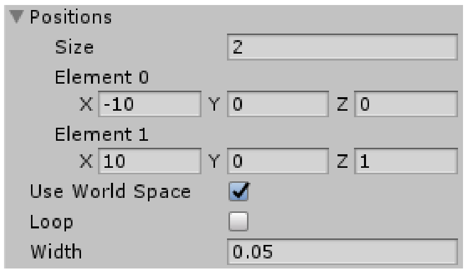
1 Creating a 2D unity project

This week we will be drawing a curve to the screen and approximating the area under the curve using rectangles. Start by creating a 2D Unity project.

2 Creating the x and y axes

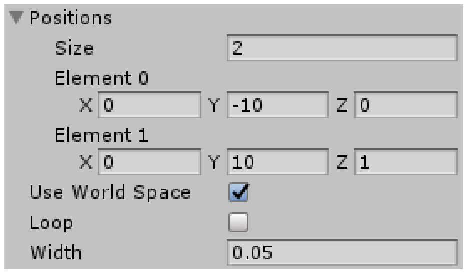
In order to make it easier to visually determine if our output is correct we are going to create two lines that will serve as the axes of the graph that we are creating.

Start by creating two empty GameObjects. Name one of these GameObjects "X Axis" and the other "Y Axis" and add the Line Renderer component to each of them. Set the width of each of the Line Renderers to about 0.05 and the material to Sprites-Default. Set the position of Element 0 of the X Axis to (-10, 0, 0) and Element 1 of the X Axis to (10, 0, 0).



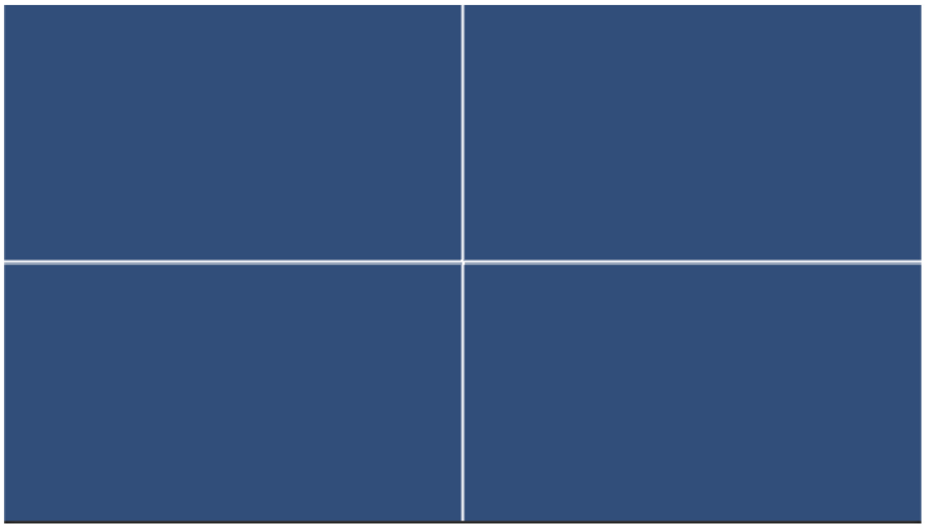
```
▼ Positions
    Size              2
    Element 0
      X -10        Y 0        Z 0
    Element 1
      X 10         Y 0        Z 1
  Use World Space   ✓
  Loop              ☐
  Width             0.05
```

Set the position of Element 0 of the Y Axis to (0, -10, 0) and Element 1 of the Y Axis to (0, 10, 0).



```
▼ Positions
    Size              2
    Element 0
      X 0          Y -10      Z 0
    Element 1
      X 0          Y 10       Z 1
  Use World Space   ✓
  Loop              ☐
  Width             0.05
```

If this has been done correctly you should now have the following output:



3 Drawing the Curve

Now that we have axes we can draw our curve. We will be plotting the curve $y = -(\frac{x}{3})^2 + 4$. This will provide us with a parabola that appears to be squished in the Y direction and has a maximum that is 4 Unity units above the x-axis.

Start by creating an empty GameObject and naming it "curve". Add a LineRenderer Component to this GameObject. You can leave the properties of this LineRenderer as the default as we will be setting them in code.

Next create a new c# script and name it "Curve". We will use this script to draw all of the output to the screen and to approximate the area. We should first create a method to find the Y value for a given X value on our curve. This should appear as follows for our curve:

```
        // Find the Y value for a given X
        private float FindY (float x) {
                return -Mathf.Pow(x / 3, 2) + 4;
        }
```

We should next create some public variables to set the properties of our curve. These should be the number of divisions that our line is broken up into, the minimum and maximum x values that we want to plot and the width of our line.

```
        // Curve variables
        public int numberDivisions = 31;
        public float domainMin = -7.0f;
        public float domainMax = 7.0f;
        public float width = 0.1f;
```

We can now start writing the code to draw our curve. Start by creating a new method and calling it "DrawCurve".

```
        private void DrawCurve () {
```

The first thing that we need to do within this method is setup the LineRenderer for drawing our curve. We will first get the LineRenderer component and then set the number of positions and the width of the line.

```
                // Setup the linerenderer
                LineRenderer lr = GetComponent<LineRenderer>();
                lr.positionCount = numberDivisions;
                lr.widthMultiplier = width;
```

Now that our LineRenderer is setup correctly we can set each of the positions of the line to the correct position to draw our curve. We will start by finding the increments along the x-axis and then we can use a for loop to find the corresponding y position for each of these x positions.

```
                // Set the positions of the linerender
                float incrementSize = (Mathf.Abs(domainMin) +
                    Mathf.Abs(domainMax)) / (numberDivisions - 1);
                for(int i = 0; i < numberDivisions; i++) {
                        Vector3 position = new Vector3();

                        position.x = ((float) i * incrementSize) +
                            domainMin;
                        position.y = FindY(position.x);
                        position.z = 0.0f;

                        lr.SetPosition(i, position);
                }
```

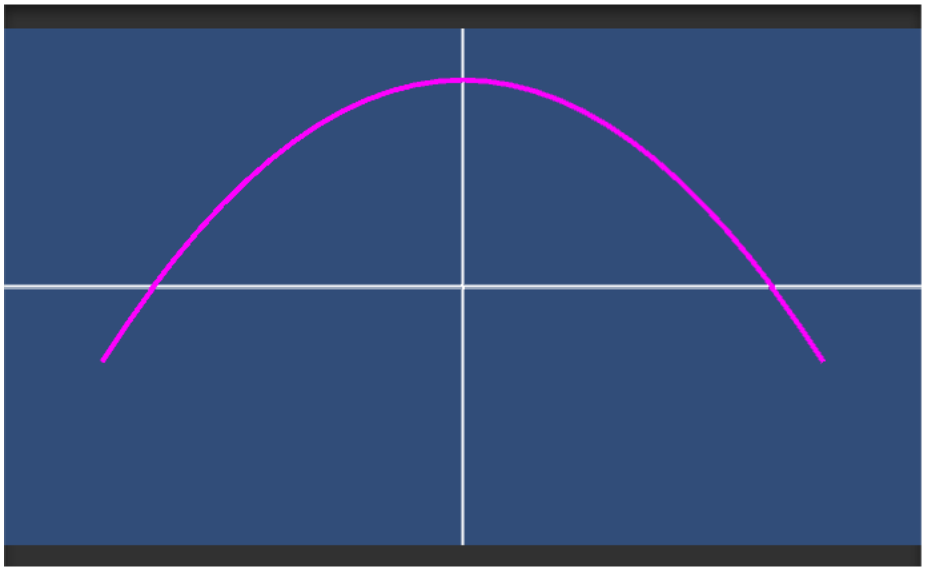
The completed DrawCurve method should now appear as follows:

```
        private void DrawCurve () {
                // Setup the linerenderer
                LineRenderer lr = GetComponent<LineRenderer>();
                lr.positionCount = numberDivisions;
                lr.widthMultiplier = width;

                // Set the positions of the linerender
                float incrementSize = (Mathf.Abs(domainMin) +
                    Mathf.Abs(domainMax)) / (numberDivisions - 1);
                for(int i = 0; i < numberDivisions; i++) {
                        Vector3 position = new Vector3();

                        position.x = ((float) i * incrementSize) +
                            domainMin;
                        position.y = FindY(position.x);
                        position.z = 0.0f;

                        lr.SetPosition(i, position);
                }
        }
```

We can now call the DrawCurve method from out Start method.

```
        // Use this for initialization
        void Start () {
                // Draw the curve
                DrawCurve();
```

When run the output should now appear as follows:



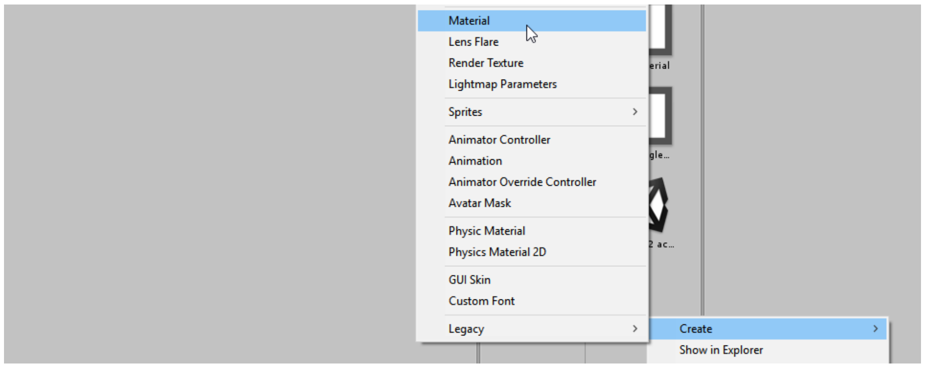
4 Filling in the area

Now that we have drawn the curve we can fill in the area the we are approximating. Start by creating an empty GameObject and calling it "AreaFill". Back in the "Curve" script we now need to create some variables that will be used to set the properties of the fill area and the rectangles that we will be drawing.

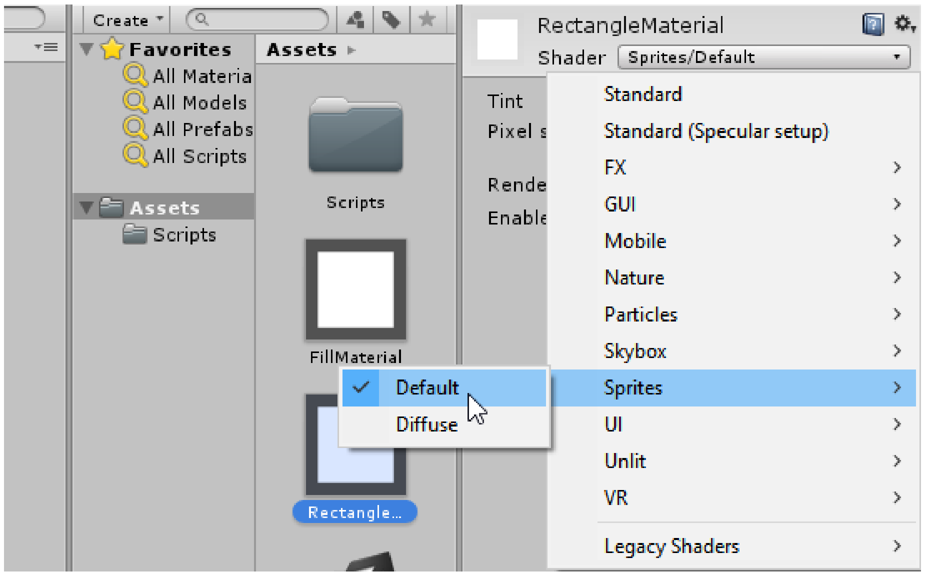
```
        // Area variables
        public Material fillMaterial;
        public Material rectangleMaterial;
        public int numberRectangles = 5;
        public int numberFillVertices = 7;
        public float x1 = -2.0f;
        public float x2 = 3.0f;
        public Color areaFillColor = new Color(0.8f, 0.3f, 0.3f,
            1.0f);
        public Color rectangleColor = new Color(0.2f, 0.6f, 0.3f,
            0.5f);

        public GameObject areaFill;
        public GameObject rectangles;

        private float areaUnderCurve = 0.0f;
```

Back in Unity we now need to set the Area Fill variable to be our newly created "AreaFill" GameObject.

We now need to create two new materials that we will use to set the colours of the Fill Area and the rectangles that we are using to approximate the area. One of these should be named "FillMaterial" and the other should be named "RectangleMaterial" A new material can be created by right clicking in the assets folder and then clicking Create > Material.

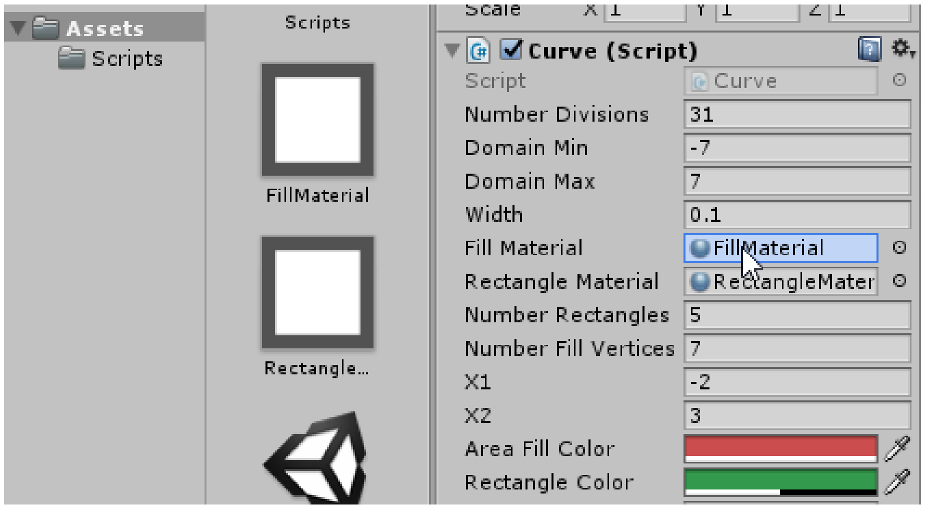


The shader for each of these materials should then be set to "Sprites/Default".



RectangleMaterial
Shader Sprites/Default

The Fill Material and Rectangle Material variables in the Curve script should then be set to FillMaterial and RectangleMaterial respectively.



Curve (Script)

Script	Curve
Number Divisions	31
Domain Min	-7
Domain Max	7
Width	0.1
Fill Material	FillMaterial
Rectangle Material	RectangleMater
Number Rectangles	5
Number Fill Vertices	7
X1	-2
X2	3
Area Fill Color	🟥
Rectangle Color	🟩

We can now start working on the code to fill the area. As with previous weeks we will be using the MeshFilter and MeshRenderer components to add a mesh to a GameObject. Start by creating a new method called "FillArea".

```
        private void FillArea () {
```

Within this method we will then add a MeshFilter and MeshRenderer to the areaFill GameObject, get the mesh from the MeshFilter and set the material of the MeshRenderer.

```
                // Add a MeshFilter and MeshRenderer to the Empty
                    GameObject
        areaFill.AddComponent<MeshFilter>();
        areaFill.AddComponent<MeshRenderer>();

        // Get the Mesh from the MeshFilter
        Mesh mesh = areaFill.GetComponent<MeshFilter>().mesh;

        // Set the material to the material we have selected
        areaFill.GetComponent<MeshRenderer>().material = fillMaterial;

        // Clear all vertex and index data from the mesh
        mesh.Clear();
```

We will now fill as much of the area as possible with a single rectangle. Start by finding the highest y value that does now exceed the boundary of the curve. This will be equal to the lower y value of the upper and lower limits.

```
        // Find the higher y value
        float lowerY = 0;

        float y1 = FindY(x1);
        float y2 = FindY(x2);

        if(y1 < y2) {
                lowerY = y1;
        } else {
                lowerY = y2;
        }
```

As our implementation will only support finding the area below a curve we should now throw an exception if the lower y value is less than 0.

```
        // Throw an error if the implementation would fail
        if (lowerY < 0) {
                throw new System.Exception("This implementation does
                    not support finding the area above a curve");
        }
```

Now we can create three lists that will hold the vertices, colors, and indices that we will need to fill in the area that we are approximating. We can also initialize them with the values needed to create our rectangle, as the vertex locations for this can be easily calculated at this point.

```
        // Create a rectangle that fills as much space as possible
            between x1 and x2
        List<Vector3> vertices = new List<Vector3>() {
            new Vector3(x1, 0, 0),
            new Vector3(x1, lowerY, 0),
            new Vector3(x2, 0, 0),
            new Vector3(x2, lowerY, 0)
        };

        // Set the colour of the rectangle
        List<Color> colors = new List<Color>() {
            areaFillColor,
            areaFillColor,
            areaFillColor,
            areaFillColor
        };

        // Set vertex indicies
        List<int> triangles = new List<int>() {0, 1, 2, 1, 2, 3};
```

Now we can fill the remaining area using a for loop (Feel free to copy and paste this code).

```
        // Find the remaining area
        float incrementSize = (Mathf.Abs(x2) + Mathf.Abs(x1)) /
            (numberFillVertices - 1);
        int currentIndex = vertices.Count;

        // Add the first triangle
        triangles.Add(1);
        triangles.Add(4);
        triangles.Add(5);
        for(int i = 0; i < numberFillVertices; i++) {
                // Find the vertices of the rectangle
                Vector3 v1 = new Vector3();

                        v1.x = ((float) i * incrementSize) + x1;
                        v1.y = FindY(v1.x);
                        v1.z = 0.0f;

                        Vector3 v2 = new Vector3();

                        v2.x = v2.x;
                        v2.y = lowerY;
                        v2.z = 0.0f;
                        vertices.Add(v1);
                        vertices.Add(v2);

                        // Set the colour at the vertices
                        colors.Add(areaFillColor);
                        colors.Add(areaFillColor);

                        // Find the indices of the rectangle
                        triangles.Add(currentIndex);
                        triangles.Add(currentIndex + 1);
                        triangles.Add(currentIndex + 2);
                        triangles.Add(currentIndex + 1);
                        triangles.Add(currentIndex + 2);
                        triangles.Add(currentIndex + 3);

                        // Increment the vertex counter
                        currentIndex += 2;

        }

        // Add the last two vertices
        Vector3 lv1 = new Vector3();

                lv1.x = x2;
                lv1.y = FindY(lv1.x);
                lv1.z = 0.0f;

                Vector3 lv2 = new Vector3();

                lv2.x = x2;
                lv2.y = lowerY;
                lv2.z = 0.0f;

                vertices.Add(lv1);
                vertices.Add(lv2);

                // Add the last two colors
                colors.Add(areaFillColor);
                colors.Add(areaFillColor);
```

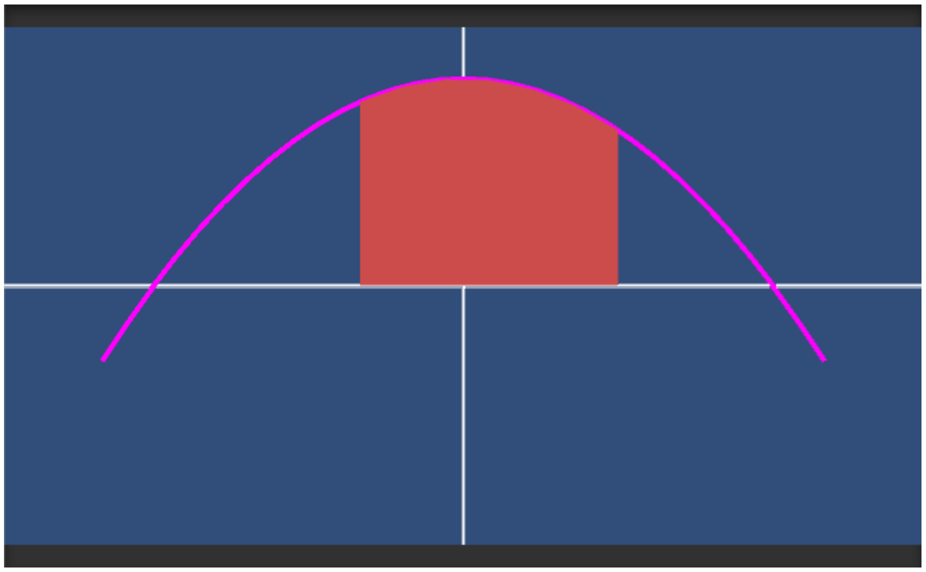
We are now able to set the the vertices, colors and indices of our mesh to the values that we found.

```
            // Set the vertices, colors and triangles of the mesh to
                be equal to our lists
            mesh.vertices = vertices.ToArray();
            mesh.colors = colors.ToArray();
            mesh.triangles = triangles.ToArray();
```

We can now call the FillArea method that we just created from our Start method.

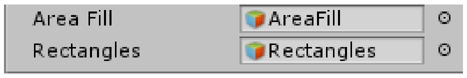
```
        // Use this for initialization
        void Start () {
                // Draw the curve
                DrawCurve();

                // Fill in the area we are finding
                FillArea();
```

If everything was done correctly the output should now appear as follows:



5 Approximating the Area

In this step we are going to both draw a graphical representation of the approximated area and perform that calculations to find that approximation. Start by creating a new empty GameObject and calling it "Rectangles". You should then set the Rectangles variable of the Curve script to the Rectangles GameObject.



Next create a method called "DrawRectangles" in the Curve script.

```
        public void DrawRectangles () {
```

We can then add the MeshFilter and MeshRenderer components to the rectangle GameObject and get the mesh from the MeshFilter and set the material of the MeshRenderer.

```
                // Add a MeshFilter and MeshRenderer to the Empty
                    GameObject
        rectangles.AddComponent<MeshFilter>();
        rectangles.AddComponent<MeshRenderer>();

        // Get the Mesh from the MeshFilter
        Mesh mesh = rectangles.GetComponent<MeshFilter>().mesh;

        // Set the material to the material we have selected
        rectangles.GetComponent<MeshRenderer>().material =
            rectangleMaterial;

        // Clear all vertex and index data from the mesh
        mesh.Clear();
```

We can now create three lists to store the vertices, colors and indices of the mesh while we are finding them.

```
        // Create empty lists to store the rectangle data
        List<Vector3> vertices = new List<Vector3>();
        List<Color> colors = new List<Color>();
        List<int> triangles = new List<int>();
```

We can now use a for loop to find all of the vertices, colors and indices needed to create the rectangles (Feel free to copy and paste this code).

```
        // Find all of the rectangles
        float incrementSize = (Mathf.Abs(x2) + Mathf.Abs(x1)) /
            (numberRectangles);
        for (int i = 0; i < numberRectangles; i++) {
                // Find the vertices of the rectangle
                Vector3 v1 = new Vector3();

                        v1.x = ((float) i * incrementSize) + x1;
                        v1.y = 0.0f;
                        v1.z = -0.1f;

                        float y1 = FindY(v1.x);
                        float y2 = FindY(v1.x + incrementSize);

                        float smallerY = y1 < y2 ? y1 : y2;

                        Vector3 v2 = new Vector3();

                        v2.x = v1.x;
                        v2.y = smallerY;
                        v2.z = -0.1f;

                        Vector3 v3 = new Vector3();

                        v3.x = v1.x + incrementSize;
                        v3.y = 0.0f;
                        v3.z = -0.1f;

                        Vector3 v4 = new Vector3();

                        v4.x = v1.x + incrementSize;
                        v4.y = smallerY;
                        v4.z = -0.1f;

                        vertices.Add(v1);
                        vertices.Add(v2);
                        vertices.Add(v3);
                        vertices.Add(v4);

                        // Set the colour at the vertices
                        colors.Add(rectangleColor);
                        colors.Add(rectangleColor);
                        colors.Add(rectangleColor);
                        colors.Add(rectangleColor);

                        // Set the indices of the triangles
                        triangles.Add(i * 4);
```

```
                                triangles.Add(i * 4 + 1);
                                triangles.Add(i * 4 + 2);
                                triangles.Add(i * 4 + 1);
                                triangles.Add(i * 4 + 2);
                                triangles.Add(i * 4 + 3);

                                // Add the area of the rectangle to the total
                                areaUnderCurve += incrementSize * smallerY;

        }
```

Note that the actual approximation of the area occurs on this line

```
                                // Add the area of the rectangle to the total
                                areaUnderCurve += incrementSize * smallerY;
```

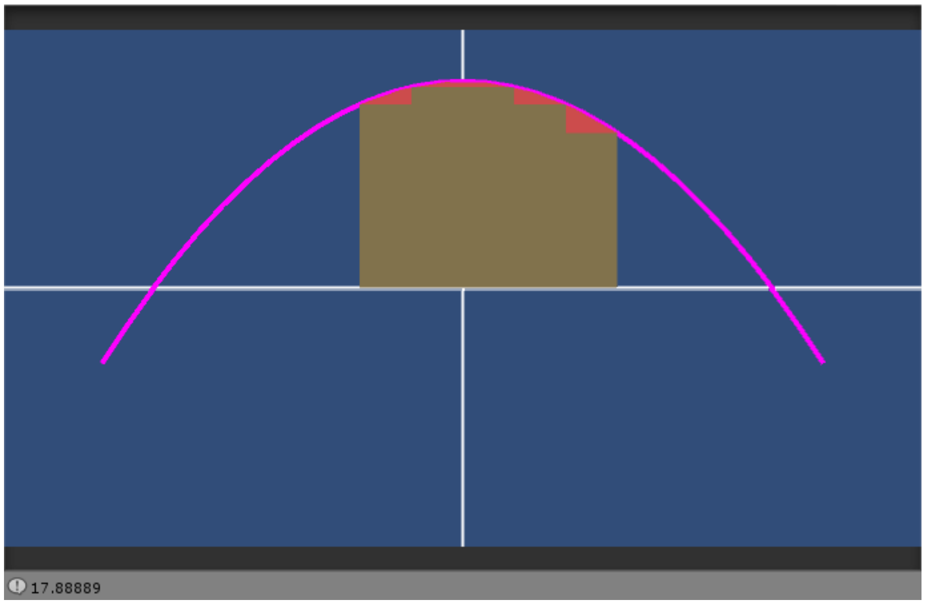
Now we can set the vertices, colors and indices of the triangle to be equal to the ones we calculated.

```
        // Draw the rectangles
        mesh.vertices = vertices.ToArray();
        mesh.colors = colors.ToArray();
        mesh.triangles = triangles.ToArray();
```

The DrawRectangles method should now be complete and we can call it from our Start method. We should also output the approximated area at this point. The completed Start method should now appear as follows:

```
        // Use this for initialization
        void Start () {
                // Draw the curve
                DrawCurve();

                // Fill in the area we are finding
                FillArea();

                // Draw the rectangles on the curve
                DrawRectangles();

                // Show the area under the curve
                Debug.Log(areaUnderCurve);
        }
```

Now everything has been completed and the output should appear as follows:



```
17.88889
```

If you wish to do so you can check your answer using WolframAlpha by typing in the query "integrate -(x/3)^2 + 4, x = -2 to 3". Your answer should approach this value as more rectangles are added.

6 Challenge

Now that you have your solution working with a parabola, get it to work with the cubic:

$$y = (x + 3)(x + 1)(x - 2)$$

This will require modifying the "FindY" method to draw the new curve, and the "FillArea" and "DrawRectangles" methods to allow finding the area above and below the x-axis.