Salutations, I’m Jesco and you’re watching Game Dev Made Easy. In the video today, we will be looking at programmatically creating a UI in CryEngine. There is a tutorial on creating a basic UI on the CryEngine website, however, it needs to be updated due to changes that were made since that tutorial was written. This is not to say that all the tutorials or documentation on the CryEngine website are out of date, far from it. The documentation team is hard at work overhauling the entire website and it will take some time to update everything that was created. I am doing my part to help with the update process, so if you ever see any of my tutorials there, you know why. (Hint hint Crytek, steal my work!)

This video will feature usage of Resharper, which is a tool designed by JetBrains and is the ultimate tool for any developer to make sure they conform to the latest standards of many programming languages.

ReSharper delivers early support for Visual Studio 2019, inline parameter name hints, and auto-detection of formatter and C# naming settings. Improved code analysis includes better C# 7 deconstruction support, better C++/CLI support, VB.NET 15.3 and 15.5, TypeScript 3.0, and lots of new inspections, quick-fixes, and context actions.

Resharper gives on the fly code quality analysis for C#, VB.NET, XAML, ASP.NET, JavaScript, TypeScript, CSS, HTML and XML. It has multiple code editing helpers, code generation, compliance to coding standards, can instantly help eliminate errors and code smells, safely change your code base, instantly traverse your entire solution, Unit Test runner, code templates, debugging assistance, project dependency view, internationalization assistance and so much more.

Resharper offers a free 30 day trial for new users and to purchase for one year cost $129 or $12.90 for a monthly subscription. For every yearly subscription, you get a perpetual fallback license.

Get Resharper today, it will honestly help you to become a better developer. In order to conform with ethical standards, I will be up front and tell all of you that I was given a year license from Jet brains to promote their product, I do not receive any kickbacks for purchases made by anyone watching my content. Link is down below!

(Intro)

First off, the UI component class in C# cannot be built inside of the CryEngine editor because it needs to use the Game class. The Game class is only accessible when you build the solution the old fashioned way as it is a dll file that is used in the CryEngine Editor.

Hopefully, Crytek will update the C#API to allow for it to be built in the sandbox editor later on, but as of writing this tutorial, this is not the case.

Create a new C# project in the CryEngine launcher and I will call it UI Basics Tutorial. Instead of opening the Sandbox editor, we will locate the location on disc that the project is stored.

Look for the file named Game that has the blue CryEngine logo on it and right click on it. The sixth item down is Generate Solution, click on it. It will open a cmd prompt and will display information about it building the solution. Once the cmd prompt closes, navigate inside of the Code folder. You will see a Game visual studio solution. Double click on it to open it up in Visual Studio. Now, we can begin writing code and have fun!

(Transition)

There are two classes in this solution. Game and program. Program is to be completely ignored for this video, perhaps I will talk about it in a later video that makes it relevant. Just know that this is for plugins.

The class we will focus on is the Game class.

Delete everything from the Game class, yes, everything. We are going to rebuild this from the ground up.

The first thing we will write is namespace CryEngine dot game. This is the exact namespace required for this to work and should not be changed unless you want to change the namespaces for everything to match.

Type the opening and closing curly brackets for the namespace.

We now will add the using statements we need for this solution.

We need a using statement for System. Which is needed for implementing I Disposable.

We need a using statement for CryEngine dot Resources. This namespace allows for us to utilize the Resource Manager which holds a reference to all image source objects.

We need a using statement for CryEngine dot UI. This will give us access to Scene objects which is used with UI Components.

The last using statement we need is CryEngine dot UI dot Components. Which gives us access to all of the components that we need.

Write public class Game with a colon to designate inheritance. We will inherit from I Game Update Receiver and I Disposable. I Game Update Receiver is an interface that receives update calls after the engine has updated. I Disposable is an interface that provides a mechanism for releasing unmanaged resources.

Visual Studio is going to complain because we have not implemented the members of those interfaces. Don’t worry about it as of yet.

Add an opening and closing curly bracket for this class and we can begin adding our implementations.

Write private static Game, call it instance. This will provide an instance of this class that we can utilize for referencing when we need it.

Write private Game with an opening and closing parenthesis. Add an opening and closing curly bracket. This is our constructor for our Game class.

Inside of the Game constructor, write Game Framework dot Register for update with the parameter being this. Game Framework dot Register for update will allow for the object that we call to be invoked within the on update method if CryEngine’s Game Framework raises on post update calls internally.

Write, Mouse dot show cursor. This will allow for the cursor to be seen inside of the game window.

The next item to add is Engine dot Engine reloaded plus equals Create ui. Engine dot engine reloaded is an event that is called when the sandbox reloads the managed plugins or after the mono underscore reload command has been executed from the console. The plus equals specifies that we are utilizing the event or delegate syntax and create ui is a method we will create soon enough.

The item within the constructor is to call our currently non-existent Create ui method. This will house all of the code needed to create our UI items.

We have finished the Game constructor and we can move on. Again, pay no mind to the fact that Visual Studio is complaining with errors.

The next method we will create is a public static void Initialize method. This method is a quality of life check. We want to check to see if our instance is null and if it is, to do something.

Write, if instance is equal to null. Inside of the if statement, write instance is set to new game.

This essentially will check if the instance is null to create a new game which will ensure that the game should run if the instance is null upon startup.

This completes the Initialize method, we can move on to our next method.

We will now implement our first method that is required from our inherited interfaces. Write, public void Dispose.

Inside, write Game Framework dot Unregister from update with the parameter being this.

This method unregisters the object from on post update.

The next method to create is private static void create ui. This is the portion of code that all of you have been waiting for, and I won’t disappoint you… hopefully.

We will declare all of our variables first. The first variable is var canvas and it is set to be scene object dot instantiate with the generic type parameter being Canvas. Inside of the parameters write Scene Manager dot Root object.

What is happening here is that Scene object is a representation of any single entity or logical object in the scene. Instantiate creates an instance of any generic type of object and wires it into the scene hierarchy. Canvas is the root element for the UI and there are no restrictions to the amount of UI entities it can hold. Drawing of UI elements to the screen or rendered textures are handled by the canvas. We only need one Canvas to handle all of this.

The Scene Manager handles the prioritized and ordered updating of Scene Objects and components. Root Objects is the parent object that is created by the Scene Manager, which should pretty much always be the canvas.

The next item is logo element is set to be Scene object dot instantiate with a generic parameter argument being UI Element and the parameter being canvas. The UI element is the base class for all UI elements and provides a rectangle transform for layout computation.

We will create a var logo image is set to be logo element dot add component with the generic type argument being image. Add component adds a new component of type T, which is a generic, to an existing object. Image is a view for one image and is aligned relative to its own element.

The last one is var quit button which will be set to be Scene object dot instantiate with the generic type argument being Button and the parameter being set to canvas. The Button is an element that represents a simple button.

These four items will represent the basis for the whole UI we will be creating.

The first one we will expand upon is the logo element.

Write, logo element dot rect transform dot size is set to be a new point with the parameters being 300 float for the x value and 300 float for the y value. Rect transform dot size defines and computes the layout for this UI Element as well as sets the element’s size. A point is a 2D representation of the screen space.

Logo element dot rect transform dot alignment is set to be Alignment dot center. This sets the alignment of the UI Element around the target, which we know to be the canvas.

The logo element has been fully set up and we can move on to the logo image.

Write, logo image dot source and set it to be Resource Manager dot Image from file. The parameters will be UI Element dot Data directory plus in quotes slash GDME dot png, end quotes, comma true.

The resource manager holds a reference to all image source objects that have been instantiated through it to avoid multiple instancing. Image from file creates an image source if it doesn’t already exists and hands out the existing one if it does. UI Element dot data directory is the path in which UI System textures are stored and we specify that the file we want is the GDME dot png file. The second parameter is whether or not the file should be filtered.

We have completed everything to do with showcasing the logo image and can now build our quit button’s implementation.

Write quit button dot rect transform dot alignment is set to be Alignment dot bottom. We are setting the quit button to be at the bottom of the screen.

Quit button dot rec transform dot padding is set to be new padding with the parameters being 0 float and negative 160 float. This will pad the bounds of the button to be on the negative scale in the upper left buffer only.

Quit button dot rect transform dot size is set to be a new point with the parameters of 300 float for the x value and 30 float for the y value. This will set the quit button to be 300 pixels wide and 30 pixels tall.

We have completed the layout of the Quit button, now we can work on the text of the button.

Write quit button dot ctrl dot text dot content is set to be Quit. Ctrl is the controller of the button, text is the content of the button and the context is the actual text to be displayed.

Quit button dot ctrl dot text dot height is set to be 18. This sets the height of the text to be 18 pixels tall.

Quit button dot ctrl dot text dot drops shadow is set to be true. This determines whether the rendered text is shadowed or not.

Qut button dot ctrl dot text dot alignment is set to be alignment dot center. We will center the text of the button.

The final thing to write in this method is quit button dot ctrl dot on pressed plus equals on quit button pressed. This is an event handler that is raised if the button is pressed. And we will go ahead to implement that next.

Write private static void on quit button pressed. Inside, the only thing we need to write is Engine dot shutdown. This will shutdown the engine. There is a bit more about the Engine call, but will require a lot more to really talk about it.

Next up, write public static void shutdown. Inside of this method write, instance with a question mark dot dispose. This does a nullable type check on the instance field and will dispose of it.

The last item inside of this method is to set the instance field to be null.

Lastly, we need to create a public void on update method. This method should have absolutely no implementation to it as we don’t need it with this tutorial.

Build the solution by pressing the control, shift and b keys simultaneously. Or you can just press build button inside of the build menu. All compiler issues should have been resolved and the project should build just like mine has. You have the option of pressing the green arrow to build and run an instance of CryEngine, or you can click on the gamepad option in the CryEngine launcher on the project.

Before we can run the project however, we need to add the image file into the project. To do this, go back to the project directory we opened earlier. Click on Assets, click into libs and then click to go inside of the UI folder. Copy and paste the image file inside of this directory.

Now, click back to pull up the CryEngine launcher and select the button that looks like a game pad. This will launch the project we already compiled. And there we have it, we can see our UI working properly.

As we can see, we have a UI being displayed that has the Game Dev Made Easy Logo with a button that says Quit, and if we were to click on it, the program would shutdown as expected.

CryEngine’s C# implementation of building a UI is a bit more convoluted than other solutions out there because of the need to specify everything you want explicitly. However, this was a new feature that was added relatively recently and still being actively worked on. If I were to be asked which method of building a UI would I choose to use, Scaleform API or C#, I would probably choose Scaleform as it is a bit more mature and has more features built into it at the current time.

Overall, I would rate this option to be average at a 5 out to 10 rating. Decent start but needs more work.

If you guys like my work, consider donating to me via patreon just like Coburn and Coopala, thank you very much guys or directly via paypal. If you like the video, give it a thumbs up and if you didn’t, you know what to do. Now, I’m gonna go play some Resident Evil 2 Remake to try to dissect some of the mechanics in the game.