Installation

This Game Engine/FrameWork was written in C#, with no GUI whatsoever to aid in modification or visually representing your work. Hence, It is required to use an IDE of some kind to write your code.

The IDE

1. This Engine was developed on the Rider IDE, and so this page will you Rider as its source of reference.
2. However, Visual Studio is also a powerfull IDE for C# and C++, and as long as you can understand the Docs, and do some basic googling, you should also be able to get the Engine Running.
3. Make sure you have Dotnet 7.0 or above installed.
4. Open your IDE, and made a dedicated Project to start making your game.

The Dependencies

5. Go to Release and download the latest release for the DLL required.
6. In the Project, Open up the dedicated Nugget tab and install *RayLib_cs*(6.0.0).
    - Incase you cannot use the inbuilt solution, you can try the command line here
7. In your dedicated IDE, Put the RayGame.dll in your local directory, and setup a reference to it.
    - In Rider, right - click the project, click Add, Add reference and browse to add the dll.
8. And there you go! You can now start Making your game!

Hello World!

Note:

Now that we finally have our project setup, lets begin writing some code!. Now, we would need 2 scripts to start with. A Entry script, and a Component to start everything else. Create a new folder if you want, organising files will be really helpful in the long run.

The Entry script

- When using this engine, we need some entry point to tell the Engine to start, thats what this script is for.
- So lets make a script called Entry.cs
- When using the engine, make sure to include:

```
using RayGame;
```

- This is the Engine code that we have imported to start making something.
- After that, lets create a very standard C# script:

```
using RayGame;
using [YOUR_NAMESPACE];;

public class Entry
{
    public static void Main()
    {

        Console.WriteLine("Hello World");
        //Note, the file does not HAVE to called Entry.
    }
}
```

- Lovely, we now have a script that we can use as an entry, from here we can initiate the Engine to start doing what we want it to do.
- Via the Engine Class, we Run an INIT function to attach a Component of choice into a prebuilt GameObject that we then use to manipulate the Engine to do what we want.
    - Although that may seem complicated, dont worry, it will be explained.

```
Engine.INIT<T>();
```

- As you can see, the INIT function requires some class to replace its Template field. That class is going to be our Custom Component.
- In the next section, we will talk about Components, and finally get something to show!

Logic

Think of Components as scripts that you attach to a GameObject to add logic to what it does. There are no restrictions about how many components a GameObject has, or how many GameObjects have the same component.

However, always keep in mind, that a GameObject can only have 1 type of component at a time, so write your logic accordingly. It is not possible to add the same component to a GameObject multiple times.

The Component Script

- Lets start talking about components.

1. There can be only 1 of them on a Gameobject
2. They must implement the "IGameComponent" Interface
3. They Must have a Container variable
4. It must have a Start() and Update() method.

```
using RayGame;
namespace [YOUR_NAMESPACE];

public class Main : IGameComponent
{
    public GameObject Container { get; set; }

    public void Start()
    {

    }

    public void Update()
    {

    }
}
```

- I have created a Main class to be my component.
- This is what an Empty Component script would look like.
- Notice how it Implements IGameComponent
- If your IDE supports it, then just by typing the name of the variable or method, A suggestion will show up to autocomplete the entire sentence.
- The Container variable is the reference you will use to reffer to your GameObject that component to attached to.

- With that said, lets add some simple code:

```
using RayGame;
namespace [YOUR_NAMESPACE];

public class Main : IGameComponent
{
    public GameObject Container { get; set; }

    public void Start()
    {
        Console.WriteLine("Hello Component!");
    }

    public void Update()
    {

    }
}
```

- And with that, we finally have our Component Script done. Lets put this all together!

Running the Engine

Now we have both an Entry Script, and a Component to use, lets finally have something we can see.

- In our Entry script, lets now use our "Main" Component in the INIT function:

```
using RayGame;
using [YOUR_NAMESPACE];;

public class Entry
{
    public static void Main()
    {

        Console.WriteLine("Hello World");
        Engine.INIT<Main>();
    }
}
```

- And with this, your window has opened up and your cosole should be showing:

> Hello World
>
> Hello Component!

A GameObject

- Now that we have all this setup, we can completely forget about our Entry Script.
- We not only consider our Main Class as the point where the Engine Starts.
- Keeping that in Mind, let me walk you through making a simple square, so that you understand how to write logic.

Creating a GameObject

- Creating a GameObject is pretty simple.
- A lot of your required functions can be found in the Engine Class. So we use that to create a GameObject.

```
var square = Engine.CreateGameObject("My Square");
```

- Here we use the Engine add GameObject to the scene. Most functions that involve GameObjects in the scene will be done through the Engine Class. We dont absolutely need to assign the GameObject to a variable, but if you want to, you can
- After that, we can now set the position of the Object.

```
square.Transform.Position = new Vector2(400, 240);
```

- The GameObject has a transform, which we manipulate the Position property to assign a new position.
- In Raylib, the top left is (0,0), hence, (400,240) would be the middle of the window.
- Despite assigning a position though, you still cannot see anything when you run the project. Thats because tou have not assigned a Renderer to the GameObject.
- Now, GameObjects have a LOT of methods and some properties you can mess around with, but we shall not be covering all of them right now.

Renderer

- So, the answer seems simple right? Just add a renderer to the Object, and you can see it. But no, there a just a few more steps before you get there.
- Lets start with adding a Renderer.

```
square.AddRenderer<MeshRenderer>();
```

- And that how we add a Mesh Renderer to the GameObject.

```
GAMEOBJECT.AddRenderer<T>();
GAMEOBJECT.AddComponent<T>();
```

- These are the methods that can add Components and Renderers to your GameObject, allowing you to create more complex activity in your scene.
- Now that we have a Renderer, we need to tell it what to render.

Meshes

- A Mesh is a class that stores vertices of a shape.
- It contains some methods that let you then manipulate the shape it contains.
- All the points of the Mesh are completely local, so you dont need to break your head trying to figure out where you should place each point. (i mean, you still do, but not that much).

```
Mesh sqr = new Mesh(new[] {(10f, 10f), (-10f, 10f), (-10f, -10f), (-10f, -10f)});
```

- I have constructed a Mesh, that takes in an Array of either tuple(float, float), or an array of Vector2.
- The Choice of types are up to you.

Putting it all Together

Finally, now that we have eveything setup, lets take a look at the final code:

```
using System.Numerics;
using RayGame;
namespace [YOUR_NAMESPACE];

public class Main : IGameComponent
{
    public GameObject Container { get; set; }

    public void Start()
    {
        Console.WriteLine("Hello Component!");

        var square =Engine.CreateGameObject("My Square");
        square.Transform.Position = new Vector2(400, 240);

        Mesh sqr = new Mesh(new[] {(10f, 10f), (-10f, 10f), (-10f, -10f), (10f, -10f)});

        MeshRenderer Renderer = square.AddRenderer<MeshRenderer>();
        Renderer.SetMesh(sqr);
```

```
    }

    public void Update()
    {

    }
}
```

- And Voilà! Your code now generates a square.

Adding some Features

- Now that we have a square, it seems a bit too small, so lets make the square bigger
- Lets also make the square rotate a little per frame.

```
square.Transform.Scale = 4.5f;
square.Transform.Rotate(2);
```

- I have moved the variable as a global one, to be used in other functions.
- I have called rotate in update, and changed the scale in start.

```
using System.Numerics;
using RayGame;
namespace [YOUR_NAMESPACE];

public class Main : IGameComponent
{
    public GameObject Container { get; set; }
    private GameObject square;

    public void Start()
    {
        Console.WriteLine("Hello Component!");

        square =Engine.CreateGameObject("My Square");
        square.Transform.Position = new Vector2(400, 240);

        Mesh sqr = new Mesh(new[] { (10f, 10f), (-10f, 10f), (-10f, -10f), (10f, -10f) });

        MeshRenderer Renderer = square.AddRenderer<MeshRenderer>();
        Renderer.SetMesh(sqr);

        square.Transform.Scale = 4.5f;
```

```
    }

    public void Update()
    {
        square.Transform.Rotate(2);
    }
}
```

Optimisations and Efficiency

- Along with what i have shown you, there are other features like collision, that you need to find and learn by yourself.
- You can do that by checking out the API section next to the Docs that you are reading.
- In said API, all Classes and methods have a </> next to it, clicking it will take you to the source code for said Class/ Method.
- The Plethora of methods available should let you do a variety of things, and i am open to adding more if requested.
- That being said, I have made a very basic version of Flappy bird, which you can check out by doing

```
Engine.INIT<RayGame.Demo.Manager>();
```

New Code

- After refining the code made earlier, this is what it looks like now:

```
using System.Numerics;
using RayGame;
namespace TestEngine;

public class Main : IGameComponent
{
    public GameObject Container { get; set; }
    private GameObject square;

    public void Start()
    {
        square = Engine.CreateGameObject("My Square",
            new Transform(
                new Vector2(400, 240),
                0,
                4.5f));

        square.AddRenderer<MeshRenderer>().SetMesh(
            new Mesh(new[]
            {
                (10f, 10f),
                (-10f, 10f),
                (-10f, -10f),
                (10f, -10f)
            }));
```

```
        // Code has been expanded with \n so that its more readable.

    }

    public void Update()
    {
        square.Transform.Rotate(2);
    }
}
```

- With the Help of different constructor, chaining, and declaring a variable inside a constructor, we have made the same program with less clutter.
- *technically, I could chain the AddRenderer< >() to the CreateGameObject(), but I decided not to.*
- And that marks the END of this tutorial, from here, you can read the docs and try making your own stuff!