CIS568: Game Design Practicum

Unreal Tutorial

INTRODUCTION

This Unreal tutorial assumes that you have gone through the Unity tutorial and understand the fundamental programming and mathematic concepts behind the Asteroids game.

This tutorial will teach you the fundamentals of Unreal Engine and show you the similarities and differences between Unreal and Unity by porting the Asteroids game to Unreal. Although the names and classes used by Unreal are different from Unity, many of them parallel one another. In addition, Unreal Engine has two different methods of scripting – Blueprints and C++. Blueprints is a visual scripting system new to Unreal Engine 4. The programming concepts that you're already familiar with can be directly applied to the visual scripting system of Blueprints. Scripting in C++ is more similar to the programming that you've done in Unity, but requires a bit more setup.

We'll be going through both of these methods, using a combination of C++ to bring over the logic from the Asteroids game, and Blueprints to set up the visual assets in the scene.

If you haven't already, create a new Epic Games account here: https://accounts.unrealengine.com/login. Open the Epic Games Launcher and sign in.

Navigate to the Unreal Engine tab and launch Unreal Engine. The version at the time of writing is 4.18.3

Note: All the parameter values used in this guide are set to a "one-size-fits-all" so you should have no issues completing the basic gameplay. However, you may want to change or modify some of the parameter values to improve or optimize some of the gameplay.

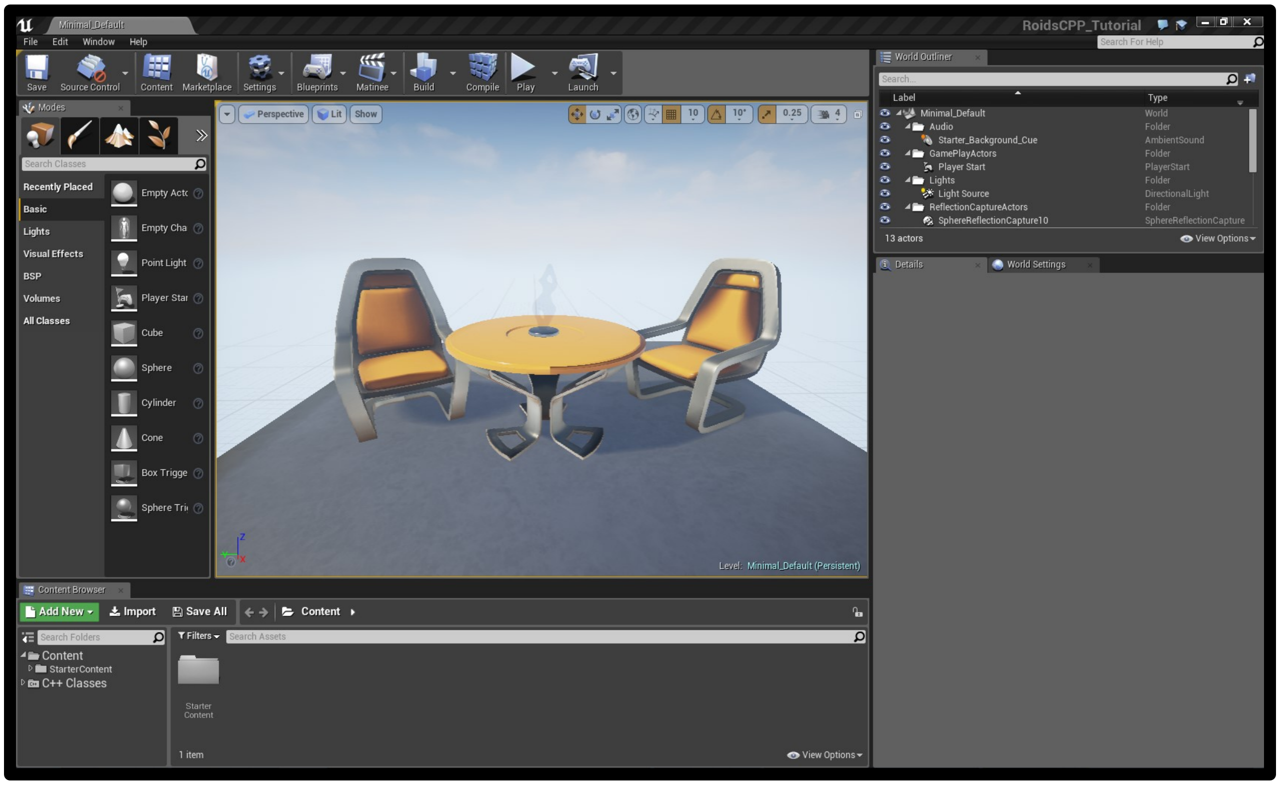
Part 1: Blueprints

In this section, we will recreate the Asteroids game using only Blueprints. This will help you get a better idea of how to use the visual scripting system in Unreal.

1. SET UP A BLUEPRINT PROJECT.

Let's start from scratch by making a new Blueprint project. Select "File -> New Project", and create a Blank project named RoidsBP. Create a new empty level and name it "MainLevel". Make sure you select and add the starter content.

The engine will open your project and the empty level. In the following subsections we will explore the panels that make up the default editor window.



1.1 Viewport

The large panel at the center of the editor window is the **Viewport** panel. Similar to the Scene Tab Unity, the Viewport is used to display and navigate the level, and is used to select and transform actors.

There are a few ways to navigate the Viewport.

1. Hold the right-mouse button to rotate.
2. Hold the left-mouse button to move forward, backward, left, and right.
3. Scroll the middle mouse button to move forward and backward.
4. Hold the middle-mouse button to move up, down, left, and right without changing rotation.
5. While holding any mouse button, press **E** and **Q** to move up and down.
6. While holding any mouse button, press **W, A, S,** and **D** to move forward, left, backward, and right.

We find it's easiest to navigate the world using the **E, Q, W, A, S,** and **D** keys with the right-mouse button.

1.2 Standard menu bar

At the top left is the **Standard Menu Bar**, containing many features of the editor not immediately accessible.

1. The **File** dropdown contains loading, saving, and importing features for levels and projects.
2. The **Edit** dropdown contains history, edit, and preferences features.
3. The **Help** dropdown contains many links around the web for documentation and community support.

1.3 Toolbar
Above the **Viewport** is the **Toolbar**, which contains buttons for common commands in the editor.

1. The **Settings** dropdown contains many of the project's fundamental game and development settings.
2. Clicking the **Build** button will build all of your levels according to the configuration settings accessible through the corresponding dropdown menu.
3. Clicking the **Play** button will play the level according to the configuration settings accessible through the corresponding dropdown menu.
4. Clicking the **Launch** button will launch the level as a stand-alone application.

1.4 Modes
To the left of the **Toolbar** and beneath the **Standard Menu Bar** is the **Modes** panel. Editor modes provide access to components that can be added to a level.

1.5 Content Browser
Beneath the **Modes** panel is the **Content Browser**, which provides access to the project's directory of assets.

1. The **Add New** dropdown contains buttons that create new assets in the selected folder.
2. Clicking the **Import** button opens a window from which existing assets can be added to the project.
3. Clicking the **Save All** button will save all modified project assets.
4. The project's directory can filtered by name and/or file type.
5. Some assets can be added to the open level by dragging their thumbnail from the content browser into the scene.
6. Double-clicking an asset will open a corresponding editor window if the file type is supported.

1.6 World Outliner
At the top right is the **World Outliner**, which provides a hierarchical list of actors that exist in a level.

1. Actors selected in the **World Outliner** are highlighted in the **Viewport**.
2. Actor parent-child hierarchies are reflected in the **World Outliner**.
3. Right-clicking empty space in the panel will open a dropdown from which new folders can be created. These folders exist solely for organizational purposes.
4. Attributes of a selected actor are exposed in the **Details Panel** beneath.

2. IMPORT SOME ASSETS.

Currently, our project doesn't have any custom Assets. Let's add our ship and asteroid assets.

Unzip the tutorial file "Unity Tutorial Assets.rar" from Canvas.

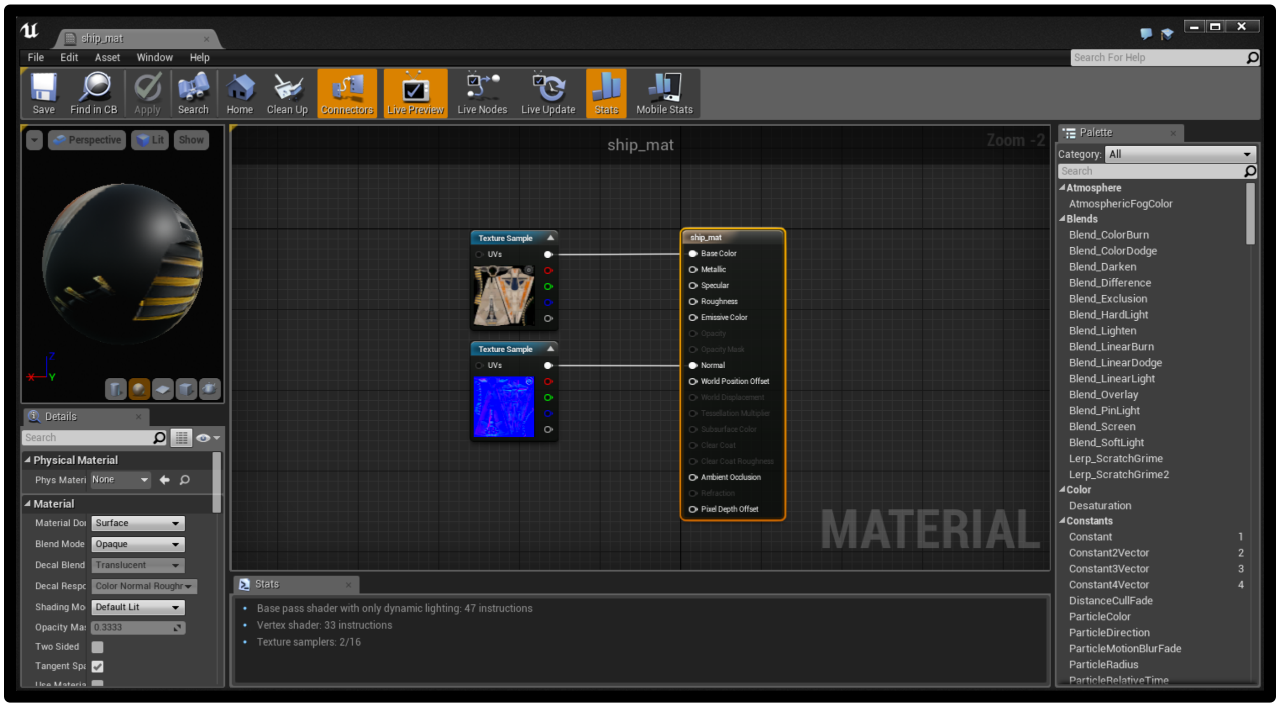
Drag in the “prop_asteroid_01.fbx” and “vehicle_playerShip.fbx” files from the Models folder into the Content Browser. Uncheck the boxes under the “Mesh” and “Material” groups and select “Import All”. Clear any messages that pop up.

Next, drag in all the .png files from the Textures folder. We will use these to create the materials for the ship and asteroid.

3. CREATE SOME MATERIALS

Select “Add New -> Material” in the Content Browser, name the material “ship_mat”, and double click on the new material. This window is the Material Editor. Similar to node graphs in other 3D packages, we can use this window to hook up nodes to create our material properties. You can use right click to move around the node graph, and left click to drag and link nodes together.

Drag the “vehicle_playerShip_orange_dff” and “vehicle_playerShip_orange_nrm” textures into the Material Editor, and connect their white pins to the “Base Color” and “Normal” pins. Select “Apply” on the Menu bar to apply these changes.



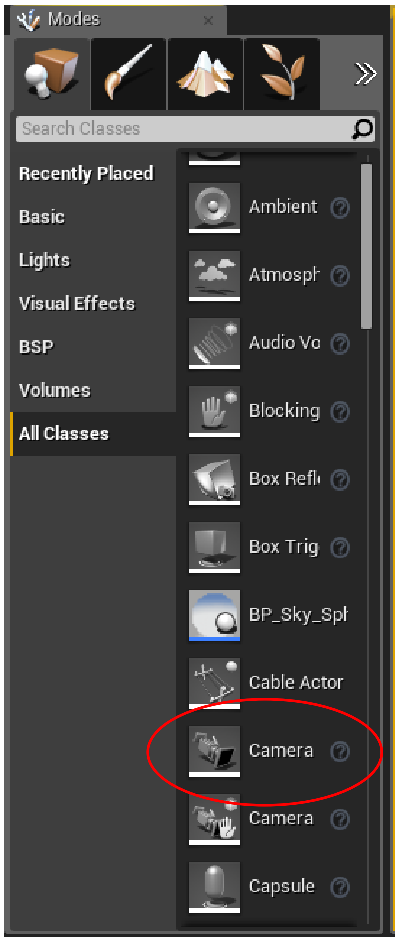
Physical Material
Phys Materi: None
Material
Material Doi: Surface
Blend Mode: Opaque
Decal Blend: Translucent
Decal Respo: Color Normal Roughn
Shading Mo: Default Lit
Opacity Ma: 0.3333
Two Sided: ☐
Tangent Sp: ✓

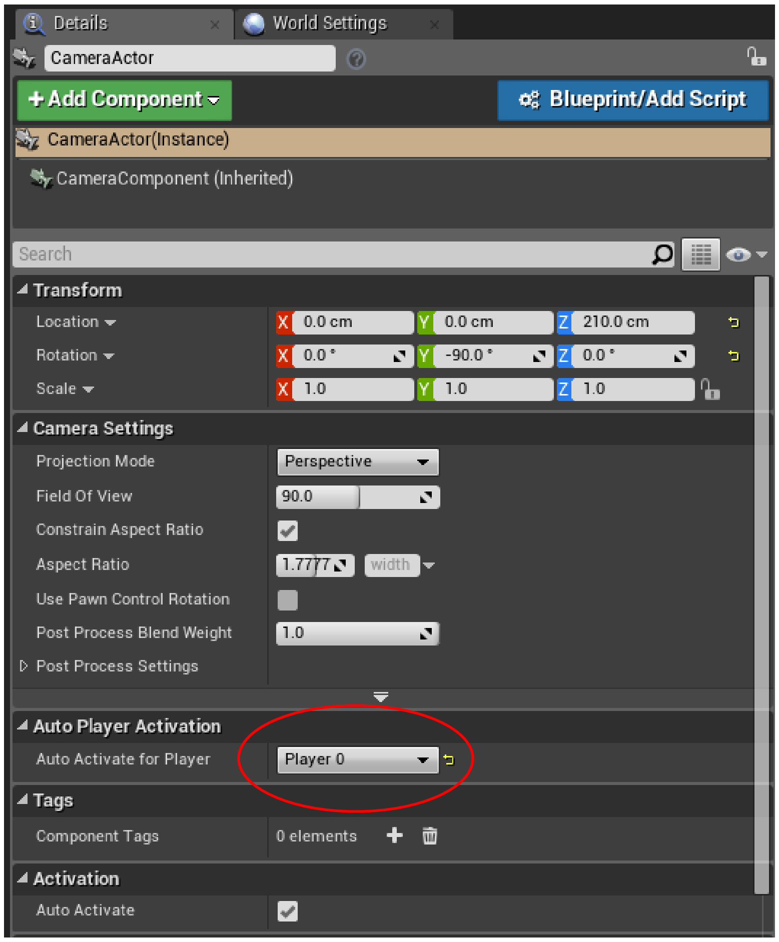
- Base pass shader with only dynamic lighting: 47 instructions
- Vertex shader: 33 instructions
- Texture samplers: 2/16

Repeat this process with the asteroid textures to create a material called “asteroid_mat”.

4. ADD A CAMERA AND LIGHT TO THE SCENE.

The first we need to do is set up the cameras. In the “Modes” panel, find “All Classes->Camera” and drag one into the viewport. Change the Location to (0.0, 0.0, 210.0) and the Rotation to (0.0, -90.0, 0.0). In addition, in the “Auto Activate for Player” dropdown, select “Player 0”.





CameraActor

CameraActor(Instance)

CameraComponent (Inherited)

Transform

	X	Y	Z
Location	0.0 cm	0.0 cm	210.0 cm
Rotation	0.0 °	-90.0 °	0.0 °
Scale	1.0	1.0	1.0

Camera Settings

Projection Mode	Perspective
Field Of View	90.0
Constrain Aspect Ratio	✓
Aspect Ratio	1.7777 width
Use Pawn Control Rotation	
Post Process Blend Weight	1.0
Post Process Settings	

Auto Player Activation

Auto Activate for Player	Player 0

Tags

Component Tags	0 elements

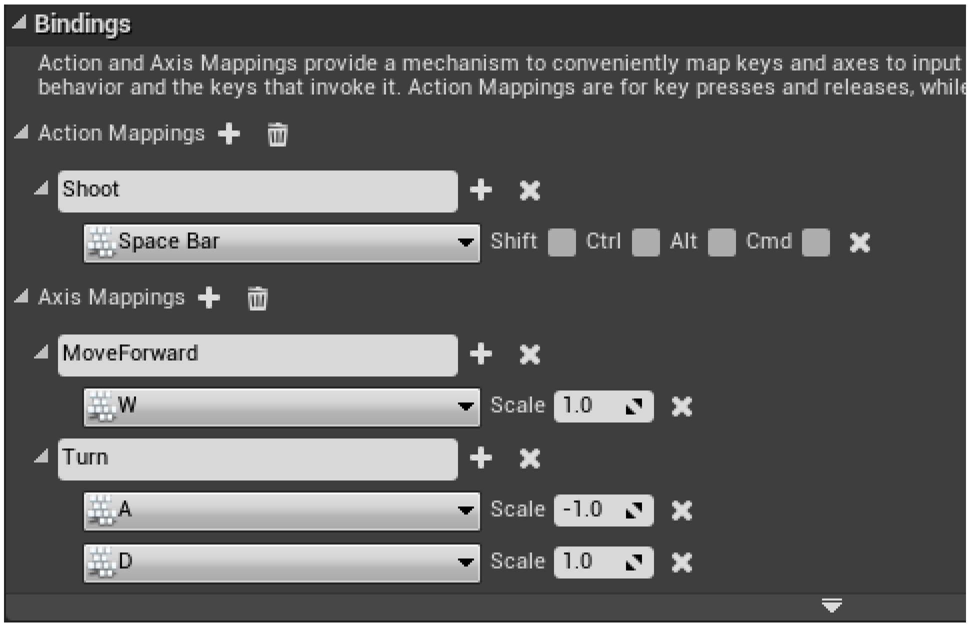
Activation

Auto Activate	✓

We also need a light in order to see anything when we play. Navigate to the “Lights” tab and drag a Directional Light into the scene. Rotate the light using the channels until you are satisfied with the lighting on the ship.

5. ADD MOVEMENT FOR YOUR SHIP.

To add movement, we first need to set up the player input for the project. Go to “Edit -> Project Settings”, and select the “Input” option on the left menu. You should see a page where you can edit the key bindings for your game. Use the plus arrows to add mappings and keys, and use the dropdown menus to find specific keys. Customize your bindings to the following arrangement:



Bindings

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while

Action Mappings

Shoot

Space Bar — Shift Ctrl Alt Cmd

Axis Mappings

MoveForward

W — Scale 1.0

Turn

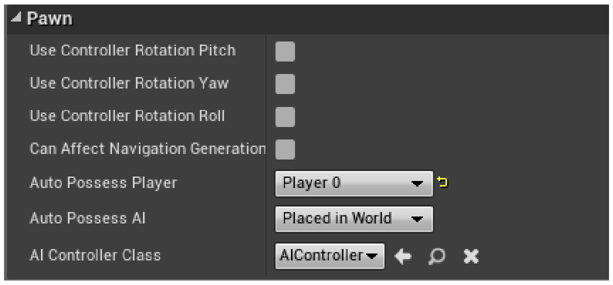
A — Scale -1.0

D — Scale 1.0

6. CREATE A SHIP IN BLUEPRINTS.

Select “Add New -> Blueprint Class” and create a new Blueprint that derives from Pawn. Call it ShipBP.

In the ShipBP(self) component Details, set “Auto Possess Player” to Player 0.

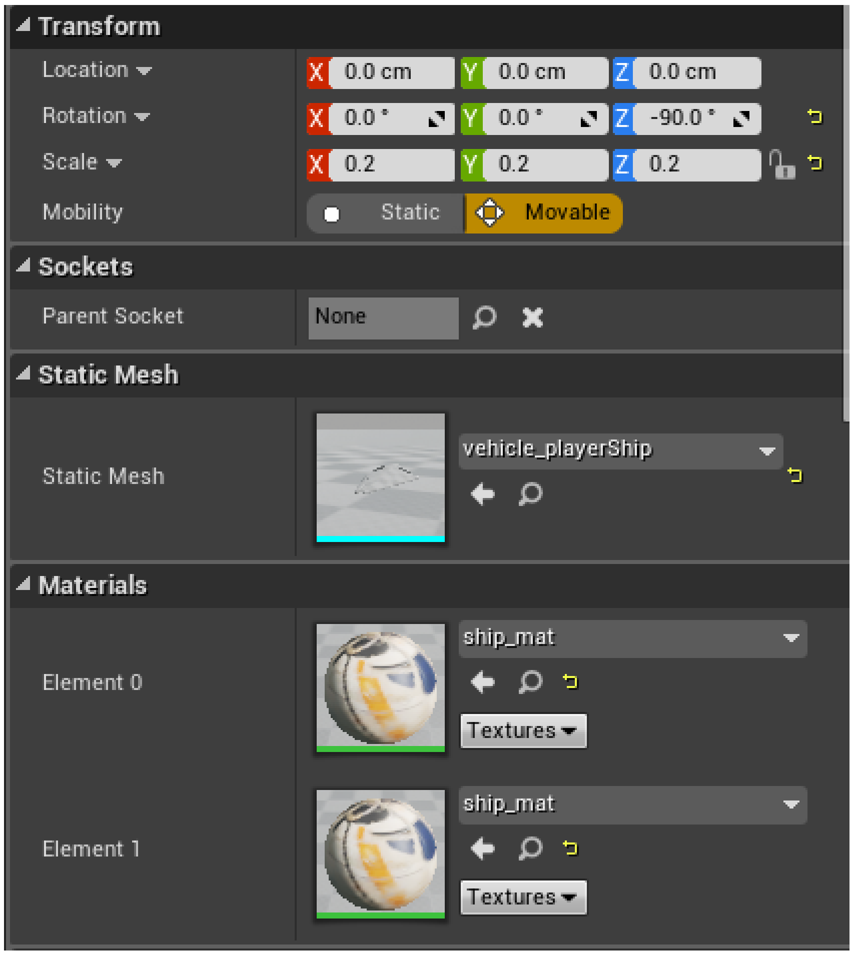


Pawn

Use Controller Rotation Pitch	☐
Use Controller Rotation Yaw	☐
Use Controller Rotation Roll	☐
Can Affect Navigation Generation	☐
Auto Possess Player	Player 0
Auto Possess AI	Placed in World
AI Controller Class	AIController

Add a Sphere Collision component and name it “ShipCollider” and set it as the Root Component by dragging it over the default root. Set the Sphere Radius to 10.0, turn on Simulate Physics, and turn off Enable Gravity. Finally make sure the collision presets are set to “BlockAllDynamic”

Add a Static Mesh component named “ShipMesh” to the blueprint. Set the mesh to “vehicle_playerShip”, and set both of the materials to “ship_mat”. In addition, set the rotation of the ship to (0.0, 0.0, -90.0).



Transform

Location X 0.0 cm Y 0.0 cm Z 0.0 cm

Rotation X 0.0 ° Y 0.0 ° Z -90.0 °

Scale X 0.2 Y 0.2 Z 0.2

Mobility Static [✓ Movable]

Sockets

Parent Socket None

Static Mesh

Static Mesh vehicle_playerShip

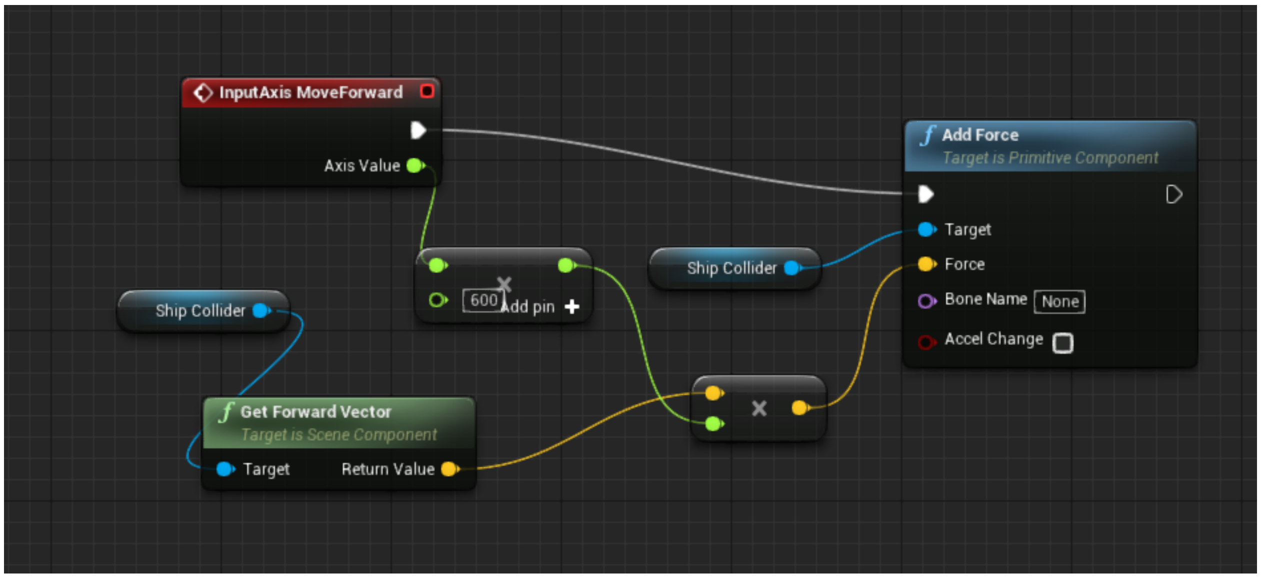
Materials

Element 0 ship_mat

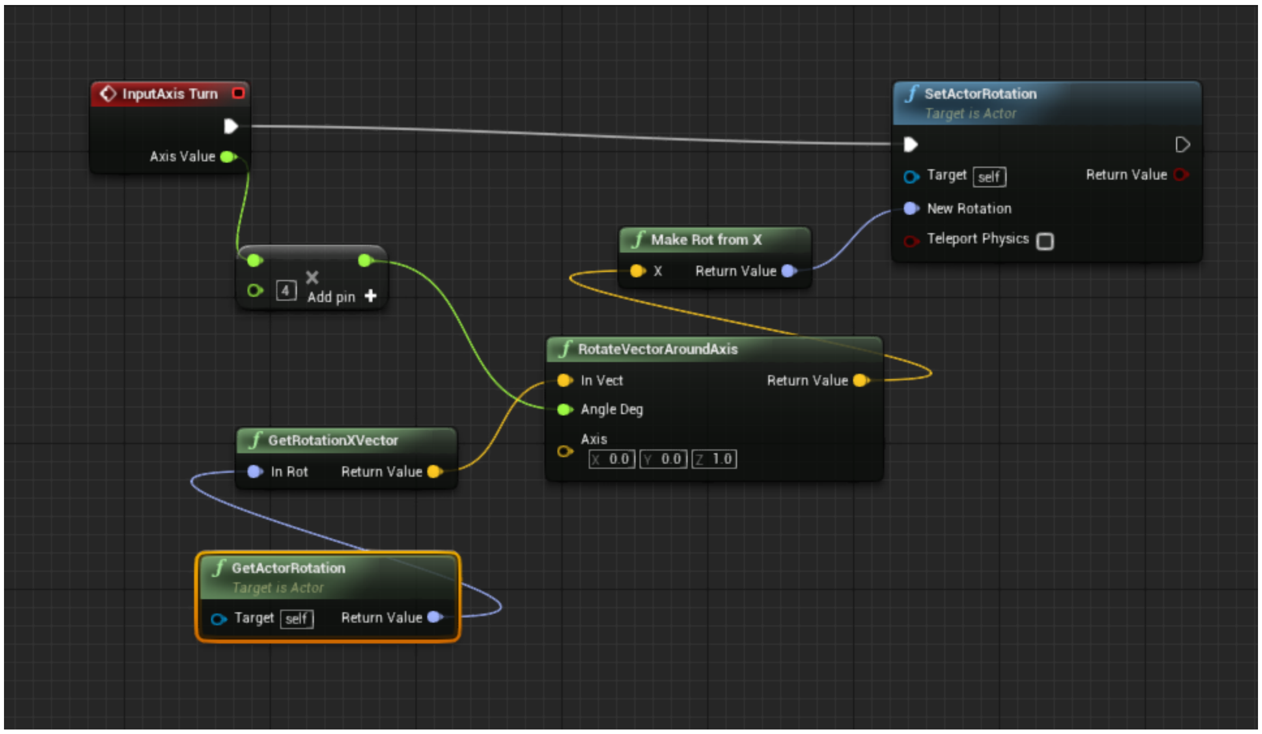
Element 1 ship_mat

Navigate to the Event Graph. We can now place the nodes to move the ship. Right click to bring up the node menu and place the MoveForward Axis Event node. The white triangle on certain nodes are Execution pins, which determine the flow of execution of the nodes. Events like BeginPlay and MoveForward begin the execution flow and allow following nodes to be executed.

Create an "Add Force(ShipCollider)" node, which will apply a force to our Collision Component. This node should spawn with a reference to our ShipCollider component already connected. Then, attach additional nodes in this arrangement (The multiplication nodes are named "float * float" or "vector * float"):



To control the rotation of the ship, place a Turn Axis Event node, and add the following nodes. The “GetRotationXVector” and “Make Rot from X” nodes are generated automatically when you try to connect a vector pin to a rotation pin, and vice versa.



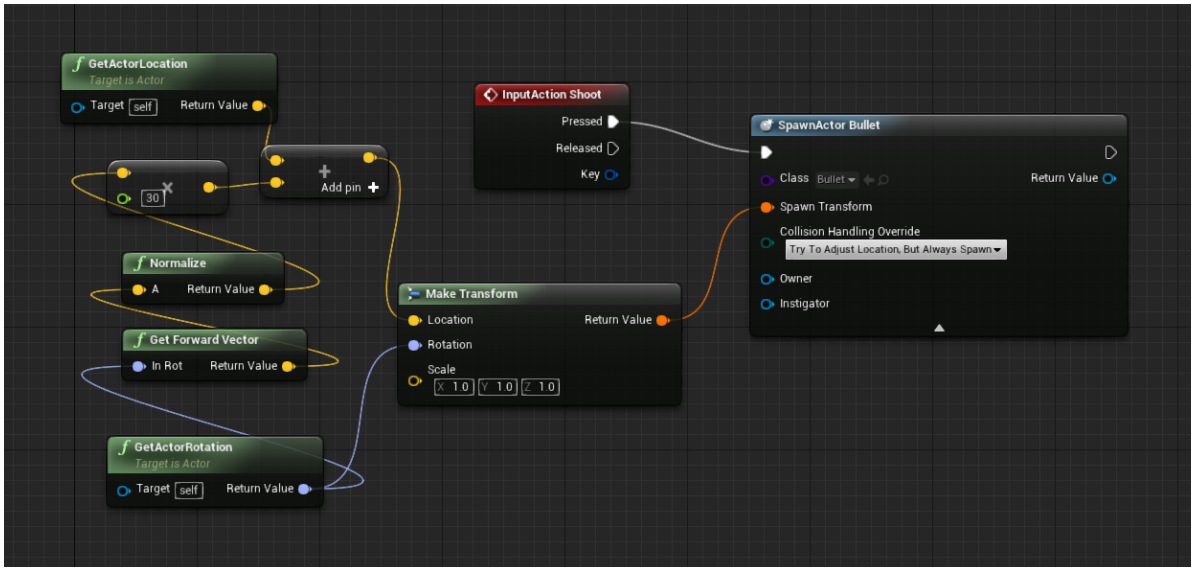
Compile the blueprint, and drag a Ship into your level at the origin. When you play, you should be able to move and turn your ship.

7. CREATE A BULLET IN BLUEPRINTS.

Let’s create a bullet using Blueprints. First, create a new Actor Blueprint and name it “Bullet”.

Replace the Root Component with a Sphere Collision component named “BulletCollider”. Set the Sphere Radius of the collider to 4.0.
Add a Sphere component named “SphereMesh” underneath your collider, which will be your bullet visual. Set the scale of the mesh to (0.08, 0.08, 0.08).
Finally, add a Projectile Movement component to your Bullet, set the Initial Speed to 100.0, and set the Projectile Gravity Scale to 0.0. This component will make your bullet move forward the moment it is instantiated.

In order to shoot these bullets, go back to the Ship Blueprint and add the following graph. In order to spawn our actor, you need the “Spawn Actor from Class” node.



This will fire a Bullet, 30 units, in front of the ship when the spacebar is pressed. Try it out! As you can see, it is very easy to use preset components like the Projectile Component to quickly create actors like this Bullet.

8. CREATE AN ASTEROID IN BLUEPRINTS.

Create a new Actor Blueprint named “Asteroid”.

Replace the Root Component with a Box Collision Component named “AsteroidCollider”. Set the Box Extent to (12.0, 15.0, 12.0). Turn on “Simulate Physics”, turn off “Enable Gravity”, and turn on “Simulation Generates Hit Events”. In addition, set the Collision Preset to “BlockAllDynamic”.
Add a Static Mesh component named “AsteroidMesh”, set the mesh to “prop_asteroid_01”, and set the material to “asteroid_mat”. Set the scale to (0.25, 0.25, 0.25), and set the Collision Preset to “OverlapAllDynamic”.

9. CREATE AN EXPLOSION.

Now, let's have the asteroid explode when it is destroyed. Our explosion will be its own Actor. We will create this Actor as a Blueprint to make customizing our particle event easier. When our asteroid dies, we will destroy the Asteroid object and spawn our Explosion in its place while playing an explosion sound.

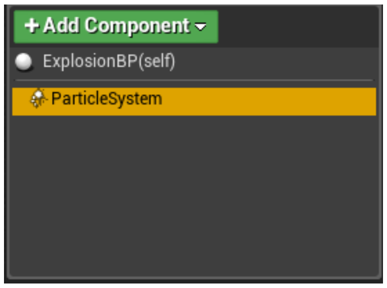
Create a new Blueprint Class which extends Actor, and name it "ExplosionBP". We'll use the Blueprint Editor to set up our explosion particle system.

The Blueprint Editor consists of:

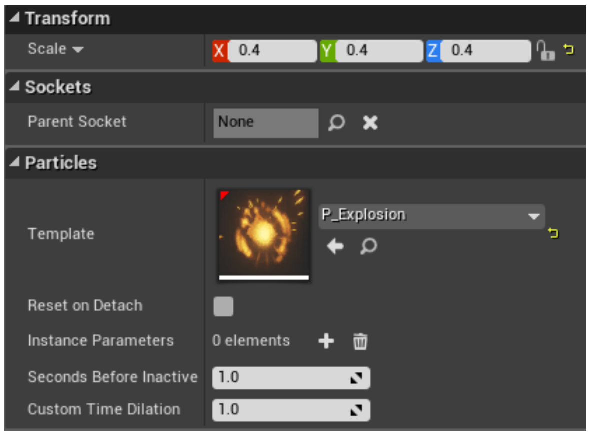
1. A **Toolbar** panel of common commands
2. A **Components** panel of variables
3. A **My Blueprint** panel containing the different events, functions, and variables of your blueprint.
4. A **Details** panel that works much like the standard Details panel
5. An **Event Graph** where you can create visual scripts
6. A **Construction Script** which is equivalent to a constructor and is run when the object is created.

In the Components panel, you can see that the Root Component is the "DefaultSceneRoot". We want to replace this component with our particle system.

Select "Add Component -> Particle System", and drag this particle system onto the root component. This should replace the root with our new component.

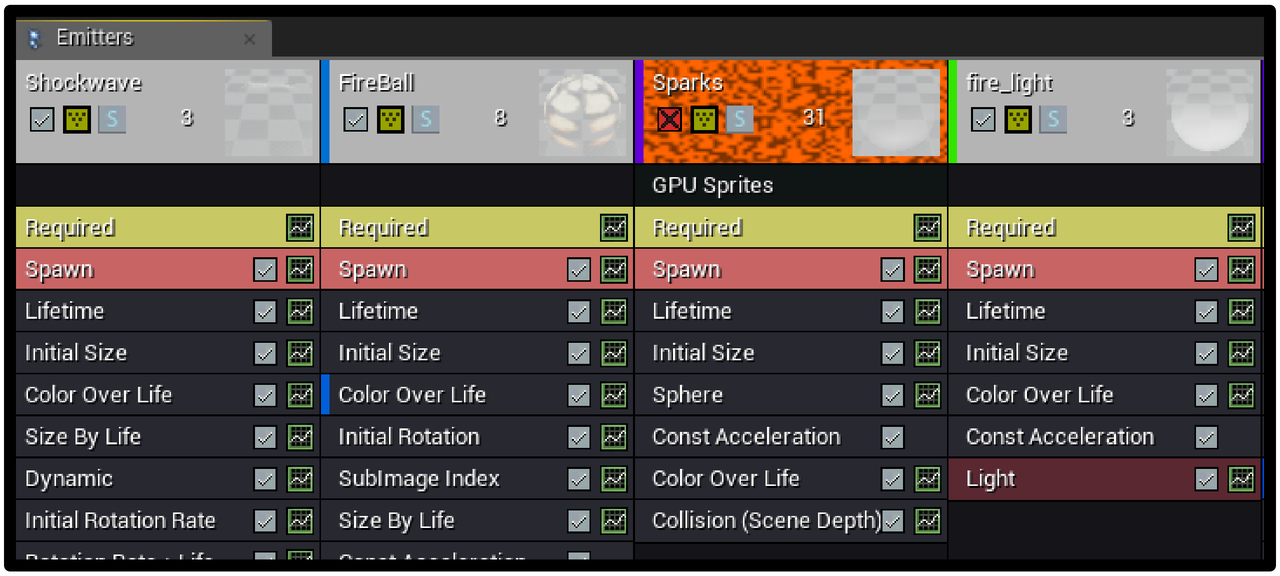


Under "Details -> Particles", open the template dropdown and select "P_Explosion". A large explosion should play in the viewport. Scale it down by setting the component scale to (0.2, 0.2, 0.2).


Transform
Scale X 0.4 Y 0.4 Z 0.4
Sockets
Parent Socket None
Particles
Template P_Explosion
Reset on Detach
Instance Parameters 0 elements
Seconds Before Inactive 1.0
Custom Time Dilation 1.0

This default explosion also contains sparks. We can remove these from our particle system by double clicking on the explosion image to open up the Particle Editor. Or if you like, you can just leave them there.

The “Emitters” tab contains all the particle emitters that make up our effect. Select the check mark on the Sparks emitter to disable it.



Emitters

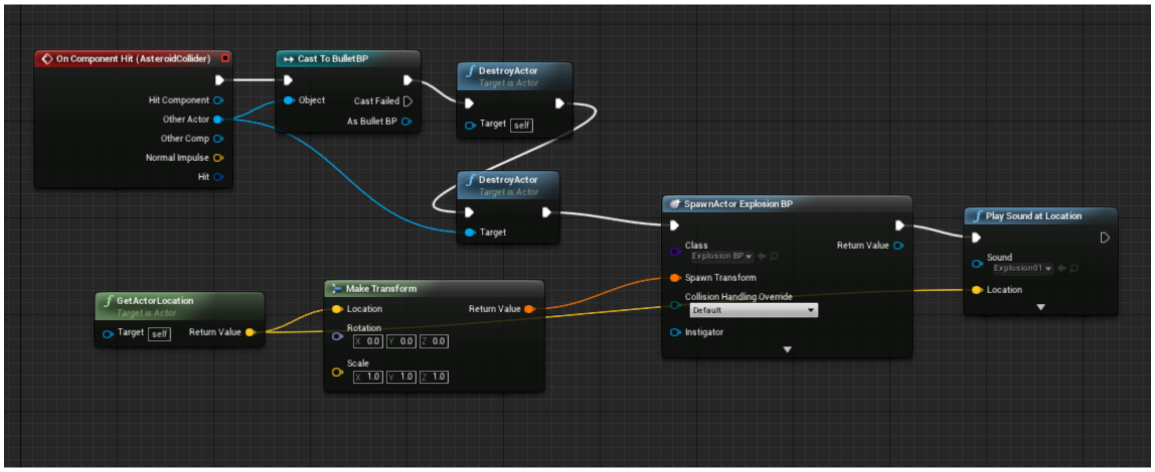
Shockwave	FireBall	Sparks	fire_light
3	8	31	3
		GPU Sprites	
Required	Required	Required	Required
Spawn	Spawn	Spawn	Spawn
Lifetime	Lifetime	Lifetime	Lifetime
Initial Size	Initial Size	Initial Size	Initial Size
Color Over Life	Color Over Life	Sphere	Color Over Life
Size By Life	Initial Rotation	Const Acceleration	Const Acceleration
Dynamic	SubImage Index	Color Over Life	Light
Initial Rotation Rate	Size By Life	Collision (Scene Depth)	
Rotation Rate + Life	Const Acceleration		

If you drag this ExplosionBP Actor into your level, you should see it explode. If it’s too large or too small, you can adjust the scale to your liking.

We also need to prepare our explosion sound effect to be played on hit.

10. HAVE THE ASTEROID EXPLODE ON COLLISION.

Now, place the following nodes in the Event Graph for Asteroid.



If you place an Asteroid in your scene, it should now explode when colliding with a Bullet.

Also make sure you’ve enabled “Simulation Generates Hit Events”

If you place an Asteroid in your scene, it should now explode and play a sound when colliding with a Bullet.

11. CREATE A GLOBAL ACTOR.

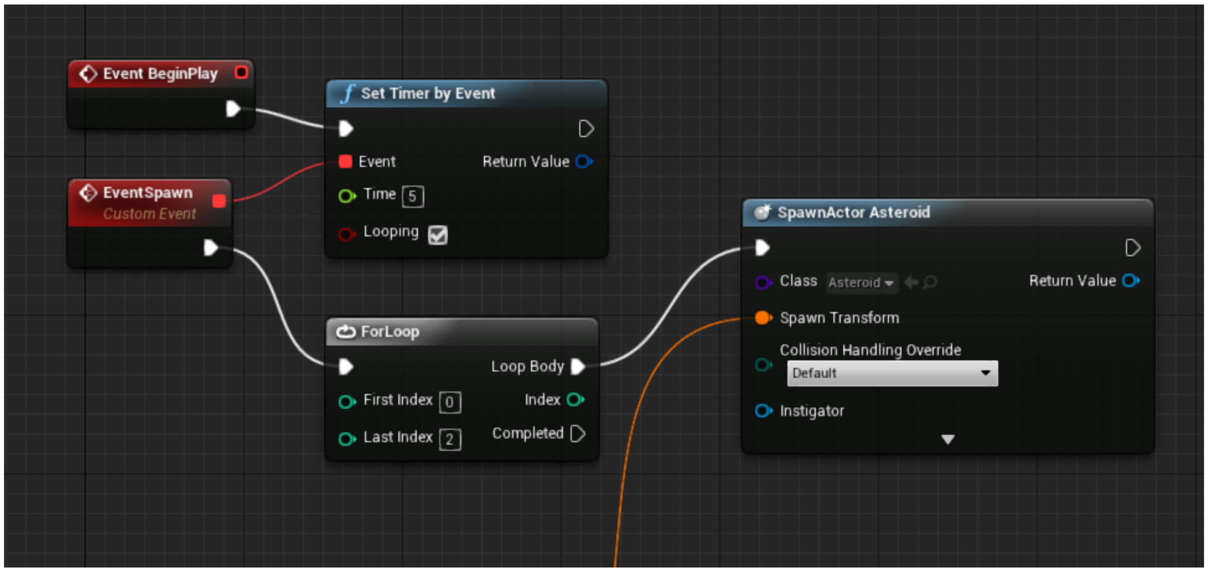
Now, let's make an actor to manage the score and spawn asteroids.

Create an Actor Blueprint named "Global". This is to parallel to the way it would be done in Unity (it is also possible to manage the score and spawn asteroids using a Level Blueprint).

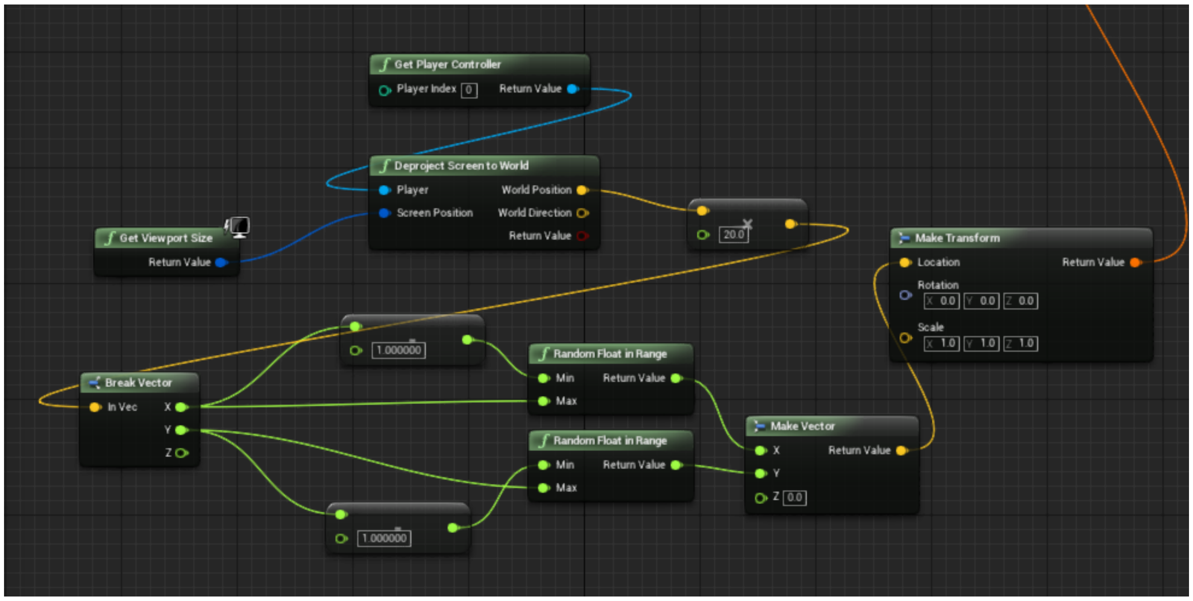
In order to get rid of the default sphere visual, we can create a Scene component and replace the DefaultSceneComponent. In the Event Graph, you can delete the Event ActorBeginOverlap and Event Tick nodes.

Right click on the graph and search for "Add Custom Event". Select this option, and name your event "EventSpawn". We will use this event to trigger spawning our asteroids in the level.

The "Timer by Event" node calls our EventSpawn custom event every 5 seconds on a loop. The ForLoop node is another Flow Control node which has the same functionality as a for loop in code. Add the following:

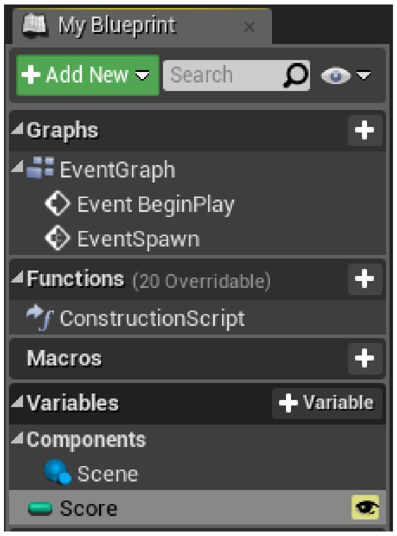


For the spawn transform, one way to do this is to pick three random points on the screen and project these points from screen space to world space. If you wire up the nodes shown below this will implement this:

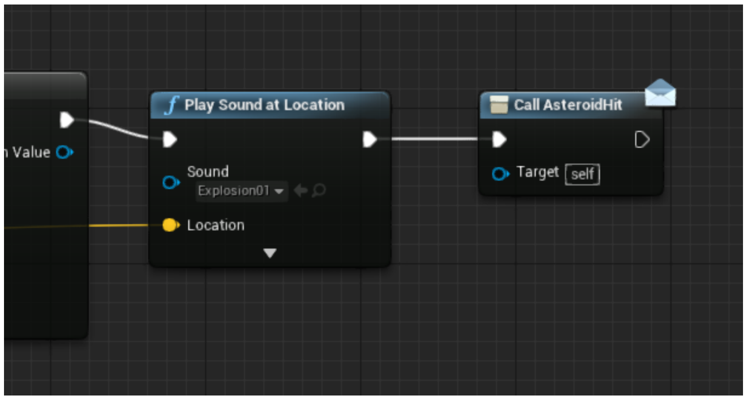


However, please note, this approach requires some fine tuning of the camera position to avoid "self-collision" with the asteroids. If you like, you can create your own spawning pattern for the asteroids.

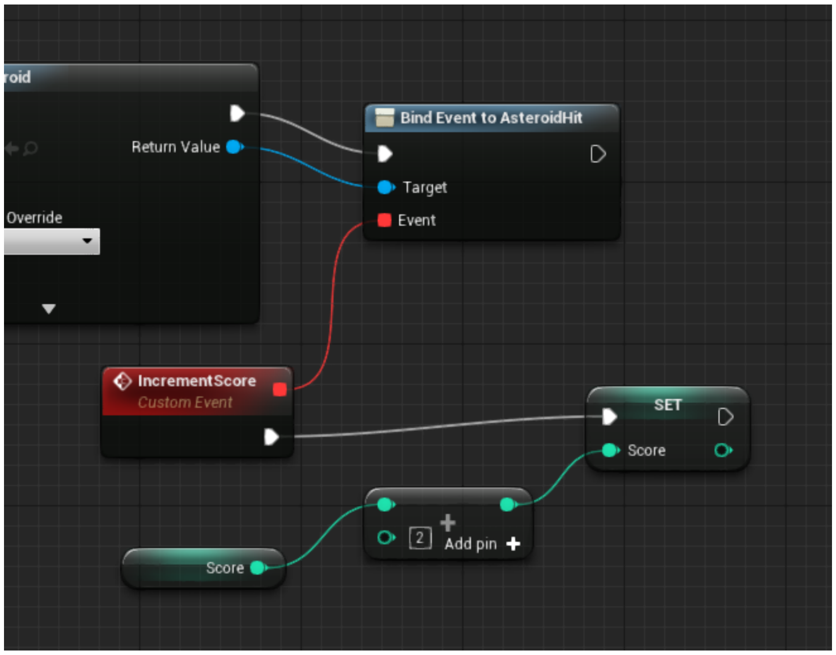
In order to keep track of the score, select the plus button in the "Variables" tab in the My Blueprint panel. Name your new variable "Score", and change the variable type to Integer in the Details panel. In addition, click the eye icon next to the variable name to make this variable public.



Now, let's have destroyed asteroids increase our score. In the My Blueprint tab of Asteroid, create a new Event Dispatcher named "AsteroidHit". Drag this dispatcher into your graph and select the "Call" option. Connect this node to the previous last node in the execution chain.



In Global, drag the Return Value pin out from the SpawnActor node and create a “Bind Event to AsteroidHit” node. Create an IncrementScore event, and connect it to this node. Use the “Get Score” and “Set Score” nodes to create the following:



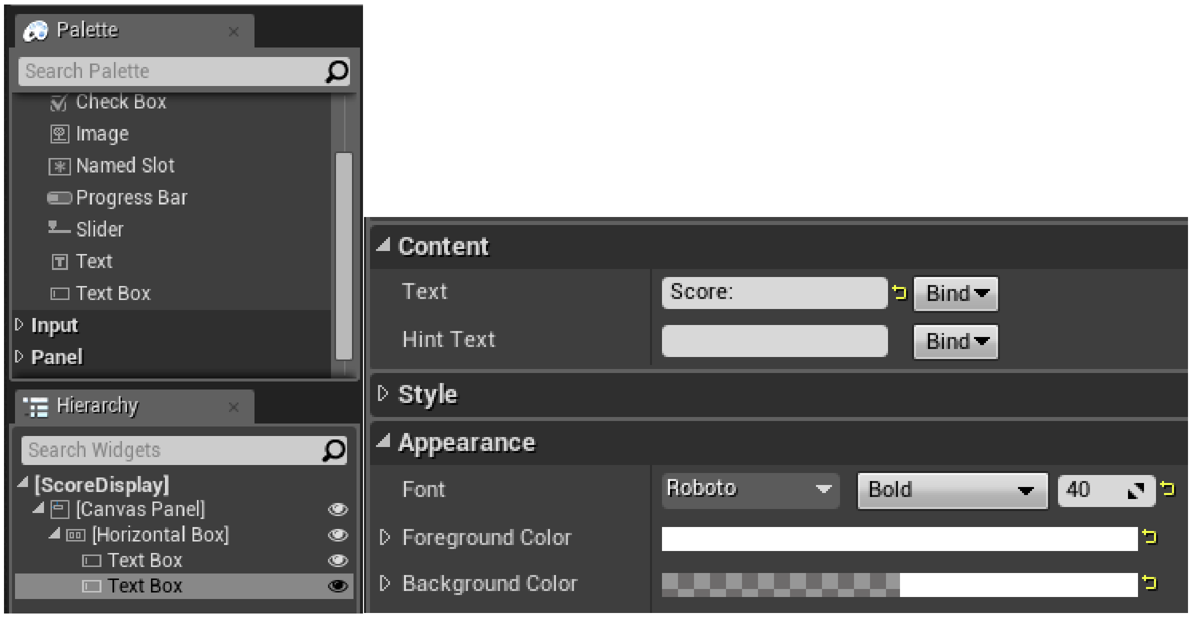
12. DISPLAY THE SCORE.

Let’s set up a Widget to display our score.

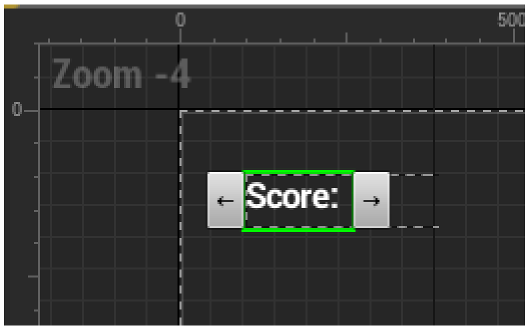
Select “Add New -> User Interface -> Widget Blueprint”. Name this widget “ScoreDisplay”. Widgets blueprints in Unreal are used to create and lay out UI elements in your game, also known as widgets. Double click on this file to open it in the Widget Blueprint Editor. The layout of the display can be seen in the Hierarchy panel. The widget blueprint starts off with a Canvas Panel widget.

In the Palette, search for the “Horizontal Box” and drag it into the top left of your Canvas Panel. Next, drag in a “Text Box” widget into this box. It should be a child of your Horizontal Box widget.

In the Text Box, type “Score: ” into the Text field in the Details panel. In the Appearance tab, change the font to 40, change the foreground color to white, and make the background color transparent (Set the Alpha channel to 0). Select Compile in the Menu Bar, and you should see your updated text.



Text: Score:
Hint Text:
Font: Roboto Bold 40

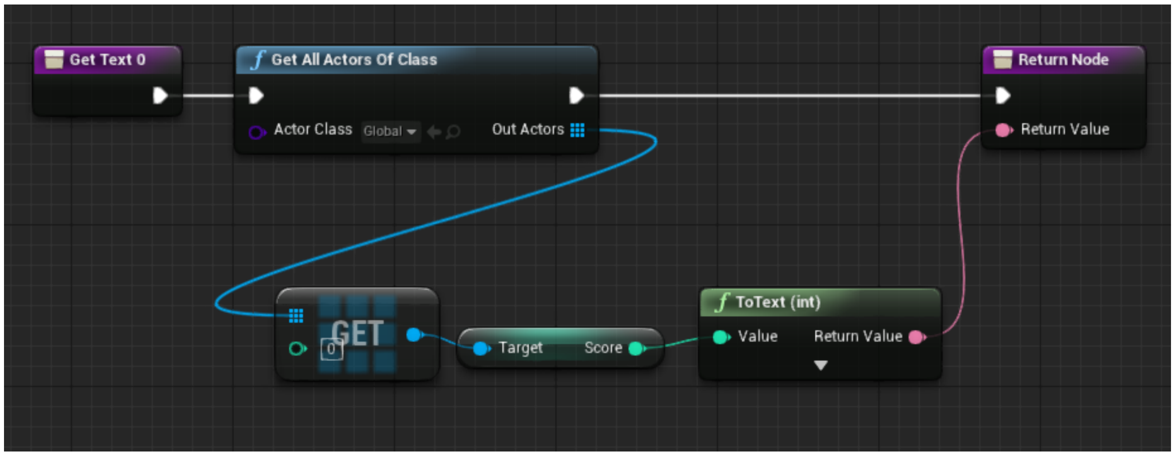


Copy and paste this Text Box into the same level of the hierarchy as the first Text Box, in order to keep our Appearance settings. Erase the text field in the Content tab. We will now plug our Score variable into this Text Box.

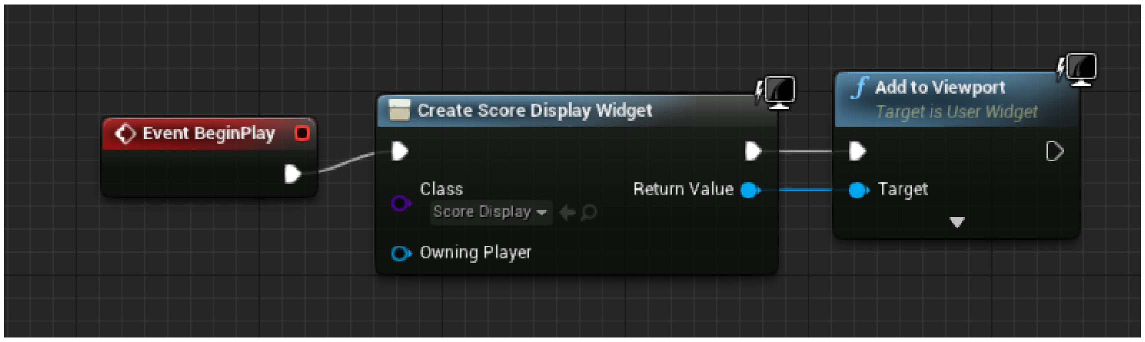
Select the drop down to the right of the Text field, and select “Create Binding”. This should send you to the Graph editor. Now, we can retrieve our Score variable to use.

Right click in the graph and search for the “Get All Actors Of Class” node. Set the Actor Class to our Global class, and drag the “Out Actors” pin out and search for the “Get” node. Create one and leave the Index value at 0. This will search through our level for Global objects, and choose the first (and only) one. Drag out the output pin from the Get node, and search for the “Get Score” node. Drag the output from this node and place a “ToText(int)” node. Finally, connect the Return Value from this node to the ReturnNode, and connect the Exec pins in the “Get All Actors Of Class” node and the ReturnNode.

Your node graph should look something like this.

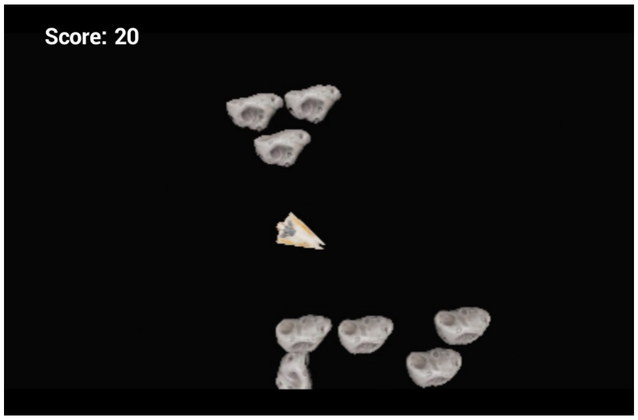


Finally, to add this Widget Blueprint to our level, we have to open the Level Blueprint. Each level has a Level Blueprint, which has its own node graph which gets executed during gameplay. This blueprint can be accessed in the main Menu Bar, in “Blueprints -> Open Level Blueprint”.



You should see an “Event BeginPlay” node and an “Event Tick” node. We want to connect our new nodes to the BeginPlay node, because we only want to load our display at the very beginning. Right click on the graph and search for “Create Widget”. Create and connect this node, and select the “ScoreDisplay” class in the Class field. Drag the Return Value pin out from this node and create an “Add to Viewport” node. The pins should automatically be connected.

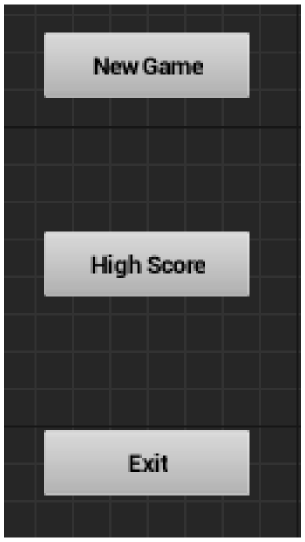
When you play the game now, you should see your counter on the top left of the screen, and it should increment by 10 every time you destroy an asteroid.



13. CREATE A MENU SCREEN.

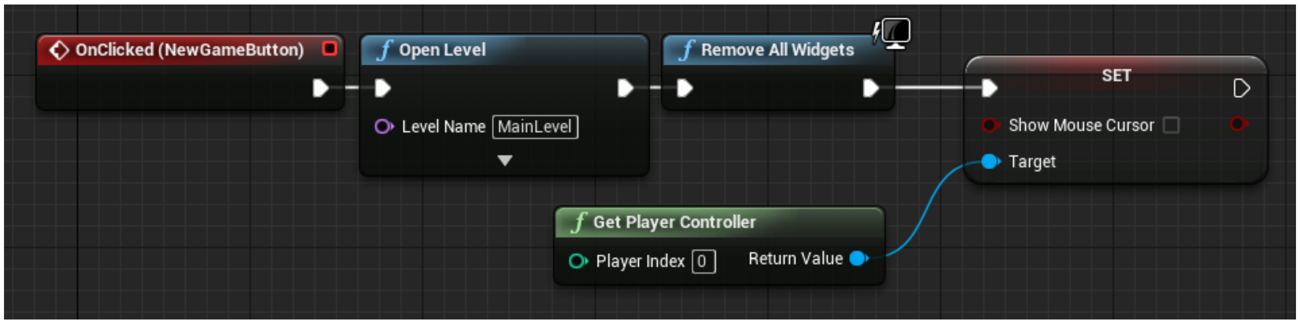
Finally, lets create our main menu screen. Create a new empty level and name it “StartMenu”. In addition, create a Widget Blueprint named “Menu” and open it in the Widget Blueprint Editor.

Create three evenly spaced Button Widgets, and drag a Text Widget onto each of them to name them “New Game”, “High Score”, and “Exit”, respectively. In the Details panel name your Button Widgets “NewGameButton”, “HighScoreButton”, and “ExitButton”.

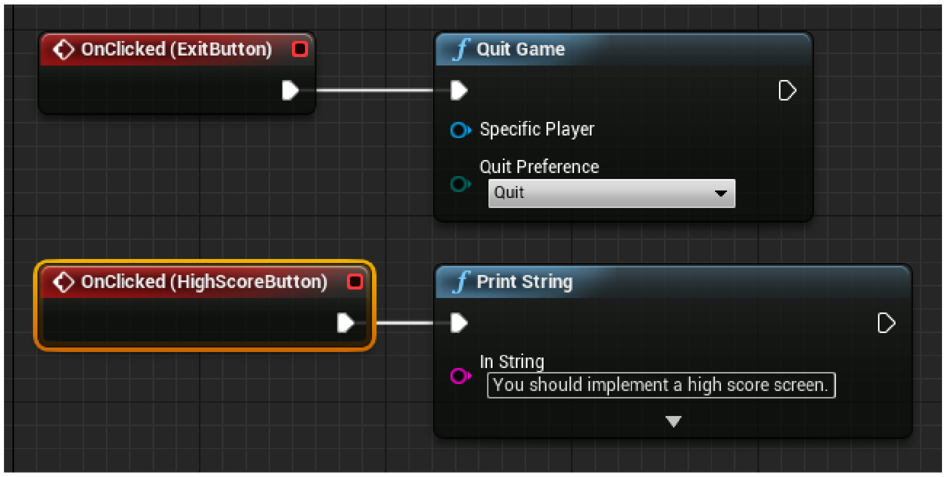


Click on the New Game button and scroll to the bottom of the Details panel. In the Events tab, select the green box next to “OnClicked”. This will bring you to an event in the Event Graph, which will get called whenever this button is clicked.

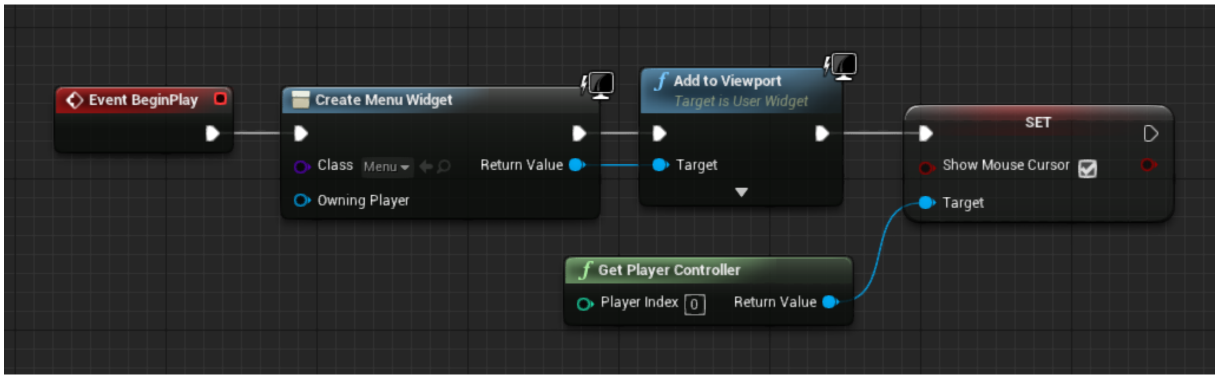
Create the following nodes. The “Set Show Mouse Cursor” node at the right can be created by creating a “Get Player Controller” node and dragging out from the Return Value pin.



In the “Level Name” input parameter of “Open Level”, please change this to the full name of your gameplay level.
Repeat this process with the other two buttons, attaching the following nodes instead:



Finally, open the Level Blueprint for StartMenu and add the following nodes.

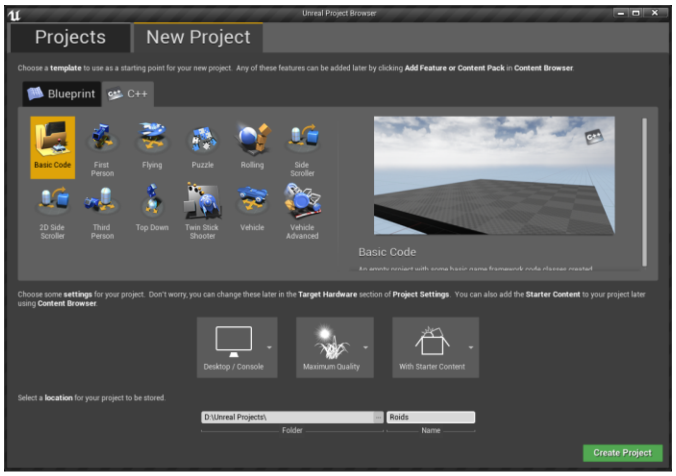


You should now be able to start your game from your Start Menu!

Part 2: C++

1. MAKE A NEW PROJECT.

Click on the “New Project” tab, followed by the “C++” tab. Create a “Basic Code” project with the default settings, named “Roids”. Be sure to select “With Starter Content”, because we are going to use some of the assets it provides.



Choose a **template** to use as a starting point for your new project. Any of these features can be added later by clicking **Add Feature or Content Pack** in **Content Browser**.

Basic Code

Choose some **settings** for your project. Don't worry, you can change these later in the **Target Hardware** section of **Project Settings**. You can also add the **Starter Content** to your project later using **Content Browser**.

Select a **location** for your project to be stored.

Folder: D:\Unreal Projects\

Name: Roids

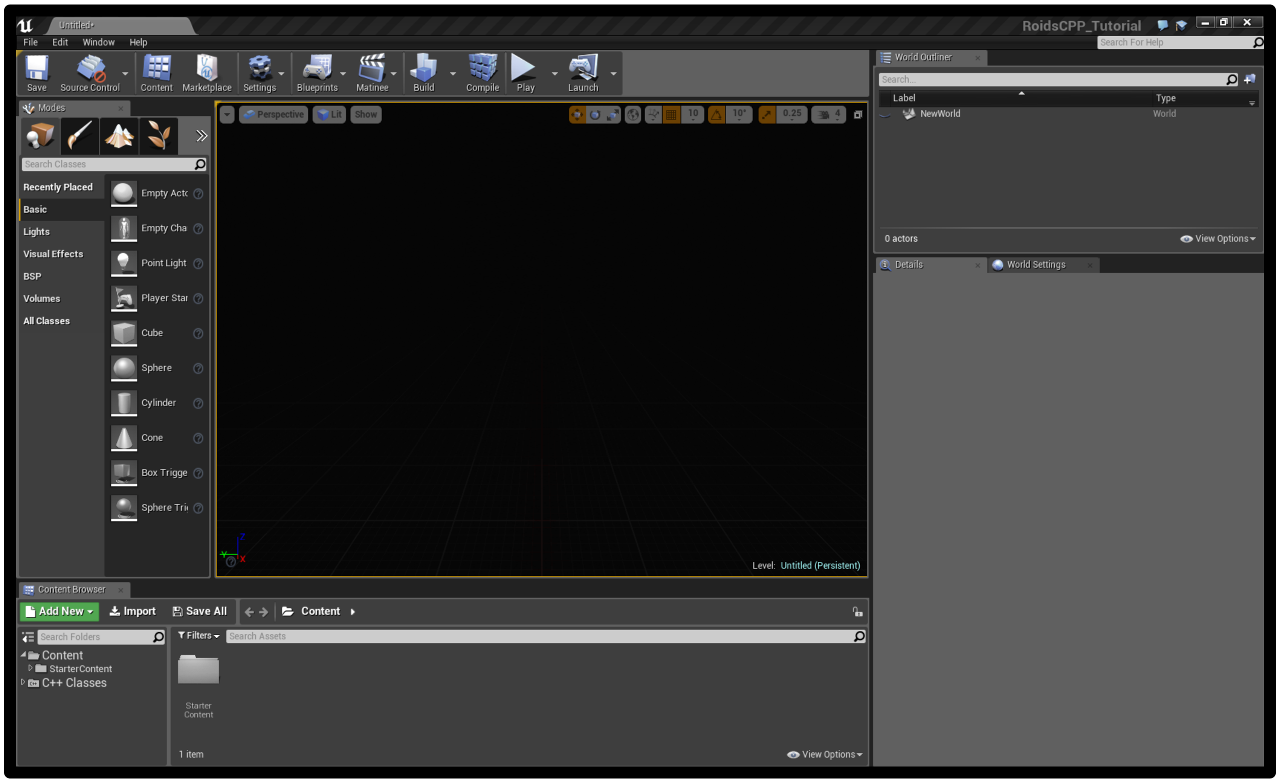
This should also open up Visual Studio 2015 when the project is created. If you don’t have Visual Studio 2015, you can download a copy here: https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx.

The engine will open your project and the Minimal_Default level from the starter content.

2. CREATE AN EMPTY LEVEL.

Currently, the default level has a number of assets which we don’t need. We can create our own empty level, to which we can add our own assets.

Select “File->New Level”, and select “Empty Level” from the popup window. Select “File->Save” to save the level, and name the level “MainLevel”.



3. IMPORT SOME ASSETS.

Currently, our project doesn't have any custom Assets. Let's add our ship and asteroid assets.

Unzip the tutorial file "Unity Tutorial Assets.rar" from Canvas.

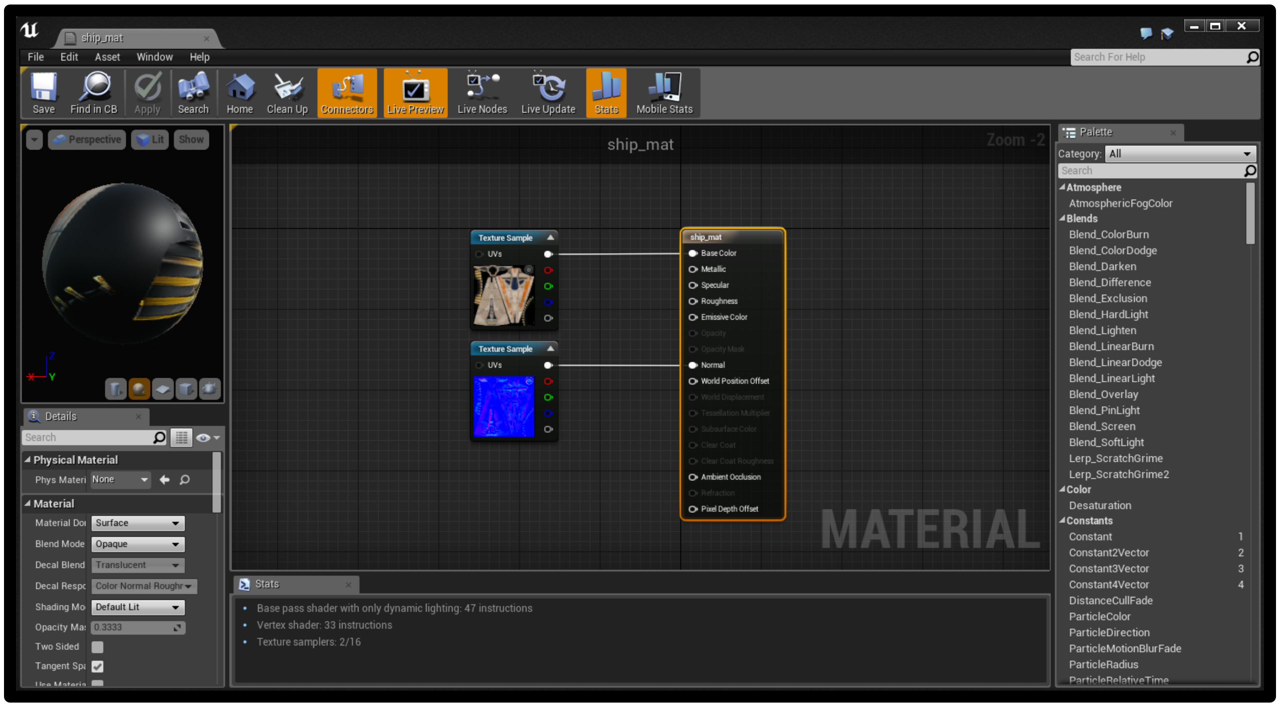
Drag in the "prop_asteroid_01.fbx" and "vehicle_playerShip.fbx" files from the Models folder into the Content Browser. Uncheck the boxes under the "Mesh" and "Material" groups and select "Import All". Clear any messages that pop up.

Next, drag in all the .png files from the Textures folder. We will use these to create the materials for the ship and asteroid.

4. CREATE SOME MATERIALS

Select "Add New -> Material" in the Content Browser, name the material "ship_mat", and double click on the new material. This window is the Material Editor. Similar to node graphs in other 3D packages, we can use this window to hook up nodes to create our material properties. You can use right click to move around the node graph, and left click to drag and link nodes together.

Drag the "vehicle_playerShip_orange_dff" and "vehicle_playerShip_orange_nrm" textures into the Material Editor, and connect their white pins to the "Base Color" and "Normal" pins. Select "Apply" on the Menu bar to apply these changes.



Physical Material

Phys Materi	None

Material

Material Doi	Surface
Blend Mode	Opaque
Decal Blend	Translucent
Decal Respc	Color Normal Roughr
Shading Mo	Default Lit
Opacity Ma:	0.3333
Two Sided	
Tangent Spa	✓

Stats

- Base pass shader with only dynamic lighting: 47 instructions
- Vertex shader: 33 instructions
- Texture samplers: 2/16

Repeat this process with the asteroid textures to create a material called “asteroid_mat”.

5. CREATE AN ACTOR.

Similar to GameObjects in Unity, any object which can be placed in a level in Unreal is an Actor.

Just like GameObjects, Actors can contain Components which control different aspects of the Actor’s functionality. These Components are arranged in a tree hierarchy.

In contrast to GameObjects, Actors do not store their Transform data – this is stored in the Root Component, which is the root node of the Component tree.

Pawns are the base class of all Actors that can be controlled by players or AI. We want our ship to be derive from the Pawn class.

Select “Add New -> New C++ Class”, and select the Pawn class in the menu. Name your class “Ship”, and select “Create Class” – don’t make your class Public or Private.

Once the class is created, “Ship.cpp” and “Ship.h” files should open in Visual Studio. Your actual Actor classes will begin with an “A”, which stands for Actor (i.e. AShip, ABullet, etc.). These C++ files will contain the logic for the ship.

In your “Ship.h” file, add the following lines that are in **BOLD**:

Important Note: all the extra header info added in a header file(*.h) should be added immediately after *“#pragma once”* and before *#include "CoreMinimal.h"*

```
    …

    #include "Components/SphereComponent.h"

    …

    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent* InputComponent) override;

    USphereComponent* ShipSphereComponent;

    …
```

This line declares our sphere collision component, also known as a USphereComponent.

Similar to Unity’s Colliders and Rigid Body components, Collision Components in Unreal manage the physics and collision detection for your Actor.

In “Ship.cpp”, add the following:

```
AShip::AShip()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you
don't need it.
    PrimaryActorTick.bCanEverTick = true;

    // Set this pawn to be controlled by the lowest-numbered player
    AutoPossessPlayer = EAutoReceiveInput::Player0;

    // Our root component will be a sphere that reacts to physics
    ShipSphereComponent =
CreateDefaultSubobject<USphereComponent>(TEXT("RootComponent"));
    RootComponent = ShipSphereComponent;
    ShipSphereComponent->InitSphereRadius(10.0f);
    ShipSphereComponent->SetCollisionProfileName(TEXT("Pawn"));
    ShipSphereComponent->SetSimulatePhysics(true);
    ShipSphereComponent->SetEnableGravity(false);
    ShipSphereComponent->SetLinearDamping(0.3);
    ShipSphereComponent->SetAngularDamping(1);
    ShipSphereComponent->SetConstraintMode(EDOFMode::XYPlane);
}
```

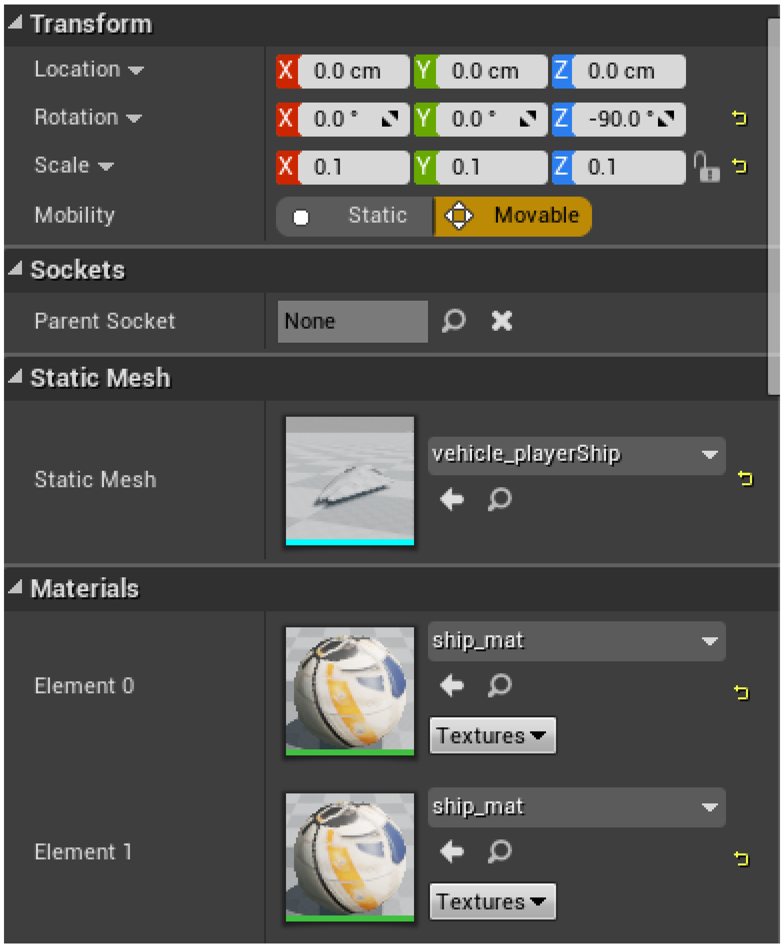
These lines will load and set up the collision component for the ship. Build your project in Visual Studio to apply these updates in Unreal.

Note: You may or may not have VS2015 generate a warning regarding Unreal Macros. This is caused by a not-fully referenced header. If that is the case, just try to compile the code directly in UE4 first.

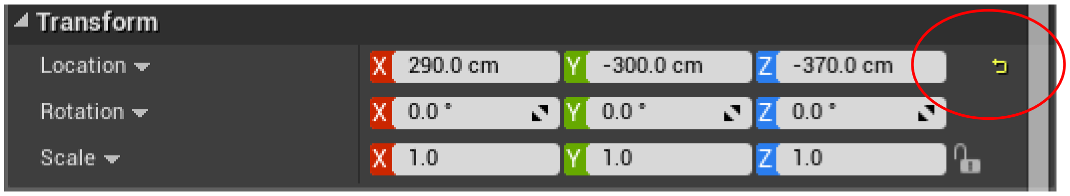
6. MAKE A BLUEPRINT CLASS FOR THE SHIP.

Select “Add New -> Blueprint Class” and search in the “All Classes” tab for the **Ship** class that we’ve just created. Name this new class ShipBP, and double click on this blueprint class.

Add a Static Mesh component named “ShipMesh” to the blueprint. Set the mesh to “vehicle_playerShip”, and set both of the materials to “ship_mat”. In addition, set the rotation of the ship to (0.0, 0.0, -90.0) since the default **forward** axis for unreal is **positive-X**, and the scale to (0.1, 0.1, 0.1).


Transform
Location X 0.0 cm Y 0.0 cm Z 0.0 cm
Rotation X 0.0 ° Y 0.0 ° Z -90.0 °
Scale X 0.1 Y 0.1 Z 0.1
Mobility Static [✓ Movable]
Sockets
Parent Socket None
Static Mesh
Static Mesh vehicle_playerShip
Materials
Element 0 ship_mat
Element 1 ship_mat

You can add a ship to our level by dragging it from the Content Browser into the viewport. You can center it at the origin by clicking the "Reset to Default" yellow arrow next to "Transform->Location" in the Details Panel.



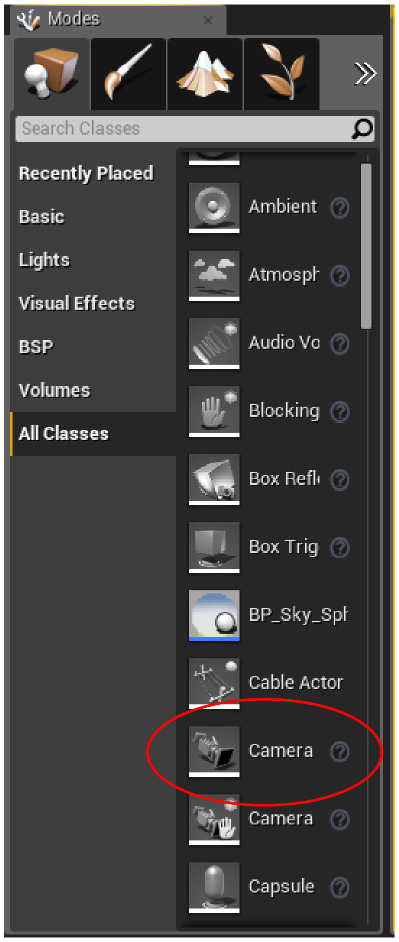
Transform

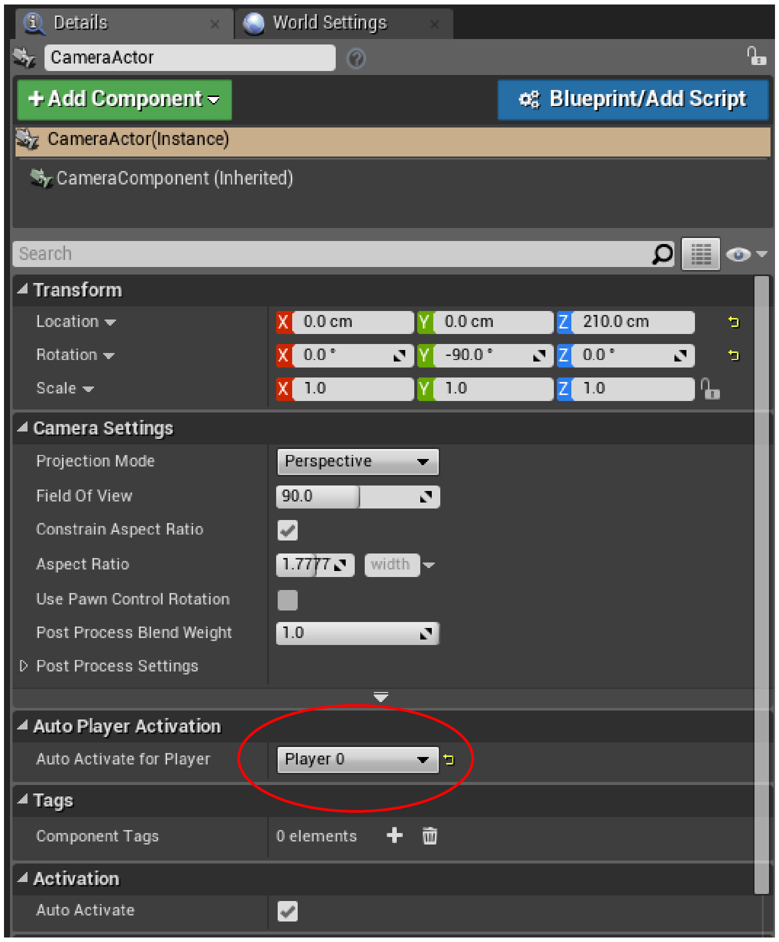
Location	X 290.0 cm	Y -300.0 cm	Z -370.0 cm
Rotation	X 0.0 °	Y 0.0 °	Z 0.0 °
Scale	X 1.0	Y 1.0	Z 1.0

7. ADD A CAMERA AND LIGHT TO THE SCENE.

Right now, if you play the scene, you will be spawned at the origin along with your ship. We need to set up the camera above the ship, so we can start off with a view of the field.

In the "Modes" panel, find "All Classes->Camera" and drag one into the viewport. Change the Location to (0.0, 0.0, 210.0) and the Rotation to (0.0, -90.0, 0.0). In addition, in the "Auto Activate for Player" dropdown, select "Player 0".





Transform

Location	X 0.0 cm	Y 0.0 cm	Z 210.0 cm
Rotation	X 0.0 °	Y -90.0 °	Z 0.0 °
Scale	X 1.0	Y 1.0	Z 1.0

Camera Settings

Projection Mode	Perspective
Field Of View	90.0
Constrain Aspect Ratio	✓
Aspect Ratio	1.7777 width
Use Pawn Control Rotation	
Post Process Blend Weight	1.0
Post Process Settings	

Auto Player Activation

Auto Activate for Player	Player 0

Tags

Component Tags	0 elements

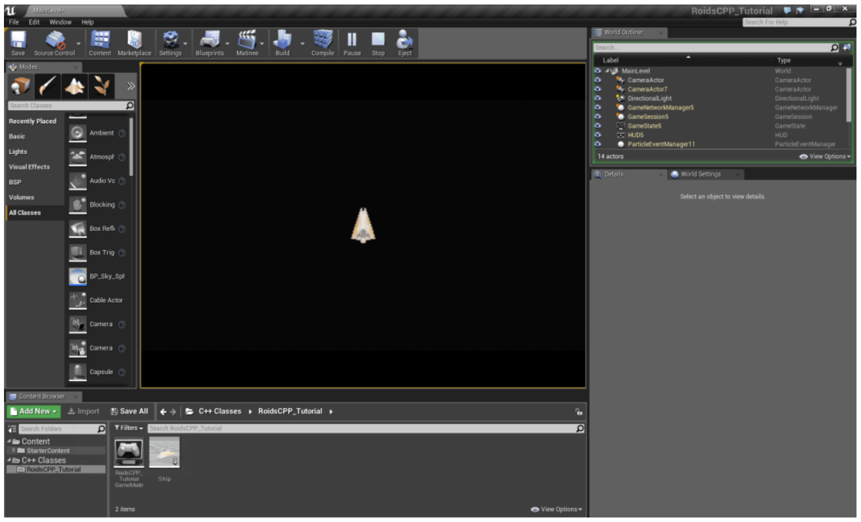
Activation

Auto Activate	✓

We also need a light in order to see anything when we play. Navigate to the “Lights” tab and drag a Directional Light into the scene. Rotate the light using the channels until you are satisfied with the lighting on the ship.

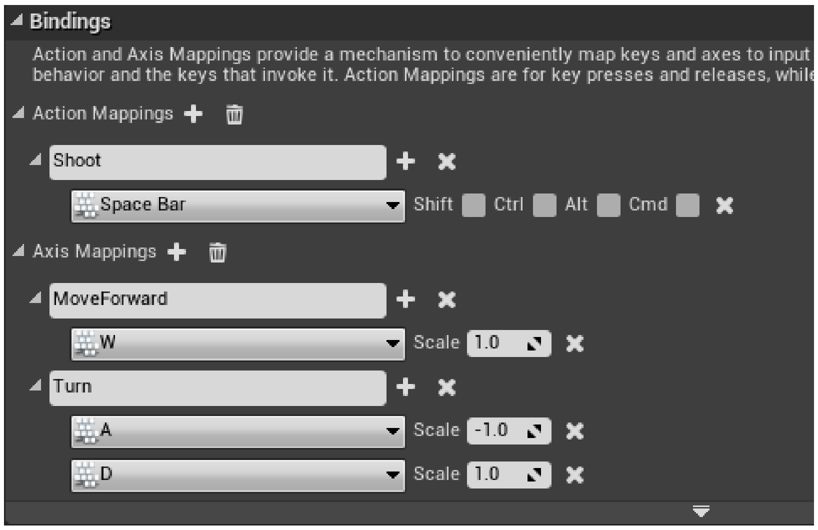
8. RUN IT!

Click on the Play button to run your game. Your textured ship should now be sitting in the center of the viewport. By default, your mouse pointer may be locked in the viewport. If that is the case, use the “Shift+F1” hotkey to unlock the mouse pointer.



9. ADD MOVEMENT TO YOUR SHIP.

To add movement, we first need to set up the player input for the project. Go to “Edit -> Project Settings”, and select the “Input” option on the left menu. You should see a page where you can edit the key bindings for your game. Use the plus arrows to add mappings and keys, and use the dropdown menus to find specific keys. Customize your bindings to the following arrangement:



Bindings

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while

Action Mappings

Shoot

Space Bar Shift Ctrl Alt Cmd

Axis Mappings

MoveForward

W Scale 1.0

Turn

A Scale -1.0

D Scale 1.0

In your Ship.h file, add the following functions which will determine the functions that get executed when you move forward or turn.

```
	…

	USphereComponent* ShipSphereComponent;

	//Input functions
	void Move_Forward(float AxisValue);
	void Move_Turn(float AxisValue);

	...
```

In Ship.cpp, add the function implementations referenced in the headers including:

```
…

#include "Components/InputComponent.h"

…

// Called to bind functionality to input
void AShip::SetupPlayerInputComponent(class UInputComponent* InputComponent)
{
	Super::SetupPlayerInputComponent(InputComponent);

	InputComponent->BindAxis("MoveForward", this, &AShip::Move_Forward);
	InputComponent->BindAxis("Turn", this, &AShip::Move_Turn);
}

void AShip::Move_Forward(float AxisValue)
{
	if (AxisValue > 0)
	{
		ShipSphereComponent->AddForce(GetActorForwardVector() * AxisValue * 600);
	}
}

void AShip::Move_Turn(float AxisValue)
{
	FRotator NewRotation = GetActorRotation();
	NewRotation.Yaw += AxisValue * 2;
	SetActorRotation(NewRotation);
}
```

If you hit play now, you should be able to control your ship using the W, A, and D keys.

Note: the forward movement speed may or may not be a little too slow. If that isthe case reduce the size of the ship mesh, or increase the value of 600 in line:

```
	ShipSphereComponent->AddForce(GetActorForwardVector() * AxisValue * 600);
```

10. ADD A BULLET ACTOR.

Now, we can create a bullet for our ship to shoot.

Create a new C++ actor named "Bullet".

Add the following lines to Bullet.h:

```
…

#include "Components/SphereComponent.h"

#include "GameFramework/ProjectileMovementComponent.h"

…

public:
….

        // Sphere collision component
        USphereComponent* CollisionComp;

        // Projectile movement component
        UProjectileMovementComponent* ProjectileMovement;
```

Also change the constructor function to:

```
        ABullet(const FObjectInitializer& ObjectInitializer);
```

Replace the constructor in Bullet.cpp with the following:

```
ABullet::ABullet(const FObjectInitializer& ObjectInitializer) : Super(ObjectInitializer)
{
        //  Set this actor to call Tick() every frame.  You can turn this off to improve performance if you
        //   don't need it.

        PrimaryActorTick.bCanEverTick = true;

        CollisionComp = ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this,
TEXT("SphereComp"));
        CollisionComp->InitSphereRadius(0.3f);
        CollisionComp->SetSimulatePhysics(false);
        CollisionComp->SetEnableGravity(false);
        CollisionComp->SetNotifyRigidBodyCollision(true);
        CollisionComp->SetCollisionProfileName(TEXT("OverlapAll"));
        RootComponent = CollisionComp;
        // Use a ProjectileMovementComponent to govern this projectile's movement

ProjectileMovement =
ObjectInitializer.CreateDefaultSubobject<UProjectileMovementComponent>(this,
TEXT("ProjectileComp"));
        ProjectileMovement->UpdatedComponent = CollisionComp;
        ProjectileMovement->InitialSpeed = 100.f;
        ProjectileMovement->MaxSpeed = 100.f;
        ProjectileMovement->bRotationFollowsVelocity = true;
        ProjectileMovement->bShouldBounce = true;
        ProjectileMovement->ProjectileGravityScale = 0.0f;
}
```

This will create a Bullet with collision and projectile components. When this bullet is spawned, it will start off with an initial speed of 100.

In order to spawn the Bullet we created in C++ from the Ship, create a blueprint class that extends the Bullet class, and name your blueprint “BulletBP”. Under the root component of the blueprint, create a Sphere (Static Mesh Component) named “BulletMesh” and scale it to (0.08, 0.08, 0.08).

Now, we can spawn these bullets from your ship.
Add the following block to your Ship.h file:

```
        …
        //Input functions
        void Move_Forward(float AxisValue);
        void Move_Turn(float AxisValue);
        void Shoot();

        // Projectile class to spawn
        UPROPERTY(EditDefaultsOnly, Category = Projectile)
        TSubclassOf<class ABullet> ProjectileClass;
        …
```

as well as an **#include “Bullet.h”** statement at the top of the file.

This will define what kind of object you are spawning from your Ship.

In Ship.cpp, we need to first find the Bullet blueprint which we are going to spawn. Add this code to the bottom of your constructor, also this header in *.cpp file:

```
….
#include "UObject/ConstructorHelpers.h"
…

static ConstructorHelpers::FObjectFinder<UBlueprint>
Bullet(TEXT("Blueprint'/Game/BulletBP.BulletBP'"));
if (Bullet.Object) {
        ProjectileClass = (UClass*)Bullet.Object->GeneratedClass;
}
```

Note: if you didn't name Bullet Blueprint as "BulletBP", replace the string name in the constructor with:

```
"Blueprint'/Game/theName.theName'"
```

where *theName* is the name you used (do not forget all the quotation marks).

We also need to bind the Shoot input to the Shoot function we are going to make. At the end of SetupPlayerInputComponent, add:

```
InputComponent->BindAction("Shoot", EInputEvent::IE_Pressed, this, &AShip::Shoot);
```

and add the following Shoot function to the file along with the line:

```
….
#include "Engine/World.h"
….


void AShip::Shoot(){
	UWorld* const World = GetWorld();
	if (World){
		FActorSpawnParameters SpawnParams;
		SpawnParams.Owner = this;
		SpawnParams.Instigator = Instigator;
		World->SpawnActor<ABullet>(ProjectileClass, GetActorLocation() +
GetActorForwardVector() * 15, GetActorRotation(), SpawnParams);
	}
}
```

This Shoot function first obtains a reference to the world, which is necessary to call the SpawnActor function. The SpawnActor function then creates an instance of a Bullet object, 15 units in front of the Ship.

If you view the World Outliner while you play, you will see that Bullet instances will appear under the MainLevel hierarchy, similar to Prefabs in Unity. However, we can instantiate classes directly instead of creating a special Prefab object in Unreal.

If you play now, you should be able to shoot bullets from the ship by pressing the spacebar.

11. ADD AN ASTEROID ACTOR.

We can now create some asteroids for us to shoot. First, create a C++ Actor named Asteroid.

Add an "**#include "Bullet.h"**" statement to the Asteroid.h file. We want to destroy both the Asteroid and Bullet when they collide.

In the body of the header file, add the following code and #includes:

```
	…
	#include "Components/BoxComponent.h"
	#include "Materials/Material.h"
	…

	// Called every frame
	virtual void Tick( float DeltaSeconds ) override;

	UBoxComponent* AsteroidBoxComponent;
	UMaterial* AsteroidMaterial;

	UFUNCTION()
	void onHit(AActor* SelfActor, class AActor* OtherActor, FVector NormalImpulse,
			const FHitResult& Hit);
}
```

We will call the onHit function when a collision is detected on our Root Component. Collisions will come with relevant information like the actor that was hit (OtherActor), the force that the actors collided with(NormalImpulse), and more(FHitResult).

In Asteroid.cpp, place the following code in the constructor, just like we did previously.

```
	…
	// Set this actor to call Tick() every frame.  You can turn this off to improve performance if you
don't need it.
	PrimaryActorTick.bCanEverTick = true;

// Our root component will be a box that reacts to physics
	AsteroidBoxComponent =
CreateDefaultSubobject<UBoxComponent>(TEXT("RootComponent"));
	RootComponent = AsteroidBoxComponent;
	AsteroidBoxComponent->InitBoxExtent(FVector(12.0f, 15.0f, 12.0f));
	AsteroidBoxComponent->SetCollisionProfileName(TEXT("BlockAllDynamic"));
	AsteroidBoxComponent->SetSimulatePhysics(true);
	AsteroidBoxComponent->SetEnableGravity(false);
	AsteroidBoxComponent->SetNotifyRigidBodyCollision(true);
	…
```

Then add the following code in “AAsteroid::BeginPlay()” after “Super::BeginPlay();”

```
	…
	OnActorHit.AddDynamic(this, &AAsteroid::onHit);
	…
```

The “**OnActorHit.AddDynamic(this, &AAsteroid::onHit);**” function is used to call the onHit function when the Root Component of the Actor detects a collision. OnActorHit is on used when you want to detect collisions on the root of the Actor. If you want to detect collision on one of the children components, you will have to use the OnComponentHit function.

Finally, add the code for the onHit function. Right now, we simply destroy the Asteroid and the colliding Bullet. In Unity, we used Tags to check the type of the colliding GameObject. In Unreal, we can simply check the type of the colliding Actor using the "IsA" function.

```
…
if (OtherActor && (OtherActor != this) && OtherActor->IsA(ABullet::StaticClass()))
{
        Destroy();
        OtherActor->Destroy();
}
…
```

Now, create a Blueprint class which extends your Asteroid class and name it "AsteroidBP". Give it a mesh as we did with the ship, with the asteroid materials and model, scaled by 0.25 in each axis. To test our collision, drag the AsteroidBP into the level. Make **sure it has the same Z coordinate as your Ship**. When you shoot a bullet at the asteroid, both the bullet and the asteroid should disappear when they collide.

12. CREATE AN EXPLOSION.

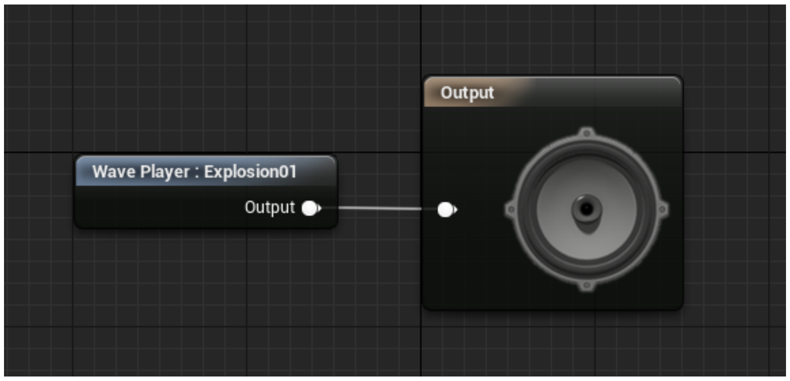
Create an Explosion Actor like we did in the Blueprints section (Part 1, step 9).

13. HAVE THE ASTEROID EXPLODE ON COLLISION.

Before adding the sound in code, we needed to add a SoundCue asset first.

In the Content Browser, select "Add New -> Sounds -> Sound Cue", and name your sound "ExplosionSound". Double click this file to open it in the Sound Cue Editor. You will see a node editor, similar to the Material Editor.

We only need one node for this graph. In the Palette on the right, search for "Wave Player" and drag one into the graph. Click on the Wave Player Node, and select the "Sound Wave" dropdown in the Details panel. Select the "Explosion01" Sound Wave. Connect the Wave Player node to the Output node and save. If you press Play in the menu bar, you should hear the explosion.



Now, we'll spawn our explosion and play our sound effect when the Asteroid is destroyed.

First add these includes in the header file:

```
…

#include "SoundDefinitions.h"
#include "Sound/SoundCue.h"
…

Public:

…

TSubclassOf<class AActor> Explosion;
USoundCue* explosionSoundCue;
…
```

We will reference our explosion class in order to spawn it, in the same way that we spawned bullets from the Ship.

For includes, add this in cpp:

```
…
#include "UObject/ConstructorHelpers.h"
…
```

At the bottom of the Asteroid constructor, add:

```
static ConstructorHelpers::FObjectFinder<UBlueprint>
FindExplosion(TEXT("Blueprint'/Game/ExplosionBP.ExplosionBP'"));
	if (FindExplosion.Object) {
		Explosion = (UClass*)FindExplosion.Object->GeneratedClass;
	}

	static ConstructorHelpers::FObjectFinder<USoundCue>
explosionSound(TEXT("SoundCue'/Game/ExplosionSound.ExplosionSound'"));
	if (explosionSound.Object != NULL)
	{
		explosionSoundCue = (USoundCue*)explosionSound.Object;
	}
```

In the onHit function, add the following code, which will spawn our explosion at the asteroid location and play the sound effect.

```
void AAsteroid::onHit(AActor* SelfActor, class AActor* OtherActor, FVector NormalImpulse, const
FHitResult& Hit){
if (OtherActor && (OtherActor != this) && OtherActor->IsA(ABullet::StaticClass()))
```

```
    {
        Destroy();
        OtherActor->Destroy();
        UWorld* const World = GetWorld();
        if (World){
            FActorSpawnParameters SpawnParams;
            SpawnParams.Owner = this;
            SpawnParams.Instigator = Instigator;
            World->SpawnActor<AActor>(Explosion, GetActorLocation(),
                                      GetActorRotation(), SpawnParams);

            UGameplayStatics::PlaySoundAtLocation(World, explosionSoundCue,
                                                  GetActorLocation());
        }
    }
}
```

If you play your level, the asteroid should now explode and play a sound when it is hit.

Also note: you may or may not have to change the name of your SoundCue and Explosion object to reflect your implementation.

14. CREATE A GLOBAL ACTOR.

Similar to how we used an empty GameObject in Unity to manage the score and spawn asteroids in Unity, we will do the same here using an Actor with no visuals.

Create a new C++ actor named "Global".

In the Global.h file, add an **#include "Asteroid.h"** statement.

Declare the following in the body of the header:

```
    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

    TSubclassOf<class AAsteroid> AsteroidClass;
    void SpawnAsteroids();

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Global")
    int32 Score;

    FTimerHandle timerHandle;

};
```

The UPROPERTY line allows us to add additional information about the following variable (Score). The EditAnywhere statement allows us to edit the Score value outside of our C++ files, and the BlueprintReadWrite statement allows us to access the Score in Blueprints. The timer handle will be used to set up our timer to spawn the asteroids.

In the Global.cpp constructor, set the AsteroidClass:

For includes, add this in cpp:

```
…
#include "UObject/ConstructorHelpers.h"
#include "Kismet/GameplayStatics.h"…
```

```
AGlobal::AGlobal()
{
	// Set this actor to call Tick() every frame.  You can turn this off to improve performance if you
don't need it.
	PrimaryActorTick.bCanEverTick = true;
	static ConstructorHelpers::FObjectFinder<UBlueprint>
Asteroid(TEXT("Blueprint'/Game/AsteroidBP.AsteroidBP'"));
	if (Asteroid.Object) {
		AsteroidClass = (UClass*)Asteroid.Object->GeneratedClass;
	}
}
```

In the BeginPlay function, we can reset the score and also use the WorldTimerManager to manage when asteroids are spawned.

```
void AGlobal::BeginPlay()
{
	Super::BeginPlay();
	GetWorldTimerManager().SetTimer(timerHandle, this, &AGlobal::SpawnAsteroids, 5.0f, true);
	Score = 0;
}
```

In contrast to Unity, we do not need multiple variables to keep track of a certain time or countdown. We can use SetTimer to call a function (SpawnAsteroids) every 5 seconds, and have it loop (the last boolean parameter).

Add the SpawnAsteroids function as follows:

```
void AGlobal::SpawnAsteroids(){
	const FVector2D ViewportSize = FVector2D(GEngine->GameViewport->Viewport->GetSizeXY());
	UWorld* const World = GetWorld();
	if (World) {
	    for (int i = 0; i < 3; i++) {
		float random1 = (float)rand() / RAND_MAX;
		float random2 = (float)rand() / RAND_MAX;
		float xPos = ViewportSize[0] * (2 * random1 - 1);
		float yPos = ViewportSize[1] * (2 * random2 - 1);

		FVector worldLoc, worldDir;
		APlayerController* cam = UGameplayStatics::GetPlayerController(GetWorld(), 0);
			cam->DeprojectScreenPositionToWorld(xPos, yPos, worldLoc, worldDir);
```

```
                                FVector spawn = worldLoc + worldDir * (worldLoc.Z / 2);
                                spawn.Z = 0;

                                AAsteroid* roid = World->SpawnActor<AAsteroid>(AsteroidClass,
                                                                FVector(spawn.X, spawn.Y, 0.0f), FRotator(0.f));

                        }
                }
}
```

This function uses the same method we used in Unity to spawn the Asteroids in the level, by finding the boundaries of the viewport in world coordinates and projecting screen coordinates to the world.

Again, similar to the Blueprint version, this approach requires some fine tuning of the camera position to avoid "self-collision" with the asteroids. If you like, you can create your own spawning pattern for the asteroids.

The DeprojectScreenPositionToWorld function sets the worldLoc and worldDir variables, in which worldLoc is the position of the camera and worldDir is the direction along which our screen position was projected. We need to spawn our Asteroid down the worldDir vector, or else they will spawn directly on top of the camera. In this case, we used the height of the camera to calculate how far away we spawned the Asteroid.

Once you have set up your Global class, you can simply drag it from the Content Browser into the level. Delete the single Asteroid sitting in your level, and play. Three asteroids should now be randomly spawned every 5 seconds.

Also note: you may or may not have to change the name of your Asteroid object to reflect your implementation.

15. UPDATE THE SCORE.

Let's connect our Asteroid to our Global object so we can keep track of our score when an asteroid is destroyed.

In order to do so, we need to create a **dynamic delegate** object in our Asteroid class. Using delegates, you can dynamically bind to a member function of an arbitrary object, then call functions on the object, even if the caller does not know the object's type. We can connect our delegate to a function in Global, so that a function to increase the score in Global is called whenever an Asteroid is destroyed.

First, add the following line to Asteroid.h. *(Note: if Visual Studio reports a syntax error, just ignore it).*

```
    #include "Asteroid.generated.h"

    DECLARE_DYNAMIC_DELEGATE(FDelegate);

    UCLASS()
    class ROIDSCPP_TUTORIAL_API AAsteroid : public AActor
```

And the following to the end of Asteroid.h.

```
    TSubclassOf<class AActor> Explosion;
    USoundCue* explosionSoundCue;

    UPROPERTY(EditAnywhere)
    FDelegate HitDelegate;

};
```

This code declares a type of delegate named “FDelegate”, and declares an FDelegate named “HitDelegate” which we can use to execute other functions.

Now, in the onHit function in Asteroid.cpp, add the following line:

```
    …
    if (World){
        FActorSpawnParameters SpawnParams;
        SpawnParams.Owner = this;
        SpawnParams.Instigator = Instigator;
        World->SpawnActor<AActor>(Explosion, GetActorLocation(), GetActorRotation(),
SpawnParams);
        UGameplayStatics::PlaySoundAtLocation(World, explosionSoundCue,
GetActorLocation());

        HitDelegate.Execute();
    }
    …
```

Now, whenever the Asteroid is hit, it will execute any functions connected to our HitDelegate. Delegates are especially useful if you want to call a number of different functions from different classes from one event.

Next, we have to connect our Global class to our spawned Asteroids.

In Global.h, add the following function which will increment our score:

```
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Global")
    int32 Score;

    FTimerHandle timerHandle;

    UFUNCTION()
    void incrementScore();

};
```

Make sure that the UFUNCTION() and UPROPERTY() lines are included, or Unreal may crash.

In Global.cpp, add these lines:

```
                FVector spawn = worldLoc + worldDir * (worldLoc.Z / 2);
                spawn.Z = 0;

                AAsteroid* roid = World->SpawnActor<AAsteroid>(AsteroidClass,
            FVector(spawn.X, spawn.Y, 0.0f), FRotator(0.f));

                roid->HitDelegate.BindDynamic(this, &AGlobal::incrementScore);

            }

    }

}

void AGlobal::incrementScore(){

    Score += 10;

}
```

The BindDynamic function binds our incrementScore function to our HitDelegate delegate. Whenever HitDelegate.Execute() is called, all the functions bound to HitDelegate will also be called.

If you play now and select your Global object, you should see the Score counter go up by 10 every time you destroy an asteroid.

Note: the implementation above only allows the Asteroid spawned by Global to be valid; any “manually placed” Asteroids will not work anymore (they can even crash the game if hit by bullet). Try to think of why and come up with a more robust implementation.

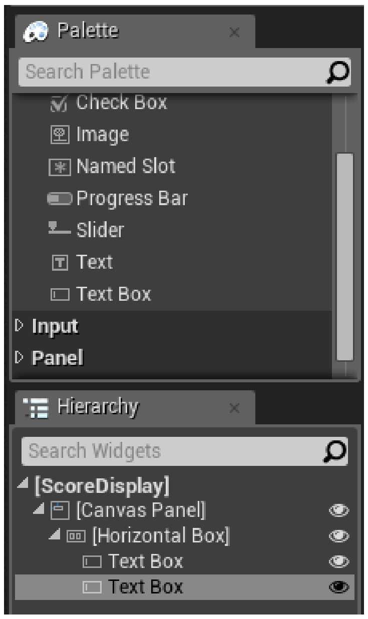
16. DISPLAY THE SCORE.

For the final section of the C++ tutorial, let’s set up a Widget to display our score.

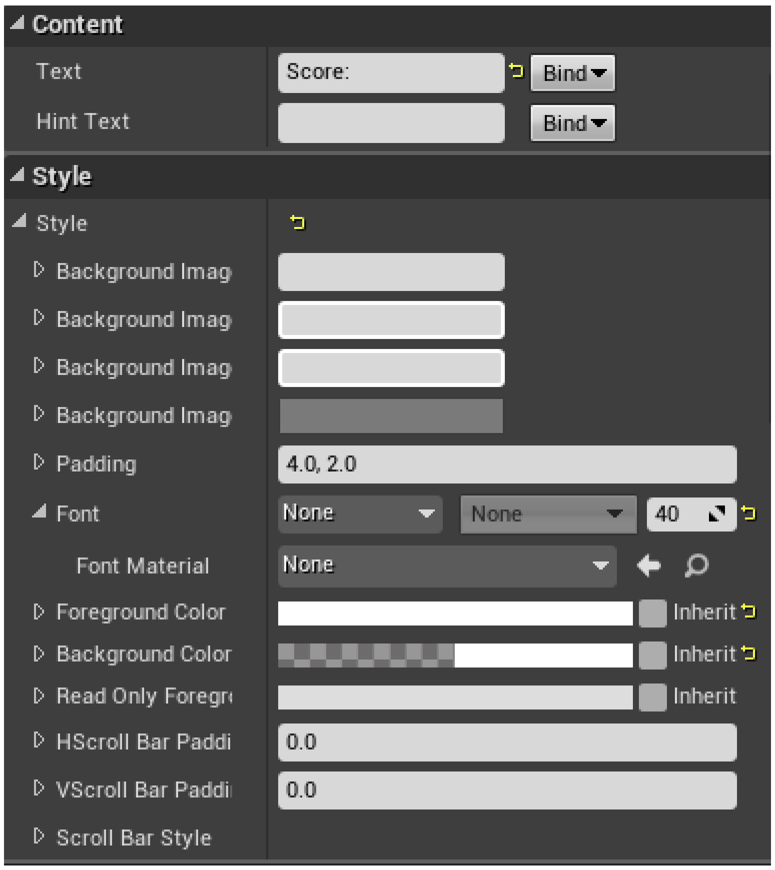
Select “Add New -> User Interface -> Widget Blueprint”. Name this widget “ScoreDisplay”. Widgets blueprints in Unreal are used to create and lay out UI elements in your game, also known as widgets. Double click on this file to open it in the Widget Blueprint Editor. The layout of the display can be seen in the Hierarchy panel. The widget blueprint starts off with a Canvas Panel widget.

In the Palette, search for the “Horizontal Box” and drag it into the top left of your Canvas Panel. Next, drag in a “Text Box” widget into this box. It should be a child of your Horizontal Box widget.

In the Text Box, type “Score: “ into the Text field in the Content panel. In the Font tab under the Style panel, change the font size to 40, change the foreground color to white, and make the background color transparent (Set the Alpha channel to 0). Select Compile in the Menu Bar, and you should see your updated text.



[ScoreDisplay]
- [Canvas Panel]
  - [Horizontal Box]
    - Text Box
    - Text Box

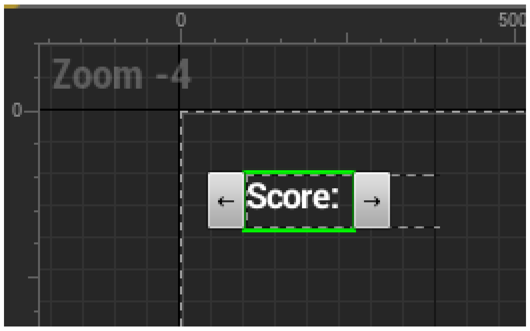


Content

Text	Score:
Hint Text	

Style

Style	
Background Imag	
Background Imag	
Background Imag	
Background Imag	
Padding	4.0, 2.0
Font	None, None, 40
Font Material	None
Foreground Color	Inherit
Background Color	Inherit
Read Only Foregr	Inherit
HScroll Bar Paddi	0.0
VScroll Bar Paddi	0.0
Scroll Bar Style	

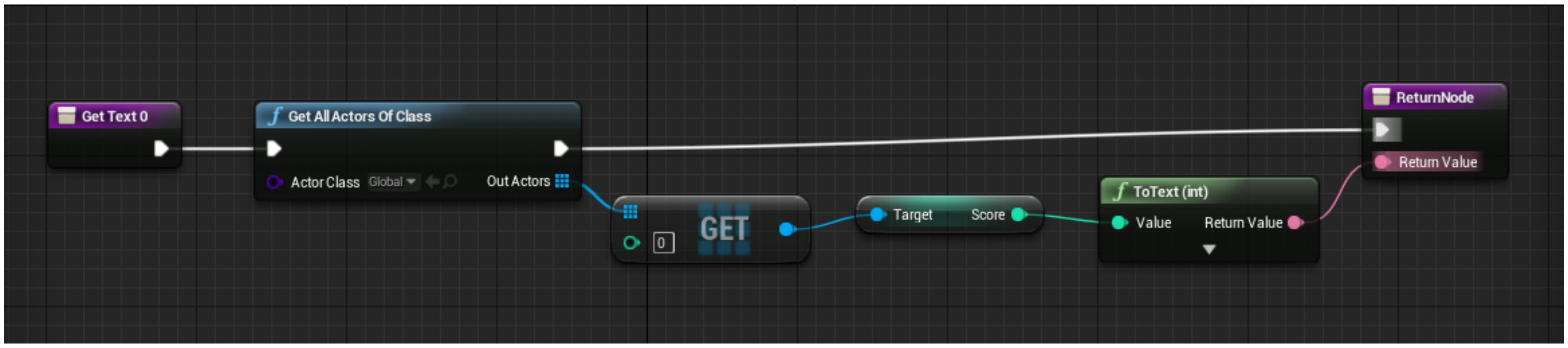


Copy and paste this Text Box into the same level of the hierarchy as the first Text Box, in order to keep our Appearance settings. Erase the text field in the Content tab. We will now plug our Score variable into this Text Box.

Select the drop down to the right of the Text field, and select “Create Binding”. This should send you to the Graph editor. Now, we can retrieve our Score variable to use.

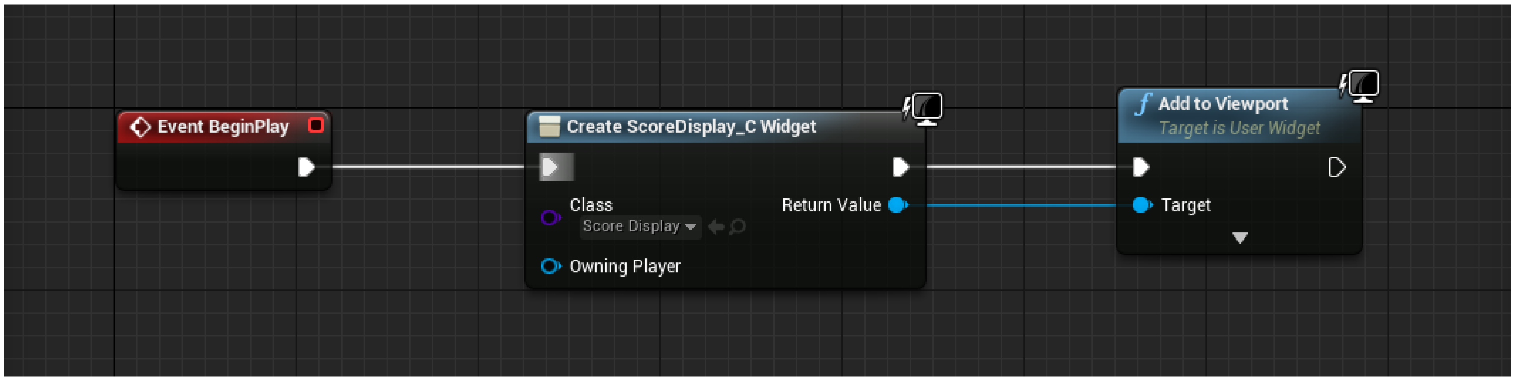
Right click in the graph and search for the “Get All Actors Of Class” node. Set the Actor Class to our Global class, and drag the “Out Actors” pin out and search for the “Get” node. Create one and leave the Index value at 0. This will search through our level for Global objects, and choose the first (and only) one. Drag out the output pin from the Get node, and search for the “Get Score” node. Drag the output from this node and place a “ToText(int)” node. Finally, connect the Return Value from this node to the ReturnNode, and connect the Exec pins in the “Get All Actors Of Class” node and the ReturnNode.

Your node graph should look something like this.

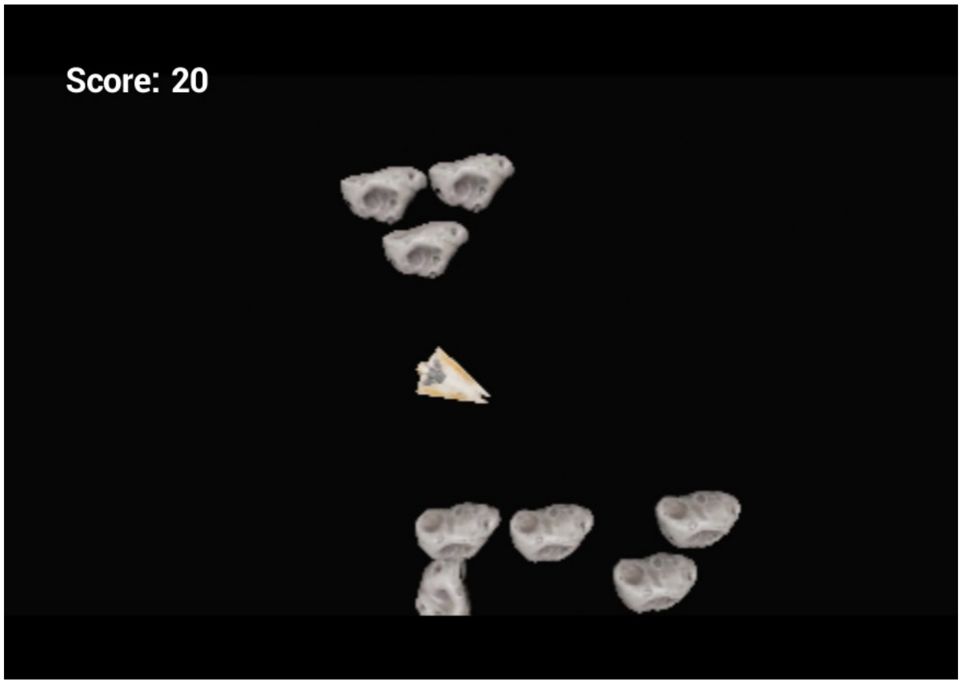


Finally, to add this Widget Blueprint to our level, we have to open the Level Blueprint. Each level has a Level Blueprint, which has its own node graph which gets executed during gameplay. This blueprint can be accessed in the main Menu Bar, in “Blueprints -> Open Level Blueprint”.

You should see an “Event BeginPlay” node and an “Event Tick” node. We want to connect our new nodes to the BeginPlay node, because we only want to load our display at the very beginning. Right click on the graph and search for “Create Widget”. Create and connect this node, and select the “ScoreDisplay” class in the Class field. Drag the Return Value pin out from this node and create an “Add to Viewport” node. The pins should automatically be connected.



When you play the game now, you should see your counter on the top left of the screen, and it should increment by 10 every time you destroy an asteroid.

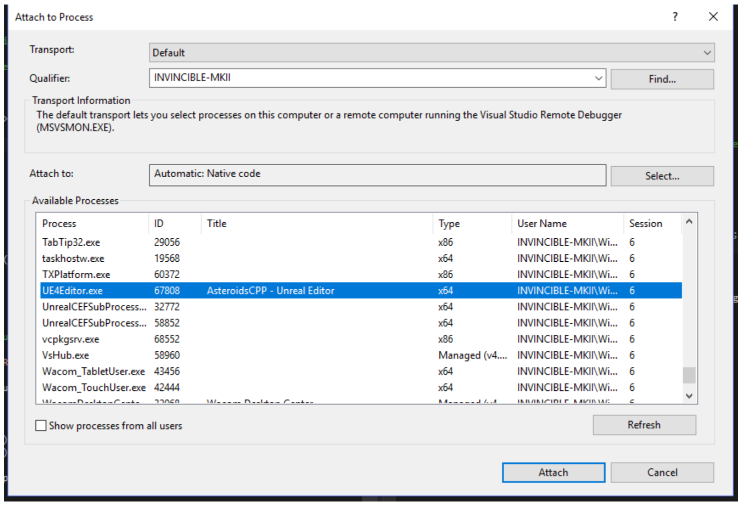


Congratulations! You should now have a port of the Asteroids game in Unreal Engine. Feel free to extend or add features to this demo to learn more about Unreal, or move on to your own creations!

BONUS SECTION

How to debug with Visual Studio:

The C++ part of the project is built as a **.dll** file and loaded by Unreal Editor. So, in order to debug with Unreal Engine, you can attach Visual Studio to Unreal by selecting **"Debug->Attach to Process"** in Visual Studio and find "UE4Editor.exe" and click **Attach**. Then, you can use breakpoints and other debugging techniques as you normally would with Visual Studio.



Attach to Process

Transport: Default

Qualifier: INVINCIBLE-MKII

Transport Information
The default transport lets you select processes on this computer or a remote computer running the Visual Studio Remote Debugger (MSVSMON.EXE).

Attach to: Automatic: Native code

Available Processes

Process	ID	Title	Type	User Name	Session
TabTip32.exe	29056		x86	INVINCIBLE-MKII\Wi...	6
taskhostw.exe	19568		x64	INVINCIBLE-MKII\Wi...	6
TXPlatform.exe	60372		x86	INVINCIBLE-MKII\Wi...	6
UE4Editor.exe	67808	AsteroidsCPP - Unreal Editor	x64	INVINCIBLE-MKII\Wi...	6
UnrealCEFSubProcess...	32772		x64	INVINCIBLE-MKII\Wi...	6
UnrealCEFSubProcess...	58852		x64	INVINCIBLE-MKII\Wi...	6
vcpkgsrv.exe	68552		x86	INVINCIBLE-MKII\Wi...	6
VsHub.exe	58960		Managed (v4....	INVINCIBLE-MKII\Wi...	6
Wacom_TabletUser.exe	43456		x64	INVINCIBLE-MKII\Wi...	6
Wacom_TouchUser.exe	42444		x64	INVINCIBLE-MKII\Wi...	6

☐ Show processes from all users

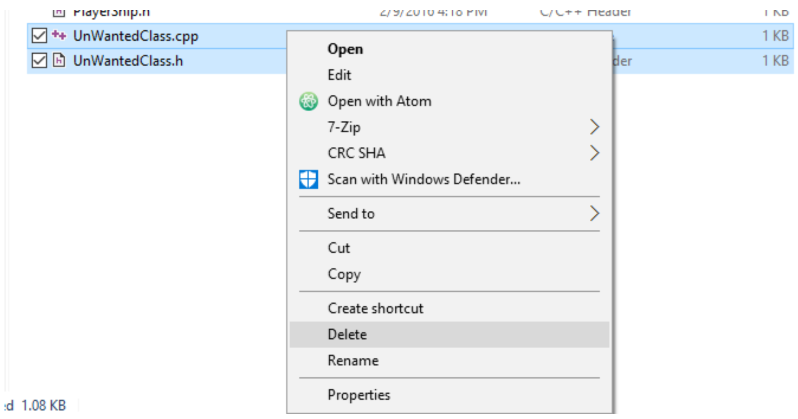
How to delete a class:

Sometimes, you'll find that you no longer need a C++ class. But if you delete the files directly in Visual Studio, the project will refuse to build and popup error message of missing files.

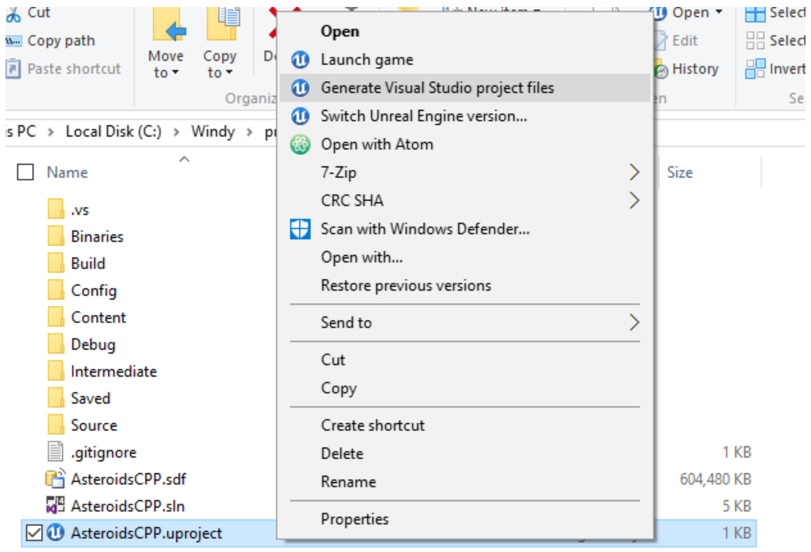
The correct approach to delete a C++ class (using Windows) is as below:

1. Close Unreal and Visual Studio.

2. Locate project folder, find and delete the **.cpp** and **.h** files of the unwanted class in **Source** folder.



3. Go back to root folder of the project, right click the **.uproject** file, and select **"Generate Visual Studio Project Files"**



3. Open the regenerated **.sln** file in Visual Studio and select **"Build->Build Solution".**

4. Open the project again, and the unwanted class should be gone.

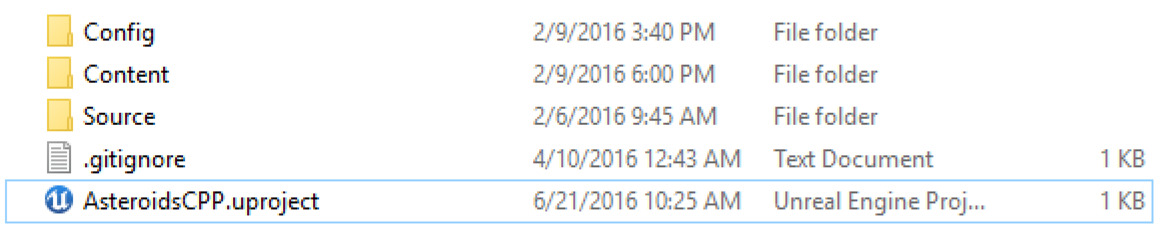
How to clean up the project folder before uploading:

It is helpful that you clean up the project folder before uploading your project to Canvas or sharing your future projects with others, so that you won't be sending an asteroid project in gigabytes.

Typically, you only need following folders and files for your project to be able to build:

Config\
Content\
Source\
YourProject.uproject

You don't need the .sln file for Visual Studio because you can right click the .uproject file and "Generate Visual Studio Project Files". A clean project folder will be like this:



Config	2/9/2016 3:40 PM	File folder	
Content	2/9/2016 6:00 PM	File folder	
Source	2/6/2016 9:45 AM	File folder	
.gitignore	4/10/2016 12:43 AM	Text Document	1 KB
AsteroidsCPP.uproject	6/21/2016 10:25 AM	Unreal Engine Proj...	1 KB

If you are using Git, you can use this **.gitignore** file:
https://github.com/github/gitignore/blob/master/UnrealEngine.gitignore

Additionally, if you are working on a group project you can add **Content/StarterContent/*** to .gitignore if you didn't change assets in it, and the person you are collaborating with can simply copy the starter content from his/her Unreal Engine folder to the cloned project.

More tutorials:

- Blueprints:
  - Official video tutorials for Blueprint: http://docs.unrealengine.com/latest/INT/Videos/PLZlv_N0_O1gYeJX3xX44yzOb7_kTS7FPM/
  - Basic flight game tutorial from Pluralsight: https://app.pluralsight.com/library/courses/creating-flight-simulator-unreal-engine-2117/table-of-contents
  - "CIS568 Unreal tutorial – Asteroids 1.0" from canvas – Contains one section on pure blueprint implementation of Asteroids.

- Networking:
  - Official video tutorials for blueprint networking: http://docs.unrealengine.com/latest/INT/Videos/PLZlv_N0_O1gYwhBTjNLSFPRiBpwe5sTwc/TbaOyvWfJE0/index.html
  - Multiplayer Shootout: A working multiplayer project that supports Steam API. Download it from the "Learn" tab of Epic Games Launcher.
  - Official docs for networking: https://docs.unrealengine.com/latest/INT/Gameplay/Networking/
  - A Video tutorial for C++ networking: https://www.youtube.com/watch?v=EGnMgeeECwo

- Virtual Reality:
  - Virtual Reality best practices (dos and don'ts): https://docs.unrealengine.com/latest/INT/Platforms/VR/ContentSetup/index.html
  - Example project with Leap Motion Plugin: https://developer.leapmotion.com/gallery/ue4-community-plugin-example-project