Game Development Fundamentals With UDK

northwind87

Published with GitBook

Table of Contents

- Introduction 0
- Intro to Unreal Editor, Maps, and BSP 1
  - Layout and Viewports 1.1
  - Creating a Basic Level 1.2
  - Content Browser 1.3
  - Lab 1 1.4
- Importing Models, Kismet, and Matinee 2
  - Importing Outside Content 2.1
  - Lab 2 2.2
  - Resources 2.3
- Landscapes 3
  - Lab 3 3.1
- Introduction to Scripting 4
  - Lecture Topics 4.1
  - Tasks and Resources 4.2
  - UnrealScript Basics 4.3
  - Engine Events, Super, Debugging 4.4
  - GameInfo, Map Association, WorldInfo, UTGame 4.5
  - Lab 4 Part 1 4.6
  - Lab 4 part 2 4.7
- HUD 5
  - Lecture Topics 5.1
  - Actor Intro 5.2
  - Iterators, AllActors, Tracing 5.3
  - UDK Characters, Controller, Pawn 5.4
  - 3D Math Basics, Vectors, Direction, Distance 5.5
  - Input Binding Basics 5.6
  - HUD & Canvas Intro 5.7
  - Lab 5 - Canvas, Input, Vector Subtract 5.8
- Camera 6

- Lecture Topics 6.1
- Camera Basics 6.2
- Camera Control 6.3
- Lab 6 - Camera Manipulation 6.4

Advanced Canvas, Basic Picking 7

- Lecture Topics 7.1
- Interface Classes 7.2
- Objects, Self 7.3
- PlayerInput 7.4
- PlayerController, Rotation, Firing 7.5
- Coding Editor Resources, Canvas, Texture2D 7.6
- Input Binding Advanced 7.7
- Lab 7 - Advanced Canvas, Basic Picking 7.8

Collision, Inventory, Basic Weapons 8

- Lecture Topics 8.1
- Collision Basics 8.2
- Component Basics 8.3
- Useful Components 8.4
- Spawn Review 8.5
- Lab 8A - Collision, Inventory, Basic Weapons 8.6
- Lab 8B - Basic Weapons, Projectiles, Aiming 8.7

AI, States, UTBot 9

- Tutorial Overview 9.1

WIP - Advanced Weapons 10

- Lecture Topics 10.1

Game Development Fundamentals

This book looks at how to build games using the unreal developer kit (UDK). It also serves as the subject notes for GAM537 and DPS937 at Seneca College.

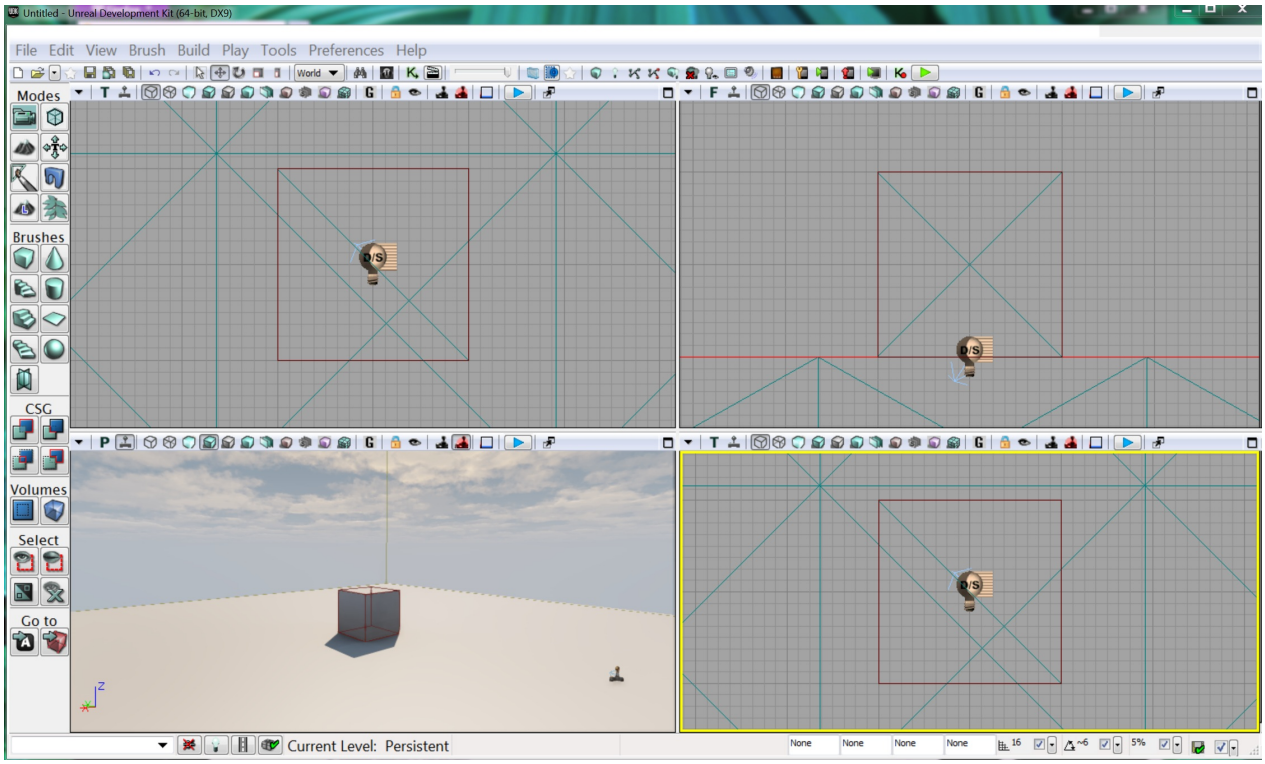
Click here to visit the Seneca GAM537/DPS937 Wiki site.

The unreal editor is the main tool for the creation of your maps. It is also where you place your art assets. Some very rudimentary scripting can also be done via the kismet and matinee system.

In this first chapter, we will explore the basics of the unreal editor. This includes how to navigate your map, how the layout of the application works as well as basic work with brushes, materials, static meshes and lighting.

Layout and viewports

Unreal Editor's basic layout is similar to that of different 3D content creation tools such as 3DS Max or blender in that there are multiple viewports. The controls on them however are different.



In the screenshot above the 3 viewports that are grey with a grid are the orthographic viewports (Top, Front, Side) and the 4th viewport with a box is perspective viewport. You can change what is shown in a view port by clicking on the buttons marked P,T,F,S above the view port (P for perspective, T for Top, F for front and S for side). You can change the size of each view port by moving your mouse pointer between them until it becomes a 2 headed arrow, click and drag. You can also change the basic layout using the view menu.

Navigation

In Perspective Viewport

rotate camera to left/right	left mouse button drag left/right
move camera forward/backwards	left mouse button drag up/down or scroll mouse wheel fwd/backward
"head turn" on camera (imagine that the camera is your head and you are moving it)	right mouse button drag
move camera up/down/left/right	hold both mouse buttons and drag up/down/left/right

In Orthographic Viewport

Move camera up/down/left/right	left or right mouse button drag up/down/left/right
Zoom in/out	both mouse buttons drag up/down or scroll mouse wheel fwd/backward

What to do when your editor acts funny

Sometimes your editor will have behaviours that are unwanted or things just don't work. You can reset your UI by doing the following:

- Close unreal and unreal ed.
- go to My Documents --> My Games --> Unreal Tournament 3-->UTGame-->Config folder
- Delete all *.ini files

Creating a Basic Level

These are the steps necessary to create a basic unreal level. It is a good idea for you to try to follow the steps here to find your way around the Editor.

BSP/CSG Geometry

Early versions of Unreal used something called BSP geometry as their primary method of creating 3D content. BSP stands for binary space partitioning and is a data structure used in graphics. However, the more accurate term for the type of geometry created by unreal is Constructive Solid Geometry or CSG for short. In any case if you are reading documentation for unreal, know that the terms are interchangeable.

The way that BSP geometry is constructed is quite different from constructing a mesh in say 3DS max. Instead of editing vertices/edges/faces, BSP geometry is typically created by adding and subtracting geometry from each other. You constantly think about the world as being made of either solid matter or empty space.

In older version of the unreal editor the world that you create starts off in additive space. That is, the world starts off solid and you carve out the shape of your world from this solid space. In the current version of UDK, the word starts off in subtractive space and you add geometry to it.

BSP geometry is typically used for building things that are big and non-detailed (such as walls and floors). For anything with any sort of detail static meshes are far better.

How to work with BSP Geometry

Building geometry with bsp involves working with brushes. Inside unreal ed, the type of brush (and in fact the type of actors as you will see later) can be identified by the colour of brush which can easily be seen in any of the orthographics views.

Outline Colour	brush type	usage
Red	builder brush	used to build a shape, it does not itself create or remove any geometry
Yellow	subtractive brush	used inside solid space to carve out a hollow
Blue	additive brush	used inside hollow space to create a solid block

To create geometry

start with a builder brush to get the shape that you want. Now, since unreal typically starts off in additive space (ie empty space) what you need to do is add something solid to it. If you are building a purely indoor level, start by creating a giant block of solid that you can then carve your level into.

In any case the process of creating geometry with BSP is the same. Start by creating a builder brush. Standard shapes for a builder brush can be created by hitting any of the buttons on left under Brushes. If you left click the button a red brush outline will show up in the view ports. You can alter parameters for that brush (size, number of sides etc) by right clicking on the button instead. This will produce a dialog box that lets you set parameters. for the brush.

Hitting the "Build" button on the brush does not create any geometry, it simply builds the builder brush.

To create geometry, you will need to create an additive or subtractive brush from the builder brush. Now, since we are starting in additive space (ie all empty) you will need to "add" to it to create somethign solid. This can be done by hitting the "add" button under CSG.

Once you have something solid, if you wish to hollow it out, create a builder brush that fits into a block of solid, and position it in to place inside the solid. After this, hit intersect button (diagonally down to left of subtract) and then hit the subtract button.

The intersect button fixes the builder brush so that if any part of the brush is in additive space (ie any part of it is in hollow space) it will modify the brush so that the part of the brush that intersects with hollow space is removed. (the final builder brush is only the component that is inside the solid block). This will prevent subtracting from additive space (remove emptiness from emptiness) which can cause wierd visual artifacts. In general you should always hit intersect before subtract. Similarly you should always hit deintersect before you hit add. Deintersect is diagonally down and to right of add button.

Content Browser

The unreal content browser allows you to explore the assets (textures, materials, meshes etc.) that you can place into your level. There are many available and you should look through the packages to see what fits best with your world. When you first start the content browser, you will see a panel on the left that lets you explore different collections and packages. On the right , you can filter to search for the item you want.

When unreal content browser first starts, it may not have loaded the contents of the packages into memory and thus, you may see only a few of the assets in a particular package. To ensure that you are seeing all the content, right click on the package in question and choose ***fully load***.

What you will see for each of the various types of contents is a preview snapshot of the object. The type of object is labeled in each of the previews but it can also be identified by the colour of the border. To explore each type of content further you can double click and set different properties.

Context Menu

Once you select an asset in the content browser, placing it into your level requires the use of the context menu. With the asset selected, if you right click into one of your view ports, it will bring up the context menu.

The context menu will typically have an entry based on your selected asset. For example if you have selected a material in the content browser, then right clicking on a surface will bring up the "apply material" menu item. This menu item will apply or place the selected asset to your map.

Lab 1

Summary

In this first lab we will be creating a simple indoor level with UDK. You will also populate this level with items using the content browser and select textures for your surfaces

Learning Outcomes

- navigate the unreal editor
- learn to work with orthographic and perspective view ports
- create geometry with BSP brushes
- create lighting effects
- add assets such as materials and meshes
- build and test a simple level

Instructions

In this first level you will create a simple indoor level with UDK.

Your level must have the following properties:

- Must be have at least 3 rooms connected by hallways
- 1 room must be non-box shaped
- 1 stair case must be present to connect any of the two rooms
- lights must exist so that level can be seen when walked through

One way that such a level can be built is to follow the following instructions (you are welcome to deviate from these instructions as long as you meet the requirements of the lab). In all cases, do not use the "hollow" setting to your brush. Its actually somewhat easier to just use solid brushes.

- Start by creating a block of solid to carve your level out.
    - box shaped builder brush with the following dimensions: 2048 width, 2048 length, 1024 height
    - next hit the add button, this will create a big block of solid out of which you will carve your level
- check out what happens when you change between lit to unlit in your perspective view

- Build a room towards the left side of the solid.
  - using the box tool create a box builder brush with the following dimensions, 512 length, 512 width, 256 height position this into the big block of solid.
  - using the top view, place this box so that it is near the bottom left corner but do not let it get too close or you will create a hole in your room.
- using either the front or side view, ensure that there is at least 512 units between the top of the solid and the ceiling
  - Use the orthographic views to ensure that it is entirely inside the solid cube (red builder brush should be inside the blue brush in 2 different orthographic view ports).
  - hit the intersect button (if you did it right in the previous step, nothing will happen as your brush will not be intersecting empty space).
  - Hit the subtract button to carve out your room
- Build a circular shaped room as your second room.
  - Using the builder brush create a cylinder height: 512, radius: 256
  - Using the top view orthographic view port place this cylinder to the right of the rectangular room
  - Using the front view orthographic port, ensure that the bottom of the two rooms are aligned
  - hit intersect then subtract
- build a hallway connecting the two rooms
  - measure the distance between the two rooms, you don't have to be too exact about it as there is a neat trick to getting precise hallways. To measure, use the middle mouse button and drag from one point to another. it will give you the distance between the points
  - create a box with a height of 256, width of 256 and length that is slightly larger (say 50 units) than the distance between the two nearest walls of the rooms. Note that you may need to switch length and width parameters if the hall way is going the wrong way.
  - align the floor of the hallway with the floor of the two rooms.
  - place the hallway so that it is between the two rooms, note that it will stick into both rooms a little at this point
  - hit the intersect button, notice how the brush will change so that it fits perfectly between the room
  - hit the subtract button to create the hallway.
- build another room rectangular room directly to the "north" of the first room. dimensions for this room is 512 length, 512 width, 256 height
  - Using the front or side view, place this room so that the floor is 256 units higher up than the floor for the first room. Ensure also that it is at least 512 units away from the first room (as measured from the two walls nearest to each other
  - build this room like the others

- connect the rooms No matter what you will need to put stairs into your level to connect the rooms and they are a bit tricky because remember that you are carving out space from solid. So its not stairs that you are building, but rather the space above the stairs. To build stairs, do the following:
  - the default height for stairs is 16 units, you should keep this for now
  - the number of steps is equal the distance between the two floors/16. If your room is 256 units above, then it is 256/16 making it 16 steps.
  - Add 256 units to the first step. this will create space above the landing.
  - if you did this right, your brush will look like stairs that have been raised. flip the stairs upside down, align the top step to the floor of the higher up room. move it into place. You may need to rotate your stairs to do this and it is easiest in the orthographic ports
  - as usual, it intersect and subtract to carve out the space above the stairs
- create a hallway between bottom of stairs to bottom room.
- add lights:
  - right click to the place you want a light, and a context menu should pop up, choose actor-> point light. This puts a light into your level, manipulate it into position.
  - repeat for multiple lights so that you can see everything
- add assets:
  - go into the content browser to find a material for your wall. (you can limit your search for this). select it.
  - right click on the surface you wish to apply the material to. If you want to use the same material on other walls, you can either ctrl-click select more than one surface, or you can use one of the selection tools from the context menu.
  - choose to apply the texture.
  - To modify how the material is applied, you can right click on the surface and bring up the properties.
  - You can apply static meshes in exactly the same manner. Select a static mesh to make it show up in your context menu. Right click on where you would like to put it. Adjust it into the correct position.
- build the geometry and lights
  - there is a button in the menu at the top that has a lightbulb and a cube which should say build geometry and lighting
  - right click into level and choose play from here.

This chapter looks at how to add models to UDK. It also provides a lab on creating content through the kismet and matinee system.

Importing Models

Due to the fact that you are not going to be able to create the assets that you need for your game, you will may need to go beyond what is available in UDK for your art assets. This section of the notes will go over the following:

- a few useful places to find models
- what to look for when looking for models for your game
- minor fixes you may need to perform on your model and how to do them.

where to find art

Here are a few places where you can start your search for your models:

- opengameart.org
- turbosquid.com

What to look for in your game art

Good art does not mean it is good art for game. Aside from aesthetics, there are real technical limitations on models that you can use for games that are different than those that you can use for still images or films. The reason for this is that game art has to be rendered in real time. Thus art has to look good without using a lot of resources.

If you take a look at the models inside UDK you can see their triangle counts.You will notice that for most art assets (things that decorate the world) the number of triangles is limited to between 500 and 1000 triangles. Some assets are bigger (there is a human statue that is around 4500 triangles). However in modelling terms this is not a lot. It is also unlikely that there will be many instances of this human statue in a level so one art asset with 4500 triangles is not that bad.

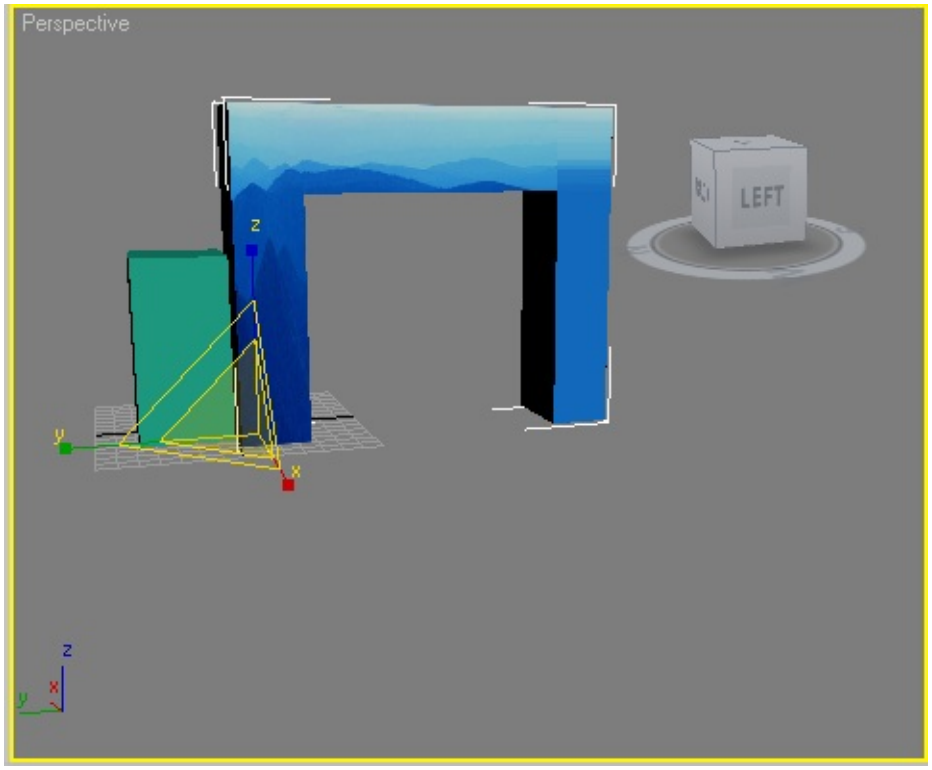
Art that is found on sites like turbosquid are not necessarily used for games. they may look awesome but they aren't meant for games. Generally speaking, you're probably safe for any art assets below 2k triangles. When looking for art, if the triangle count gets over 2k, ask if you really need it and how often it will be used. If the art asset is over 10K, it must be something that is absolutely necessary for your level. anything that goes much higher than that... you need to look for an alternate

Fixing Art Assets

When you get art assets sometimes there are things in it that you may not need. For example, the modeller may have placed a floor underneath the model so that the model looks like it is sitting on a floor when it was rendered. Since your floor will be your game world you will want to remove that. You can fix a model using a 3D content creation tool. There are several such tools available. For example 3DS Max, Maya or Blender. The instructions on dealing with export below are for 3DS Max, which is also available in the lab computers.

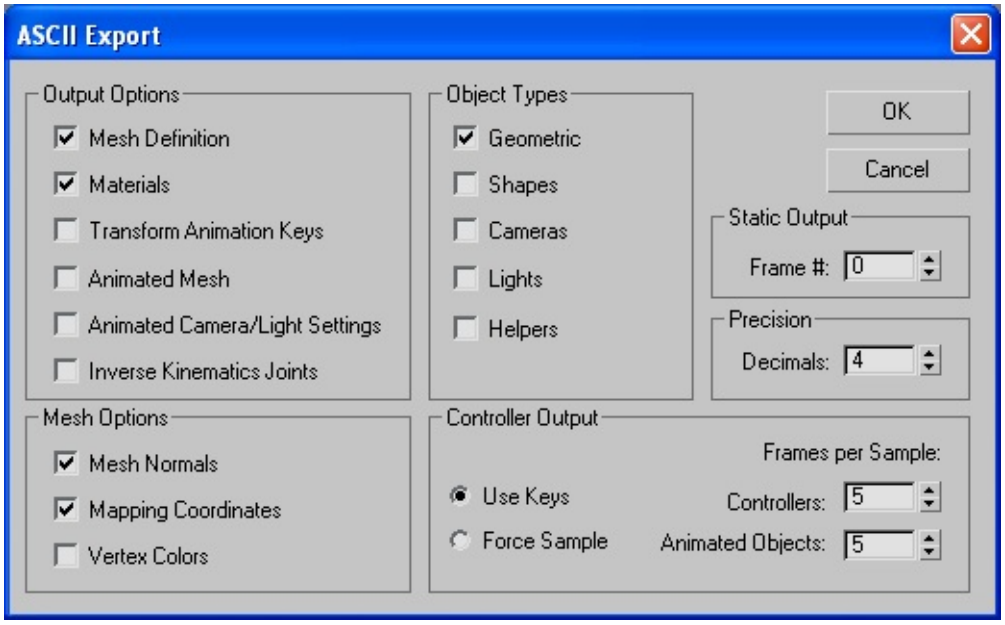
Scale your model

16 units is 1 foot. Before you export your model you should size it so that it looks appropriate in your world. You can do this by creating a box that is 6 feet tall and 1.5 feet wide (96 X 24). this is the approximate height of an Unreal character. Using it as a guide do a uniform scale of your object so that it is appropriate



Export

You can export a model as an Ascii Scene Export (ASE) from max. Unreal understands this format and will load it.



You can also export using FBX which is a format defined by autodesk.

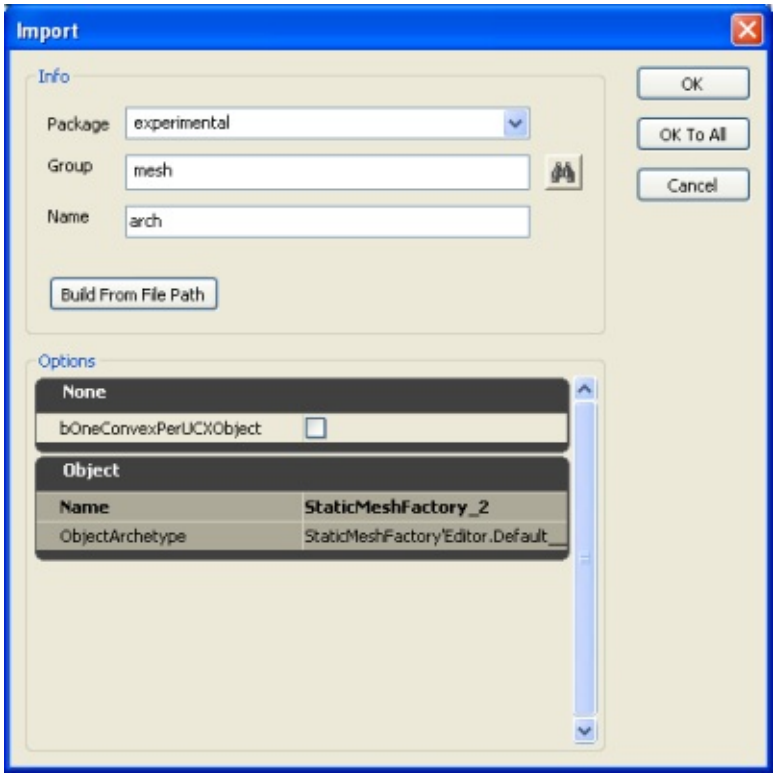
Importing Into Unreal</h2>

Once you import your mesh and texture into Unreal, you will be able to see the package in the package list of your content browser (if not you can navigate to it anyways).

Importing the Mesh

To import a static mesh, in your content browser click the import button (located under the package list), navigate to your ase or fbx file to import it. For package name choose something OTHER THAN MyPackage or the same name as your map!.
Group name should be something like meshes. The name of your mesh cannot have any

spaces in it. If your original ase/fbx file had spaces in the name, you will need to edit this to ensure it aligns with unreal naming conventions.



Importing the texture

To use a texture on a model you must create a material with it. To do this, start by importing the texture into your package.

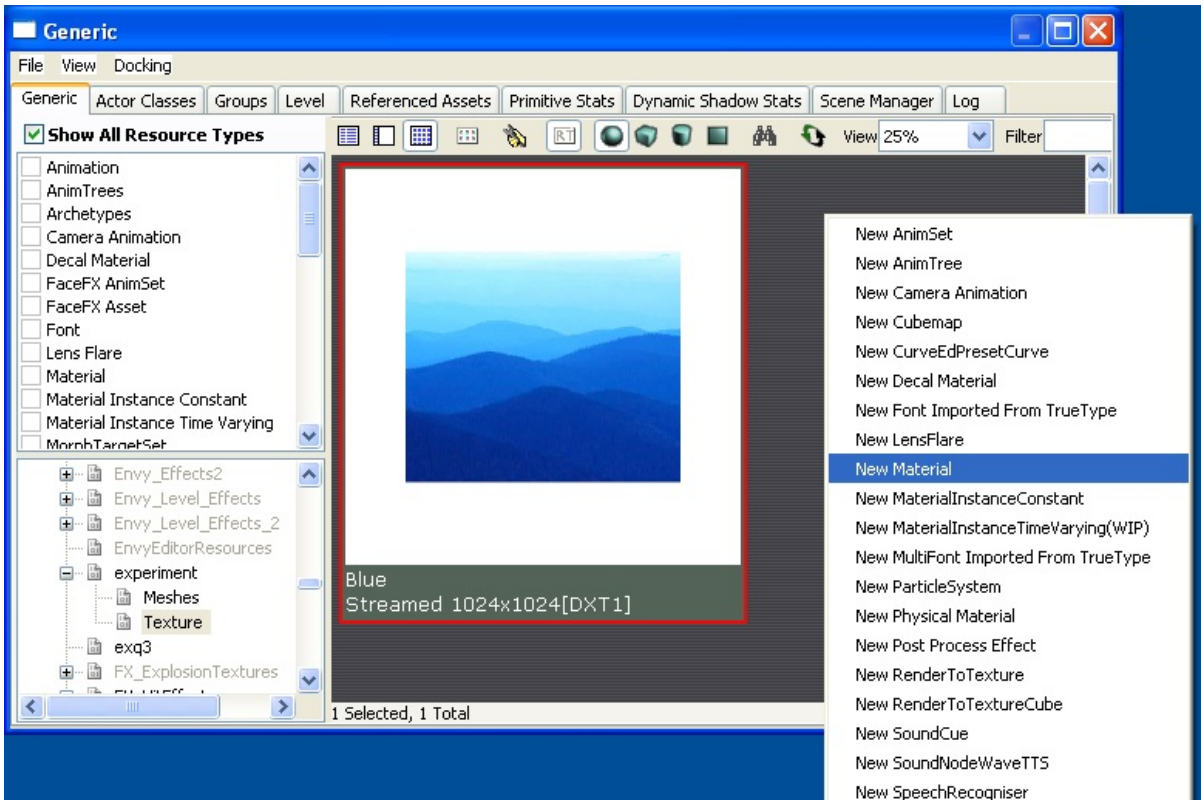
In the content browser:

- hit the import button
- choose your texture. NOTE: not all image formats are supported. You may need to convert your image to a supported form (bmp, tga, dtx, etc) which can be done using programs like photoshop or gimp.
- Enter your package, group and texture name into each of the fields in the popup box. NOTE: the image name cannot have a space in it.

Once your texture is imported, you can make a new material by doing the following:

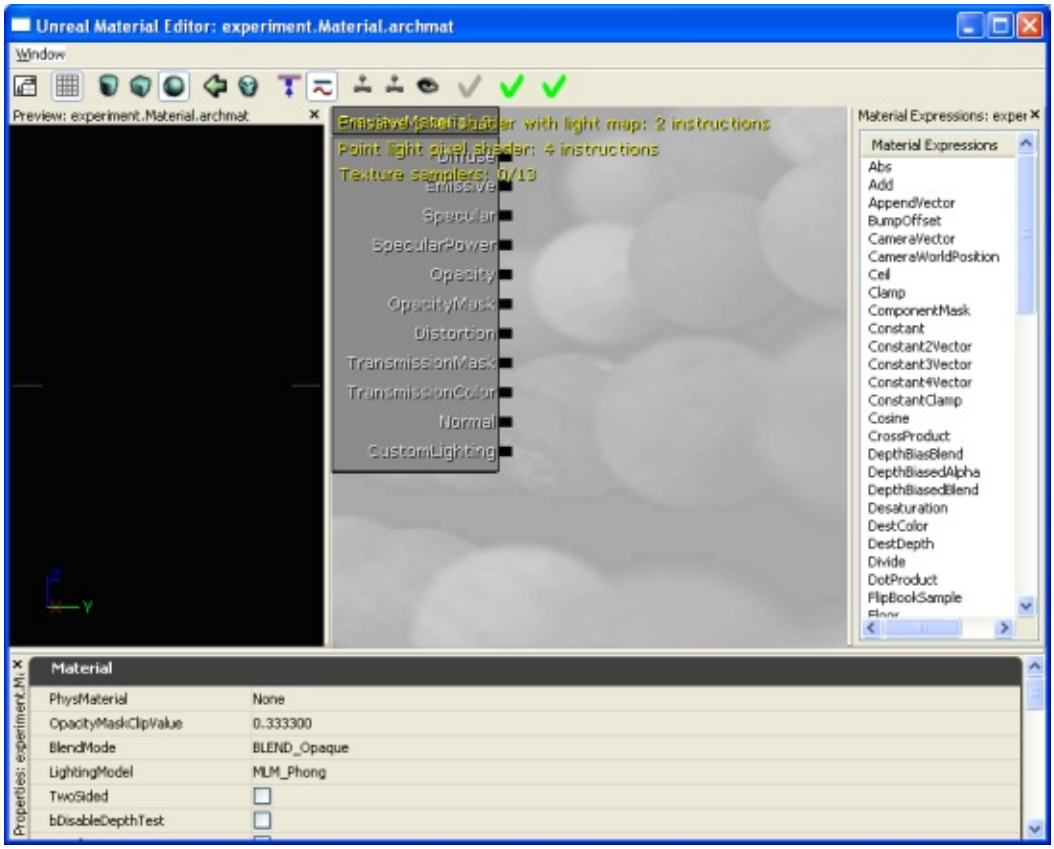
- Click on your texture so that it is selected
- right click in the grey area of your texture browser (the pane where the samples show up but just in an occupied spot).

- from the context menu choose new material.

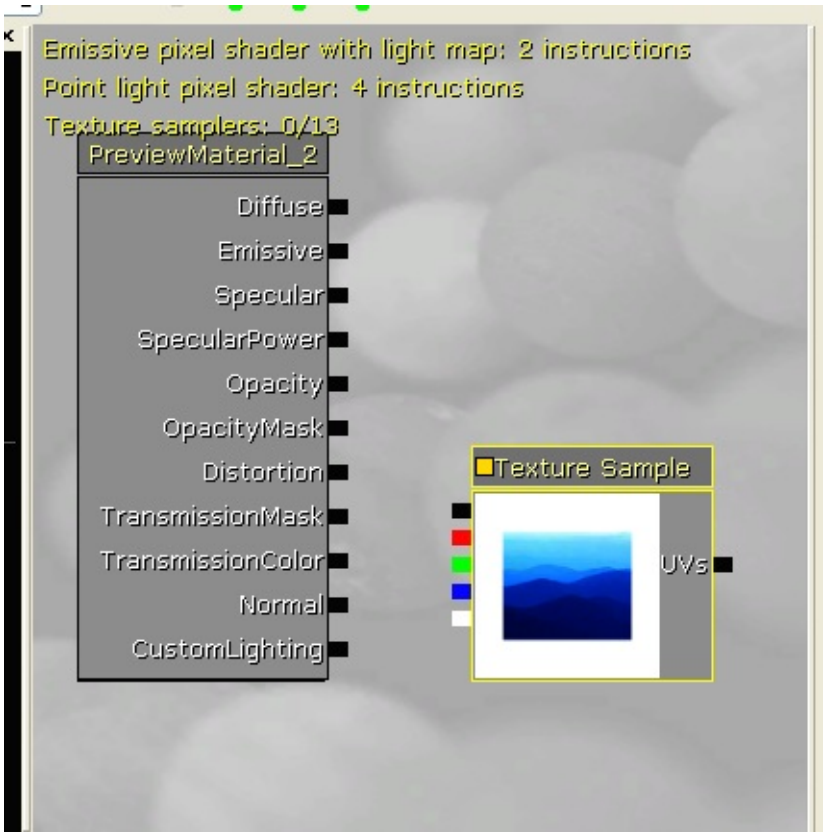


- this will once again bring up the new package item dialog box. Fill in package name as being same as what you were using for your mesh, group should be material, and name is whatever best describes your material

If you did everything correctly you should be able to see something like the following:

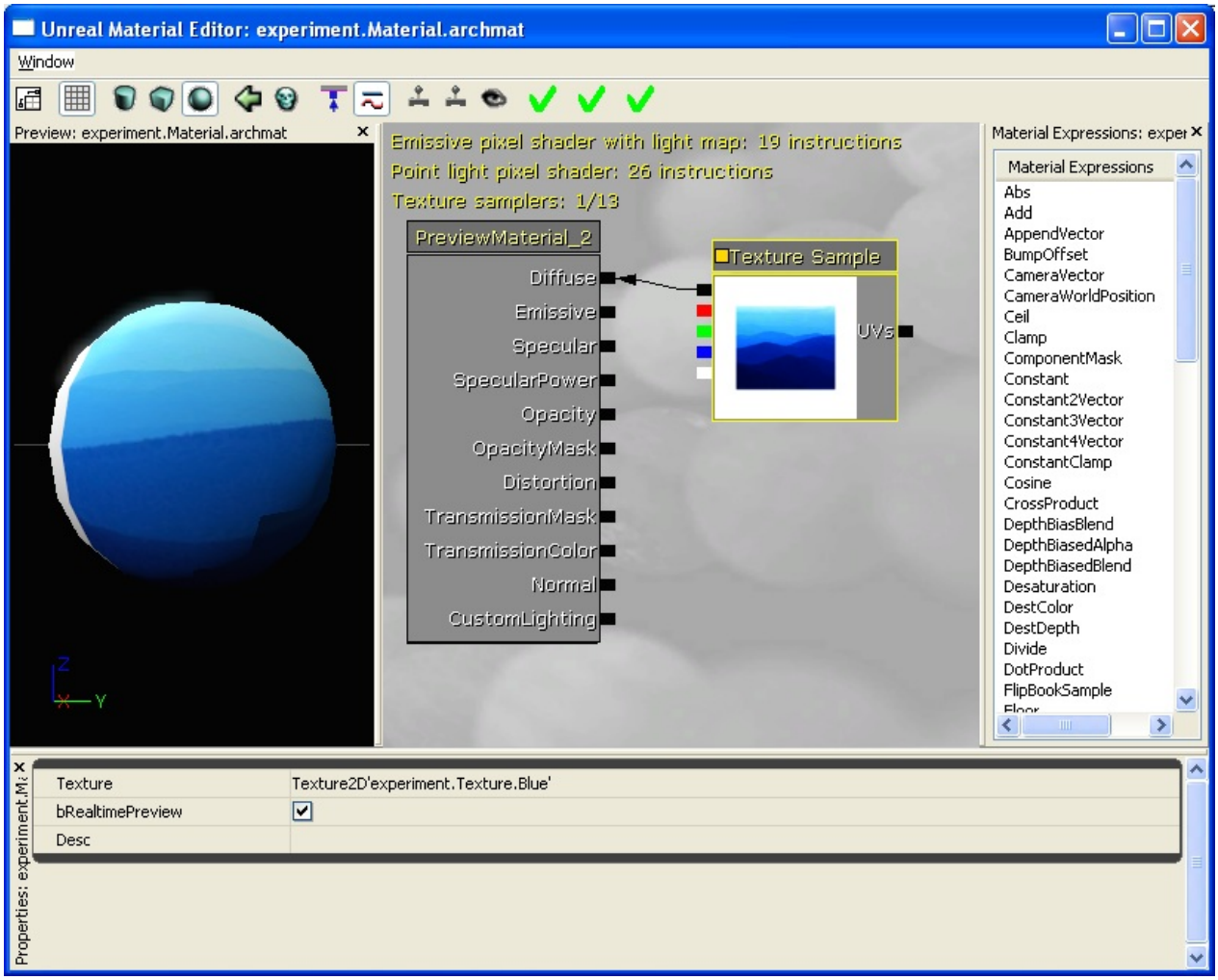


clicking on a blank area in the middle panel (where it looks like there are little spheres) choose ***New TextureSample*** from the context menu. As long as you had your texture selected, you will see it show up in a little box in that area.



If you did not have it selected, the sample shown will look black. However you can apply your texture by selecting it in the content browser window, then going back to the material editor, right clicking on the texture box and choose "use current texture" from the context menu.

Once you have the texture sample loaded into the material editor, connect the diffuse dot with the top dot black dot in the texture sample. You should now see a sphere with your texture applied.



Click the close box and it will ask if you want to save it.
Choose yes and your new material is now part of your package. This however doesn't save your package, choose file save from the content browser window in order to save the package itself. Remember to name your package something different than your map!

Applying the texture to your mesh

In the content browser window double click on the mesh you wish to apply the texture to. This will bring up the static mesh editor dialog box.

- Select the material (not the texture) for your mesh from the content browser window
- In the static mesh editor window expand:
  - LODInfo-->[0]-->Elements-->[0]
  - Click on the Material item (it probably says none on it) and click the arrow to apply the selected material.

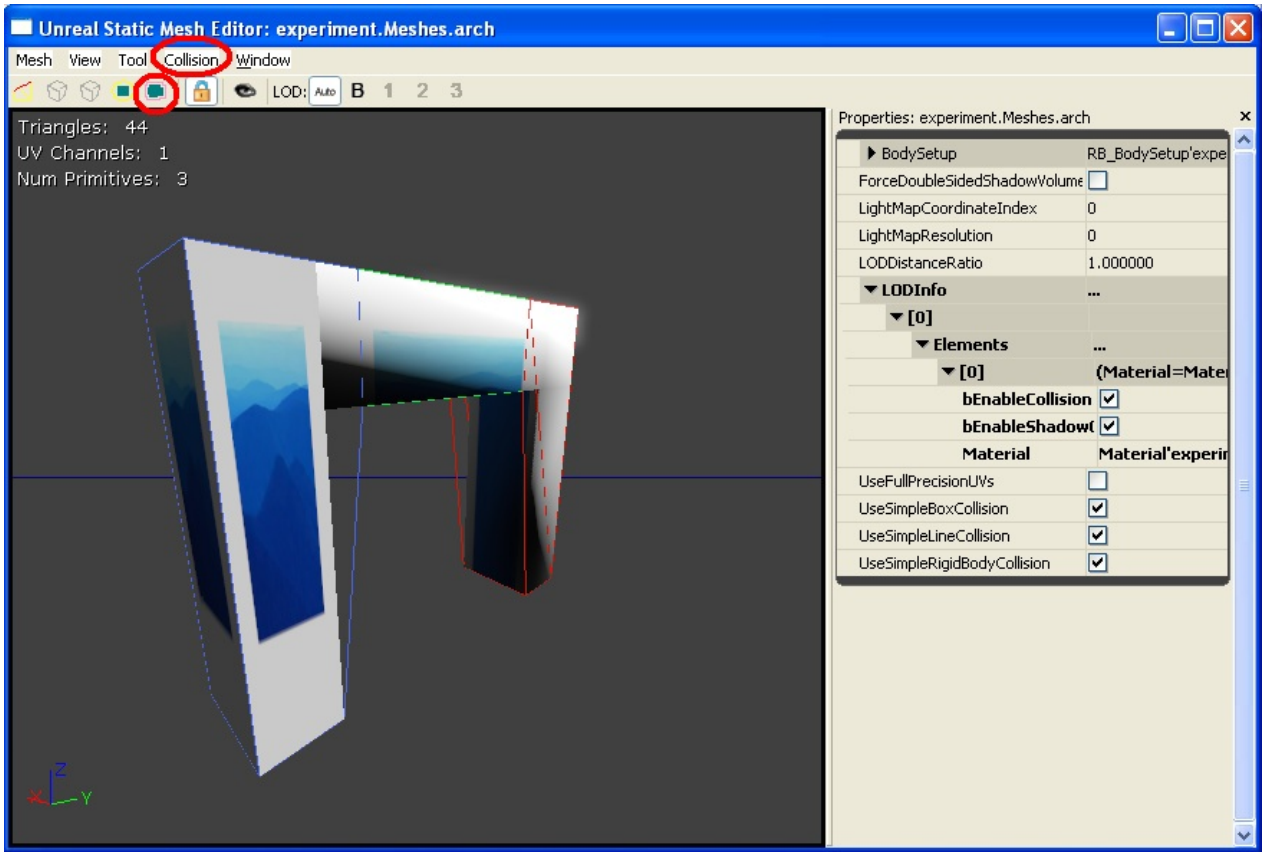
Collisions

When you bring in your mesh for unreal it will typically not have any collision on it. That is, you can run right through it like its not there. Depending on the usage of the mesh this might not be an issue.
However, if you need your mesh to be more solid you will have to add collision to the box.

There are two ways that you can do this. You can either create a custom collision mesh in 3DS max and bring that into unreal or you can use the tools supplied by unreal to make this mesh. It is easier to do it in unreal but less exact. Depending on the situation, it may be worth your while to create a custom collision mesh.

Creating a Collision Mesh Inside Unreal

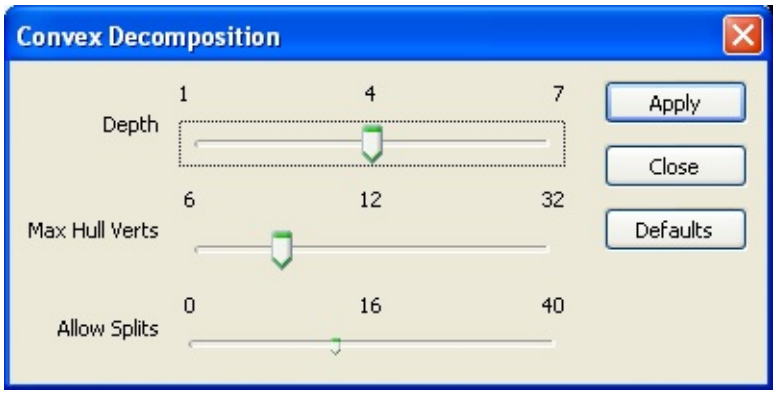
Firstly in order to see what the collision mesh looks like you need to click on the show collision button. Do this before applying collisions so that you can see what it looks like:



For simple meshes where a simple bounding box will do, You only need to choose 6DOP - simplified collision from the collision menu. However, that may not always work. For example, suppose you have the archway as shown in the image above. If you use a simple bounding box, you will not be able to run through it.

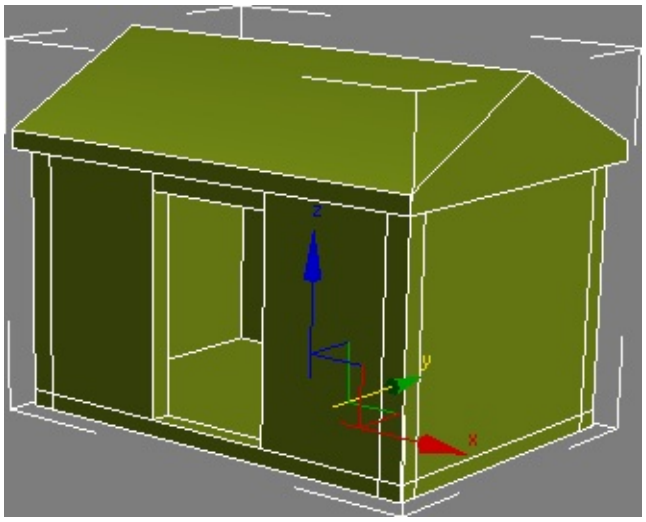
In general any convex shape (think of i as shapes without "dents") can be handled with a simple bounding box. (one of the 6DOP or 10Dop selections will work). However, for shapes that are concave (such as this arch) you will need to something more complex. To do this choose auto convex from the collision menu and it will bring up a dialog box. Adjust the

sliders as low as possible while maintaining the mesh in a way that is satisfactory for what you want. You can adjust and apply the sliders as you go to see how the collision mesh looks.



Custom Collision

Depending on the shape of your object, you may wish to create a custom collision box for it. While autoconvex will do a alot for collisions, it may not be enough to handle what you need. If that is case, a custom collision box is the way to go. To create a custom collision mesh you will need to add a mesh in 3DS Max to your model and export that again along with your original model.

Suppose you have made the house as shown below and you wish to add a custom collision model to it. 

The easiest way to do this is to create a number of simple primitive geometry and place them where you wish for collision to be detected.
For this house, you can create a box for the floor, a triangular prism for the roof and 5 boxes for the walls (3 for the side and back and two separate ones for the front. Each of these objects needs to be named with the following conventions:

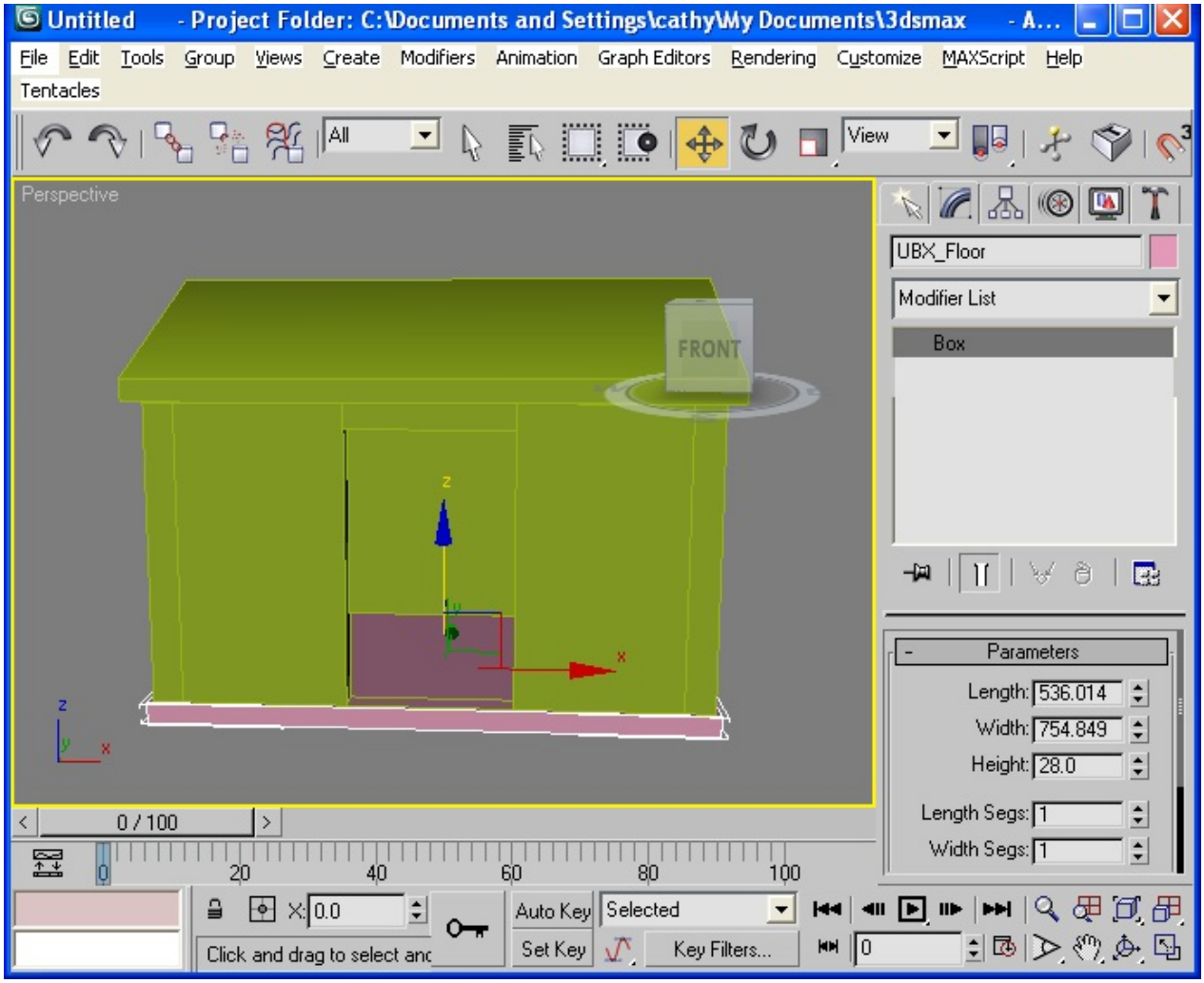
- For unaltered boxes prefix the item with UBX_
- For unaltered spheres prefix the item with USX_

- For custom convex objects prefix the item with UCX

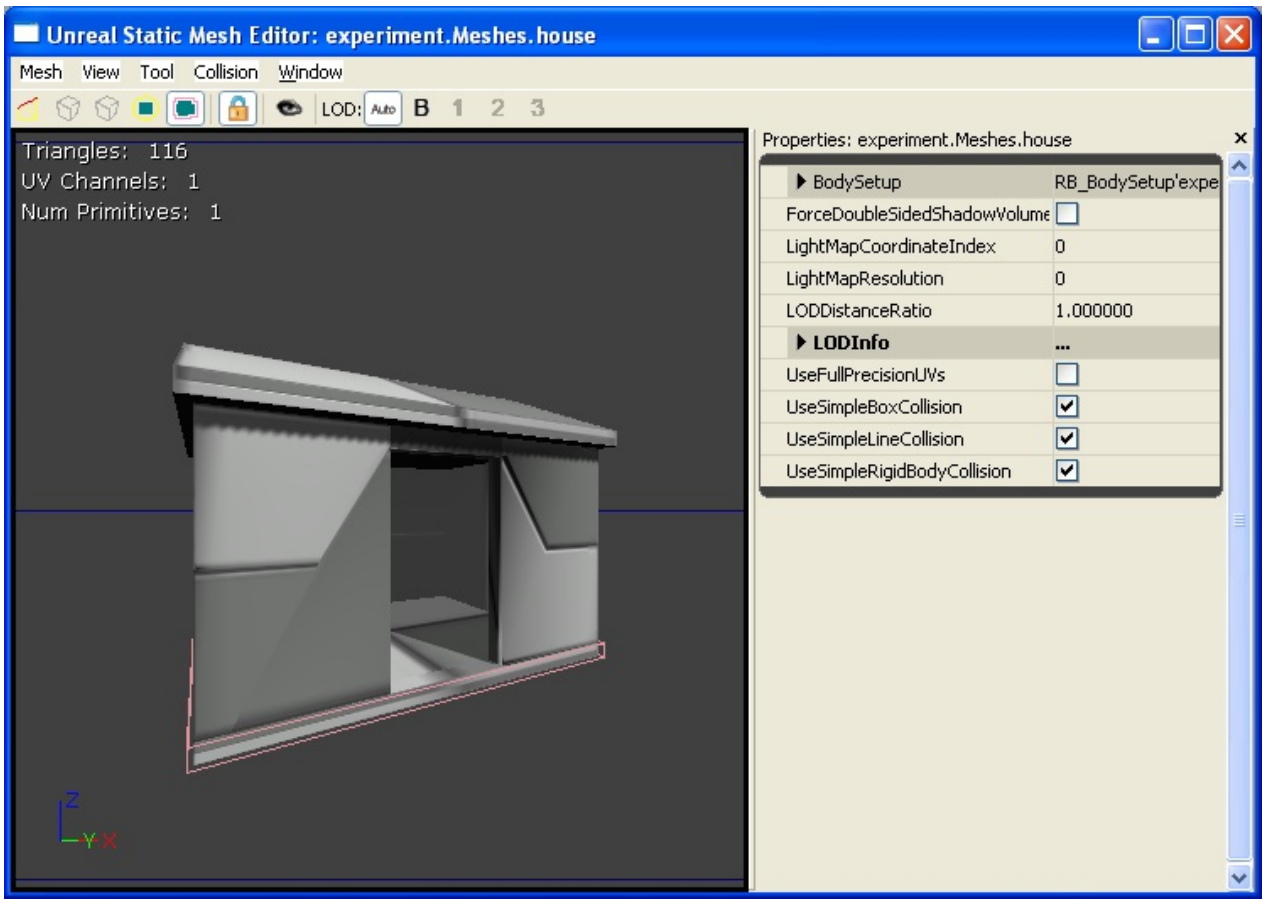
Unaltered boxes and spheres means that they cannot have altered it beyond the basic settings used to build the primitives.

Custom primitives must not be concave (ie no indents).

Below shows the same house with a custom collision box on the floor and the naming convention for that box.



Once you have completed creating the custom collision boxes select all the objects (model + all collision boxes and export as usual. When you bring it into unreal, your object will have a custom collision box.


Unreal Static Mesh Editor: experiment.Meshes.house
Mesh View Tool Collision Window
LOD: Auto B 1 2 3
Triangles: 116
UV Channels: 1
Num Primitives: 1
Properties: experiment.Meshes.house
BodySetup
RB_BodySetup'expe
ForceDoubleSidedShadowVolume
LightMapCoordinateIndex
0
LightMapResolution
0
LODDistanceRatio
1.000000
LODInfo
...
UseFullPrecisionUVs
UseSimpleBoxCollision
UseSimpleLineCollision
UseSimpleRigidBodyCollision

Lab 2

Summary

This lab goes through the Kismet and Matinee System. In it you will create a door that is set to open if a user runs into a trigger and close after it.

Learning Outcomes

Learn how to hook up a simple kismet and matinee system

Submission Requirements

Once complete, show your work to Cathy or Hasan

Instructions

Add a mesh for the door

If you are adding a mesh that will move as part of a kismet matinee sequence you cannot add it as a regular static mesh as it will not be able to move. Instead you will need to add it as an InterpActor. This is done in the same way as adding a regular static mesh except that you will need to choose a different menu item.

Notice that the mesh outline in the orthographic viewports are shown in pink and not blue

Setting up mesh properties

Select the mesh and right click to bring up the interpActor properities dialog box. In that dialog box choose the following settings:

- In the Collision Tab, change the collision type to COLLIDE_BlockAll
- In the DynamicSMActor tab expand the StaticMeshComponent--> Lighting make sure

the bCastDynamicShadow is unchecked

If your mesh looks black (instead of showing the texture) you will also need to turn on the lighting.

- In the DynamicSMActor tab expand the LightEnvironment category. Expand the LightEnvironmentComponent and make sure bEnabled is checked.
- Right click in your map and choose Add Actor --> Add Trigger from the context menu. A light switch icon should now show up on your map.

Adding the Trigger

To see how what the collision box looks like hit the **c** key.

to change the size open the Trigger's property dialog box, Trigger Tab-->CylinderComponent category and edit the CollisionHeight and CollisionRadius values.

Kismet

Although the trigger and interpActor are in the scene you still need to hook them up so that when someone hits the trigger, the interpActor moves.

To do this we must now set up the Kismet/Matinee sequence.

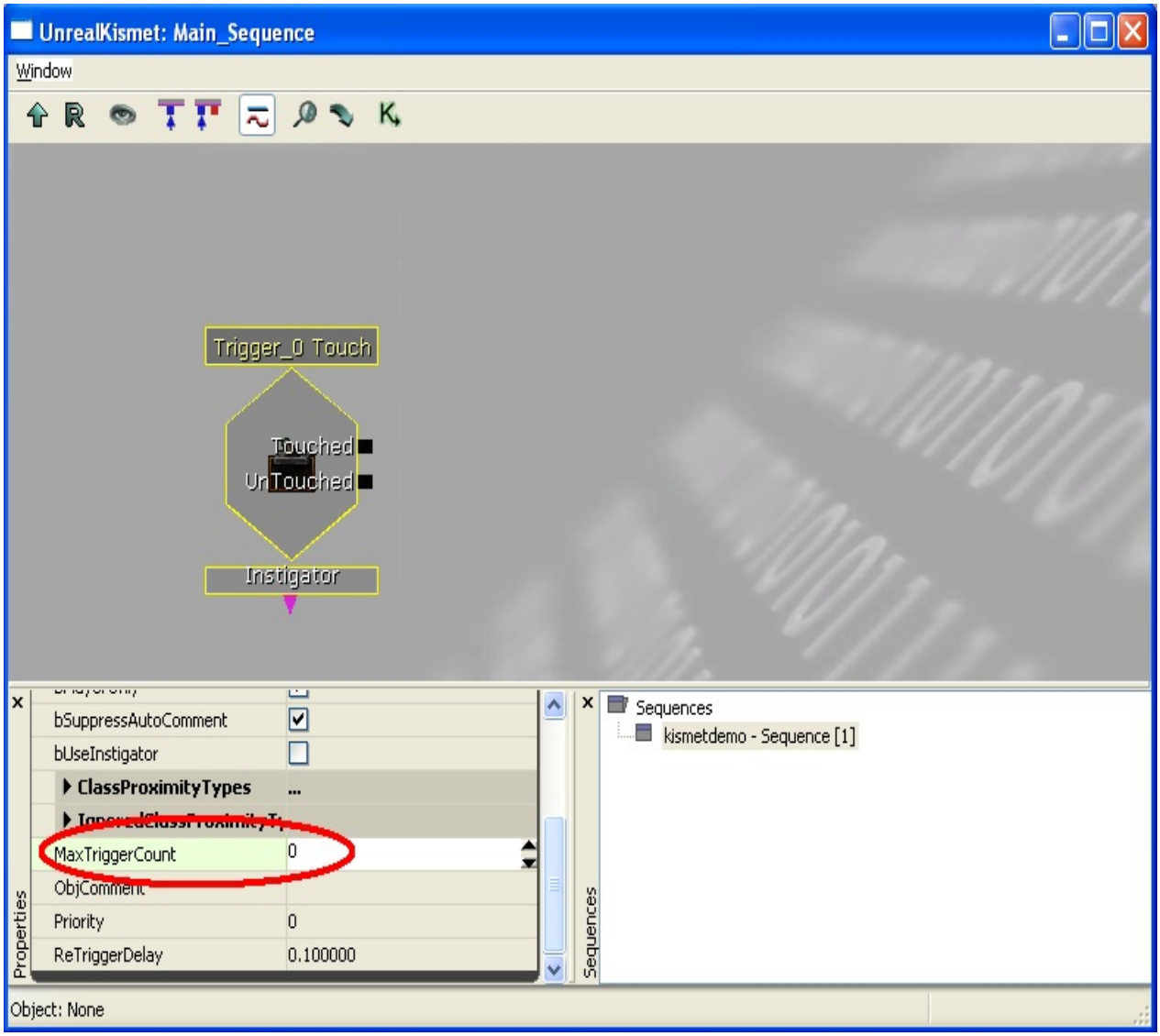
Opening the Kismet Editor

Hit the Kismet button on the toolbar ()

Select the Trigger from your map and in the kismet ediotr window right click and choose **New Event Using Trigger_0 --> Touch** (or whatever your trigger's name is... you can find this out by selecting the object in the map. Its name will show up at the bottom status bar). This means that when the trigger is touched or untouched the associated matinee (we have not made this yet) will be played.

You may also wish to change the MaxTriggerCount so that it works more than once (the default is 1). To do this change the value for MaxTriggerCount to 0. You will also note that if you did everything properly your kismet object shows the actor it is associated with and also

shows how the trigger will get activated. Below is a picture of what you should see if you did this part properly.



Matinee

Matinee is a sequence that can play once some even occurs. Think of it as a short animation.

Creating the Matinee object

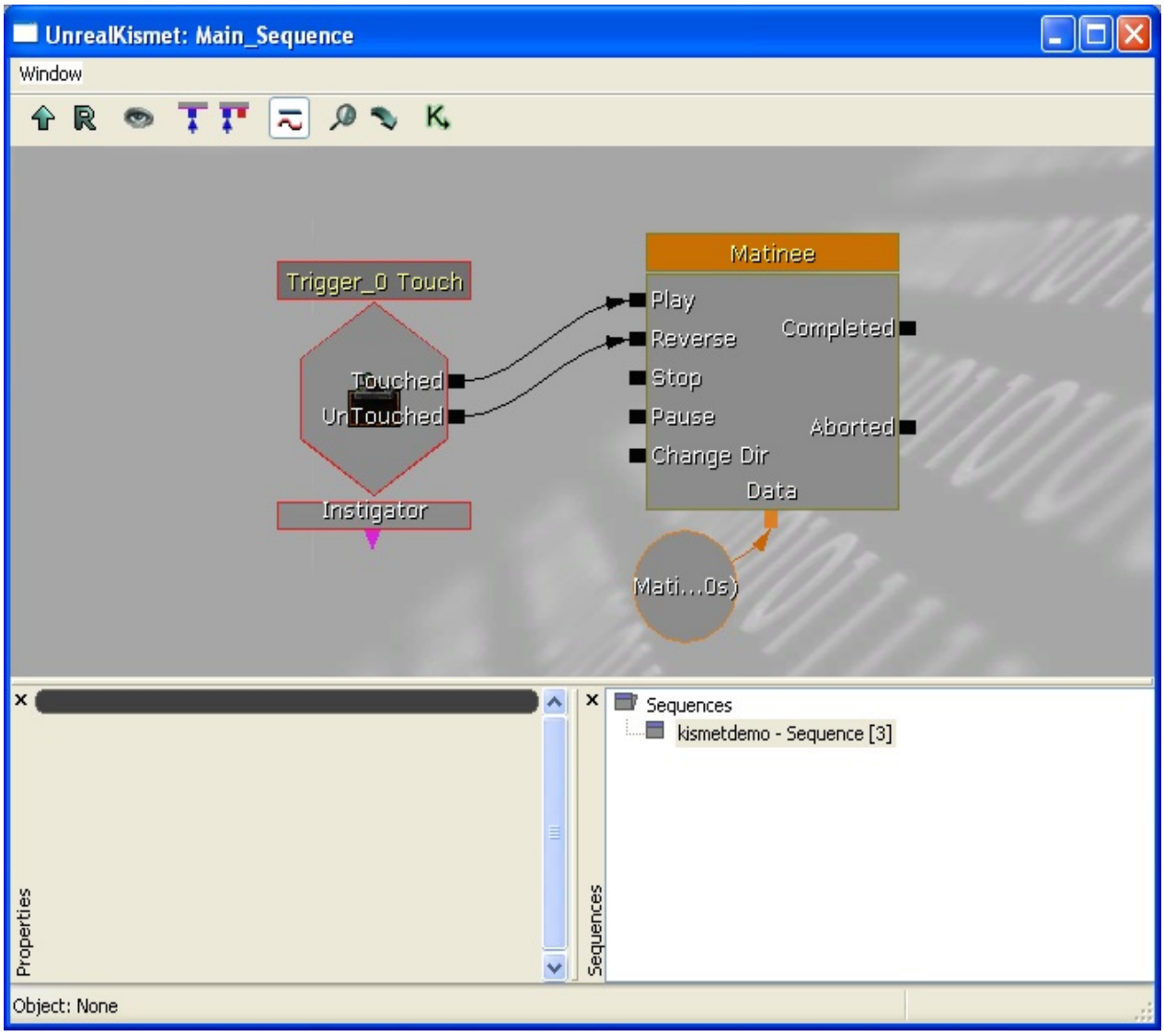
To build a matinee object with your InterpActor do the following

- Select the InterpActor in your map.
- Right click in the Kismet editor window in an area to the right of the kismet object and choose new Matinee.

Connecting the Matinee object to a Kismet Object

- Drag a line between the touch and Play nodes of the kismet/matinee objects
- Drag a second line between the untouch and Reverse nodes of the objects.

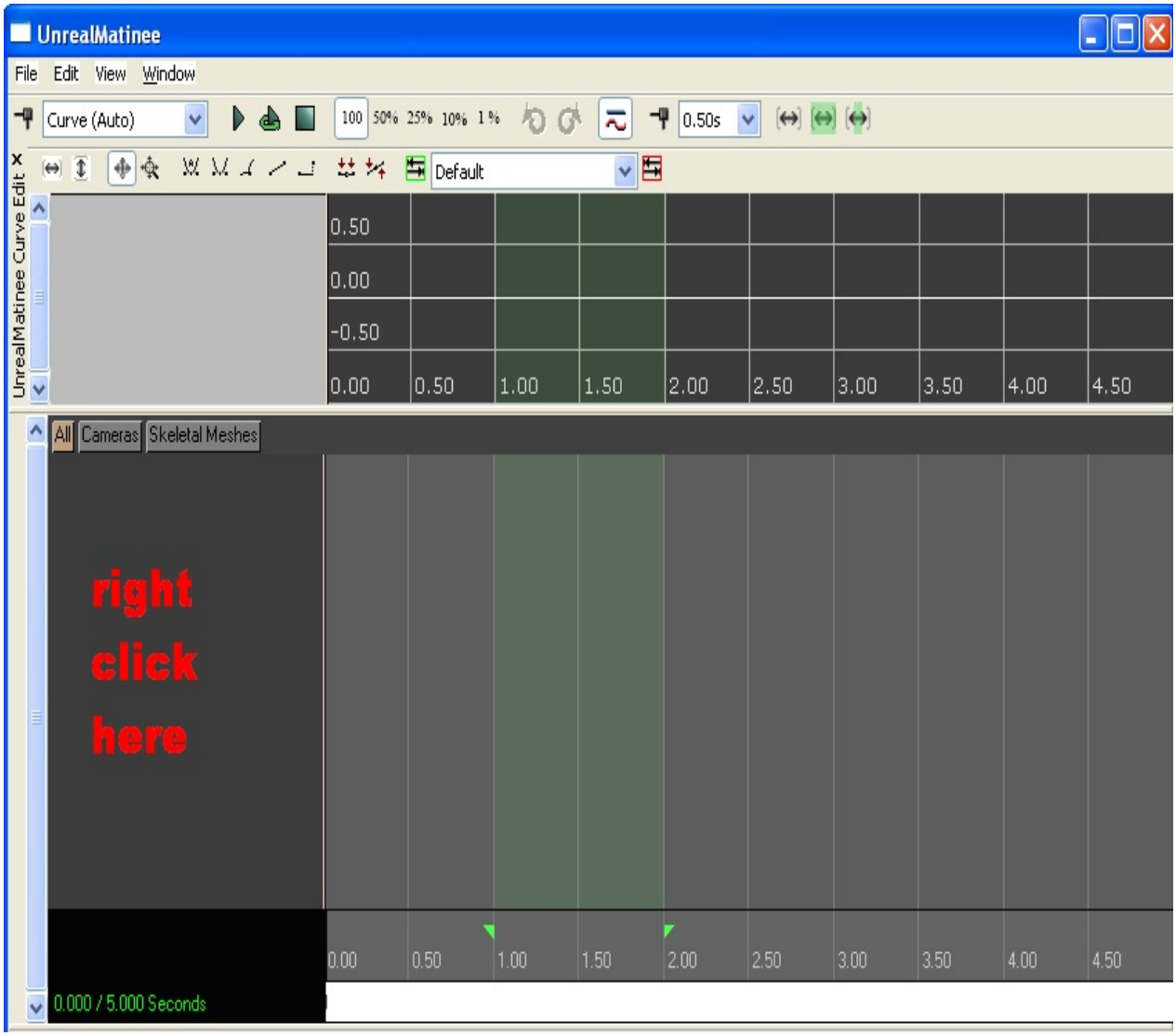
What this means is that when the object is touched, the Matinee animation will play When the object is untouched the animation will play in reverse.



Creating the Matinee Sequence

A Matinee sequence is like an animation. It gets played on trigger of some event as defined by the kismet object it is attached to. To create the animation:

- **Ensure that your InterpActor is Selected at this point!**
- Double click on the Matinee Object.
- Right click in the area under the All/Camera/Skeletal mesh button on the left and choose **new empty group** from the context menu

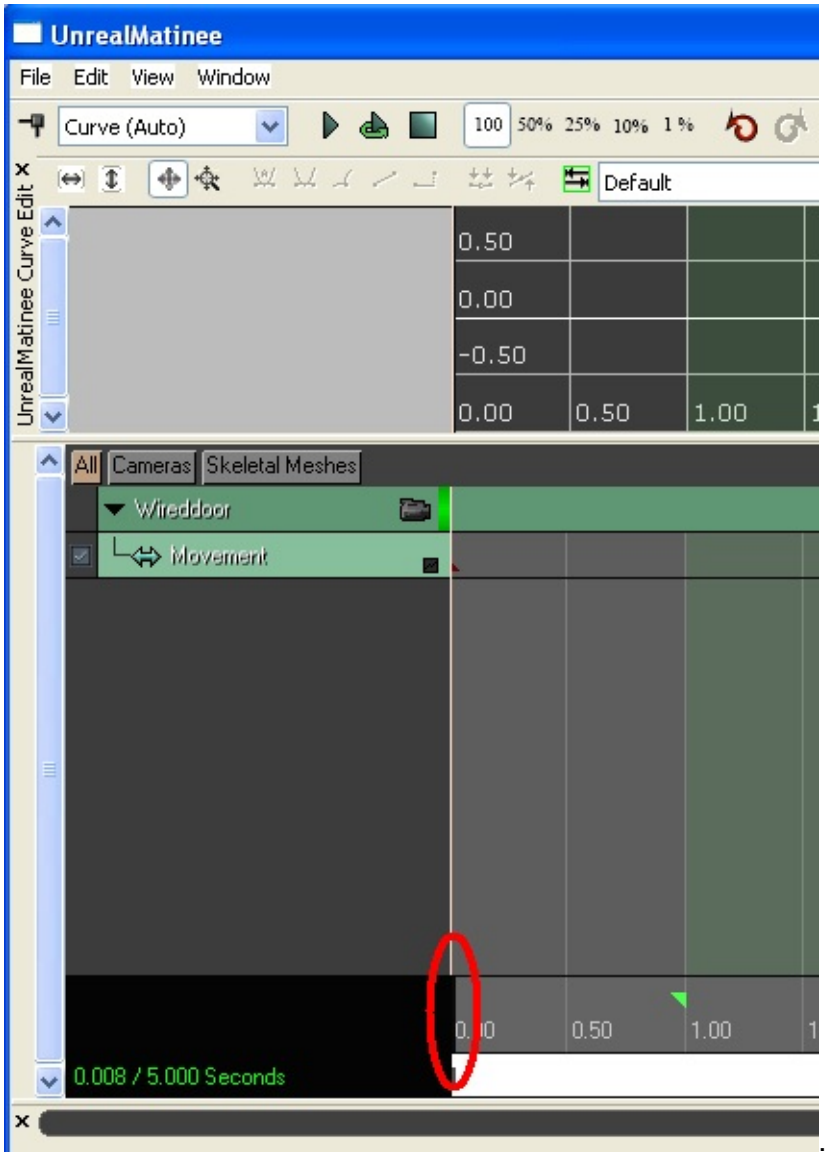


- Type in text for the group name in the popup box. For example movingdoors.
- Right click on the box that has your group name you netered in the previous step and choose **new movement track**

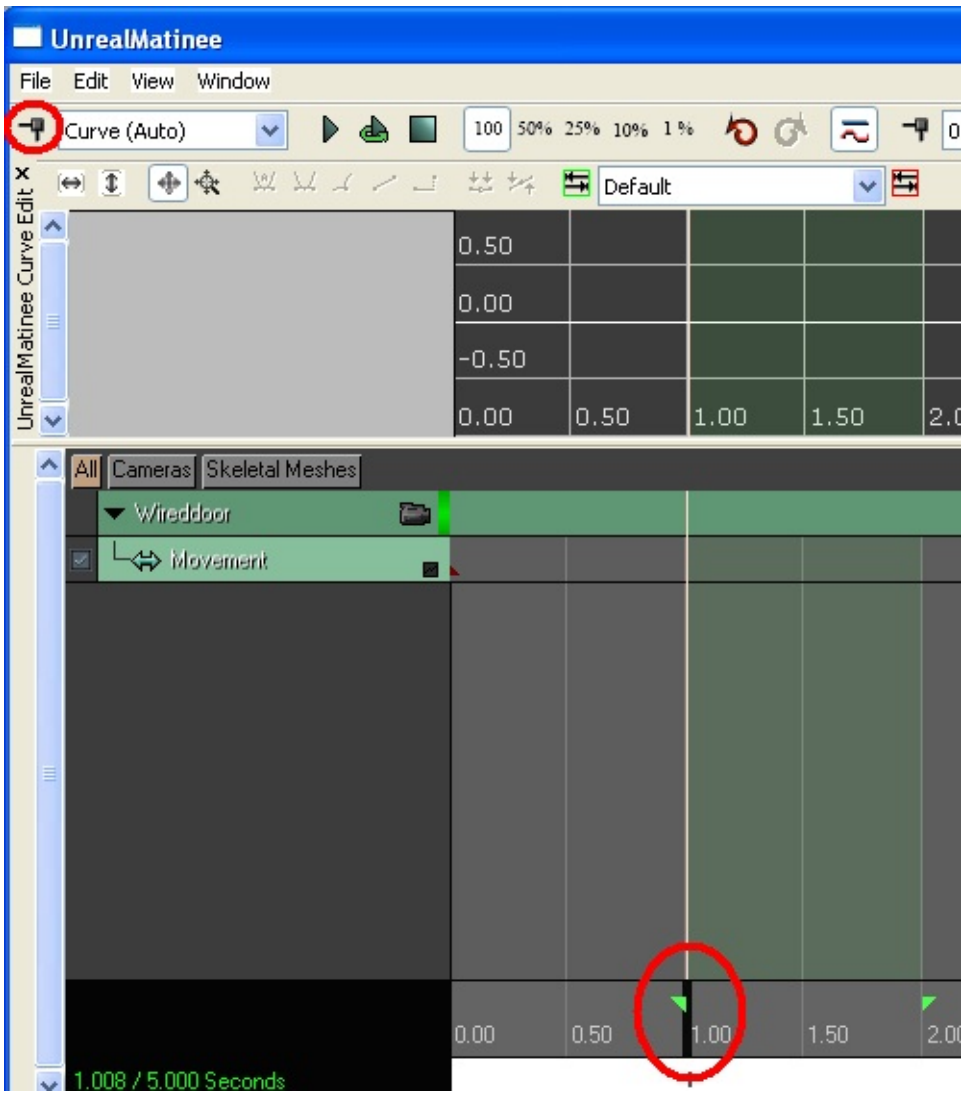
Creating the actual animation

Animation is done by setting a series of key frames. The actor will move based on its position change from its position in one keyframe to the next. You can set up multiple keyframes to make it look more interessting (a door that sputters and stalls before opening for example) but in this example we will simply add one key frame with the door in the open position.

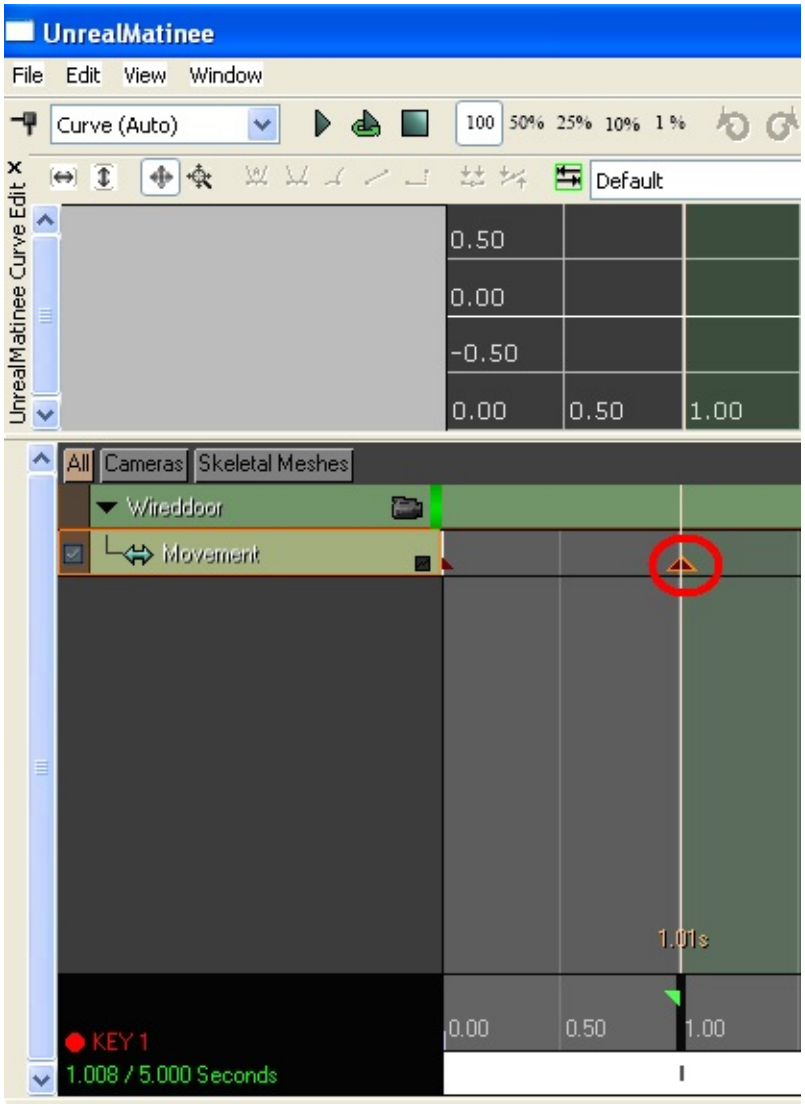
Grab the timer bar and drag it to the 1 second mark or however long you think the door should take to open The timmer bar is circle in the image below.



At the one second mark, indicate that it is a key position by hitting the key button in the button bar of the matinee editor window. Ensure that you have the movement track you want selected (click where it says Movement) before doing this.



If you did it properly the word Key1 will now be on the left of the matinee window and there will be an orange triangle at your 1 second line. Click to select that orange triangle. In your map, you should see the words "Adjust Key 1" in each of your view ports. Pick the viewport that will be best suited to make your adjustment and move your door to its open position.



Close the kismet/matinee windows

Save, build and that should pretty much work.

Resources

-
-

Landscapes provide the means to create outdoor worlds in UDK. This chapter looks at how to create landscapes via a lab.

Lab 3

Summary

This lab will introduce you to the creation of outdoor worlds

Learning Outcomes

- terrains and how they are built
- terrain layers

Submission Requirements

Show Cathy or Hasan

Instructions

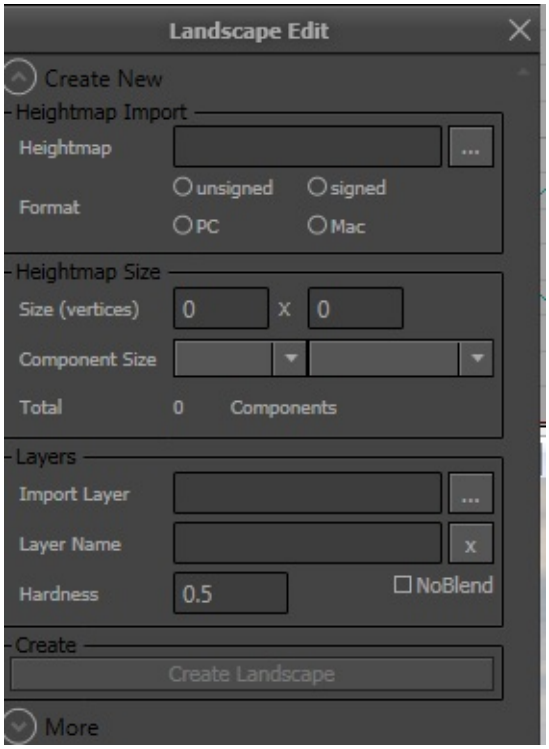
In this lab you will be creating an outdoor environment using UDK's landscape tool. The environment will feature land at different heights, the use of different landscape materials and other decorations.

UDK has 2 methods of creating terrain. Their uses are a bit different. The old method is called terrain and you can find out more about it here

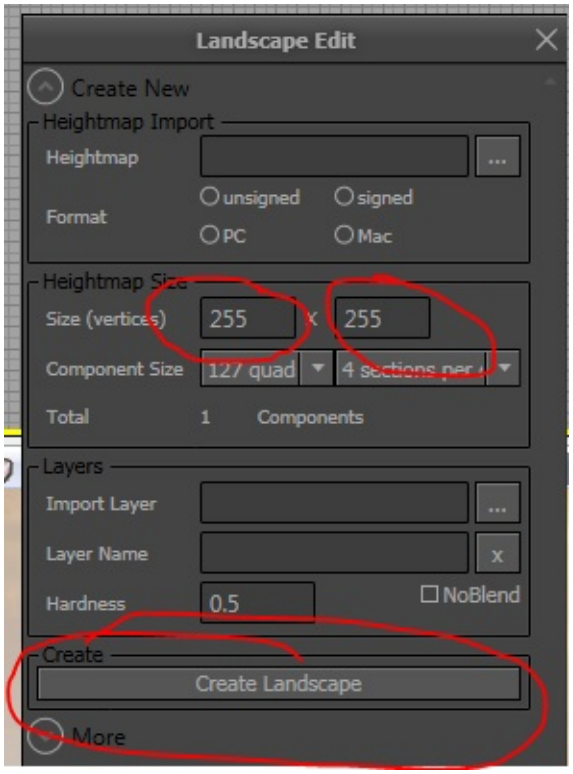
We will however be using the new method of terrain creation called Landscape.

To access the landscape editor, use this icon:  . Notice it is the mountain with the L in it. The mountain without the L is the old terrain editor.

This will bring up the landscape menu.



To create a landscape, we need to specify the size and subdivisions. For now use: 255 and 255 for vertice. Once you hit enter on the second value for number of vertex, it will fill in the other pieces and enable the "create landscape" button. Hit the create landscape button:



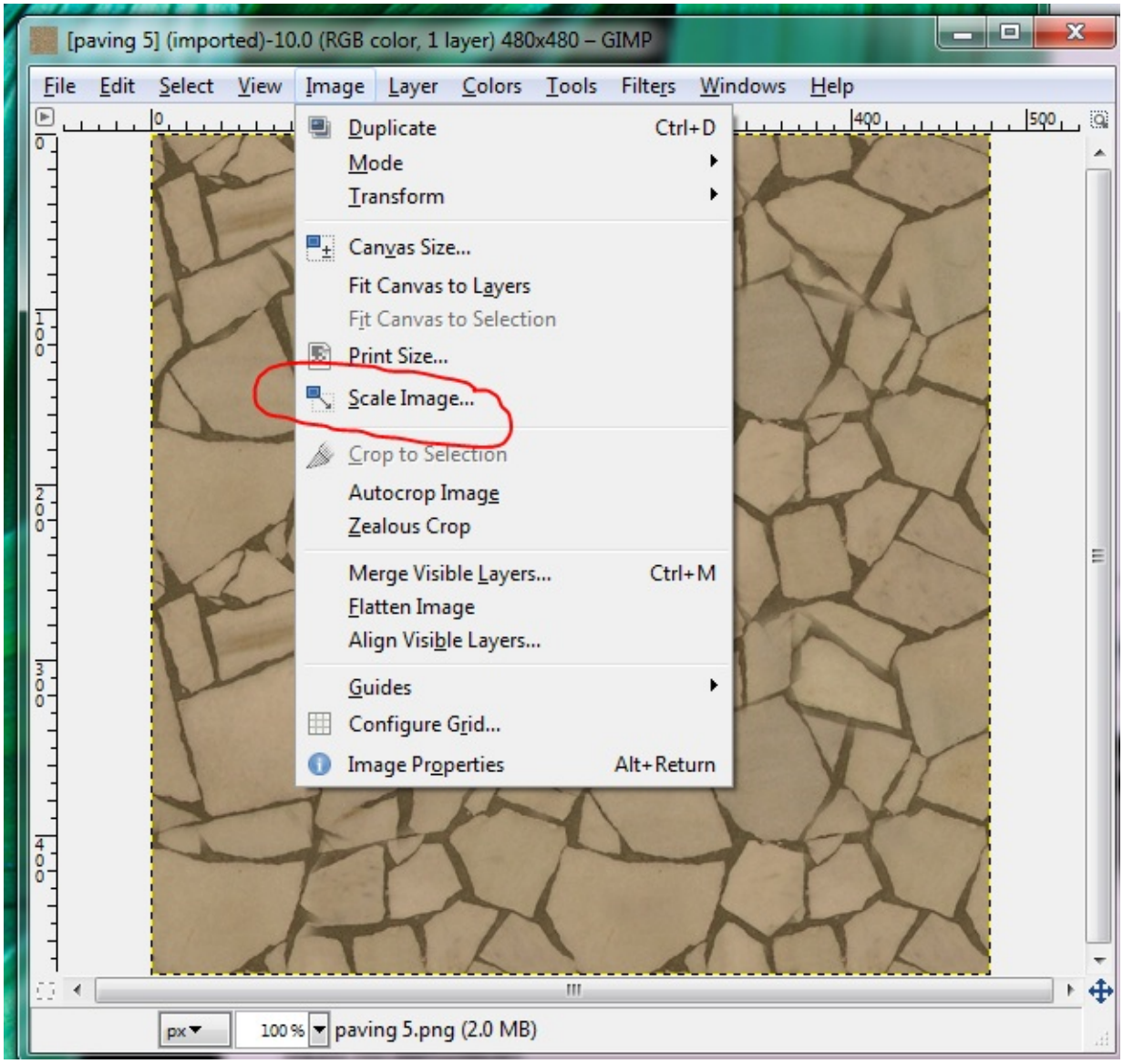
At this point you should see that your world now has a big flat plane on it.

Now, what we want to do next is to create a some ground textures. There are some in UDK, but if you wish you can get others from the internet. If you are importing textures though, you need to remember that textures sizes has to be powers of 2. So a file that is 300 X 300 is not

going to work. You will need to resize the image in an application like photoshop to a power of 2 size (such as 256 X 256).

For example, here is a package that has quite a big collection of textures that are tileable and seamless (does not look disjoint when you put it side by side like tiles). However the the images are not power of two so you will need to adjust them.

Adjusting an Image



Use an application like gimp or photoshop. The schools computer has photoshop but if you do not have it and don't want to buy a copy then gimp is a good open source alternative. To adjust the image using gimp, open the file, go to image menu--> scale image.. and set the size. In general scaling down (making it smaller) gives better results. Once you are done, choose export from file menu and export it as a .bmp file (note that unreal does support other formats... just choose one it supports.

Creating a Terrain Material

In any case choose any 3 textures you wish to use, scale them properly and bring them into unreal.:

- click import button
    -
- choose the file you are importing
- fill in info as necessary

Once they are in create a new material with the samples that you had imported as texture samples. This is done by doing the following:

- right click in the grey part of your content browser in the package where you want the material to go and choose new material
    -
- fill in the package, group and name for your material as appropriate
- go to content browser and select a texture
- double click on the new material you had created to open the material editor
- in the material editor, in the grey part to the right side, right click and choose texture-->new texture sample:

- Repeat for each texture.

Now, once you have done that we will create texture coordinates for the material. These allow you to specify scaling and tiling offsets.

- right click in the grey to the right of the texture sample choose Terrain-->new TerrainLayerCoords

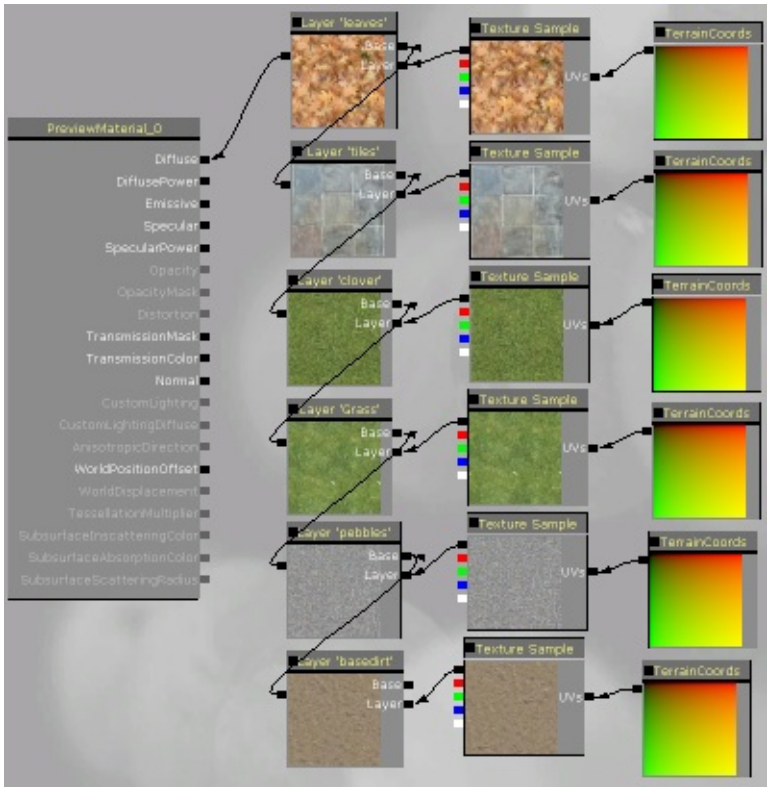
- drag the uv from a texture to a TerrainLayerCoord object. The objects represent how the texture will be scaled and tiled.

- repeat if you want different mappings for each of your textures. Otherwise hook them all up to the same coord object
- To change any of the tiling/scaling of the texture, double click on the associated TerrainCoord object and modify the parameters:

Next, we will set up terrain layers:

- right click in material editor in grey area and choose Terrain-->new TerrainLayerWeight
- repeat for each texture. alternatively you can copy and paste the first one
- for each TerrainLayerWeight, double click to change it
- The layer name should be something that you can associate with the texture
- the preview weight should be 1 for normal weight, higher to make it a "heavier" texture in the blend

Hook up the pieces as described here:



Once you are done save your package.

Now we will edit the landscape and apply the texture to it.

- right click anywhere on the landscape to bring up the context menu and choose landscape properties
- in the content browser make sure you have the landscape material you wish to apply selected
- click on the left arrow beside the Landscape Material property (this may take a minute and the material will look black in your viewport)

    -

- Now, at this point you should see that your land has turned to a black or grey. The reason is because you haven't actually applied any of the layers of the material to your landscape yet. To do so you must set up layers.

To set up layers in landscape editor

- open the landscape editor (click mountain with L in it)
- scroll down until you see the section labelled Target Layerand click on the edit radio button

- This will add put in a section for adding a new layer. In the Layer Name, type in the name that you used for a layer in your material and hit the + sign
    -
- You will know you did it correctly if a sample image for that layer shows up in the sample box. Another Add New Layer section is automatically added for the next Layer
    -
- pick the texture that will be the most common throughout your scene click change your brush to the component brush, make sure that you have paint as your paint tool, then hold the ctrl key and left click in on the landscape and it will apply that texture throughout. Note that it may take a moment. Also note that your component brush will slightly alter the highlighting on your landscape so the colour may look wierd after you apply the material, change your brush and it will be as expected
- To add other layers, select the layer you wish to add, and ctrl click into the scene with your paint brush. The materials will blend together automatically. The more you press, the more the layer gets applied.
- To alter the height, choose heightmap in the layer target layer
- to lower with heightmap use ctrl-shift click. to raise use ctrl click

This chapter will introduce you to UnrealScript and the fundamental classes of the UDK.

Lecture Topics

Day 1

- Do lab 4 part 1 on the board and give some background information on the UDK while going through
- Introduce UnrealScript

Day 2

- Do Lab 4 part 2 git resume on the screen, talk a bit about git, branches, remotes, and git checkout
- Talk about default properties
- Talk about Actor
- Talk about Actor lifecycle
  - PostBeginPlay
  - Tick
  - Destroyed
- Talk about debugging functions
  - `Log
  - ScriptTrace
  - GetScriptTrace
- Talk about GameType class
  - Talk about some of the useful properties of the GameType class, such as:
    - PlayerControllerClass
    - HUDType
    - DefaultPawnClass
  - Introduce idea of Player Controller and Pawn
- Talk about UTGame class and MapPrefixes issue
- Talk about UTPlayerController class

Tasks

- Prepare your scripting workflow

Resources

- Actor Components
- UnrealScript Game Flow

UnrealScript

UnrealScript is the programming language that is used to create gameplay code in the UDK. It is similar to Java conceptually and syntactically. This section will serve as a very basic outline as to what you can expect from UnrealScript; it does not cover everything that UnrealScript has to offer. I encourage you to read the UnrealScript reference on your own time, especially the overview section; this will get you up to speed on exactly what you will get from UnrealScript. With that in mind, UnrealScript offers the following familiar features:

- Basic types
  - byte
  - int
  - bool
  - float
  - string
  - enumeration
  - name
    - A name is effectively a special string, used to hold the name of a symbol, eg a class name (ie var name someName;)
  - object reference (more on this below)
  - class reference restrictors (more on this below)
- Structs
  - Similar to C/C++ structs
- Classes
  - Single inheritance (multiple inheritance not allowed)
  - **Object** class at the root of the hierarchy
  - **No constructors!**
  - Garbage collected
- Member variables
  - Variables are fixed type (ie closer to Java as opposed to JavaScript)
  - Declared through var keyword (ie var int myVariable;)
    - May need to be declared at top of file
  - Initial values of variables declared in Defaultproperties block
- Functions
  - Function local variables
    - Declared through local keyword (ie local int myVariable;)
    - Must be declared at top of function
  - Function return types
  - Function parameters

    - Member functions (default)
    - Static functions
- Variable length arrays declared through the Array<*type*> keyword (ie var Array myArrayVariable;)
- Fixed length arrays declared through the [] syntax (ie var int[6] myFixedArrayVariable;)

In addition, UnrealScript also has quite a few non-standard features such as:

- Language is **Case insensitive!**
- Actor states
- Rich, typed, literal format
- Class default properties and no constructor

Please look at the reference above for more information on those.

Classes

Classes in UnrealScript are part of *packages*. A package is a folder placed inside the UDK/UDK-xxxx-xx/Development/Src directory; the name of the folder is the name of the package. A folder named *Classes* must be placed directly inside the package folder. UnrealScript class files must be placed inside a Classes folder.

To cause the compiler to compile your package, your package name must be added to the **EditPackages** list in DefaultEngine.ini, under the **UnrealEd.EditorEngine** section. For example, to cause the compilation of an example package named **Foobar** we would update the following *UnrealEd.EditorEngine* section:

```
[UnrealEd.EditorEngine]
+EditPackages=UTGame
+EditPackages=UTGameContent
```

To the following:

```
[UnrealEd.EditorEngine]
+EditPackages=UTGame
+EditPackages=UTGameContent
+EditPackages=Foobar
```

Note the **+**. See Lab 5 part 1 for more information.

Technical Class Information

An UnrealScript class is stored in a .uc file. The name of the class must match the name of the uc file. For example: `class Foo must reside in Foo.uc`

A class must inherit from a single other class. The highest ancestor in the hierarchy is the class **Object**. It contains many useful functions. See Object.uc for more information. Another useful class is **Actor**. Actor directly derives from Object and it contains the functions and variable members required to represent a real object in the map (or otherwise) with such information as location, rotation, physical representation, graphical representation, etc...

For example, consider the following class declaration:

```
class Foo extends Object;

var int SomeInt;

function float GetFloatFromInts(int a, int b)
{
    local float ret;
    ret = a + b;
    return ret;
}

Defaultproperties
{
    SomeInt=5
}
```

Note some things:

- Class is called Foo, declared with the **class** keyword, must reside in Foo.uc
- Inherits from Object and is therefore the descendant of it
- Has one member public integer variable, *SomeInt*
- Has one function that returns float and accepts two ints
  - Contains a local float variable
- Has Defaultproperties section containing default value for SomeInt
  - **If SomInt was not mentioned in defaultproperties, default value would have been 0**

Member Variables

To define member variables in a class, we use the keyword *var* and follow it with the type of the variable and then the name of the variable. The formal spec looks like: `var [([property_group_name])] [specifier_1 [, specifier_2 [, ...] ] ] variable_name;`

For example, we can declare some simple variables as follows:

```
var Actor A;
var(Important) const private int B;
```

Notice:

- A is a simple variable. It is of type actor. No privacy specifier is provided so it defaults to **public**; this is different from most languages
- B has a number of features:
  - An editor property window group name is specified, so this variable would appear underneath that category in the editor property window
    - Could have also been left empty, ie (), and it would have been placed in a category that adopts this class' name
  - It is const so its value cannot be changed after the object's construction
    - The value should be set in the default properties
  - It is private, so it cannot be read from or written to from any other classes, including children ie direct children

Class Reference Restrictors

An important type of variable is the **class reference restrictor**. This type of variable allows you to store a reference to a *class*. This is useful in situations when you need your code to use different classes under different circumstances or receive a class and work with it. The syntax for a class restrictor is as follows: `var class<Controller> myClassReference;`

The above code segment declares a member variable class reference restricted to classes that are *descendants* of the **Controller** class. For example, we can store the classes *Controller*, *PlayerController*, *AIController*, *UTBot*, etc... in this member. We can assign class literals to such a variable. For example: `myClassReference = class'UTBot';`

The syntax to the right of the **=** operator is known as a class literal. It allows you to obtain a reference to a particular class in the system. In this case, we are obtaining a reference to the class *UTBot*. We can assign *UTBot* to this class restrictor because *UTBot* is a descendant of *Controller*. The inheritance chain of *UTBot* looks as follows: `UTBot > UDKBot > AIController > Controller > Actor > Object`

Notice that *Controller* lies within the inheritance chain of *UTBot* which is why we can assign it to **myClassReference**. By contrast, the following snippet would cause a syntax error: `myClassReference = class'Actor'; // This causes syntax error`

The syntax error is because the class *Actor* does **not** inherit from *Controller*.

Uses of Class Reference Restrictors

We can use class reference restrictors for two things primarily:

1. Passing the class reference to other functions
    - This is useful when you write functions that accept class references or when you need to pass a class to an engine function, for example **spawn**, which instantiates an Actor given a class of Actor to instantiate (more on this later)
2. Retrieving static variables or default variables and for calling static functions

The first use case is very simple so it needs no explanation. The second use case should be elaborated. Given a class reference, we can do any of the following:

- Retrieve the default values of all member variables
- Call a static function

Let's look at each use case in turn. Consider the following class:

```
class RestrictorTest extends Actor;

var int IntMember;
var float FloatMember;

static function RestrictorTest InstantiateRestrictorTest()
{
    return Spawn(class'RestrictorTest');
}

static function int GetDefaultIntMember()
{
    return class'RestrictorTest'.default.IntMember;
}

static function float GetDefaultFloatMember()
{
    return class'RestrictorTest'.default.FloatMember;
}

DefaultProperties
{
    IntMember=10
    FloatMember=3.5
}
```

The above class has 2 member variables and 3 static functions. Note the three static functions:

- InstantiateRestrictorTest
- GetDefaultIntMember
- GetDefaultFloatMember

Each one of these functions can be called without an instance variable. For example, we could have the following section of code:

```
var RestrictorTest myTest;

myTest = class'RestrictorTest'.static.InstantiateRestrictorTest();
```

This will call the static function **InstantiateRestrictorTest** which calls the **spawn** function returning an instance of the *RestrictorTest* class. Note the following:

- Use of the class literal **class'RestrictorTest'** being passed to the spawn function inside the code of *InstantiateRestrictorTest* to indicate that this is the class that should be instantiated
- Use of the suffix **static** in the above code segment to gain access to static functions in the class *RestrictorTest*

Similarly, calling **GetDefaultIntMember** and **GetDefaultFloatMember** will return **10** and **3.5** respectively due to the syntax used in those two functions. Note the use of the suffix **default** to gain access to the default value of all members in the class.

As Local References

Class reference restrictors can also be declared as local function variables. Simply change the keyword **var** for the keyword **local**, all other properties are exactly the same.

Functions

Like in other programming languages, functions can be declared and defined in UnrealScript. The basic syntax of function declaration is:

```
[function_specifier_1 [, function_specifier_2 [, ...] ] ]
function [<return type>]
function_name(
    [ [param_1_specifier_1 [, param_1_specifier_2 [, ...] ] ] <param 1 type> param_1_name
    [, [param_2_specifier_1 [, param_2_specifier_2 [, ...] ] ] , <param 2 type> param_2_n
    [, ...] ] ]
);
```

Function declarations can also be followed with a body as opposed to a semicolon. Here are some examples of common function specifiers:

- exec

    - Function can be called from console or from key bindings
- static
    - Function executed without a current object (Java/C++-style static function)

Return types can be any object type in UnrealScript.

Parameter specifiers include:

- optional
    - Parameter can be omitted at call site. Parameter can be given default value with **=** operator
- out
    - C++-style reference variable. When value is updated in function, value update is reflected at call site

For the full list of variable specifiers, see UDN UnrealScript Variables:Variable Specifiers

Function Body Variables

Like in other programming languages, a function can have a body wrapped in curly braces. While most statements are straight forward, variable declaration is slightly strange. Function variables are declared with the **local** keyword. This keyword works like a simplified version of the **var** keyword above.

The spec looks something like this: `local <variable type> variable_1_name [, variable_2_name [, ...] ] ];`

Some things to note:

- Local variables, like instance member variables, cannot be initialized
- Local variables must be **declared at the top of the function, before ANY other statements!**

Otherwise, local variables are similar to instance member variables.

Engine Events, Super, Debugging

The UDK is a game engine and utilizes an event-based programming style. If you've never seen that term before, it means that an overarching framework exists where some kind of simulation is occurring and as part of the simulation, events are fired that waiting components can then respond to. In this style, independent components do not really take proactive action, they wait for the framework to tell them:

```
A particular event has occurred, looks like you signed up for updates for that event, please respond to the event's firing
```

In this case, the overarching framework is the UDK and the simulation is the game running in one of its various states. All UnrealScript classes participate in the event response by **overriding** special functions defined in *Object* and more commonly in *Actor*. We call these special functions that are built to be overridden **Engine Events**.

Basic Events

The following events are all defined on **Actor**. There are many events that are defined on Actor. For a more complete list, see Actor.uc or BeyondUnreal Actor Events). Listed here are examples of critical events:

- function PostBeginPlay()
  - Called after the Actor has been constructed and initialized
  - This function can serve as a sort of initializer or default constructor for an Actor
  - Be warned that it is **NOT GUARANTEED** that this function is executed **BEFORE** some other actor may have gained a reference to this Actor
  - From BeyondUnreal:
    - Called by the engine pretty soon after the Actor has been spawned or the game has begun
    - Most actors use PostBeginPlay() to initialize their UnrealScript values. The actor's PhysicsVolume and Zone are valid but the actor is not yet in any State
- function Tick(float deltatime)
  - Called every frame
  - Deltatime is the time, in seconds, since the last tick event for the engine
- function Destroyed()
  - Called before an Actor has been deallocated, but after *Destroy()* has been called on this Actor
  - Pretty soon after this function is called, all other actors in the simulation will lose their reference to this Actor

- At some point in the future, this Actor will be garbage collected

Super

When overriding a function, it is frequently useful that we want to *add* functionality to our parent's version of the function, as opposed to completely replacing it. This would require the ability to call our parent's version of a function. This is done by using the **super** keyword.

The super keyword syntactically provides an object that represents all of the parent class' members. Calling any functions through the super keyword will execute the parent's version of that function. For example:

```
class FooPawn extends Pawn;

function PostBeginPlay()
{
    // Calls own version of tick
    tick(100);

    // Calls parent's version of tick
    super.tick(200);

    // Calls parent's version of PostBeginPlay
    super.PostBeginPlay();
}

function tick(float deltatime)
{
    `Log(deltatime$" since the last tick");
    super.tick(deltatime);
}

Defaultproperties
{
}
```

Notice:

- Class extends Pawn (an important class in the UDK, prominent part of a UDK character)
- Function **tick** produces a log message at every call and then calls the parent's version of tick
- Function **PostBeginPlay** calls the following, in sequence:
  - Own version of tick, passing it 100 (ie 100 seconds)
  - Parent version of tick, passing it 200 (ie 200 seconds)
  - Parent version of PostBeginPlay

This is not great code! Your code should not need to explicitly call tick from PostBeginPlay, ever... but it gets the point across about events.

`Log

Logging is an universally accepted form of debugging and it is useful for our purposes. Logging in the UDK is done using the **`Log** macro function. Note the ` in front of the symbol name; this is required! Log is not a function, it is a *macro*. The UDK mandates that calling a macro *requires* ` ahead of the macro's name (if you're wondering how to say it, the ` symbol is called *grave* or in the unix world, *backtick*).

This macro accepts any value type and dumps its value to the script log. Any value can be passed to Log and be printed. Longer messages can be composed by using the operators **@** and **$**. For example:

```
class Sup extends Actor;

var float A;
var int B;

function PostBeginPlay()
{
    `Log("[A: "$A@", B: "@B$"]");
}

Defaultproperties
{
    A=6.5
    B=72
}
```

Spawning an Actor of type Sup would produce the following message once PostBeginPlay was called, ie after construction: `[A: 6.5, B: 72]`

Note that **@** and **$** are **completely interchangeable**. They splice the two items around them, producing a concatenated string.

ScriptTrace

Script tracing shows you how and *when* your functions are being called. There are two functions that we can use to perform script tracing:

- ScriptTrace()
  - Dumps the script trace immediately to the log
- GetScriptTrace()

- Returns the script trace as a string which you can use however way you see fit, including logging it via `Log

Here's an example of how a trace would look like once dumped to log. Consider the following set of class excerpts:

```
class Base extends Actor;

function Tick(float deltatime)
{
    ScriptTrace();
}
```

```
class Derived extends Base;

function DerivedFunction(float dt)
{
    super.Tick(dt);
}

function Tick(float deltatime)
{
    DerivedFunction(deltatime);
}
```

The following would be the script trace that is produced when Derived's tick function is called by the engine:

```
ScriptLog: Script call stack:
        Function SomePackage.Derived:Tick
        Function SomePackage.Derived:DerivedFunction
        Function SomePackage.Base:Tick
```

Notice that the function at the bottom of the stack is the one currently executing. This is because the **ScriptTrace()** call is placed in Base.Tick. Each level in the trace is called a *stack frame* and it represents the function that called the function in the stack frame immediately below it.

As you can see, script tracing can be extremely useful for finding when your functions are being called and if they are behaving correctly.

GameInfo, Map Association, WorldInfo, UTGame

In the UDK, the GameInfo actor is what you would think of as a **game type**. The GameInfo Actor is a class that designates and pulls together the various components required to form a game. This is mostly done in the defaultproperties of the class as the heavy lifting is mostly done by the GameInfo super-class itself meaning that you should not have to update the class' functionality if you just need a basic game type.

Associating a GameInfo Subclass With a Map

To use a GameInfo subclass that you've created, we must associate it with a map and then load that map. The process to do so is as follows:

1. Create your GameInfo subclass and compile it
2. Start the editor and open/create your map
3. Open the **WorldInfo** properties by navigating to: `View -> World Properties`
4. Expand the **Game Type** category in the properties window
5. Minimally set **Default Game Type** to your GameInfo subclass. Additionally, set the other properties:
    - `Game Type For PIE`
        - PIE stands for: **Play In Editor**
        - This is useful if you want to test your map with your compiled game type when you click the various *play in editor* buttons
            - This is a useful property and should likely be set to your compiled game type
    - `Game Types Supported On This Map`
        - Useful for maps that need to support multiple game types while allowing the base game info actor decide which game type to spawn dynamically
        - Really only useful for maps built to work specifically with UT in order to deal with the various UT game types using the same maps. For example:
            - UTDeathMatch
            - UTTeamGame
6. If your GameInfo subclass extends UTGame, you will need to address some complications. See the **UTGame** section in this document.

WorldInfo & Accessing Map's GameInfo

In any game session, there exists a **single** GameInfo or GameInfo-subclass actor that acts as the gameinfo for the map. It can be accessed through: `WorldInfo.Game`

In this case, WorldInfo is variable of type *WorldInfo* that is defined on Actor as follows: `var const transient WorldInfo WorldInfo;`

- Notice that the specifiers *const* and *transient* are defined here
  - const implies that this property cannot be changed as it is set for you by the engine
  - transient means that the value in this variable will not last past the lifetime of the current map ie when the map changes for any reason, this value is set to None
    - Yes, it is possible for some actors to sometimes survive past the change of map :)

Like GameInfo, every game session has a **single WorldInfo** Actor that encapsulates the properties of the map itself. The WorldInfo Actor defines Game as follows: `var GameInfo Game;` When using a custom game type ie a subclass of GameInfo (as is typically the case), you will need to cast this variable to the desired class first before being able to get its properties.

WorldInfo Properties

WorldInfo also has other useful members:

- `var float TimeSeconds`
  - Time since the game began in seconds
  - Affected by game speed, pauses, etc... ie **NOT** realtime
- `var float RealTimeSeconds`
  - Wallclock amount of time that has passed since the game began in seconds
  - Unaffected by pauses, game speed, etc...
- `var float DeltaSeconds`
  - Amount of time since the last tick in seconds
  - This is the value passed to every actor's Tick event
- `var string ComputerName`
  - The name of the local machine, according to the OS

Useful Functions to Call

WorldInfo also has some functions that could be useful to call:

- `native final function SetMusicVolume(float VolumeMultiplier);`
  - Sets a multiplier on the current volume of music
  - Value between 0 and 1 to scale the volume
- `native function float GetGravityZ();`

    - Returns the Z component of the current world gravity
- `native final static function WorldInfo GetWorldInfo()`
    - Allows non-actor derived classes to get access to the current WorldInfo Actor

GameInfo Properties

Subclassing *GameInfo* or a descendant of it let's you define a new game type. Let's look at some GameInfo member variables that are useful to set in the defaultproperties:

- `var class<HUD> HUDType`
    - This is the HUD that player controller's will use when playing your game
- `var class<PlayerController> PlayerControllerClass`
    - The class of PlayerController to spawn for your game
- `var class<Pawn> DefaultPawnClass`
    - The class of Pawn to use for controllers in this game
        - Selection of pawn class to spawn for controllers can be overridden in *GetDefaultPlayerClass* (see below)
- `var float GameSpeed`
    - The speed of the game as a fraction
    - Defaults to 1

Useful Functions to Override

For basic games, setting the above in the defaultproperties of your game type is adequate. For more complex requirements, it can also be useful to override some of the functions defined in GameInfo. Here are some of the more important ones:

- `function float RatePlayerStart(PlayerStart P, byte Team, Controller Player)`
    - This function is called when spawning a player and is passed a PlayerStart, a team index (may not be relevant), and the player's or bot's controller Actor
    - This function will be called for every player start in the map whenever a player or bot is being spawned
    - You can use it to restrict at which player starts various players may start by returning 0 when a player start is inappropriate and 1.0 when the player start is desireable for that player
        - For additional granularity, you may return an arbitrary float from this function. The function responsible for repeatedly calling RatePlayerStart (GameInfo.ChoosePlayerStart) will then choose the player start with the highest score to spawn the player at
- `function Killed( Controller Killer, Controller KilledPlayer, Pawn KilledPawn, class<DamageType> damageType )`
    - Called when a player or bot is killed

    - Override to redefine what the game should do when a player or bot is killed
        - Note that this is **not** the only place where you could detect a player's death but any code that must be performed for *every* death, regardless of the type of player/player+pawn combination that died, should be placed here
- `function RestartPlayer(Controller NewPlayer)`
    - Performs the logic required to restart a player or start the player for the first time
    - Can be overridden to selectively restart players or bots based on certain conditions
        - This can be done by either choosing to call the *super* version of this function to let the normal restart player process proceed or not calling the super version, thereby aborting the process
- `static event class<GameInfo> SetGameType(string MapName, string Options, string Portal)`
    - Should return the class of game that should be spawned for a given map name
    - By default returns *Default.Class*
        - This returns the class of the current GameInfo actor as a class object
- `function bool ShouldRespawn( PickupFactory Other )`
    - Determines whether a PickupFactory is allowed to respawn
- `function class<Pawn> GetDefaultPlayerClass(Controller C)`
    - Should return the class of Pawn to spawn for a Controller
    - Defaults to *DefaultPawnClass*
    - This function can be overridden to return different Pawn classes based on changing conditions
- `function StartMatch()`
    - Begins the match, spawns pawns, etc...
    - Override to know when game play actually begins

Useful Functions to Call

You may also call some functions to perform certain tasks:

- `function ResetLevel()`
    - Restarts the game without reloading the level
- `function SetGameSpeed( Float T )`
    - Sets the speed of the game as a fraction from 0.000001 to infinity
    - Game speed can also be set as a default property
- `function int GetNumPlayers()`
    - Returns the number of players and bots currently in the game

UTGame

UTGame is a prominent subclass of GameInfo that defines the properties required to start a basic game of UT. We care about this class because it makes basic testing a bit easier for us as it gives us bots that react, shoot, gives us a player controller that has guns, handles scoring in a UT game style, etc...

However, it also comes with some peculiarities. Let's look at UTGame's properties:

- `var class<UTBot> BotClass`
  - The class of controller to use for bots
  - *UTBot* is a subclass of *AIController* which is the AI counterpart to *PlayerController*
- `var bool bUseClassicHUD`
  - Must be set to true if you want to set a custom HUD type because otherwise, UTGame will override the value you place in *HUDType*
- `var array<string> MapPrefixes;`
  - Array of map prefixes
  - UTGame enforces a map naming convention of *<Prefix>-<MapName>.udk*, eg *DM-Deck.udk*
    - UTGame uses the Prefix part to determine the type of GameInfo actor to spawn given a particular map
    - This means that you have two options if you want to sub-class UTGame and have your game type be loaded (as opposed to UTGame):
      1. Make sure your game type's maps are named *<Prefix>-<MapName>.udk* then set the first element of MapPrefixes equal to whatever your map name's prefix is, ie:
         - `MapPrefixes[0]="L1"`
      2. Override the function *SetGameType* to return your game's class (ie copy the contents of this function from GameInfo) as this is where UTGame performs the prefix-based map selection

Lab 4 Part 1

Summary

Welcome to your first UnrealScript lab! This lab will lay the foundation for your education in UnrealScript.

In this lab, you will:

- Deploy the startup package
- Initialize your labs repository with the required basics
- Create, compile, and test an UnrealScript actor and package

Learning Outcomes

- The development workflow relating to UnrealScript in GAM537
- How your git labs repository is to be used in this course
- The foundation of UnrealScript development

Files to Implement

- Hello.uc

Submission Requirements

To an instructor, show the following:

- Your compiled Foobar.u
- Your updated labs repository

Instructions

Deploying the quickstart project

We will use Visual Studio 2013 and nFringe version 1.3.0.5 to develop UnrealScript for the UDK. If you are doing this lab on a lab computer, VS2013, nFringe, and the UDK version February 2015 will already be installed. If you are doing this at home, ensure that the above 3 are installed noting that **nFringe must be installed AFTER VS2013**.

UDK source code

The directory where all UDK UnrealScript code resides is `UDK-2015-01/Development/Src` We call this the **development directory**. Unlike other development environments, when developing for the UDK there is only **one** VS project that you will need and it will be placed in the development directory. We call this project the quickstart project and we will be providing it here.

Quickstart project

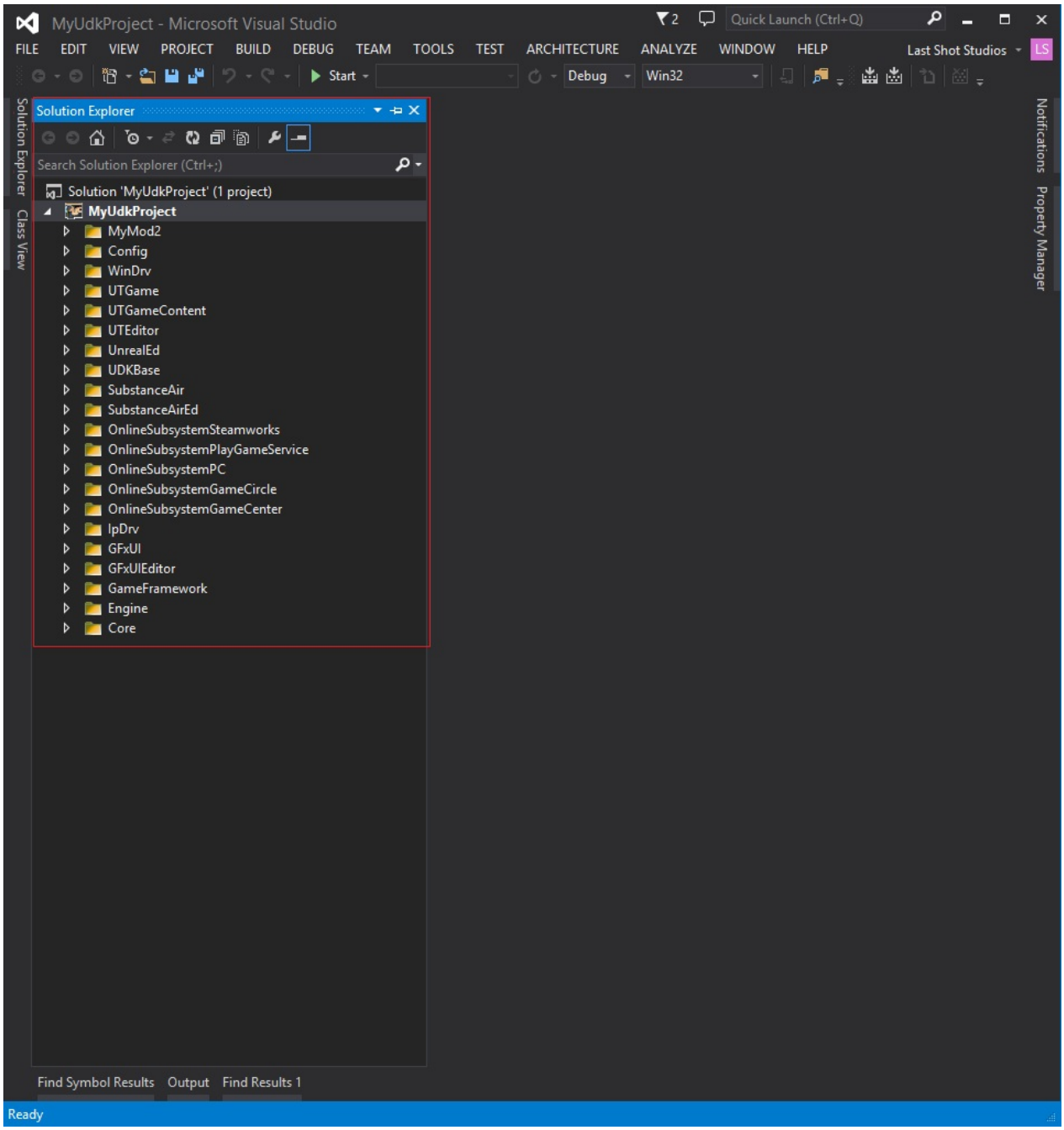
Download the quickstart project archive located here. Extract this folder to the development directory. In the development directory, in addition to many folders, you should have the following files:

- MyUdkProject.sln
- MyUdkProject.ucproj
- ScriptModifiers.xml

UnrealScript development overview

This is a good time to give you an overview of UnrealScript development. Open the provided **solution file MyUdkProject.sln** by double clicking the solution file. This should open visual studio 2013.

Open the solution explorer tab. You should see something like the following:



Most of the filters (in VS-speak, the "folders" underneath the project are called *filters*) are UnrealScript packages. In the UDK, code is organized into packages, each package containing a **Classes** directory and this directory containing a collection of UnrealScript files (.uc).

UnrealScript Breakdown

In Unreal, a single .uc file represents an UnrealScript class. This is closer to Java than it is to C++ in the sense that UnrealScript files are allowed to contain exactly **one** class. We will look at the details of .uc files in the second part of this lab on Thursday.

As stated above, a single UnrealScript file (.uc) is never compiled by itself. UnrealScript files are always compiled in packages; this is to make distribution of code to third parties more sane. UnrealScript packages can contain zero or more UnrealScript files.

Create a Foobar package

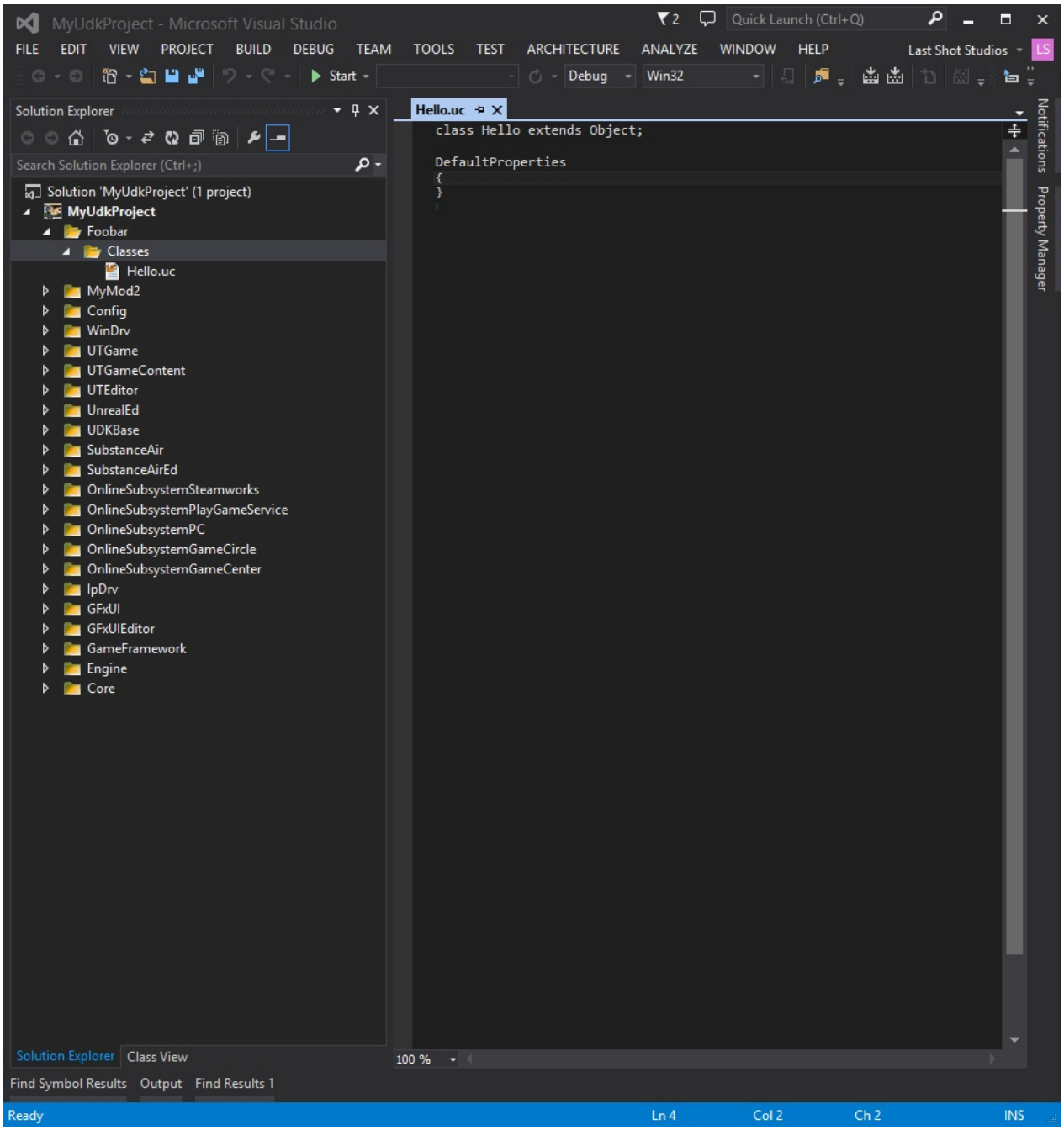
Let's learn how to create packages by creating a new package that we will compile. Do the following:

1. Right click the project in the solution explorer (the **project**, not the solution!), highlight **Add**, click **New folder**.
2. Name your new folder **Foobar**
    - This will have created a new folder for you in the development directory
3. As stated above, UnrealScript files must be located in a Classes directory that sits underneath the package directory. Right click your new Foobar package, highlight **Add**, click **New Folder**.
4. Name your new folder **Classes**

Populate Foobar package with Hello.uc

1. Right click your Classes directory, highlight **Add**, click **New Item...**
2. From the template list, pick **UnrealScript File**
3. Name it **Hello**

You should have something that looks like the following:



This is as far as we go with regard to coding today. If you have a solution that resembles the above, you're doing well. Now we will compile our Foobar package.

Updating configuration

Unlike other environments that you may be accustomed to, the UDK is configuration-file-heavy; meaning that it relies a lot on centralized configuration files. This is somewhat unfortunate because it inhibits more modular styles of development that you may be used to from other courses (eg placing .cpp and .h files wherever you like then informing the compiler to compile them or creating VS projects wherever you like and compiling/debugging them).

So, in UDK development, we will master the art of updating configuration files. Today, we will play around with **DefaultEngine.ini**.

If you've played PC games in the 90's or early 00's, you may be familiar with ini files already. Ini files serve as places where configuration settings are stored, divided into sections. Each section is identified by a leading tag like: `[NameOfSection]`

Following the tag will be some number of configuration lines, each line typically broken down as follows: `<Append directive, eg +>ConfigOption=Value`

1. Open **DefaultEngine.ini** by expanding the **Config** filter in your solution explorer. Note all the lovely ini files! Today, we care only about DefaultEngine.ini so double click it
2. Find the section **UnrealEd.EditorEngine**
3. Append the following line at the bottom of the section: `+EditPackages=Foobar`

The EditPackages lines tell the compiler which packages to compile. In this case, we're telling the compiler to compile your Foobar package. I also want to take a minute to explain why we have a **+** (plus) sign ahead of the line.

Default config files

The Default*.ini files are **NOT** actually the ini files that the UDK uses to load its configuration! They are in fact the **base templates** that the UDK uses to **generate** its ini files. In this case, **DefaultEngine.ini** is used to generate **UDKEngine.ini**. **UDKEngine.ini** is actually the file that the UDK reads.

So you might be wondering, *why don't we modify UDKEngine.ini directly?*

The answer is that the UDK will regenerate this file every time the Default*.ini files change or sometimes arbitrarily. This means that changes in the UDK*.ini files are **not permanent** which is why we put our changes in the Default*.ini files instead.

The reason why this whole mechanism exists in the first place is because multiple Default*.ini files are sometimes **combined** together during the generation of a UDK*.ini file.

Append directive

Since the Default*.ini files are used to generate UDK*.ini files, they contain special syntax to inform the configuration engine how certain lines should be treated. Here is a breakdown of the two forms that you will see today:

1. A line that does not start with any append directive, eg: `ConfigurationLine=Value`

2. This form of line will produce exactly one line like it in the output ini file. This means that if you have multiple configuration lines with the same configuration line name, the last one will be the one in the output file. For example, given the following input ini section:

```
FavouriteColour=Red
FavouriteColour=Green
```

3. Will produce the following output ini section:

```
FavouriteColour=Green
```

4. A line that starts with the **+** (plus) directive
5. This form of line tells the configuration engine to **append** this line to existing output. For example, given the same input as above, we get the following output:

```
FavouriteColour=Red
FavouriteColour=Green
```

Notice how each EditPackages=* line tells the compiler to compile a single package. Since we need to have multiple of these lines in the output ini, we prefix our added entries with the plus sign. We will learn at least one more directive later in the course.

Full List of Configuration Special Characters

Here's the full list of config file special characters, taken from the UDN:

Special Characters

- `+`
  - Adds a line if that property doesn't exist yet (from a previous configuration file or earlier in the same configuration file).
- `-`
  - Removes a line (but it has to be an exact match).
- `.`
  - Adds a new property.
- `!`
  - Removes a property; but you don't have to have an exact match, just the name of the property.

Note: . is like + except it will potentially add a duplicate line. This is useful for the bindings (as seen in DefaultInput.ini), for instance, where the bottom-most binding takes effect, so if you add something like:

```
[Engine.PlayerInput]
Bindings=(Name="Q",Command="Foo")
.Bindings=(Name="Q",Command="Bar")
.Bindings=(Name="Q",Command="Foo")
```

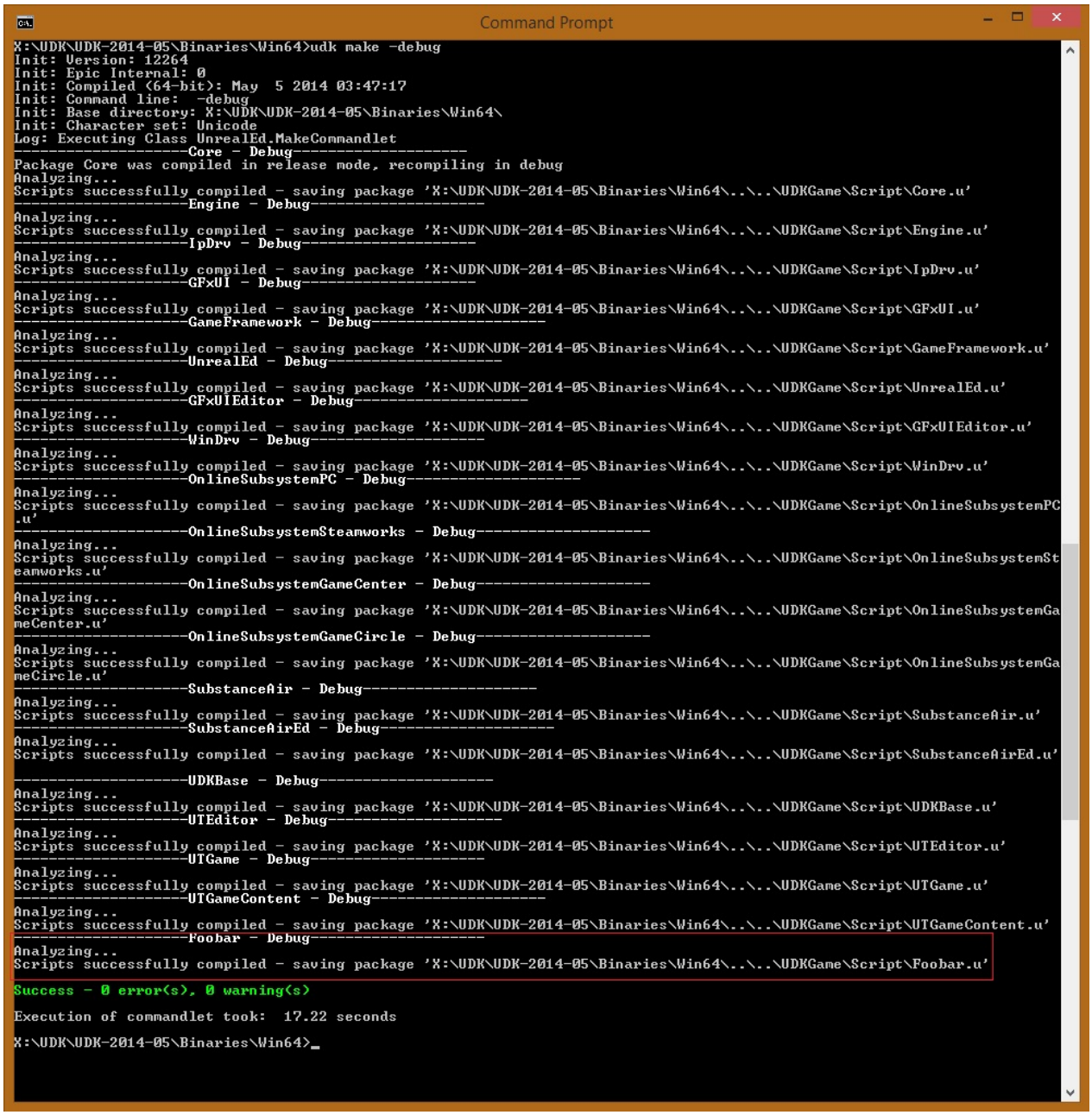
It will work appropriately. Using a + there would fail to add the last line, and your bindings would be incorrect. Due to configuration file combining, the above usage pattern can happen.

Compiling

There are many ways to compile. Today, we will learn how to compile through the command line.

1. Open the command line. Hit the **Windows** key then type **cmd** then hit enter.
2. The command line window should now be open. Navigate to the UDK binaries directory (UDK/UDK-2015-01/Binaries).
3. If you are currently using a 64-bit OS (eg windows 8.1), cd into the **Win64** directory. Otherwise, cd into the **Win32** directory.
    - This is a good time to make sure that you've saved the configuration changes to DefaultEngine.ini!
4. Type: `udk make -debug`

You should see something that looks like this:



Notice that at the bottom of the screen the output shows that the **Foobar** package was compiled. Verify that this is the case by visiting the script output directory: `UDK/UDK-2015-01/UDKGame/Script`

You should be able to find the file **Foobar.u** here. This is your first UnrealScript package, congratulations! If you can't find it here, ask for help... There is likely a problem with your configuration file.

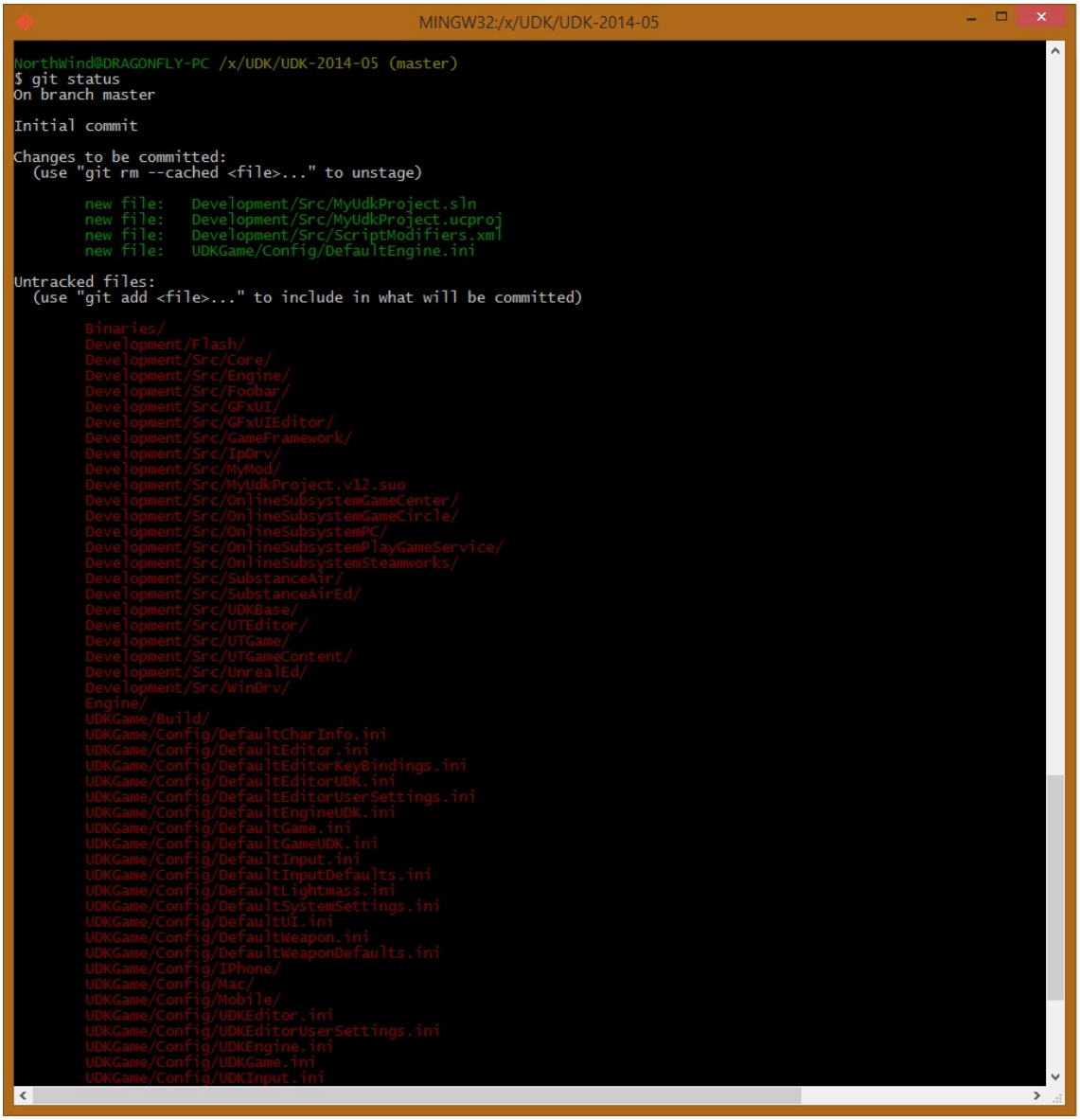
This is all that we need to do as far as compilation is concerned. In the future, this will be our primary way to compile in the UDK.

Linking your labs git repo

Linking your labs repo is simple, but not straight forward... There are many files in the UDK and we only need to **track** a few of them while ignoring many. Additionally, the locations of these files is inconvenient for us (as you will see in a minute) adding a further complication. Finally, synching the local directory with your repository must only update a few files and in-place as opposed to creating a new directory all-together. This means that our git workflow will be **non-standard** but this is something we will have to live with.

Make sure you hit Save-All in VS2013 a few times immediately before you do the following!

1. Open git bash (windows key then type **git** or **git bash**)
2. Navigate to the root UDK directory (cd /c/UDK/UDK-2015-01)
3. Initialize a git repo (git init)
4. Add your labs repo as a remote (git remote add origin https://github.com/Seneca-GAM537/ **<your seneca id>**.git)
5. Add the following files to the staging area:
    - MyUdkProject.sln (UDK/UDK-2015-01/Development/Src)
    - MyUdkProject.ucproj (UDK/UDK-2015-01/Development/Src)
    - ScriptModifiers.xml (UDK/UDK-2015-01/Development/Src)
    - The Foobar package (UDK/UDK-2015-01/Development/Src/Foobar)
    - DefaultEngine.ini (UDK/UDK-2015-01/UDKGame/Config)
    - Your staging area should look something like this:



    - Note all of the untracked files, this is part of what I mean by *inconvenient*
        - It is a good idea to develop a good .gitignore file to make your life easier. You can do this now if you wish or later on. When you do, add it to your repo as usual.
6. Commit these files with a good message (git commit -m "**<Use a good message here!>**")
7. Push up your changes (git push origin master)

Ensure that everything went correctly by viewing your repository's state on github.

That is all for part 1.

Lab 4 Part 2

Summary

In this lab, you will get an introduction to UDK game types, the UnrealScript class hierarchy, Object, Actor, some of the basics of UnrealScript, write a game type, and spawn an actor.

Learning Outcomes

- Knowledge of UnrealScript class hierarcy
- Introduction to UnrealScript and where to find resources to learn more about UnrealScript
- Introduction to game types
- How to write a new game type
- How to write an actor

Files To Implement

- L4Game.uc
- L4PlayerController.uc
- L4-Map.udk

Submission Requirements

Run your map as described above and show your instructor the special message displayed in the log window.

Instructions

Lab prep

You will use some of the skills that you learned in part 1 of this lab. If you forgot how to do something, please go back to part 1 to see how it is done.

Resume your work from github

Last lab, we initialized your lab repository. Today, we will see how we can resume where we left off. Use these instructions every time you need to begin work again on a lab computer. **Note that this is only required because the lab computers destroy your local repository when they shut down! On your home computer, you should not need to do this if you completed part 1.**

1. Open git bash
2. Navigate to the root-ish UDK directory (UDK/UDK-2015-01)
3. Init a git repo in this directory
4. Add your github labs repository as a remote (I will assume that you named your remote origin)
5. Fetch from your remote (git fetch origin)
    - This will make the local repo aware of the current state of your remote's git database
6. Checkout your remote's current master branch (git checkout -f origin/master)
    - Notice the **-f** parameter. This tells checkout to **force** the checkout. This is required because your repository has files that will overwrite the already existing files and git does not like that. **-f** forces git to proceed.
7. Create a local master so that you can make commits (git checkout -b master)
    - Notice that we used the checkout command here. This is slightly confusing but when the **-b** option is passed to checkout, it merely creates a branch at the current commit and sets the created branch as the current branch.

You should now be exactly where you left off last time.

Lab Instructions

Using the instructions from last lab, do the following:

- Create a package called Lab5 (ensure that you create the correct directory structure!)
- In this package, create two UnrealScript files: L4Game.uc, L4PlayerController.uc
- Open L4Game.uc and update it to extend the class **UTGame**
    - Set the default property **PlayerControllerClass** equal to **class'L4PlayerController'**
        - This will cause the UDK to use your player controller class
    - Set the default property **MapPrefixes[0]** equal to **"L4"**
        - This, in combination with the in-editor game type setting, will cause the UDK to use your game type when loading the map you will create soon
- Open L4PlayerController.uc and update it to extend the class **UTPlayerController**
    - In this class, above the defaultproperties block and after the extends line, create

    the function PostBeginPlay
    - This function does not return anything
    - This function does not accept any parameters
    - Have this function log out a special message (ie *This is a special message*)
    - After logging out a message, call the *super* form of this function
- Compile your package (remember, compile in debug!)
- Open the editor by using the command: **udk editor -debug**
- Create a map in the editor (the basic map template will do), set the game type in the map to your game type (L4Game)
    - Save this map as L5-Map.udk somewhere under **UDKGame/Content**
- Run your map navigating to the UDK binary in the commandline and executing the command: **udk L5-Map.udk -log -debug**

This should open your map. In the command prompt window, you should see your special message displayed. Ensure that you add and commit all of the files that you created and/or changed in this lab to your labs repository then **push**! If your L4-Map.udk file is not too large, you can consider adding it too to your repo.

This chapter will introduce you to the HUD and Canvas classes in the UDK, as well as Actors, characters, and introduce some math concepts.

Lecture Topics

- utgame map prefix
- PlayerController class selector
- PlayerController/Pawn interaction
- PlayerController/HUD interaction
- HUD class selector
- Associating maps with game types
- exec functions
- Actor location
- Vector addition/subtraction/length
- Iterator functions
- Input bindings
- Canvas
    - Setting colour
    - Drawing a box
    - Writing text

Actor Intro

Actor is the base class of all objects that may be placed in the map. In most situations, this will be the base class of objects in your map that don't need to be derived from anything more specific. Seeing as it is the base class of all placeable objects in the map, let's take a look at the various functionality it provides as this functionality will be made available to all classes that descend from it, including your own.

Location, Rotation, Scale

Since all Actor objects can be placed in the map, the Actor class contains the properties required to store *transformation* information. That is, the position, rotation, and scale of the object.

Location

```
/** Actor's location; use Move or SetLocation to change. */
var(Movement) const vector            Location;
```

Location is the position of the Actor in the map. It is defined in Actor.uc as a vector.

Rotation

```
/** The actor's rotation; use SetRotation to change. */
var(Movement) const rotator           Rotation;
```

Rotation is the amount of 3D angular offset this Actor has from the *default rotation*. It is stored as a rotator.

Scale

```
/** Scaling factor, 1.0=normal size. */
var(Display) const repnotify interp    float    DrawScale <UIMin=0.1 | UIMax=4.0>;

/** Scaling vector, (1.0,1.0,1.0)=normal size. */
var(Display) const interp    vector    DrawScale3D;
```

Scale is a factor that affects the current *size* of an object. Scale tends to be tricky/glitchy for game engines. In the UDK, Actor has two scaling variables:

- DrawScale
- DrawScale3D

DrawScale is a floating point variable while **DrawScale3D** is a vector. DrawScale is known as a *uniform scale* because it applies evenly to all 3 dimensions of the object. **DrawScale3D** on the other hand is a vector and affects each dimension differently since it has 3 members: *X*, *Y*, *Z*. **Note: While DrawScale/DrawScale3D will always affect the representation of an Actor correctly (ie particles, meshes, etc...), it may not affect the collision of the actor correctly**.

Functions

Actor contains some of the most useful functions in the UDK. We will look at two in this article; a more exhaustive list will be examined in a later article. For a reference, read **Actor.uc**.

PlayerController GetALocalPlayerController()

This function returns a player controller locally present at this computer. If none is present, this returns none. If more than one player controller exists (split screen), one of the player controllers is returned. Otherwise, the player controller of the player sitting at this computer is returned.

Actor Spawn(...)

Spawn's signature looks as follows:

```
native noexport final function coerce actor Spawn
(
    class<actor>      SpawnClass,
    optional actor      SpawnOwner,
    optional name     SpawnTag,
    optional vector   SpawnLocation,
    optional rotator  SpawnRotation,
    optional Actor    ActorTemplate,
    optional bool      bNoCollisionFail
);
```

The spawn function is used to instantiate an Actor. The class of actor to instantiate is passed in as the first parameter. An optional spawn location and rotation could be provided, otherwise the actor is spawned at the current actor's transform.

SetTimer(...)

```
/**
 * Sets a timer to call the given function at a set
 * interval.  Defaults to calling the 'Timer' event if
 * no function is specified.  If InRate is set to
 * 0.f it will effectively disable the previous timer.
 *
 * NOTE: Functions with parameters are not supported!
 *
 * @param InRate the amount of time to pass between firing
 * @param inbLoop whether to keep firing or only fire once
 * @param inTimerFunc the name of the function to call when the timer fires
 */
native(280) final function SetTimer(
    float InRate,
    optional bool inbLoop,
    optional Name inTimerFunc='Timer',
    optional Object inObj);
```

Causes a function to be called after a specified amount of time.

Properties:

- float InRate
  - Time in seconds before the target function is called
- optional bool inbLoop
  - Whether the target function should be called repeatedly in a loop
- optional name inTimerFunc='Timer'
  - The name of the target function to call on this actor. Defaults to the function `Timer`
- optional Object inObj
  - Target object to call the target function on. Defaults to the current Actor

String GetHumanReadableName()

Returns a friendly name to refer to this Actor as.

Iterators, AllActors, Tracing

In UnrealScript, an iterator is a type of specialized loop that allows you to go over a number of objects in the world according to a set of criteria.

Actor Iterators

Let's look at the iterators available through the Actor class. To use any of the following iterators, we must write the iteration loop in a member function in a class that is a descendant of Actor. The following list was fetched from the UDN UnrealScript Iterators Reference:

- **AllActors (class BaseClass, out actor Actor, optional class InterfaceClass)**
  - Iterates through all actors in the level. If you specify an optional InterfaceClass, only includes actors that implement the InterfaceClass.
- **DynamicActors(class BaseClass, out actor Actor, optional class InterfaceClass)**
  - Iterates through all actors with bStatic=false
- **ChildActors(class BaseClass, out actor Actor)**
  - Iterates through all actors owned by this actor.
- **BasedActors(class BaseClass, out actor Actor)**
  - Iterates through all actors which use this actor as a base.
- **TouchingActors(class BaseClass, out actor Actor)**
  - Iterates through all actors which are touching (interpenetrating) this actor.
- **TraceActors(class BaseClass, out actor Actor, out vector HitLoc, out vector HitNorm, vector End, optional vector Start, optional vector Extent)**
  - Iterates through all actors which touch a line traced from the Start point to the End point, using a box of collision extent Extent. On each iteration, HitLoc is set to the hit location, and HitNorm is set to an outward-pointing hit normal.
- **OverlappingActors(class BaseClass, out actor Actor, float Radius, optional vector Loc, optional bool bIgnoreHidden)**
  - Iterates through all actors within a specified Radius of the specified location (or if none is specified, this actor's location).
- **VisibleActors(class BaseClass, out actor Actor, optional float Radius, optional vector Loc)**
  - Iterates through a list of all actors who are visible to the specified Location (or if no location is specified, this actor's location).
- **VisibleCollidingActors (class BaseClass, out actor Actor, float Radius, optional vector Loc, optional bool bIgnoreHidden)**
  - returns all colliding (bCollideActors==true) actors within a certain radius for which a

trace from Loc (which defaults to caller's Location) to that actor's Location does not hit the world. Much faster than AllActors() since it uses the collision hash.

- **CollidingActors (class BaseClass, out actor Actor, float Radius, optional vector Loc)**
  - returns colliding (bCollideActors==true) actors within a certain Radius. Much faster than AllActors() for reasonably small radii since it uses the collision hash
- **ComponentList(class BaseClass, out ActorComponent out_Component)**
  - returns all components in the actor's Component's list
- **AllOwnedComponents(class BaseClass, out ActorComponent OutComponent)**
  - returns all components attached "directly or indirectly" to the actor
- **LocalPlayerControllers(class BaseClass, out PlayerController PC)**
  - returns all local PlayerControllers

Using Iterators, AllActors

The simplest iterator is AllActors. It allows you to go through all of the actors in a level that inherit from a particular base class. Let's see a simple example of using **AllActors**:

```
function Foobar()
{
    local NavigationPoint point;
    foreach AllActors(class'NavigationPoint', point)
    {
        `Log("Iterating over navigation point "$point);
    }
}
```

Iterators are invoked with the special *foreach* syntax followed by the name of the iterator, passing it any parameters in parameters like a function. Following the invocation, a code body surrounded by curly braces must be present; this code body will be invoked with every actor in the level matching the iterator's criteria.

The AllActors iterator receives two parameters:

- class BaseClass
  - This is the base class of the actors we are interested in iterating over. Our loop body will only be invoked with actors which are descendants of this class
- out Actor Actor
  - This is the variable that will be used to store the current object we are iterating over. This is similar to a counter in a for loop.
  - Note that even though the type of this variable is Actor, the variable that you pass the iterator **must be equivalent to or an ancestor of the class that you pass in through BaseClass**

In the example above, the iterator loop did the following:

- The base class that was passed in was *NavigationPoint* so the loop body would have been called once for every instance of a *NavigationPoint* object in the level
- The iteration variable passed in was *point*. Note that the type of *point* was *NavigationPoint* to match the class that we passed in as the first argument
- For each *NavigationPoint* object in the level, the following message would have been logged: `Iterating over navigation point <name_of_navigation_point>`

TraceActors

While AllActors is a very useful iterator, it is very generic. An example of a more niche and complex iterator is *TraceActors*. *TraceActors* shoots a line through the world beginning at a specific coordinate and ending at a specific coordinate, iterating over actors intersecting that line. Let's look at the parameters of this iterator:

- class BaseClass
  - Like AllActors, *TraceActors* will process only actors that are descendants of a given base class which is passed in here
- out Actor Actor
  - This is the iteration variable. Even though the specification shows this parameter's type as Actor, **it must be equivalent to or an ancestor of the class that you pass in through BaseClass**
- out vector HitLoc
  - The location in the world at which the trace line makes contact with the current iteration actor
- out vector HitNorm
  - The normal of the surface on the actor that the trace line makes contact with, facing out of the actor
- vector end
  - The end point of the trace
- optional vector start
  - The starting point of the trace. Defaults to **this** actor's location
- optional vector Extent
  - The *extent* of the trace line, effectively the radii of a box that is swept along the trace line. This should typically be left out of the iterator call.

Here's an example of using the trace actors iterator:

```
function Foobar()
{
    local Pawn iterPawn;
    local vector hitLoc, hitNorm, end, start;

    end = vect(100, 0, 0);
    start = vect(0, 0, 0);

    foreach TraceActors(class'Pawn', iterPawn, hitLoc, hitNorm, end, start)
    {
        `Log("Contacted pawn "$iterPawn$" at location "$hitLoc$" and normal "$hitNorm);
    }
}
```

Let's break down what this example will do:

- It will iterate over all actors of base class *Pawn*, storing the iteration actor in *iterPawn*
- The trace line will start at **(0, 0, 0)** and end at **(100, 0, 0)**
    - This means that if a Pawn was positioned at **(50, 10, 10)** and had a collision object big enough to contact the trace line, eg a cylinder with a **radius of at least 10** and a **height of at least 10** (to account for the pawn's position being 10 on the Y and 10 on the Z away from the line), it would be considered for iteration

An alternative way to trace actors is to use the **Trace** function on Actor which will find only the first hit actor, or the world, or None. Read more about Actor's Trace function at beyondunreal wiki

Other Iterators

The other iterators have more specific use cases. Read about them in the above UDN reference pages for more information.

UDK Characters, Controller, Pawn

In the UDK, a character is split into two parts:

- A physical representation responsible for interacting with the world and notifying the logical part of interactions and events
- A logical part that issues commands to the physical part and responds to event notifications from the physical part

The physical part of a character is the **Pawn** class and its descendants. The logical part of a character is the **Controller** class and its descendants.

Pawn

Pawn is the simpler of the two as it is a basic Actor that has the ability to move around the map in some way, collide with other Actors, and generally interact with the map. What a Pawn lacks is the *will* to do anything as it is a lot like a vehicle without a driver. Let's look at some of the more important of Pawn's properties.

```
var editinline repnotify Controller Controller;
```

The **Controller** property on a *Pawn* is a member variable that let's us retrieve the Controller object currently associated with this Pawn. If this Pawn currently has no Controller associated, this will be *None*.

```
var class<AIController> ControllerClass;    // default class to use when pawn is controll
```

The ControllerClass property stores the class of Controller to spawn when **SpawnDefaultController** is called on this pawn. This property is declared to be restricted to classes inheriting from AIController; as we will see soon, AIController is a class that inherits from Controller and it is the base class of all AI decision making controllers.

UTPawn

While the Pawn class contains the generic functionality that every *Pawn* would need (like being associated with a Controller), **UTPawn** contains the functionality required to make the Pawn shoot and run around like an Unreal Tournament character (this is what *UT* stands

for). We will sometimes extend this class when we need our Pawns to retain the functionality of a UT character.

Be warned that if you choose to use UTPawn, you should also make sure to use UTBot, UTPlayerController, and UTGame; otherwise, your code may not execute or may behave unexpectedly.

GetHumanReadableName

UTPawns override this Actor base function to return the *name* of the pawn as a character.

Controller

Much like how a Pawn is the physical part of a character that interacts with the world, a Controller is the decision making part of the character and is responsible for issuing commands to the pawn. In this way, a player and an AI could both be considered *decision makers* as a player issues commands through input devices while AI examines its environment to find the next command to issue to the pawn.

Let's look at some of the more important properties in the *Controller* class.

```
/** Pawn currently being controlled by this controller.  Use Pawn.Possess() to take contr
var editinline repnotify Pawn Pawn;
```

The **Pawn** property on a *Controller* is a member variable that let's us retrieve the Pawn object currently associated with this Controller. If this Controller currently has no Pawn associated, this will be *None*.

PlayerController, AIController

While the base class *Controller* is the most generic form of Controller in the UDK, it is not the class that you will often want to subclass. Depending on if you need to create a controller that acts based on the state of input devices or if you need to create an autonomous controller requiring no human input, you should either subclass **PlayerController** or **AIController**, respectively.

We will learn more about PlayerController and AIController in future notes. For now, it suffices for you to know that **PlayerController** represents a controller guided by input devices and that **AIController** represents an autonomous controller.

UTPlayerController, UTBot

Similar to how Pawn is descended by UTPawn which represents a Pawn possessing the traits required to be an Unreal Tournament character, **UTPlayerController** and **UTBot** descend from PlayerController and AIController respectively and represent human and bot controllers possessing the required traits to function as the decision making for Unreal Tournament characters.

Future

We will cover more Controller functionality in the future; this is all you need to get started.

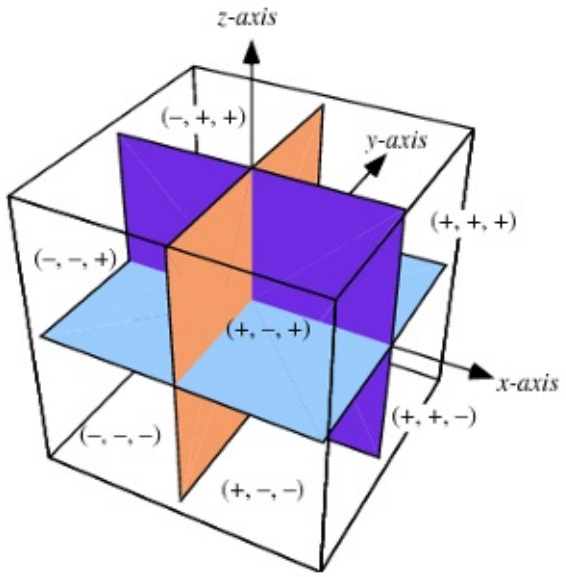
3D Math Basics, Vectors, Direction, Distance

One of the most useful forms of math to be used in the UDK is 3D vector math. Frequently, we need to be able to plot and modify coordinates given existing coordinates in a 3D environment. We also need to be able to orient various objects towards and away from each other. We use vector, rotator, and quaternion math for this. This notes section will deal with the basics of vector math, including vector subtraction, as well as directioning and rotators.

Dimensions

A dimension is a *degree of freedom*. This concept is not intuitive but it can be related to the real world in order to better grasp it. For example, in the real world, we have 3 spatial dimensions and 1 time dimension (at minimum). Since we are only dealing with spatial quantities, let's disregard the time dimension. This leaves us with 3 spatial dimensions.

If we think of the real world universe as a huge box, we can represent positions along its width with a dimension, we can call this dimension **X**; positions along its length can also have a dimension, let's call this one **Y**; finally we will use a dimension to plot positions along the box's height, we will call this dimension **Z**.

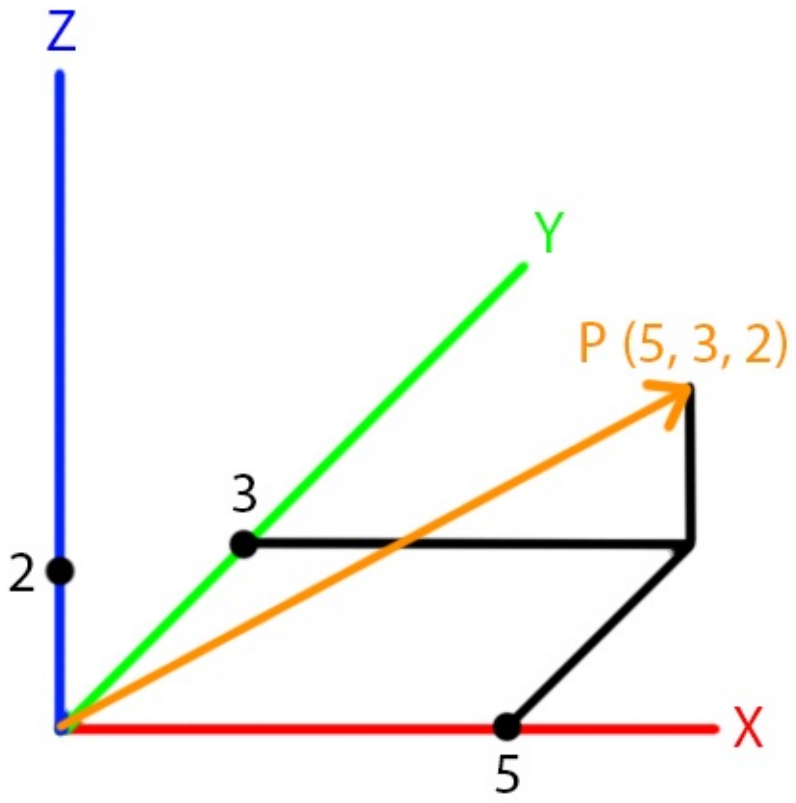


Now there are some complications to this example, like the fact that depending on the viewer's current rotation, width, length, and height may have different directions. This is solved by the fact that all virtual worlds that use this type of position information always have

a **canonical** rotation, which is the one **original** rotation that the world has; its neutral rotation, if you will. The width, length, and height of the universe are always interpreted with this rotation in mind.

Vectors then describe a specific point in this universe. We examine vectors in the following section.

Vector Position



A vector is a mathematical quantity that represents a position along multiple dimensions. That is why vectors are typed by the number of dimensions that they encompass. The following are some common vector types:

- Vector 1, storing 1 dimensional position, eg X
- Vector 2, storing 2 dimensional positions, eg X, Y
- Vector 3, storing 3 dimensional positions, eg X, Y, Z

In the UDK, the type *vector* is a Vector 3 since it has 3 dimensions. It is ideal for describing coordinates in the UDK world. For example, a vector that holds the values **X:+5, Y:+3, Z:+2** is a vector that describes a position in the world that is:

- 5 units along the positive X axis from the origin of the world
- 3 units along the positive Y axis from the origin of the world
- 2 units along the positive Z axis from the origin of the world

You may have noticed use of the term **origin**. The origin of the world is simply the vector **(0, 0, 0)**.

UDK, Declaration

The type used to store a vector in the UDK is *vector*. It can be used as follows:

```
var vector MemberVectorA;
var vector MemberVectorB;
local vector LocalVector;
```

Like any other variable type, we can declare a variable as a member variable or as a local variable. We can initialize a vector declared as a member variable by using **parenthesis** syntax. For example:

```
Defaultproperties
{
    MemberVectorA=(X=10,Y=3,Z=7)
    MemberVectorB=(Z=6)
}
```

In this case, we initialized **MemberVectorA** to a vector with an X component of 10, Y component of 3, and Z component of 7. Similarly, we initialized **MemberVectorB** to a vector with an X and Y component of 0, and a Z component of 6. When using parenthesis initialization syntax, any component not mentioned is set to 0.

UDK, Assignment

As you would expect, we can assign to/from vectors. We can read and write to individual vector components, ie:

```
local vector ExampleVector;
local float ExampleFloat;

ExampleVector.X = 10;
ExampleVector.Y = 62;
ExampleFloat = ExampleVector.Y;
```

In this case, we set the X and Y components of ExampleVector to 10 and 62, respectively. Then, we read the Y component and assign it to ExampleFloat; in this case, ExampleFloat's final value is 62. Notice that we did not assign anything to ExampleVector's Z component; by doing so, we leave the value at 0. In UnrealScript, all variables initialize to 0 or the equivalent of 0.

We can also assign whole vectors:

```
local vector A, B;

A.X = 64;
A.Z = 73;
B = A;
```

In this case, we assign 64 to A's X component and 73 to A's Z component. This means that A's XYZ components currently hold the values 64, 0, 73. We then assign B to A. By the end of the segment, both vectors now hold the values 64, 0, 73.

As a convenience, we can create and assign **vector literals** by using the function **vect**. For example:

```
local vector A;
A = vect(10, 0, 20);
```

In this case, A now holds the XYZ values 10, 0, 20. Note that the **vect** keyword **cannot** be passed variables; it must be passed float literals. For example, the following would be a **syntax error**:

```
local vector A;
local float X;

X = 10;
A = vect(X, 0, 20);  // Syntax error, cannot pass variable to vect
```

Vector Scaling

One of the most important operations that could be performed on a vector is scaling. A vector can be scaled by a float by multiplying each of its components by the floating point value. For example:

Given the vector **X:2, Y:4, Z:6**, we could do the following:

```
(2, 4, 6) * 3 = (6, 12, 18)
(2, 4, 6) * 0.5 = (1, 2, 3)
```

The result of multiplying a vector by a floating point value is another vector. This operation allows us to linearly magnify or shrink a vector.

UDK, Float Multiplication

In the UDK, float multiplication is done naturally. For example:

```
local vector A, B, C;

A = vect(2, 4, 6);
B = A * 2;
C = A * 0.5;
```

The result of this code is that the vector B will hold the value **(4, 8, 12)** while the vector C will hold the value **(1, 2, 3)**.

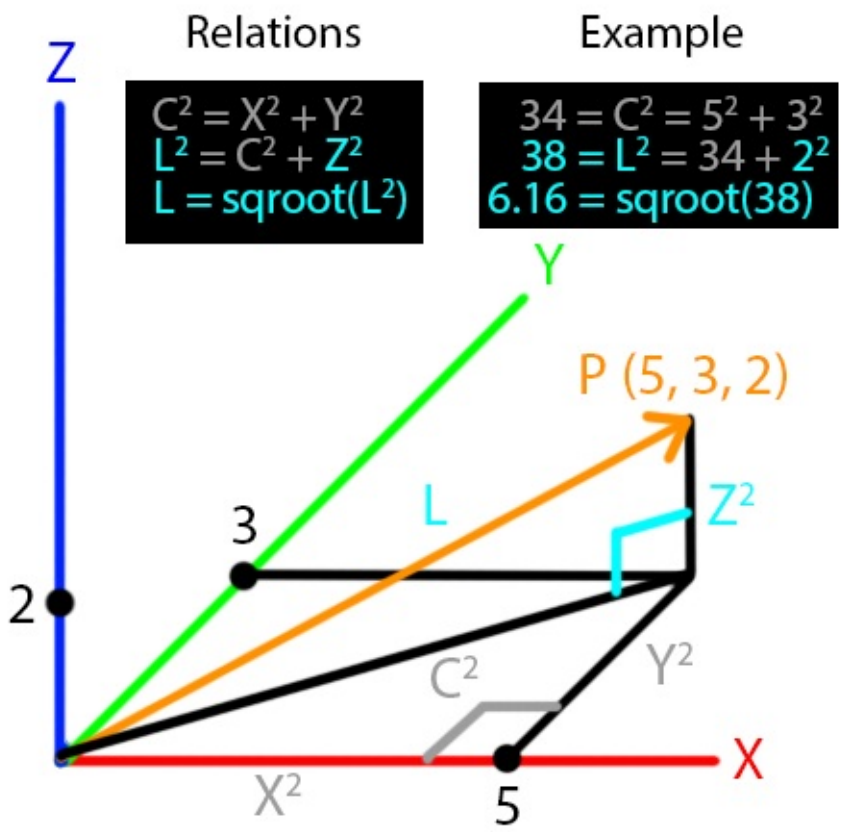
Vector Length

It can be useful to find the **length** of a vector. This is done by using the pythagorean theorem, applying it twice to find the length of a vector. As a reminder, the pythagorean theorem as follows:

```
Let A and B be the lengths of the non-hypotenuse sides of a right angle triangle.
Let C by the length of the hypotenuse of the same right angle triangle.

Then:
A^2 + B^2 = C^2
```

We can use this to find the length of our vector from the example before:



In the above example, we can form two right angle triangles:

- The *grey* triangle flat on the base, formed between the X and Y components
- The *cyan* triangle upright, formed between the Z component and the hypotenuse from the *grey* triangle

So with the pythagorean theorem in mind, we can:

1. Find the squared length of the hypotenuse of the *grey* triangle
2. Use the squared length of the hypotenuse from the last calculation in finding the squared length of the hypotenuse of the *cyan* triangle

Once we have the squared length of the hypotenuse of the *cyan* triangle, we can take the square root to get the length of the hypotenuse of the *cyan* triangle, which is equivalent to the length of the vector.

So, our final equation looks like: `length(vector) = sqroot(vector.X^2 + vector.Y^2 + vector.Z^2)`

UDK, Length or Size

We use the function **VSize** to find the length of a vector in the UDK. It is defined in **object.uc** as follows:

```
native(225)        static final function    float    VSize        ( vector A );
```

Notice that this function accepts a vector and returns a float. The returned value is the length of the vector. For example, given our vector in the picture above:

```
local float sz;

sz = vsize(vect(5, 3, 2));
```

After this piece of code, sz will contain the value 6.16.....

Vector Direction, Normalization

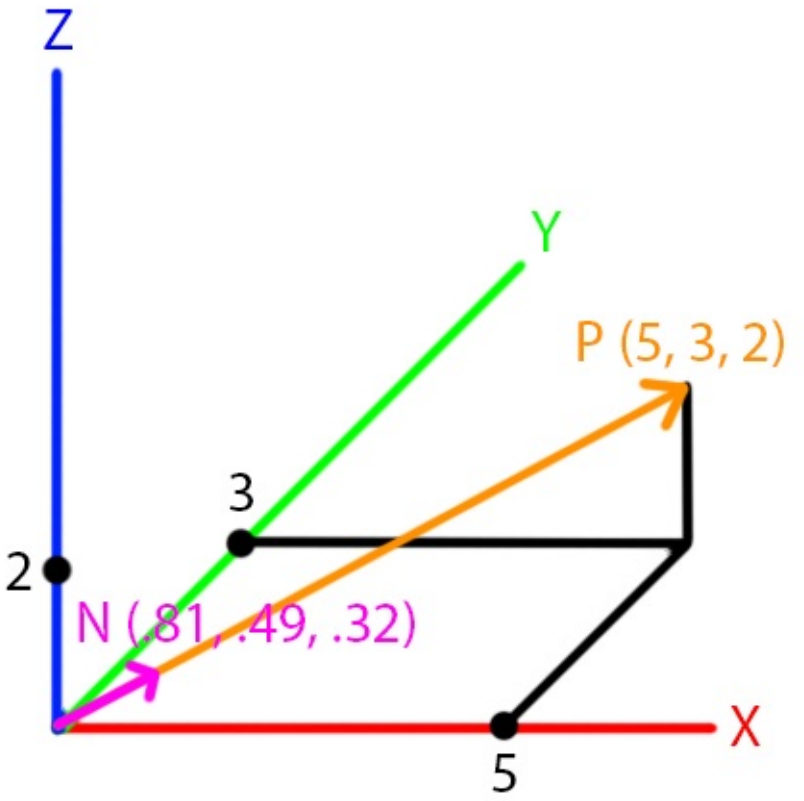
Since we've extracted **length** from a vector, we can also extract **direction** from a vector. A direction vector is nothing more than a vector with a size of 1. We call this type of vector a **unit** vector.

So why 1? Why is that size significant? Why not 2, or 63? The reason is that a size 1 vector can be multiplied by any float to generate a vector with the **same direction**, sized at **exactly the value of the float**.

We can get a size 1 vector from any existing vector (except the vector (0, 0, 0)) by dividing each of the components of the vector **by the size of the vector**. For example, given our previous example vector, we can find a direction vector as:

```
exampleVector = (5, 3, 2)
length = 6.16...
direction     = (5 / 6.16..., 3 / 6.16..., 2 / 6.16...) = (0.81..., 0.49..., 0.32...)
```

We get the approximate direction vector **(0.81, 0.49, 0.32)**. Visually, this looks like:



We call this process, **vector normalization**; the idea is that a **normalized** vector is a vector whose length is **exactly equal to 1**.

UDK, Normal

To get a normal vector in the UDK, we use the **Normal** function. It is defined in **object.uc** as follows:

```
native(226)        static final function    vector    Normal        ( vector A );
```

This can be used as follows:

```
local vector A;

A = normal(vect(5, 3, 2));
```

In this case, A will be approximately equivalent to the vector **(0.81, 0.49, 0.32)**.

Vector Addition, Subtraction

We can **add** and **subtract** vectors from each other. Addition and subtraction in vectors function similarly to how they do with *scalar* floating point values. That is, the following relations are held:

```
If A + B = C, then A = C - B
```

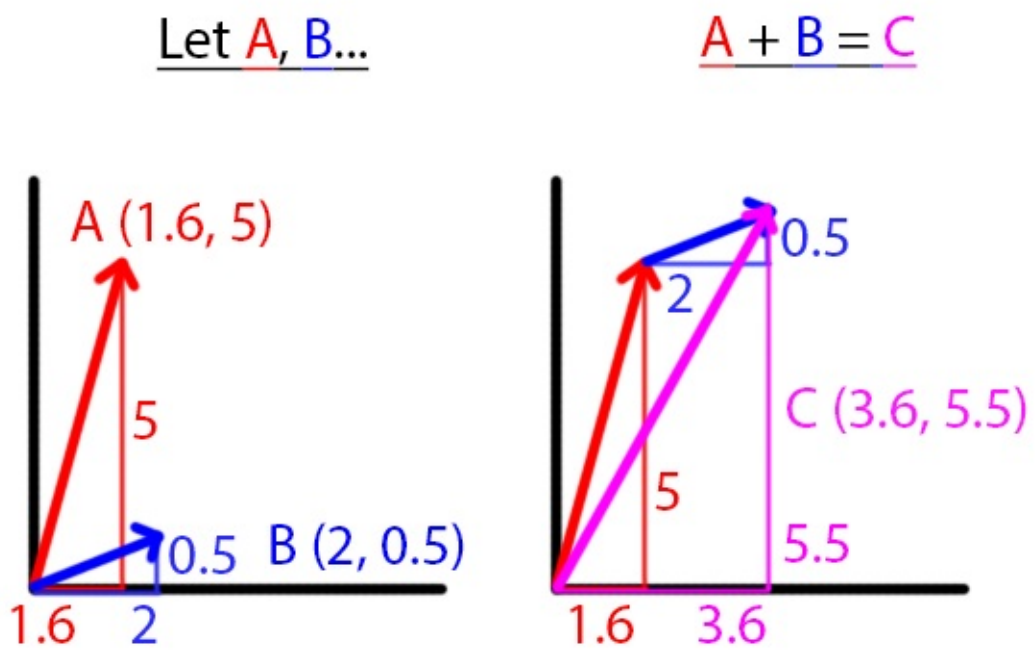
If we substitute some float scalars above, we see that this naturally holds:

```
2 + 3 = 5, then 2 = 5 - 3
```

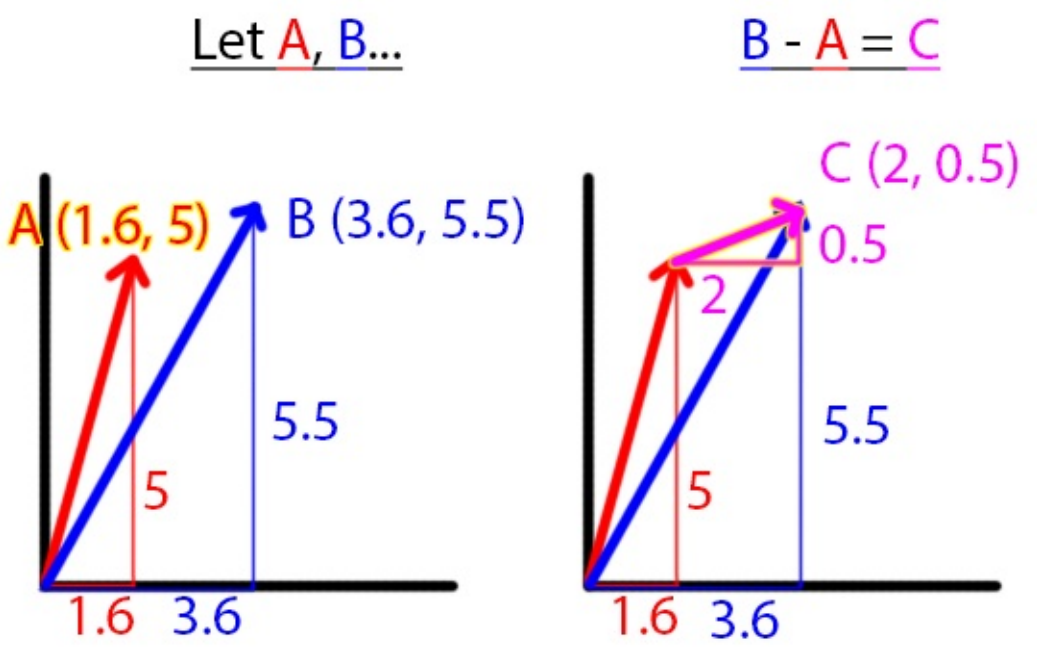
When adding and subtracting vectors, we simply apply the operations on each component, respectively. For example:

```
(1, 2, 3) + (4, 5, 6) = (5, 7, 9), then (1, 2, 3) = (5, 7, 9) - (4, 5, 6)
```

Visual Addition



Visual Subtraction



UDK, Add/Subtract Operators

This functions as you would expect in the UDK:

```
local vector A, B, C, D;

A = vect(1, 2, 3);
B = vect(4, 5, 6);
C = A + B;
D = C - B;
```

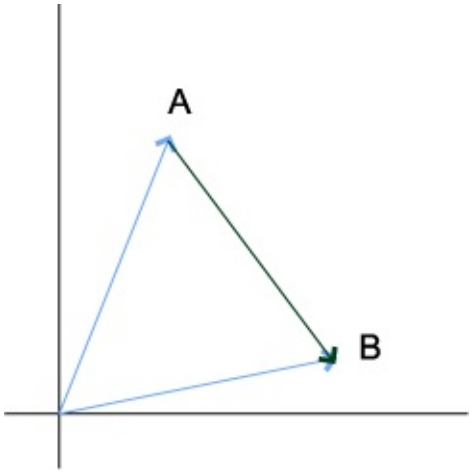
at the end of the above segment, **C** holds the value **(5, 7, 9)** and **D** holds the value **(1, 2, 3)**.

Vector Subtract For Orientation

Since:

- Vectors can be used as coordinates
- The relationship **"C - B = A, therefore A + B = C"** holds

Notice the following visual phenomenon:



In the picture above, adding the **green** vector to the **A** vector, results in the **B** vector. Therefore, **B - A = green vector**. We can use this relationship to find a vector that **travels** from a **source coordinate to a destination coordinate**. We can use this in a number of very useful ways:

Getting Offset Vector

An offset vector is a vector that when added to a source vector, will move it by a certain amount. For example, to get a vector that transforms a given vector **A** to a given vector **B** when **added to A**, we simply subtract **B from A**. For example:

```
Source = (1, 2, 3)
Destination = (10, 10, 10)
Offset = Destination - Source = (10, 10, 10) - (1, 2, 3) = (9, 8, 7)

Now notice that:
Source + Offset = Destination

So:
(1, 2, 3) + (9, 8, 7) = (10, 10, 10)
```

Getting Distance

If we can generate a vector that travels from a source to a destination, we can get the distance between those two points by finding the length of the offset vector. For example:

```
Source = (1, 2, 3)
Destination = (10, 10, 10)
Offset = Destination - Source = (10, 10, 10) - (1, 2, 3) = (9, 8, 7)
Distance = length(9, 8, 7) = 13.93.....
```

Getting Direction Vector

A direction vector is similar to an offset vector but is sized 1. It is useful when we don't need the **distance**.

Use this when you need to:

1. When you are doing some vector math and the functions you are using don't tolerate vectors with a size other than 1
2. When you need to get a vector **in a particular direction** but sized **the same as another vector**, eg **when rotating vectors**

This is done by taking the normal of the offset vector. For example:

```
Source = (1, 2, 3)
Destination = (10, 10, 10)
Offset = Destination - Source = (10, 10, 10) - (1, 2, 3) = (9, 8, 7)
Distance = length(9, 8, 7) = 13.93.....
Direction = normal(offset) = (9, 8, 7) / 13.93... = (0.65..., 0.57..., 0.5...)
```

Input Binding Basics

Input in the UDK is as simple as associating various input commands with exec functions that should be called when those inputs are triggered. Let's look at this in detail.

DefaultInput.ini

DefaultInput.ini is the configuration file where input bindings should be placed.

The name of the property responsible for associating a particular input with an exec function is **Bindings**. Let's look at an example:

```
.Bindings=(Name="F4",Command="Playersonly")
```

Note a few things:

- We're prefixing the **Bindings** keyword with the **.** operator which appends this line to the Bindings array when input.ini is built from the default configuration files
- We are associating the button **F4** with the exec function **Playersonly** such that when F4 is pressed, the function Playersonly is called
    - Note that the function is **not** continuously called, it is called once per *keydown* event

The full list of inputs that could be associated is on the UDN here.

In the simplest use case, this is adequate and will fulfill most of your requirements. You can do a few more tricks, however.

Binding to Member Variables

It is also possible to associate inputs with member variables directly. To do this, we define a target *input* variable on a class that derives from **Controller** or **Input**; typically, this would be done either on a subclass of **PlayerController** or **PlayerInput**.

For example, assume that the following variable is defined on a subclass of the PlayerInput class:

```
var input byte isPressing;
```

Note use of the special variable qualifier **input**. This allows the input system to set the value of this variable directly, if later referenced in DefaultInput.ini.

Let's look at how we could use this in DefaultInput.ini:

```
.Bindings=(Name="LeftMouseButton",Command="Button isPressing")
```

The system will now set the value of isPressing to 1 if the left mouse button is currently pressed, and 0 otherwise. Also notice use of the command qualifier **Button**. Let's look at qualifiers like this in detail.

Command Qualifier: Axis

Axis sets a **float** variable usually within the range -1.0f and 1.0f. The main exception to this is the mouse input axes. Let's look at an example of **Axis** in action:

For example, assume the following variable is defined on a subclass of the PlayerInput class:

```
var input float mouseYAxis;
```

We can then use this variable like this:

```
.Bindings=(Name="MouseY",Command="Axis mouseYAxis")
```

mouseYAxis will now hold the number of pixels that the mouse has moved on the Y axis in the last frame. You can learn more about axis on the UDN here

Command Qualifier: Button

Button sets a **byte** variable to either 0 or 1, depending on whether the key is currently released or pressed, respectively.

For example, assume the following variable is defined on a subclass of the PlayerInput class:

```
var input byte f3Pressed;
```

We can then use this variable like this:

```
.Bindings=(Name="F3",Command="Button f3Pressed")
```

f3Pressed will now be set to 1 if the button **F3** is currently pressed, and 0 otherwise.

Command Qualifier: Count

Contrary to how this qualifier is named, this does **not** create an input mapping that increments a variable. Instead, the **number of samples** of the particular input in the last frame is reported; that is, how many times in the last frame the input mapping was queried. This is useful for things like mouse smoothing.

Count operates on **byte** variables.

For example, assume the following variable is defined on a subclass of the PlayerInput class:

```
var input byte mouseXSamples;
```

We can then use this variable like this:

```
.Bindings=(Name="MouseX",Command="Count mouseXSamples")
```

Command Qualifier: Toggle

Toggle flips a byte from 0 to 1 or from 1 to 0 every time the associated input is pressed down.

For example, assume the following variable is defined on a subclass of the PlayerInput class:

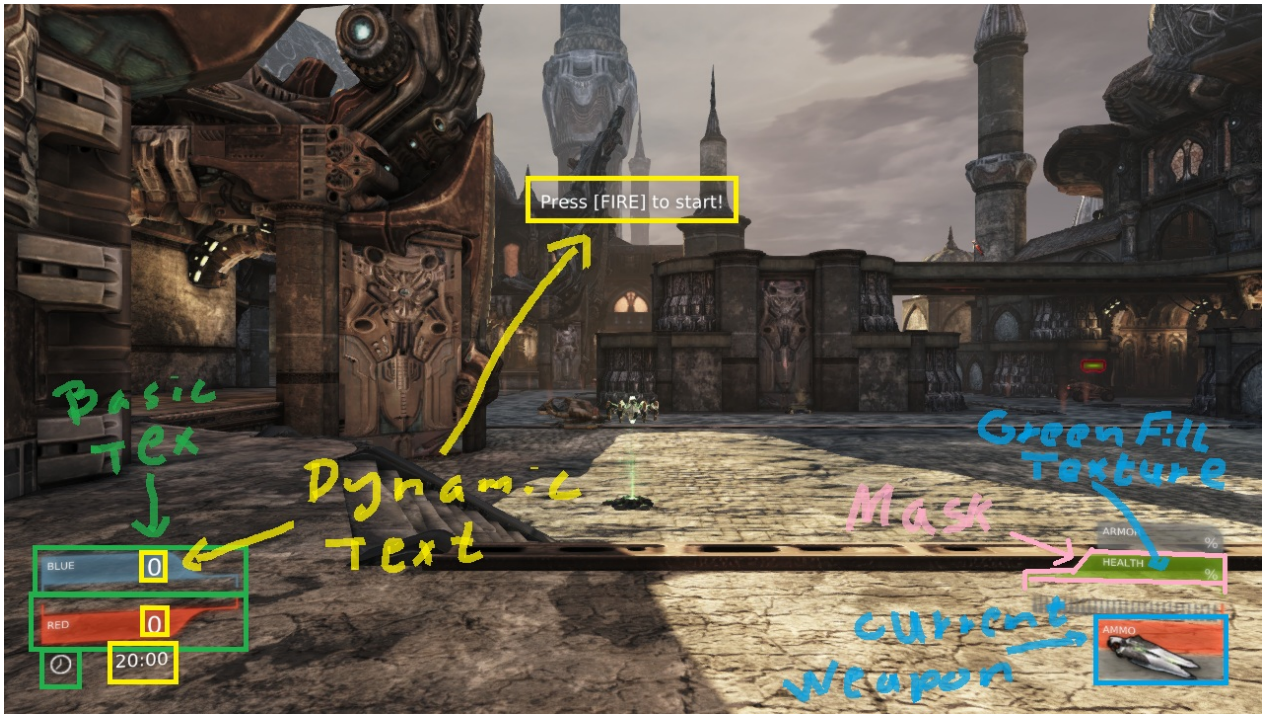
```
var input byte MState;
```

We can then use this variable like this:

```
.Bindings=(Name="M",Command="Toggle MState")
```

HUD & Canvas Intro

The HUD of a game is what defines what flat on-screen features are visible on a screen. These features are used to display important information to the player. For example:



In this case, the HUD contains a number of dynamic elements and static elements.

1. The text in the middle of the screen, the team score, and the remaining time are all dynamic text elements
2. The current weapon image is static as long as the weapon does not change, ie it changes when the player changes weapon
3. The ammo stock (the bullets) reflect how much ammo is in the weapon where grey bullet icons reflect the lack of a bullet and white/red bullets means remaining ammo. If the weapon's ammo is greater than 32, the excess ammo is not displayed
4. The player's health is the combination of a transparency mask and a green square texture that is scaled from the left side to the right side

UDK HUD Implementations

There are two ways to build a UI using the UDK.

1. Scaleform. This is a technology that backbones on Adobe flash. Although this is an incredibly useful and flexible technology, we will not be discussing it as it requires a level of familiarity with Flash and ActionScript (AS3 to be specific). If you are comfortable with these technologies and would like to explore Scaleform for your

project, please feel free to do so; if you wanted to go down this road, you should start by reading the Scaleform quickstart in the Scaleform part of the UDN available here.

2. Canvas. This is a way to programmatically add 2D assets to the screen. This method is what we will be discussing in this course as it requires no prior expertise and is simple to hook into game logic.

Attaching a HUD to a game type

In order to attach a HUD to a game type and have that HUD be used whenever that game type is used, we set the default property **HUDType** to be the HUD class itself. For example, this is how the HUD is set in the **UTGame** class. `HUDType=class'UTGame.UTHUD'`

Now if your game type did *NOT* inherit from UTGame somewhere in the inheritance chain, you would be done. However, if it *DOES* inherit from UTGame at some point, then you additionally need to set the default property **bUseClassicHUD** to true. The variable bUseClassicHUD is defined in the class UTGame:

```
/** Whether to use Classic UTHud */
var bool bUseClassicHUD;
```

To see why that is, take a look at the following function found in the class **UTGame**:

```
/** handles all player initialization that is shared between the travel methods
 * (i.e. called from both PostLogin() and HandleSeamlessTravelPlayer())
 */
function GenericPlayerInitialization(Controller C)
{
    if ( !bUseClassicHUD )
    {
        HUDType = bTeamGame ? class'UTGFxTeamHUDWrapper' : class'UTGFxHudWrapper';
    }
    super.GenericPlayerInitialization(C);
}
```

Notice that if **bUseClassicHUD** is false (which it is by default), the player's HUDType is *overridden to be one of class'UTGFxTeamHUDWrapper' or class'UTGFxHudWrapper'*, regardless of what the value of HUDType *happens to be* in the default properties of the game type!

HUD Class

The HUD class itself is not a particularly complex class, but it does contain just enough that you can get a lot done. It has two important members:

1. The DrawHUD function. This member function returns nothing and receives no arguments. This is the function that is called every frame (tick) to actually draw the elements of the HUD. This is where your HUD drawing code should reside.
2. The Canvas member variable. This is a reference to a Canvas class instance that can be used to *actually draw* onto the screen.
3. The PlayerOwner member variable. This is a reference to the *PlayerController* object that owns this HUD. Defined as follows: `var PlayerController PlayerOwner; // always the actual owner` Use this to gain access to the player controlling the HUD.

Additionally, the HUD class itself has a few functions that can be used to make drawing some objects simpler; we will discuss those further on in this article.

Drawing with Canvas

The Canvas class represents a *stateful* drawing surface. That is to say, you should imagine this class like a machine with many levers. You set the levers to desired positions indicating your preferences for later actions then press the execute button performing an action. Let's look at some of the most useful properties that could be set.

As an analogy, imagine that you had to design a system that draws boxes on a screen; for this example, we can assume that the edges of these boxes will be parallel to their matching screen edges. So, the minimal set of information that you would need to draw a box would be:

- How far up the screen the top edge is
- How far up the screen the bottom edge is
- How far from the left of the screen the left edge is
- How far from the left of the screen the right edge is

In this case, you would assume that the coordinate for the top edge is greater than the coordinate for the bottom edge. Similarly for the right and left edge. Typically in GUI systems, this information is stored in the form of the **top left corner** and the **bottom right corner**.

The simplest way that we could imagine this being put together is as follows:

A single function accepting four parameters that actually draws the rectangle. Eg: `DrawRect(float top, float left, float bottom, float right);` Alternatively: `DrawRect(float top, float left, float width, float height);`

- Advantages:
  - Simplicity, as the correlation between the code and the function performed is intuitive
- Disadvantages:

- Adding parameters with future revisions means either changing the function signature or adding more and more function overloads
- Repeated drawing calls that use the same top left corner must resupply the top left parameter on every call

Instead of going down this road, the UDK takes some of those parameters and rips them out into their own functions. Those functions then set *state* on the Canvas object which drawing functions use when actually drawing. The above analogy is instead implemented as follows:

```
SetPosition(float left, float top);
DrawRect(float width, float height);
```

Instead of passing both top-left along with width-height, we separate out the two concepts. This means we can now do code like this:

```
SetPosition(100, 200);
DrawRect(50, 50);
DrawRect(100, 100);
DrawRect(200, 200);
```

Notice how we didn't have to resupply the top left position on every draw call. This is what it means for an object to be *stateful*.

Note that Canvas has a wide array of functionality and we will not be able to discuss all of it. Please take a look at the canvas technical guide on UDN to learn more.

Canvas Properties

The most important properties of the Canvas object are its position, colour, and size.

Position

This is typically the top left position of the next drawing call. This is set by calling SetPos: `SetPos(float x, float y, float z = 1.0);` In this case, x and y are pixel coordinates where the top left corner of the next draw should be performed. Note that based on the draw call itself, it is possible that this position may *not* be treated as the top-left corner for that particular draw call (eg when drawing *centered* text). Notice that the z parameter has a default value; therefore, we typically omit passing a value for the z parameter.

Colour

This is the colour of the next draw call (if the next draw call uses a colour). Colours are represented by four **integer** values ranging from 0 to 255 where 0 is no contribution and 255 is maximum contribution:

- Red. This is the contribution of the red channel.
- Green. This is the contribution of the green channel.
- Blue. This is the contribution of the blue channel.
- Alpha. This is how transparent you want the next draw to be where **0 is completely transparent** and **255 is completely opaque**.

Colour is set by calling SetDrawColor or SetDrawColorStruct. SetDrawColor:
`SetDrawColor(byte R, byte G, byte B, optional byte A = 255);`

SetDrawColorStruct is just a wrapper for SetDrawColor that accepts a Color struct object.

```
/** Set the draw color using a color struct */
final function SetDrawColorStruct(color C)
{
    SetDrawColor(C.R, C.G, C.B, C.A);
}
```

Note that a **Color** struct has four integer members, R, G, B, and A that function similarly to what is described above. **BE CAREFUL** that numeric variables in the UDK are initialized to 0 which implies the following:

```
function myFunc()
{
    local Color myColor;
    myColor.R = 255;
}
```

In this simple function we are describing a color object with Red set to 255 while blue, green, **and ALPHA** are all set to zero. When writing code like this, **make sure that you explicitly set the alpha value of your to colour struct to something other than 0!**

Size

The size of the screen in pixels can be retrieved from the Canvas object. This is useful when you need to draw objects to the screen in proportion to the size of the screen as opposed to in absolute pixel dimensions.

Retrieve the canvas members using the **SizeX and SizeY** members. These represent the dimensions of the screen in pixels, with 0,0 being the top-left corner. These are defined in **Canvas.uc** as:

```
var const int SizeX, SizeY;  // Zero-based actual dimensions.
```

HUD Utility Functions, Draw*Line

The HUD class itself provides two useful functions that you may choose to use. They are **Draw3DLine** and **Draw2DLine**. They are defined in **HUD.uc** as follows:

```
// Draw3DLine  - draw line in world space.
native final function Draw3DLine(vector Start, vector End, color LineColor);
native final function Draw2DLine(int X1, int Y1, int X2, int Y2, color LineColor);
```

Notice that unlike *Canvas'* draw functions, these draw functions accept a color parameter. This is because, unlike Canvas, the HUD class does *not* hold any colour state and these are merely utility functions.

Use **Draw3DLine** when you need to draw a line in the map world given two world locations as a starting point and an ending point.

Use **Draw2DLine** when you need to draw a line on the screen given pixel coordinate starting and ending points.

Lab 5 - Canvas, Input, Vector Subtract

Summary

You will implement a basic HUD and use Canvas as well as HUD functions to draw objects to the screen according to the state of the player controller.

Learning Outcomes

- Familiarity with Canvas and how to learn more about it
- Familiarity with new inputs
- Essential 3D math

Files to Implement

- L5Game.uc
- L5Player.uc
- L5HUD.uc
- L5-Map.udk
- Update: DefaultInput.ini

Submission Requirements

Show your instructor your working hud and inputs by starting the map, adding some bots using the **]** key, updating the counter with the **.** key, and displaying the correctly working hud on-screen.

Instructions

- Create the script package **Lab6** and implement all of the following in it
- Create a game type called L5Game, have it extend the **UTDeathmatch** class, and associate it with maps with the prefix **L5**
  - Set the player controller class for this game type to be the class **L5Player**
  - Set the hud class for this game type to be the class **L5HUD**
- Create a basic map with the game type set to L5Game

- Create the class L5Player that extends UTPlayerController
  - Add an integer member variable to this class to track how many times **IncrementSpecialMessage** was called
  - Add the exec member function **IncrementSpecialMessage** to L5Player
    - Have this function increment the integer member described above every time this function is called
    - Have this function output a debug log message every time it is called so that you know that it was called while developing
  - Add a **Pawn** type member variable named **closestPawn** to this class to track the closest **Pawn** that is not this player's pawn
  - Every tick (override the function PlayerTick), have this class iterate over all pawns in the world
    - Store the closest pawn that is not this player's own in the above member variable
- Add 2 new input bindings to DefaultInput.ini:
  1. Bind the **.** (period key, beside comma) to the **IncrementSpecialMessage** command
  2. Bind the **]** key to the command **AddBots 3** (You can copy paste this command)
- Create the class L5HUD that extends the class HUD
  - Create a **DrawHUD** member function to display the following screen features:
    1. Draw a blue box sized 0.2 of screen width and 0.5 of screen height with the top left positioned at 0.7 of the screen width and 0.1 of the screen height
    2. If our local player's pawn is not none and the local player's closest enemy pawn is also not none:
       - Draw a green 3D line that starts at the player pawn's location and ends at the player's closest enemy pawn's location
       - Write text inside the blue box detailing the name of the targeted pawn, something like: `Found close enemy Monarch`
         - In this case, Monarch is the close enemy pawn's name
    3. If the local player's special message counter is above 0, write another line of text inside the outline box: `Counter: 21`
       - In this case, the value beside counter is the value currently contained in the special message counter integer defined above

Your end result should look something like this:

Camera

Lecture Topics

- CalcCamera
- PlayerController bBehindView
- Rotators
- Casting

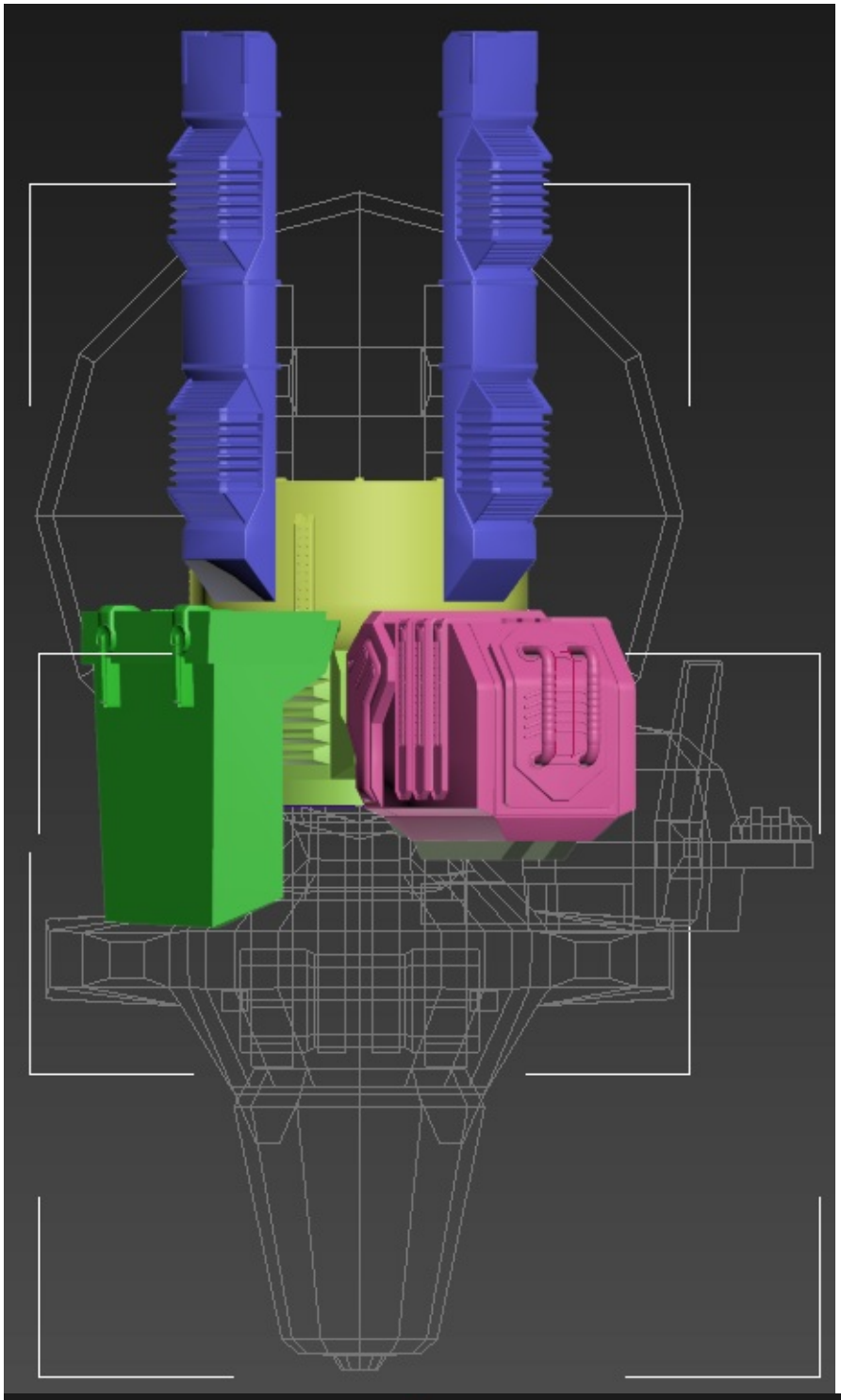
Camera Basics

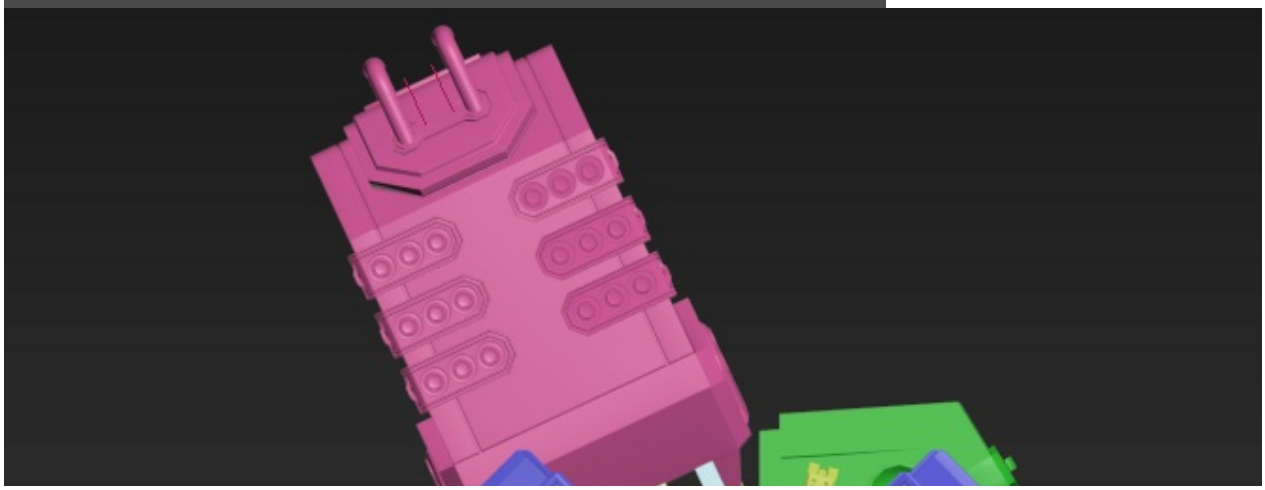
A camera in the most abstract sense is a perspective from which the world should be viewed. There are two types of cameras that could be set up:

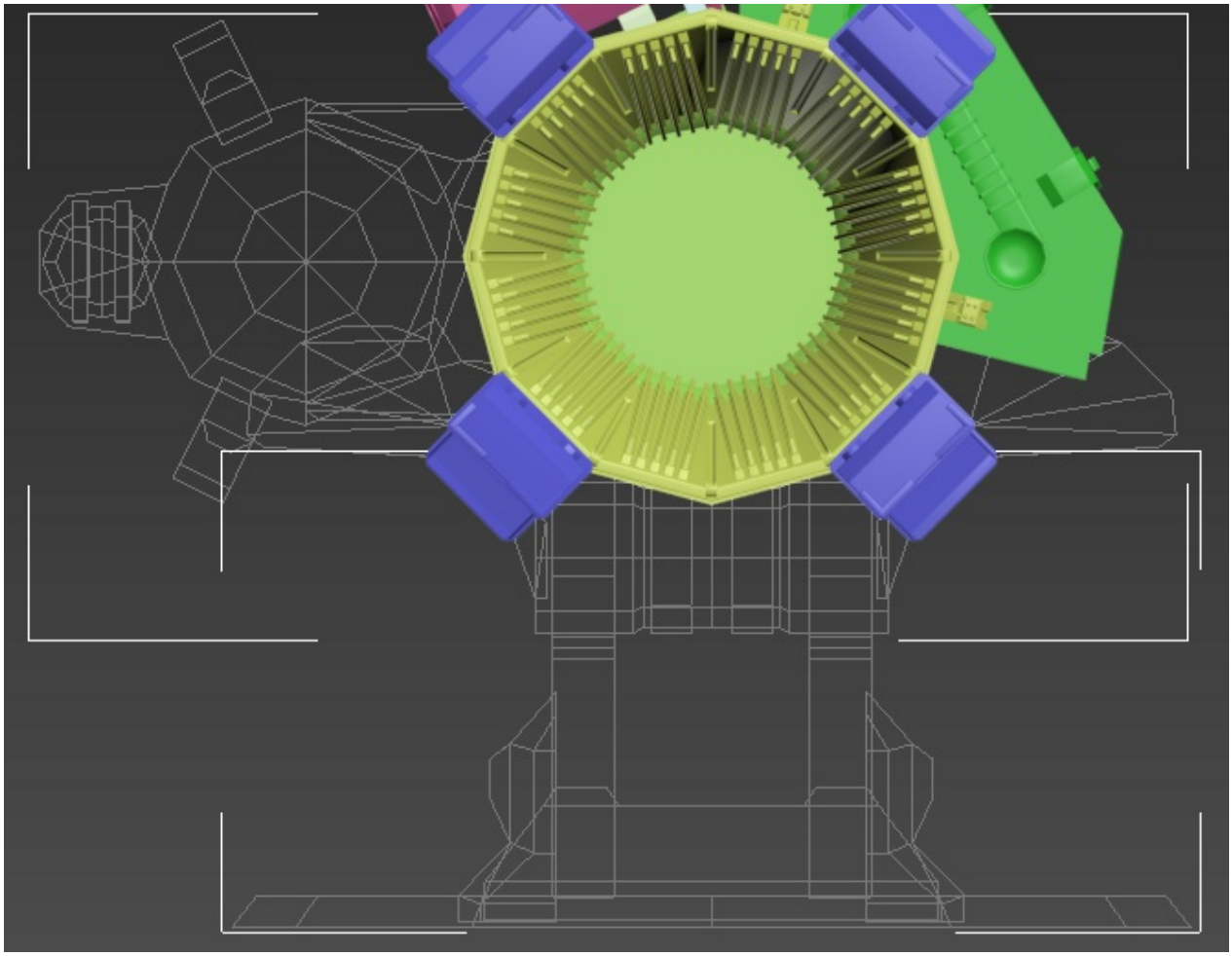
- Orthographic
- Perspective

Orthographic Camera

This type of camera views a "flat" world, that is to say that the position objects in the image is not affected by the distance of the object from the camera. This is different from the way human beings view the world. It looks something like this:



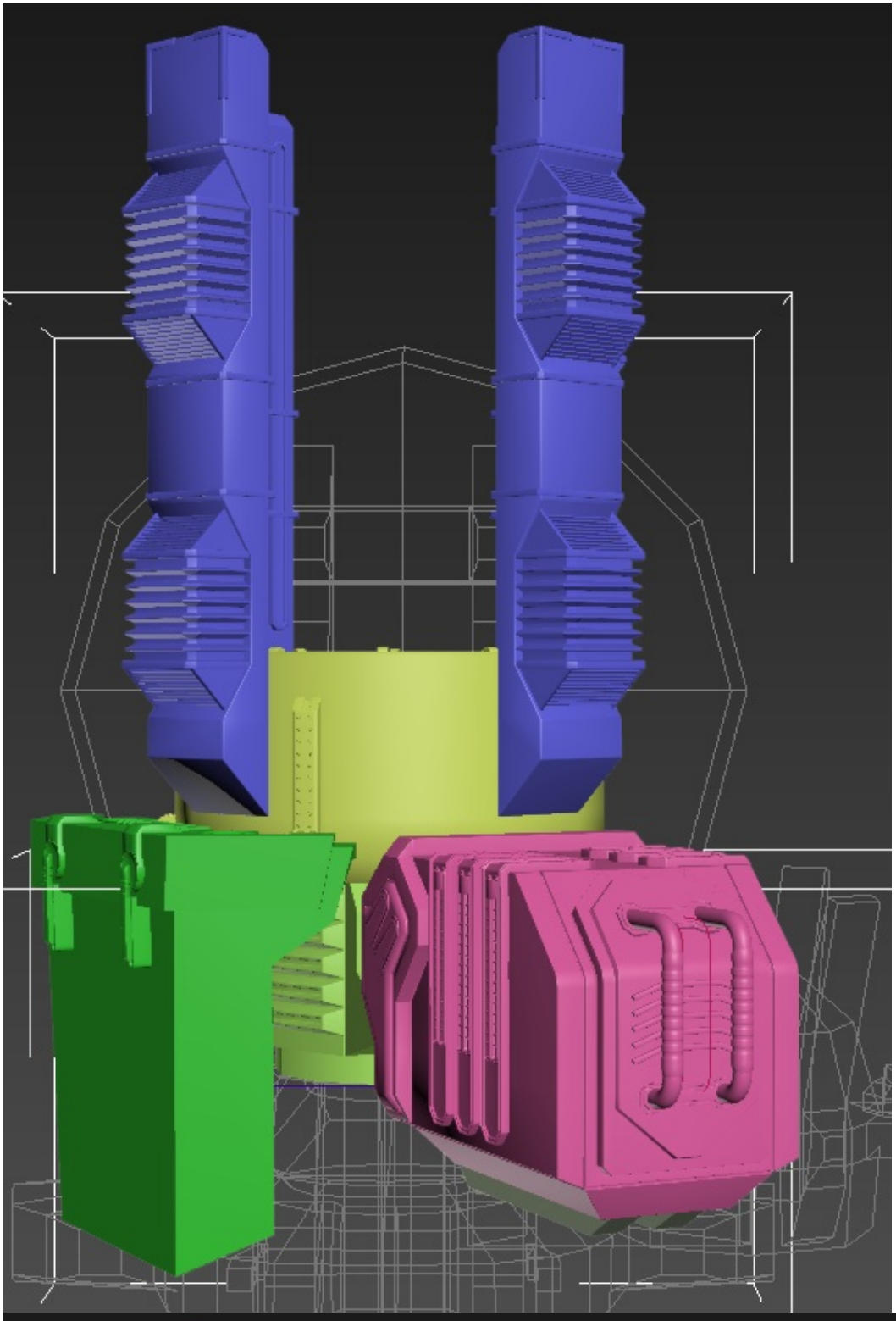


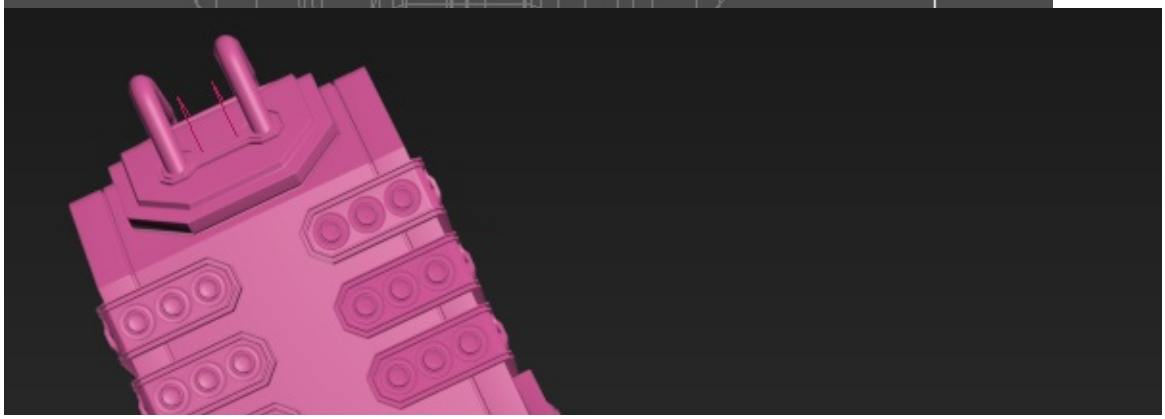


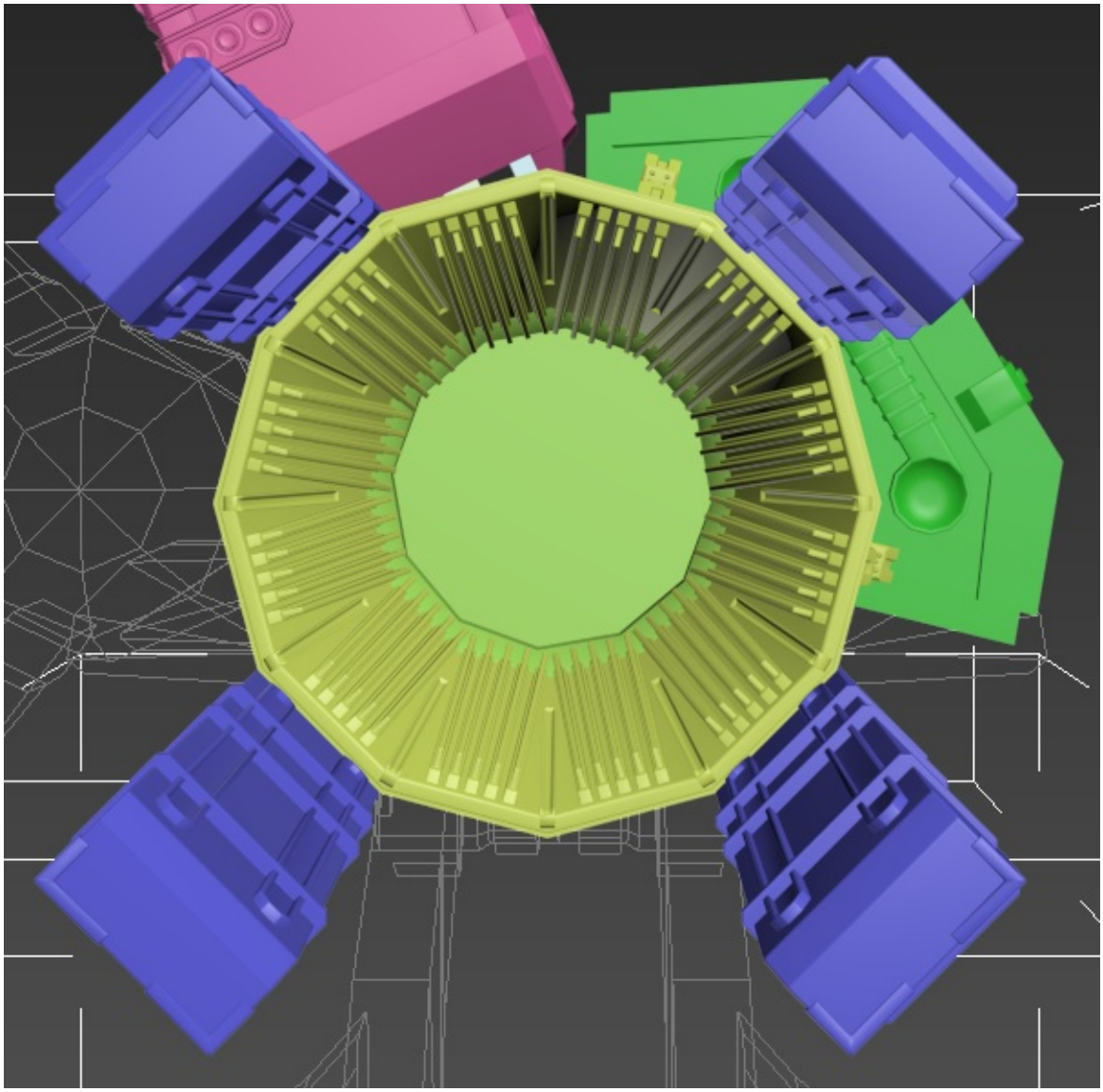
Notice how the horizontal/vertical position of object details is not affected by the distance of the detail from the camera.

Perspective Camera

This type of camera is what we typically associate with a "normal" vision of the world. In contrast to above, it looks something like this:







Notice how the same details in the objects now deviate from the center of the screen based on their distance from the camera. We call this **perspective correction** and it is the basis of a perspective camera. This is the default camera used in the UDK.

Camera Control

In the UDK, a camera is composed of three properties:

- Position
- Rotation
- Field of View

Position

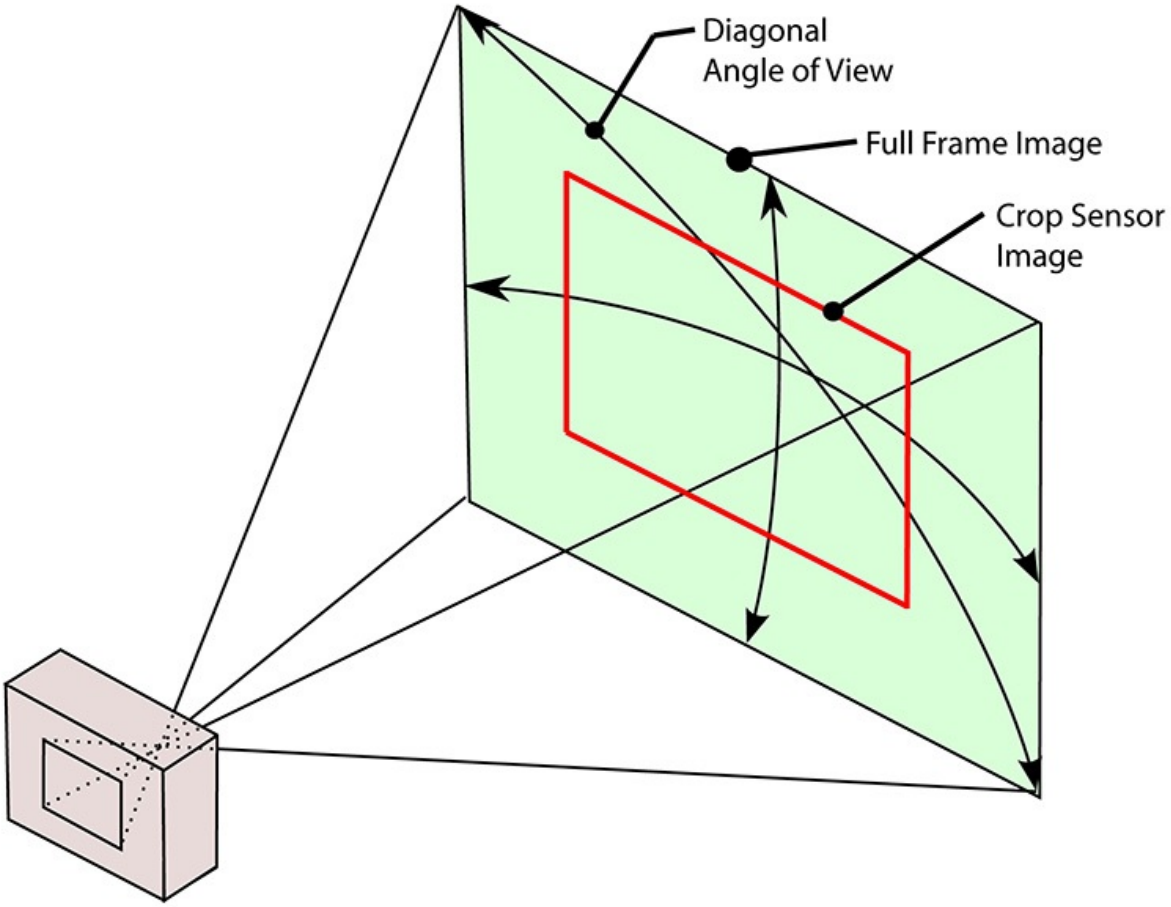
This defines the location in the world from which the view should be constructed. This position will be the location of the viewer.

Rotation

This defines the rotation from which the world is viewed.

Field of View

In a perspective camera (the default camera), this is a measure of the maximum angle that a light ray may deviate from the center directional line of the camera and still be visible. This may either be a measure of the horizontal deviation or the vertical deviation. Typically, the field of view is the angle of the entire field so the maximum deviation is half of the field of view. This is better explained using a diagram:



In the UDK, the field of view defaults to somewhere between 60 and 90 degrees. For our purposes, we can typically ignore this property and focus on position and rotation.

UDK Implementation

In the UDK, the simplest way to control the camera is to override the function **CalcCamera** on a derived UTPawn class.

CalcCamera

This function is found on all actors but made particularly useful in UTPawn. To make use of this, your controller must also derive from UTPlayerController **and** must be possessing the pawn in question.

```
 function bool CalcCamera( float fDeltaTime, out vector out_CamLoc, out rotator out_CamRot, out float out_FOV )
```

Let's look at the signature of this function:

- Return type: Bool

    - This function should return true if its calculations should control the camera
- float fDeltaTime
    - The amount of time in seconds that has passed since the last frame/tick
- out vector out_CamLoc
    - An out vector parameter that should hold the final position of the camera in the world
- out rotator out_CamRot
    - An out rotator parameter that should hold the final rotation of the camera in the world
- out float out_FOV
    - An out float parameter that should hold the final field of view of the camera in standard angles (ie 0-360). This defaults to a valid value

For example, the following UTPawn derived class positions the camera at the location (0, 0, 500) and rotates it to pitch upwards by 45 degrees:

```
class ExamplePawn extends UTPawn;

var rotator CamRot;
var vector CamLoc;

function bool CalcCamera( float fDeltaTime, out vector out_CamLoc, out rotator out_CamRot
{
    out_CamLoc = CamLoc;
    out_CamRot = CamRot;
    return True;
}

DefaultProperties
{
    CamLoc=(X=0, Y=0, Z=500)
    CamRot=(Pitch=8192, Roll=0, Yaw=0)
}
```

Lab 6 - Camera Manipulation

Summary

In this lab, you will implement a pawn and a player controller that will allow you to control the position and direction of the camera.

Learning Outcomes

Files to Implement

- L6Game.uc
- L6Pawn.uc
- L6PlayerController.uc
- L6-Map.udk
- Update: DefaultInput.ini

Submission Requirements

Show your instructor that you can manipulate the camera's three rotational components with keys on the keyboard individually and that you can manipulate the camera's three positional components with keys on the keyboard individually. Finally show that you can lock and unlock the camera.

Instructions

- Create a script package called Lab6 and implement the following classes in it

L6Game

- Create a game type called **L6Game** that extends *UTGame*
  - As usual with game types that extends UTGame, be careful to set the MapPrefixes[0] entry appropriately
  - For this lab, set the MapPrefixes[0] entry in the *defaultproperties* to **"L6"**
- We will need to link up our player controller and pawn classes to this game type

    - Set the **PlayerControllerClass** property to the class *L6PlayerController*
    - Set the **DefaultPawnClass** property to the class *L6Pawn*

L6-Map.udk

- Create the map L6-Map.udk and set its gametype to **L6Game**. You can only do this once you are able to successfully compile the Lab6 package including the **L6Game** class.

L6Pawn

- This pawn class will have code to explicitly control the camera. This class should extend *UTPawn*
- Add the following member variables:
    - bool L6CamLocked
        - Should default to false
    - vector L6CamOffset
        - Should default to (X=200, Y=0, Z=100)
    - rotator L6CamRot
- Override the function: **simulated function bool CalcCamera(float fDeltaTime, out vector out_CamLoc, out rotator out_CamRot, out float out_FOV)** and implement it as follows:
    - Set *out_camLoc* to be the sum of this pawn's location and L6CamOffset
    - Branch: If L6CamLocked is true...
        - Set *out_CamRot* to be a rotator casted from a vector that is the result of subtracting *out_CamLoc* from this pawn's location
    - If L6CamLocked is false...
        - Set *out_CamRot* to be equal to *L6CamRot*
    - Return true

L6PlayerController

- This player controller class will update its L6Pawn's camera explicitly
- This class should extend *UTPlayerController*
- Initialize the inherited member variable *bBehindView* to true
- Implement the exec function **ToggleCamLocked**
    - This function should check if this controller is attached to an **L6Pawn**
        - If it is, it should invert the value of the pawn's L6CamLocked variable
- Implement the following exec functions:

```
* IncCamOffsetX
* DecCamOffsetX
* IncCamOffsetY
* DecCamOffsetY
* IncCamOffsetZ
* DecCamOffsetZ
```

    - These functions should check if the controller is attached to an **L6Pawn**
        - If it is, they should update the pawn's L6CamOffset variable
        - The *X, *Y, *Z functions should update the X, Y, or Z property of L6CamOffset respectively
        - The Inc* functions should increment the property by an amount (eg 100)
        - The Dec* functions should decrement the property by an amount (eg -100)
- Implement the following exec functions:

```
* IncCamRotPitch
* DecCamRotPitch
* IncCamRotYaw
* DecCamRotYaw
* IncCamRotRoll
* DecCamRotRoll
```

    - These functions should act similarly to the offset functions above in concept, except that they should update the L6CamRot property instead
    - The properties in a rotator are Pitch, Yaw, and Roll. The functions above should update them respectively.
    - The Inc functions should increase their property by an amount eg 2500
    - The Dec functions should decrease their property by an amount eg -2500

Update DefaultInput.ini

- Add a keyboard input for each of the exec functions implemented in L6PlayerController
    - Recommended:
        - Use the numpad keys - 1, 2, 4, 5, 7, 8 for camoffset functions
        - Use insert, delete, home, end, pageup, pagedown for camrot functions
        - Use numpad key 9 for the ToggleCamLock function

Lecture Topics

Day 1

- Interface classes
- Objects, Self
- PlayerInput
- PlayerController, Rotation, Firing

Day2

- Coding Editor Resources, Canvas, Texture2D
- Projection (needs notes)
- Input Binding Advanced

Interface Classes

An interface in UnrealScript is a lot like a class except that it does not define the functionality that it declares. This is best illustrated with an example of an interface named **Breakable**:

```
Interface Breakable;

function float Add(float A, float B);
function PrintString(string S);
```

Notice a couple of things:

1. The keyword used to define an Interface is **Interface**
2. The functions defined have no bodies

An interface in UnrealScript works a lot like how an interface works in Java. It is useful for classifying objects into categories. In order to make use of an interface, a class must **implement** it. This is done via the **implements** keyword. For example, here's an Actor subclass **SampleActor** that implements the above interface.

```
class SampleActor extends Actor implements(Breakable);

function float Add(float A, float B)
{
    return A + B;
}

function PrintString(string S)
{
    `Log(S);
}

Defaultproperties
{
}
```

The implements keyword indicates that this Actor will fulfil the contract set out by the **Breakable** interface, that's to say that it **provides implementations for all of the functions defined in all of the interfaces that it implements**. In this case, the Actor implements only a single interface, *Breakable*, so it needs to provide implementations for its functions, as seen above.

Multiple Interface Implementation

It is possible for a class to implement multiple interfaces. This is done by declaring all of the interfaces that the actor implements in a comma-separated list in the parentheses beside the **implements** keyword. For example, given the following additional interface definition:

```
Interface Malleable;

function Bend(float angle);
```

We can redefine our SampleActor above as follows:

```
class SampleActor extends Actor implements(Breakable, Malleable);

function float Add(float A, float B)
{
    return A + B;
}

function PrintString(string S)
{
    `Log(S);
}

function Bend(float angle)
{
    `Log("I now bend by "$angle);
}

Defaultproperties
{
}
```

In this way, it is possible for a class to implement any number of interfaces. It is required that a class that implements an interface must provide implementations for all of the functionality that the interface declares.

Interface Use Cases

There are two situations where interfaces are useful and they are typically inter-related.

Iterating Over Actors Implementing An Interface

This is one of the most common use cases for an interface as it can provide a way for us to organize our Actor types without using the strict rules of single parent inheritance. Let's say for example that you would like to iterate over all actors in the scene that implement the **Malleable** interface. You may write code like this:

```
// Assume that this is inside a function
local Actor iterActor;

foreach AllActors(class'Actor', iterActor, class'Malleable')
{
    // Do stuff with iterActor ...
}
```

Note that we've passed a third argument to AllActors. The third argument is optional but we are using it in this case. It tells the iterator that we are only interested in Actors that **implement the Malleable interface**.

This is all fine and good but we cannot call any of the interface's functions on iterActor in its current state. This is because, as far as UnrealScript is concerned, iterActor is of type Actor and type Actor does **not** implement Malleable. But, we know that iterActor *does* implement Malleable because we've definitely requested this from AllActors! To solve this issue, we must cast.

Casting To An Interface

An interface is like a class and so we typically have to cast an object *to it* before we can use its functionality. For example, we can update the above example as follows:

```
// Assume that this is inside a function
local Actor iterActor;
local Malleable malleable;

foreach AllActors(class'Actor', iterActor, class'Malleable')
{
    malleable = Malleable(iterActor);

    // Always test to ensure that the cast succeeded.
    // Although in this case it will surely succeed...
    if (malleable != None)
    {
        malleable.Bend(62);
    }
}
```

In order to use the functions declared in Malleable, we had to cast our Actor variable to a Malleable variable, just like you do with classes; the underlying object instance remains the same, the syntactic symbol we used to refer to it has simply changed.

Interface Inheritance

It is possible for Interfaces to extend each other but this is not recommended for a number of reasons.

1. The way that UnrealScript is set up, if a class implements two interfaces that share a common ancestor, that class' *virtual function table* (the underlying mechanism by which virtual functions are achieved in UnrealScript) will be corrupt
2. Conceptually, interfaces that extend each other are less useful. Normal class-style inheritance can be used to achieve this. The power of interfaces comes from their ability to cut horizontally through the inheritance tree, so it is not recommended that they in turn create their own inheritance hierarchy.

Objects, Self

While Actors are useful for a lot of things, there are instances where much of their functionality is not required or may possibly not even make sense. For this, we have objects.

Object Class

Objects are classes that derive from Object or from one of its non-actor subclasses. They can be used to hold data, functionality, exec functions, etc...

Like Actors, Objects are also classes, meaning that they have accesses to much of the same base-level UnrealScript functionality, such as:

- Inheritance
- Member variables
- Functions
- Function overriding
- Interface implementation
- Exec functions
- Default properties

But additionally, Objects have access to a very important keyword: **within**

Constructing Objects

Consider the following Object class named **SpecialObject**:

```
class SpecialObject extends Object;

var int AnInt;
var float AFloat;

function increment()
{
    AnInt++;
    AFloat++;
}

Defaultproperties
{
    AnInt=6
    AFloat=7
}
```

This is a basic Object class that contains two members and a function. We can construct and store it as follows. Consider the following Actor:

```
class SpecialActor extends Actor dependson(SpecialObject);

var SpecialObject MySpecialObject;

function PostBeginPlay()
{
    MySpecialObject = new class'SpecialObject';
}

Defaultproperties
{
}
```

Notice a few interesting things:

- **dependson** keyword
  - This keyword forces the compiler to compile the given class/struct **first** before compiling this class. This is important when multiple classes make use of the same classes/structs. Typically this is not required but when dealing with objects and/or structs, it is recommended.
- MySpecialObject variable
  - Notice that our declaration of a variable of type MySpecialObject is very similar to how you would declare any other variable
- Notice how we populate the MySpecialObject variable in the PostBeginPlay event (called after this actor is constructed)
  - To construct an Object we use the **new** keyword. This keyword is **not** used to construct actors! Only objects are constructed this way!
  - The new keyword has a special syntax but it is straight forward
    - The simple version we used above can be prototyped as follows: `ObjectVar = new <class'InClass'>;`
    - The full prototype is as follows: `ObjectVar = new[(InOuter, InName, InFlags)] <class'InClass'>[(InTemplate)];`
      - We will ignore inName, inFlags, and inTemplate for now. We will look at InOuter in the next section

Outer Objects, Within, and Self

Objects can be instantiated with what is known as an *Outer* object encapsulating them.

If the instantiated Object class also declares the keyword **within**, that Object will have access to **all of the members of the containing object declared in the Within specifier**. The *within* specifier will also make it **impossible** to instantiate this Object class without passing in an instance of the required outer Object

This is best described with an example. Consider the following two classes:

```
class ContainerActor extends Actor;

var ContainedObject contained;
var int num

function PostBeginPlay()
{
    contained = new(self) class'ContainedObject';

    contained.inc();
    contained.print();
}

Defaultproperties
{
    num=6
}
```

```
class ContainedObject extends Object within ContainerActor;

function inc()
{
    num++;
    // This could also be written Outer.num++;
}

function print()
{
    `Log(num);

    // This could also be written as: `Log(Outer.num);
}

Defaultproperties
{
}
```

ContainerActor

First look at the actor. Notice that we have used the new keyword slightly differently this time. In parentheses, we are passing **self** in as a parameter immediately after the new statement. This is how we pass in an outer object. In this case, self is a special keyword that implies *the current object*. The net result of this statement is that an object of class **ContainedObject** will be instantiated and it will have the current Actor attached as an Outer for it.

The new keyword also accepts a few more parameters but they are not important; as far as we are concerned, these are the only two relevant uses of the new keyword.

ContainedObject

Let's look at *ContainedObject*. By specifying the **within** keyword on class declaration, we are giving this object access to all of the members of a *ContainerActor* object. It also means that *ContainedObject* can never be instantiated without being passed a *ContainerActor* as an Outer.

Looking at the functions, notice that we liberally make use of the member variables of the Outer *ContainerActor*. For example, in *inc* and in *print*, we update and read the num variable, respectively.

Note that we could have also accessed that variable through the builtin `Outer` variable. This variable gives you a way to get a reference to the Outer object as an Object (so that you could cast it if you wanted to, or pass it to another function, etc...). The type of the **Outer** variable is the same as what is declared in your **within** specifier. If your object *does not* have a within specifier, Outer's type will be **Object**.

PlayerInput

A **PlayerInput** class is used to encapsulate that part of a player controller that receives input from the input devices currently attached. Typically, it contains the following:

- Variables to track certain input values (ie whether player should be ducking, how much the mouse has moved in the last tick, etc...)
- Exec functions called directly from DefaultInput.ini that update specific values as required
- An override of the function PlayerInput, which returns nothing and accepts **float deltatime**
    - This function should perform any time-related updates to the input object or reset any variables that need resetting as required

Association With PlayerController

PlayerController has code for automatically constructing and storing the associated PlayerInput object. This is done via two variables:

1. `var class<PlayerInput> InputClass`
    - This variable holds the class of PlayerInput that this PlayerController should construct. This is defined in the defaultproperties of PlayerController classes
    - UTPlayerController assigns the class **UTPlayerInput** to this
2. `var PlayerInput PlayerInput`
    - A wonderfully named variable that stores this player controller's current player input object. Any objects wishing to gain access to the player controller's player input object should access it via this variable

Outer Actor

PlayerInput is **not** an Actor. However, it is instantiated by the PlayerController Actor. Upon instantiation, that player controller instance is set up as an outer for this object. So, this object has access to all of the members of the containing player controller, at least all of the members defined in the class *PlayerController*, not any subclass of it; to gain access to members defined in more derived classes, you should cast the **Outer** variable present on the PlayerInput object.

This can be used to gain access to the PlayerController's Pawn (through the Pawn member) or HUD (through the myHUD member).

Creating Custom PlayerInput Class

If creating a new player input class, it is recommended that you extend an existing one. One such useful PlayerInput class is **UTPlayerInput** (**PlayerInput** itself defines a lot of functionality, use whichever is suitable to your needs). This will give you access to the following variables that are maintained by that class's *PlayerInput()* function:

Axial Variables

- aMouseX, aMouseY
  - How many *pixels* the mouse moved in the last tick. This number may be too large to be useful and may need to be scaled.
- aForward, aStrafe
  - How many *units* the player wants to move forward or left/right. The exact meaning of the *units* in this case is unclear. Take a look at DefaultInput.ini to see how these are updated and decide for yourself how you would use them.
- aTurn, aUp
  - How many *units* the player wants to turn their view left/right or up/down. Similar issue to above.

Take a look at PlayerInput.uc for many more of these and look at DefaultInput.ini to see how they are bound to various input devices.

Using Custom PlayerInput Class

Using a custom-defined PlayerInput class is as simple as setting the custom player input class as the default value of a player controller's **InputClass** variable.

PlayerController Rotation, Firing

The PlayerController is responsible for ensuring that the camera is up to date with where the Player *should* be looking as well as starting and stopping the fire cycle.

UpdateRotation

This is the function that is used to keep the rotation of the camera up to date. We know that UpdateRotation keeps the camera upto date because its code updates the player controller's rotation based on the input values in aTurn and aLookUp. This is how UpdateRotation looks in PlayerController:

```
function UpdateRotation( float DeltaTime )
{
    local Rotator    DeltaRot, newRotation, ViewRotation;

    ViewRotation = Rotation;
    if (Pawn!=none)
    {
        Pawn.SetDesiredRotation(ViewRotation);
    }

    // !!!!!!!!!!!!!!!!!!!!!!!! GAM537/DPS937 Note: These are the lines that we care ab
    // Calculate Delta to be applied on ViewRotation
    DeltaRot.Yaw    = PlayerInput.aTurn;
    DeltaRot.Pitch    = PlayerInput.aLookUp;
    // !!!!!!!!!!!!!!!!!!!!!!!! GAM537/DPS937 Note

    ProcessViewRotation( DeltaTime, ViewRotation, DeltaRot );
    SetRotation(ViewRotation);

    ViewShake( deltaTime );

    NewRotation = ViewRotation;
    NewRotation.Roll = Rotation.Roll;

    if ( Pawn != None )
        Pawn.FaceRotation(NewRotation, deltatime);
}
```

Notice that UpdateRotation is also responsible for taking into account any viewshake (such as from weapon fire) and for making the Pawn face in the same direction as the player controller. Finding this information for yourself involves a little bit of research, reasoning, and intuition based on the names of these functions.

This means that if we wanted to handle view rotation our own way or to subvert it entirely, this function would be the place for us to place our code.

Following the call-chain

If we follow the call-chain for this function, we find that it looks like the following:

- PlayerTick
- PlayerMove
- UpdateRotation

In most programming languages, following the chain of calls would be simple. In UnrealScript, this is complicated by the fact that Actors could have states (see the notes on States). Notice that inside PlayerController.uc, there are **8** versions of the PlayerMove function, not counting the initial prototype. If we look at each of them in turn, we notice that almost all of them end up calling UpdateRotation within their code.

StartFire

This is the function responsible for starting a fire cycle. It accepts a byte (an unsigned integer of 1 byte-length) that indicates which firemode the weapon should fire (for example, rocket launcher has primary fire and alt fire). Notice that in DefaultInput.ini, this function is directly called from the key binding system to indicate that the user has pressed down on the fire button.

Here's the relevant section in DefaultInput.ini:

```
; Removed BaseInput.ini aliases
.Bindings=(Name="GBA_Fire",Command="StartFire | OnRelease StopFire")
```

Now that we know which function is responsible for starting a fire cycle, we know what part of the code to override if we wanted to change what the player does when the fire button is pressed.

For a breakdown of the line above, please see the notes on Input Binding Advanced.

Coding Editor Resources, Canvas & Texture2D

Similarly to how we can specify a class in UnrealScript by using the **class'SomeClass'** syntax, we can specify any other type of resource that can be imported into a upk through the editor. This is done through the following syntax: `<ObjectType>'FullyQualifiedName'`

In this case, <*ObjectType*> is any kind of UDK editor resource, for example a Texture2D. FullyQualifiedName is whatever is the fully qualified name of this resource, eg SomePackage.SomeGroup.Something or SomePackage.Something if it is not in a group. If you use the "Copy fully qualified name to clipboard" command from the editor right click menu, it will include the object type.

One of the most common cases where this will be useful is when passing textures or materials to a function but can also commonly apply to meshes and static meshes.

For example, to specify a particular Texture2D, we can use the following syntax: `Texture2D'Package.Group.TextureName'`

Texture2D

This is the UDK's version of an image, a so-called Texture2D; the 2D specification is needed because it is possible to have 1D and 3D textures as well. One of the most common reasons to use a Texture2D in code is when dealing with Canvas. A common use-case is placing an image on the screen. To do this, Canvas has the function DrawTexture with the prototype: `Canvas.DrawTexture(Texture2D textureToUse, float scale);`

- Texture2D textureToUse
  - This is the texture that should be placed on the screen. It is alpha-blended, meaning that the alpha value at each pixel is used to determine the opacity of that pixel when blending it with what is already on the screen (alpha = 255 means complete replacement with this pixel, alpha = 0 means this pixel will have no effect)
- float scale
  - This is the scale of the texture. For example, if we draw a 512x512 texture on a 1024x1024 screen at scale 1.0, it will occupy 1/4 of the screen. However, if we draw it at scale 2.0, it will occupy the entire screen

Here's an example of how we would use DrawTexture. For this example, assume that the texture *AwesomePackage.AwesomeGroup.AwesomeTexture* exists and is a valid Texture2D:

```
Canvas.SetPos(0, 0); // Set position to top-left of screen
Canvas.SetDrawColor(255, 255, 255); // Set the draw color to white

// Draw the texture at scale 1
Canvas.DrawTexture(Texture2D'AwesomePackage.AwesomeGroup.AwesomeTexture', 1.0);
```

But what part of the screen will this texture occupy? For that, we will need to get some of the properties of a Texture2D.

Working With Texture2D in Code

Like other data types, it is possible to declare variables that can hold instances of resources, such as a Texture2D in this case, and to get the properties of those resources. For example, consider the following function:

```
function AcceptTex(Texture2D myTex)
{
    `Log("Texture size is "$myTex.SizeX$"x"$myTex.SizeY);
}

function CallingFunc()
{
    local Texture2D tex;

    tex = Texture2D'AwesomePackage.AwesomeGroup.AwesomeTexture';
    AcceptTex(tex);
    // Could also have called it as:
    // AcceptTex(Texture2D'AwesomePackage.AwesomeGroup.AwesomeTexture');
}
```

Notice some things about the above:

- We use Texture2D as the data type of a function parameter and of a local variable
- We assign into the variable by using a Texture2D specifier, like we would with an UnrealScript class
- We pass the variable into the function like it was a literal specifier
- **We query the size of the texture by retrieving its SizeX and SizeY properties**

To find what properties a Texture2D has, we open Texture2D.uc (package Engine, or use Visual Studio Ctrl + , then Texture2D **Enter**). Here's a collection of interesting properties:

```
/** The width of the texture.                                                */
var const int SizeX;

/** The height of the texture.                                                */
var const int SizeY;

/** The original width of the texture source art we imported from.            */
var const int OriginalSizeX;

/** The original height of the texture source art we imported from.           */
var const int OriginalSizeY;
```

Most importantly, we can get the size of the texture this way. For novelty, we also have access to the size of the original file that was used to create this texture.

Like Texture2D, there is an UnrealScript file for every resource type that can be imported into the editor. Whenever you want to know what you have access to, simply open the relevant uc file.

Determining Texture Screen Size

Because we have access to the size of the texture, we can use this information in combination with the size of the screen to find how much of the screen the texture will occupy. For example, assume that we have the following HUD class:

```
class SomeHUD extends HUD;

function DrawHUD()
{
    local Texture2D tex;

    tex = Texture2D'Package.Group.GreatTexture';

    Canvas.SetPos(0,0);
    Canvas.SetDrawColor(255, 255, 255);
    Canvas.DrawTexture(tex, 1.0);
}

Defaultproperties
{
}
```

In this example, how much of the *horizontal* screen did the texture occupy? To find out, we can use its SizeX as follows: `Texture.SizeX / SizeX -> proportion of horizontal screen occupied`

So if we wanted to codify this, we would end up with a function like this (assume this function is in the above class):

```
function PrintProportion(Texture2D tex)
{
    local xProp, yProp;
    xProp = tex.SizeX / SizeX;
    yProp = tex.SizeY / SizeY;
    `Log("X proportion is: "$xProp$", Y proportion is: "$yProp);
}
```

Input Binding Advanced

We will look at more advanced uses of input binding. For the complete reference of the capabilities of the UDK input system, please see the UDN KeyBinds Reference.

Consider the following line from **DefaultInput.ini**:

```
; Removed BaseInput.ini aliases
.Bindings=(Name="GBA_Fire",Command="StartFire | OnRelease StopFire")
```

Let's note some things here:

- The name in the binding is *GBA_Fire*. If we consult our list of udk key binds, we find that GBA_Fire is not one of them. In this case, what is happening is called **input aliasing**. This means that the command in this binding is being associated with the name GBA_Fire and GBA_Fire could later on in the document be issued as a command. More on this in a second.
- The command contains a pipe and an OnRelease modifier
  - The net effect is that when GBA_Fire is triggered, StartFire is called once, then when the key is released, StopFire is called once
  - It is possible to call many commands in a key bind by repeated use of the **|** operator

Input Aliasing

This is best illustrated with an example. Consider the following bind:

```
.Bindings=(Name="ExampleBind",Command="AddBots 3")
```

We bound the command *AddBots 3*, which spawns 3 bots, to the name ExampleBind. Again, since ExampleBind is not a mappable input, this becomes an input alias. This allows us to do the following:

```
.Bindings=(Name="H",Command="ExampleBind")
.Bindings=(Name="X",Command="ExampleBind")
```

We've bound the keys H and X to perform our ExampleBind on keypress. Now, if we wanted to change what ExampleBind did, we would have to do it in only one place, while continuing to have multiple inputs all mapping to our ExampleBind.

Lab 6 - Advanced Canvas, Basic Picking

Summary

You will implement a more advanced hud utilizing screen textures as well as basic cursor selection.

Learning Outcomes

- Ability to use Canvas in a more advanced way
- Familiarity with basic picking techniques

Files to Implement

- L7Game.uc
- L7HUD.uc
- L7Pawn.uc
- L7Player.uc
- L7PlayerInput.uc
- L7Selectable.uc
- L7-Map.udk

Submission Requirements

Show your completed lab to your instructor(s). Your end result should look something like this:

Neutral (nothing selected, cursor control inactive)

Selecting (cursor control active)

Neutral, After Selecting (pawn selected, cursor control inactive)

Instructions

Download the file L7Content.upk and place it in UDKGame/Content. You may add it to your repository.

Lab7 Package

Create and set up a new development package, name it Lab7. The following code should reside in the Lab7 package.

L7Game

- Create the class L7Game extending **UTGame**
- Set up the default properties as follows:
  - Set the hud type for this game to be the class **L7HUD**
  - Set bUseClassicHUD to true
  - Set element 0 for MapPrefixes array to "L7"
  - Set the default player controller class to **L7Player**
  - Set the default pawn class to the class **L7Pawn** (variable name: *DefaultPawnClass*)

In order for this class to compile successfully at this point in the lab, you will need to create stub versions of the referenced classes (L7Pawn will extend UTPawn).

L7-Map.udk

After successfully compiling the above (create the required stubs), create the map **L7-Map.udk** and save it in UDKGame/Content/Maps.

Add a few player starts to the map before saving it, this will make bots spawn more evenly. Make sure that you rebuild map paths after you do this.

L7Selectable

- Create the interface **L7Selectable**
- This interface doesn't necessarily need any functions but you may have it prototype a function named GetLocation3D which returns a vector and accepts nothing

L7Pawn

- Create the class **L7Pawn**, it should extend **UTPawn** and implement **L7Selectable**
- If you wrote **L7Selectable** to have the function *GetLocation3D* then implement that function and have it return the actor's current location
    - Otherwise, you're done for this class

L7PlayerInput

- Create the class L7PlayerInput which extends UTPlayerInput
- This class should have 3 member variables:
    - Variable of type **IntPoint** named *MousePosition*
    - Variable of type **bool** named *bControllingCursor*
    - Variable of type **bool** named *bAttemptSelect*
- Create the exec function ToggleControllingCursor
    - It should invert the current value of bControllingCursor ie set it to false if it is true and set it to true if it is false
- Create the exec function StartAttemptSelect
    - If cursor control (bControllingCursor) is currently true, it should set bAttemptSelect to true
    - Otherwise, it should do nothing
- Override the function PlayerInput which accepts a float deltatime, and returns nothing
    - This function should first call its super version
    - Then, if cursor control is active:
        - It should update the values in the MousePosition variable with the values in aMouseX and aMouseY
            - Take care that MousePosition's members do not exceed the size of the

screen or go below 0
    - You can use the **Clamp** function for this (look it up in Object.uc)
  - Take care that aMouseY is subtracted as opposed to added to MousePosition.Y as the values are the inverse of how you would expect a mouse to behave
  - The values in aMouseX and aMouseY are likely too large, multiply them by a suitable factor before addition, for example 0.1

L7Player

- Create the class **L7Player**, it should extend **UTPlayerController**
- In default properties, set the **InputClass** member to *class'L7PlayerInput'*
- Override the UpdateRotation function, which accepts a float deltatime, and returns nothing
  - This function should check if cursor control is currently active
    - If it is **NOT**, it should call the super version of this function
    - Otherwise, it should do nothing
- Override the **exec** function StartFire, which accepts an **optional** byte FireModeNum, and returns nothing
  - This function should check if cursor control currently active
    - If it is **NOT**, it should call the super version of this function
    - Otherwise it should do nothing

L7HUD

- Have this class extend HUD
- Add a class member typed *Actor* and named **Selected**
- Create the function DrawCenteredTextureOnCanvas, returning a **Vector2D**, and accepting two arguments:
  - A **Texture2D** named *tex*
  - A **float** named *screenScale*
- This function will:
  - Store the current canvas position in a local variable
  - Update the current canvas position such that the texture that was passed in is drawn centered around the current coordinates (Canvas.DrawTexture interprets the current screen position as the top left corner of the texture)
  - Use the function Canvas.DrawTexture which accepts a Texture2D and a scale, calculating a scale such that:
    - The texture occupies the proportion of **vertical** screen space passed in through **screenScale**

        - You will find the scale by finding how many pixels the texture *should* take by multiplying the passed in scale by the vertical height of the screen (the *SizeX* member in **HUD**) then dividing that result by the vertical size of the texture
    - Restore the current canvas position before returning
    - Return the size of the drawn texture in on-screen pixels, stored in a Vector2D (Vector2D objects have X and Y members but no Z)
- Create the function DrawHUD, returning nothing and accepting nothing
    - This function should iterate over all **L7Selectable** interface Actors in the scene, drawing a selectable square texture on the screen position of each of these actors
        - The screen scale of the selection square should be 0.1
        - If cursor control is currently off, the selection rectangles should be drawn white
            - Otherwise, they should be drawn red or pink
        - Use the DrawCenteredTextureOnCanvas function defined above to do this
        - Use the Canvas projection function to find the on-screen position of an actor
            - While you may use Actor's Location member to find the current 3D location of actors in question, the slightly *cleaner* approach would be to use the **GetLocation3D** function we defined above (if you defined it) since we know that every actor that we are iterating over implements **L7Selectable**
            - You can cast an Actor to an interface like you can cast anything else
        - Your function should not draw a selectable rect for the player's pawn
        - Your function should not draw a selectable rect for actors whose screen coords x or y members are less than 0 or greater than the screen's SizeX and SizeY, respectively
        - Your function should not draw a selectable rect for actors whose screen coords z member is less than or equal to 0
        - The currently selected actor should have the texture **L7Content.Selected** drawn on it
            - All other selectable actors should have the texture **L7Content.Selectable**
    - This function should draw a targeting reticle at the current mouse coords
        - The screen scale of the targeting reticle should 0.1
        - If the reticle's center is within the bounds of a selectable rectangle, the reticle should use the **L7Content.ReticlePicking** texture
        - Otherwise, the **L7Content.Reticle** texture should be used
        - If cursor control is currently off, the reticle should be white
            - Otherwise, it should be green
    - This function should update the selection state if the player input's bAttemptSelect variable is true
        - If the player input's bAttemptSelect variable is true, this class should attempt to select something then after the attempt, bAttemptSelect should be set to false

    - To attempt selection, the list of actors implementing the L7Selectable interface should be iterated over
        - The player's own pawn should not be considered for selection
    - The **closest** selectable actor to the screen whose selectable rectangle overlaps the center of the reticle should be set as selected
        - This is indicated by setting the `Selected` member variable equal to it
    - Finding the closest actor to the screen can be found by finding the actor with the smallest Z member when its location is projected using canvas
    - If no suitable actors are found, Selected should be set to None

DefaultInput.ini

- Bind the **Z** key to the command **ToggleControllingCursor**
- Bind the **left mouse button** to the commands **GBA_Fire** and **StartAttemptSelect**

Collision, Inventory, Basic Weapons

Lecture Topics

- Components
  - Role of a component
  - Components list on Actor
  - Defining component objects in default properties
  - StaticMeshComponent
  - SkeletalMeshComponent
  - Other components
- Collision
  - bCollideActors
  - bCollideWorld
  - Events
    - Touch
    - Untouch
    - Bump
- InventoryManager
  - AddInventory
  - RemoveFromInventory
  - FindInventoryType
  - CreateInventory
- Inventory
- Weapon/UTWeapon
  - Display
    - SkeletalMeshComponent
      - SkeletalMesh
      - AnimSets
      - Animations <- AnimNodeSequence
    - AnimNodeSequence object
    - Mesh
    - WeaponIdleAnims[]
  - Basics
    - FireInterval[]
    - WeaponProjectiles[]
    - WeaponFireTypes[]
    - WeaponFireAnim[]
  - Ammunition
    - MaxAmmoCount

        - AmmoCount
        - ShotCost[]
    - Logic
        - FireAmmunition
        - ConsumeAmmo
        - AddAmmo
        - Instigator
        - GetPhysicalFireStartLoc
        - GetAdjustedAim
        - Pawn.GetAdjustedAimFor
        - Pawn.GetViewRotation
        - CurrentFiringMode
        - FiringStatesArray[]
- Projectile
    - Behaviour Properties
        - LifeSpan
        - AccelRate <- float
        - Speed
        - MaxSpeed
        - Damage
        - DamageRadius
        - MomentumTransfer
        - bRotationFollowsVelocity
        - MyDamageType
        - bCollideActors
        - bCollideWorld
    - Display Properties
        - Can add any component as display, eg static mesh
        - Particle Systems
            - ProjFlightTemplate
            - ProjExplosionTemplate
    - Functions
        - Init(Vector direction)
        - Explode
        - ProjectileHurtRadius
    - Events
        - Touch/ProcessTouch
        - HitWall
    - Tracking
        - Hint at tracking

- UTDamageType
  - Holds information, never spawned
  - Useful static functions
    - DeathMessage
    - SuicideMessage

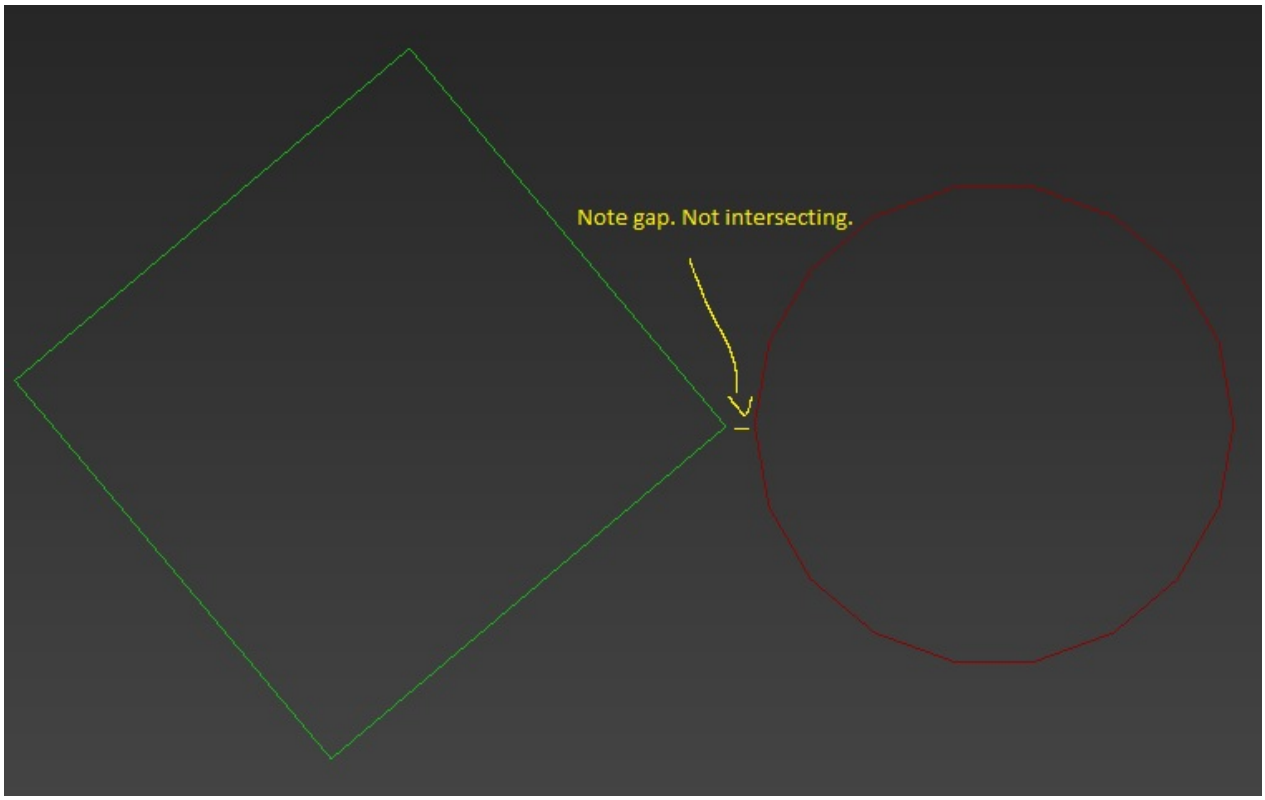
Collision Basics

Collision in game engines is divided into two general steps: collision detection and collision resolution.

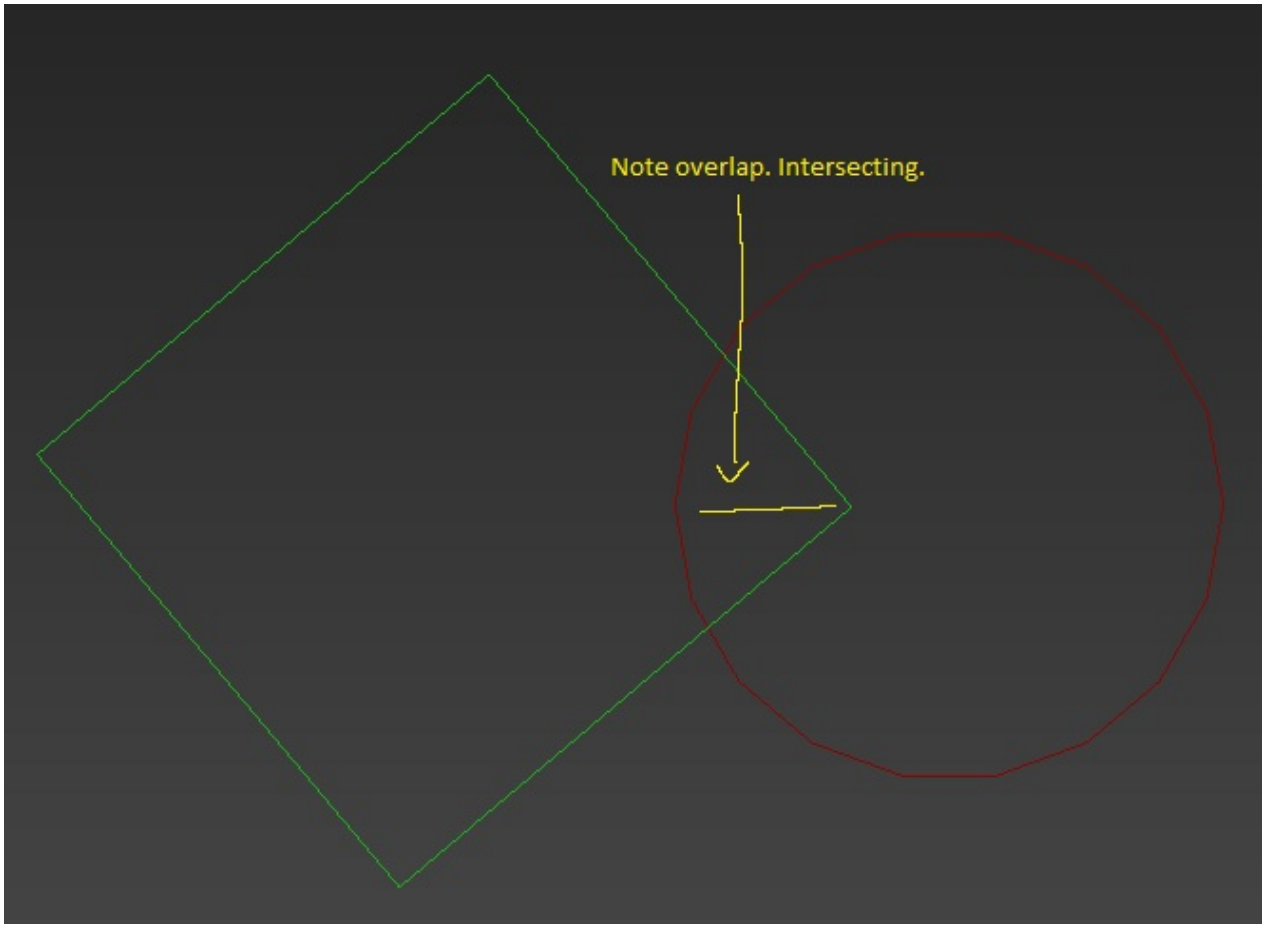
Collision Detection

This step involves finding/reporting when two objects are intersecting. This is the fundamental aspect of collision detection. Look at the following general illustration of what it means for two objects to not be intersecting and to be intersecting:

Not Intersecting:



Intersecting:



Collision Resolution

Once two objects have been found to be intersecting, we can do two things about:

1. Nothing. Let them remain intersecting and either do nothing or report the intersection.
2. Push them apart.

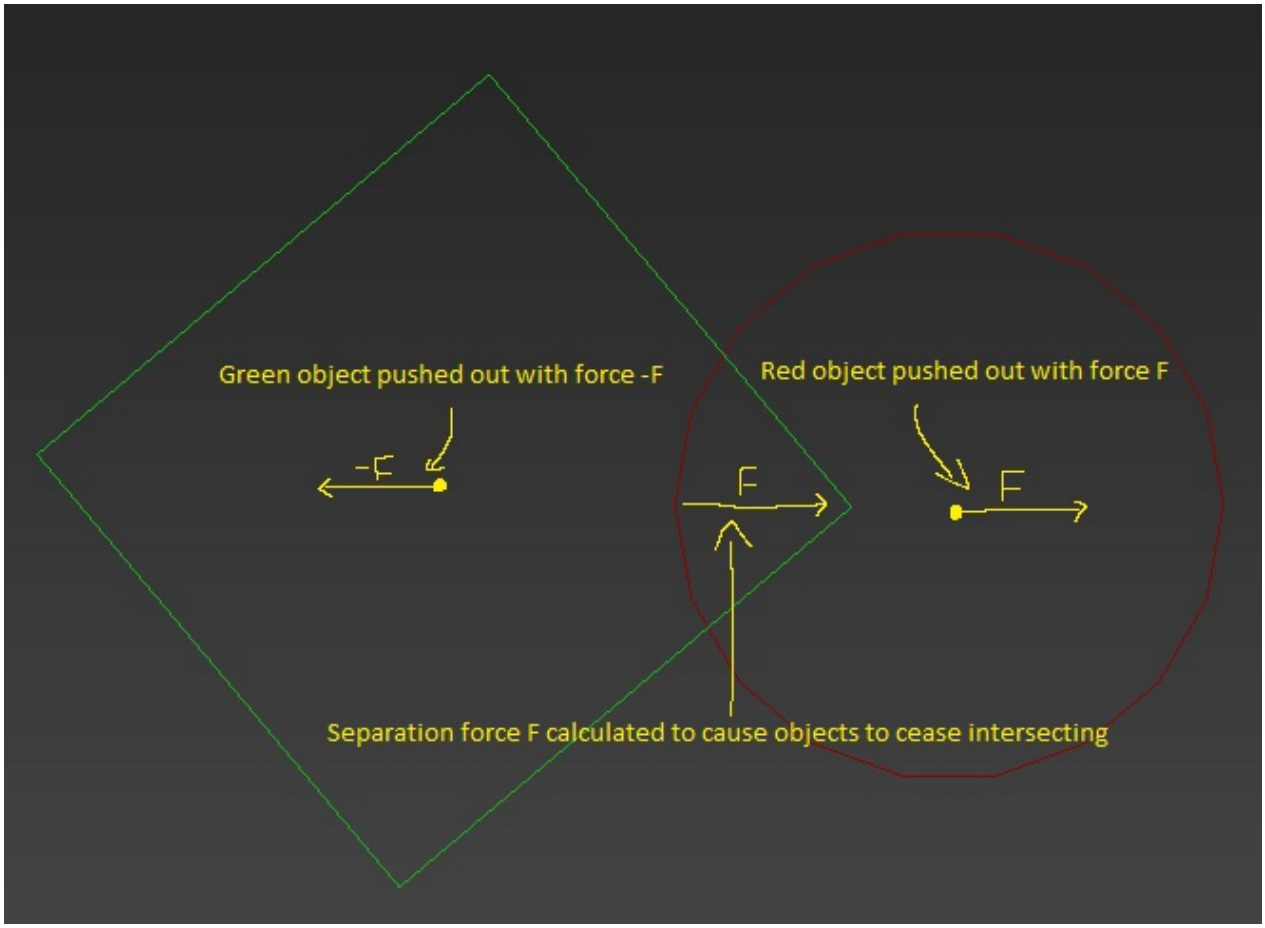
Depending on the requirements of the situation, we opt to do different things.

Allow Intersection

When we allow the intersection, we simply allow the objects to continue overlapping or to pass through each other on their own time. The physics engine does not act to push them apart.

Push Apart

In this situation, the physics engine finds a possible push vector and applies a force to cause the objects to cease intersecting. Such a vector may look as follows:



UDK

In the UDK, the math is done behind the scenes. We set the relevant properties on Actors and the UDK does the heavy lifting.

To use the UDK's collision system, an Actor must:

- Set its properties accordingly
- Must have one or more collision primitive components placed upon it

Properties

- var bool bCollideActors;
  - Whether this actor intersects other actors
- var bool bCollideWorld;
  - Whether this actor intersects with world geometry such as BSP and editor-placed static meshes
- var bool bBlockActors;
  - Whether this actor prevents other actors from passing through it

Engine Events

A number of engine events exist to notify an actor of when it collides or blocks other actors. They are:

- Touch
  - `event Touch(Actor Other, PrimitiveComponent OtherComp, vector HitLocation, vector HitNormal);`
- Bump
  - `event Bump(Actor Other, PrimitiveComponent OtherComp, Vector HitNormal);`

For other events, see the collision technical guide posted at the bottom of this page.

Typical Combinations

Let's look at some common combinations:

bCollideActors	bCollideWorld	bBlockActors	Result
false	false	false	No notifications are sent, other actors allowed to pass through this actor
true	true	false	Touch called when actor intersects with other actors or world, other actors allowed to pass through this actor
false	false	true	Other actors not allowed to pass through this actor, Bump called when actor contacts other actors
true	true	true	Other actors not allowed to pass through this actor, Bump called when actor contacts other actors, note that Touch is **not** called with this combination

Collision Primitives

See:

- Component Basics
- Useful Components

More Information

For more information, see the UDN collision reference pages:

- Collision Technical Guide
- Collision Reference

Component Basics

Components are objects that could be attached to actors to attach low level engine features or resources to those actors.

Component, Components Array

The root class of all components is the class `Component`. If you looked at this class in your UDK source, you would see that it is mostly barebones. Aside from the cpptext section, there is not a whole lot there. This class mostly exists for the engine to place some low level code that we're not allowed to see.

The first useful class in the heirarchy is `ActorComponent`. While this class is not significantly more useful than `Component`, it is the class used for Actor's component array. The following variable is the array that all actor's have that holds all components attached to an actor at any given time:

```
/**
 * Actor components.
 * These are not exposed by default to level designers for several reasons.
 * The main one being that properties are not propagated to network clients
 * when is actor is dynamic (bStatic=FALSE and bNoDelete=FALSE).
 * So instead the actor should expose and interface the necessary component variables.
 *
 * Note that this array is NOT serialized to ensure that the components array is
 * always loaded correctly in the editor.  See UStruct::SerializeTaggedProperties for det
 */

/** The actor components which are attached directly to the actor's location/rotation. */
var private const array<ActorComponent>    Components;
```

Adding a Component

To add a component to an actor, first you need to decide what type of component you want to add. For example, let's try adding a `StaticMeshComponent` (feel free to look this class up in the source tree). The easiest way to do this is by instantiating this component directly inside the Defaultproperties section of a class. Consider the following actor class:

```
class ExampleActor extends Actor;

Defaultproperties
{
    Begin Object Class=StaticMeshComponent Name=DisplaySM
        StaticMesh=StaticMesh'Awesome.Planet.Exterior'
    End Object
    Components.Add(DisplaySM)
}
```

In this example, we created a static mesh component, we named it *DisplaySM*, and we added it to the actor's components list. This was done using the **Begin Object** block syntax. This syntax is broken down as follows:

Begin Object Block

```
Begin Object Class=<Class of component to instantiate> Name=<Name to give to object>
    Property1=Property1Value
    Property2=Property2Value
    ...
End Object
```

These blocks can only exist within a Defaultproperties section. By adding such a section, you're telling the UDK to instantiate one of these components every time this actor is instantiated. The Components.Add(...) line tells the UDK to add the created component to the actor's components array.

Let's look at he parts of this block:

- class=<Class of component to instantiate>
  - The desired class of the component. There are many types of components in the engine; you tell the engine the type of component class to instantiate here.
- Name=<Name to give to object>
  - A name to refer to this component instance by. There can be many Begin Object blocks in a Defaultproperties section so referring to each individual component is done using this name.

Object Name

Note that an object's name is **only valid within the Defaultproperties section**. If you need to refer to this component inside actor code, create an actor variable and set its value equal to the instanced component. For example, we can add the following to `ExampleActor` above:

```
...
var StaticMeshComponent mySM;

function printSM()
{
    `log("My sm is "$mySM.StaticMesh);
}

Defaultproperties
{
    ...
    mySM=DisplaySM
}
```

Notice how we added a StaticMeshComponent reference variable to our actor and we added a function that could print out the current StaticMesh property on that component.

Updating A Parent Class' Component

It is possible through extending a class to update any of its component's properties through the Defaultproperties section. Look at the following example:

```
class ExampleBase extends Actor;

Defaultproperties
{
    Begin Object class=StaticMeshComponent Name=DisplaySM
        StaticMesh=StaticMesh'Foo.Bar.A'
    End Object
    Components.Add(DisplaySM)
}
```

We define an actor class named ExampleBase. It has a static mesh component property with a specified static mesh. We can override this in a child class:

```
class ExampleChild extends ExampleBase;

Defaultproperties
{
    Begin Object Name=DisplaySM
        StaticMesh=StaticMesh'Foo.Bar.B'
    End Object
}
```

Notice how in `ExampleChild`, our begin object block **does not have a class= property**. When we do this, we are telling the UDK to take the existing properties of the parent's component whose name is **DisplaySM** and update them with matching properties in this begin object block.

In this case, we changed the **StaticMesh** property to refer to the static mesh resource *Foo.Bar.B* instead of *Foo.Bar.A* which was in the parent.

Notice how we didn't have to add the component to the Components array in the child. This is because this is already being handled for this component in the parent.

More Information

For more information, see the UDN's components technical guide:

- Actor Components Technical Guide

Useful Components

This page will list the most commonly used components:

StaticMeshComponent

Extends MeshComponent. This component attaches a static mesh to an actor as well as adding the collision of that static mesh to the actor.

Properties

- StaticMesh StaticMesh
  - The static mesh resource to attach to the actor

Functions

- bool SetStaticMesh(StaticMesh NewMesh, optional bool bForce);
  - Sets a new static mesh on this instance

CylinderComponent

Extends PrimitiveComponent. A pure collision component with a width and a height. Used to add a collision cylinder to an actor. Note: invisible!

Properties

- float CollisionHeight
  - The height of the cylinder
- float CollisionRadius
  - The radius of the cylinder

Functions

- SetCylinderSize(float NewRadius, float NewHeight);
  - Sets a new size of cylinder for this instance

Other Components

MeshComponent

Extends PrimitiveComponent. Base class of all mesh components.

Properties

- array<MaterialInterface> Materials
  - Array of materials currently applied to this instance

Functions

- MaterialInterface GetMaterial(int ElementIndex);
  - Returns the material from the materials array at the given index
- SetMaterial(int ElementIndex, MaterialInterface Material);
  - Sets a new material in the materials array at the given index
- int GetNumElements();
  - Returns the number of materials/number of submeshes in the materials array/mesh

PrimitiveComponent

Extends ActorComponent. Base class for many components that can be positioned on an actor and could provide collision information.

Properties

Transform

- vector Translation;
  - Location, relative to owning actor
- rotator Rotation;
  - Rotation, relative to owning actor
- float Scale;
  - Uniform scale, relative to owning actor
- vector Scale3D
  - 3D scale, relative to owning actor

Collision

- bool CollideActors
  - Whether this component collides with other actors. In order for this to have any effect, the actor's bCollideActors property must also be true
- bool BlockActors

    - Whether this component blocks other actors. In order for this to have any effect, the actor's bBlockActors property must also be true

Lighting

- LightEnvironmentComponent LightEnvironment;
    - The light environment for this component. Useful for components that need to be 3D rendered, eg meshes. Meshes added to the world through the editor have this automatically set. Meshes on actors are considered dynamic and need to have their light environment explicitly set through this property. A good value for this property is a **DynamicLightEnvironment** component with default properties instantiated on the actor in the defaultproperties block.
- bool CastShadow;
    - Whether this component casts a shadow at all
- bool bCastDynamicShadow;
    - Whether this component casts a shadow from dynamic lights
- bool bCastStaticShadow;
    - Whether this component casts a shadow from static lights

Visibility

- bool HiddenGame;
    - Whether this component is invisible during game play
- bool HiddenEditor;
    - Whether this component is invisible in the editor
- bool bOwnerNoSee;
    - Whether this component is invisible to its owner
- bool bOnlyOwnerSee;
    - Whether this component is only visible for its owner

Functions

Transform

- SetTranslation(vector NewTranslation);
    - Sets relative location to owning actor
- SetRotation(rotator NewRotation);
    - Sets relative rotation to owning actor
- SetScale(float NewScale);
    - Sets relative scale to owning actor
- SetScale3D(vector NewScale3D);
    - Sets relative 3D scale to owning actor

- SetAbsolute(optional bool NewAbsoluteTranslation,optional bool NewAbsoluteRotation,optional bool NewAbsoluteScale);
  - Sets the above 3 values at the same time in the absolute/world
- vector GetPosition();
  - Returns the world space location of this component
- rotator GetRotation();
  - Returns the world space rotation of this component

Visibility

- SetHidden(bool NewHidden);
- SetOwnerNoSee(bool bNewOwnerNoSee);
- SetOnlyOwnerSee(bool bNewOnlyOwnerSee);

Lighting

- SetLightEnvironment(LightEnvironmentComponent NewLightEnvironment);

Spawn Review

The spawn function is used to instantiate and place an actor in the world. In the UDK, this function is a member function of the class `Actor`. Therefore, any actor object (that is, an object whose class that is a descendant of `Actor`) is able to spawn other actors.

Let's look at the signature of this function:

```
/** Spawn an actor. Returns an actor of the specified class, not
 * of class Actor (this is hardcoded in the compiler). Returns None
 * if the actor could not be spawned (if that happens, there will be a log warning indica
 * Defaults to spawning at the spawner's location.
 *
 * @note: ActorTemplate is sent for replicated actors and therefore its properties will a
 * at initial creation on the client. However, because of this, ActorTemplate must be a s
 * (an actor archetype, default object, or a bStatic/bNoDelete actor in a level package)
 * or the spawned Actor cannot be replicated
 */
native noexport final function coerce actor Spawn
(
    class<actor>      SpawnClass,
    optional actor      SpawnOwner,
    optional name     SpawnTag,
    optional vector   SpawnLocation,
    optional rotator  SpawnRotation,
    optional Actor    ActorTemplate,
    optional bool      bNoCollisionFail
);
```

Let's look at the parameters:

- class<Actor> SpawnClass
  - The class of actor to spawn. Notice that the parameter is a class restrictor limiting this to classes descending from Actor. The spawn function can **only instantiate classes that derive from Actor**.

From this point forward, all following parameters are optional:

- optional actor SpawnOwner
  - The owning actor. This defaults to None. If provided, the spawned actor's `Owner` variable is set to the actor that is passed in.
- optional name SpawnTag
  - A name to set as the actor's `tag` variable. Defaults to '' (empty/zero-length name).
- optional vector SpawnLocation

    - The location to place the actor at in the world. Defaults to the location of the actor on which the spawn function was called.
- optional vector SpawnRotation
    - The rotation to rotate the actor to in the world. Defaults to the rotation of the actor on which the spawn function was called.
- optional actor ActorTemplate
    - An actor template, archetype, or a static actor that could be used to initialize the default properties of the spawned actor. Typically set to None/skip.
- optional bool bNoCollisionFail
    - Typically, a blocking actor cannot be spawned while colliding with something that is blocking it. If we pass true here, it means that a collision check is not performed and the actor is forcibly spawned.

Example

Consider the following use of the spawn function in the PostBeginPlay event of an example actor:

```
class ExampleActor extends Actor;

var class<Actor> ClassToSpawn;

function PostBeginPlay()
{
    local vector SpawnLoc;
    local rotator SpawnRot;
    local UTPawn spawnedPawn;

    SpawnLoc.Z = 500;       //500 units up
    SpawnRot.Pitch = 8192;  //45 degrees pitched up

    // We will spawn a UTPawn actor at the location 0, 0, 500 and rotated to pitch 45 deg

    // The first way we will spawn using a class literal:
    spawnedPawn = spawn(class'UTPawn',,, SpawnLoc, SpawnRot);

    // The second way, we will use the class restrictor variable on this class:
    spawnedPawn = spawn(ClassToSpawn,,, SpawnLoc, SpawnRot);

    // Notice that in both situations we passed in SpawnLoc as the spawn location
    // and SpawnRot as the spawn rotation.
}

DefaultProperties
{
    ClassToSpawn=class'UTPawn'
}
```

Lab 8A - Collision, Inventory

Summary

You will implement a placeable actor that adds a counter inventory item.

Learning Outcomes

- Ability to manipulate the inventory manager
- Ability to add and interact with inventory items

Files to Implement

- L8Game.uc
- L8CounterItem.uc
- L8SpawnerBase.uc
- L8-Map.udk

Submission Requirements

Show your completed lab part A to your instructor(s) including a barrel that can be touched by the player and that adds a counter item to the player's inventory and then announces the counter item's counter value.

Instructions

L8Game

- Create the class L8Game which inherits from UTGame
- As usual, ensure that this game type is correctly set up to handle opening maps with the prefix "L8"

L8CounterItem

- Create the class L8CounterItem inherting from the class Inventory

- This class should have a single integer member variable named `counter`. Initialize this to 0.
- Implement the function `increment`, which should accept no parameters and return nothing
  - This function should simply increment the value of the `counter` variable
- Implement the function `announce`, which should accept no parameters and return nothing
  - This function should log a message that includes the current value of the `counter` variable

L8SpawnerBase

- Create the class L8SpawnerBase which inherits from Actor
  - Declare this class to be **placeable** and to be **ClassGroup Lab8**
- Add a static mesh component to this object and set the component's static mesh property to `StaticMesh'E3_Demo.Meshes.SM_Barrel_01'`
- Ensure that this actor is set to collide actors
- Implement the function Touch on this class, returning nothing and accepting the following parameters: `Actor Other, PrimitiveComponent OtherComp, Vector HitLocation, Vector HitNormal`
  - Have this function emit a log message every time it executes
  - Check if the incoming `Other` actor is in fact a `Pawn`, if it is...
    - Check if the class `L8CounterItem` is in its inventory
      - If it is not, create the item in/using Other's inventory manager
    - Call the increment function on Other's counter item
    - Call the announce function on Other's counter item

Lab 8B - Basic Weapons, Projectiles, Aiming

Summary

You will implement a basic weapon that shoots a projectile from the air and onto the ground. The projectile will fly towards the player from a pre set position.

Learning Outcomes

- Ability to create weapons and projectiles
- Ability to calculate flight paths such that an actor is targeted

Files to Implement

- L8SpawnerBase.uc
- L8Weapon.uc
- L8Projectile.uc

Submission Requirements

Show your completed lab part B to your instructor(s) including the weapon being picked up from the barrel actor and firing causing the weapon to shoot a projectile at the player from the center high point of the map.

Instructions

L8Weapon

- Create the class L8Weapon which inherits from UTWeapon
- Give it the following properties:
  - Add an `AnimNodeSequence` component and give it a name. Give it the following properties:
    - `bCauseActorAnimEnd` should be set to true

    - Add a skeletal mesh component. Give it a name. Give it the following properties:
        - Component should display the skeletal mesh `SkeletalMesh'WP_RocketLauncher.Mesh.SK_WP_RocketLauncher_1P'`
        - Set the first `AnimSets` property entry to `AnimSet'WP_RocketLauncher.Anims.K_WP_RocketLauncher_1P_Base'`
        - Set the `Animations` property to the name of the `AnimNodeSequence` component created above
    - Set the `Mesh` property to the name of the skeletal mesh component created above
    - For fire mode 0, set the following:
        - Set the fire interval to 1
        - Set the weapon projectile class = `class'L8Projectile'`
        - Set the weapon fire type to projectile
        - Set the weapon fire animation to `WeaponAltFireLaunch1End`
        - Set the shot cost to 1
    - Set the weapon idle animation to `WeaponIdle`
    - Set the maximum ammo count to 100
    - Set the ammo count to 50
- Override the function GetPhysicalFireStartLoc, returning a vector, and accepting an optional vector `AimDir`
    - Have this function return a location in the world that is 0 on the X and Y, and 500 on the Z
- Override the function GetAdjustedAim, returning a rotator, and accepting a vector `StartFireLoc`
    - Have this function return a rotator that represents the direction from `StartFireLoc` to the player's character's current location

L8Projectile

- Create the class L8Projectile which inherits from UTProjectile
- Give it the following properties:
    - Add a static mesh component. Give it a name. Give it the following properties:
        - Component should display the static mesh `StaticMesh=StaticMesh'LT_Light.SM.Mesh.S_LT_Light_SM_Light01'`
    - Add this component to the component list

L8SpawnerBase

- Update the class L8SpawnerBase from part 1 as follows:
    - In the Touch function:
        - If the contacted actor is a Pawn (which you checked for in part A), after incrementing the counter item and calling `announce`, do the following:

- Check the pawn's inventory for existence of the class `L8Weapon`
  - If not found, have the pawn's inventory manager create an inventory of class `L8Weapon`

AI, States, UTBot

Tutorial Overview

In this tutorial, we will create an AI that can follow a patrol path and opportunistically attack the player. We will also see how we can control the lifecycle of a bot, including preventing them from being restarted by the game. We will use the UT* branch of classes for this tutorial.

Downloading the map

- To start, download this base map.
- Place the map in UDKGame/Content/Maps

Patrol System

L9PatrolPath

- Create the class `L9PatrolPath`, have it extend PathNode, ensure it is placeable
- Set its sprite component to `Texture2D'MobileResources.HUD.AnalogHat'`
- Give it the *Patrol* property NextNode of type L9PatrolPath

L9AISpawner

- Create the class `L9AISpawner`, have it extend L9PatrolPath, ensure it is placeable
- Give it the static mesh `StaticMesh'NodeBuddies.NodeBuddy_PerchUp'`
- Give it the *Spawner* property BotsToSpawn of type int, have it default to 3
- Give it the *Spawner* property SpawnInterval of type float, have it default to 2
- Set the property `bStatic` to false as a default

PostBeginPlay()

- Override to set the timer to SpawnInterval, have it loop
- Execute super postbeginplay

L9Bot SpawnBot()

- Create this function
- Place the following code in it:

Timer()

- Implement this function
- If BotsToSpawn is <= 0, return
- Call the function SpawnBot. If it does **not** return None, decrement BotsToSpawn

```
local L9Bot NewBot;
local Pawn NewPawn;
local rotator StartRotation;

NewBot = Spawn(class'L9Bot');
if (NewBot == None)
{
    `log("Couldn't spawn "$class'L9Bot'$" at "$self);
    return None;
}

StartRotation = Rotation;
StartRotation.Yaw = Rotation.Yaw;

NewPawn = Spawn(class'UTPawn',,,Location,StartRotation);
if ( NewPawn == None )
{
    `log("Couldn't spawn "$class'UTPawn'$" at "$self);
    NewBot.Destroy();
    return None;
}

NewPawn.SetAnchor(self);
NewPawn.LastStartTime = WorldInfo.TimeSeconds;
NewBot.Possess(NewPawn, false);
NewBot.Pawn.PlayTeleportEffect(true, true);
NewBot.ClientSetRotation(NewBot.Pawn.Rotation, TRUE);
WorldInfo.Game.AddDefaultInventory(NewPawn);
WorldInfo.Game.SetPlayerDefaults(NewPawn);

NewBot.EnterPatrolPath(self);

return NewBot;
```

L9LinkGun

- Create the L9LinkGun class, have it extend UTWeap_LinkGun
- Set the default property WeaponRange to 500

byte BestMode()

- Override this function to return 0

bool CanAttack(Actor Other)

- Have this function return true if its super version returns true and the distance between other and instigator is <= our WeaponRange

L9Game

- Create the class `L9Game`, have it extend UTGame
- Set the array property element 0 `DefaultInventory` to the class `L9LinkGun`

RestartPlayer(Controller aPlayer)

- Override this function to test if aPlayer is an L9Bot and is in the state Dead
  - If they are, call destroy on them
  - Otherwise, call our super version

Logout(Controller exiting)

- Override this function to test if `exiting` is an L9Bot
  - If it is, simply return
  - Otherwise, call our super version

L9Bot

- Create the class L9Bot, have it extend UTBot
- Give it the following properties:

```
var float AggroDistance;
var float EscapeDistance;
var float AttackDistance;
var float PatrolPointReachedThreshold;
var L9PatrolPath NextPatrolPoint;
var L9AISpawner MySpawner;

var private Actor _intermediate;
var private float _dist;
```

- Set the default of PatrolPointReachedThreshold to 50
- Set the default of AggroDistance to 600
- Set the default of EscapeDistance to 1500
- Set the default of AttackDistance to 300

State FollowingPatrolPath

- Create this state, make it an auto state
- Create the `Begin` state label. Place the following code under it:
  - Check if NextPatrolPoint is not None
    - If it's not, check if the next patrol point is directly reachable
      - If it is, move towards it
      - If not, assign an intermediate path into _intermediate
        - Move toward _intermediate
  - Call LatentWhatToDoNext

State HuntingPlayer

- Create this state
- Create the `Begin` state label. Place the following code under it:
  - Check if Enemy is not None and Pawn is not None
    - Calculate the distance between our pawn and the Enemy then place this value in _dist
    - If _dist is greater than the escape distance or a fast trace fails between our pawn's location and the enemy location, set Enemy to None
    - Otherwise if _dist is <= attack distance
      - Have the pawn switch to its best weapon then fire weapon at enemy
    - Otherwise
      - If enemy is directly reachable, move towards the enemy
      - Otherwise, assign an intermediate path into _intermediate
        - Move toward _intermediate
  - Call LatentWhatToDoNext

bool IsPawnTouchingActor(Actor other)

- Implement this function
- Have this function return false if pawn is None or if other is None
- Return true if the distance between our pawn and other is less than the patrol point reached threshold

EnterPatrolPath(L9PatrolPath path)

- Implement this function
- Set the next patrol point to path
- if path is an L9AISpawner, assign it to MySpawner
- If path is none, go to state ''

- Otherwise, go to state 'FollowingPatrolPath'

PawnDied(Pawn P)

- Implement this function
- If MySpawner is not none, increment its BotsToSpawn variable
- Call the super version of this function

HuntEnemy(Pawn p)

- Implement this function
- If our Pawn is None, return
- Assign P into Enemy
- Assign P into Focus
- Go to state 'HuntingPlayer' at label 'Begin'

NotifyTakeHit(Controller InstigatedBy, Vector HitLocation, int Damage, class<DamageType> damageType, Vector momentum)

- Implement this function
- Call the super version of it
- If InstigatedBy is not None and its Pawn is not None
  - Call HuntEnemy on its pawn

Possess(Pawn aPawn, bool bVehicleTransition)

- Implement this function
- Call the suerp version of this function
- If NextPatrolPoint is not None, go to state 'FollowingPatrolPath' at label 'Begin'

protected event ExecuteWhatToDoNext()

- Implement this function
- Create the local variable of type PlayerController PC
- If our pawn is None, call destroy() and return
- Call GetALocalPlayerController() and assign the result into pc
- If pc is not None and pc's Pawn is not none and the distance between our pawn and pc's pawn is <= aggro distance then call HuntEnemy on pc's pawn and return
- If the next patrol point is not none:

    - Call IsPawnTouchingActor passing it Next Patrol point. If it returns true, assign NextPatrolPoint.NextNode into NextPatrolPoint
- If next patrol point is not none, go to state 'FollowingPatrolPath' at label 'Begin'

Updating the Map

- Open your map
- Place an L9AISpawner in one of the rooms
- Place another L9AISpawner in another room on the other side of the map
- Create two loops of L9PatrolPath objects spanning from each spawner
    - Have the loops move through 4 rooms creating a cyclic path
- Connect the L9PatrolPath objects via their `NextNode` property
    - Include the L9AISpawner objects in this process
- Create a path node at every hallway entrance of each room and one additional path node in the center of each empty room
- Rebuild geometry
- Rebuild lights (without lightmass)
- Rebuild paths
- Save and exit

Completed Tutorial

Download the completed tutorial code here.

- Place the `Lab9` folder in Development/Src
- Place `L9-Map.udk` in UDKGame/Content/Maps
- Before running the game or opening the map in the editor, be sure to **compile**

WIP - Advanced Weapons

Lecture Topics

- Projectile
  - Tracking
    - SeekTarget
    - BaseTrackingStrength
    - HomingTrackingStrength