Instituto Politécnico Nacional
Escuela Superior de Cómputo
Computer Graphics
3 Parametric curves
Malagón Baeza Alan Adrian
Rosas Trigueros Jorge Luis
24/04/2023
08/05/2023

Theoretical framework

Parametric curves are mathematical functions that describe the motion of a point in space. In Unity, parametric curves can be used to create smooth and natural movements of objects in 3D space. By defining a set of equations that control the position, rotation, and scale of an object over time, developers can create complex and dynamic animations with precise control. Parametric curves in Unity can be used to create a wide range of effects, from simple movements to complex particle systems and procedural animations.

Moving 3D objects along parametric trajectories in Unity is a powerful technique for creating dynamic and engaging animations. By defining a set of equations that control the movement of an object over time, developers can create complex and natural motion patterns that are difficult to achieve with traditional animation techniques. This approach can be used to animate everything from character movements to vehicle physics, and it is particularly useful for creating realistic and immersive experiences in games and simulations.

Hermite curves are a type of parametric curve that are particularly useful for creating smooth and continuous animations in Unity. By defining a set of control points and tangent vectors, developers can use Hermite curves to create complex and natural motion patterns that follow a specific path. This technique can be used to create a wide range of effects, from simple animations to complex simulations. One application of Hermite curves in Unity is to create volutes, which are spiral or scroll-shaped objects commonly found in architecture and design. By defining a Hermite curve that follows the shape of a volute, developers can animate a 3D model along this curve to create a realistic and visually appealing effect.

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 Curves

Butterfly Curve

```
using UnityEngine;
using System.Collections;

public class ButterflyCurve : MonoBehaviour
{
    public Color c1 = Color.yellow;
    public Color c2 = Color.red;
    public int lengthOfLineRenderer = 1000;
    public float tIni = -20.0f;
    public float tFin = 20.0f;


    void Start()
    {
        LineRenderer lineRenderer =
gameObject.AddComponent<LineRenderer>();
        lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));
        lineRenderer.widthMultiplier = 0.2f;
        lineRenderer.positionCount = lengthOfLineRenderer;

        // A simple 2 color gradient with a fixed alpha of 1.0f.
        float alpha = 1.0f;
        Gradient gradient = new Gradient();
        gradient.SetKeys(
            new GradientColorKey[] { new GradientColorKey(c1,
0.0f), new GradientColorKey(c2, 1.0f) },
            new GradientAlphaKey[] { new GradientAlphaKey(alpha,
0.0f), new GradientAlphaKey(alpha, 1.0f) }
        );
```

```
        lineRenderer.colorGradient = gradient;
    }

    (float x, float y) butterfly(float t)
    {
        float xCoord = Mathf.Sin(t) * (Mathf.Exp(Mathf.Cos(t)) - 2
* Mathf.Cos(4 * t) - Mathf.Pow(Mathf.Sin(t / 12), 5));
        float yCoord = Mathf.Cos(t) * (Mathf.Exp(Mathf.Cos(t)) - 2
* Mathf.Cos(4 * t) - Mathf.Pow(Mathf.Sin(t / 12), 5));
        return (xCoord, yCoord);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);

    void Update()
    {
        LineRenderer lineRenderer = GetComponent<LineRenderer>();

        float delta = (tFin - tIni) / (lengthOfLineRenderer-1);
        float t = tIni, x, y;

        for (int i = 0; i < lengthOfLineRenderer; i++)
        {
            (x, y) = butterfly(t);
            newPosition.x = x;
            newPosition.y = y;
            lineRenderer.SetPosition(i, newPosition);
            t += delta;
        }
    }
}
```

This code generates a LineRenderer component that draws a butterfly curve using the butterfly function, which takes in a value t and returns the corresponding x and y coordinates of the curve. The butterfly function is defined within the script.

In the Start method, a new LineRenderer component is added to the game object, and its material, width multiplier, and number of positions to be rendered are set. A gradient is also created and set as the color gradient of the LineRenderer.

In the Update method, the LineRenderer component is retrieved and updated by iterating through the length of positions to be rendered and calling the butterfly function for each value of t, setting the returned coordinates as the current position of the LineRenderer. The t value is incremented by a calculated delta value, which is determined by the difference between tFin and tIni, divided by the number of positions to be rendered minus one.

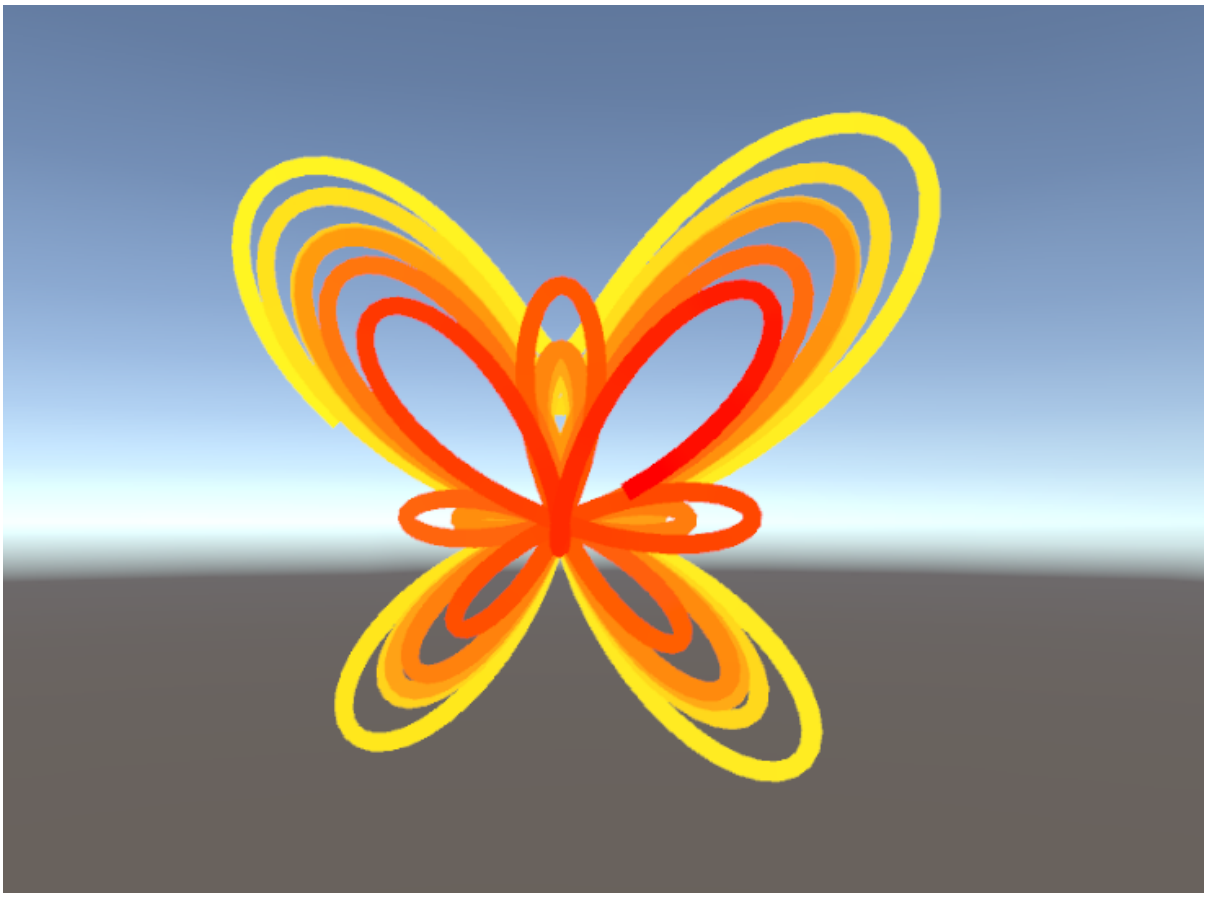

Figure 1. Butterfly Curve Displayed

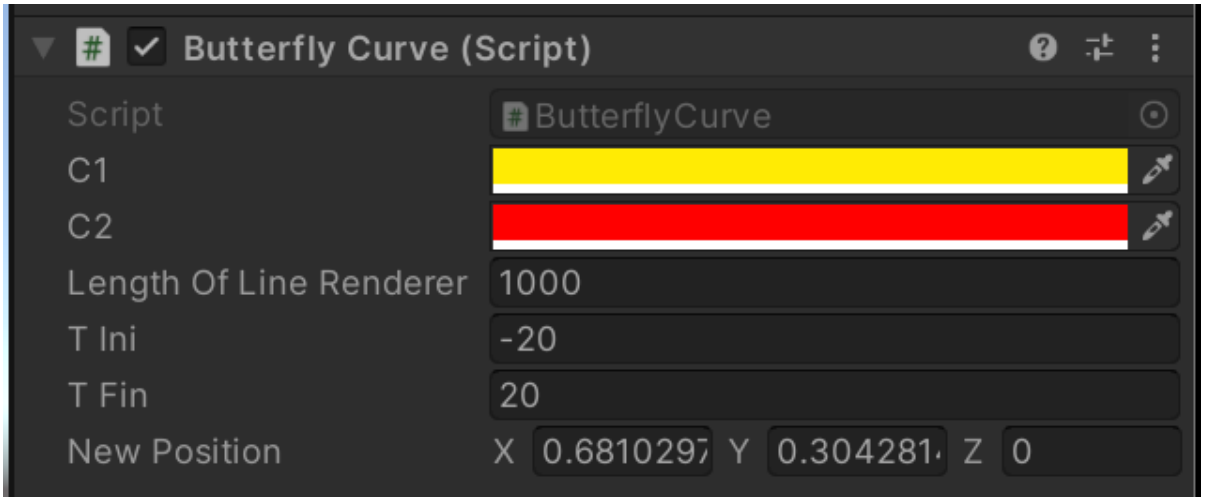

Figure 2. Butterfly Curve Script Parameters

Lissajous Curve

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

public class LissajousCurve : MonoBehaviour
{
    public Color c1 = Color.yellow;
    public Color c2 = Color.red;
    public int lengthOfLineRenderer = 1000;
    public float tIni = 0.0f;
    public float tFin = 2 * Mathf.PI;
    public float kx = 3.0f;
    public float ky = 2.0f;

    void Start()
    {
        LineRenderer lineRenderer =
gameObject.AddComponent<LineRenderer>();
        lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));
        lineRenderer.widthMultiplier = 0.2f;
        lineRenderer.positionCount = lengthOfLineRenderer;

        // A simple 2 color gradient with a fixed alpha of 1.0f.
        float alpha = 1.0f;
        Gradient gradient = new Gradient();
        gradient.SetKeys(
            new GradientColorKey[] { new GradientColorKey(c1,
0.0f), new GradientColorKey(c2, 1.0f) },
            new GradientAlphaKey[] { new GradientAlphaKey(alpha,
0.0f), new GradientAlphaKey(alpha, 1.0f) }
        );
        lineRenderer.colorGradient = gradient;
    }

    (float x, float y) lissajous(float t)
    {
        float xCoord = Mathf.Cos(kx * t);
        float yCoord = Mathf.Sin(ky * t);
        return (xCoord, yCoord);
```

```
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);

    void Update()
    {
        LineRenderer lineRenderer = GetComponent<LineRenderer>();

        float delta = (tFin - tIni) / (lengthOfLineRenderer - 1);
        float t = tIni, x, y;

        for (int i = 0; i < lengthOfLineRenderer; i++)
        {
            (x, y) = lissajous(t);
            newPosition.x = x;
            newPosition.y = y;
            lineRenderer.SetPosition(i, newPosition);
            t += delta;
        }
    }
}
```

This code defines a script for drawing a Lissajous curve using Unity's LineRenderer component. It begins by defining some public variables, including two colors for the curve, the number of points to draw, the start and end values of the parameter t, and the values of the parameters kx and ky.

In the Start() method, a LineRenderer component is added to the game object that this script is attached to. The material and widthMultiplier of the LineRenderer are set, and its positionCount is set to the number of points to draw. Finally, a gradient is defined for the colors of the LineRenderer.

The lissajous() method calculates the x and y coordinates of a point on the Lissajous curve given the parameter t and the values of kx and ky.

In the Update() method, the LineRenderer is obtained, and the values of delta and t are computed. A for loop then iterates through the desired number of points, calling lissajous() to get the x and y coordinates for each point, and setting the position of the LineRenderer at that point to (x,y).

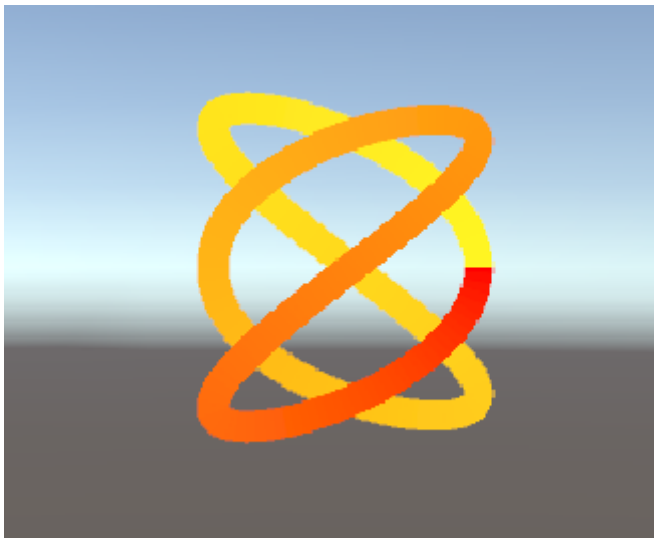

Figure 3. Lissajous Curve Displayed

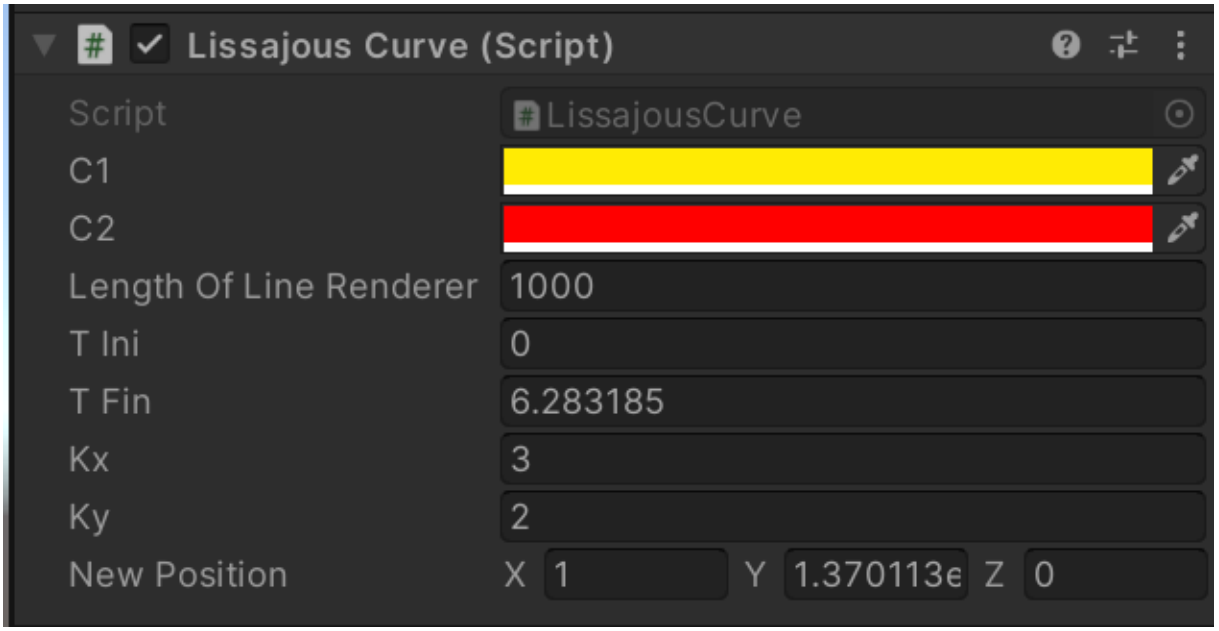

Figure 4. Lissajous Curve Script Parameters

Hypotrochoid

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

public class Hypotrochoid : MonoBehaviour
{
    public Color c1 = Color.yellow;
    public Color c2 = Color.red;
```

```
    public int lengthOfLineRenderer = 1000;
    public float tIni = 0.0f;
    public float tFin = 1000.0f;
    public float R = 8.0f;
    public float r = 3.0f;
    public float d = 2.00f;

    void Start()
    {
        LineRenderer lineRenderer =
gameObject.AddComponent<LineRenderer>();
        lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));
        lineRenderer.widthMultiplier = 0.2f;
        lineRenderer.positionCount = lengthOfLineRenderer;

        // A simple 2 color gradient with a fixed alpha of 1.0f.
        float alpha = 1.0f;
        Gradient gradient = new Gradient();
        gradient.SetKeys(
            new GradientColorKey[] { new GradientColorKey(c1,
0.0f), new GradientColorKey(c2, 1.0f) },
            new GradientAlphaKey[] { new GradientAlphaKey(alpha,
0.0f), new GradientAlphaKey(alpha, 1.0f) }
        );
        lineRenderer.colorGradient = gradient;
    }

    (float x, float y) hypotrochoid(float t)
    {
        float xCoord = (R - r) * Mathf.Cos(t) + d * Mathf.Cos((R -
r) / r * t);
        float yCoord = (R - r) * Mathf.Sin(t) - d * Mathf.Sin((R -
r) / r * t);
        return (xCoord, yCoord);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);

    void Update()
    {
        LineRenderer lineRenderer = GetComponent<LineRenderer>();
```

```
        float delta = (tFin - tIni) / (lengthOfLineRenderer - 1);
        float t = tIni, x, y;

        for (int i = 0; i < lengthOfLineRenderer; i++)
        {
            (x, y) = hypotrochoid(t);
            newPosition.x = x;
            newPosition.y = y;
            lineRenderer.SetPosition(i, newPosition);
            t += delta;
        }
    }
}
```

This is a script in C# for Unity that draws a Hypotrochoid shape using a Line Renderer component. The shape is defined by mathematical equations that depend on some parameters.

- The script starts by defining some public variables:

  - `c1` and `c2`: Colors used for the line renderer.
  - `lengthOfLineRenderer`: The number of points used to draw the shape.
  - `tIni` and `tFin`: The initial and final values of the parameter `t` used to generate the shape.
  - `R`, `r` and `d`: Parameters of the Hypotrochoid equations.

- In the `Start()` method, the line renderer component is added to the game object and some of its properties are set (material, width, position count, and gradient).

- The `hypotrochoid()` method computes the x and y coordinates of a point on the Hypotrochoid curve, given a value of the parameter `t` and the parameters `R`, `r` and `d`. The equations used are:

  - x = (R - r) * cos(t) + d * cos((R - r) / r * t)
  - y = (R - r) * sin(t) - d * sin((R - r) / r * t)

- The `Update()` method generates a set of points on the curve by iterating over values of `t` and calling the `hypotrochoid()` method to get the coordinates of each point. The position of the line renderer points is set using the `SetPosition()` method of the line renderer component.

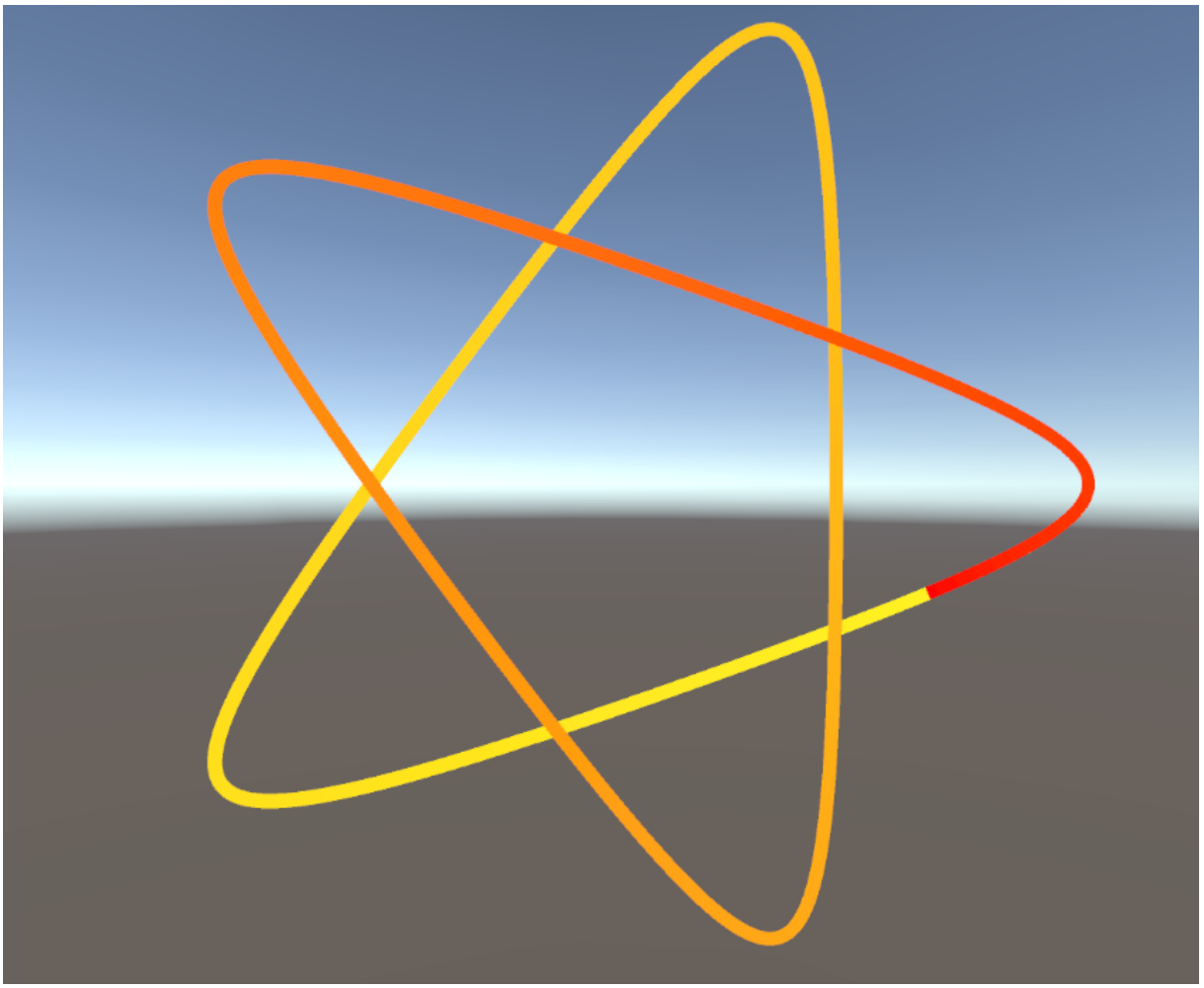

Figure 5. Hypotrochoid Displayed

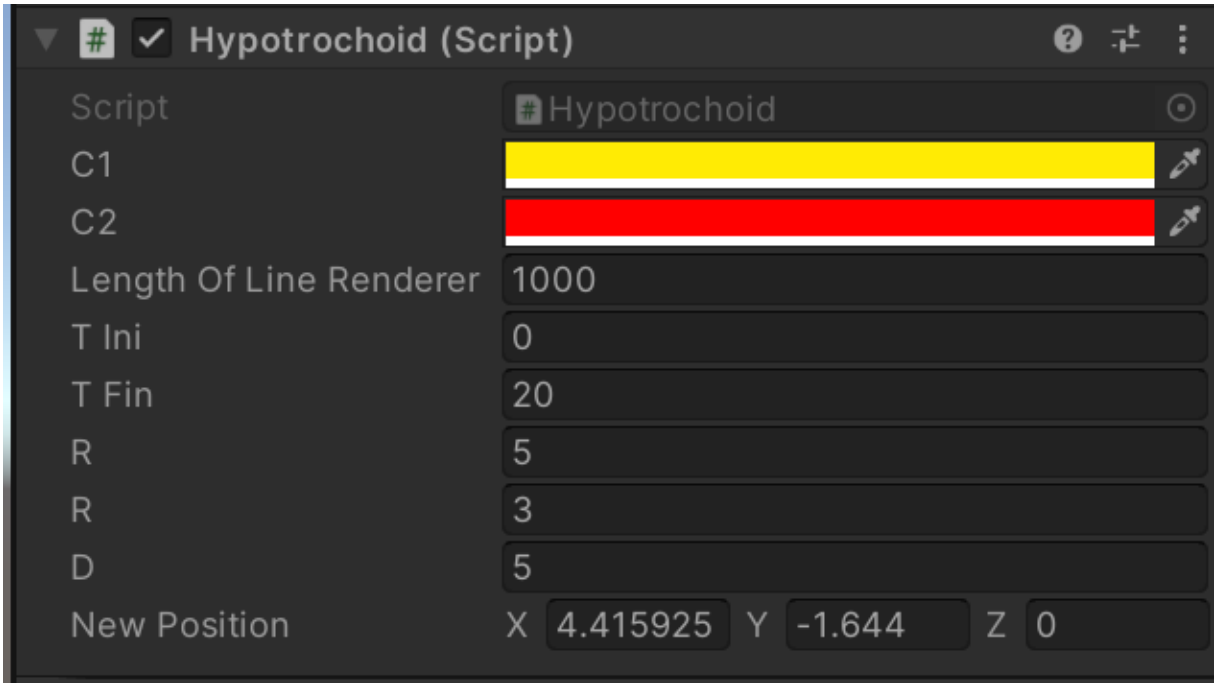

Figure 6. Hypotrochoid Script Parameters

Camera

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

public class CameraScript : MonoBehaviour
{
    public float speed = 10f;
    private float angle = 0f;
    private Vector3 center = Vector3.zero;

    void Update()
    {
        Vector3 pos = transform.position;
        Vector3 towardsCenter = center - pos;
        towardsCenter.Normalize();
        transform.rotation =
Quaternion.LookRotation(towardsCenter);
        transform.RotateAround(center, Vector3.up, speed *
Time.deltaTime);
    }
}
```

This script controls the rotation of the camera around a specified center point. It requires the use of the Unity engine and the Vector3 and Quaternion classes.

The 'speed' variable determines the speed at which the camera will rotate around the center point. The 'angle' variable is not used in this script.

In the 'Update' method, the current position of the camera is retrieved using 'transform.position'. The vector towards the center point is calculated by subtracting the camera's position from the center and normalizing the result. The camera's rotation is then set to face towards the center point using 'Quaternion.LookRotation'. Finally, the camera is rotated around the center point using 'Transform.RotateAround'. The first parameter of 'RotateAround' specifies the center point, while the second parameter determines the axis around which the camera will rotate (in this case, the Y axis, represented by 'Vector3.up'). The rotation speed is determined by multiplying 'speed' by 'Time.deltaTime'.

This script will help us to better visualize the following parametric curves in 3D

Helix

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

public class Helix : MonoBehaviour
{
    public Color c1 = Color.yellow;
    public Color c2 = Color.red;
    public int lengthOfLineRenderer = 1000;
    public float tIni = 0.0f;
    public float tFin = 30.0f;
    public float radius = 1.0f;
    public float pitch = 0.1f;

    void Start()
    {
        LineRenderer lineRenderer =
gameObject.AddComponent<LineRenderer>();
        lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));
        lineRenderer.widthMultiplier = 0.2f;
        lineRenderer.positionCount = lengthOfLineRenderer;

        // A simple 2 color gradient with a fixed alpha of 1.0f.
        float alpha = 1.0f;
        Gradient gradient = new Gradient();
        gradient.SetKeys(
            new GradientColorKey[] { new GradientColorKey(c1,
0.0f), new GradientColorKey(c2, 1.0f) },
            new GradientAlphaKey[] { new GradientAlphaKey(alpha,
0.0f), new GradientAlphaKey(alpha, 1.0f) }
        );
        lineRenderer.colorGradient = gradient;
    }

    (float x, float y, float z) helix(float t)
    {
```

```
        float xCoord = radius * Mathf.Cos(t);
        float yCoord = radius * Mathf.Sin(t);
        float zCoord = pitch * t;
        return (xCoord, yCoord, zCoord);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);

    void Update()
    {
        LineRenderer lineRenderer = GetComponent<LineRenderer>();

        float delta = (tFin - tIni) / (lengthOfLineRenderer - 1);
        float t = tIni, x, y, z;

        for (int i = 0; i < lengthOfLineRenderer; i++)
        {
            (x, y, z) = helix(t);
            newPosition.x = x;
            newPosition.y = y;
            newPosition.z = z;
            lineRenderer.SetPosition(i, newPosition);
            t += delta;
        }
    }
}
```

This code defines a script for a helix in Unity, which is represented by a Line Renderer. The helix is defined parametrically, with three parameters: radius, pitch, and time (t). The radius determines the radius of the helix, the pitch determines the distance between each coil of the helix, and the time parameter determines the position of the helix along its length.

The script sets up the Line Renderer with some initial parameters, including the color and number of positions along the curve. The helix function calculates the x, y, and z coordinates of the helix at a given time t, and the Update function sets the position of each point along the Line Renderer based on the coordinates calculated by the helix function.

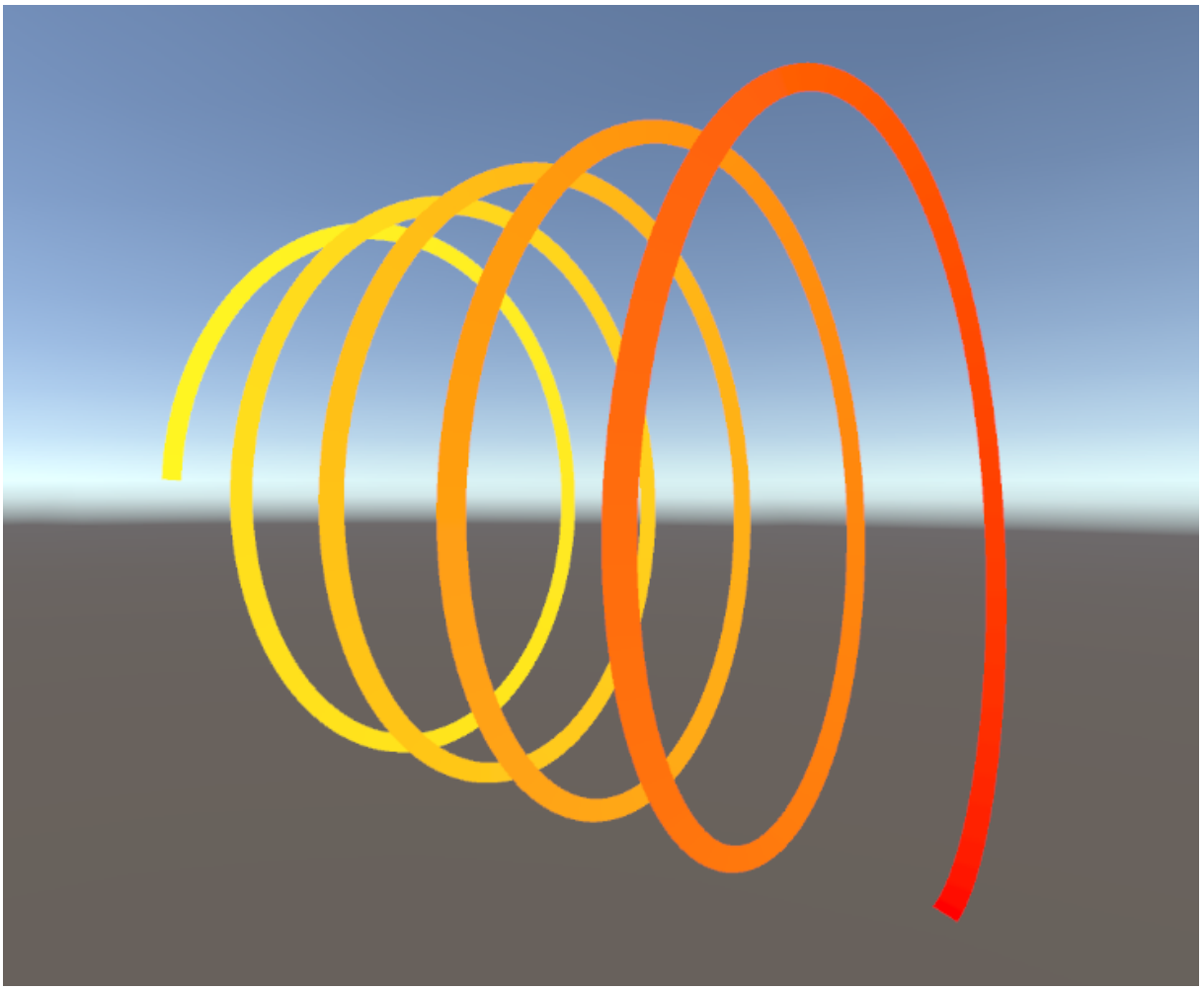

Figure 7. Helix Displayed

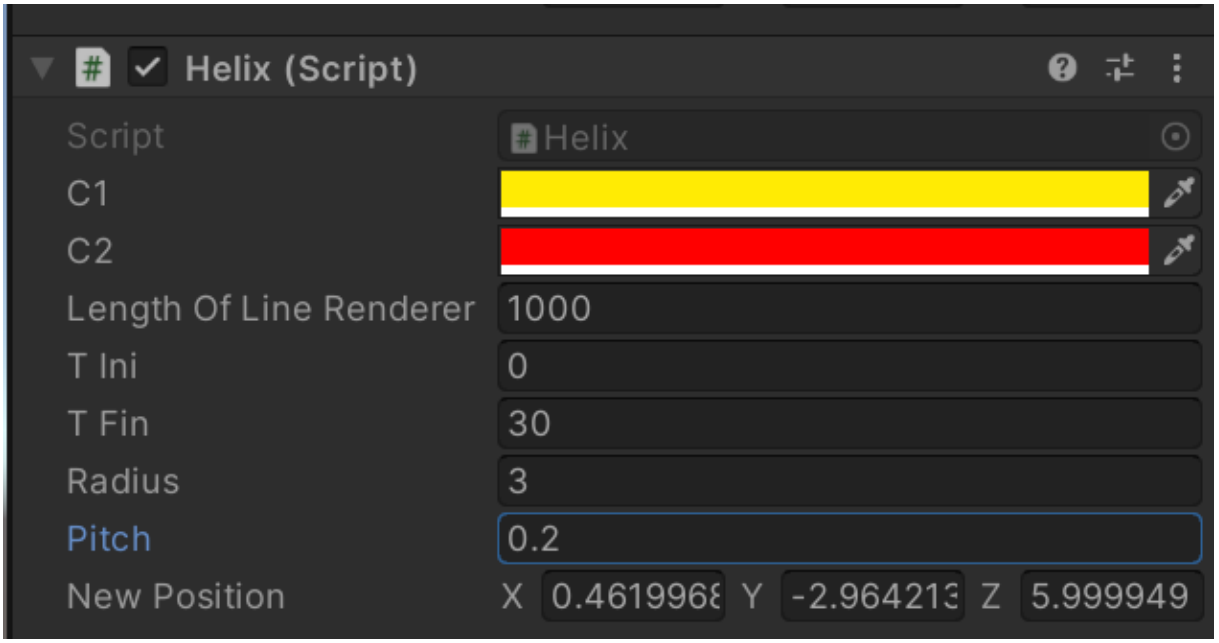

Figure 8. Helix Script Parameters

Torus

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

public class Torus : MonoBehaviour
{
    public Color c1 = Color.yellow;
    public Color c2 = Color.red;
    public int lengthOfLineRenderer = 1000;
    public float tIni = 0.0f;
    public float tFin = 100;
    public float R = 2.0f;
    public float r = 0.5f;

    void Start()
    {
        LineRenderer lineRenderer =
gameObject.AddComponent<LineRenderer>();
        lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));
        lineRenderer.widthMultiplier = 0.2f;
        lineRenderer.positionCount = lengthOfLineRenderer;

        // A simple 2 color gradient with a fixed alpha of 1.0f.
        float alpha = 1.0f;
        Gradient gradient = new Gradient();
        gradient.SetKeys(
            new GradientColorKey[] { new GradientColorKey(c1,
0.0f), new GradientColorKey(c2, 1.0f) },
            new GradientAlphaKey[] { new GradientAlphaKey(alpha,
0.0f), new GradientAlphaKey(alpha, 1.0f) }
        );
        lineRenderer.colorGradient = gradient;
    }

    (float x, float y, float z) torus(float t, float u)
    {
        float xCoord = (R + r * Mathf.Cos(u)) * Mathf.Cos(t);
        float yCoord = (R + r * Mathf.Cos(u)) * Mathf.Sin(t);
        float zCoord = r * Mathf.Sin(u);
```

```
        return (xCoord, yCoord, zCoord);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);

    void Update()
    {
        LineRenderer lineRenderer = GetComponent<LineRenderer>();

        float deltaT = (tFin - tIni) / (lengthOfLineRenderer - 1);
        float deltaU = (2 * Mathf.PI) / (lengthOfLineRenderer - 1);
        float t = tIni, u = 0.0f;
        float x, y, z;

        for (int i = 0; i < lengthOfLineRenderer; i++)
        {
            (x, y, z) = torus(t, u);
            newPosition.x = x;
            newPosition.y = y;
            newPosition.z = z;
            lineRenderer.SetPosition(i, newPosition);

            t += deltaT;
            u += deltaU;
            u %= (2 * Mathf.PI);
        }
    }
}
```

This is a C# script for a Torus object in Unity. The Torus is rendered as a 3D shape by a LineRenderer component, which draws a line through a set of points defined by the script.

The script defines several public variables that can be edited in the Unity Editor, including the colors for the Torus, the number of points to draw, and the radii of the Torus.

The Start() method initializes the LineRenderer component and sets its width and color gradient.

The torus() method computes the (x, y, z) coordinates for a given pair of angles (t, u) using equations for the parametric representation of a torus.

The Update() method generates the points for the LineRenderer by looping through values of (t, u) and computing the corresponding (x, y, z) coordinates. These coordinates are stored in a Vector3 object and used to update the position of the LineRenderer point at each iteration.

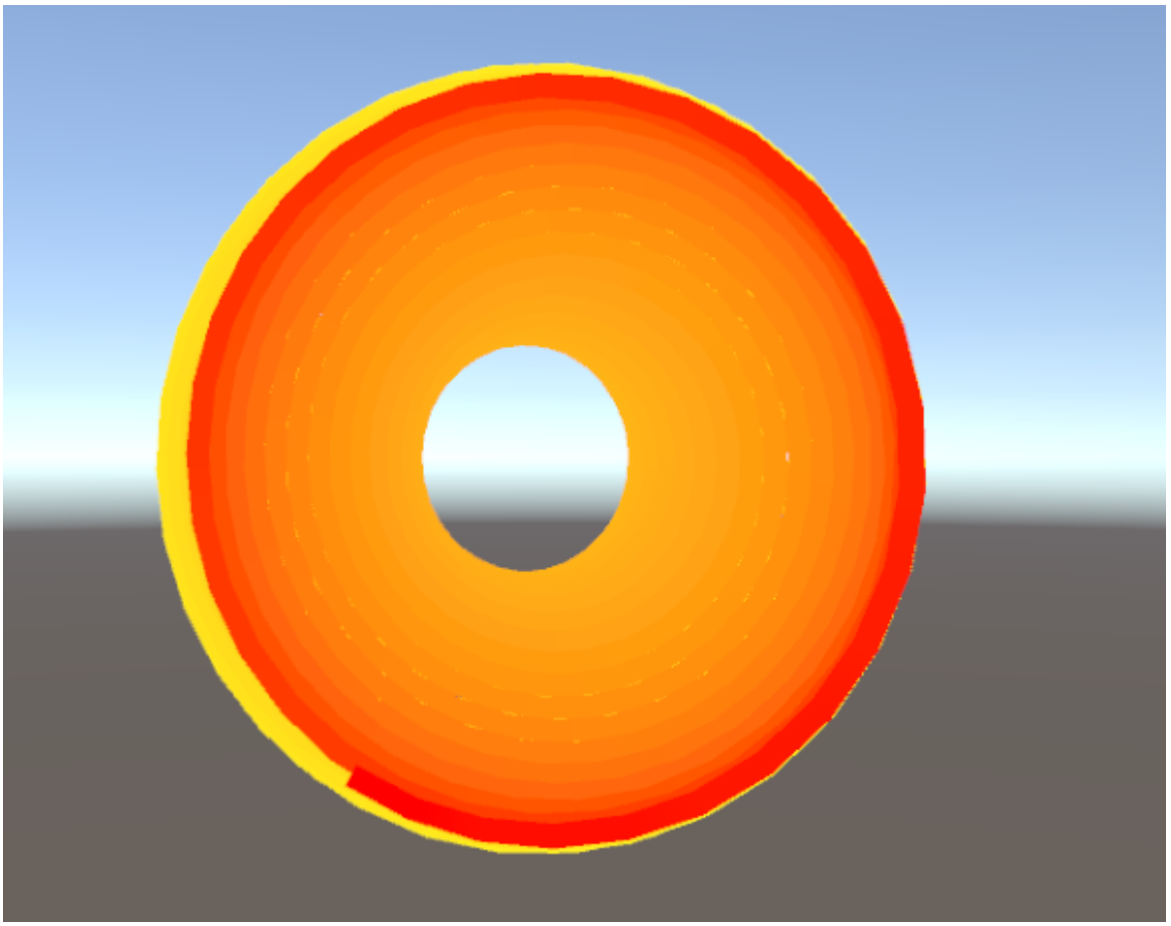

Figure 9. Torus Displayed

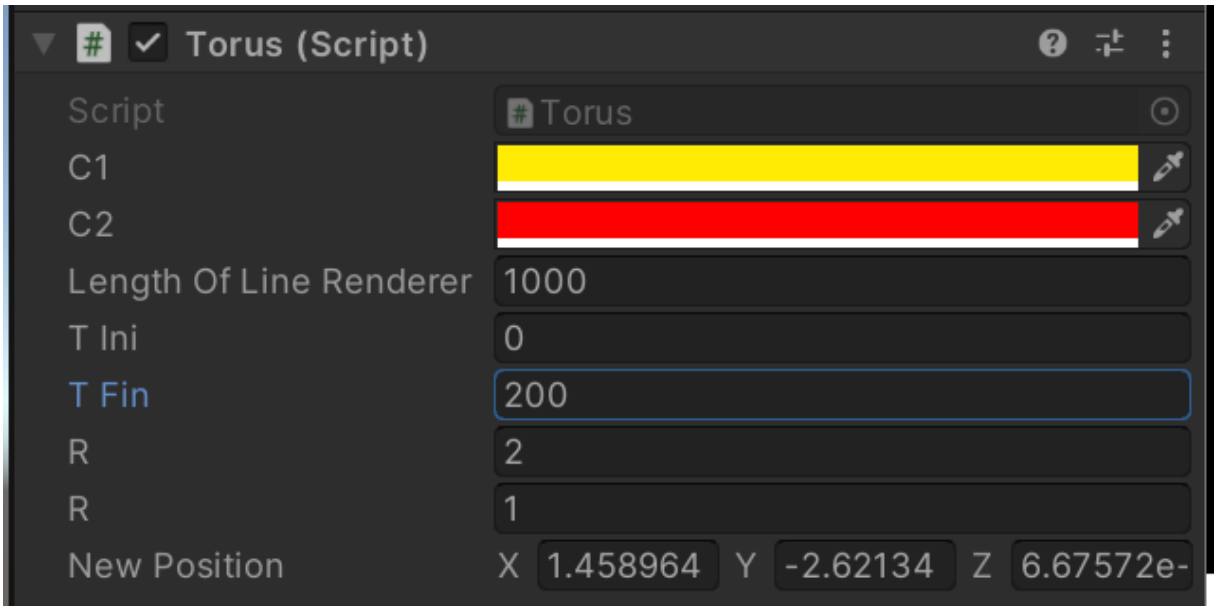

Figure 10. Torus Script Parameters

Parametric Trajectories

Parabola

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

public class Parabola : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    (float x, float y) parabola(float t)
    {
        float x = t;
        float y = t * t;
        return (x, y);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);
    public float t = 0;
```

```
    public float direction = 1.0f;

    // Update is called once per frame
    void Update()
    {
        float x, y;
        t += Time.deltaTime * 5 * direction;
        (x, y) = parabola(t);
        newPosition.x = x;
        newPosition.y = y;
        transform.position = newPosition;

        // Change direction when reaching the max points
        if (newPosition.x >= 5.0f || newPosition.x <= -5.0f)
        {
            direction *= -1.0f;
        }
    }
}
```

This is a C# script for a Unity game object called "Parabola". It contains a function called "parabola" that takes a float parameter "t" and returns a tuple of two float values (x, y) that represent the position of a point on a parabolic curve.

In the "Update" method, the script updates the value of "t" by adding the product of "Time.deltaTime" (a measure of time since the last frame) and 5 multiplied by the "direction" variable. The parabola function is then called with the updated value of "t" to get the new position of the game object.

The new position is then assigned to a Vector3 variable called "newPosition", and the game object's position is set to this new position using the "transform.position" property.

Finally, the script checks whether the game object has reached the maximum points of the x-axis (either 5.0 or -5.0), and if so, it changes the direction by multiplying the "direction" variable by -1.0.

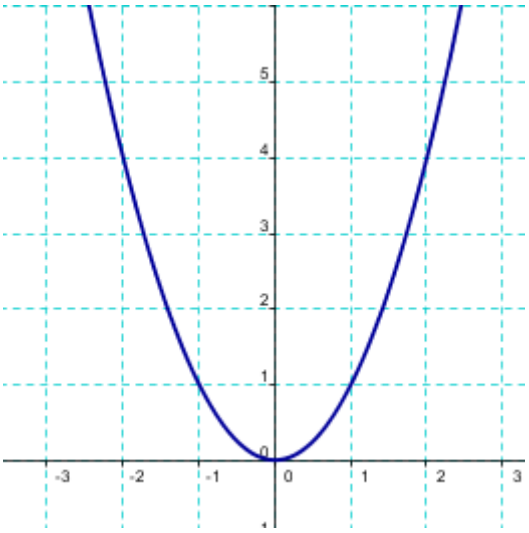

Figure 11. Expected trajectory of the parabola

Trefoil Knot

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

public class TrefoilKnot : MonoBehaviour
{
    public float a = 1.0f;
    public float b = 2.0f;
    public float c = 3.0f;

    void Start()
    {
    }

    (float x, float y, float z) trefoil(float t)
    {
        float xCoord = (a + b * Mathf.Cos(c * t)) * Mathf.Cos(t);
        float yCoord = (a + b * Mathf.Cos(c * t)) * Mathf.Sin(t);
        float zCoord = b * Mathf.Sin(c * t);

        return (xCoord, yCoord, zCoord);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);
    public float t;
```

```
    void Update()
    {
        float x, y, z;
        t += Time.deltaTime * 5;
        (x, y, z) = trefoil(t);
        newPosition.x = x - 7;
        newPosition.y = y;
        newPosition.z = z;
        transform.position = newPosition;
    }
}
```

This is a C# script for a Unity game object called "TrefoilKnot". It contains a function called "trefoil" that takes a float parameter "t" and returns a tuple of three float values (x, y, z) that represent the position of a point on a trefoil knot.

In the "Update" method, the script updates the value of "t" by adding the product of "Time.deltaTime" (a measure of time since the last frame) and 5. The trefoil function is then called with the updated value of "t" to get the new position of the game object.

The new position is then assigned to a Vector3 variable called "newPosition", with the x-coordinate shifted left by 7 units. Finally, the game object's position is set to this new position using the "transform.position" property.

The script also defines three public float variables "a", "b", and "c" that control the shape of the trefoil knot. These variables can be adjusted in the Unity editor to create different variations of the knot. The "Start" method is empty and not used in this script.

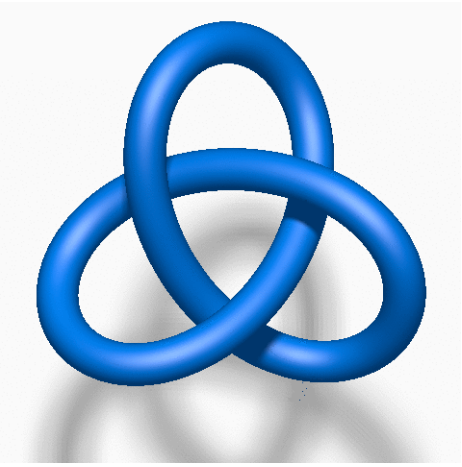

Figure 12. Expected trajectory of the TrefoilKnot

Rhodonea Curves

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

public class RhodoneaCurves : MonoBehaviour
{
    public int n = 5;  // number of petals
    public float a = 1.0f;  // radius of the petal
    public float stepSize = 0.005f;  // step size between each
point on the curve

    void Start()
    {
    }

    (float x, float y) rhodonea(float t)
    {
        float cosT = Mathf.Cos(t);
        float sinT = Mathf.Sin(t);

        float xCoord = a * Mathf.Cos(n * t) * cosT;
        float yCoord = a * Mathf.Cos(n * t) * sinT;

        return (xCoord, yCoord);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);
    public float t;

    void Update()
    {
        float x, y;
        t += stepSize;
        (x, y) = rhodonea(t);
        newPosition.x = x - 7;
        newPosition.y = y;
        transform.position = newPosition;
    }
}
```

This is a C# script for a Unity game object called "RhodoneaCurves". It contains a function called "rhodonea" that takes a float parameter "t" and returns a tuple of two float values (x, y) that represent the position of a point on a Rhodonea curve.

In the "Update" method, the script updates the value of "t" by adding the "stepSize" variable, which determines the distance between each point on the curve. The rhodonea function is then called with the updated value of "t" to get the new position of the game object.

The new position is then assigned to a Vector3 variable called "newPosition", with the x-coordinate shifted left by 7 units. Finally, the game object's position is set to this new position using the "transform.position" property.

The script also defines three public variables: "n" determines the number of petals in the curve, "a" determines the radius of each petal, and "stepSize" determines the distance between each point on the curve. These variables can be adjusted in the Unity editor to create different variations of the curve. The "Start" method is empty and not used in this script.

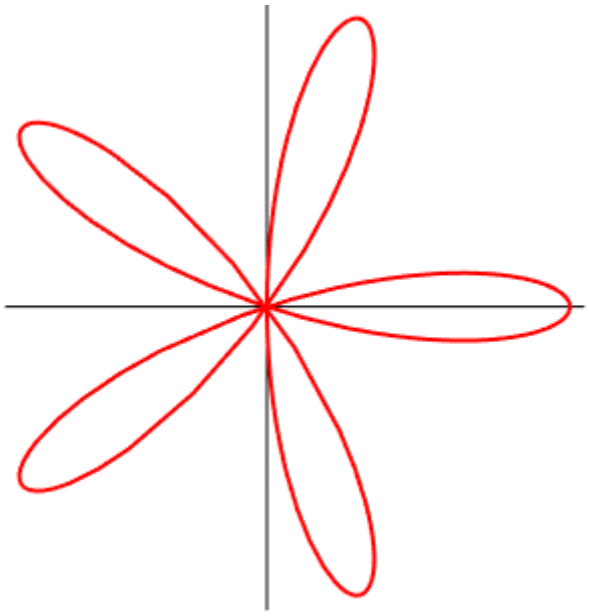

Figure 13. Expected trajectory of the Rhodonea Curves

Cycloid

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

public class Cycloid : MonoBehaviour
{
    private float timer;
    private bool canRestart;

    // Start is called before the first frame update
    void Start()
    {
        timer = 0.0f;
        canRestart = false;
    }

    (float x, float y) cycloid(float t)
    {
        return (0.5F * t - 0.8F * Mathf.Sin(t), 0.5F - 0.8F *
Mathf.Cos(t));
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);
    public float t;

    // Update is called once per frame
    void Update()
    {
        float x, y;
        t += Time.deltaTime * 5;//*0.01F
        (x, y) = cycloid(t);
        newPosition.x = x - 7;
        newPosition.y = y;
        transform.position = newPosition;

        if (canRestart)
        {
            timer += Time.deltaTime;

            if (timer >= 10.0f)
```

```
            {
                t = 0.0f;
                timer = 0.0f;
                canRestart = false;
            }
        }
        else if (t >= 6.28f)
        {
            canRestart = true;
        }
    }
}
```

This script defines a cycloid curve and moves a game object along the curve using Unity's Update function. The cycloid curve is defined in the cycloid function and takes a time parameter t as input, which is used to calculate the x and y coordinates of the curve. The function returns a tuple (x, y) of the coordinates.

In the Update function, the cycloid function is called with an increasing time parameter t, and the returned (x, y) coordinates are assigned to a Vector3 variable newPosition, which is used to set the position of the game object. The game object moves along the cycloid curve.

The script also includes a timer that is used to restart the cycloid curve after a set time period (10 seconds in this case). The canRestart boolean is used to check if the timer is running, and once it is finished, the t parameter is reset to 0 and canRestart is set back to false. The script also checks if t has reached the end of the cycloid curve and sets canRestart to true to start the timer.

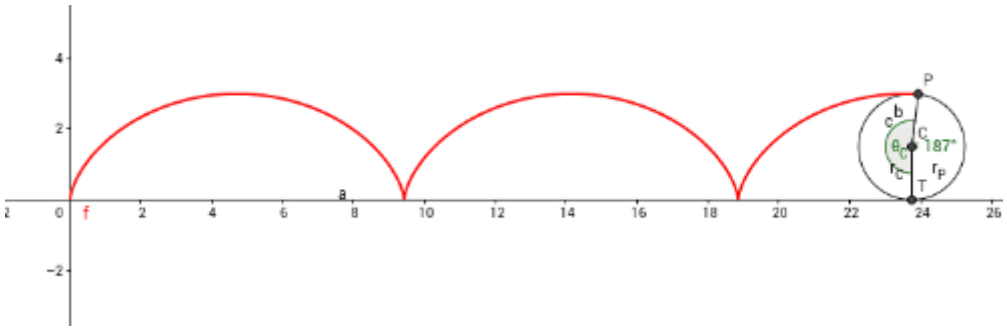

Figure 14. Expected trajectory of the cycloid

Astroid

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

public class Astroid : MonoBehaviour
{
    void Start()
    {
    }

    (float x, float y) astroid(float t)
    {
        float xCoord = Mathf.Pow(Mathf.Cos(t), 3);
        float yCoord = Mathf.Pow(Mathf.Sin(t), 3);

        return (xCoord, yCoord);
    }

    public Vector3 newPosition = new Vector3(0, 0, 0);
    public float t;

    void Update()
    {
        float x, y;
        t += Time.deltaTime * 5;
        (x, y) = astroid(t);
        newPosition.x = x - 7;
        newPosition.y = y;
        transform.position = newPosition;
    }
}
```

This is a script in Unity that defines the behavior of an object in the game world. The object moves along a path defined by the astroid curve.

- The `Start()` method is called once at the beginning of the script, but it is not used in this case.
- The `astroid(float t)` method calculates the x and y coordinates of a point on the astroid curve at a given value of t. The astroid curve is defined by the equation x = cos^3(t), y = sin^3(t).

- The `newPosition` variable stores the current position of the object in the game world.
- The `t` variable keeps track of the current value of t, which is used to calculate the position of the object on the astroid curve.
- The `Update()` method is called once per frame, and it updates the position of the object based on the current value of t. It uses the `Time.deltaTime` value to calculate the distance to move the object in each frame.
- The `x` and `y` values are calculated by calling the `astroid(float t)` method with the current value of t.
- The `newPosition.x` and `newPosition.y` values are set to the calculated `x` and `y` values, respectively.
- Finally, the `transform.position` of the object is set to the new position stored in `newPosition`.

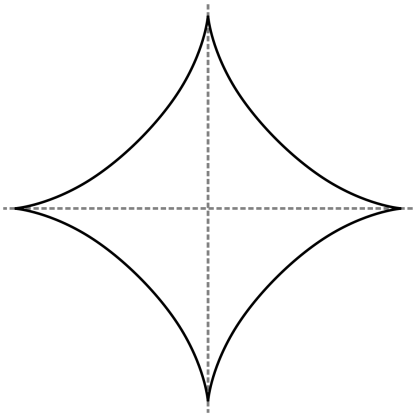

Figure 15. Expected trajectory of the astroid

Hermite Curves

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

public class HermiteCurves : MonoBehaviour
{
    public Color c1 = Color.yellow;
    public Color c2 = Color.red;
    public int lengthOfLineRenderer = 1000;
    public Vector3 startPoint = new Vector3(0, 0, 0);
    public Vector3 endPoint = new Vector3(10, 10, 0);
    public Vector3 startTangent = new Vector3(2, 2, 0);
    public Vector3 endTangent = new Vector3(-2, -2, 0);

    void Start()
    {
        LineRenderer lineRenderer =
gameObject.AddComponent<LineRenderer>();
        lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));
        lineRenderer.widthMultiplier = 0.2f;
        lineRenderer.positionCount = lengthOfLineRenderer;

        // A simple 2 color gradient with a fixed alpha of 1.0f.
        float alpha = 1.0f;
        Gradient gradient = new Gradient();
        gradient.SetKeys(
            new GradientColorKey[] { new GradientColorKey(c1,
0.0f), new GradientColorKey(c2, 1.0f) },
            new GradientAlphaKey[] { new GradientAlphaKey(alpha,
0.0f), new GradientAlphaKey(alpha, 1.0f) }
        );
        lineRenderer.colorGradient = gradient;
    }

    public Vector3 CalculateHermite(Vector3 p0, Vector3 p1, Vector3
m0, Vector3 m1, float t)
    {
        float t2 = t * t;
        float t3 = t2 * t;
```

```
        float h00 = 2 * t3 - 3 * t2 + 1;
        float h10 = t3 - 2 * t2 + t;
        float h01 = -2 * t3 + 3 * t2;
        float h11 = t3 - t2;

        return h00 * p0 + h10 * m0 + h01 * p1 + h11 * m1;
    }

    public Vector3 newPoint = new Vector3(0, 0, 0);

    void Update()
    {
        LineRenderer lineRenderer = GetComponent<LineRenderer>();

        float delta = 1.0f / (lengthOfLineRenderer - 1);

        for (int i = 0; i < lengthOfLineRenderer; i++)
        {
            float t = delta * i;
            newPoint = CalculateHermite(startPoint, endPoint,
startTangent, endTangent, t);
            lineRenderer.SetPosition(i, newPoint);
        }
    }
}
```

This is a Unity script written in C# that creates a Hermite curve and displays it as a line in the Unity game engine.

First, the script defines some variables for the color and length of the line renderer, as well as the starting and ending points of the Hermite curve and their respective tangents.

In the Start() method, a LineRenderer component is added to the game object the script is attached to, and some properties are set to define the appearance of the line.

Next, the script defines a method called CalculateHermite() which takes four Vector3 points (p0, p1, m0, m1) and a float value (t) as input parameters. This method calculates a new point on the Hermite curve using the Hermite interpolation formula.

In the Update() method, the script loops through a number of points equal to the lengthOfLineRenderer variable, and calculates a new point on the Hermite curve for each point in the loop using the CalculateHermite() method. The resulting points are then set as the position of the line renderer using the SetPosition() method.

This creates a smooth line that follows the path of the Hermite curve defined by the starting and ending points and their respective tangents. As the script is attached to a game object, the line is displayed in the Unity game engine.

Volute

A volute is a decorative, spiral-shaped ornament or design that is often found in architecture, furniture, and other decorative arts.

In architecture, a volute is typically used as a decorative element on columns or as part of a capital, which is the topmost part of a column or pillar. The volute can be found in various forms, such as a single spiral curve or multiple, overlapping curves. It is often used as a transitional element between the column and the capital or as a design element in its own right.

In furniture design, a volute can be used as a decorative element on the arms or legs of a chair or as a decorative motif on the backrest or headboard of a bed.

The shape of a volute is often inspired by the natural curves and spirals found in shells, plants, and other natural forms. The design can be simple or complex, depending on the intended use and the style of the architecture or furniture.

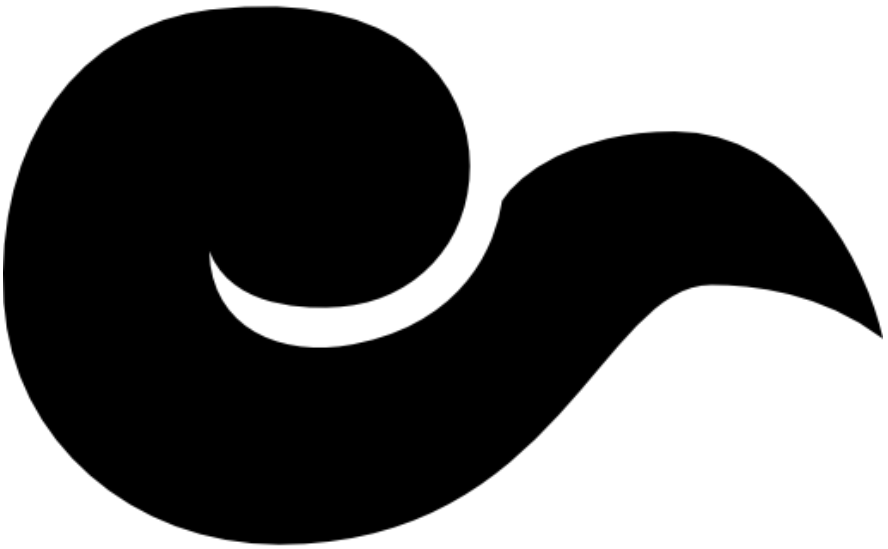

Figure 16. Volute

To build a volute in Unity I used 4 Hermite curves in total. The parameters of each of these curves and the final result are shown below.

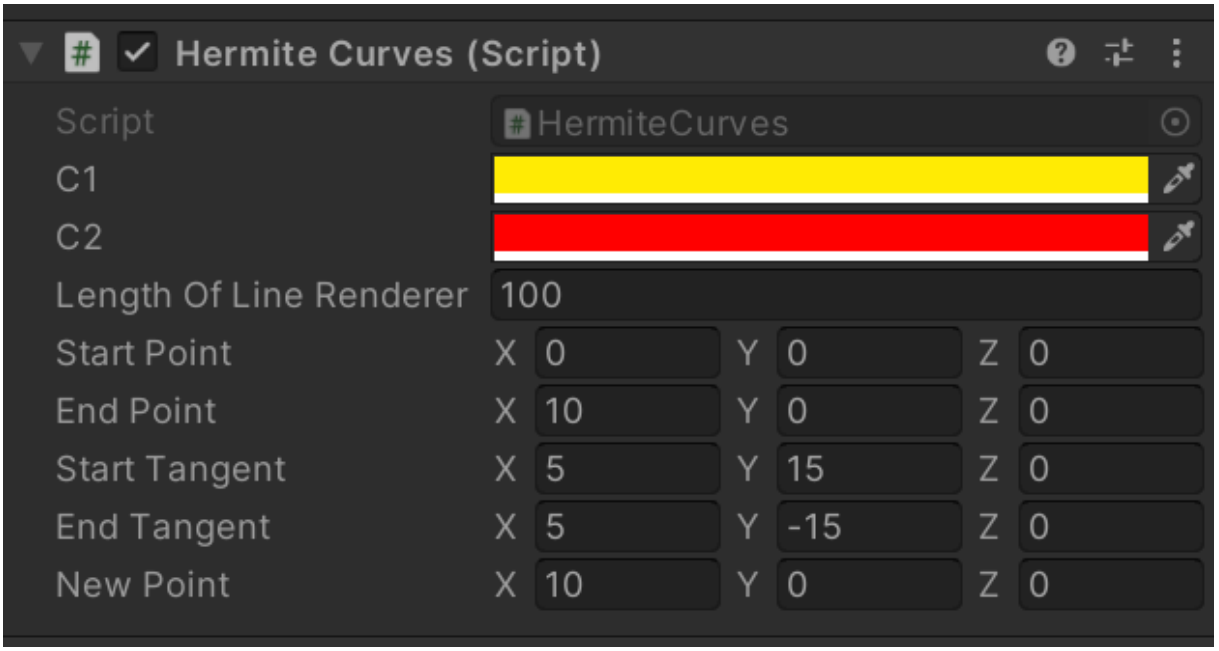

Figure 17. First Hermite Curve to build the Volute

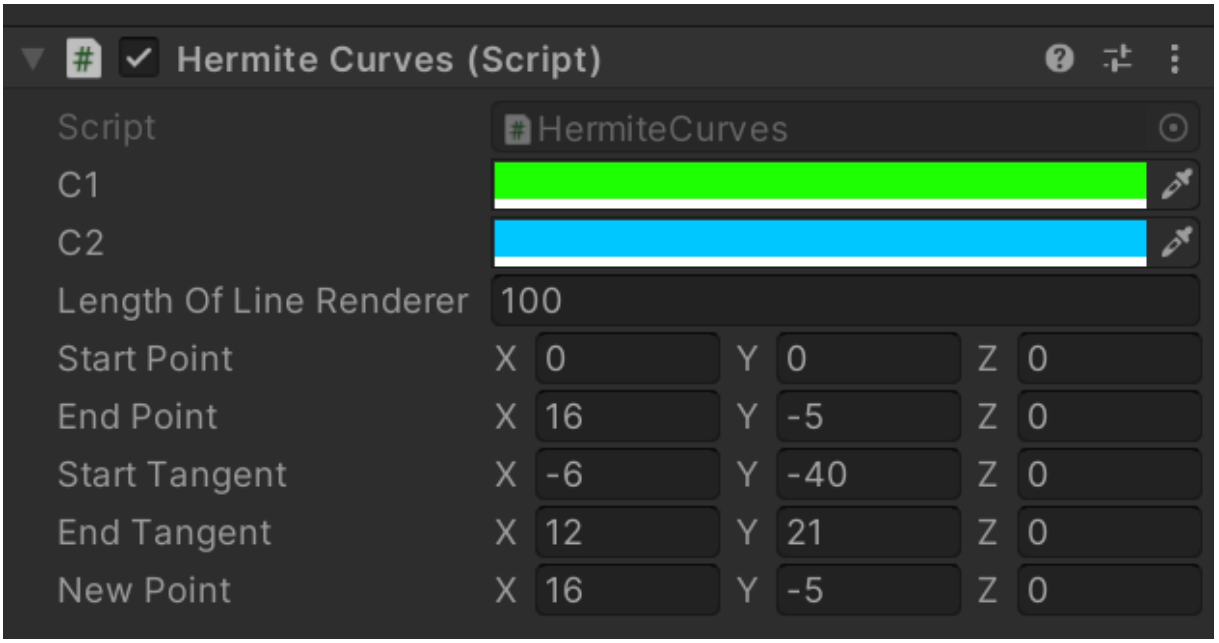

Figure 18. Second Hermite Curve to build the Volute

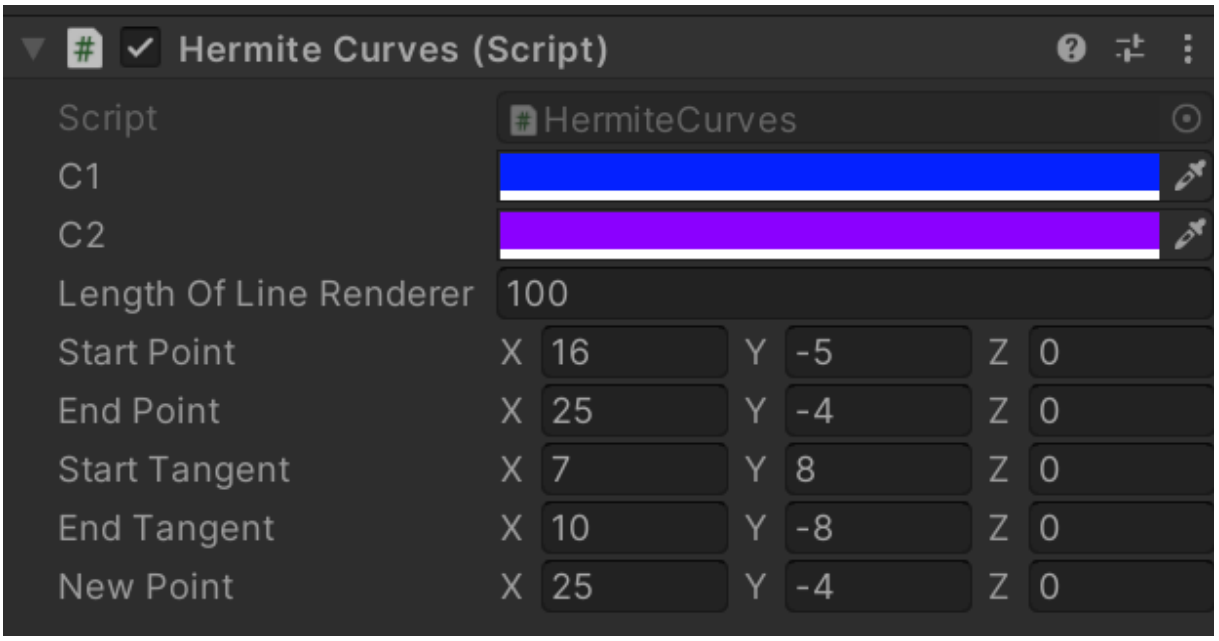

Figure 19. Third Hermite Curve to build the Volute

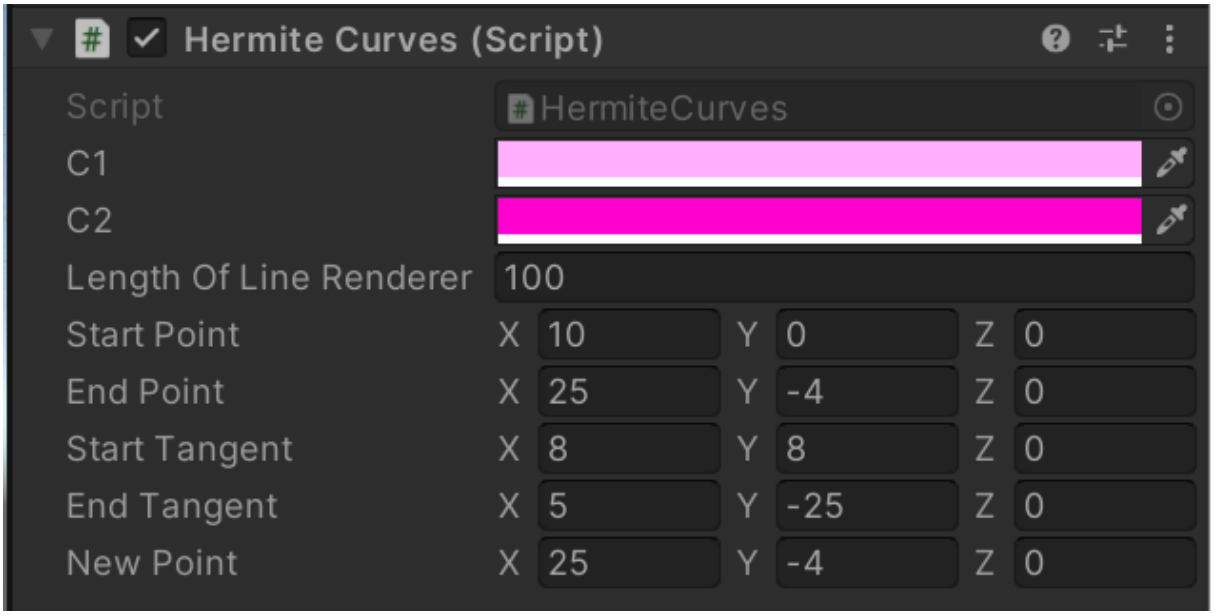

Figure 20. Fourth Hermite Curve to build the Volute

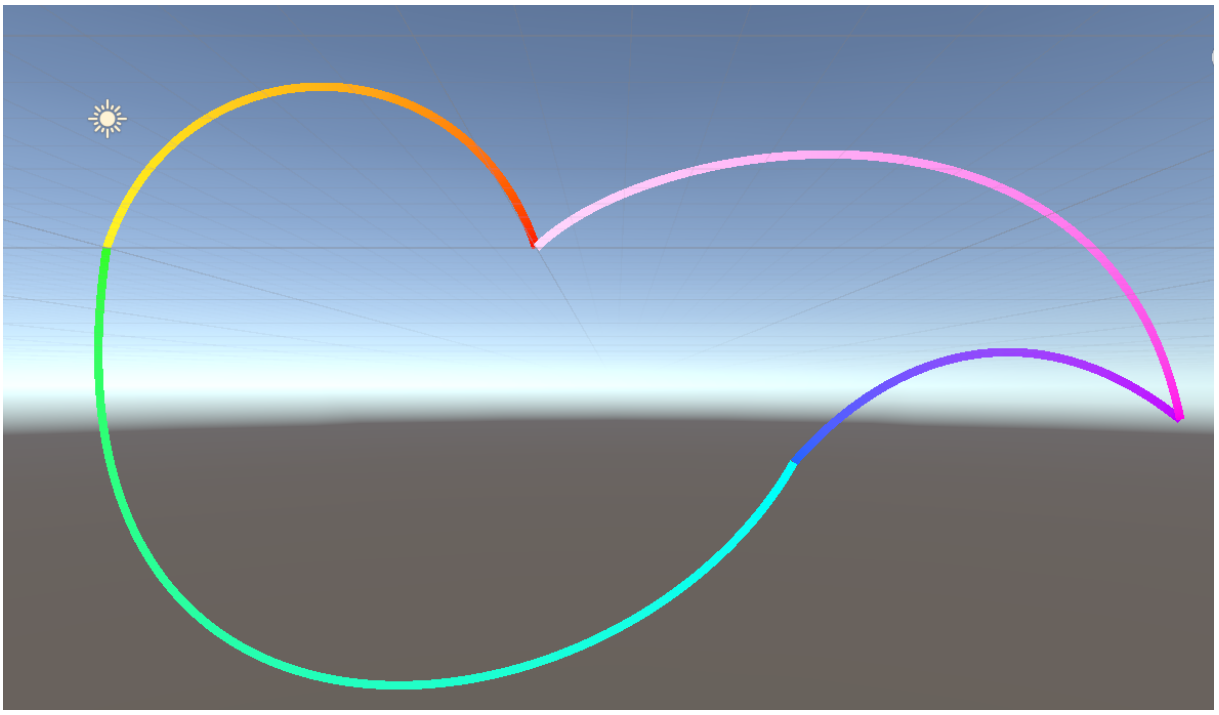

Figure 21. Volute built with 4 Hermite curves

Club

The club is one of the four suits in a standard deck of playing cards used for games like poker. The club symbol is depicted as a three-leaf clover with a stem, usually colored black. Each leaf is curved, creating a rounded shape overall. The stem extends from the bottom of the clover and is also curved, giving the symbol a slightly elongated shape. The club symbol is often associated with the element of wood, and in some interpretations, it is said to represent a weapon or a tool.

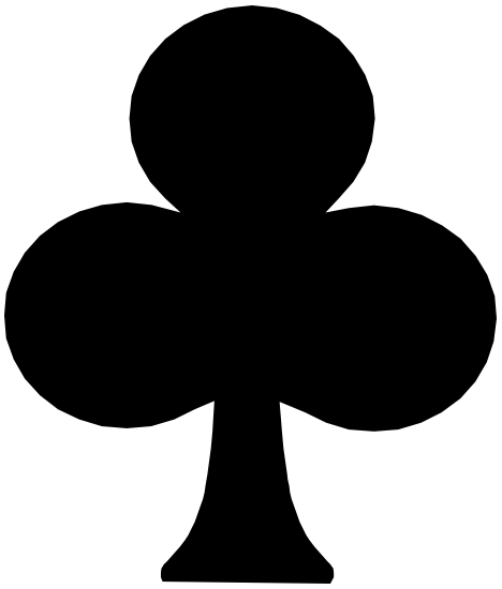

Figure 22. Club

To build a club in Unity I used 6 Hermite curves in total. The parameters of each of these curves and the final result are shown below.

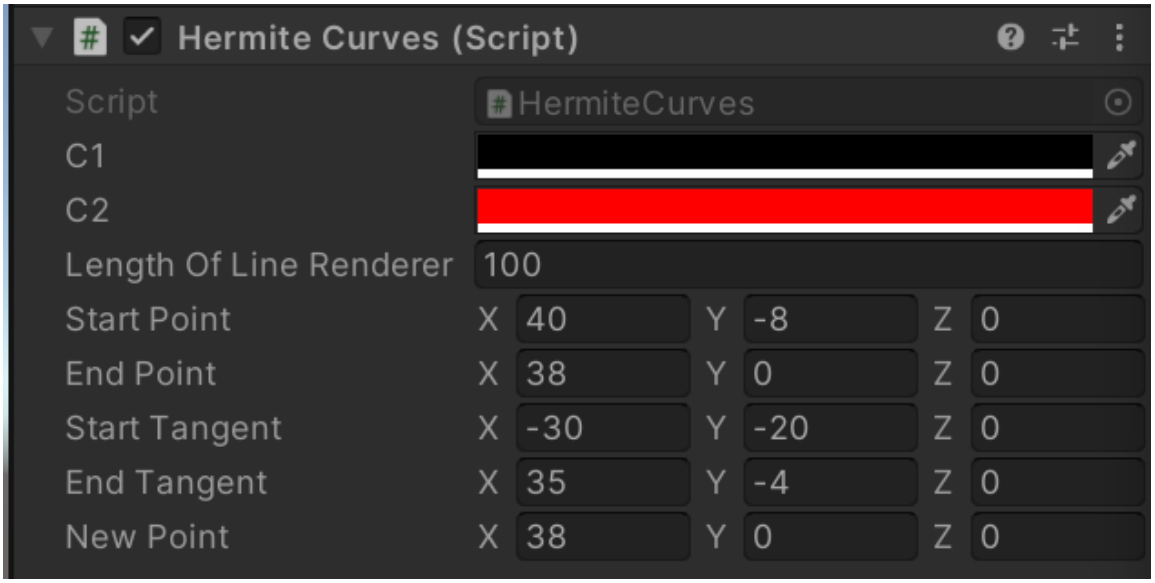

Figure 23. First Hermite Curve to build the Club

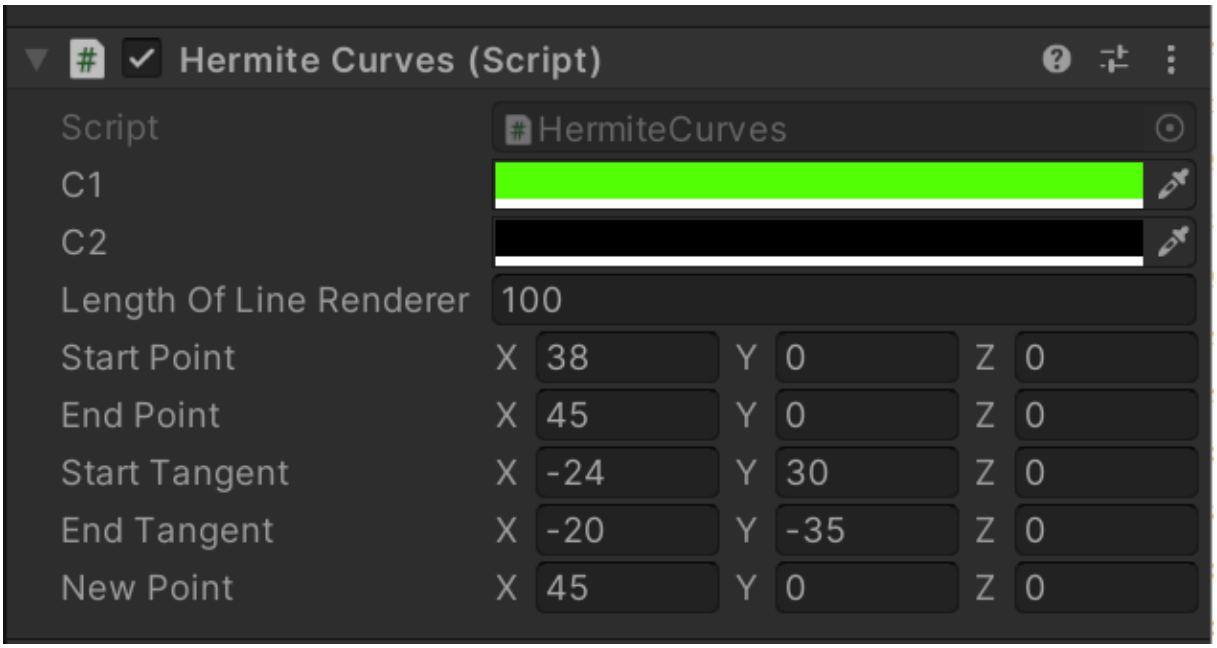

Figure 24. Second Hermite Curve to build the Club

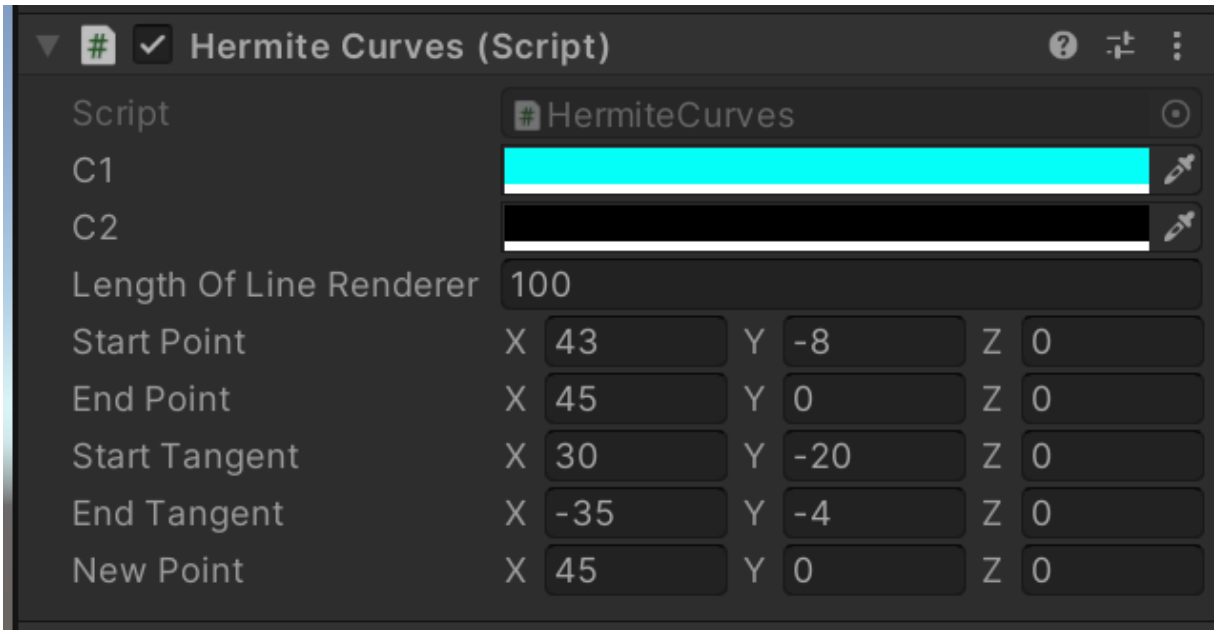

Figure 25. Third Hermite Curve to build the Club

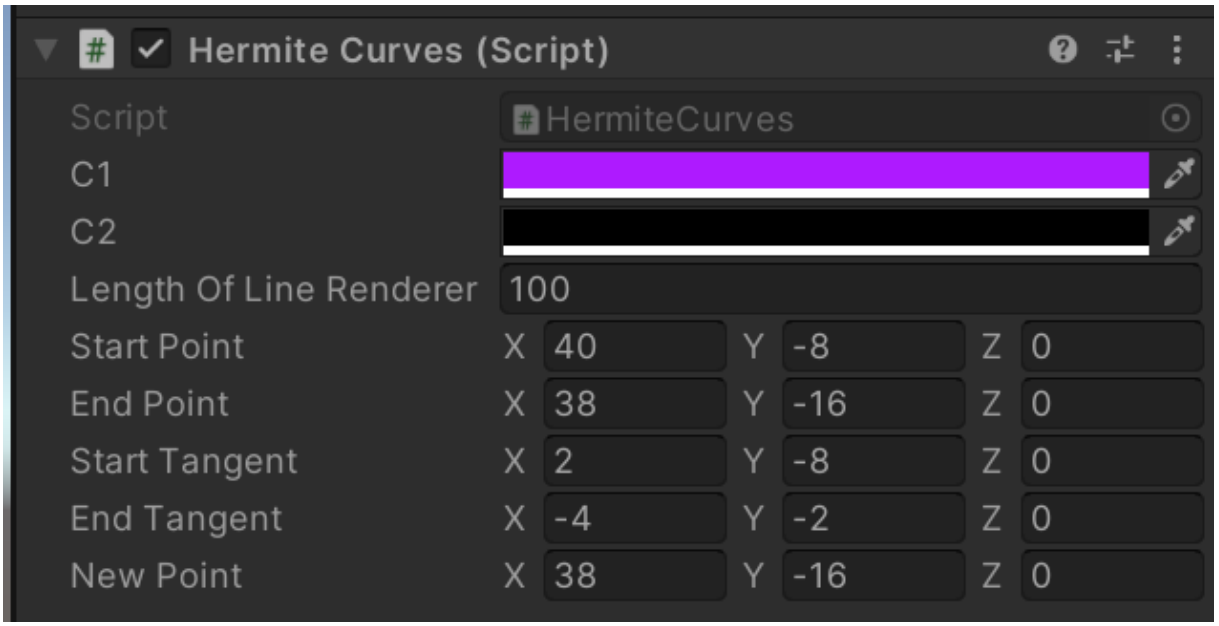

Figure 26. Fourth Hermite Curve to build the Club

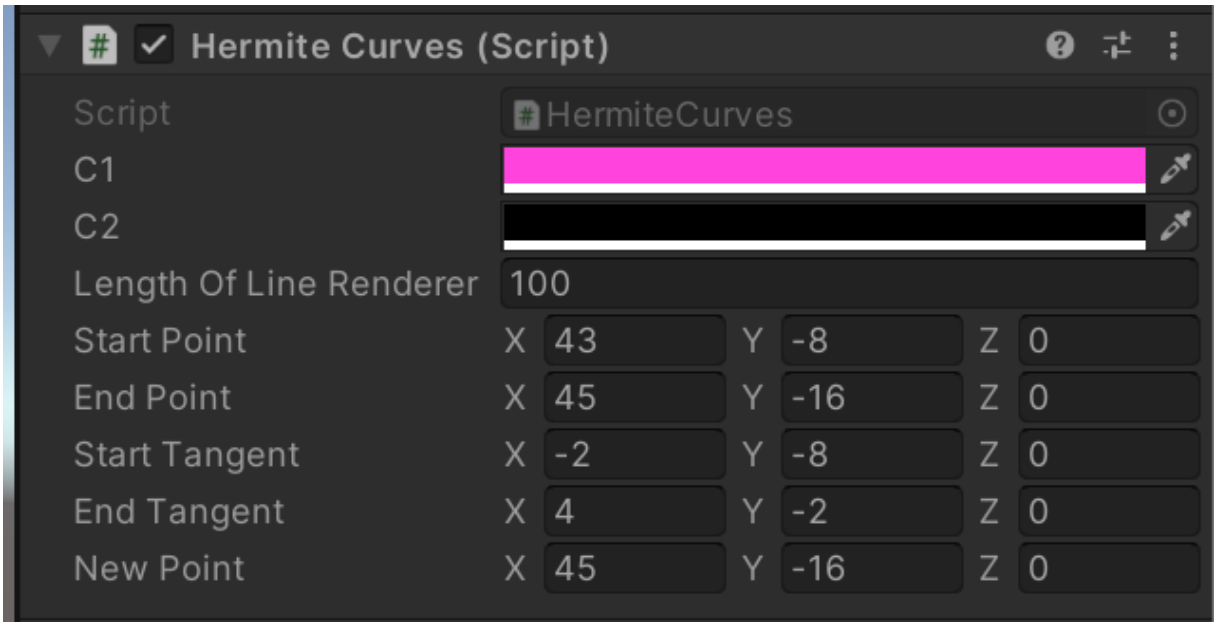

Figure 27. Fifth Hermite Curve to build the Club

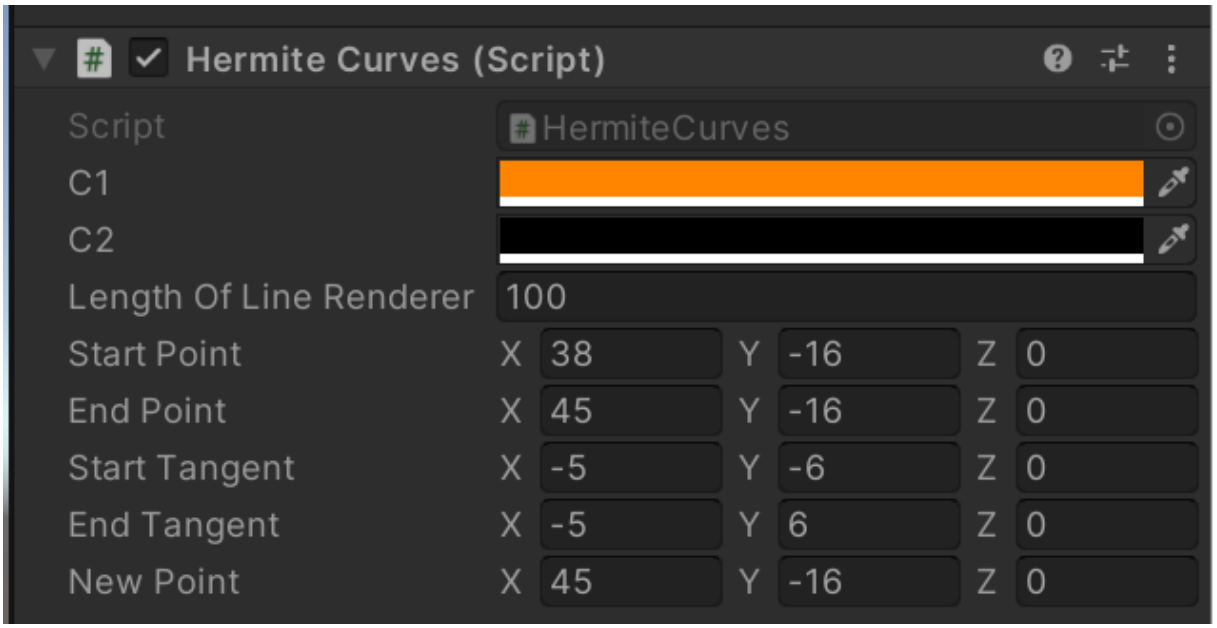

Figure 28. Sixth Hermite Curve to build the Club

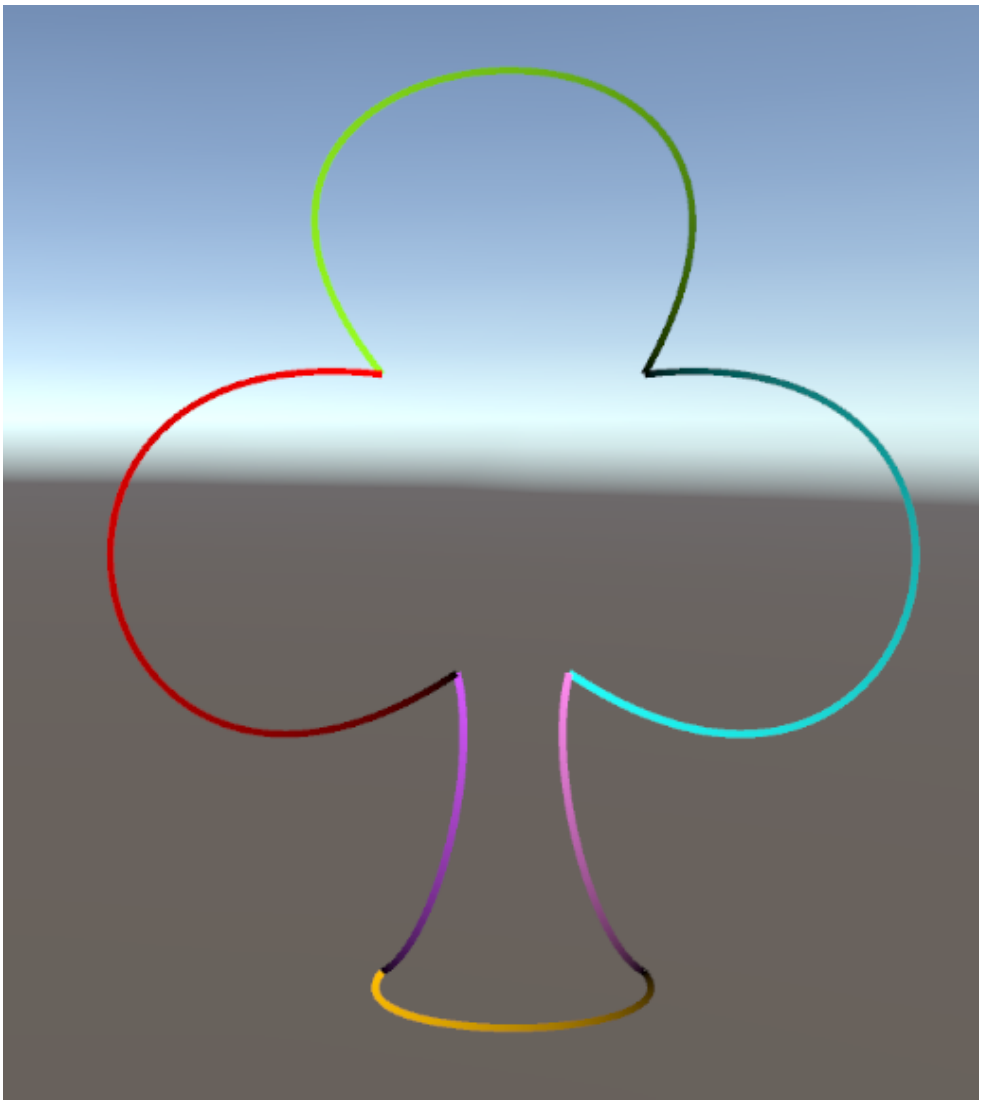

Figure 29. Club built with 6 Hermite curves

Conclusions and recommendations

In Unity, rendering parametric curves and moving objects in parametric trajectories can be accomplished using the built-in animation tools or by scripting the movement using mathematical formulas. Hermit curves can be used to build other curve shapes by adjusting the control points of the curve. By using these techniques, it is possible to create complex animations and movements in Unity. To achieve the best results, it is recommended to use the built-in animation tools for simplicity and ease of adjustment, and to use smooth mathematical formulas to avoid jerky movements. Experimenting with different Hermit curve configurations can help create custom curve shapes. To optimize animations for performance, it is important to minimize the number of objects being animated and reduce the complexity of curve shapes whenever possible. By following these recommendations, you can create high-quality, optimized animations and movements in Unity.

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. Park, H., Lee, J., Lee, Y., & Kim, H. (2017). A GPU-based real-time Hermit interpolation for soft and smooth curves. The Journal of Supercomputing, 73(3), 1213-1227. https://doi.org/10.1007/s11227-016-1889-9
3. Unity Technologies. (2021). Unity documentation. https://docs.unity3d.com/Manual/index.html
4. Unity Technologies. (2021). LineRenderer.SetPosition Method. Retrieved September 10, 2021, from https://docs.unity3d.com/ScriptReference/LineRenderer.SetPosition.html
5. Wikipedia contributors. (2021, September 1). Parametric equation. In Wikipedia, The Free Encyclopedia. Retrieved September 10, 2021, from https://en.wikipedia.org/wiki/Parametric_equation
6. Wikipedia contributors. (2021, August 28). Parametric surface. In Wikipedia, The Free Encyclopedia. Retrieved September 10, 2021, from https://en.wikipedia.org/wiki/Parametric_surface
7. Farin, G. (2014). Curves and surfaces for CAGD: A practical guide (5th ed.). Morgan Kaufmann Publishers.