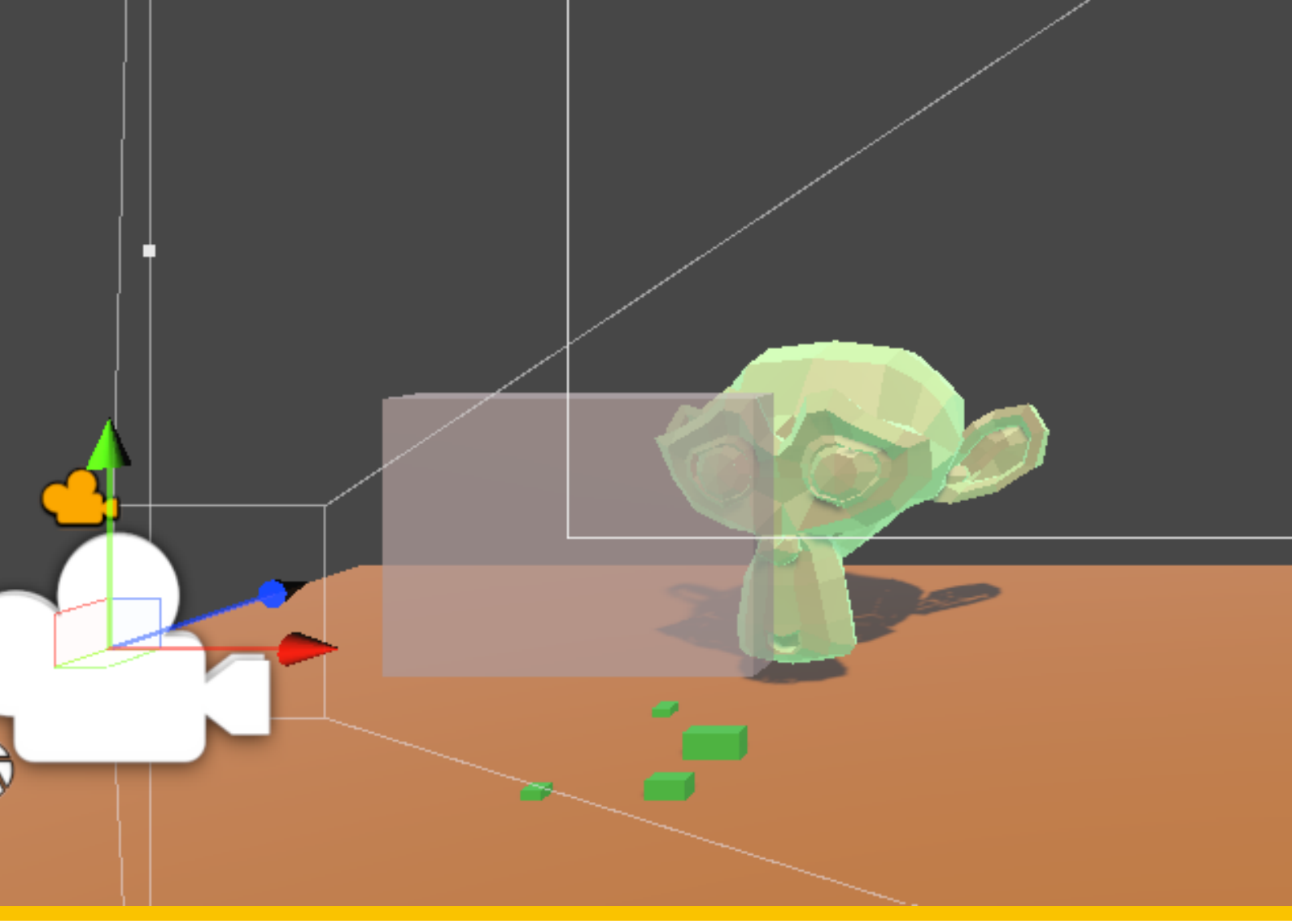

# Projekt 2

## AR / POSE Estimation


| | |
|---|---|
| Course of study | Bachelor of Science in Computer Science |
| Author | Bernhard Messerli |
| Advisor | Marcus Hudritsch |
| Expert | Some expert |


Version 1.0 of December 19, 2023

- Technik und Informatik
- Informatik

# Abstract

In meiner Projektarbeit möchte ich einen tieferen Einblick erhalten wie Augmented Reality an einem eigentlich einfachen Anwendungsfall umgesetzt werden kann.
Stellen Sie sich folgende Szene vor: In einem Raum hängt ein grösserer Bildschirm. Eine Tiefenkamera, die gerade oberhalb dem Bildschirm montiert ist, trackt den Betrachter, sobald sich dieser innnerhalb dem getrackten Bereich befindet. Die Tiefenkamera ist vom Typ Intel Realsense Kamera D435.
In einer virutellen Szene, welche mittels der Gameengine Unity dargestellt wird, wird die Position von der getrackten Person auf eine virutelle Kamera gemappt. Der Blick der virtuellen Kamera entspricht dem Blick des Betrachters. In dieser Szene wird nun ein virtuelles Fenster dargestellt. Dieses Fenster wird perspektivisch in 2D dargestellt und erscheint als verzerrtes Rechteck - je nach Betrachtungswinkel. Nun soll dieses Polygon entzerrt und dann auf dem Bildschirm vor dem Betrachter dargestellt werden.
Dem Betrachter erscheint so auf dem Bildschirm ein virtuelles Fenster, deren sichtbare Objekte sich mit der Position des Betrachters perspektivisch verschieben. Der Betrachter, schaut in den Bildschirm und erlebt eine virtuelle Szene, welche sich so verhält, wie wenn er durch ein reales Fenster in die reale Welt blickt.
Dieses Empfinden des Betrachters versuche ich mit dieser Arbeit zu erreichen.

# Contents

# 1. Erstes Kapitel

## 1.1. Einführung

Bilder faszinieren mich schon sehr lange. Anfangs waren es Gemälde, Zeichnungen, Skizzen, analoge Fotografien, später dann digitale Bilder und bewegte Bilder.
Ich bin nun in der Vertiefung Computer Perception and Virtual Reality an der BFH Bern. Im Gespräch mit dem Dozenten Marcus Hudritsch kamen wir auf die Thematik von Raum- und Farbempfinden im Zusammenhang mit Augmented Reality. Es ging darum ein Projektthema zu finden in dieser Vertiefung der Informatikausbildung.
Marcus Hudritsch hat mir dann von dieser Umsetzung erzählt: Ein Bildschirm welcher ein künstliches Fenster in eine virtuelle Welt suggeriert. Ich stelle mir vor, der Betrachter steht etwas verwirrt vor diesem Screen und begreift erst allmählich. Zuerst begreift er, dass sich diese Szene auf dem Screen bewegt, dann nach einigem hin- und hergehen, dass sich die Szene mit ihm im Gleichschritt bewegt und dann nach genauem Beobachten, wie sich die virtuellen Objekte verändern. Dass die perspektivische Darstellung stimmt und es ist, als schaue er durch ein Fenster in eine künstliche Welt von Objekten.
Details werde ich in dieser Arbeit später aufzeigen. Wichtig ist mir hier deutlich zu machen, dass es in dieser Umsetzung auch darum geht, den Betrachter von der Realität in die Virtualität hinein zu begeleiten und etwas ins Grübeln zu bringen.
Dies wird nur gelingen wenn die Illusion von einem Fenster unserem gewohnten Empfinden gerecht wird und es sich als Realtime Simulation anfühlt.

## 1.2. Komponenten

### 1.2.1. Physische Komponenten

#### Intel RealSense D435

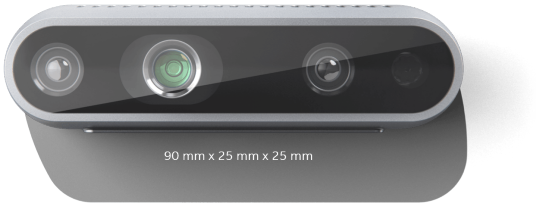


**Figure 1.1.:** Intel RealSense D435, Frontside

Die Kamera IntelRealSense D435 von Intel hat ein breites Sichtfeld, einen globalen Shutter und einen Tiefensensor, der sich für Anwendungen mit schnellen Bewegungen eignet. Gerade durch ihr breites Sichtfeld eignet sich die Kamera für Anwendungen in der Robotik, Virtual und Augmented Reality, bei Szenen wo es darum geht möglichst viel zu sehen. Sie hat eine Reichweite bis zu 10 Meter. Das Intel RealSenseSDK bietet plattformunabhängig Unterstüzung bei der Umsetzung. Für mein Projekt sollte diese Kamera also genügen. Wie gross das Sichtfeld dann wirklich ist, wird sich zeigen.

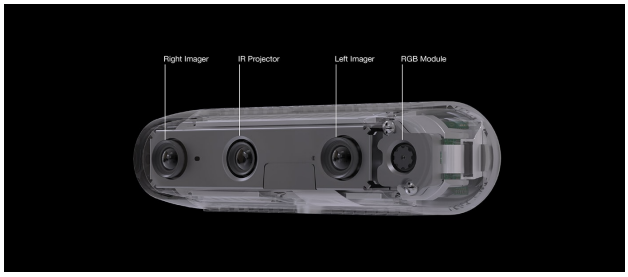


**Figure 1.2.:** Intel RealSense D435, Backside

Die Global-Shutter-Sensoren haben eine hohe Lichtempfindlichkeit bei schlechten Lichtverhältnissen und ermöglichen die Steuerung von Robotern auch in dunklen Räumen. [1]
Die Lichtverhältnisse in einem Raum der künstlich beleuchtet werden kann, spielen in meinem Projekt eine untergeordnete Rolle. Wichtig ist eigentlich nur, dass diese Tiefenkamera die getrackten Positionen richtig erfasst und möglichst in Realtime updatet.

### 1.2.2. Software Komponenten

#### Intel RealSense Viewer

Der Intel RealSense Viewer hat zwei Frames die er anzeigen kann, den RGB Frame und den Depth Frame. Die Abbildung zeigt einen Screenshot vom Depth Frame.

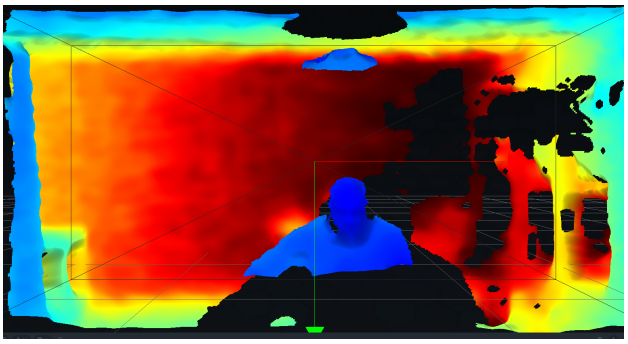

**Figure 1.3.:** Intel RealSense Viewer, Depth Frame

Die blauen Farbpixel zeigen einen nahen Bildpunkt, die roten Farbpixel einen entfernten Bildpunkt.
Für mein Projekt eignent sich dieser Viewer einzig dazu die verschiedenen Kameraeinstellungen auszuprobieren.

#### Intel RealSenseSDK 2.0

Intel hatte vor einigen Jahren eine führende Rolle in Anwendungen im Bereich Augmented Reality. Mit dem Intel RealSenseSDK 2016, dem Vorgänger vom RealSenseSDK 2.0 gab es ein ToolKit, welches Facetracking implementiert hatte. Die rechte und die linke Hand konnten beispielsweise seperat fürs Tracking angesteuert werden und mehr noch: Sogar einzelne Finger einer Hand konnten getrackt werden.
Ich habe noch versucht dieses ältere SDK zu installieren und dann mit einer älteren Unity Version zu verwenden. Leider ohne Erfolg. Intel hatte eine ganze Videoreihe dazu erstellt, welche aber heute vom Netz genommen wurde.
Intel schließt seine Abteilung rund um die RealSense-Kameras, die viele Jahre interessante, aber vom Markt kaum umgesetzte Lösungen hervorgebracht hatte. Sie passt nicht zum neuen Geschäftsmodell, welches rund um die Kernthemen von Intel aufgebaut und von dem neuen Foundry-Geschäft unterstützt wird. [2]
Das neue SDK 2.0 taugt dagegen nur noch wenig und eignet sich nicht für mein Projekt. Das Toolkit implementiert kein Facedetection mehr.
Eigentlich sehr Schade, das aufgebaute Know-How still zu legen.

### Unity

Unity ist eine Game Engine die von einer Open Source Community betreut und weiterentwickelt wird. [3]
Unity bietet für dieses Projekt ein anwendungsfreundliches Toll mit welchem die virtuelle Szene dargestellt werden kann. In Unity können viele Plugins installiert werden, die mir dann auch sehr nützlich sein werden. Ein weiterer Vorteil von Unity ist die grosse Community und die vielen Beiträge wie Probleme gelöst werden können.

### Nuitrack

Nuitrack ist eine Tracking Software, die viel mehr kann, als ich für dieses Projekt benötige.
Nuitrack stellt in ihrem SDK viele Anwendungen zur Verfügung, diese sind aber auf 3 Minuten Spieldauer limitiert. Wahrscheinlich haben sie von Intel gelernt und haben ein Pricing welches für volle Anwendungen $99.99 pro Jahr kostet. In der nächsten Abbildung sind alle implementierten Features dargestellt.

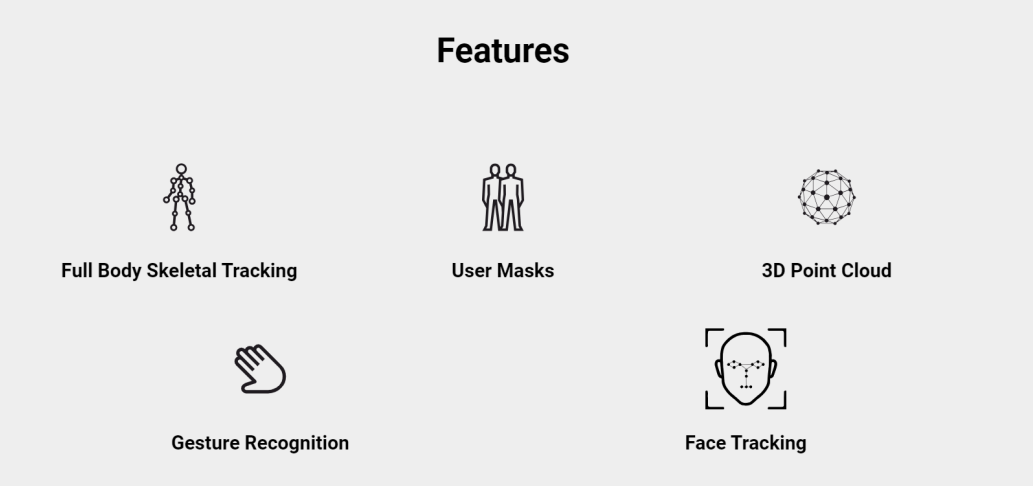


**Figure 1.4.:** Nutitrack SDK, Features

Dieses SDK ist enorm mächtig und hat alles zu bieten, was ich für mein Projekt brauche. Wichtig für mein Projekt ist das Facetracking und das Full Body Skeletal Tracking. Die Nutirack SDK hat eine gute Anbindung zu Unity und ist gut dokumentiert. [4]

### 1.2.3. OpenCV

OpenCV ist eine Open Source Computer Vision Grafik Library. In meinem Projekt benötige ich diese Library um ein bliebiges Polygon mit- vier Seiten zu entzerren und als Rechteck darzustellen. Es gibt einen Wrapper für diese Library, welcher sich in Unity einbinden lässt. Dieser ist kostenpflichtig.
Herr Hudritsch konnte mir im Rahmen der Ausbildung eine Free Version zur Verfügung stellen. Vielen Dank an dieser Stelle für die Software.
Informationen zu OpenCV finden sich unter: https://opencv.org/ und für das Unity Plugin unter: https://enoxsoftware.com/opencvforunity/.

### 1.2.4. Homographie

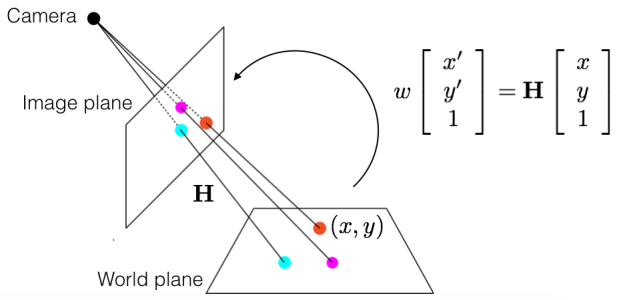


**Figure 1.5.:** Homographie, Visualisation

Die Homographie Matrix H beschreibt wie die Originalpunkte x,y zu liegen kommen in der Bildebene. Die Homographie Matrix benötigt jeweils 4 Punkte in der Quellebene und die entsprechenden 4 Punkte in der Zielebene.

$$\begin{bmatrix} \hat{x}_i z_a \\ \hat{y}_i z_a \\ z_a \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

**Figure 1.6.:** Homographie, Abbildungsgleichung

Wird diese Homographie Transformation in 2D gemacht, dann reicht eine 3x3 Matrix wie in Figure 1.6 dargestellt.
In meinem Projekt werden Textures in 2D transformiert, daher reicht eigentlich

eine 3x3 Matrix.
Die 8 Punkte werden nun als Vektoren in einer Matrix A dargestellt. Die Matrixmultiplikation von A mit der Homographie Matrix die wir suchen, wird in einem Gleichungssystem von 8 Gleichungen und 8 Unbekannten gelöst.

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1\hat{x_1} & -y_1\hat{x_1} \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2\hat{x_2} & -y_2\hat{x_2} \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3\hat{x_3} & -y_3\hat{x_3} \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4\hat{x_4} & -y_4\hat{x_4} \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1\hat{y_1} & -y_1\hat{y_1} \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2\hat{y_2} & -y_2\hat{y_2} \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3\hat{y_3} & -y_3\hat{y_3} \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4\hat{y_4} & -y_4\hat{y_4} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = h_{33} \begin{bmatrix} \hat{x_1} \\ \hat{x_2} \\ \hat{x_3} \\ \hat{x_4} \\ \hat{y_1} \\ \hat{y_2} \\ \hat{y_3} \\ \hat{y_4} \end{bmatrix}$$

**Figure 1.7.:** Homographie, Gleichungssystem

Der Streckungsfaktor h33 zeigt, dass die Homographie Matrix noch skaliert werden kann. Im Default Fall wird h33 eins gesetzt. [5] [6]

### 1.2.5. Kameramodell

Um auf die Eckpunkte vom Screen zu kommen, sollte es reichen, wenn wir die Position der Kamera, deren Rotationswinkel, sowie den Winkel vom Field of View und den Viewport der Kamera kennen. Weiter kennen wir die World Positions von den Eckpunkten vom Screen. Da der Screen statisch ist und sich nicht bewegt, können wir dieses Model ausser acht lassen.
also Was bleibt sind: der Viewport (1), die Projektion (2) (Kamera intrinsisch) und das Kamera-Modell (3) (Kamera extrinsisch oder View). Unsere Abbildungsmatrix fürs Kameramodell sieht dann so aus: Matrix m = (1) * (2) * (3) Und mittels Anwendung auf die Eckpunkte vom Screen: P1' = P1 * m.

# 2. Zweites Kapitel

## 2.1. Ergebnisse

### 2.1.1. Einbinden von Nuitrack

Nuitrack kann unter folgendem Link für alle Platformen runtergeladen werden: **Nuitrack**.
Die Software ist Platform unabhängig verfügbar und wird im gewünschten Folder auf der Festplatte installiert.

#### Import Nuitrack Wrapper in Unity

Unter folgendem Git Repo kann das Nuitrack Plugin runtergeladen werden: **Nuitrack Unity Plugin**. Da dieses Plugin für verschiede Sensoren, also Kameras konfiguriert wurde, muss dann noch der Sensor spezifiziert werden. In meinem Projekt ist dies das Plugin für die IntelRealsense D435 Kamera.

#### Skeleton Tracking mit NuitrackSDK

Das Tutorial von Nuitrack ist meine Ausgangslage um eine Person zu tracken. Der Link führt zu diesem Tutorial: **Nuitrack Facetracking**.
Das Tutorial ist gut beschrieben und lässt sich genau so umsetzen wie beschrieben.

### 2.1.2. Virtuelle Szene in Unity

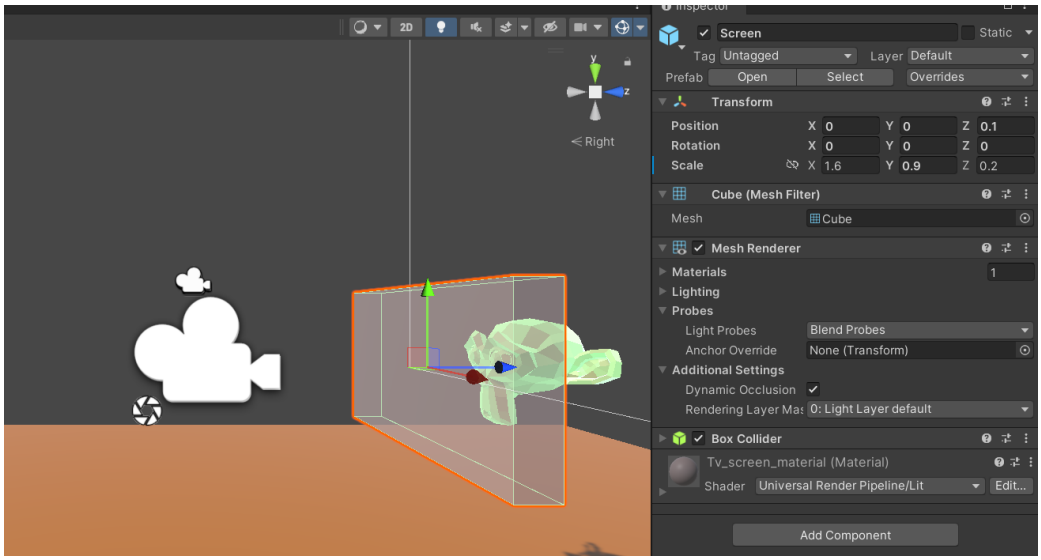


**Figure 2.1.:** Unity, Szene

In Unity baue ich die Szene so auf, dass die Render-Kamera in positiver z-Achse ausgerichtet ist. Anfangsposition der Render-Kamera ist (0, 0.2, -1.5). Den Screen welcher dann entzerrt werden soll, setze ich auf den Nullpunkt. Die Mitte vom Screen hat die Position (0, 0, 0). Die Eckpunkte vom Screen haben die folgenden Positionen: P0(-0.8, -0.45) P1( 0.8, -0.45) P2( 0.8, 0.45) P3(-0.8, 0.45) Der Screen hat die Grösse 1600 x 900 mm. Dieser wird als schmaler Kubus dargestellt und transparent leicht grau eingefärbt, damit ich dann erkennen kann ob mein Rendern und Entzerren richtig funktioniert. Die Main-Kamera setze ich als Referenz auf die Position der Render-Kamera. Dies damit ich eine Ansicht in der Szene bekomme. Die Main-Kamera zeigt dann die Szene und in der Game Ansicht zeigt die Render-Kamera den Output vom entzerrten Screen. Gut möglich, dass man dies auch mit weniger Kameras darstellen kann.
Figure 2.1 zeigt den Aufbau der Szene mit dem grau eingefärbtem Screen und dessen Werte im Inspektor Fenster rechts daneben.

### 2.1.3. Render Kamera zeigt Position vom Betrachter

Die Render-Kamera wird als Referenz auf den obersten Knochen vom Skelett gemappt. Dieser oberste Knochen entspricht dem Kopf, genauer der Stirn. Damit wird dann auch die Main-Kamera, welche die Render-Kamera referenziert auf den Kopf gemappt.

### 2.1.4. Render Kamera zeigt RenderTexture

Die Render-Kamera wird nun stets auf den Mittelpunkt vom Screen gerichtet. Der Output der Render-Kamera erzeugt dann eine RenderTexture der Grösse 750 x 750 Pixel. Je grösser die Anzahl Pixel, desto besser wird dann die Qualität vom Endbild.

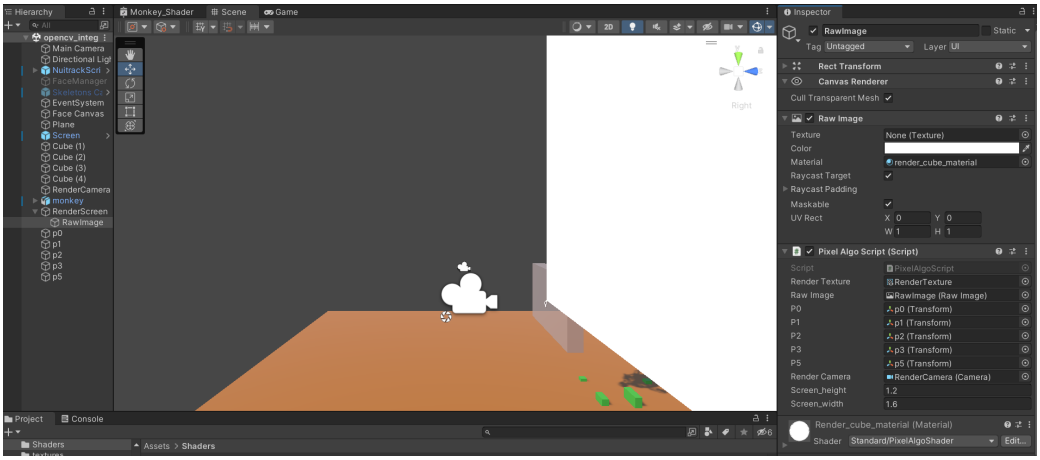


**Figure 2.2.:** Unity, RawImage

Figure **??** zeigt das Output Bild, welches auf der weissen Ebene dargestellt wird. Diese entspricht nun dem Output Bild für den Screen und zeigt ein Format 16 : 9.

### 2.1.5. Matrix Shader

Im Netz habe ich einen Rotations Shader gefunden und versucht diesen so anzupassen, dass die Entzerrung gelingt. Leider ohne Erfolg. Das Probelm bei diesem Shader: Zwar funktioniert das Scaling und das Shearing, leider nicht aber die Translation. Ich habe dann versucht eine Translation hinzuzufügen, was auch funktioniert hat.

```
// Use this shader on an object together with the above example script.
// The shader transforms texture coordinates with a matrix set from a script.
Shader "Standard/RotatingTexture"
{
    Properties
    {
        _MainTex("Base (RGB)", 2D) = "white" {}
        _Translation("Translation", Vector) = (0, 0, 0, 0)
    }
        SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 pos : SV_POSITION;
            };


            float4x4 _TextureRotation;
            float4 _Translation;

            v2f vert(float4 pos : POSITION, float2 uv : TEXCOORD0)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(pos);
                o.uv = mul(_TextureRotation, float4(uv,0,1)).xy;
                // Apply translation in x and y direction
                o.uv += _Translation.xy;
                return o;
            }

            sampler2D _MainTex;
            fixed4 frag(v2f i) : SV_Target
            {
                return tex2D(_MainTex, i.uv);
            }
            ENDCG
        }
    }
}
```

Figure 2.3.: Unity, MatrixShader

Figure 2.3 zeigt den umgebauten Matrix-Shader mit einer Möglichkeit die x und y Koordinaten der Pixel zu verändern. Leider weiss ich aber nicht, wie ich mit dieser Rotationsmatrix das Polygon selektieren soll und dann entzerrt darstellen. Als Einstieg in die Unity Shader hat es trotzdem etwas geholfen, mehr aber auch nicht.

### 2.1.6. Pixel Shader

Damit ich diese Entzerrung vom Polygon machen kann, benötige ich einen Pixel-Shader. Oder einen Shader welcher die Maintexture ändert. Ich erstelle nun so Schritt für Schritt einen Shader welcher als Parameter die Maintextur als Eingabewert hat. Und diese Maintexture dann im Shader jeweils angepasst wird.
Im Script wird in der Update Funktion für jedes Pixel der Farbwert gesetzt und am Ende die Textur dem Shader übergeben.

```
Shader "Standard/PixelAlgoShader"
    {
    Properties
    {
        _MainTex("Base (RGB)", 2D) = "white" {}

    }

            SubShader
            {
                Pass
                {
                    CGPROGRAM
                    #pragma vertex vert
                    #pragma fragment frag

                    struct v2f
                    {
                        float2 uv : TEXCOORD0;
                        float4 pos : SV_POSITION;
                    };


                    float4x4 _TextureRotation;


                    v2f vert(float4 pos : POSITION, float2 uv : TEXCOORD0)
                    {
                        v2f o;
                        o.pos = UnityObjectToClipPos(pos);
                        o.uv = mul(_TextureRotation, float4(uv,0,1)).xy;

                        return o;
                    }

                    sampler2D _MainTex;
                    fixed4 frag(v2f i) : SV_Target
                    {
                        return tex2D(_MainTex, i.uv);
                    }
                    ENDCG
                }
            }
    }
```

Figure 2.4.: Unity, Pixel-Shader

Figure 2.4 zeigt den Shader, welcher die Texturen verändert, indem die Pixelwerte der Maintexture verändert werden.

### 2.1.7. Homographie

Unter dem folgenden Link habe ich ein Git Repo gefunden, welches die Homographie Matrix mit Gauss Elimination durchführt: **Homographie Matrix Berechnung**. Zuerst habe ich versucht die Entzerrung mit einer Homographie Matrix zu erreichen. Leider hat dies nur geklappt für die Ansicht ohne seitliche Verschiebung, also mit einem x-Wert von Null. Sobald ich etwas abweiche von dieser zentralen Position, zeigt sich ein Shearing im Output Bild, welches sich auch noch anders verhält, je nachdem ich in positiver x-Achse gehe oder in negativer

x-Achse. Ich habe da länger darüber nachgedacht, meine Homographie Matrix mehrmals geprüft, fand aber nicht heraus was das Problem war. Komisch ja auch, dass es in zentraler Position den richtigen Output gab, bei seitlicher Verschiebung sich dann aber ein Shearing zeigt, welches mit grösserer Abweichung nach links oder rechts dann auch noch grösser wurde. Somit brauche ich nun einen anderen Ansatz um die Entzerrung zu lösen.

### 2.1.8. OpenCV

OpenCV hat bereits Methoden, welche diese Entzerrung vornehmen. Das Plugin von OpenCV findet sich unter: **OpenCV Unity Plugin**. Die grösste Schwierigkeit ist bei OpenCV, dass Unity einen anderen Aufbau der Matrizen hat und dass OpenCV keine Textures kennt. Die grösste Schwierigkeit war dann zu merken, dass die Pixelwerte der Textures in OpenCV nicht einfach so auf der CPU verfügbar sind. Diese werden auf der GPU gespeichert und mössen dann exklusiv abgefragt werden. Die Funktionen von OpenCV funktionieren tadellos. Damit die Entzerrung gelingt, gehe ich vom Zielbild aus und berechne die Perspektivische Matrix rückwärts vom Zielbild zum Inputbild, was dann dem Inversen der Perspektivischen Matrix entspricht. Nun wird diese Matrix auf jedes Pixel des Zielbildes angewendet und so den Farbwert an der entsprechenden Stelle im Inputbild geholt. Da diese Operation mit der Anzahl Pixel steigt, habe ich hier ein Format vom Zielbild 800 x 450 Pixel gewählt.

```
0 Unity Message | 0 references
void Update()
{
    //adjustFov();
    calculateEdgeScreenPoints();
    //doCalculateTransformVertices();


    rendTex = toTexture2D(renderTexture);


    Mat inputMat = new Mat(rendTex.height, rendTex.width, CvType.CV_8UC4);
    Mat outputMat = new Mat(texture.height, texture.width, CvType.CV_8UC4);

    OpenCVForUnity.UnityUtils.Utils.texture2DToMat(rendTex, inputMat, true, 0);


    Mat src_mat = new Mat(4, 1, CvType.CV_32FC2);
    Mat dst_mat = new Mat(4, 1, CvType.CV_32FC2);
    src_mat.put(0, 0, srcPolygon[0].x, srcPolygon[0].y, srcPolygon[1].x, srcPolygon[1].y, srcPolygon[3].x, srcPolygon[3].y, srcPolygon[2].x, srcPolygon[2].y);
    dst_mat.put(0, 0, 0.0, 0.0, outputMat.cols(), 0.0, 0.0, outputMat.rows(), outputMat.cols(), outputMat.rows());

    Mat perspectiveTransform = Imgproc.getPerspectiveTransform(src_mat, dst_mat);
    Imgproc.warpPerspective(inputMat, outputMat, perspectiveTransform, new Size(outputMat.cols(), outputMat.rows()));


    Texture2D outputTexture = new Texture2D(outputMat.cols(), outputMat.rows(), TextureFormat.RGBA32, false);

    OpenCVForUnity.UnityUtils.Utils.matToTexture2D(outputMat, outputTexture);

    for (int y = 0; y < outputTexture.height; y++)
    {
        for (int x = 0; x < outputTexture.width; x++)
        {
            Color color = outputTexture.GetPixel(x, y);
            texture.SetPixel(x, y, color);
        }
    }


    texture.Apply();

    rawImage.material.SetTexture("_MainTex", texture);
}
```

**Figure 2.5.:** Unity, OpenCV

In Figure 2.5 zeigt sich diese Problematik. Die Textures werden in OpenCV Mat, also Matrizen umgewandelt. Die wichtigen Methoden von OpenCV sind dann:
Mat perspectiveTransform = Imgproc.getPerspectiveTransform(srcMat, dstMat);
In getPerspectiveTransform wird die Homographie-Matrix anhand der vier Quellpunkte und der vier Zielpunkte berechnet. Die andere wichtige Methode von OpenCV:
Imgproc.warpPerspective(inputMat, outputMat, perspectiveTransform, new Size(outputMat.cols(), outputMat.rows()));
In dieser warpPerspective Methode passiert die Entzerrung. In outputMat ist dann die gewünschte, entzerrte Textur, aber noch nicht als Textur. Deshalb wird dann diese OpenCV Mat in eine Textur umgewandelt.
Damit überhaupt etwas angezeigt wird braucht es nun noch diese explizite Abfrage der Pixelwerte, welches im Code in der For Schleife passiert.
Am Ende wird die neu generierte Textur dem Pixel-Shader übergeben. Dies passiert mit der letzten Zeile:
rawImage.material.SetTexture(" MainTex", texture);

### 2.1.9. Kameramodell

Ich habe versucht mittels Kameramodell die Transformationsmatrix für die Eckpunkte vom Screen zu berechnen. Dazu braucht man den Viewport, die intrinsiche (Projektionsmatrix) und die extrinsische Kameramatrix (Viewmatrix). Da sich der Screen nicht verschiebt, braucht es die Modellmatrix nicht.

```
0 references
void doCalculateTransformVertices()
{
    setViewport();
    Vector3[] vertices = getMatrixScreenVertices();
    Matrix4x4 view_x_projection_matrix = getViewMatrix_x_ProjectionMatrix();
    Matrix4x4 clip_to_viewport_matrix = new Matrix4x4();
    clip_to_viewport_matrix[0, 0] = 0.5f;
    clip_to_viewport_matrix[0, 1] = 0;
    clip_to_viewport_matrix[0, 2] = 0;
    clip_to_viewport_matrix[0, 3] = 0;
    clip_to_viewport_matrix[1, 0] = 0;
    clip_to_viewport_matrix[1, 1] = 0.5f;
    clip_to_viewport_matrix[1, 2] = 0;
    clip_to_viewport_matrix[1, 3] = 0;
    clip_to_viewport_matrix[2, 0] = 0;
    clip_to_viewport_matrix[2, 1] = 0;
    clip_to_viewport_matrix[2, 2] = 1;
    clip_to_viewport_matrix[2, 3] = 0;
    clip_to_viewport_matrix[3, 0] = 1;
    clip_to_viewport_matrix[3, 1] = 1;
    clip_to_viewport_matrix[3, 2] = 0;
    clip_to_viewport_matrix[3, 3] = 1;
    Matrix4x4 result =  view_x_projection_matrix;
    //Matrix4x4 result = view_x_projection_matrix * clip_to_viewport_matrix;
    Vector2[] screenPoints = new Vector2[4];
    for (int i = 0; i < vertices.Length; i++)
    {
        screenPoints[i].x = vertices[i].x * result[0, 0] + vertices[i].y * result[0, 1] + vertices[i].z * result[0, 2];
        screenPoints[i].y = vertices[i].x * result[1, 0] + vertices[i].y * result[1, 1] + vertices[i].z * result[1, 2];

    }
```

**Figure 2.6.:** Unity, Berechnung Kameramatrix

```
1 reference
Matrix4x4 getViewMatrix_x_ProjectionMatrix()
{
    return GL.GetGPUProjectionMatrix(renderCamera.projectionMatrix, false) * renderCamera.worldToCameraMatrix;
}

1 reference
void setViewport()
{
    GL.Viewport(new UnityEngine.Rect(0, 0, Screen.width, Screen.height));
}

1 reference
Vector3[] getMatrixScreenVertices()
{
    Vector3[] vertices = new Vector3[]
    {
        p0.position,
        p1.position,
        p2.position,
        p3.position
    };

    return vertices;
}
```

**Figure 2.7.:** Unity, Methoden Kameramatrix

In Figure 2.6 versuche ich die Kamera-Matrix zu berechnen, wie bereits im Kapitel Kameramodell beschrieben. In Figure 2.6 verwende ich die Unity Methoden: GetGPUProjectionMatrix, welche mir eine resultierende Matrix aus Viewmatrix * Projectionmatrix berechnet. Nun fehlt mir noch die Viewportmatrix. Ich habe versucht diese Matrix in einem ersten Schritt mit GL.Viewport festzulegen und dann noch etwas anzupassen mit der Matrix clip-to-viewport-matrix. Leider ohne Erfolg. Deshalb habe ich mich dann für einen etwas vereinfachten Ansatz entschieden.

### 2.1.10. Projektion der Eckpunkte auf der Zielebene

Mein Ansatz den ich umgesetzt habe funktioniert wie folgt: Unity hat eine Funktion welche mir die Eckpunkte von der Clipping Ebene entsprechend dem z Wert herausscheibt. Siehe Figure **??**

```
void calculateEdgeScreenPoints()
{

    Vector3 w0 = renderCamera.ViewportToWorldPoint(new Vector3(0, 0, -renderCamera.transform.position.z));
    Vector3 w1 = renderCamera.ViewportToWorldPoint(new Vector3(0, 1, -renderCamera.transform.position.z));
    Vector3 w2 = renderCamera.ViewportToWorldPoint(new Vector3(1, 1, -renderCamera.transform.position.z));
    Vector3 w3 = renderCamera.ViewportToWorldPoint(new Vector3(1, 0, -renderCamera.transform.position.z));
```

**Figure 2.8.:** Unity, Methode ViewportToWorldPoint

```
Vector3 v = w3 - w0;
Vector3 u = w1 - w0;



Vector3 n;

Vector3 qn = cloneDirVector(dir_screen_vec);
float dp = distance;
qn = qn.normalized;


n = qn;

Vector3 pr0 = -cam + r0;
Vector3 pr1 = -cam + r1;
Vector3 pr2 = -cam + r2;
Vector3 pr3 = -cam + r3;

//Debug.Log(" d1 - d3 is: " + d1 + " " + d2 + " " + d3);
//d is not for all points the same
//i want the vector cam + (-cam + r0) * d is point on the plane
//float d = q0.x * n.x + q0.y * n.y + q0.z * n.z;
float d = w0.x * n.x + w0.y * n.y + w0.z * n.z;
float t0 = (d - n.x * cam.x - n.y * cam.y - n.z * cam.z) / (pr0.x * n.x + pr0.y * n.y + pr0.z * n.z);
float t1 = (d - n.x * cam.x - n.y * cam.y - n.z * cam.z) / (pr1.x * n.x + pr1.y * n.y + pr1.z * n.z);
float t2 = (d - n.x * cam.x - n.y * cam.y - n.z * cam.z) / (pr2.x * n.x + pr2.y * n.y + pr2.z * n.z);
float t3 = (d - n.x * cam.x - n.y * cam.y - n.z * cam.z) / (pr3.x * n.x + pr3.y * n.y + pr3.z * n.z);
```

**Figure 2.9.:** Unity, Ebene

```
pr0 = pr0 * t0 + cam;
pr1 = pr1 * t1 + cam;
pr2 = pr2 * t2 + cam;
pr3 = pr3 * t3 + cam;
```

**Figure 2.10.:** Unity, Projektionsvektoren

Ich wähle diese Ebene durch den Nullpunkt, auf welchen sich die Render-Kamera immer richtet. Die Berechnung der Ebene ist in der Abbildung Figure 2.9 dargestellt. Nun projeziere ich die Eckpunkte vom Screen in der Verlängerung von der Render-Kamera durch die Eckpunkte. Die Berechnung dieser perspektivischen Vektoren in Richtung Eckpunkte vom Screen ist in der Abbildung Figure 2.10 dargestellt. Die Schnittpunkte mit der Ebene müssen nun nur noch relativ zu den Ecken der Clipping Ebene berechnet werden. Siehe untenstehende Graphik. Figure 2.11

Eigentlich hat auch der Richtungsvektor in y-Richtung eine x-Komponente, aber nur, wenn der Richtungsvektor in x-Richtung eine y-Komponente hat. Zur Vereinfachung nehme ich an, dass die Kamera horizontal ausgerichtet ist und dann wird dieser y-Anteil in x-Richtung gleich Null.

```
float y0 = (pr0.y - w0.y) / u.y;
float x0 = (pr0.x - w0.x - y0*u.x) / v.x;
//x0 = (pr0.x - q2.x -y0 * u.x) / v.x;
//y0 = (pr0.y - q2.y -x0 * v.x) / u.y;


float y1 = (pr1.y - w0.y) / u.y;
float x1 = (pr1.x - w0.x -y1*u.x) / v.x;
//x1 = (pr1.x - q2.x - y1 * u.x) / v.x;
//y1 = (pr1.y - q2.y - x1 * v.x) / u.y;


float y2 = (pr2.y - w0.y) / u.y;
float x2 = (pr2.x - w0.x -y2*u.x) / v.x;
//x2 = (pr2.x - q2.x - y2 * u.x) / v.x;
//y2 = (pr2.y - q2.y - x2 * v.x) / u.y;


float y3 = (pr3.y - w0.y) / u.y;
float x3 = (pr3.x - w0.x -y3*u.x) / v.x;
//x3 = (pr3.x - q2.x - y3 * u.x) / v.x;
//y3 = (pr3.y - q2.y - x3 * v.x) / u.y;
```

**Figure 2.11.:** Unity, Eckpunkte vom Screen

## 2.2. Analyse

Ziel dieser Arbeit war es, dem Betrachter ein virtuelles Fenster zu suggerieren. Dieses Fenster sollte sich also genauso verhalten wie ein reales Fenster.
Bei dieser Umsetzung zentral ist unser optisches Sehen, welches auf perpektivischem Sehen beruht. Dieses perspektivische Sehen galt es genau so umzusetzen. Dies wird in Unity von der Render Kamera gelöst, welche ein perspektivisches Bild ausgibt. Das ist in dieser Arbeit also sehr gut umgesetzt.
Der etwas schwierigere Teil war es, im Bildschirm nur einen Teil von diesem Bild darzustellen. Dazu dient der Screen in der virtuellen Szene um zu bestimmen, was genau auf dem Bildschirm dargestellt werden soll. Diesen Bildschirm genau zu erfassen erfordert ein genaues Wissen vom intrinsischen und extrinsischen Verhalten der virtuellen Render-Kamera und deren Position und Ausrichtung. Daraus lässt sich dann eine Matrix berechnen um im perspektivischen Bild der Render-Kamera die Eckpunkte vom Screen zu bestimmen.
Diesen Teil habe ich in meiner Arbeit nicht so umsetzen können. Ich habe das etwas vereinfacht. Mein Ansatz geht in die gleiche Richtung aber so, dass es sich in Unity möglichst einfach umsetzen lässt. In Unity mit der Methode ViewportToWorldPoint können die Eckpunkte vom geklippten Bild zurückgegeben werden. Ausgehend von diesen Punkten konstruiere ich diese Ebene und projeziere dann perspektivisch den Screen auf diese Ebene.
Dies funktioniert soweit gut, einzig wenn die Kamera nicht ganz horizontal ausgerichtet ist, bekomme ich in meiner geklippten Ebene für den horizontalen Richtungsvektor auch Werte für die y-Komponente die nun nicht mehr Null sind.
Bei der Bestimmung der relativen Anteile jedes Eckpunktes bezüglich der Richtungsvektoren bekomme ich zwei Gleichungen mit zwei Unbekannten. Damit das ganze etwas einfacher wird, setzte ich bei dem horizontalen Richtungsvektor die y-Komponente gleich Null. Im Endbild sieht man daher ganz aussen an den Rändern, dass die Eckpunkte vom Screen minim nicht ganz richtig gewählt wurden. Da mir nun aber die Zeit fehlt belasse ich dies so.
Insgesamt denke ich, dass der Eindruck vom virtuellen Fenster recht gut vermittelt wird. Die Auflösung ist in etwa auch so, dass ein Bildschirm in der Pixelgrösse 1200 x 900 ein akzeptables Bild zeigt. Die Anpassungsgeschwindigkeit an neue Positionen seitens Betrachter erscheint ohne grosse Hacker.
Insgesamt erzielt diese Arbeit ein gutes Resultat, so meine Einschätzung.

## 2.3. Fazit

Wie bereits in der Analyse zum Schluss festgehalten, denke ich, dass diese Umsetzung, mit dem Ziel dem Betrachter ein virtuelles Fenster zu suggerieren gut funktioniert.
Ich bin in dieses Projekt gestartet und wusste nicht genau, wo und viele Hürden es zu nehmen gab, um dann wirklich am Schluss einen akzeptablen Output zu

erhalten.
So war ich schon recht erleichtert als ich sah, dass das Face- und Skeletontracking von Nuitrack zusammen mit der Intel RealSense Kamera D435 funktionierte. Somit war der Einstieg geschafft und ich konnte das Projekt weiterentwickeln.
Eine nächste grössere Hürde war die Entzerrung vom gerenderten Screen erfolgreich machen zu können. Dabei habe ich verschiedene Ansätze getestet und war dann auch wieder erleichtert als die Umsetzung mit dem OpenCV Plugin funktionierte.
Eine letzte grössere Hürde war die Bestimmung der Eckpunkte vom gerenderten Screen. Auch hier funktionierte in der Unity Umsetzung schliesslich erst die zweite Variante.
Bei all diesen zeit- und denkintensiven Schritten habe ich viel gelernt und bin dankbar, am Schluss eine funktionierende Umsetzung vorweisen zu können.

## 2.4. Ausblick

In einer nächsten erweiterten Umsetzung, könnte ich mir vorstellen, dass der Eindruck von einem virtuellen Fenster noch gesteigert werden kann.
Dies indem die virtuelle Szene mit zwei Kameras, eine Kamera fürs linke Auge und eine Kamera fürs rechte Auge die virtuelle Szene filmt. Nun müssten die beiden Bilder in einer Updatemethode von Unity abwechselnd eingespielt werden. Da die update Frequenz deutlich grösser ist, als das menschliche Auge feststellen kann, könnte ich mir vorstellen so einen 3D Effekt der Szene für den Betrachter erzeugen zu können. Hier gibt es aber dann noch ein Problem: Die rechte und die linke Kamera erhalten nicht ganz die gleichen Eckpunkte für den gerenderten Screen. Möglicherweise helfen da Durschnittswerte beider Kameras. Es wäre also interessant zu sehen ob dieser Ansatz wirklich funktioniert.

# Declaration of Authorship

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those acknowledged.

All statements taken from other writings, either literally or in essence, have been marked as such.

I hereby agree that the present work may be reviewed in electronic form using appropriate software.

December 19, 2023

A. Muster

C. Example

# Bibliography

[1] Intel. Intel.

[2] Computer Base. Tiefenkameras & co: Intel gibt realsense zugunsten des kerngeschäfts auf.

[3] Unity. Unity.

[4] Nuitrack. Nuitrack sdk.

[5] Faisal Qureshi. Homography.

[6] Yalda Shankar. Homography estimation.

# List of Figures

# List of Tables

# Listings

# Glossary

**AP SoC** **All Programmable System-on-Chip (AP SoC)** was introduced by Xilinx. It represents a IC which comprise a hard-core processor core surrounded by an FPGA fabric. This type of ICs are highly configurable and provide algorithm partitioning capabilities. This provides high benefit for highly scale-able applications as well as fast time-to-market

**ARM** **ARM** A family of processor architectures. The hard processor type which forms the basis of the Zynq processing system is an ARM Cortex-A9 version. The term 'ARM' may also be used to refer to the developer of the processor, i.e. a company of the same name

**ASIC** **Application-Specific Integrated Circuit (ASIC)** An integrated circuit which is designed for a specific use, rather than general-purpose use

**BibTeX** Program for the creation of bibliographical references and directories in TEXor LATEX documents

**RTOS** **Real-Time Operating System (RTOS)** A category of operating systems defined by their ability to respond quickly and predictably for a given task

**SoC** **System-on-Chip (SoC)** A single chip that holds all of the necessary hardware and electronic circuitry for a complete system. SoC includes on-chip memory (RAM and ROM), the microprocessor, peripheral interfaces, I/O logic control, data converters, and other components that comprise a complete computer system

**ZedBoard** **ZedBoard** A low cost development board featuring a Zynq-700 0 SoC, and a number of peripherals

**Zynq** **Zynq** Xilinx' AP SoC. The characteristic feature of Zynq is that it combines a dual-core ARM Cortex-A9 processor with traditional Series-7 FPGA logic fabric

**Zynq Book** **Zynq Book** A book that summarizes all the important aspects when working with Zynq and provides a strong and easy understandable introduction to the topic. The book has been written by a team of University of Strathclyde Glasgow in cooperation with Xilinx

# A. Anhang

## A.1. Fortschritte

### A.1.1. Woche 1-2

**Besprechung: 25.September 2023** Ich habe mit Marcus Hudritsch besprochen, dass ein Ziel dieses Projekts ist, dass der Betrachter von der Intel RealSense D435 erfasst wird, diese Daten werden dann in Unity importiert und Unity kann dann einige virtuelle Objekte erzeugen. Diese Objekte werden nun verschoben wenn sich der Betrachter bewegt.
Die Kamera werde ich am Freitag, 29.Septemper 2023 abholen können. Das Modell der Kamera habe ich mir notiert, damit ich mit den Recherchen beginnen kann.
**Einbindung mit Unity** Ich habe verschiedene Quellen im Internet studiert, wie man die Intel Kamera in die Unity Game Engine importieren kann. Mir ist nun aber noch nicht so ganz klar, nach welchem Ansatz dies dann funktioniert. In einem Bericht wurde die Intel RealSenseSDK 2.0 mit CMAKE kompiliert, was dann ein Unity Projekt generiert. Leider ist dieses Unity Projekt nicht brauchbar, da viele Teile fehlen. Ein anderer Ansatz nimmt ein von RealSenseSDK 2.0 generiertes Paket und dieses wird dann in Unity importiert. Dieser Ansatz scheint eher zu funktionieren.
**RealSense D435** Die Kamera habe ich bis jetzt noch nicht. Ich habe mal ein paar Youtube Videos geschaut, wie die Kamera mit der Software: Intel RealSense Viewer benutzt werden kann. Mittels Skripttext der eingebunden wird zeigt der Cursor in Echtzeit an, wie gross die Distanz des vom Cursor gewählten Punktes von der Kamera bis zum Punkt auf dem Objekt ist.
**Latex Dokument** Die Projektarbeit 2 werde ich mit Latex machen. Da ich noch keine Erfahrungen mit Latex habe, lerne ich die nötigen Grundlagen und versuche eine Dokumentstruktur aufzubauen wie sie üblich ist für eine solche Arbeit.
**Erkenntnisse** Ich habe versucht die Intel RealSenseSDK 2016 mit Unity 2017 laufen zu lassen. Das hat soweit auch in Windows 11 funktioniert. Leider ist aber die Kamera D435 nicht kompatibel mit der SDK und wird in Unity nicht erkannt.
Das neuste SDK von Intel wurde stark reduziert und bietet in Unity keine Anwengugen mit Face Tracking.
Ich werde also nun versuchen mit der Nuitrack SDK mein Projekt aufzubauen. Diese SDK ist gut dokumentiert, bietet Features für Facetracking und auch für

Skeletal Tracking.

### A.1.2. Woche 3-4

**Besprechung: 4.Oktober 2023** Ich habe mit Marcus Hudritsch besprochen, dass es mit dem RealSense SDK von Intel nicht möglich sein wird, da dieses abgespeckte SDK die nötigen Tools fürs Tracking nicht mehr bietet. Ich werde also mit dem Nuitrack SDK versuchen die Person zu tracken.

### A.1.3. Woche 5-6

**Besprechung: 18.Oktober 2023** Das Tracking mit Nuitrack läuft soweit. Dieses SDK bietet sogar mehr Features als dass ich benötige. Beispielsweise ist es möglich auch die Rotation vom Kopf, also die Drehung in den Achsen x, y und z zu tracken. Die Rotation in der x Achse entspricht dem pitch, die Rotation in der y Achse einem yaw und die Rotation in der z Achse einem roll. Ein kleiner Nachteil vom Nuitrack SDK ist, dass es nach 3 Minuten stoppt. Wer die Applikation länger laufen möchte muss dann bezahlen. Mit Marcus Hudritsch habe ich besprochen, dass ich nun eine künstliche Wand aufbauen kann mit einem Fenster darin. Die virtuelle Kamera filmt dann die ganze Szene etwas vor dieser Wand. Was uns jetzt interessiert ist eigentlich nur der Ausschnitt des Fensters innerhalb der künstlichen Wand.

### A.1.4. Woche 7-8

**Besprechung: 1.November 2023** Ich habe nun dieses Fenster als grau eingefärbten Screen dargestellt. Mit Marcus Hudritsch habe ich das weitere Vorgehen besprochen. Mir war bis jetzt noch nicht ganz klar, wie ich vorgehen muss um diesen Screen zu entzerren. Wir haben besprochen, dass mit der LookAt Funktion in Unity die Kamera stets auf den Mittelpunkt vom Screen gerichtet werden kann. Weiter braucht es einen Shader um die Bildmanipulationen vorzunehmen. Da ich noch keine Erfahrung mit Shadern habe, ist mir da vieles noch völlig unklar.

**Besprechung: 10.November 2023** Diese Besprechung hielten wir vor Ort in Biel. Zusammen schauten wir an, was beim Aufbau noch verbessert werden muss. Der Blickwinkel vom Betrachter war bisher eher zu hoch, so dass der Screen zu stark von oben betrachtet wird.

### A.1.5. Woche 9-10

**Besprechung: 15.November 2023** Ich habe nun einen Shader welcher die Bildmanipulation mittels einer Matrix möglich macht. Leider stammt dieser Shader

von einer Rotations-Matrix. Rotation und Scaling funktionieren normal. Eine Translation ist aber so noch nicht möglich. Weiter ist mir nicht klar, wie ich mit dieser Matrix die Eckpunkte vom Screen finden kann und dann auch noch entzerren. Dieser Shader mit der Rotations-Matrix scheint mir also nur wenig weiter zu helfen.

### A.1.6. Woche 11-12

**Besprechung: 29.November 2023** Ich habe nun einen Shader der als Pixel Shader wirkt eingebaut. Zuerst habe ich mit einer Homographie Matrix versucht das Polygon vom Screen zu entzerren. Leider hat dies nur funktioniert, wenn ich genau gerade vor dem Screen stand. Sobald ich etwas seitlich stand resultierte dies in einer Verzerrung. Leider stimmt mein Algorithmus noch nicht, wie ich die Eckpunkte vom Screen finde. Weil ich mit der Homographie Matrix nicht weitergekommen bin, habe ich OpenCV über einen Wrapper in mein Projekt eingebaut. Dies hat erst beim zweiten Versuch geklappt. Ich habe auch noch mit EmguCv als Wrapper rumgeübt, was aber nicht wirklich funktioniert hat. Bei OpenCV hatte ich grosse Probleme die Funktionen so einzubauen, dass auch ein sichtbares Bild angezeigt wird. OpenCV verwendet andere Matrizen als Unity und auch die Textures müssen als in OpenCV als Matrix dargestellt werden. Weiter werden die Texturen zwar auf der GPU gespeichert, sind dann aber nicht einfach verfügbar auf der CPU und müssen speziell aufgerufen und abgefragt werden. Nun soweit der Stand von meinem Projekt. Die Entzerrrung mit OpenCV funktioniert nun auch zusammen mit dem PixelShader. Was noch fehlt ist die Berechnung der Eckpunkte vom Screen. Mein Ansatz dies zu bewerkstelligen ist eine Clipping Plane durch den Mittelpunkt vom Screen zu legen mit der Normalen dieser Ebene in Richtung vom Mittelpunkt zur Kamera. Verschiebt sich die Kamera, dreht sich diese Ebene um den Mittelpunkt vom Screen, welcher in meinem Projekt die Koordinaten (0, 0, 0) hat. Nun muss ich nur die Eckpunkte vom Screen perspektivisch, das heisst in Richtung Eckpunkt zur Kamera mit dieser Clipping Ebene schneiden. Die Schnittpunkte relativ zum gesamten Clipping Fenster entsprechen dann den Eckpunkten auf meiner Projektion. Es scheint, dass Unity die Clipping Plane etwas anders berechnet, als ich dies tue. Ich habe noch leichte Abweichungen. Marcus Hudritsch hat mir das Kameramodell noch etwas näher gebracht. Möglich, dass ich mit diesem Ansatz eher zum Ziel komme.