Hello, this is Jesco, you’re watching game dev made easy and in today’s video, we are going to continue on with the Migration from Unity to CryEngine series. In the last episode, we showcased basic movement and skipped out on jumping. I figured I should rectify that little situation as none of the current templates actually showcase a jumping mechanic in CryEngine and it seems to be a fairly popular question even with Unity.

(Intro)

Now, before we jump into the code, I think it is only fair to go in and discuss what jumping actually is as well as discuss how jumping works in a game. So, what is jumping in a 3D game? Well, jumping is taking the vertical position and applying a vertical impulse and velocity to the model for it to rise from the floor. Gravity will be affecting the model for it to drop back down at the same speeds after it reaches a certain height ratio to land back down on the floor.

Some games give the player the ability to jump once or to have a double jump. Both of these follow the same principles to complete a jump. You will notice that games don’t really allow for continuous jumping as then the model would go well away from the limits of where the player should be. Even games where you can have jumping combos have a limit to how high and how often you can jump. Now, how can we initiate, control and specify all the parameters for allowing a player to jump?

The basic jump loop is a basic check to see if the player is on the ground, if they aren’t, then the player can jump. When they jump, add the vertical impulse and velocity to the model, the ability to jump is locked because the check for if the player on the ground is changed, once it reaches the high point, gravity will take effect and pull the model back to the ground. When the model lands on the ground, the check for if the model is on the ground is reset to allow for jumping again.

That wasn’t so difficult to understand in plain English was it? Nah, absolutely not. So let’s go ahead and move into building the ability to jump in Unity.

(Transition)

With Unity, things are pretty quick and easy. After all, Unity was designed to allow for rapid prototyping. The first thing to do is to add 2 3D components. I will go with a Cylinder for the player model and a plane for the ground.

The only things we need to do is position the cylinder above the plane, change the tag from untagged to player and add a rigidbody component to the cylinder model. Press play button to make sure you can see the cylinder and plane properly.

Create a folder called Code. Now create a C# script and call it player jump. To get it out of the way and attach it to the cylinder, for obvious reasons.

With all of the setup out of the way, we can jump into the fun part. Writing the actual code. Which isn’t anywhere near as scary as you might think.

So open it up in Visual Studio by double clicking on the player jump script. The very first thing I’m going to do is delete everything from within the class and do a sanity check for my curly brace conventions.

Alright, so the first thing I want to do is create a private float jumpforce with the value of 10 float. This will be the y velocity of our cylinder’s ability to jump.

Next, I will create a private Boolean called is grounded with the value being set to false. This is going to be used to say whether or not the cylinder can or cannot jump.

The last globally scoped variable will be a private Rigidbody called player. We will be utilizing the built in physics functions for our movement.

And before you start typing in the comments, Can you show an example without using Rigidbody? The answer is no. Shame on you for even asking. FOR SHAME!!!!

(looking side to side clip with voice going “ahem”)

The first method we need is the start method. So type void Start. Inside of this method we will type one thing and that one thing is player is set to get component with Rigidbody as the type specified. This is so that we don’t have to do anything extra in the editor as it will find the rigidbody component that is already attached to it.

Let’s take care of the jump method next. Write void Jump and inside we will create an if statement.  
The if statement will be if input dot get key down the key code will be the space key and is grounded is equal to true. Inside of the if statement we will write, player dot velocity is set to be a new vector 3 value of 0 comma jump force comma 0.   
This should be rather self-explanatory but we are checking if the jump button which is the space bar is down and the Boolean flag for is grounded is set to true, then apply this vertical velocity to the cylinder.

Now set the Boolean value for is grounded to be false. We don’t want to allow for jumping after it has been called while the player is in the air.

The next method to create is our update function. Write void Update. We want to update the script’s loop.

Now write our Jump method inside of the update function. This is to make sure that we actually call this method and update the position of the cylinder.

At this point, you are probably thinking that this will not work and that I have no idea what I’m talking about as the Boolean value for is grounded will always be set to false. And if I were to go back to the Unity editor and test, that would be true. However, we aren’t done with the code just yet so hold your horses!

The last method we will create is an On Collison Enter method. Write void OnCollisionEnter with the parameter of Collision. This will check to see if any collisions have occurred.

Write an if statement. If collision dot game object dot name is not equal to player, with a capital P. We want to check if the player has collided with anything other than itself.

Write is grounded is set to true. This will make sure that even if the cylinder is not on the ground on load that the player cannot jump.

Go back to the Unity Editor and press the play button, works like a charm.

As we can see, creating the ability to jump in Unity is rather easy and does not have too much work is involved, even if we wanted to make sure that jumping cannot occur unless the player is on the ground. Now, let’s see how we can create this exact same mechanic in CryEngine.

(Transition)

I’m going to immediately start off and say right off the bat that implementing this in CryEngine is a bit more involved and a few ideas from Unity do not carry over at all. So I will always recommend that you have a firm grasp of the C# programming before even attempting to use CryEngine. This isn’t to scare you away from CryEngine at all, it is just the honest truth. Much of the documentation does not have code examples available and the tutorials on the CryEngine website are not always the most up to date, but they are working hard at doing those updates.

Open up the CryEngine launcher and create a blank C# project with the name of Jump Mechanic. Now launch the project.

We need to create 3 folders right off the bat. Code, Objects and Levels.

Inside of the Objects folder, create a folder called Primitives. This will serve as where our primitive model will be.

In Blender, Maya, 3DS Max or any other 3D modeling software you have, create a cylinder and export it as fbx. Alternatively, download an fbx cylinder from Turbosquid or any other website. This is important, CryEngine does not support the OBJ file format, so make sure you choose fbx only.

Next, right click inside of the Primitives folder and click on import. Navigate to the folder where your model is and click on open. Select import all and the model with the associated mtl file will appear in the folder.

Click on File and New. Click on level and in the Level Name portion, put a forward slash and write Level01. Click save. This will create a new level and load it up for us.

Navigate over to the create objects menu on the top left hand side of the view screen. Click on it and you will see a dropdown list where Objects will appear, click on objects which will open primitive shapes, click on primitive shapes and long click on the Cylinder to drag and drop it into the Level.

Click on your code folder and right click inside to open a new popup menu, highlight new and select C# script. Name the C# Script to be PlayerJump.

Click on the Cylinder in the level, this will open the properties menu. Click on Add Component and select the Player Jump Script. This has pretty much set us up for being able to just write the code and test afterwards.

Double click on the Player Jump script to open it up in Visual Studio. I would highly suggest looking at the video in the eye card if this is the first video of mine you are watching as there are some steps for setting up CE to work with Visual Studio environment.

And now ladies and Gentlemen, the moment you’ve all been waiting for! It is time to write the code we need to get things working.

The very first thing we want to do is delete the On Game play start and the on update methods. We want to start completely from scratch.

The first method we will create is the Prepare Rigidbody method. This method will set our Entity, which you can read as gameobject to be a physical entity with a rigidbody.

Write var physics entity is set to entity dot physics. This defines that our physics entity will use physics. We only want to make sure that the physics entity isn’t null. To do this, create an if statement.

If physics entity is equal to null, return. Basically, exit from this method if the physics entity is null.

Next up we will create another var called parameters and it will be set to be a new living physicalize params. The Living physicalize params is a entity type that represents player characters that can move through and interact with the physical world.

We need to create a var called player dimensions which is set to be parameters dot player dimensions. We will be using this to control some basic physics parameters.

Write parameters dot mass is set to 90 float. This set’s the entity’s mass to be whatever the value is in Kilograms.

Write player dimensions dot pivot height is set to 1 float.

This is the offset from the central ground position that is considered as the entity’s center.

(We all float down here clip)

Write entity dot physics dot physicalize with the parameter set to be parameters. This will set all of the things we did before to the Entity with the specific parameters we created.

That concludes the Prepare Rigidbody method, so now we can move on to our on gameplay start method.

Create the on gameplay start method by writing protected override void on gameplay start. You can think of this as exactly the same thing as the Start method in Unity.

Write base dot on gameplay start. What this does is call the original on gameplay start method’s base items in addition to whatever we want to add afterwards.

Last item in this method is to write Prepare rigidbody. This will actually call the prepare rigidbody method and we want it to become active when the gameplay actually starts up.

We will now create a Jump method for handling the ability to jump. So write private void Jump.

Inside, we have a few things to do. The first of which is to create a float value called jump force with the value being set to 10 float. This will be the value in which the player will move on the vertical axis.

We will create a Boolean value called grounded and it will be set to entity dot physics dot get status, with the type of status being Living status dot is flying. This is a convenience method given to us by Crytek in the Entity physics class. This will return true if the entity is not on the ground.

Now we can write our if statement.

If input dot key down key id dot space and not grounded. We want to make sure that the space key is pressed down and to check if the is flying status is not true.

Inside of the if statement write entity dot physics dot jump new vector 3 with the parameters of 0 float comma 0 float comma jump force. This is another convenience method that Crytek has graced us with. It will automatically apply an impulse for us and move the character in the direction we choose while still being affected by gravity. It is also important to reiterate the fact that the z axis in CryEngine is the vertical axis unlike with Unity where the y axis is the vertical axis.

We have finished the Jump method itself, I told you it wasn’t too bad. It isn’t anywhere near as scary to implement when you have a firm grasp of what is made available to you in CryEngine. And finding what is available to you is probably the hardest part. But I digress, We have one more method to create and that is our update method!

Write protected override void On update with the parameters set to float frame time. Obviously, this is the exact same thing as the Update method in Unity with the slight change being from delta time to frame time.

Write base dot on update with the parameter of frame time. This is to call the basic functionality of the base on update method and keep proper timing.

Write Jump.

We have completely finished our code and now it is time to jump back to the CryEngine Sandbox editor and press the play button. Press the space bar and you will see that the Cylinder will not move vertically unless it has touched the ground.

As you can see, implementing the same mechanics in Unity and CryEngine are different but similar in many respects. It is all about knowing how to approach a problem given the tools at your disposal. If you have any other mechanics that you would like for me to focus on, leave a comment below letting me know which ones you would like to see in a future Migration from Unity to CryEngine episode.

Also, the first 5 people to like and comment on this video along with being in my discord server will be entered in a randomized drawing for one person to receive a copy of The Escapist 2. Finally, I want to give a warm shout out to my patreon supporters Loki and Coop for their continued support.