Salutations, I’m Jesco and you’re watching Game Dev Made Easy. Today’s topic is implementing the State Pattern in Unity and CryEngine. The State Pattern is a design pattern that allows for an object to appear to change classes at runtime. You can think of the State Pattern as an alternative to a State Machine, the end results are very similar but the way you write them, are very different. The State Pattern is designed to be more loosely coupled from your code base so it is far easier to use it in different projects, the State Machine is much more tightly coupled from the code base and is better suited for a specific project. With that all out of the way, roll the intro!

(Intro)

The object of this video is to have as similar of an implementation with as similar a result as possible. Now, obviously things will have to be a little different as some functions don’t exist in the other engine or have different names. I am just making sure we are all on the same page here.

We will have the same classes and for the most part, the same methods for both Engines.

We will have a Character class, an ICharacter interface, a Wait State, a Wander State and a Character Behavior class.

The reason we have everything separated like this is that each dot cs file has a specific purpose. The Character class will define the context of how the State Pattern functions. The ICharacter interface will define our base methods that we want to make sure is always present in our derived classes. The wait state class is one of the states that we want the character to go into, in this case, to wait for a few moments before moving on to the next state. The wander state class is the other state we want the character to go into, in this particular instance, we want the character to move in a random direction. The character behavior class will derive from the engine specific class to allow it to attach to an object in the editor, for Unity, that would be Monobehaviour and for CryEngine, it would be EntityComponent.

Let’s do a quick transition and we will get into implementation in Unity.

(transition)

Create a new 3D project in Unity and call it Unity State Pattern. We can begin the fun journey from here.

Inside of the Assets folder, create a folder called Code. Now create another folder and call it State Pattern. Inside of the State Pattern folder, create a new C Sharp file and call it ICharacter. Create a second C Sharp file and call it Character. The next C Sharp file to create is the Wander State. The second to last file is called the Wait State. Go up by one in the directory tree to create our final C Sharp file, which we will call Character Behavior.

Open up the project in Visual Studio by double clicking on the Character Behavior script. Locate the ICharacter code file and it is time to make the changes we need to set this one up.

First things first, delete everything from within the code file. Yes, everything! Your editor should be as blank as mine right now.

We do need some using statements, but only one necessary for this one. Using Unity Engine.

Next, we will create our namespace for this project. Write namespace, the name preceding it should be Character State Control as this namespace is going to do exactly as the name suggests. Control the Character’s state.

We are going to create a public interface called ICharacter. The naming is important as the I before the name lets people know that it is an interface instead of a class. An interface only contains the signatures of methods, properties, events or indexers. It cannot implement them.

What makes an interface useful is that you can have different implementations of the methods, properties, events or indexers within the different classes or Structs that inherit from it.

The method we want to create will be called void Handle Movement with a parameter of Vector3 called position. This method will handle the movement that we want generated and by taking the Vector3 information, we can manipulate the positions manually. More on that later.

I should note that in the recording, I create an Update method, this is not important as it is never used in this code. So you can ignore that part. The code base which will be located in my github will not contain the update method.

The ICharacter interface has now been completed. We can now move on to the Character class.

The character class only needs the using statements of System and Unity Engine, everything else can go bye bye. Write namespace and the name will be the same as before, which is Character State Control.

Public class Character can stay as it is, but we will remove the inheritance from MonoBehaviour.

Inside of the class, create a public Game Object and call it character Object. This will essentially be an empty object that we can use to directly be referenced, more on this in the final class we create at the end.

Create a private ICharacter instance and call it underscore state. We want and need a direct reference to the interface to allow for switching states.

We can move on to creating our constructor, but we will leave it empty for now. A constructor is used to initialize an instance of a class. Which we will make use of later on.

The next method is a public void and it will be called Set State. The parameter should be ICharacter with the name of new State.

Inside of this method, underscore state is set to new state. This allows for us to easily switch between the states.

The next method will be our update method. Write public void Update. Inside, write underscore state dot handle movement. The parameter will be set to be character object dot position. This will tell the handle movement method to update the object’s position as dot position is a Vector 3 value.

We now need two methods that will essentially be helper methods. Basically, helper methods are just regular methods that help you do something else.

The first helper method will be a private Vector3 called Get Direction. It will take a parameter of Vector3 called destination. If you haven’t already guessed by the naming convention of this method, it allows for us to determine the direction of an object.

Inside, write return opening parenthesis destination minus character object dot position with a closing parenthesis. Now write dot normalized.

Essentially, we are subtracting the destination from the character object’s position and normalizing the results. This will ensure that the length or the magnitude of the vector is always 1.

The last method within this class is public void Move Toward. It will take a parameter of Vector3 and it will be called destination. This will allow for our character object to move toward what we want it to.

Inside, write var direction is set to be Get Direction with the parameter of destination. Pretty straight forward, here. The direction is the normalized subtraction between the destination and the character object’s position.

Finally, write character object dot transform dot translate opening parenthesis direction times time dot delta time times 1 float.

This will allow for smooth movement and the 1 value represents our speed value. It will be able to move faster if we were to change that value.

Alright, save for the constructor, this class is complete. This isn’t exactly too complicated or confusing yet. But yes, we have not explicitly set a destination nor have we assigned the character object to any specific object. Moreover, we aren’t inheriting from Entity Component so this class won’t be used by any particular object. The plot thickens, but don’t worry. This will make more sense in due time.

We have three more classes to create and implement, so we have plenty of time to really figure out what’s happening.

Let’s implement the Wait State class next.

Again, everything can go poof as we don’t need it. Again, we will only be using the System and Unity Engine namespaces.

Our namespace will still be the same as before of Character State Controller.

Write public class Wait State and it will inherit from ICharacter. A pattern is definitely emerging where we try to name things according to what they are supposed to do and this class is no different. This class will implement our wait or transition state.

Allow for Visual Studio to automatically implement the interface members so it will stop yelling at us.

Inside of the class, the first thing we will write is private Character with the name of underscore character base state.

This will grab the reference to the Character class that we need.

Create a private float called underscore timer and set it to be the value of 0 float. We are going to be using this for a timer… obviously.

Write public Wait state with the parameter being character named character for simplicity. This is another constructor and we will be filling this one out.

Inside of the constructor write, underscore character base state is set to be character.

We are making sure that the state will correctly reference to the Character which should never be null… or we’re in trouble.

Next up, write Debug dot log and fill in the string value to be… I’m in the wait state!

This will write to the Unity Editor console for debugging purposes.

The next method to fill out is the Handle Movement method.

Timer plus equals time dot delta time. Which is the timer that will count the ticks and return it as a float value.

Write, if timer is greater than 5 float. Open and closing curly bracket, leave this blank as we will come back to it later.

Time to move on to implementing the Wander state now.

Starting from the point of deleting everything from the class, we will using the system and Unity Engine namespaces.

The namespace we will call this is again, Character State Controller.

The class is public and it is going to be called Wander State which will inherit from ICharacter. The theme just continues on.

We can now implement the ICharacter interface to stop Visual Studio from yelling at us.

Create a private Character with the name of underscore character base state. Which again we will need for referencing.

Create a private Vector3 called underscore destination.

Alright, now we can implement the Wander State’s constructor.

Write public Wander State which will take a parameter of Character called character.

Inside of the constructor, write underscore character base state is set to character. We have gone over the reasons already, so let’s move on.

Next up, write underscore destination is set to be Get Random Destination. Now we haven’t created this method yet so Visual Studio will yell at us for this and Unity will also complain. Just ignore it for now.

The final item is to write debug dot log and fill in the string value to be… I’m in the wander state!

Alright, we aren’t going to fill out the Handle Movement or Update methods quite yet. We have some helper methods to generate first.

The two methods we will create will be Get Random Destination and Destination Reached.

Let’s start with the first one. Write public Vector3 Get Random Destination. This method returns a vector 3 value. Now in Unity, the Z axis is the depth value. Remember that.

Inside of this method, write return new Vector3.

Write Unity Engine dot Random dot Range, write the values of 1 and 15. This will set the X value.

We are going to hard code the Y value to be 0 float.

Write Unity Engine dot Random dot Range, the values will again be 1 and 15. This will set the Z value.

The purpose of this method is to return a random vector 3 value for us to use.

The last helper method is Destination reached.

Write public bool Destination Reached and it will take a Vector3 value called current position and a second Vector3 value called destination.

Write inside of this method, return Vector3 dot Distance with the parameters being current position and destination is less than 0 point 5 float.

This checks if the distance between the current position and destination has a value of being less than 0.5. It will return true if it is and returns false if it isn’t.

Let’s take care of the Handle Movement Method that is inherited from the ICharacter interface.

Inside of this method write, if Destination reached with the parameters being position and underscore destination.

Within the if statement write, underscore character base state dot Set State create a new Wait state with the parameter being underscore character base state.

This is where the magic happens for switching states. Once the Destination Reached Boolean method returns true, it will switch from the Wander State to the Wait State.

Outside of the if statement, write underscore character base state dot Move toward with the parameter being underscore destination.

This class is now complete, so let’s go back and finish filling out the wait state.

Over in the Handle movement method in the wait state, we had an if statement that we didn’t fill out.

Inside of that if statement write underscore character base state dot Set State create a new Wander state with the parameter being underscore character base state.

That fixes that little issue we left behind in the Wait State class, so let’s head over to the Character class to finish that one off.

We created a constructor for the Character class but we did not fill that one out. Inside simply write underscore state is set to be new wander state with the parameter being this.

We have one final class to implement. That would be the Character Behavior class.

Unlike the others, we will keep everything as per how the template loads a new script in Unity.

At the top of the class write, using Character State Control. This will give us access to everything that is public we defined in the methods associated with that namespace.

Inside of the class the first thing we want to do is create a private Character named underscore character. This will grab the reference we need for the character class.

In the start method write,

Underscore character is set to be a new Character.

The next line we write underscore character dot character object is set to this dot game object. We are explicitly saying that the Game Object we created in the Character class is this Game Object that we are using in the editor.

In the update method, simply write underscore character dot update.

That’s it. The code is done! Now all we have to do is create a new level and attach this script to an object that we want to represent the player.

Import a model you want to use into the engine. (Psst: look at the top left eye card in the top right. Follow this link and you can use one of the models from the last video tutorial for this part).

Drag and drop the model into the scene and place the position of the object to be at 0 for the x value, 0 for the y value and 0 for the z value.

Drag and drop the Character Behavior script onto the model and press the play button. You can now watch the model move between the wander and wait states automatically. If you click on the console within the Unity Editor, you will see when the object switches states.

Whew, this video was far too long, so the CryEngine portion will be in a separate video of its own so click on that video that should be showing up on the screen right about now. I hope to see you guys over there.

(outro)