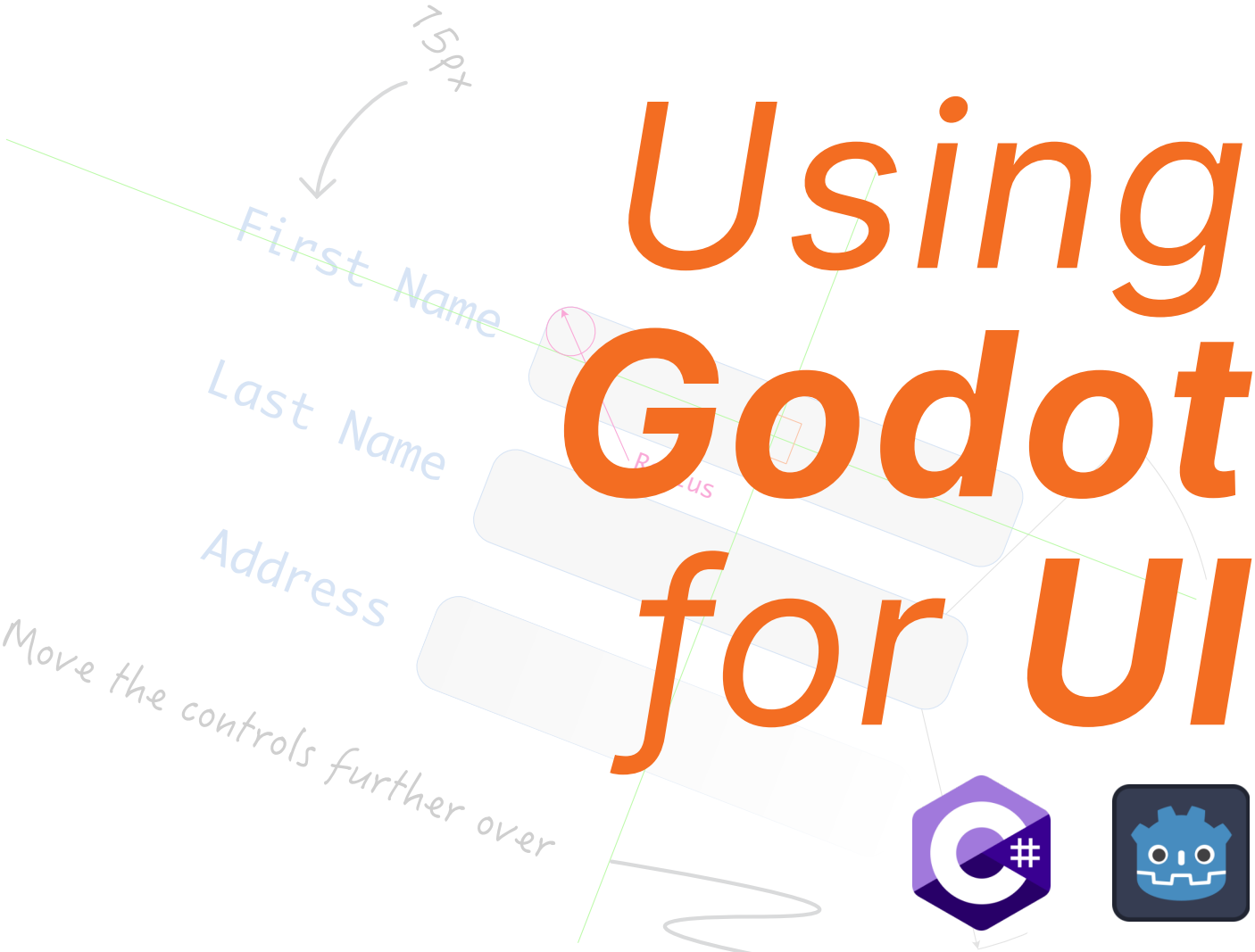
Using Godot for UI



Can a game engine really help with cross-platform user interfaces?

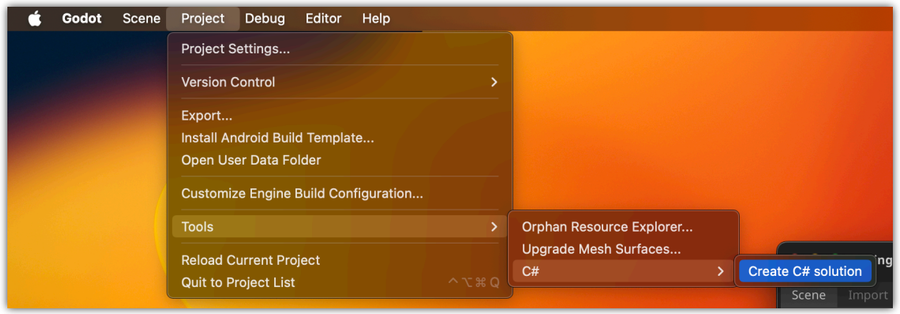
Cross-platform user interfaces can be challenging. There are many ways of producing user interfaces that run on multiple platforms – both desktops and mobile platforms – but they can have varying degrees of complexity and tradeoffs.

Godot is a cross-platform game engine that can export its output to desktop and mobile platforms. It is free and open source, and is very light weight. It comes in a standard version and a .Net version, which we will be focused on for this article. It can be downloaded from the Godot website[1] (godotengine.org). Further, we are only focusing on desktop user interfaces in this article.

Godot is a tool for designing 2D and 3D games. Games have user interfaces and those interfaces are very different from the regular desktop interfaces we are used to on Windows and MacOS. There is a learning curve when learning Godot, but when looking at user interfaces specifically, it does not seem to be any more significant than learning another user interface technology like MAUI or any of the Electron-based UI solutions.

This article will pursue a specific use case for needing a cross platform user interface. It will employ the Godot Engine to handle the user interface and back that with C# programming that makes use of that UI.

This article is not a Godot tutorial, nor a tutorial on C#. It is a guide on using a Godot UI connected to a C# project. We will set up a simple UI using some of the built-in Godot UI components. After that, we will explore a more exotic UI that has a non-rectangular window and buttons.



The code for this article can be found on GitHub[2] (the link to which is at the end of the article).

Setting Up

The first step is to set up the Godot project. Create a project called “Using Godot for UI”. After the project is created, create the C# solution (Project → Tools → C# → Create C# Solution). After creating the solution, build the project.

Now create a new scene named “UI” based on the **User Interface** node, name the node “UI” and save it in a “scenes” folder inside Godot. This will be the starting point for a simple UI using the Godot built-in Godot UI components.

Simple UI

Creating a simple UI is straightforward and can be accomplished easily using Godot’s own UI components. Godot allows for simple or complex theming of the components it provides, but that is beyond the scope of what we are covering in this article.

Though this is not a full tutorial on the UI components Godot provides, it does help to understand a little about how Godot lays out controls.

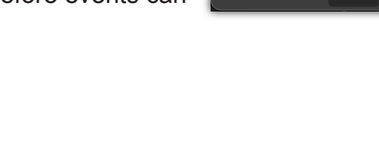
Add a **VBoxContainer** node to the UI node you have already created. This node will be displayed as a small square in the top left of the UI view. Controls added as children of this node will flow vertically down the UI.

There is also a corresponding **HBoxContainer** for controls that need to flow horizontally. These **VBoxContainer** and **HBoxContainer** nodes can be nested. There are also other container controls, and combining them in various ways can lead to an endless variety of layouts.

For our simple example, we will just add a **Label**, **LineEdit** and **Button** control (in that order) as children of our **VBoxContainer**. In the **Label**, set its *Text* property to “You Typed”. In the **LineEdit**, set the *Placeholder Text* property to “Type Here…”. And finally, set the **Button’s** *Text* property to “Click Me”.

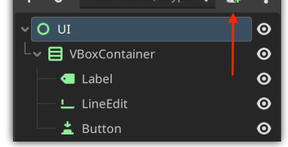
We do not want all of those controls to be in a tiny cluster at the top left our of window, so select the **VBoxContainer** and in the Layout area select “Full Rect”.

If you run the project now you will see a simple UI with a label, a text area and a button. (You may need to make the current scene the startup scene the first time you run the program.)

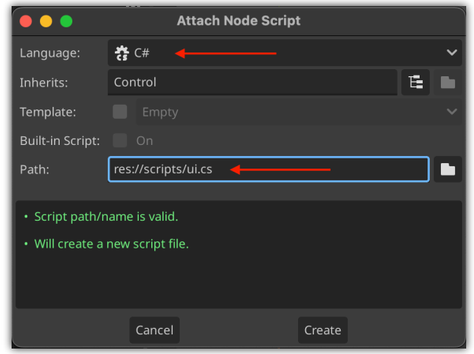


Connecting Events

Create a folder named “scripts” under the “res://” item in the **FileSystem** section. This is for organization only and has no impact on the functionality of the code. Click the UI node and click the button to attached a script to the node. In the dialog that comes up for attaching a script to the node, ensure that it is a C# script, inherits **Control** and it is stored in the “Scripts” folder as “ui.cs”.



Click the “Create” button and your configured editor for C# scripts will be launched. The editor can be configured in the **Editor Settings** area, but doing so is not covered in this article.



Before events can be handled by a C# script, they first need to be connected. In our simple example we want to take the text a user types into the **LineEdit** and put it in the **Label** when the user clicks on the “Click Me: **Button**.

We will have to get a connection to the **LineEdit** control to be able to read its *Text* property in order to add it to the **Label’s** *Text* property. That is done by creating a variable to hold a reference to the **LineEdit** and **Label** controls control, then assigning those references in the “_Ready()” function override.

Note: The assumption that the user has knowledge of using basic Godot events is taken in this article. There are numerous tutorials on how to use Godot events. The Godot documentation is also a great reference for using events, and has sample code in both C# and GDScript[3].

We have not taken the liberty of giving our controls descriptive names in this simple example application. We are still referring to the controls by their default names of “Label”, “LineEdit” and “Button”. For a more complex UI this is definitely not the best practice, but instead the controls should be named appropriately.

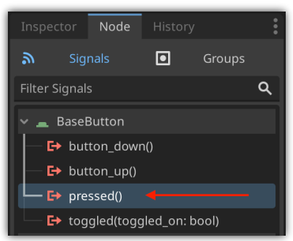
Add a private variable to the UI class of the type **LineEdit** and name it "_lineEdit". Also add a variable of type **Label** and name it "_label", In the "_Ready()" function we will get a reference to these controls and assign them to the variables we created. This will allow us to get and set the properties of these controls. We do not need a reference to the Button because we will actually never modify any of its properties. We only need to respond to its signal.

```
using Godot;

Namespace UsingGodotForUi.scripts;

public partial class ui : Control {
    private LineEdit _lineEdit;
    private Label _label;

    public override void _Ready() {
        _lineEdit = GetNode<LineEdit>("VBoxContainer/LineEdit");
        _label = GetNode<Label>("VBoxContainer/Label");
    }

    public void _on_Button_pressed() {
        _label.Text = "You Typed: " + _lineEdit.Text;
    }
}
```

Next we have to create the function that will respond to the "pressed()" signal that Godot will emit when the **Button** is clicked. This function should be private, return void and we will call it "_on_Button_pressed()". When the **Button** is pressed, the function will be called. It will then take what is in the **LineEdit's** Text property, prepend it with "You Typed: " and put that in the **Label's** *Text* property. Inside the function, use the references to the **Label** and **LineEdit** controls we assigned in the "_Ready()" function and create the string that will be shown in the **Label**. Return to Godot and build the project.

If the Godot project were to be ran now, there would still not be any functionality when the **Button** is pressed. The *signal* from Godot is not yet connected to the function we want to execute. Connecting the "pressed()" signal to the "_on_Button_pressed()" function is done from Godot, and its *Signals* management.

To attached the signal, select the **Button**, in the Node panel (by default it is near the Inspector) make sure Signals is selected, then click on the "pressed()" signal. Now click the "Connect" button at the bottom of the list of signals and a dialog will open that allows connecting a signal to a function that will handle that signal. In the dialog that is shown select the "Pick" button at the bottom right. This should show the methods in our attached C# script. Chose the "on_Button_pressed()" method and click OK. This will return to the previous screen showing the nodes. Click the "Connect" button on this screen and the signal should now be connected to the function that will handle the event.



If the Godot project is ran now, the functionality should work as we expect. Anything typed into the **LineEdit** should be displayed in the **Label** when the **Button** is clicked.

A More Interesting Button

What if we wanted an irregular shaped button? What if our particular project called for a little more pizzaz and a regular, rectangular button would not do? Furthermore, what if we wanted to lay these irregular buttons out in a way that would cause rectangular bounding boxed to overlap one another? And we wanted hover and pressed states? Can we do this?

Yes!

But we will have to combine a few varied pieces of knowledge to accomplish task. We need to understand some graphics processes, be able to design a simple button using vectors, and get a little more under-the-hood with Godot and scene organization.

Using Godot, we can accomplish this with relative ease compared to other UI toolkits that would normally be used for desktop applications. The outline of what needs to be done is:

- Understand a **Texture Button**
- Create vectors for our button states
- Create a Godot scene that contains the new button's functionality

Understanding a Texture Button

Godot has a control called a **Texture Button** that provides what we need to make irregular shaped buttons. This is accomplished by using the *Texture* properties of the button, and filling them with the look we want for our new button. The texture properties we will use are *Normal*, *Pressed*, *Hover*, *Disabled*, *Focused* and *Click Mask*.

The texture properties of the **Texture Button** correspond to the states we want our button to be in. Godot handles swapping the textures from the button based on the texture properties we utilized. For example, if we only define the *Normal* texture property, the button will always appear with this texture.

A texture in this context is just an image. It can be a simple color, or could be an actual photograph of an actual button of some sort. If we use an intricate image for the button and ever need to change it's color, we would have to replace the texture and rebuild the application. But we want our button to be more flexible in allowing color choice, so we will use another property of the **Texture Button** called *Self Modulation*.

If you've ever played a video game and saw your character flash a bright color after taking damage, or some part of the UI to flicker or pulsate colors, you have probably witnessed modulation. Modulation works by taking the colors in the texture and combining them with another, specified color. Where the original texture is white, the color will turn into the combined color. Where the original texture is black, no color change will take place. Colors in between are mixed to varying degrees with the combined color. This allows for textures to have their colors modified without having to rebuild a project or their texture assets.

The image with the three C# logos shows how modulation affects the texture. The first logo is the original. The second has been modulated with pure red (FF0000) and the last logo

has been modulated with pure green (00FF00). The white ‘C’ in each logo shows the pure color that was modulated with the logo (red and green respectively). The other colors have been mathematically combined to product varying degrees of the red or green modulation.

Leveraging our knowledge of modulation, we know know that we can design a texture that can produce an interesting button, but still allow some flexibility in colorizing that button. If we design our button in grayscale, we can the modulate that button with whatever color we choose in the UI and it will take on that hue.

Vectors for the New Button

Creating vectors using design software is completely out of scope for this article. But some discussion of the textures that need created will be covered. We will make our button use a star shape. This will allow demonstrating irregular shaped buttons how to set them up in Godot.

The star button design will have one SVG for each texture property. All the SVGs will be the same size, but the stars will be filled with gradients that are slightly different. Doing this will allow one color to be chosen, then that color will be modulated with the button’s texture that correspond to the state. This will give some visual variance to the button.

***Normal* Texture**. This is the texture for the normal state of the button when it is not in any other state.

***Pressed* Texture**. This is the texture for the button when it is pressed.

***Hover* Texture**. This is the texture of the button when it has been pressed.

***Disabled* Texture**. This is the texture of the button when it has been disabled.

***Focused* Texture**. This is the texture of the button when it has been focused, but is not pressed or disabled.

These button textures vary only in their internal gradient intensities. The lighter button shapes, for example the Hover state, will be a more intense color than the darker textures, like the Disabled state.

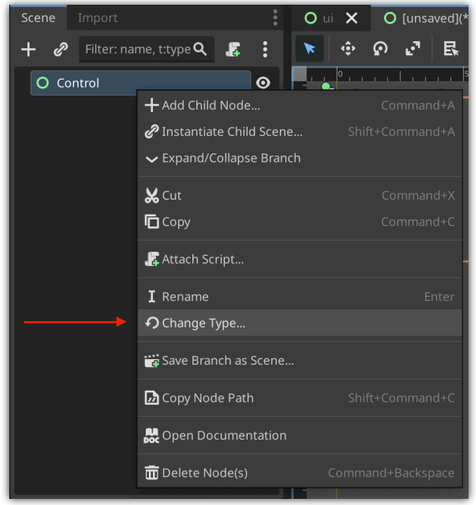
The textures are included in the code repository, but the artistic reader may choose to make their own button textures if they desire.

Creating the Button Scene

Create an “assets” folder along side the “scenes” and “scripts” folder in the Godot project. This will hold the button textures that will be used to create our texture button.

Next create a new scene based on the User Interface type. Select the single root node of this scene and then change its type to a **Texture Button**. Rename this node to “StarButton”, and save this scene in the “scenes” folder as “star_button.tscn”.

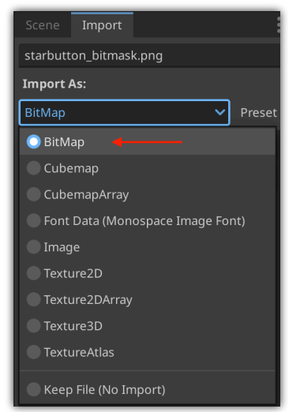
Now we need to add the textures to the button. Do this by dragging the corresponding SVG texture to the proper texture slot. For example, drag the “normal” texture to the *Normal* texture slot.



Note: The size of the SVG images will affect the size of the buttons. If using the code from the repository for this article, the SVG images were scaled to only 10% of their original size when they were imported.

Next we want to define the *Click Mask* texture. This must be a BitMask. To get this format, we have to import an image (a PNG in this case) that is the same size as our other textures that defines the shape of our click area. The white areas of the image will register button clicks, and the black areas will not. Godot will generate this image for us if we use a PNG and tell Godot to import it as a BitMask.

To do this, we need to click on the image that we will use as the mask, then in the Import tab near the Scene tree, we will select “BitMap”, then reimport the image. After this, drag the image to the *Click Mask* property of our StarButton. Save the scene again.



Now we set the *Self Modulate* property to the default color we want to use for our button. You may choose any color you prefer, but the sample code uses a light blue (65BFD0).

Now attached a script to he StarButton node. Make sure it is a C# script that inherits from TextureButton and is named “star_button.cs” saved in the “scripts” folder.

Setting properties in the script that can be modified from the Godot interface will make things easier for us in working with our UI controls. We are going to set a property called ButtonBaseColor that will be used to set the base color of the instance of this button when we use it in our UI. We will default the value to the same value we set the Self Modulate color to above (again, in the example code this is 65BFD0).

To expose this property to Godot we decorate the variable declaration with “[Export]”. When the control is used in other scenes, we can now use this property to define the base color of the Star Button. We have to set the Self Modulate property in the “_Ready()” function to make this color value take effect.

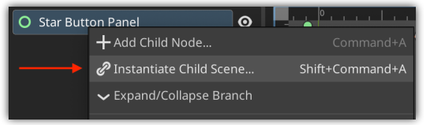
(The Godot UI itself will not show the selected

```
public partial class star_button : TextureButton {
    [Export] public Color ButtonBaseColor { get; set; } = Color.
FromHtml("65bfd0");

    public override void _Ready() {
        SelfModulate = ButtonBaseColor;
    }
}
```

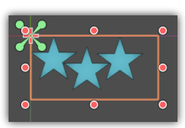
color because this script is not executed during design time. There is a way to reflect the changes using a Tool script, but we will not implement that functionality in this article.)

Creating Some Stars

Now we will create another scene and add multiple Star Buttons to it. Create a new scene that is a User Interface type. Rename the single root node from Control to Star Button Panel. Set the *Custom Minimum Size* property to 200px by 100px. Save this scene in the "scenes" folder as "star_button_panel.tscn".



Create a child of the Star Button Panel by instantiating the "star_button.tscn". This will create a Star Button as a child of the Star Button Panel. Duplicate the Star Button two more times and arrange the Star Buttons so their arms slightly nest inside each other.



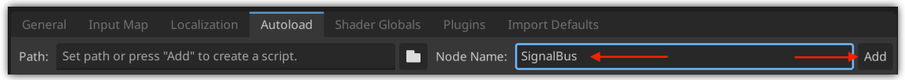
Save the scene again. Now we will return to the UI scene we created earlier and instantiate the "star_button_panel.tscn" as a child of the **VBoxContainer** and place it below the **Button**. Now run the project and the three Star Buttons appear on the UI. Move your mouse over the buttons and notice that they only lighten and show the hover state when the mouse is over the star shape itself, and not when it is between the stars. The *Click Mask* used on the star **Texture Button** has made the button interaction not overlap.

Signaling Across Scenes

But how are we going to connect the "pressed()" signal from the Star Buttons to the script in the UI scene? You can't click on any of the Star Buttons and attach their "pressed()" signal to the UI's script.

We will use what is called a *signal bus*. In our application, the signal bus will act as a globally available switchboard that will allow different parts of the application to send and receive signals from other parts of the application.

Godot has a mechanism to help with this called "Autoload". The Autoload settings are in the Project Settings on the Autoload tab. In this dialog, put the name "SignalBus" in the Node Name field and press the "Add" button. A dialog appears where some information about the script can be set, so ensure it is a C# script that inherits from Node, that it is in the "scripts" folder and is named "signal_bus.cs". Click "Create" and the script will be shown in the list, and should be enabled. (If it is not enabled, select the checkbox to make it enabled).



The signal bus script will contain the definitions of the signals we want to make available to all scenes that may have an interest in them. These signals can be emitted by any code in the project, but we will use them to echo the "pressed()" signal from our Star Buttons.

The first step is to define the signal we want the Star Button to emit globally when pressed. Signals in Godot in C# are defined using *delegates*, and also have a name that ends with *EventHandler*. We will define a signal named StarButtonPressedEventHandler in the signal bus, and that signal will pass the name of the Star Button that was pressed as a parameter.

```
[Signal]
public delegate void StarButtonPressedEventHandler(string
                                                   buttonName);
```

The next step is to undertake is to connect the Star Button's "pressed()" signal to our "star_button.cs" script. Do this as we've done before in this article by creating a function in the Star Button script named "_on_Button_pressed()", and attaching the "pressed()" signal from our Star Button to this function.

```
private void _on_Button_pressed() {
    var signalBus = GetNode<signal_bus>("/root/SignalBus");
    signalBus.EmitSignal(signal_bus.SignalName.StarButtonPressed,
                         Name);
}
```

Inside this function is where we re-broadcast the signal on the signal bus. Get a reference to the global signal bus and assign it to a variable (in this instance it is named "signalBus"). Then using that variable, emit the signal we defined named "StarButtonPressed" and pass the name if this StarButton as the signals single parameter. You can see that the signal name "StarButtonPressed" was derived from our delegate automatically and has had the "EventHandler" part of the delegate named removed. Godot wires this up for us.

So far only half of the work has been completed. Running the project now should work, but nothing is listening to the signals the Star Buttons are emitting. The global "StarButtonPressed" signal needs a listener somewhere to respond to the signals being emitted.

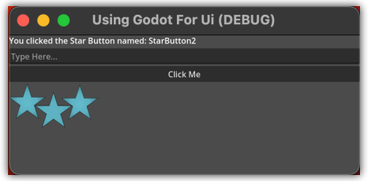
We will make our UI scene fill in the **Label** with the name of the star button that was clicked. Whenever a star button is pressed, the Label will be changed to "You clicked the Star Button named: <Name>". This should prove that we are receiving clicks from any of our Star Buttons. It also shows that we only need to have one listener for the event, and not one listener for each Star Button individually.

Return to the "ui.cs" script so we can

modify it to handle clicks from our Star Buttons. Create a method that will handle the StarButtonPressed signal. This is just a private void function named “_on_star_button_pressed”. It should take one string parameter we will call “buttonName”. In this function we will set the *Text* property of the **Label** to “You clicked on the Star Button named: <Name>”.

Because the signal we are listening for will be emitted from our global signal bus, we need to get a reference to the signal bus in our “_Ready()” function. After that, we can connected the signal to our handler function in “ui.cs”. Getting a reference to the signal bus is done exactly as it was when we emitted the signal from our Star Button script. But this time we add a handler function to the signal instead of emit the signal.

Running the project now should show that clicking any of the Star Buttons results in the name of that button being displayed in the *Text* of the **Label**.



Now we have made irregular shaped buttons with irregular shaped click areas that can communicate across scene boundaries. This opens a lot of potential for developing user interfaces that are exciting or non-standard. Think of a process control room in a factory. The UI of an application could be created that allows interacting with it in ways that would very complex for traditional desktop UI frameworks. Being able to leverage C# to integrate with existing libraries allows complex, exciting and interactive user interfaces (yes, Godot can handle animation easily) to interact with existing systems.

But what if we wanted to go completely over the top and not only have irregular button shapes, but also wanted the application window itself to have an irregular shape? How would we move it around the screen? Godot can handle this for us, and can even do it in a cross-platform compatible way!

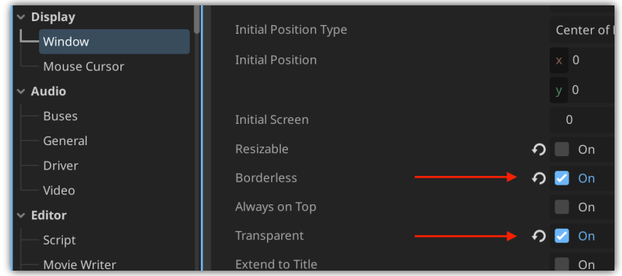
```
public partial class ui : Control {
    private LineEdit _lineEdit;
    private Label _label;

    public override void _Ready() {
        _lineEdit = GetNode<LineEdit>("VBoxContainer/LineEdit");
        _label = GetNode<Label>("VBoxContainer/Label");

        var signalBus = GetNode<signal_bus>("/root/SignalBus");
        signalBus.StarButtonPressed += _on_star_button_pressed;
    }

    private void _on_Button_pressed() {
        _label.Text = "You Typed: " + _lineEdit.Text;
    }

    private void _on_star_button_pressed(string buttonName) {
        _label.Text = "You clicked the Star Button named: " +
                      buttonName;
    }
}
```


We can have that star shaped window running on Mac and Windows (and Linux), and still develop only in Godot and C# without having to worry about different code for different platforms.

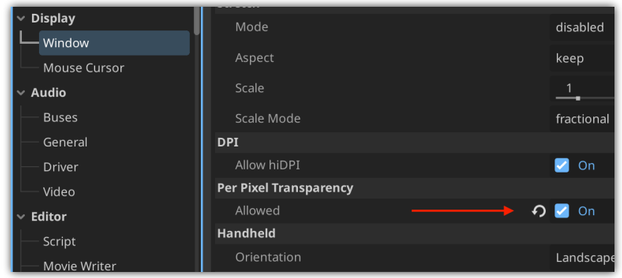
Star Window

Now we will create a star shape and use that as our applications window. But to get the window to actually have the shape we desire requires quite a few change to the project settings. We will do those changes first, so open the Project Settings dialog.

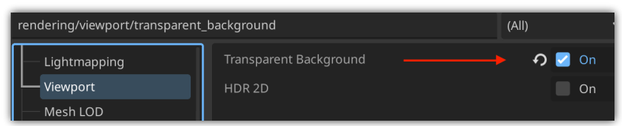
We need to set some properties under the Display → Window section. Enable *Borderless* – this allows our window to exist with no title bar and decorations. Also enable *Transparent*, so our window can be transparent. (Here we have also disabled *Resizable* because there will be no window border to drag for resizing.)



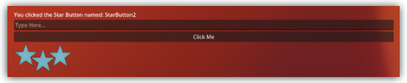
Still in the Window section, scroll down and enable the *Per Pixel Transparency* property.



Finally, go down to Rendering → Viewport and enable the Transparent Background property, which will allow the entire window background to be transparent.



Now if you were to run the project, the window would be transparent (except for the controls, of course) and have no title bar or decorations. You can’t move or close the application without using the Godot editor to stop the project from running, or force quitting the application. You could use the applications menu on Mac and probably quit the app using the task bar in Windows as well.



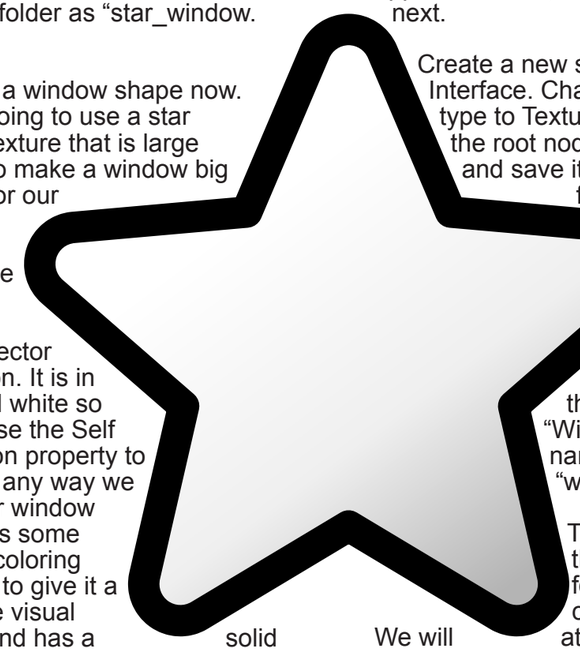
If you ran the application in this way, you will notice that it still functions exactly as we expect. Aside from being able to manage the applications existing on the screen, it is fully functional.

But it has no shape at all, it just has a transparent background. Giving the window a shape involves more designing with vector software. We

can be as complex or as simple as we like. In this article we are going to keep things simple.

We will create a new scene that is based on a 2D Scene instead of a User Interface. Create this scene and rename it's root node to "StarWindow". This root node is a Node2D. Now save the scene in the "scenes" folder as "star_window.tscn".

We need a window shape now. We are going to use a star shaped texture that is large enough to make a window big enough for our simple example. The shape was created using a vector application. It is in black and white so we can use the Self Modulation property to color it in any way we want. Our window shape has some gradient coloring internally to give it a little more visual appeal, and has a solid black border with will be unaffected by any color we use for modulation therefore it will remain black.



Import your window shape into Godot and drag it to the screen in our Star Window scene. Godot will create a Sprite2D for us and make it a chile of the "StarWindow" node. Try to center the sprite in the window area, but there is no need to be too exact as anything outside this star shape will be transparent. In the sprite, go to the Canvas Item → Visibility section and set the Self Modulate property to any color of your choosing. (For the example code, we chose a green color – 76B82C.) Save the scene.

Before moving on, go into the Project Settings and change the startup scene to our "star_window.tscn" scene. Run the project and there should be a green star on the display. You can't move it around or close the application now, so we shall fix that next.

Create a new scene that is a User Interface. Change the root node type to Texture Button. Name the root node "WindowClose" and save it in the "scenes" folder as "window_close.tscn". Repeat those same steps for another scene but this time name the node "WindowMove" and name the scene "window_move.tscn".

These scenes will be the controls we use for moving and closing the window. We will attach scripts to them to handle the moving and closing functionality, as well as giving them a texture so they be visible in our UI.

This simple example only uses a single texture for each of the WindowClose and WindowMove **Texture Buttons.** Different textures for the different states can be used as before to get the hover, pressed and other states. The textures in the example application are named "windowclose.svg" and "windowmove.svg". These were imported into Godot and assigned to the *Normal* texture property of the close and move buttons respectively. Each of the controls had their *Self Modulation* property set to a color (reddish DA0029 for the "WindowClose" control, and a bluish 2F3DFF for the "WindowMove" control). Now that the close and move buttons are created we need to react to the interaction with these buttons.

Select the "WindowClose" scene and attach a script to the root node. Ensure it is a C# script, inherits from **TextureButton**, and is saved in the "scripts" folder as "window_close.cs".

The code uses the "gui_input()" signal to handle the functionality of closing the window. Wire up the "gui_input()" signal as we have done with other signals before. The "gui_input()" signal passes an **InputEvent** that we can use to see what inputs are happening within our **Texture Button**. We are interested in a mouse click, so that is what we will target, and we will quit the application if the button is clicked.

Now return to the "StarWindow" scene and select the root node. Instantiate a child scene and chose our "window_close.tscn" scene. Move the control to a feasible location – in the right arm if you are matching the example code – and the save the scene. Now run the project and when the close button is clicked the window should close.

```
public partial class window_close : TextureButton {
    private void _on_gui_input(InputEvent @event) {
        if (@event is InputEventMouseButton mouseEvent) {
            if (mouseEvent.ButtonIndex == MouseButton.Left) {
                GetTree().Quit();
            }
        }
    }
}
```

Setting up the "WindowMove" scene is largely the same. The only different is we will need to keep up with whether or not the window is being dragged, and the position from which it is being dragged. Go ahead and set up the "gui_input()" signal and handler as done with the "WindowClose" scene. The code for dragging the window is listed later in the article for reference.

After the event is connected and the code is in place, instantiate the "window_move.tscn" scene as a child of the "StarWindow" node. Place the move button somewhere on the star (the example uses the left arm) and save the scene.

Now run the scene and the star shaped window should be able to be dragged around by clicking and holding the left mouse button on the move button. Clicking the close button should close the window.

Conclusion

This article has shown that irregular shaped buttons and windows are possible, and easily possible of using a game engine to develop a UI is acceptable. The Godot game engine can provide much more functionality that we have covered here – things like animations for controls, etc. But we have touched on the introductory points that should get you up and

running with simple user interfaces using Godot and C#.

The entire code for this article is in the repository listed at the end of this article. The source files used to create the images are included in that repository as well.

Thank you!

```
public partial class window_move : TextureButton {
    private bool _isDragging;
    private Vector2 _dragPoint;

    private void _on_gui_input(InputEvent @event) {
        if (@event is InputEventMouseButton mouseEvent) {
            if (mouseEvent.ButtonIndex == MouseButton.Left) {
                _isDragging = !_isDragging;
                _dragPoint = GetGlobalMousePosition();
            }
        }

        if (@event is InputEventMouseMotion && _isDragging) {
            var position = DisplayServer.MouseGetPosition();
            GetWindow().Position = (Vector2I)(position - _dragPoint);
        }
    }
}
```

Author: Matt Runion
Contact: mrunion@gmail.com

Endnotes

[1] The version of Godot used for this article is the 4.2.2 stable .Net version

[2] The code for this article can be found at the following GitHub repository: https://github.com/mrunion/usinggodotforui

[3] Using Signals in Godot https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html