Paper2D Tutorial in UE4

Benny Peake

Version 1.1, Tested with UE4 v4.9

March 2016

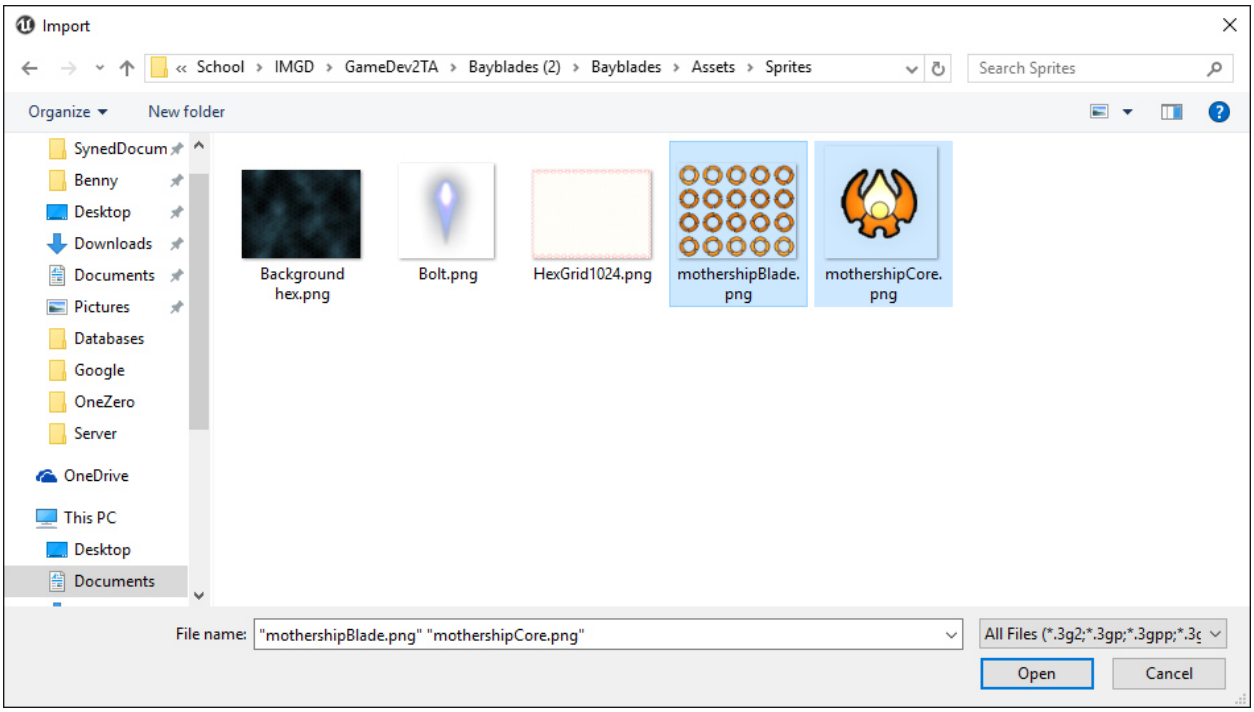
Overview:

This tutorial is designed to give you a quick intro to using Paper2D in Unreal 4. It's recommended that you already have some background knowledge of how to use Unreal 4, although a lot of the basics will be explained here as well.

For this tutorial we will be making a very basic game with a space ship that can fly around. We won't be adding gameplay to it as this tutorial focuses more on how you can use Unreal for your 2D game.

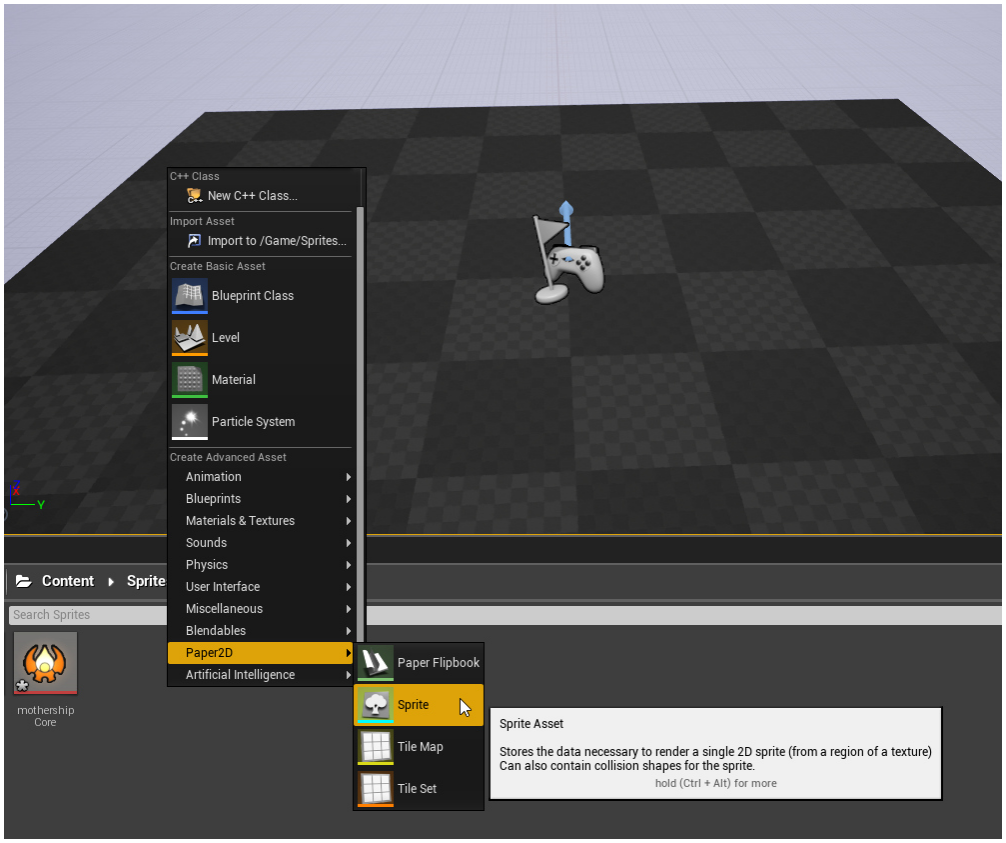
Setup:

Create a new project, "C++, Basic Code" with no starter content. In order to begin making our game we need to have some assets for our ship. From the Content Browser, create a new folder called "sprites" (you can do this by right clicking and pressing "new folder"). In the Content Browser, press import and select the mothershipBlade and mothershipCoree pictures (both .png).

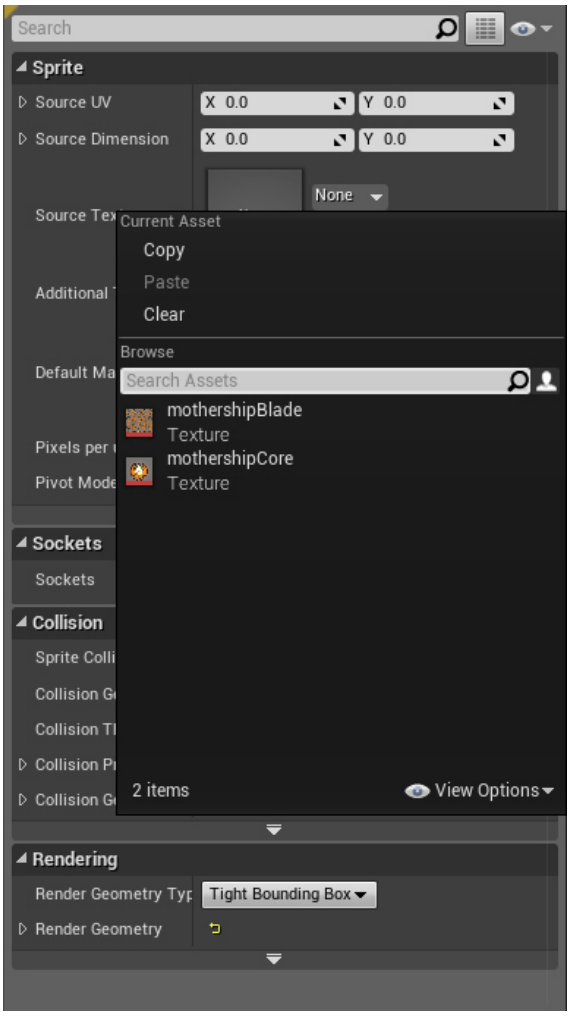


Once these textures have been imported create a Ship folder within sprites, and then underneath this create a Core and Blade folder. Place the sprites into their respective folders. This is mainly for organization.

The content is currently imported as textures, however in order to use it in our game we need it to be a sprite asset. Luckily this is pretty easy to do. Let's start with the core sprite. Right click next to the asset and go down to Paper2D/Sprite (note! If you do not see "Paper2D" as an option as in the picture below, you may need to scroll the popup window).

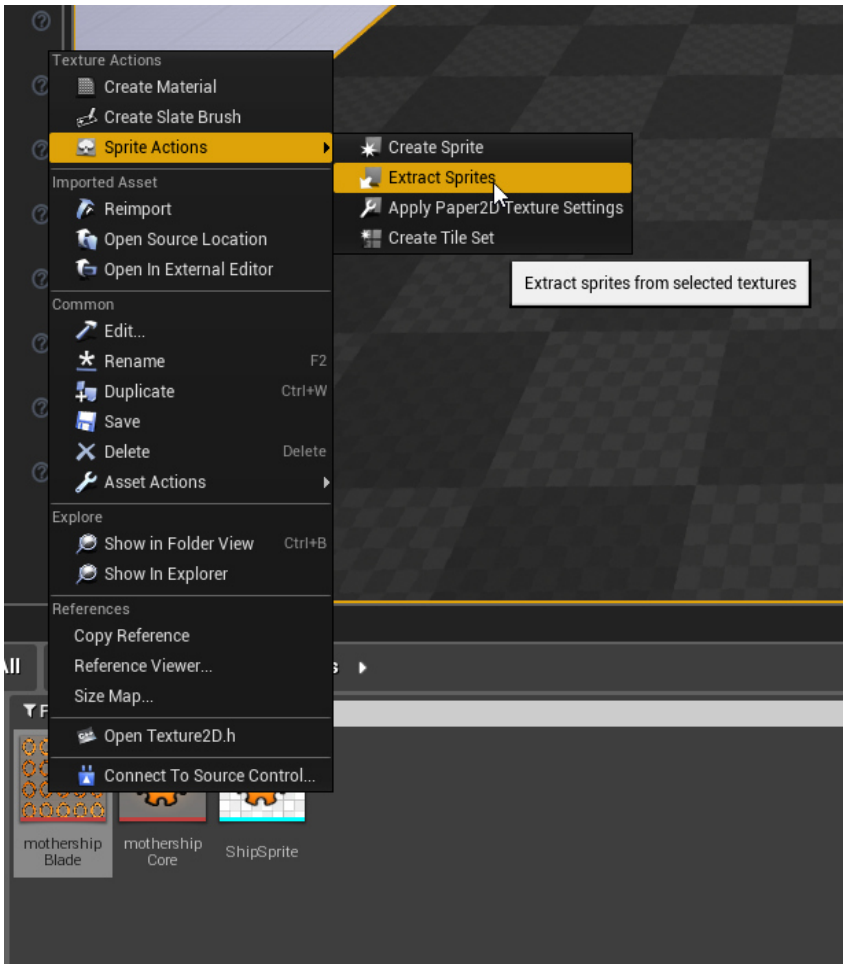


This will create a new sprite asset. Name it *ShipSprite* and double click it. This will open up the sprite editor. For this sprite we have relatively little to do as far as editing. On the right hand side under *source texture* click the drop down and select *mothershipCore*.

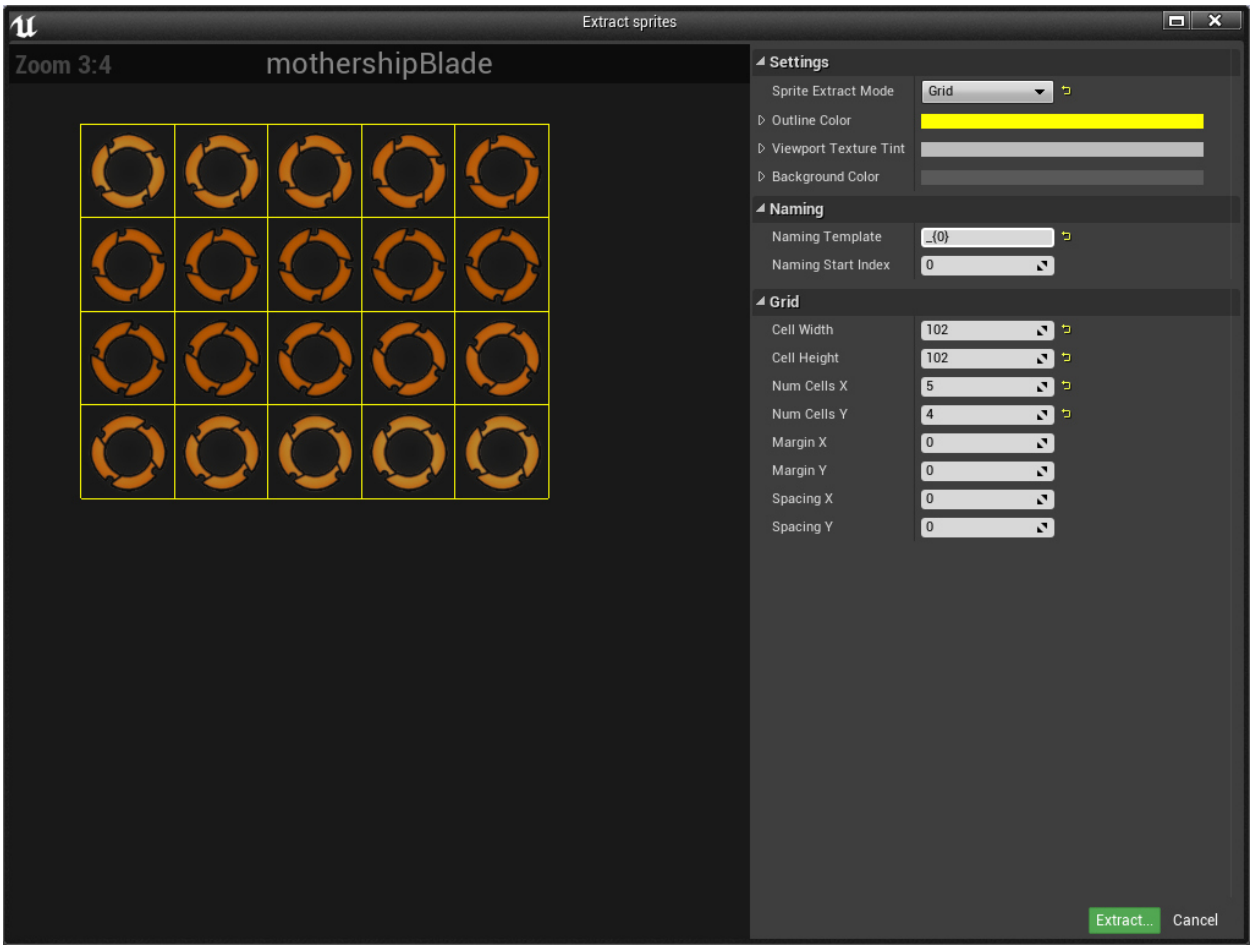


Then under *Collision* set *Sprite Collision* to none, we will be using a separate collider for our collision. When done save and close the spite editor.

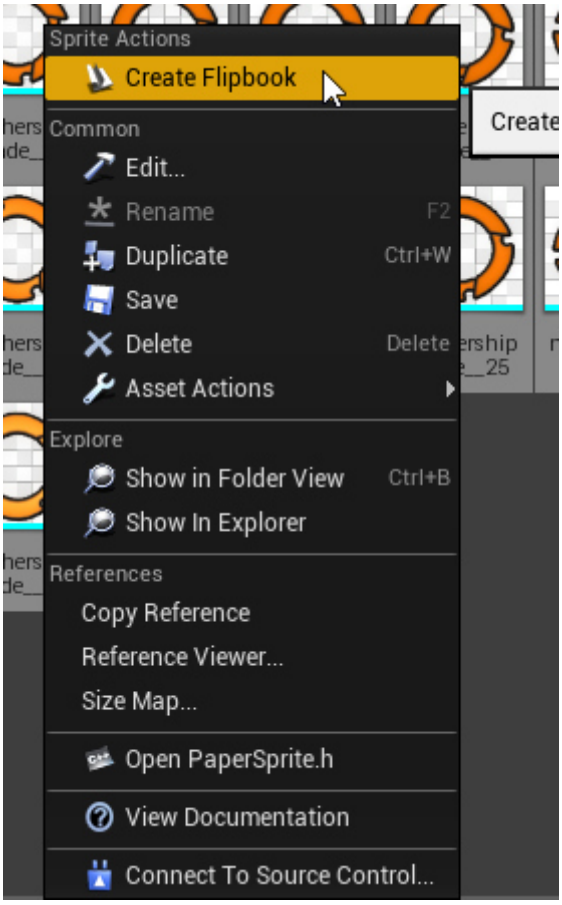
Now we have to edit the *mothershipBlade* into sprite format. This sprite is an animation where all the frames are stored on a single sprite sheet. In order to use it we have to create a sprite for each frame. Luckily Unreal has some quick tools for dealing with this. Right click on *mothershipBlade* and select *Sprite Actions/Extract Sprite*.



This will bring up a tool for extracting the sprites from a single sprite sheet. By default the editor will try to auto detect sprites, however with an animated sprite this can cause the sprite to look like its shifting if the regions it creates are different sizes. Instead we will tell unreal to extract using a grid. On the right hand side set *Sprite Extract Mode* to grid. Change the *Cell Width* and *Cell Height* to 102. Set *Num Cells X* to 5 and *Num Cells Y* to 4. You should also change the *Naming* Template to "_{0}". This will cause the new sprites to be name *mothershipBlade_{Cell Number}*. Your editor should look like this.



Press *Extract* and you should now see a sprite for each frame of the animation in your editor. Now we need to turn these into an animation, or a *Flipbook*. To do this select the first frame, hold shift, and select the last frame. Right click on the sprites and select *Sprite Actions/Create Flipbook*.



This will create a new Flipbook asset from the selected sprites.

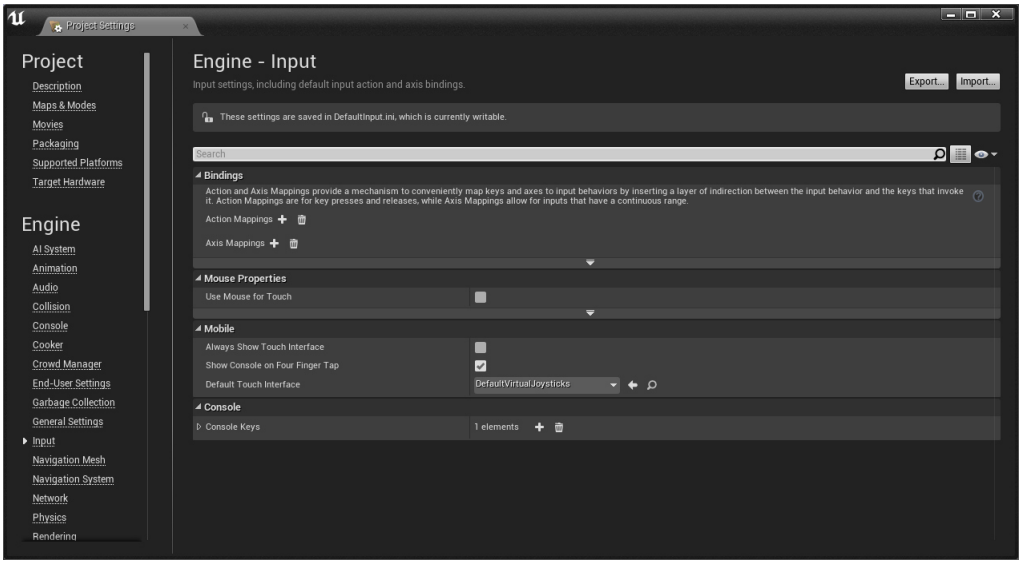
The last thing we need is a background. Under *Sprites* create a new folder *Background*. Import the hex grid texture like before and create a sprite from it. *Set Collision/Sprite Collision* to *None*.

With that all of our assets have been imported.

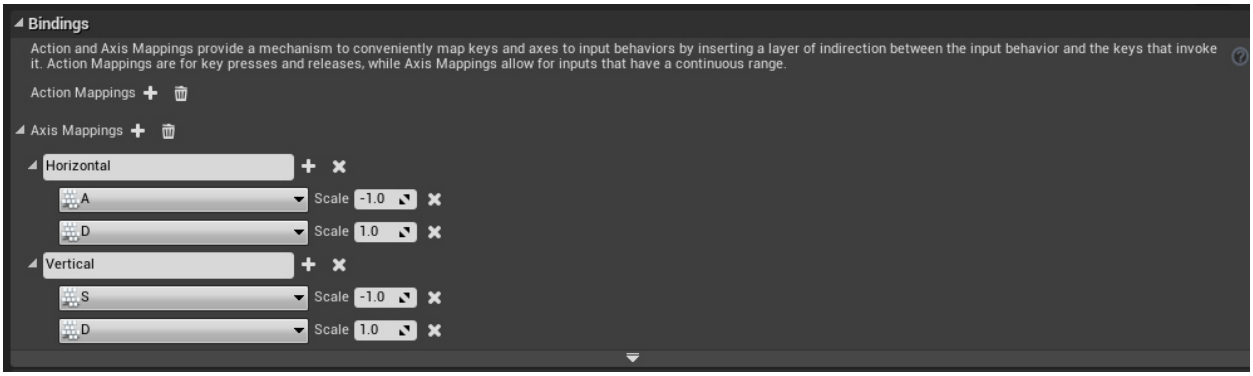
Code:

Now that we have our assets all set we can begin actually coding. We will be putting together a C++ class for our space ship. Before we can begin however there are a few things to do.

First we need to setup our input. Our input will be setup just like we would any other game. Go to *Edit/Project Settings* and find *Input* on the left hand side. You should see a menu like so:



For this demo we are going to create two *Axis Mapings*, one for horizontal input and one for vertical. Setup your binding like so:



W,A,S,D is the choice shown above, however you could also setup gamepad input or anything else that you like.

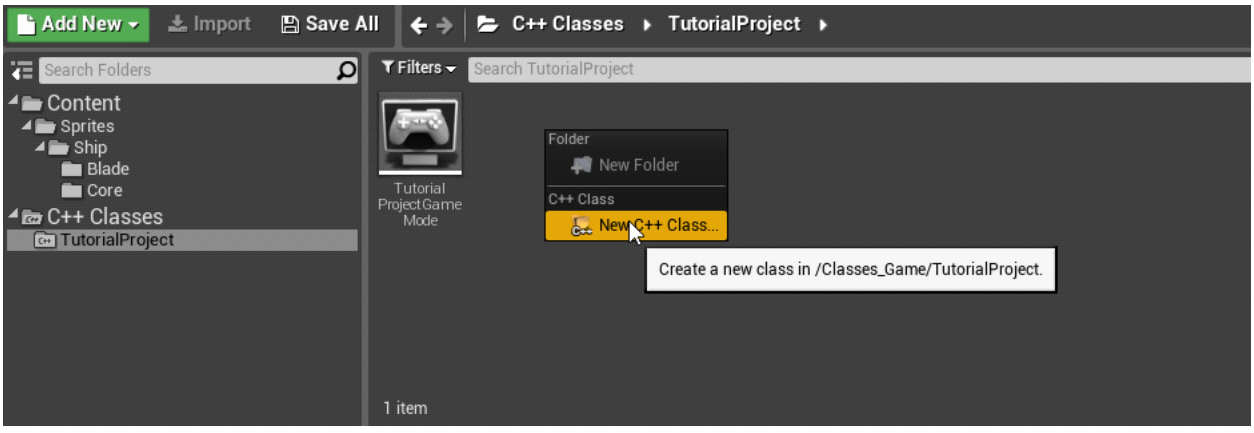
Close the Project Settings window when finished.

Now that we have our input setup we need to setup our engine for 2D. Exit out of the editor (be sure to save all your work!) and find the Unreal Projects folder for your tutorial in a Windows Explorer. There should be a Visual Studio Solution inside the same folder. If there is not then you can right click on the Unreal Engine Project File and press *Generate Visual Studio Project Files*. Open up the Visual Studio Solution and look for the .build.cs file inside your project directory (under the "Games" folder then under the "Source"). Modify the file to look like the below (note, the name "TutorialProject" will differ depending upon what you have named your saved tutorial):

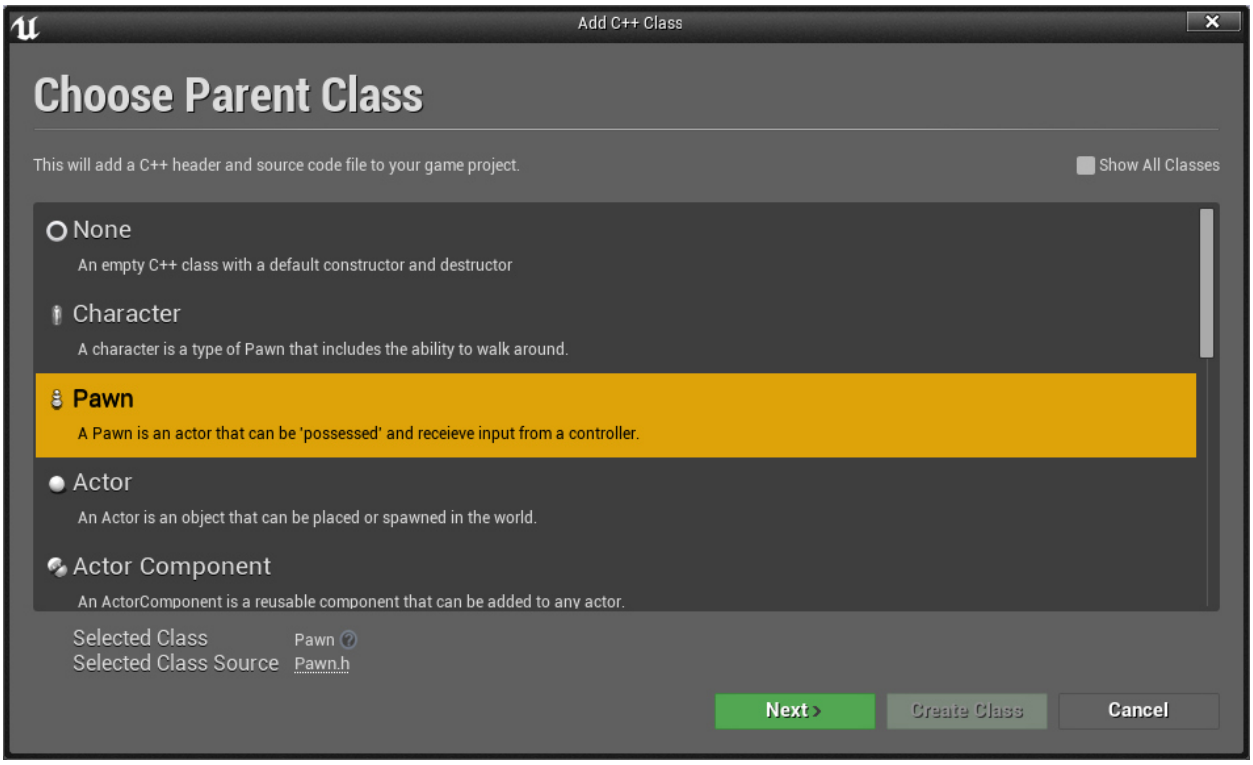
```
public class TutorialProject : ModuleRules
{
    public TutorialProject(TargetInfo Target)
    {
        PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "Paper2D" });

        PrivateDependencyModuleNames.AddRange(new string[] {  });

        // Uncomment if you are using Slate UI
        // PrivateDependencyModuleNames.AddRange(new string[] { "Slate", "SlateCore" });

        // Uncomment if you are using online features
        // PrivateDependencyModuleNames.Add("OnlineSubsystem");
        // if ((Target.Platform == UnrealTargetPlatform.Win32) || (Target.Platform == UnrealTargetPlatform.Win64))
        // {
        //      if (UEBuildConfiguration.bCompileSteamOSS == true)
        //      {
        //          DynamicallyLoadedModuleNames.Add("OnlineSubsystemSteam");
        //      }
        // }
    }
}
```

The only difference should be the addition of Paper2D in the PublicDependencyModuleNames list. This file is used to tell Unreal how to generate our Solution (.sln) file. By default the Paper2D plugin is not used, by adding it we now have access to Paper2D inside our C++ code. To apply the changes re-generate the solution file by following the same steps as listed above.

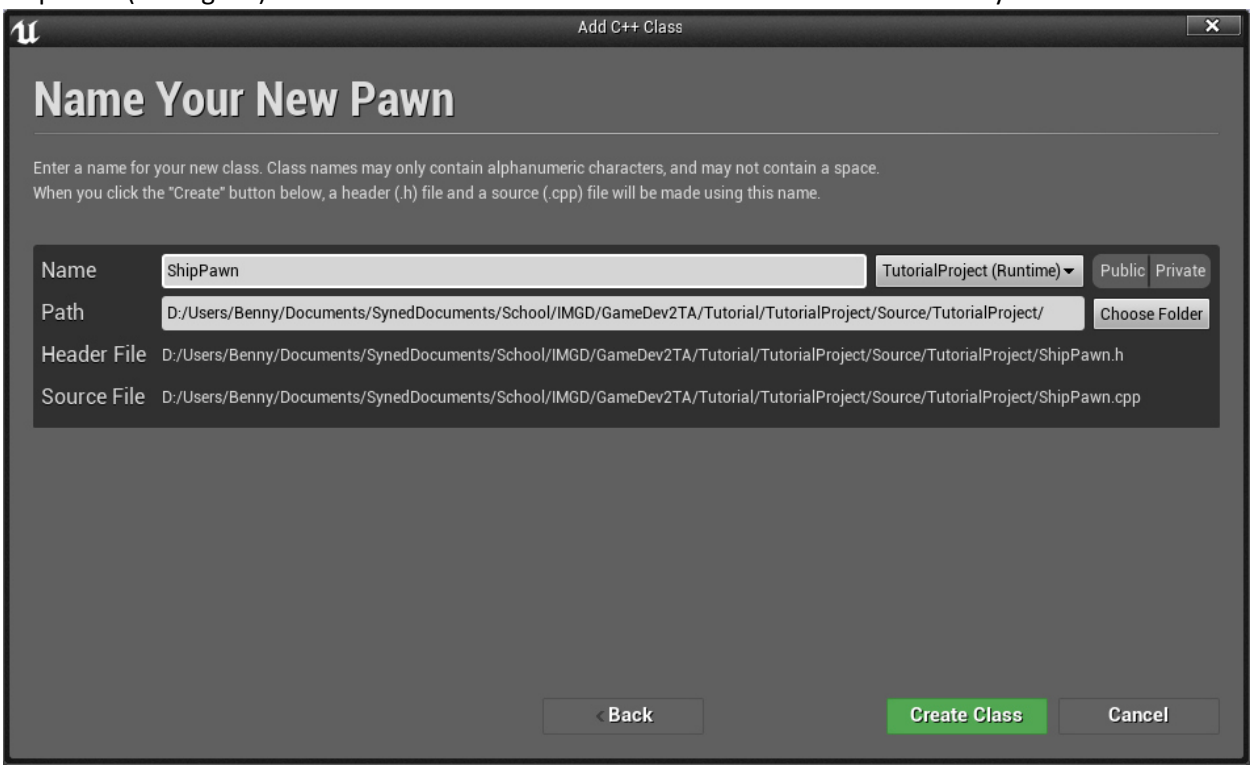
We are now ready to begin our C++ programming. Open the Unreal Editor and Visual Studio Project (note it can take a while for Visual Studio to parse all the engine files, this is normal). In Unreal, go into the C++ folder, right click, and press *New C++ Class*



A dialog will come up asking for the object type. We are going to make this a player object so select *Pawn*.



Hit next. You should now see a screen asking to name the class. We are going to name this class ShipPawn (so original). Hit Create Class and Unreal should auto create the class for you.



You should now have a ShipPawn.h and ShipPawn.cpp file inside your Visual Studio project. The first thing to do is to fill out the header file. At the top of the file put these includes:

```
#include "PaperSpriteComponent.h"
#include "PaperFlipbookComponent.h"
```

This adds in the Paper2D scripts for use. It's important to make sure that the generated header file is included last.

Now in the class add the following to the end

```
private:
	UFUNCTION()
	void OnHorizontal(float val);

	UFUNCTION()
	void OnVertical(float val);

private:
	UPROPERTY(VisibleDefaultsOnly)
	UFloatingPawnMovement *m_movementComponent;

	// The  ship sprite component.
	UPROPERTY(VisibleDefaultsOnly)
	UPaperSpriteComponent *m_shipVisual;

	// The blade animated sprite component.
	UPROPERTY(VisibleDefaultsOnly)
	UPaperFlipbookComponent *m_bladeVisual;
```

The variables we will use to store our main components of our ship and the two methods will be used to receive input. We mark the two methods as UFUNCTION() so that way Unreal will know about them.

NOTE: The modifier VisibleDefaultsOnly on a pointer will still allow members of the objects to be edited, it just means that the pointer itself cannot be changed.

Now we will edit the .cpp file. First, we start with the constructor. In Unreal the constructor is only ran once for each type (not for each object instance). This effectively creates a "template" for each type. The template can then have modifications done on it per instance allowing for variations from the template. We will use the constructor to setup our components.

```
// Sets default values
AShipPawn::AShipPawn()
{
  // Set this pawn to call Tick() every frame.
  PrimaryActorTick.bCanEverTick = true;

  // Create the root component for this object.
  USphereComponent *collision = CreateDefaultSubobject<USphereComponent>(TEXT("Collision"));
  RootComponent = collision;
  collision->SetSphereRadius(28.0f);
  collision->RelativeLocation = FVector(0.0f, 0.0f, 0.0f);
  collision->SetCollisionProfileName("Pawn");

  // Setup movement.
  m_movementComponent = CreateDefaultSubobject<UFloatingPawnMovement>(TEXT("Movement"));
  m_movementComponent->SetUpdatedComponent(RootComponent);
  m_movementComponent->MaxSpeed = 500.0f;
  m_movementComponent->Acceleration = m_movementComponent->MaxSpeed * 5;
  m_movementComponent->Deceleration = m_movementComponent->MaxSpeed * 5;

  // Setup the spring arm that the camera will attach to.
  USpringArmComponent *springArm = CreateDefaultSubobject<USpringArmComponent>(TEXT("SpringArm"));
  springArm->AttachTo(RootComponent);
  springArm->SetRelativeRotation(FVector(0.0f, -1.0f, 0.0f).Rotation());
  springArm->SetAbsolute(false, false, false);
  springArm->TargetArmLength = 500.f;
  springArm->bEnableCameraLag = true;
  springArm->CameraLagSpeed = 3.0f;

  // Setup the camera.
  UCameraComponent *camera = CreateDefaultSubobject<UCameraComponent>(TEXT("Camera"));
  camera->AttachTo(springArm, USpringArmComponent::SocketName);
  camera->SetWorldRotation(FVector(0.0f, -1.0f, 0.0f).Rotation());
  camera->ProjectionMode = ECameraProjectionMode::Orthographic;
  camera->OrthoWidth = 700.0f;

  // Create the sprite for the ship.
  m_shipVisual = CreateDefaultSubobject<UPaperSpriteComponent>(TEXT("ShipVisual"));
  m_shipVisual->AttachTo(collision);
  m_shipVisual->RelativeLocation = FVector(0.0f, 0.0f, -7.0f);
  m_shipVisual->SetSprite(ConstructorHelpers::FObjectFinder<UPaperSprite>
                (TEXT("/Game/Sprites/Ship/Core/ShipSprite")).Object);

  // Create the animated sprite component.
  m_bladeVisual = CreateDefaultSubobject<UPaperFlipbookComponent>(TEXT("BladeVisual"));
  m_bladeVisual->AttachTo(collision);
  m_bladeVisual-> SetFlipbook(ConstructorHelpers::FObjectFinder<UPaperFlipbook>
          (TEXT("/Game/Sprites/Ship/Blade/mothershipBladeFlipbook")).Object);
  m_bladeVisual->RelativeLocation = FVector(-2.0, 0.0f, 0.0f);
  m_bladeVisual->RelativeScale3D = FVector(1.5f);
}
```

NOTE: make sure to replace the names of the assets (e.g., the sprite) with the names used in your project.

For a brief explanation of the constructor code, first we setup the collision for our object. We give it a radius and collision profile. We also set it to the root component so all things will be children of it.

Next we create our movement component. This defines how our object should move. Because we are doing a space ship we just use a floating component. We tell the floating component to use the collider as the updated component.

Next we create a spring arm. We will use this component for our camera. The spring arm is like an extended arm with a point to mount things at the end. We can have the spring arm have a dampening effect, however, which will allow our camera to follow slightly behind. We also setup the spring arm to not track the rotation of our pawn.

Next we create our camera and attach it to the end of our spring arm. We also set it up to be an orthographic camera. This causes objects to not get smaller as they get farther away from the camera. We then set the width of the camera to be a little wider than default to give us more of a view.

Finally, we add a sprite and a flipbook component for the visuals. One will represent the ship core and the other will represent the animated rotating blade.

Next we have to setup the input in the same .cpp file.

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

  InputComponent->BindAxis("Horizontal", this, &AShipPawn::OnHorizontal);
  InputComponent->BindAxis("Vertical", this, &AShipPawn::OnVertical);
}

void AShipPawn::OnHorizontal(float val)
{
  FVector input = FVector(val, 0.0f, 0.0f);
  AddMovementInput(input);
}

void AShipPawn::OnVertical(float val)
{
  FVector input = FVector(0.0f, 0.0f, val);
  AddMovementInput(input);
}
```

First, in SetupPlayerInputComponent we bind the vertical and horizontal axis to the corresponding method calls. We then have a method to handle applying the horizontal input to the horizontal axis (x) and vertical to the vertical axis (z). We then add this to the AddMovementInput which will tell the movement component to move.

In Unreal, we now can create a Blueprint using this pawn. In the Content Browser, create a folder named "Blueprints" and inside right click and press *Create Basic Asset / Blueprint Class*. Under the *All Classes/Actor/Pawn* select *ShipPawn* as the base class and name the Blueprint *ShipPawnBlueprint*.

Finally, we need to setup the game to actually use our pawn. Inside the pre-generated game mode .cpp

```
ATutorialProjectGameMode::ATutorialProjectGameMode()
{
  DefaultPawnClass = ConstructorHelpers::FClassFinder<AShipPawn>(TEXT("/Game/Blueprints/ShipPawnBlueprint")).Class;
}
```

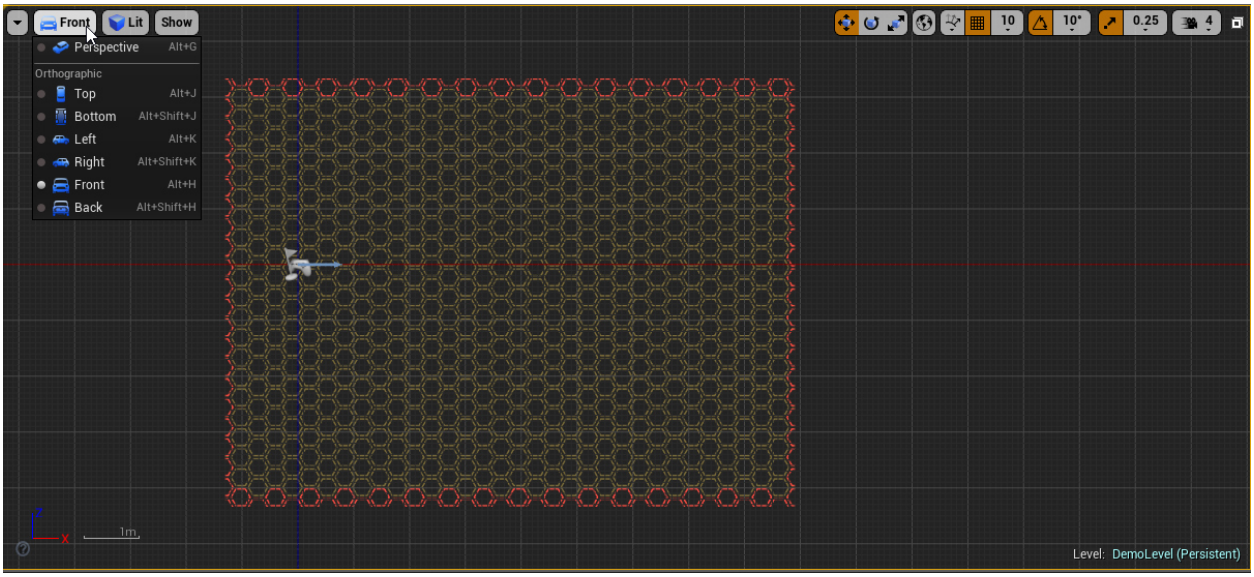
file, make the following constructor (adjusting the name to be appropriate for your tutorial name).

```
public:
    ATutorialProjectGameMode();
```

This tells the game mode to find the Blueprint we created and use it as the pawn. Add an entry for the constructor in the .h file, too.

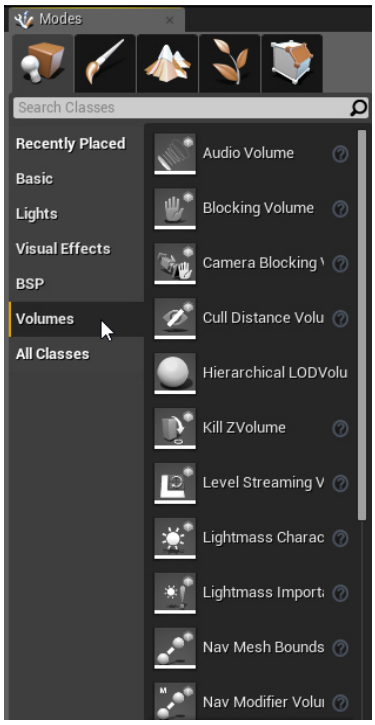
Level Setup:

Before editing it is recommended that you put your camera into orthographic mode. You can do this by pressing the button as shown and selecting “Front” and “Lit”.

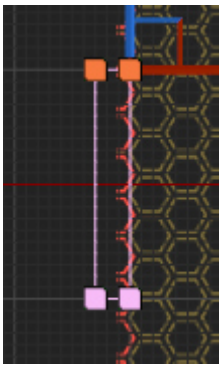


There are a few things to do in our level setup. First, we need to create a new blank scene. With the new blank scene, create a player start point and place it at [0, 0, 0]. Next, drag in the background sprite that we imported at the start and place it at [0, 500,0]. This put the background on a farther back plane so it will appear behind the player.

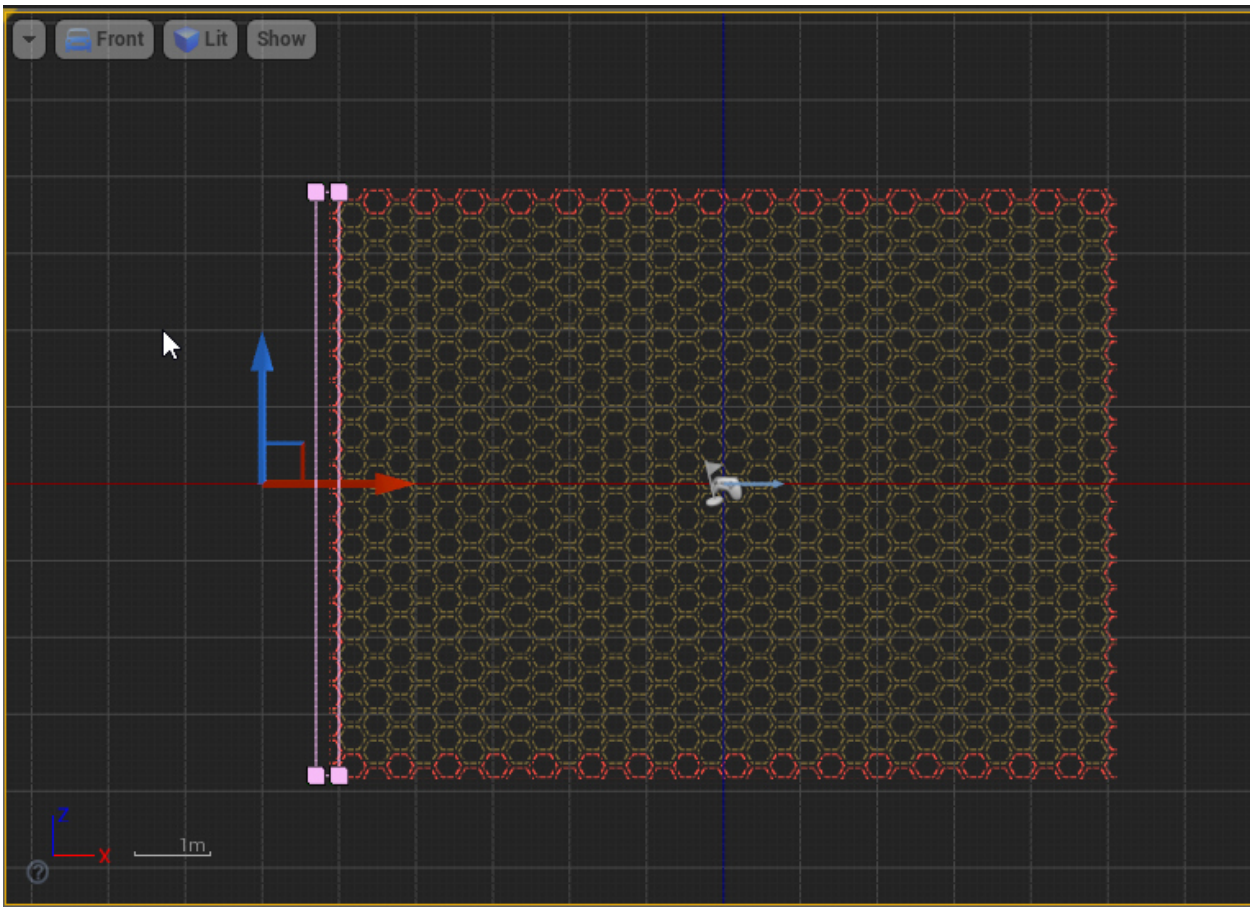
We still want to setup our player to stay in bounds, however. From the *Modes* panel, go to *Volumes* and find *Blocking Volume*.



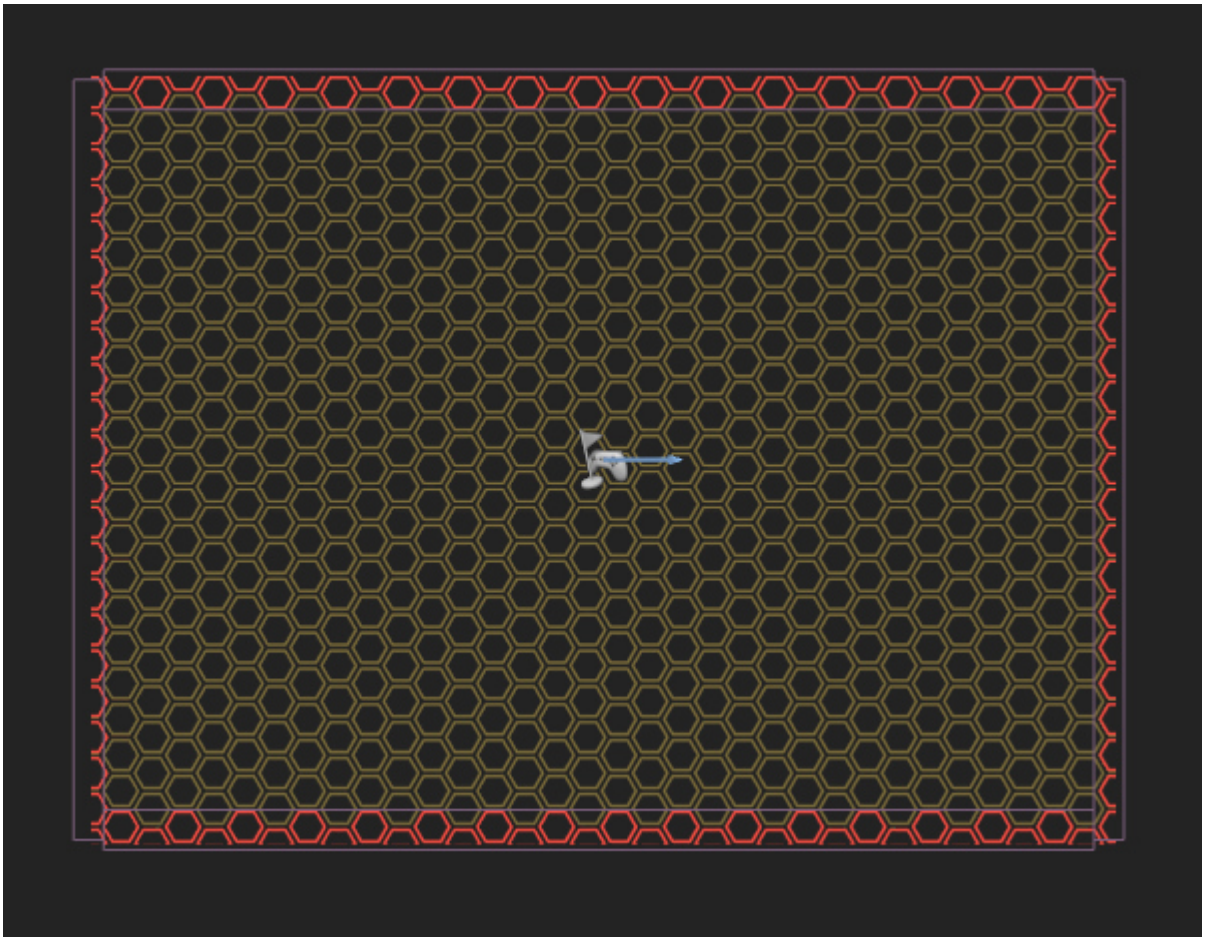
Drag the volume into the scene. Then go into volume editing mode by clicking the last tab on the *Modes* bar on the right hand side. You can now select and move individual vertices. Use this to create a wall like shown.



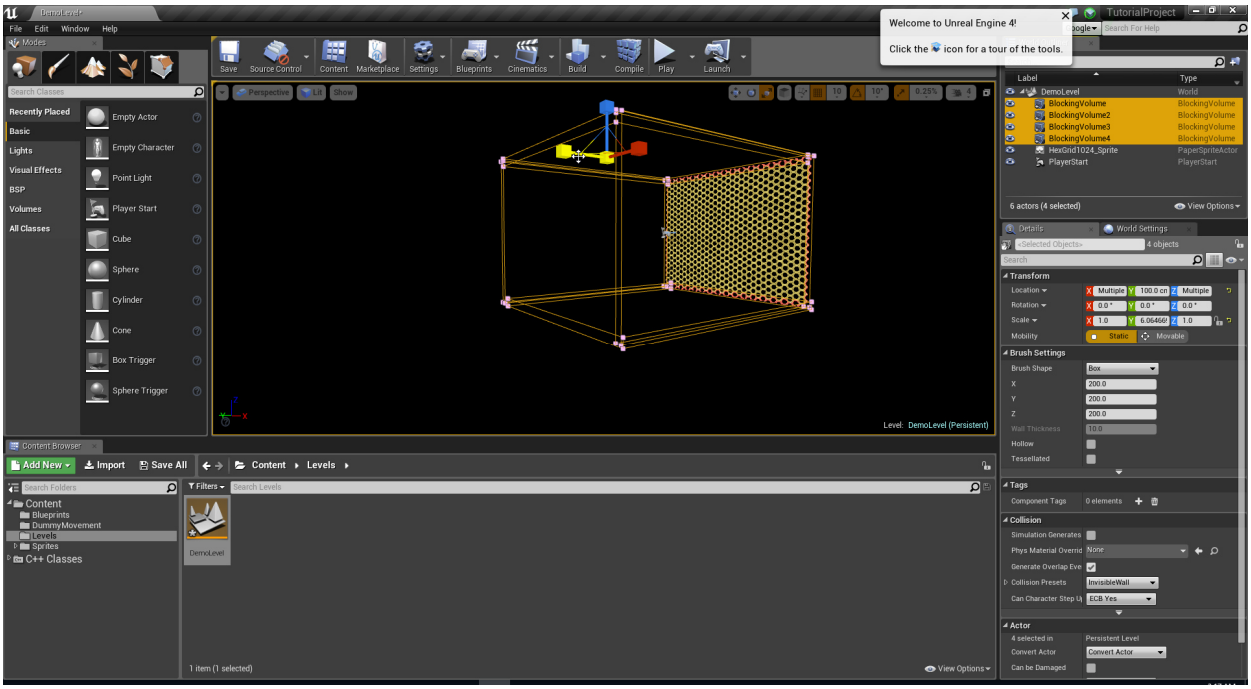
You can then make this wall the size of the edge of the background to create a blocking wall for the arena.



Do this four times to create walls all around the arena. You should end up with a level that looks like so.



Make sure the walls are all aligned along the z-axis and then extend them back. This ensures that the player can hit them.



Set the game mode in *Project Settings/Game/Maps & Modes/Default Modes* to be your tutorial game mode.

With that you should be able to run the game and move the ship around. Select *Play* or *Launch* and try it out!