Unfurl the shader group in the Inspector
- Change the Albedo Texture from Yellow to Blue
- This will live update the walls in your editor
- Undo changes with **Ctrl+Z**

Prefabs to make prefabs to make prefabs…

Drag a copy of **Proto_Corridor_MASTER** into your hierarchy, now we're going to use this prototype to make some more.

For a corridor system we're going to need:

- A dead end
  - Delete 1 wall
- A mid-section(currently we have a box really)
  - Delete 2 opposing walls
- A corner (we don't need 2, just rotate ;))
  - Delete 2 adjoining walls
- A T-Junction
  - Delete 3 adjoining walls
- A cross-roads
  - Delete all 4 walls

Walkthrough

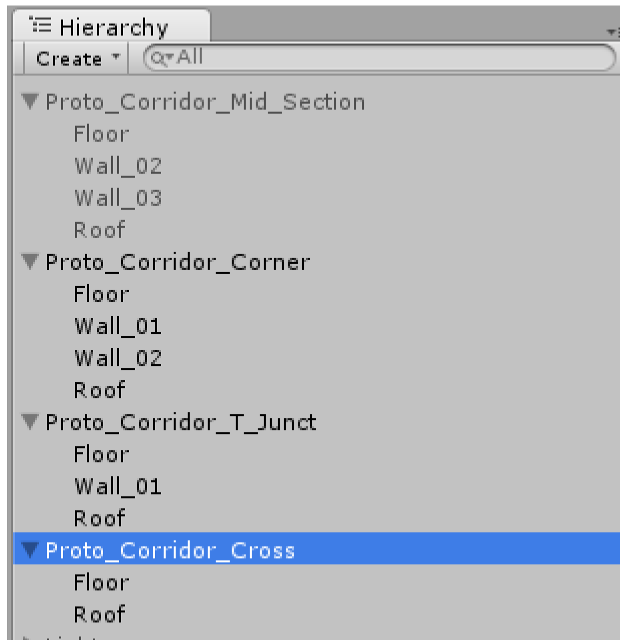
Mid-Section Piece – Linking Hierarchy Prefab copies to MASTER

- Unfurl **Proto_Corridor_MASTER** in the hierarchy view
- Delete **Wall_01** and **Wall_04** (this will break the link to the prefab data for these children objects but retain the rest, neat right?)
- Rename **Proto_Corridor_MASTER** in the hierarchy as **Proto_Corridor_Mid_Section**

Now based on the specified wall deletion above, run through the above steps for each respectively creating: **Proto_Corridor_Corner, Proto_Corridor_T_Junct, Proto_Corridor_Cross, Proto_Corridor_Dead_End** in your hierarchy.

All done! Simple, ease, elegant.

To compare your work at this point your Scene Hierarchy view should look similar to the following:



Now we have a master template in the form of our Prefab which is linked to the children versions in the scene hierarchy including textures and components (you can add anything you normally would). If we ever want to make an object unique from the prefab data, we simply edit it in the hierarchy view and inspector.

If you're going to edit objects quite heavily from the prefab often, consider making another template. Reusability in your resources is one of the most valuable transferrable skills you can impose on your projects. This is one example of many ways of linking objects, components and their data.

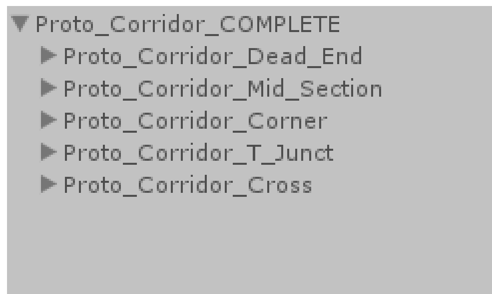
So we have Reusability… Now what?

We duplicate these pieces, stitch them together using snap dragging, and lay out a floor plan for interior levels! Now that you know how to use empties, quads and prefabs you can use these techniques to make custom objects of your own!

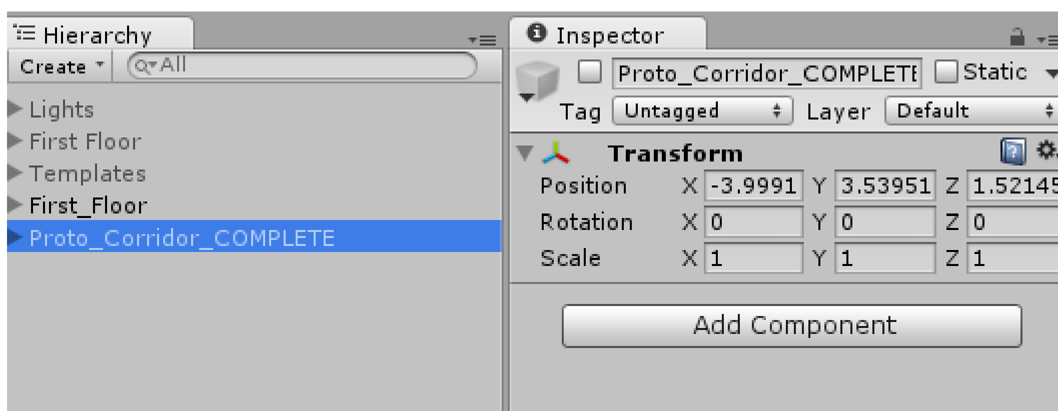
Layout your level

First off we're going to back up our hierarchy pieces

- Create a new empty in your scene hierarchy, name it **Proto_Corridor_COMPLETE – This is more for our records at the moment in case we want to revert to these later**
- Select all your **Proto_Corridor** Groups and move them into this new parent group



- Duplicate the group
- Rename Duplicate **First_Floor** - This is the one we'll be working with
- Uncheck the display checkbox in the inspector for the group **Proto_Corridor_COMPLETE – This will hide it from the scene and take it out your pipeline**

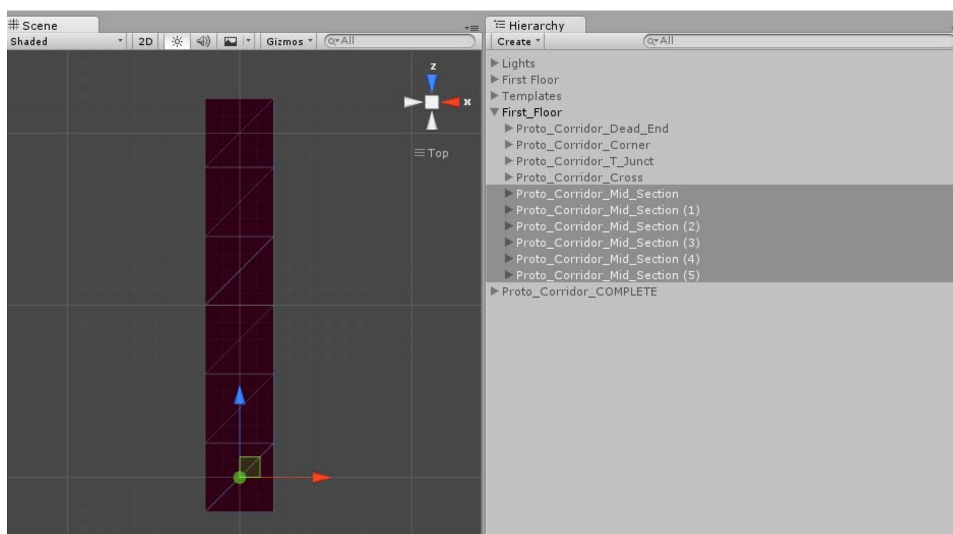


Now to compose the level, first change your **Edit | Snap Settings** to 4 units

- Unfurl **First_Floor** Group
- Hide all other groups except the **Mid_Section** – Same as we just did above by unchecking

Remember to come back and recheck the box when we start to use them!!

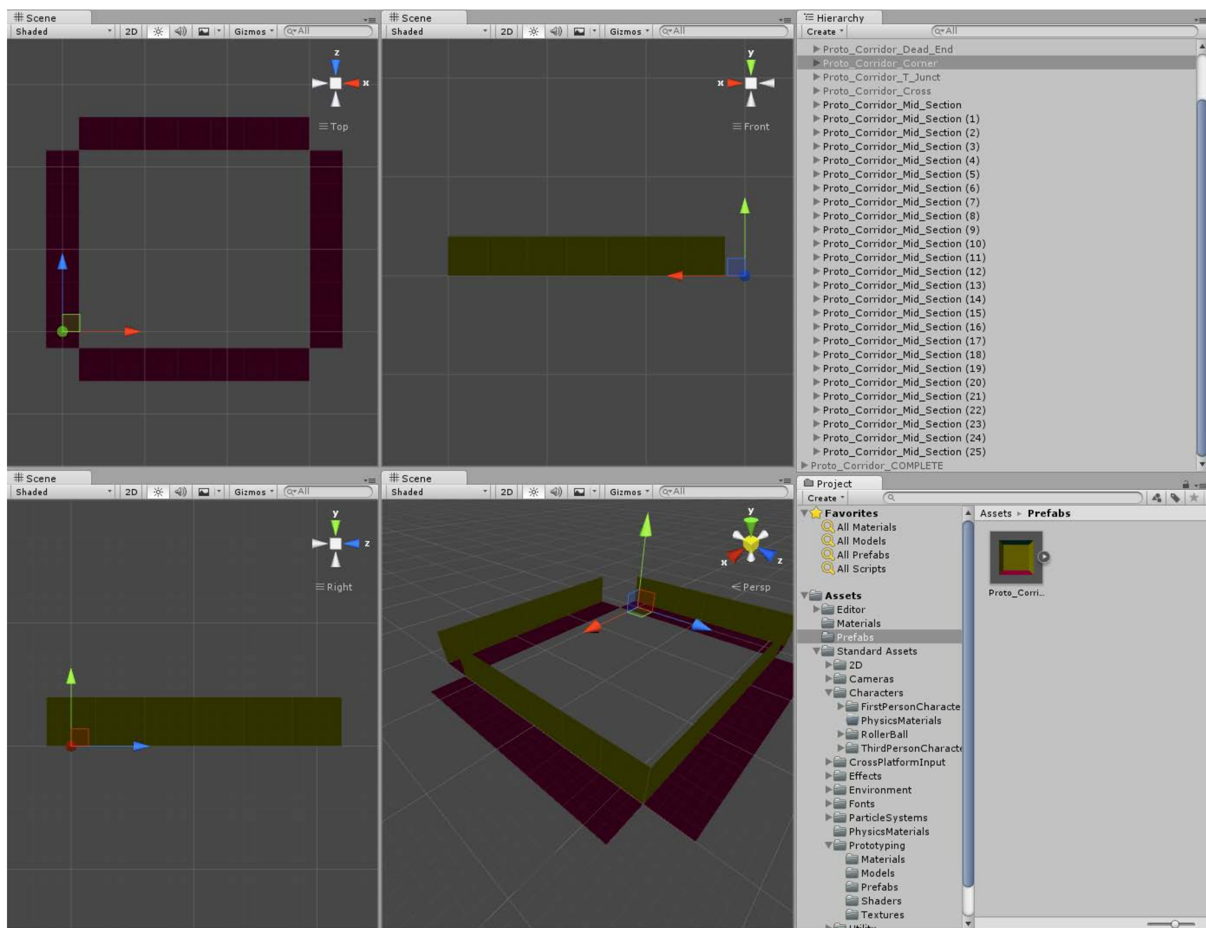
Using the Top Ortho Viewport (offers the best vantage point for level layout) duplicate the Mid_Section Group, Snap Drag (Hold Ctrl) to the top and start to build a 6 piece corridor (6 x 4 unit metres = 24 metres) section.



Now to start making an adjacent row of the same pieces, remember to keep an eye out for which way around the walls are!

- Duplicate the last piece,
- Rotate it 90 degrees
- Move it to an adjacent end (leave room for corners – 4 units wide – see below example)
- Duplicate to make this corridor longer
- If you want parallel corridors select the group and duplicate them

Continue to layout a corridor network and try different layouts (For quick “hacky” editing duplicate your Empty Group and then move it around, anything goes wrong with your design, you can just dump that group and restart from the previous one! Feel free to go back and delete sections to adjoining areas.



Above is an over simplified example (yours should be more complex) of reuse and duplication, and all similar pieces to give the map a general layout and indication as to where our joins and corners are to be. That way we simplify the task of placing the relevant sockets and can add more complexity.

Remember that these objects are still all paired with the Prefab we made earlier? Want to change that Pink to a more fitting themed texture? Update the Prefab data (highlighted earlier in this tutorial) and it will be reflected through your level. Excellent if you want to do a texture splat and see how it comes out for a play through.

Once you've completed a layout you're happy with, place a spawn point for an FPSController within your scene and walk around to play test your level. REMEMBER TO LOG YOUR PLAYTHROUGH!

Investigation – Project Relevance

Now that you have a set of objects to layout your level, check out different ways of laying them out from other games! Most games follow a simple corridor structure and diversify the environment to make it more interesting for the player using props and scene objects.

It's now up to you to find out:

- What layout you want to do
- Player flows and optimum routes
- The texture styles you're will/likely to use
- Lighting types – we've covered a couple already!
- Ramps/Stairs going up and down – changing the dynamic and flow of the level

Revise this tutorials information thoroughly, try different attempts with new scenes (you can add the prefab masters to lay out multiple levels)

Addition- Investigate: Scene Graphs – This will help you understand the simple elegance that builds the complex structures and functions within Unity

Next up we're going to look at loading a scene from another this is going to rely on your external map created in earlier tutorials and the interior you've created in this one.

MAKE SURE YOU HAVE A PLAYER SPAWN POINT IN EACH SCENE! It will load up fine, but you won't be able to walk around (we'll investigate persistent objects later)

Portal Box – Loading a Scene on Proximity

In this tutorial we'll look at attaching a script to an object, using that objects collider as a trigger that change some scene objects and allow the player to enter another scene.

Setup Build Settings

First off, before try to load a scene, we need to tell Unity that we're going to have the scenes within the build. (If you've already made multiple levels, please make sure you're naming convention is consistent – it is marked in your assessment)

- Go to **File | Build Settings…**
- Open Scene **02_Level_002** (if it's not already from previous tutorial) from your Project view
- Click Add Current in the build settings view
- Open Scene **02_Level_001** from your Project view
- Click Add Current again

Setup Scene

Add Cube

Add a new unit cube (1x1x1), rename it **Portal_Cube** in this tutorial, this object is going to be a parent to a spotlight, we'll allocate it to a prefab to ensure object consistency when scripting later.

Next we're going to edit the cubes collision parameters a bit.

With the **cube** selected in the **inspector**, change the **Box Collider** size to 4,1,4

Add Spotlight

With the cube still selected, right click its name in the hierarchy and select **Light | Spotlight**. Rename it **Trigger_Light** Now take note at this point to what this means. We now have a child spotlight attached to the cube, therefore when you position the light, it's origin should be the boxes origin NOT Global Space. This is called **Relative Positioning** in other words, it is *relatively positioned* in respect to it's parents position.

With that in mind let's set the spotlights values (if omitted, leave as default):

- **Position**: 0, 3, 0
- **Rotation**: 90, 0, 0

- **Scale**: 1, 1, 1
- **Range**: 10
- **Spot Angle**: 45.3
- **Intensity**: 8
- **Bounce Intensity**: 1

We have our objects all primed and ready now to add the brains behind the scenes via some scripting.

Turn off the spotlight for now by unchecking its box in the inspector.

Add Script – Evaluate, Develop, Test

Now let's identify what we want from this Portal Box exactly. First off we want it to **teleport** us **between levels** so we need a **load scene feature** when the user enters a **certain area** (colliders anyone?). However, we also need some **visual cues** to tell the user that they **can do something** with the object, so we'll likely need **shader values** and access to the **spotlight child components**. Lastly, rather than just sucking the user into a new level without telling them what's happening, we should add a **text prompt** and wait for the **user to confirm interaction**.

So to trim the feature requirements above:

- Load Scene on key press
- Manipulate shader and spotlight on proximity
- Text prompt on proximity

Easy! So let's get started.

- If you haven't already got a **Script Folder** in your root folder of **Assets** add it in the project view now.
- Add a new **C# Script** in that folder called **Portal_Box_Load_Level**
- Select your **Portal_Box cube** from the scene Hierarchy
- **Drag and drop** the **script** onto the **inspector view** of the **Portal_Box**

Make Prefab

Navigate to the Prefab folder (if you don't have one, add it now to your root Asset folder) in your project view and **right click | Create | Prefab,** rename to **Portal_Box**

Select your **Portal_Box** object in the scene hierarchy view and drag it onto the newly created Prefab this will allows to place it on other scenes (provided we account for it in the script of course!) (*If you don't remember what this is all about refer back to the previous interiors tutorial now*)

Add to other Scene from Prefab

With your current scene open, ensure the placement of your object is adequate and accessible to player for the allocated collision area, if not adjust it now.

Open up the scene that you wish to link to (in my case **02_Level_002**, the interior level) and add a **Portal_Box** Prefab by dragging and placing it in the scene (similar to how you would do with the **FPSController**). Again, make sure this accessible by the player character.

Also, in each scene ensure that you have an **FPSController** object added as the script is going to look specifically for it when doing collision checks later. If you get errors to references check that you have renamed this to something like **Player**. Lastly, before begin making the mechanics of the sequence, put the **FPSController** and **Portal_Box** somewhat close, no point running for miles to test a feature!

Sequence is Set – Add the Scripting Magic

Let's identify some key member variables we're likely to use in the upcoming script events.

- Shader access – we'll need a renderer
- Spotlight Child object – to manipulate the values throughout the script
- Player Text – a Boolean to track whether the UI should display text – UI sits in its own event loop

Ayrshire College

Open up the script **Portal_Box_Load_Level** in **MonoDevelop** or **Visual Studio** and update with the following code and outline object dependencies:

```
using UnityEngine;
using System.Collections;

/***
 * This Script is designed to work with the Portal_Cube Prefab
 * Required Objects:
 *  Cube - Name Portal_Box
 *  Spotlight - Name Trigger_Light
 *  Player Object - Name FPSController
 *  Level List: - (Find a better way, from file perhaps??)
 *      : 02_Level_001
 *      : 02_Level_002
 *  Please rename these accordingly if you change them within your project
 ***/

public class Portal_Box_Load_Level : MonoBehaviour
{
    Renderer rend = null; //Access the shader values attached to the Cube
    Light childSpotlight = null; //Access Spotlight Components
    bool showText = false; //TextUI Selection – Pass to UI from Cube Collider

    // Use this for initialization
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Excellent, now we have a renderer to access our cubes shader values, a local light component and a Boolean check.

Locate Shader Variables

Now we need to find the shader value we're going to use to update the cubes colour. To do this:

- Go back to Unity Editor
- Select your **Portal_Box** and locate the **shader** where it says **Default-Material Shader Standard**.
- **Right Click** the material and select **Edit Shader…**

This will bring up a fairly long list of different values, the one that concerns us is the top property called **_Color**. We're going to use that string value to track it in our script.

Note: You don't have to have the shader attached to view these values, it's just a convenience that we have it already attached to our cube.

Update Renderer

Return to the script editor.

So from looking we discovered 2 key things one the name of the Shader: **Standard** and two the name of the property we need in the shader **_Color**.

Now with those values in mind let's now initialise our Renderer object and update the shader values to start as **Red** (Off status) and change to **Green** (On Status).

```
    // Use this for initialization
    void Start()
    {
        rend = GetComponent<Renderer>();
        rend.material.shader = Shader.Find("Standard");
        rend.material.SetColor("_Color", Color.red);//Note how to find these
values in Unity
    }
```

Note the shader name and the shader property that are accessed using the Find function and setting the values. If you want to manipulate shaders in code, this is a really quick and easy way to do so.

That said you way want to avoid situations where you call the renderer a lot as it can be a heavy weight in terms of object memory impact and potential to cause slow. Don't pre-optimise, try it out and optimise if need be. We'll profile later, for now let's stick with this format.

Detect FPSController Proximity

To do this we need to hook into the collider events and spot the player object by creating some overloading methods. Add these methods between your **Start()** and **Update()** methods (There's always time for code cleanliness!)

```
    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
        }
    }

    void OnTriggerStay(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
        }
    }

    void OnTriggerExit(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
```

```
        }
    }
```

Update Shader Values

Using the same code as before for the Start() method, simply omit the initialisation. However, we add a couple more lines to ensure that the renderer isn't null when this code attempts execution.

```
    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            if (rend != null)
            {
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.green);
            }
        }
    }

    void OnTriggerStay(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
        }
    }

    void OnTriggerExit(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            if (rend != null)
            {
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.red);
            }
        }
    }
```

Update Spotlight Values

Now we need to access a child component of this object (the cube) to update its enabled status depending on proximity.

```
    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            //Debug.Log("OnTriggerEnter() - FPSController Enters Box");// -
Uncomment if required for debugging
            if (rend != null)
```

```
            {
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.green);
            }

            childSpotlight = this.GetComponentInChildren<Light>();
            if (childSpotlight != null)
            {
                if (childSpotlight.name == "Trigger_Light")
                {
                    childSpotlight.enabled = true;
                }
            }
        }
    }

    void OnTriggerStay(Collider other)
    {    }

    void OnTriggerExit(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            if (rend != null)
            {
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.red);
            }
            childSpotlight = this.GetComponentInChildren<Light>();
            if (childSpotlight != null)
            {
                if (childSpotlight.name == "TriggerLight")
                {
                    childSpotlight.enabled = false;
                }
            }
        }
    }
```

Add GUI Text

Great! Now we have proximity events change the colour of the cube and whether the light is on. Last user experience detail that require now is to add a text prompt to the users screen.

In Unity's UI system you hook into via the overloaded method for OnGUI()

So let's add the code below the update() method, it's fairly self-explanatory and feel free to change the placements and/or text.

```
    void OnGUI()
    {
        if (showText)
            GUI.Label(new Rect(Screen.width/4, Screen.width/4, Screen.width,
 Screen.height), "Press [Return] to Leave Area");
```

```
    }
```

Load Level

Lastly, we have to the capture key event we just asked for from the player and translate that into loading a level. Since we want this to be a constant option for the player we'll put this code in the OnTriggerStay() method.

```
    void OnTriggerStay(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            if (Input.GetKeyDown(KeyCode.Return))
            {
                if (Application.loadedLevelName == "02_Level_002")
                {
                    Application.LoadLevel("02_Level_001");
                }
                else if (Application.loadedLevelName == "02_Level_001")
                {
                    Application.LoadLevel("02_Level_002");
                }
            }
        }
    }
```

Now in this piece of code you should pay close attention to 2 important things here:

1. I used the keycode instead of the string value for the key press, user input should try to use the fastest variable type available. Don't just blindly copy code! Analyse, evaluate, question! – I could be giving you dodgy code and you wouldn't even know…
2. The loadlevel methods checks to see what scene it currently resides in (the Portal_Box) and what level to decide it should to do. Now this has obvious shortcomings as you'd need to remember each level and portal and such.
    a. That said… You could look into a more optimal solution e.g. loading from a file list of possible levels, taking parameters for current and exit levels etc

That absolute last pieces of code to now add are the Boolean checks to update the GUI cycle as we have the check in the method but not yet setting the values anywhere useful. Let's fix that. Add the first one to OnTriggerEnter within the collision check and set to true with the player and the same for OnTriggerExit setting to false.

```
    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            showText = true;
            if (rend != null)
            {
```

```
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.green);
            }

            childSpotlight = this.GetComponentInChildren<Light>();
            if (childSpotlight != null)
            {
                if (childSpotlight.name == "Trigger_Light")
                {
                    childSpotlight.enabled = true;
                }
            }
        }
    }

…
<other code>
…
    void OnTriggerExit(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            showText = false;
            if (rend != null)
            {
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.red);
            }
            childSpotlight = this.GetComponentInChildren<Light>();
            if (childSpotlight != null)
            {
                if (childSpotlight.name == "TriggerLight")
                {
                    childSpotlight.enabled = false;
                }
            }

        }
    }
```

Playtest – Debug – Experiment – Extend

Now that the Portal_Box_Load_Level is complete and it's attached to existing Prefabs within your game world, you can now just hit play, go check your code works and enjoy warping between worlds via cube express!

Please re-read through the tutorial or ask for help if you're stuck. Outlined below is a full copy of the script.

You now have a tonnes of skills to go off and make your interior and your exterior a lot more interesting. You could introduce:

- Fast travel
- Infinite corridor sequences
- more fun and interesting interactive objects than a portal light cube!
- Go and experiment with all you've learned from these tutorials, write up your ideas, log your debugging and testing and most of all
- RELISH IN DESIGN YOUR GAME IN MORE AMAZING AND INTERESTING WAYS

Continue to add features each week and in your spare, make this project a portfolio piece worth remembering and one that is going to showcase how awesome you are right now!

Ayrshire College

Completed Script

```
using UnityEngine;
using System.Collections;

/***
 * This Script is designed to work with the Portal_Cube Prefab
 * Required Objects:
 *  Cube - Name Portal_Box
 *  Spotlight - Name Trigger_Light
 *  Player Object - Name FPSController
 *  Level List: - (Find a better way, from file perhaps??)
 *       : 02_Level_001
 *       : 02_Level_002
 *  Please rename these accordingly if you change them within your project
 ***/


public class Portal_Box_Load_Level : MonoBehaviour
{
    Renderer rend = null;
    Light childSpotlight = null;
    bool showText = false;

    // Use this for initialization
    void Start()
    {
        rend = GetComponent<Renderer>();
        rend.material.shader = Shader.Find("Standard");
        rend.material.SetColor("_Color", Color.red);//Note how to find these
values in Unity
    }



    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            showText = true;
            //Debug.Log("OnTriggerEnter() - FPSController Enters Box");// -
Uncomment if required for debugging
            if (rend != null)
            {
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.green);
            }

            childSpotlight = this.GetComponentInChildren<Light>();
            if (childSpotlight != null)
            {
                if (childSpotlight.name == "Trigger_Light")
                {
                    childSpotlight.enabled = true;
                }
            }
        }
    }
```

```
    void OnTriggerStay(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            //Debug.Log("OnTriggerStay() - FPSController Stays in Box");// -
Uncomment if required for debugging

            if (Input.GetKeyDown(KeyCode.Return))
            {
                if (Application.loadedLevelName == "02_Level_002")
                {
                    Application.LoadLevel("02_Level_001");
                }
                else if (Application.loadedLevelName == "02_Level_001")
                {
                    Application.LoadLevel("02_Level_002");
                }
            }
        }
    }

    void OnTriggerExit(Collider other)
    {
        if (other.gameObject.name == "FPSController")
        {
            showText = false;
            //Debug.Log("OnTriggerExit() - FPSController Exits Box");// -
Uncomment if required for debugging
            if (rend != null)
            {
                rend.material.shader = Shader.Find("Standard");
                rend.material.SetColor("_Color", Color.red);
            }
            childSpotlight = this.GetComponentInChildren<Light>();
            if (childSpotlight != null)
            {
                if (childSpotlight.name == "TriggerLight")
                {
                    childSpotlight.enabled = false;
                }
            }

        }
    }

    // Update is called once per frame
    void Update(){}

    void OnGUI()
    {
        if (showText)
            GUI.Label(new Rect(Screen.width / 4, Screen.width / 4, Screen.width,
Screen.height),
                "Press [Return] to Enter");
    }
}
```