VR FRAKTALE DOKUMENTATION

Lars Mager - lama0007@stud.hs-kl.de

Marcel Ochs - maoc0002@stud.hs-kl.de

29.07.2022

AR VR - Hochschule Kaiserslautern Standort Zweibrücken

Einleitung

Aufgabe

Ziel der Arbeit ist es, eine VR Anwendung mit OpenVR in der Unity Engine zu entwickeln. In dieser soll der Nutzer 2D Fraktale betrachten können. Zudem soll es möglich sein, Einstellungen an den Fraktalen vorzunehmen, welche in Echtzeit das Fraktal verändern. Als Beispiel werden das Mandelbrot, die Julia-Menge und Sierpinski-Dreieck vorgegeben. Der Raum soll an eine Galerie erinnern. Über Teleportation kann der Nutzer sich in der virtuellen Welt bewegen und so alle Fraktale, die im Raum verteilt sind, betrachten.

Probleme

Zu Beginn der Arbeit wurde das Sierpinski-Dreieck mit einem gewöhnlichen Algorithmus gerendert. Der CPU hat sich dabei als zu leistungsschwach herausgestellt, um die Fraktale in Echtzeit anzupassen. Dabei gab es Renderzeiten von mehreren Sekunden. Um dennoch die Fraktale darzustellen, mussten das Programm schneller laufen. Das sollte mithilfe eines Shader-Programms erzielt werden. Das Shader-Programm wird auf der Grafikkarte ausgeführt und mit deren Hilfe ist es dann möglich, die Fraktale in wenigen Millisekunden zu zeichnen.

Umsetzung

Erstellung der Galerie

Für die Galerie wurde in Blender ein Modell entworfen, mit einem Maß von 10 × 10 Meter. Als Referenz werden einige Google Bilder unter dem Suchbegriff *Bildgalerie* ausgewertet. Standardmäßig haben alle Bildergalerien einen weißen Hintergrund, um die Wirkung auf das Bild zu stärken. Daher wird der Raum relativ schlicht gehalten, mit einer weißen Wand und minimalen Verzierungen in den ersten 50 Zentimeter Höhe. Um den Raum zu füllen, damit werden zusätzlich Säulen modelliert und Fenster in die Decke platziert.

User Interface

Das User-Interface (UI) beinhaltet Slider und Buttons zur Bedienung der Fraktale. Dafür wurde das Unity UI System verwendet. Die einzelnen UI-Elemente verwenden die public Methoden der Fraktal-Skripte. Die Methoden verändern die Werte, die der Compute Shader verwendet und erlauben dadurch ein Bearbeiten der Bilder. Bei der Julia Menge wurde ein Slider zum Verändern der komplexen Zahl verwendet, um den Bereich von -2 bis 2 zu begrenzen. Die public Methoden für Slider benötigen einen Parameter vom Typ float, um den wert der Sliders an die Methode zu übergeben. Dadurch kann die Komplexe Zahl eingestellt werden, in dem der Nutzer die reele und imaginäre Zahl beliebig bearbeitet.
Im folgenden Code Beispiel sieht man die Methode zur Veränderung der imaginären Zahl im Slider. Der Parameter der übergeben wird ist dabei die imaginäre Zahl, die als Float vom Slider übergeben wird. Zusätzlich wird der Text über dem Slider geändert als “Imaginär = (der Parameter als zweistellige Gleitkommazahl)”. Mit der UpdateTexture Funktion wird der veränderte Wert dann an den Compute Shader übergeben. Das Ergebnis ist in Abbildung 1 zu sehen.

```
public void NewImag(float imag)
    {

        imag2 = imag;
        textMeshImag.text = "Imaginär = " + (Mathf.Round(imag * 100) *
0.01) ;
        UpdateTexture();
    }
```

Einblendung des User Interfaces

Der User soll nicht permanent von allen User Interfaces abgelenkt sein und jedes der Fraktale aus allen Position verändern können. Um dies zu verhindern wird ein Skript benötigt, welches überprüft, ob der Nutzer in der Nähe ist oder nicht. Dafür bekommt der Player einen Collider und die einzelnen Fraktale auch. Der Bereich des Fraktal Colliders beinhaltet den Bereich, in dem der Nutzer das UI bedienen kann. Dem Fraktal wird ein Skript hinzugefügt mit dem Namen *CheckCollider*. Ist der Player in dem begrenzten Bereich wird mit der Methode *OnTriggerEnter* das UI aktiviert, beim Verlassen wird es mit der *OnTriggerExit* Methode wieder geschlossen.

```
    private void OnTriggerEnter(Collider other)
    {
        if (other == player)
        {
            canvas.SetActive(true);
        }
    }
```

Shader-Programm

Für jedes Fraktal ist ein Shader-Programm (Shader) vorgesehen, welcher den Code beinhaltet, um es zu zeichnen. Das Ziel des Shaders ist immer, mithilfe der gegebenen Buffer-Attribute die Farbe des momentanen Pixels zu bestimmen. So kann der Shader dann parallel für jedes Pixel ausgeführt werden, was zu dem Geschwindigkeitsvorteil gegenüber der CPU Methode aus Abschnitt *Probleme* wird. Dazu stehen in der Grafikkarte tausende Rechenkerne zur Verfügung.

Die Algorithmen, die das Fraktal erzeugen, müssen dies also für jedes Pixel können. Um für ein Fraktal den richtigen Algorithmus zu finden, kann auf die Webseite Shadertoy zurückgegriffen werden. Dort werden Shader für verschiedenste Zwecke mit Shader-Code veröffentlicht. Diese Beispiel-Shader mussten dann aber noch aus einem OpenGL “GL SL-Shander” in einen DirectX “HL SL-Shader” umgeschrieben werden.

Codebeispiel HLSL-Shader

Die verwendeten Compute-Shader sind in HLSL geschrieben. Sie sind in der Funktion einem OpenGL “Fragment-Shader” gleich. Deren Funktion ist immer, eine Farbe für die derzeitigen Pixel auf die von DirectX vorgesehene Variable *Result[id.xy]* zu speichern. Dabei gibt es einen Buffer, der mit vorbestimmten Attributen gefüllt ist. Diese wurden zuvor zum Start der Anwendung gefüllt und im Laufe der Anwendung immer wieder aktualisiert. Zudem ist die Position auf der Zielfläche des Shaders mit der UV-Map angegeben. In unserem Fallbeispiel beinhaltet sie die x und y Koordinaten, auf denen das Fraktal erzeugt wird. Das Fraktal sollte dabei auch nur auf eine Fläche gerendert werden, nicht aber wie beispielsweise bei einem Diffuse-Shader ganze Objekte anzeigen.

Shader ausführen

Jeder Shader wird über ein C# Skript eingeleitet. Das Skript updatet die Bufferattribute immer dann, wenn der Nutzer neue Eingaben tätigt. In der Methode *UpdateTexture* wird der Shader beauftragt, die Textur zu rendern. Es folgt der Beispielcode der Mandelbrot C# Klasse aus dem GitHub Repository von Coderious.

```
//Struct welches dem Inhalt des Attribut Buffers gleich sein muss
public struct DataStruct
{
    public double w, h, r, i;
    public int screenWidth, screenHeight;
}

DataStruct[] data = new DataStruct[1];

// Wird zu Beginn aufgerufen
```

```
void Start()
{
    height = width * renderHeight / renderWidth;

    data[0] = new DataStruct
    {
        w = width,
        h = height,
        r = rStart,
        i = iStart,
        screenWidth = renderWidth,
        screenHeight = renderHeight
    };

    buffer = new ComputeBuffer(1, 40);

    renderTexture = new RenderTexture(renderWidth, renderHeight, 0);
    renderTexture.enableRandomWrite = true;
    renderTexture.Create();
    GetComponent<Renderer>().material.mainTexture = renderTexture;

    UpdateTexture();
}

// Hier bekommt der Shader den Buffer und wird ausgeführt
void UpdateTexture()
{
    int kernelHandle = shader.FindKernel("CSMain");

    buffer.SetData(data);
    shader.SetBuffer(kernelHandle, "buffer", buffer);

    shader.SetInt("iterations", maxIteration);
    shader.SetTexture(kernelHandle, "Result", renderTexture);

    shader.Dispatch(kernelHandle, renderWidth / 24,
            renderHeight /    24, 1);

    RenderTexture.active = renderTexture;
}
```

Shader-Code

Es folgt ein Shader-Beispiel aus dem GitHub Repository von Coderious. Wobei im Data Struct w für width, h für height, r für den render Start und i für den itertations Start steht.

```
// CSMain als Kernel festlegen
#pragma kernel CSMain

// Data ist die Schablone, welche die Werte im Buffer bestimmt. Nur ein
// Buffer mit exakt diesen werten wird angenommen.
struct data
{

    double w, h, r, i;
    int screenWidth, screenHeight;
};

// Diese Variable soll die Farbe als Ergebnis bekommen
RWTexture2D<float4> Result;
// Der Buffer mit allen Attributen für das derzeitige Pixel
StructuredBuffer<data> buffer;

uint iterations;

[numthreads(24, 24, 1)]
void CSMain(uint3 id : SV_DispatchThreadID)
{
    double real, imag;
    double real2, imag2;
    double xAdjust = (double)id.x / buffer[0].screenWidth * buffer[0].w;
    double yAdjust = (double)id.y / buffer[0].screenHeight *
buffer[0].h;
    uint ite = 0;

    float4 CalcColor = { 0.0f , 0.0f, 0.0f, 1.0f };

    real = buffer[0].r + xAdjust;
    imag = buffer[0].i + yAdjust;

    //Cardioid-bulb checking
    double q = (real - 0.25) * (real - 0.25) + imag * imag;
    if (q * (q + (real - 0.25)) < 0.25 * (imag * imag))
    {
        Result[id.xy] = CalcColor;
        return;
    }
```

```
    // Iteriert, bis herausgefunden wurde, ob der Bildpunkt in der
    // Mandelbrot-Menge liegt.
    for (uint i = 0; i < iterations; i++)
    {
        real2 = real * real;
        imag2 = imag * imag;

        if (real2 + imag2 > 4.0)
        {
            break;
        }
        else {
            imag = (2.0 * real * imag + buffer[0].i + yAdjust);
            real = (real2 - imag2 + buffer[0].r + xAdjust);
            ite++;
        }
    }

    if (ite != iterations)
    {
        int colorNr = ite % 16;

        switch (colorNr)
        {
        case 0:
        {
            CalcColor[0] = 66.0f / 255.0f;
            CalcColor[1] = 30.0f / 255.0f;
            CalcColor[2] = 15.0f / 255.0f;

            break;
        }
        // Weitere 12 Cases welche Farbe bestimmen
        // [...]
        }
    }

    // Ergebnisfarbe des Shaders
    Result[id.xy] = CalcColor;

    // Farbe wurde bestimmt, der Shader hat seine Aufgabe erfüllt
}
```

Verwendung der Prefabs

Zum Verwenden der einzelnen Fraktale in anderen Projekt wird nur der Ordner mit dem Namen Fraktale benötigt. In diesem Ordner befindet sich ein Canvas Prefab, das *CheckPlayer* Skript und die einzelnen Fraktale mit Shadern und Skripten. Die einzelnen Fraktale sind in Unterordner mit Prefab und passenden Skripten eingeordnet. Dadurch kann jedes einzelne der Fraktale direkt verwenden. Die Prefabs können direkt in die Szene gezogen werden und sind Game Ready. Damit das *CheckPlayer* Skript funktioniert brauch der XR Player Rig einen Collider und den Tag Player.
Die Modelle befinden sich in dem Ordner mit dem Namen FBX und können auch direkt in andere Projekte kopiert werden. Um die verwendeten Texturen und Materials zu benutzen, muss zusätzlich der Materialordner hinzugefügt werden.

Fazit

Das Projekt wurde mit den Fraktalen Mandelbrot, Julia-Menge, Sierpinski und Kochflocke umgesetzt (Siehe Abbildung 1, 2). Alle Fraktale konnten mit UI-Knöpfen Plus und Minus vergrößert oder verkleinert werden und über ein Steuerkreuz die Anzeige bewegen. Das kann in Abbildung 2 auf der rechten Seite erkannt werden. Für das Julia Fraktal ist es möglich, die Eingabe aus der Mandelbrotmenge mit zwei Schiebern zu bestimmen. Bei der Kochflocke ist es möglich, die Anzahl der Iterationen zu bestimmen. Das hat Einfluss auf die Genauigkeit und die Details, welche erzeugt werden.

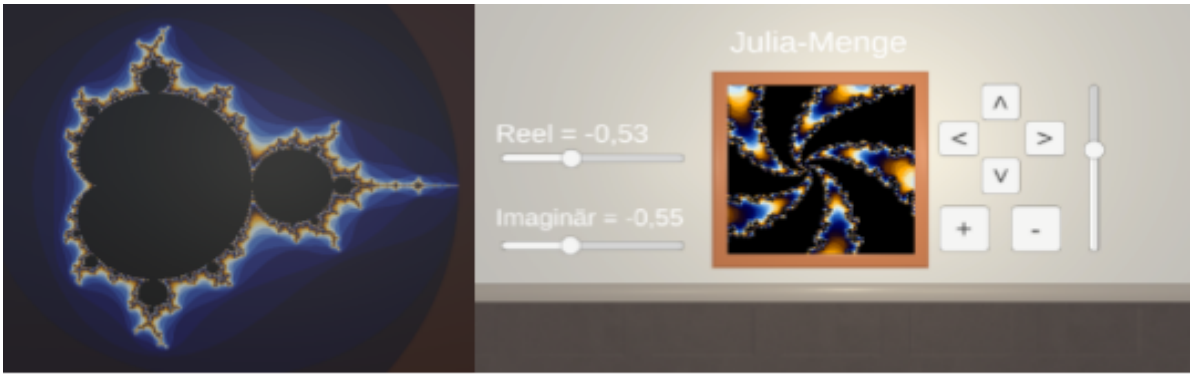

Abbildung 1: Mandelbrot und Julia Menge

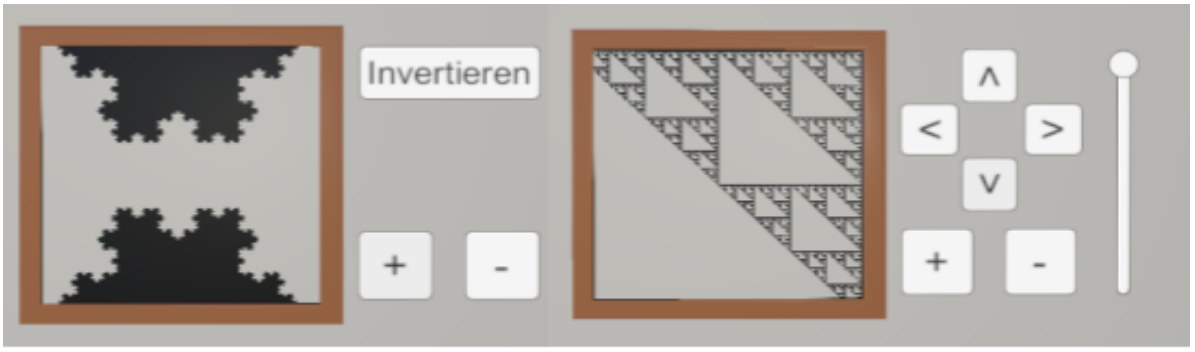

Abbildung 2: Alternative Kochflocke und Sierpinsky

Ausblick

In der Abgabe und während der Entwicklung sind einige Ideen gekommen, was an der Anwendung hinzugefügt oder verbessert werden könnte. Es folgt eine Auflistung aller Ideen:

- Mehrere Räume: Ein Raum für jedes Fraktal
- Fraktale Speichern: Bild des derzeitigen Fraktales auf dem Rechner abspeichern
- Räume füllen mit Fraktal-Screenshots von Nutzern
- 3D Fraktale: Eine dreidimensionale Darstellung

Quellenangaben

Nützliche Hyperlinks und verwendete Assets

Hier sind einige nützliche Webseiten gelistet, die interessant werden können, sollte jemand in dem Gebiet eintauchen wollen.

https://www.shadertoy.com/ Online Code-Editor für GL SL-Shader. Große Bibliothek für Shader-Code.

https://github.com/Coderious-GitHub/MandelbrotCPU Beispiel mit Mandelbrot-Shader in Unity Engine

https://polyhaven.com/textures CC0 lizenzierte Texturen für die verwendeten Materialien

https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022 Controller für Oculus Brille