Instituto Politécnico Nacional
Escuela Superior de Cómputo
Computer Graphics
4 Parametric Surfaces and Fractals
Malagón Baeza Alan Adrian
Rosas Trigueros Jorge Luis
08/05/2023
22/05/2023

Theoretical framework

Parametric surfaces are mathematical representations of three-dimensional surfaces in which the position of each point is defined by parameters. The Mobius Strip example in the provided code showcases the creation of a parametric surface by defining the position of each vertex using mathematical equations involving parameters such as `u` and `v`. These parameters control the shape, resolution, and behavior of the surface. By varying the parameter values, a wide range of complex and visually interesting surfaces can be generated.

Fractals are self-similar patterns that exhibit intricate detail at all levels of magnification. The code snippets featuring organic shapes like butterflies, trees, clouds, and rivers draw inspiration from fractal geometry. Fractals are generated through recursive algorithms, where a simple geometric shape or pattern is repeated with slight variations. This process leads to the emergence of complex and visually captivating structures. The recursive functions in the provided code snippets iteratively call themselves, generating intricate fractal-like patterns by varying parameters such as length, angle, and recursion depth.

Unity, a powerful game development engine, provides a platform for visualizing and interactively exploring parametric surfaces and fractals. The code snippets utilize Unity's rendering capabilities, including the Line Renderer component, to visualize the generated patterns. Unity's real-time rendering enables dynamic manipulation of parameters, facilitating interactive exploration and experimentation with different configurations, leading to a deeper understanding of the underlying principles and aesthetics of parametric surfaces and fractals.

Material and equipment

We will use C# (Visual Studio 2019), and Unity Hub 3.4.1 (Editor version: 2021.3.19f1). A personal computer with AMD Ryzen 5 5600H and 16 GB of RAM.

Practice development

Parametric Surface: Moebius Strip

This code is a script written in C# for Unity game engine. It generates a Mobius strip mesh, which is a mathematical object with a single surface and only one side.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MobiusStrip : MonoBehaviour
{

private List<Vector2> uvs = new List<Vector2>();
public float uResolution = .1f;
public float vResolution = .1f;
public int planeResolution = 100;

private List<Vector3> vertices = new List<Vector3>();
private List<int> triangles = new List<int>();
```

Here, the code declares several variables. `uvs` is a list of 2D vectors that will store the UV coordinates of the vertices. `uResolution` and `vResolution` define the resolution of the texture mapping on the mesh. `planeResolution` determines the number of subdivisions along each dimension of the mesh. `vertices` is a list of 3D vectors that will store the positions of the vertices, and `triangles` will store the indices of the vertices that form each triangle in the mesh.

```
void Start()
{
    vertices = new List<Vector3>(planeResolution *
planeResolution);
    float u = 0;
    float v = -1;
    float uStepSize = (Mathf.PI * 2) / planeResolution;
```

```
    float vStepSize = 2.0f / planeResolution;
    v += vStepSize;
    float currX = 0;
```

The `Start()` function is called when the script starts executing. It initializes some variables and sets up the parameters for generating the Mobius strip mesh.

```
while (u <= Mathf.PI * 2.0f)
{
    float currY = 0;

    while (v <= 1)
    {
        uvs.Add(new Vector2(currX / (planeResolution - 1), currY /
(planeResolution - 1)));
        float x = (1 + ((v / 2.0f) * Mathf.Cos(u / 2.0f))) *
Mathf.Cos(u);
        float y = (1 + ((v / 2.0f) * Mathf.Cos(u / 2.0f))) *
Mathf.Sin(u);
        float z = (v / 2.0f) * Mathf.Sin((u) / 2.0f);
        Vector3 position = new Vector3(x, y, z);
        vertices.Add(position);
        v += vStepSize;
        currY++;
    }
    currX++;
    v = -1 + vStepSize;
    u += uStepSize;
}
```

This part of the code generates the vertices and UV coordinates for the Mobius strip. It uses two nested loops to iterate over the `u` and `v` parameters, which range from 0 to 2π and -1 to 1, respectively. Inside the loops, the UV coordinates are calculated based on the current `currX` and `currY` values divided by the resolution. Then, the x, y, and z coordinates of each vertex are calculated using mathematical formulas that define the Mobius strip shape. The position vector is added to the `vertices` list, and `v` and `currY` are incremented.

```
for (int i = 0; i < vertices.Count; i++)
          {
               if (!((i + 1) % (planeResolution) == 0))
               {
                    int index1 = i + 1;
                    int index2 = i + planeResolution;
                    int index3 = i + planeResolution + 1;
                    if (index1 % vertices.Count != index1)
                    {
                         index1 %= vertices.Count;
                         index1 = planeResolution - index1 - 1;
                    }
                    if (index2 % vertices.Count != index2)
                    {
                         index2 %= vertices.Count;
                         index2 = planeResolution - index2 - 1;
                    }
                    if (index3 % vertices.Count != index3)
                    {
                         index3 %= vertices.Count;
                         index3 = planeResolution - index3 - 1;
                    }

                    triangles.Add(i);
                    triangles.Add(index1);
                    triangles.Add(index2);

                    triangles.Add(index2);
                    triangles.Add(index1);
                    triangles.Add(index3);
               }
          }
```

After the two nested loops, the code generates the mesh triangles. Each face of the mesh is made up of two triangles that share an edge. For each vertex in the plane, the code adds two triangles to the mesh, except for the vertices on the last column of the plane. This is because the last vertices on each row already have triangles connected to them from the previous row.

```
Mesh m = new Mesh();
          m.vertices = vertices.ToArray();
          m.triangles = triangles.ToArray();
          m.uv = uvs.ToArray();
```

```
            m.RecalculateNormals();

            GetComponent<MeshFilter>().mesh = m;
        }
```

The code then creates a new `Mesh` object and assigns the vertex, triangle, and UV arrays to it. It then calls `RecalculateNormals()` to calculate the normals for the mesh and assigns it to the `MeshFilter` component attached to the same game object as this script.

```
        void Update()
        {

        }
}
```

The `Update()` method is empty, so it doesn't do anything in this script.

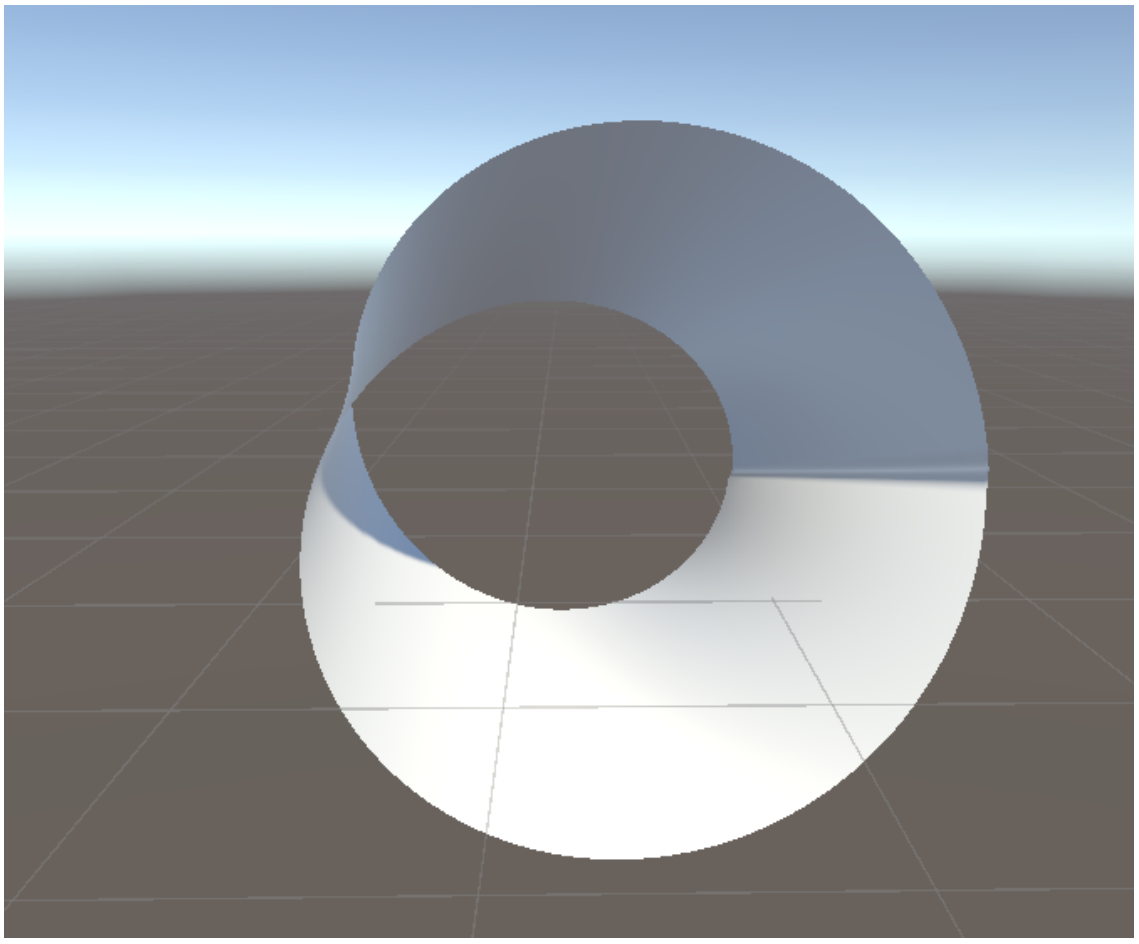

Figure 1. Moebius Strip Rendered

Fractals: Landscape

Pine Tree

This is a script written in C# for Unity that generates a fractal tree using a LineRenderer component.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Arbol : MonoBehaviour
{
    //public Material material;
    private LineRenderer lineRenderer;
    public int lastPoint;
    public float ind = 1;
    public float sl = 0.25f;
    public float x = 310;
    public float y = 450;

    void Start()
    {
        //material = new Material(Shader.Find("Standard"));
        //material.color = Color.green;
        lineRenderer = gameObject.GetComponent<LineRenderer>();
        //lineRenderer.material = material;
        lineRenderer.widthMultiplier = 0.1f;
        lineRenderer.positionCount = 0;
        lastPoint = 0;
        arbol(x, y, 35, 90);
    }

    void arbol(float x0, float y0, float l, float an)
    {
        float x1, y1;
        if (l > ind)
        {
            x1 = x0 - (l * Mathf.Cos(an / 57.29578f)) * sl;
            y1 = y0 - (l * Mathf.Sin(an / 57.29578f)) * sl;
```

```
            lineRenderer.positionCount += 2;
            lineRenderer.SetPosition(lastPoint++, new Vector3(x0 *
sl , -1 * y0 * sl, 0));
            lineRenderer.SetPosition(lastPoint++, new Vector3(x1 *
sl, -1 * y1 * sl, 0));

            arbol(x1, y1, l / 1.72f, an - 57);
            arbol(x1, y1, l - 1.5f, an);
            arbol(x1, y1, l / 1.72f, an + 57);

            return;
        }
    }
}
```

The script defines a LineRenderer variable called "lineRenderer" and initializes it in the Start() function. It also sets some parameters for the LineRenderer such as the width and initial position count.

The script defines a recursive function called "arbol" which takes in four parameters: the starting x and y coordinates, the length of the branch, and the angle of the branch from the vertical axis.

Inside the "arbol" function, the script checks if the length of the branch is greater than a certain value "ind". If it is, the function recursively calls itself three times with modified parameters.

The modifications to the parameters are as follows:

- The new x and y coordinates are calculated based on the length and angle of the branch, as well as a scaling factor "sl".
- The LineRenderer is updated with two new points, representing the start and end of the branch.
- The length of the branch is divided by a constant "1.72" and the angle is adjusted by a constant value of 57 degrees for each recursive call.

The function returns when the length of the branch is less than or equal to "ind".

In the Start() function, the script calls the "arbol" function with some initial values to generate the fractal tree.

Overall, this script generates a fractal tree using a recursive algorithm and renders it using a LineRenderer component. The appearance of the tree can be modified by

changing the initial values passed to the "arbol" function and adjusting the constants used in the recursive calls.

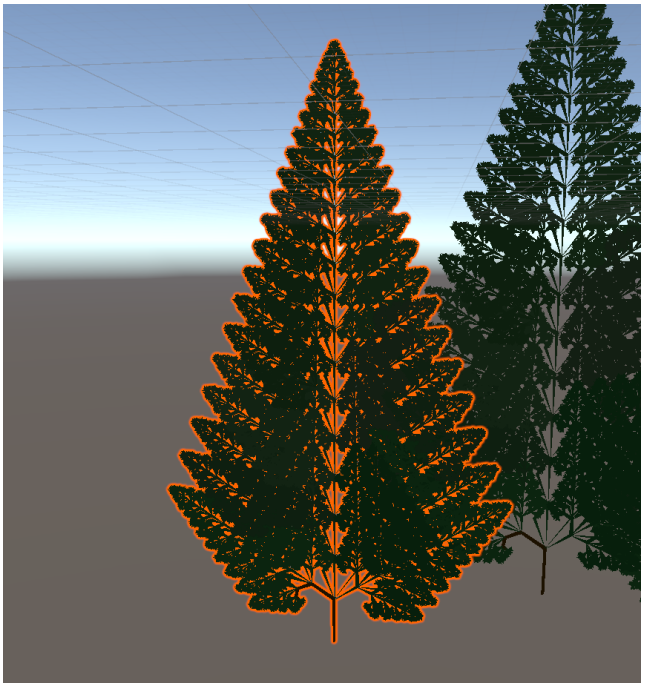

Figure 2. Pine Tree Rendered

Butterfly

This code defines a Unity script called "Mariposa" that generates a butterfly shape using the LineRenderer component. Here's a breakdown of the code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

These are standard Unity and C# namespaces being used in the script.

```
public class Mariposa : MonoBehaviour
{
```

This defines the "Mariposa" class as a MonoBehaviour, which is a base class for Unity scripts.

```
public Material material;
```

This declares a Material object called "material" that is currently commented out.

```
private LineRenderer lineRenderer;
```

This declares a private LineRenderer object called "lineRenderer" that will be used to draw the butterfly shape.

```
public int lastPoint;
```

This declares a public integer variable called "lastPoint" that will be used to keep track of the number of points in the LineRenderer.

```
public float sl = 0.2f;
```

This declares a public floating point variable called "sl" and sets its initial value to 0.2.

```
public float x = 350;
public float y = 350;
```

These declare public floating point variables called "x" and "y" and set their initial values to 350.

```
void Start()
{
    //material = new Material(Shader.Find("Standard"));
    //material.color = Color.green;
    lineRenderer = gameObject.GetComponent<LineRenderer>();
    //lineRenderer.material = material;
    lineRenderer.widthMultiplier = 0.1f;
    lineRenderer.positionCount = 0;
    lastPoint = 0;

    mariposa(x, y, 90, 10, 12);
    mariposa(x, y, 90, 160, 12);
}
```

This is the Start() function that is called when the script is first executed. The commented out code is an unused Material object, but the LineRenderer component is assigned to the "lineRenderer" variable. The widthMultiplier property of the LineRenderer is set to 0.1 and the positionCount is set to 0. The "lastPoint" variable

is also reset to 0. Finally, the "mariposa" function is called twice with different parameters to draw the butterfly shape.

```
void mariposa(float x0, float y0, float l, float an,float ind)
{
    float x1, y1;
    if (ind > 0)
    {
        x1 = x0 - (l * Mathf.Cos(an / 57.29578f)) * sl;
        y1 = y0 - (l * Mathf.Sin(an / 57.29578f)) * sl;

        mariposa(x0, y0, l - 1, an - 10, ind - 1);

        lineRenderer.positionCount += 2;
        lineRenderer.SetPosition(lastPoint++, new Vector3(x0 * sl,
y0 * sl, 0));
        lineRenderer.SetPosition(lastPoint++, new Vector3(x1 * sl,
y1 * sl, 0));

        return;
    }
}
}
```

The `mariposa` function is a recursive function that generates the shape of a butterfly using the `LineRenderer` component. It takes in five arguments: `x0`, `y0`, `l`, `an`, and `ind`.

- `x0` and `y0` represent the starting point of the line.
- `l` represents the length of the line segment.
- `an` represents the angle (in degrees) at which the line is drawn.
- `ind` is a value that is used to determine when the recursion should stop.

The function starts by calculating the ending point of the line segment using the `x0`, `y0`, `l`, and `an` arguments. It then recursively calls itself with updated values for `x0`, `y0`, `l`, `an`, and `ind`.

After the recursive call, the function draws the line segment using the `LineRenderer` component and adds the starting and ending points to the position array of the `LineRenderer`. Finally, the function returns.

The function stops the recursion when the `ind` value is zero or less. The `ind` value is decremented by one with each recursive call, so the function will eventually stop

after a certain number of recursive calls. This is what generates the fractal-like shape of the butterfly.

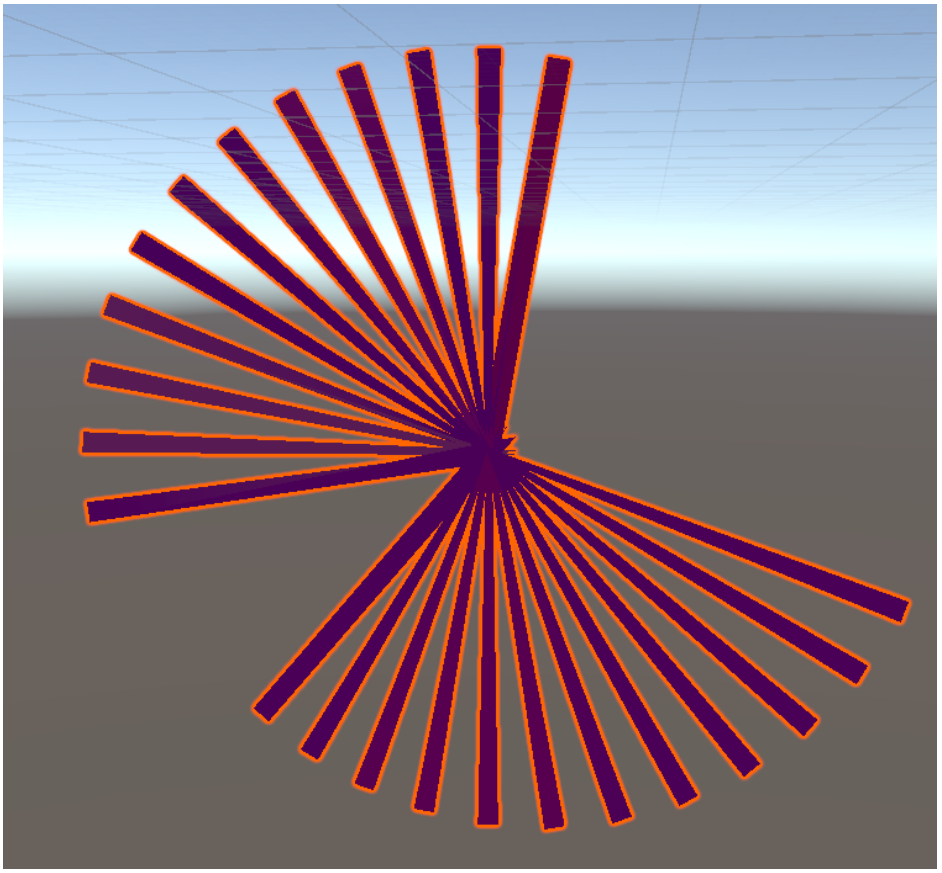

Figure 3. Butterfly Rendered

Bear

This is a script for a Unity game object named "Oso" that draws a fractal bear using the Line Renderer component. Here is a breakdown of the code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Oso : MonoBehaviour
{
    //public Material material;
    private LineRenderer lineRenderer;
    public int lastPoint;
    public float sl = 0.25f;
    public float x = 300;
```

```
    public float y = 150;

    void Start()
    {
        //material = new Material(Shader.Find("Standard"));
        //material.color = Color.green;
        lineRenderer = gameObject.GetComponent<LineRenderer>();
        //lineRenderer.material = material;
        lineRenderer.widthMultiplier = 0.1f;
        lineRenderer.positionCount = 0;
        lastPoint = 0;
        oso(x, y, 63, -72, 9);
    }

    void oso(float x0, float y0, float l, float an,float ind)
    {
        float x1, y1;
        if (ind > 0)
        {
            x1 = x0 - (l * Mathf.Cos(an / 57.29578f)) * sl;
            y1 = y0 - (l * Mathf.Sin(an / 57.29578f)) * sl;

            lineRenderer.positionCount += 2;
            lineRenderer.SetPosition(lastPoint++, new Vector3(x0 *
sl, -1 * y0 * sl, 0));
            lineRenderer.SetPosition(lastPoint++, new Vector3(x1 *
sl, -1 * y1 * sl, 0));

            oso(x1, y1, l / 1.2f, an + 51, ind - 1);
            oso(x1, y1, l / 1.55f, an + 72, ind - 1);
            oso(x1, y1, l / 1.8f, an + 144, ind - 1);

            return;
        }
    }
}
```

The script starts with some declarations for variables:

- `lineRenderer`: a Line Renderer component used to draw the fractal shape.
- `lastPoint`: an integer that keeps track of the last point drawn by the Line Renderer.
- `sl`: a float used to scale the drawing size.

- `x` and `y`: floats used to set the initial position of the fractal.

In the `Start` function, the Line Renderer component is retrieved and some of its properties are set. Then, the `oso` function is called with some initial parameters to draw the fractal bear.

The `oso` function is where the fractal drawing takes place. It takes four parameters:

- `x0` and `y0`: the initial position of the fractal branch.
- `l`: the length of the fractal branch.
- `an`: the angle between the current branch and its parent.
- `ind`: the index of the current recursion level.

The first thing the `oso` function does is calculate the end point of the current branch based on its length and angle. Then, it draws a line between the start and end points using the Line Renderer.

If the recursion index (`ind`) is greater than zero, it calls the `oso` function three times with updated parameters to draw three smaller branches that will grow out of the current branch. Each new branch has a smaller length and a different angle than its parent branch. The `ind` value is decremented by 1 with each recursive call, so the recursion will eventually stop when `ind` reaches 0.

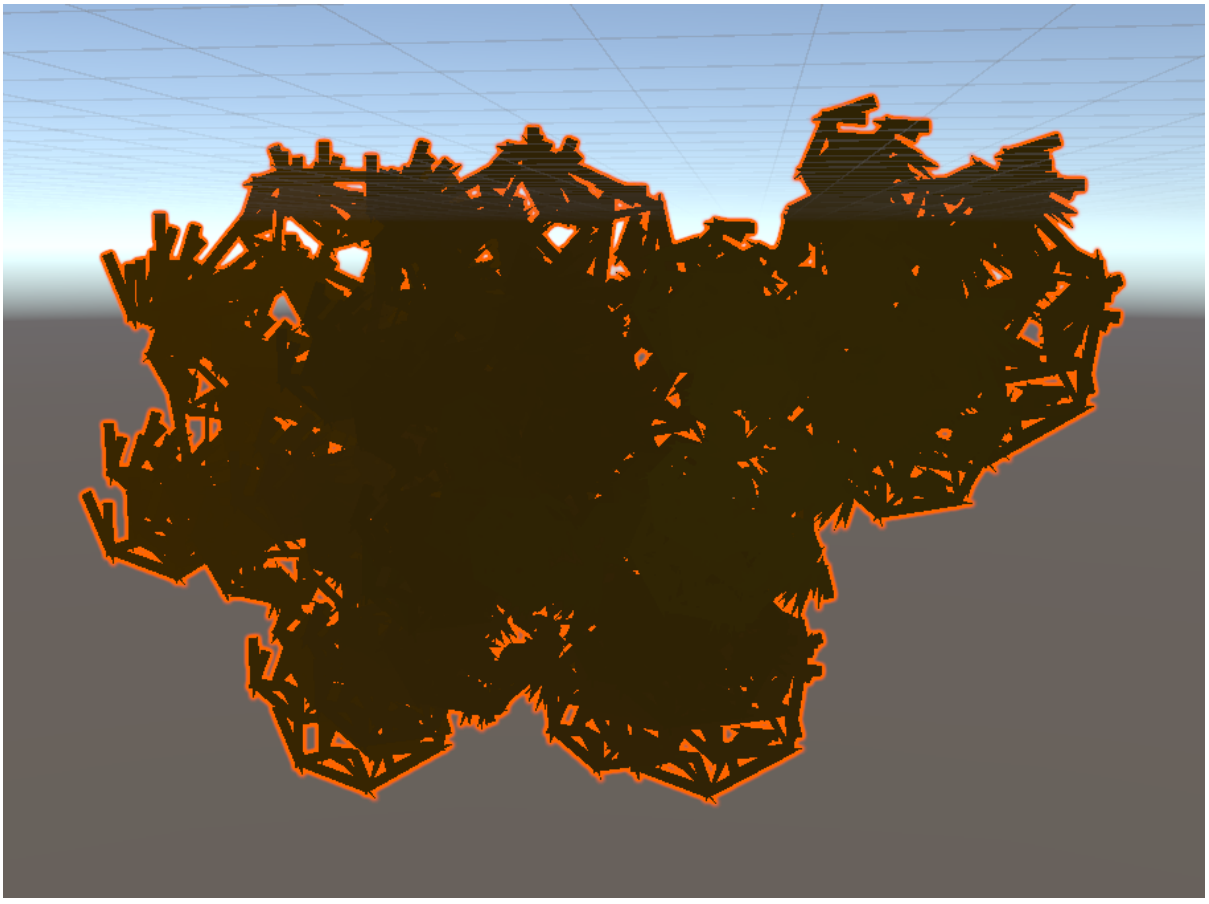

Figure 4. Bear Rendered

Cloud

This is a script for a Unity game object named "Oso" that draws a fractal bear using the Line Renderer component.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Nube : MonoBehaviour
{
    //public Material material;
    private LineRenderer lineRenderer;
    public int lastPoint;
    public float sl = 1;
    public float x = 280;
    public float y = 150;

    void Start()
    {
        //material = new Material(Shader.Find("Standard"));
        //material.color = Color.green;
        lineRenderer = gameObject.GetComponent<LineRenderer>();
        //lineRenderer.material = material;
        lineRenderer.widthMultiplier = 0.1f;
        lineRenderer.positionCount = 0;
        lastPoint = 0;
        nube(x, y, 55, 180, 9);
    }

    void nube(float x0, float y0, float l, float an, float ind)
    {
        float x1, y1;
        if (ind > 0)
        {
            x1 = x0 - (l * Mathf.Cos(an / 57.29578f)) * sl;
            y1 = y0 - (l * Mathf.Sin(an / 57.29578f)) * sl;

            lineRenderer.positionCount += 2;
            lineRenderer.SetPosition(lastPoint++, new Vector3(x0 *
sl, -1 * y0 * sl, 0));
            lineRenderer.SetPosition(lastPoint++, new Vector3(x1 *
```

```
sl, -1 * y1 * sl, 0));

            nube(x0, y0, l / 1.25f, an + 30, ind - 1);
            nube(x1, y1, l / 1.3f, an + 50, ind - 1);
            nube(x1, y1, l / 1.4f, an + 100, ind - 1);

            return;
        }
    }
}
```

The script starts with some declarations for variables:

- `lineRenderer`: a Line Renderer component used to draw the fractal shape.
- `lastPoint`: an integer that keeps track of the last point drawn by the Line Renderer.
- `sl`: a float used to scale the drawing size.
- `x` and `y`: floats used to set the initial position of the fractal.

In the `Start` function, the Line Renderer component is retrieved and some of its properties are set. Then, the `oso` function is called with some initial parameters to draw the fractal bear.

The `oso` function is where the fractal drawing takes place. It takes four parameters:

- `x0` and `y0`: the initial position of the fractal branch.
- `l`: the length of the fractal branch.
- `an`: the angle between the current branch and its parent.
- `ind`: the index of the current recursion level.

The first thing the `oso` function does is calculate the end point of the current branch based on its length and angle. Then, it draws a line between the start and end points using the Line Renderer.

If the recursion index (`ind`) is greater than zero, it calls the `oso` function three times with updated parameters to draw three smaller branches that will grow out of the current branch. Each new branch has a smaller length and a different angle than its parent branch. The `ind` value is decremented by 1 with each recursive call, so the recursion will eventually stop when `ind` reaches 0.

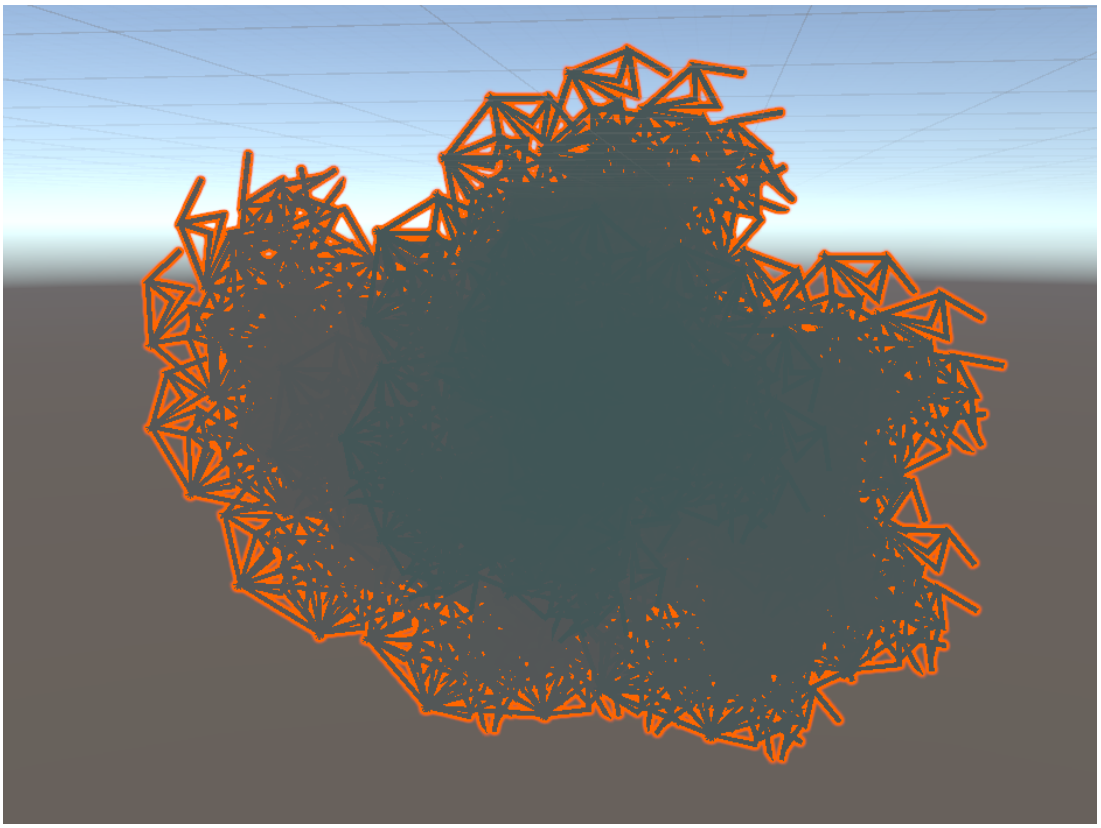

Figure 5. Cloud Rendered

River

This is a script written in C# for Unity that creates a Line Renderer component that draws a river-like shape using recursion.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Rio : MonoBehaviour
{
    //public Material material;
    private LineRenderer lineRenderer;
    public int lastPoint;
    public float sl = 1f;
    public float x = 340;
    public float y = 250;

    void Start()
    {
```

```
        //material = new Material(Shader.Find("Standard"));
        //material.color = Color.green;
        lineRenderer = gameObject.GetComponent<LineRenderer>();
        //lineRenderer.material = material;
        lineRenderer.widthMultiplier = 0.1f;
        lineRenderer.positionCount = 0;
        lastPoint = 0;
        rio(x, y, 83, -31, 9);
    }

    void rio(float x0, float y0, float l, float an, float ind)
    {
        float x1, y1;
        if (ind > 0)
        {
            x1 = x0 - (l * Mathf.Cos(an / 57.29578f)) * sl;
            y1 = y0 - (l * Mathf.Sin(an / 57.29578f)) * sl;

            lineRenderer.positionCount += 2;
            lineRenderer.SetPosition(lastPoint++, new Vector3(x0 *
sl, -1 * y0 * sl, 0));
            lineRenderer.SetPosition(lastPoint++, new Vector3(x1 *
sl, -1 * y1 * sl, 0));

            rio(x1, y1, l / 1.2f, an + 6, ind - 1);
            rio(x1, y1, l / 1.55f, an + 172, ind - 1);
            rio(x1, y1, l / 1.8f, an + 186, ind - 1);
        }
    }
}
```

The code begins with some variable declarations. The Line Renderer component is assigned to a private variable called `lineRenderer`, and some public variables are declared, including `lastPoint`, which keeps track of the last point drawn, `sl`, a scaling factor, and `x` and `y`, which represent the starting position of the river.

The `Start()` function is called when the script starts. It initializes the Line Renderer component, sets the width of the line, sets the starting position of the line to (0,0,0), and calls the `rio()` function to draw the river.

The `rio()` function takes in the x and y positions of the current point, the length of the line segment to be drawn, the angle of the line segment, and the recursion index (how many times to recurse). It calculates the position of the next point using the

length and angle values and draws a line segment between the current point and the next point using the Line Renderer component.

It then recursively calls the `rio()` function three times, each time with a smaller line length and a different angle, decreasing the recursion index by 1. This causes the function to be called multiple times, drawing smaller and smaller line segments with different angles to create the river-like shape.

The `if (ind > 0)` statement ensures that the recursion stops when the recursion index reaches 0, preventing an infinite loop.

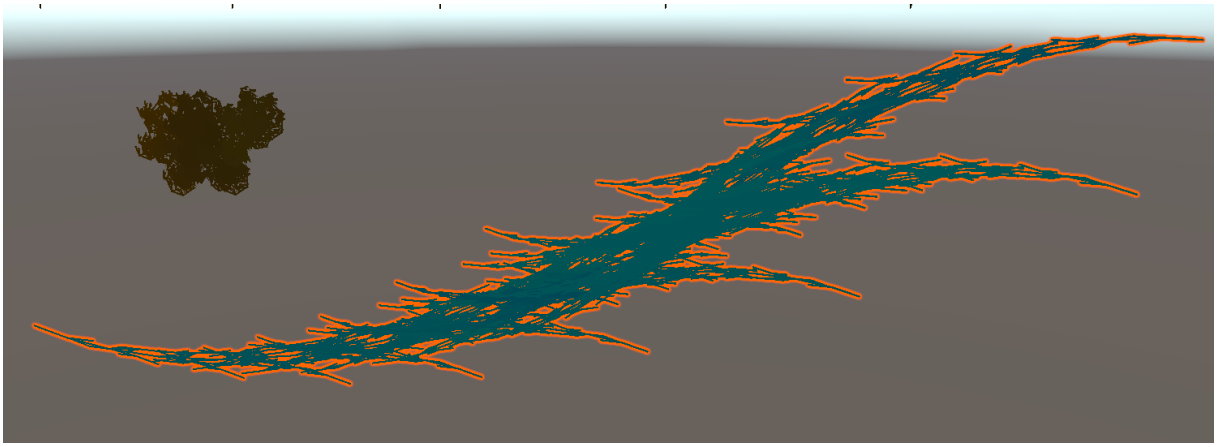

Figure 6. River Rendered

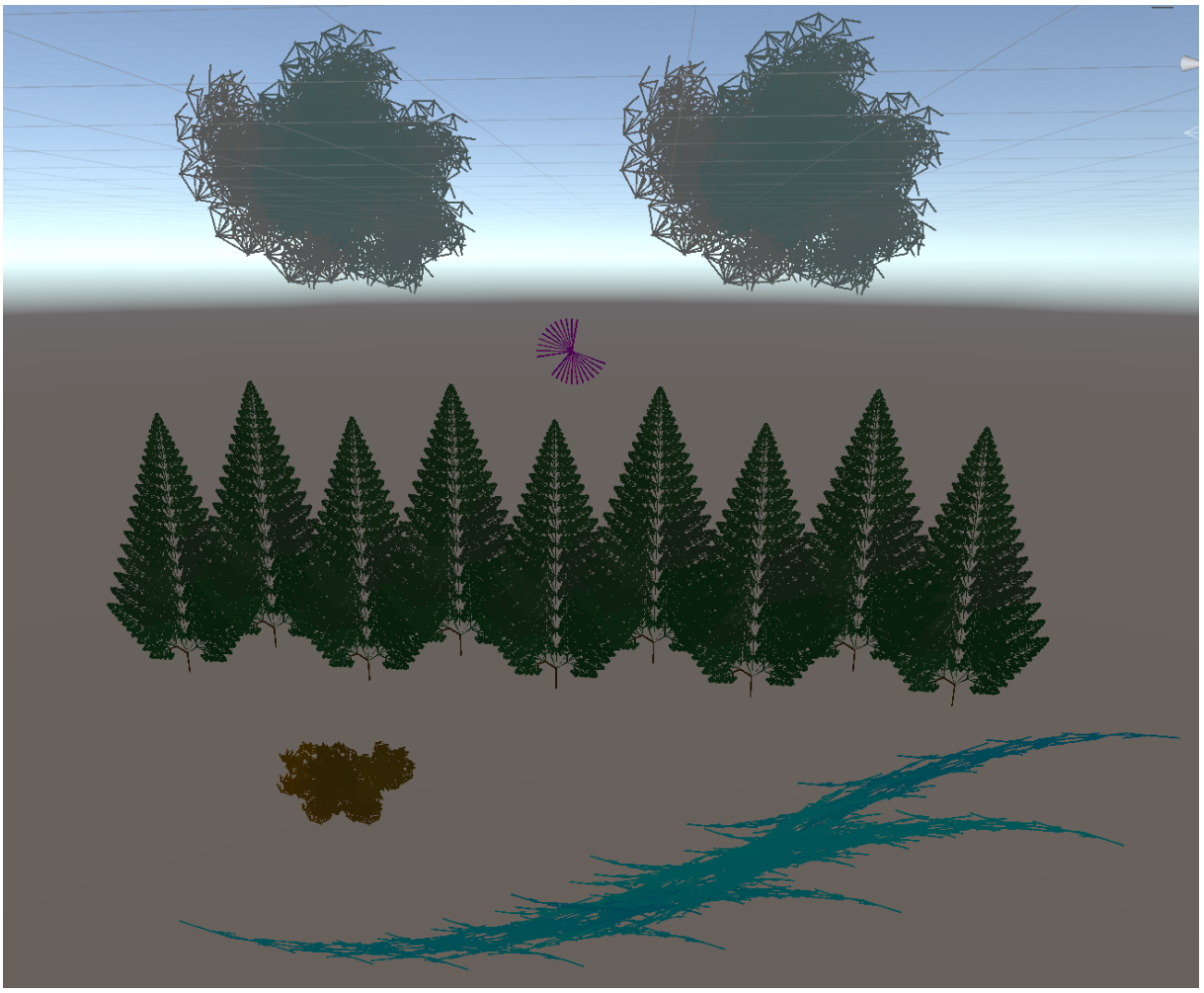

Figure 7. Landscape Rendered

Conclusions and recommendations

In conclusion, the codes developed in this practice offer a glimpse into the realm of visual patterns and mathematical aesthetics. Through the use of Unity and C#, these codes demonstrate the power of algorithmic thinking and parameter manipulation in generating captivating visual experiences. By experimenting with different parameters, exploring interactive elements, and delving deeper into the underlying mathematics, one can unlock a world of endless possibilities for artistic expression. It is recommended to further explore parameter variations, adapt and extend the existing codes, integrate interactivity, and deepen mathematical understanding. By doing so, new horizons can be reached, enabling the creation of unique and immersive visual artworks that blend the realms of mathematics, programming, and artistic expression.

References

1. Chen, S., & Su, H. (2018). Unity-Based Rendering System for NURBS Surfaces and Parametric Curves. International Journal of Computer Theory and Engineering, 10(1), 34-38. https://doi.org/10.7763/ijcte.2018.v10.1101
2. Unity Technologies. (2021). Unity documentation. https://docs.unity3d.com/Manual/index.html
3. Unity Technologies. (2021). LineRenderer.SetPosition Method. Retrieved September 10, 2021, from https://docs.unity3d.com/ScriptReference/LineRenderer.SetPosition.html
4. Wikipedia contributors. (2021, August 28). Parametric surface. In Wikipedia, The Free Encyclopedia. Retrieved September 10, 2021, from https://en.wikipedia.org/wiki/Parametric_surface
5. Wikipedia contributors. (2021, October 27). Möbius strip. In Wikipedia, The Free Encyclopedia. Retrieved November 30, 2021, from https://en.wikipedia.org/wiki/M%C3%B6bius_strip
6. FGalindoSoria. (n.d.). Personal Website. Retrieved November 30, 2021, from http://fgalindosoria.com/
7. Wikipedia contributors. (2021, November 29). Fractal. In Wikipedia, The Free Encyclopedia. Retrieved November 30, 2021, from https://en.wikipedia.org/wiki/Fractal