HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

BACHELORARBEIT

Herr
Thimo Müller

Dynamisches Pixel-Art

Mittweida, 2019

Fakultät Angewandte Computer- und Biowissenschaften

BACHELORARBEIT

Dynamisches Pixel-Art

Autor:
Herr

Thimo Müller

Studiengang:
Medieninformatik und Interaktives Entertainment

Seminargruppe:
MI14w-2B

Erstprüfer:
Professor Alexander Marbach

Zweitprüfer:
Daniel Stockmann

Einreichung:
Mittweida, 19.02.2019

Verteidigung/Bewertung:
Mittweida, 2019

Faculty Hochschule Mittweida

BACHELORTHESIS

Dynamic Pixel-Art

author:
Mr.

Thimo Müller

course of studies:
Media Informatics and Interactive Entertainment

seminar group:
MI14w-2B

first examiner:
Professor Alexander Marbach

second examiner:
Daniel Stockmann

submission:
Mittweida, 19.02.2019

defence/ evaluation:
Mittweida, 2019

Bibliografische Beschreibung:

Müller, Thimo:
Dynamisches Pixel-Art. - 2019. - 1, 30, 1 S.
Mittweida, Hochschule Mittweida, Fakultät Angewandte Computer- und Biowissenschaften, Bachelor Thesis, 2019

Referat:

In diesem Dokument handelt es sich um eine Beschreibung und Darstellung von Methoden zur Darstellung von Pixelgrafiken anhand von 3D-Objekten in Unity. Des Weiteren beinhaltet dieses Dokument eine Beschreibung von Lighting and Shadows in Unity und wie diese für 2D-Grafiken genutzt werden können.
Es wird beschrieben was Computergrafiken sind und wie diese dargestellt werden, sowie was Schader sind und wie diese funktionieren.
Des Weiteren werden Beispiele von Methoden zu den oben genannten Gebieten erläutert, die bereits vorhanden sind und ihre Funktionsweiße und Nutzen analysiert.
Es werden Methoden erarbeitet und dargestellt wie Pixelation von 3D-Grafiken möglich ist, sowie das Erstellen von Lighting und Shadows bei 2D-Grafiken. Aus diesen Methoden wird für jeweils ein Gebiet eine Vorangehensweise ausgewählt und ein Proof of Concept erstellt.
Diese werden Analysiert und evaluiert, sowie mögliche Nutzen erläutert.

Inhaltsverzeichnis

1. Einführung und Motivation

In diesem Kapitel werden die Ziele, die Motivation und der Aufbau der Arbeit vorgestellt. Es werden zwei Herangehensweisen untersucht 3D-Objekte in Pixel-Art darzustellen.
Des Weiteren wird untersucht wie ein 2D-Objekt in Unity Schatten werfen kann.
Es wird beschrieben welchen Nutzen und welche Vorteile diese Methoden haben. Weiterhin wird ein Proof of Concept für die Methoden der Darstellung von 3D-Objekten als Pixel-Art und der des Schattenwurfes erstellt. Weitere Methoden werden theoretisch angegangen.

1.1 Aufgabenbeschreibung

Gegenstand der Bachelorarbeit sind Lighting und Shading für 2D-Sprites (z.B. Pixel Art), sowie die Darstellung von 3D-Objekten als zweidimensionale Sprites beziehungsweise Pixel-Art.
Für das Lighting und Shading der Pixel Art werden Möglichkeiten untersucht, inwiefern dynamisches Lighting auf 2D-Sprites angewandt werden kann und welche Zusatzinformationen das Sprite benötigt wie z.B. Normal Maps.

Das Ziel dieser Arbeit ist Methoden zu untersuchen und zu entwerfen, welche 3D-Objekte als Pixel Art anzeigen sowie Licht und Schatten auf 2D-Objekte wirken lassen. Weiterhin ist dafür ein Proof of Concept zu erstellen. Dazu ist es wichtig zu verstehen, wie Shader in Unity funktionieren.

Ebenfalls wird untersucht, wie ein animiertes Sprite Schatten werfen oder von Schatten anderer Objekte beeinflusst werden kann.
Um 3D-Objekte als 2D-Sprites darstellen zu lassen, sollen im Rahmen der Bachelorarbeit verschiedene Lösungsansätze untersucht und verglichen werden.
Dieser Vorgang könnte z.B. wie die Rotoskopie funktionieren, eine Technik in welcher über echte Filmaufnahmen gezeichnet wird.
Wenn dies möglich ist, soll das Konzept um Physikeffekte, Partikel, dynamische Umgebungen (Zerstörung, Wasser, bewegliche Hintergründe z.B. Vegetation) und volumetrische Effekte (Explosionen, Rauch, Feuer, Nebel) erweitert werden. Im Rahmen des Möglichen wird ein Proof of Concept an ausgewählten Beispielen herausgearbeitet.

1.2 Aufgabengebiete der Pixel-Art Erstellung

Das Kreieren von Pixel Art für Videospiele ist sehr zeitaufwendig und dessen Nachbearbeitung umständlich und aufwändig.

Auch können Assets nicht mehrmals benutzt werden, da meist jedes Objekt individuell ist. Durch das Erstellen eines 3D-Objektes welches in Pixel Art umgewandelt wird, kann all dies umgangen werden. Nicht nur können Animationen schnell und einfach angepasst werden, sondern auch bereits bestehende Assets erneut benutzt werden.

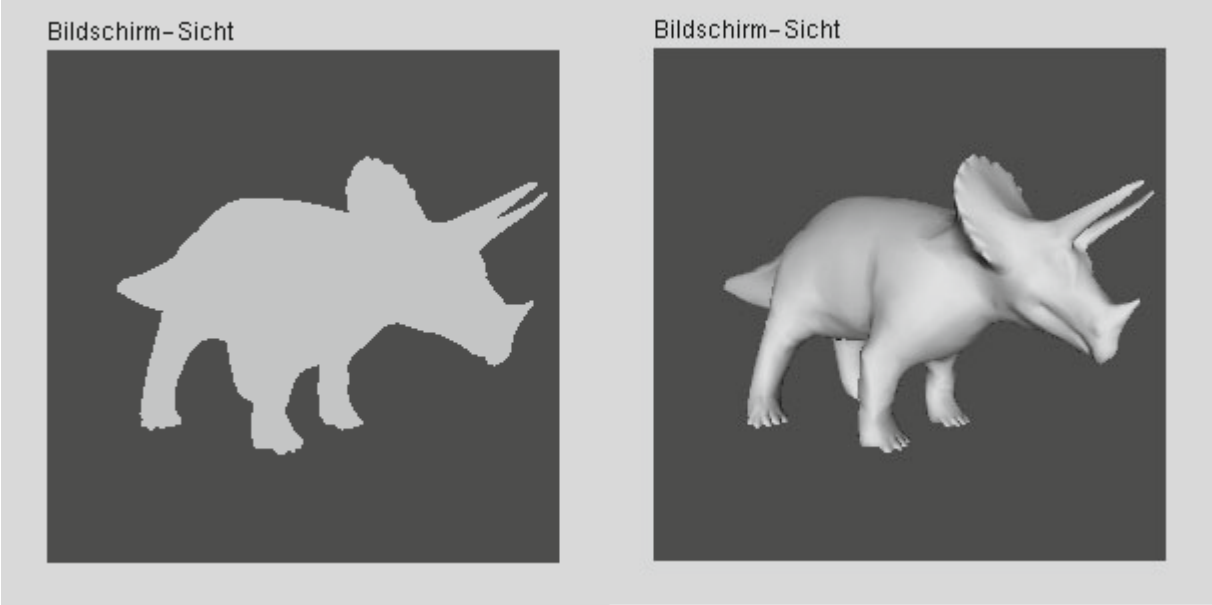
Dadurch ist es ein Leichtes mit wenigen Klicks ein abgeändertes Asset zu erhalten. Weiter können

dadurch ebenfalls Effekte wie Feuer oder Partikelsysteme schnell in Pixel Art umgewandelt werden.

1.3 Aufgabengebiete des Lighting und Shading

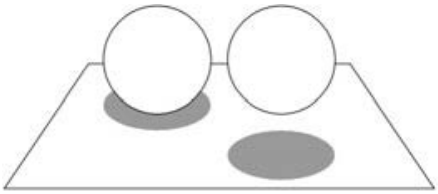
Lighting und Shading werden in der Computergrafik verwendet um auf einem zweidimensionalen Bildschirm einen dreidimensionalen Effekt zu kreieren. Ohne eine Beleuchtung und die damit verbundenen Schattierungen wirkt eine Grafik flach. Dies wird auch „Formwahrnehmung aus Schattierungen (Shape from Shading)“ genannt.

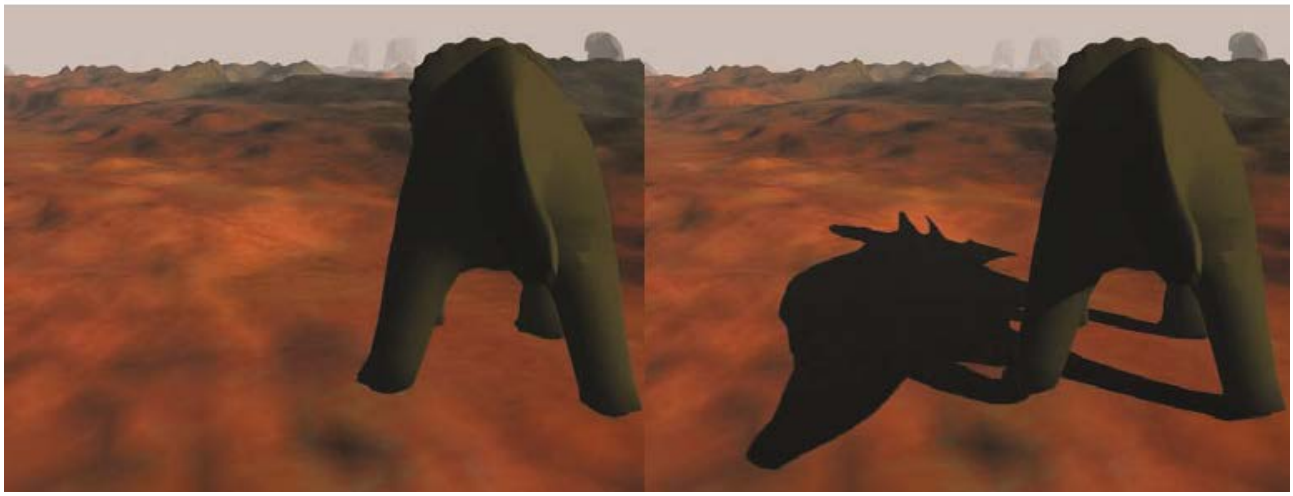
Das Generieren von Farbwerten, welche als Beleuchtung und Schatten wahrgenommen werden, erfolgt in zwei Stufen. In der ersten Stufe wird durch eine bestimmte Beleuchtungsformel jeweils mit den Eigenschaften und der Position der Lichtquellen in Verbindung mit dem Objekt ein Farbwert für jeden einzelnen Vertex berechnet. In der zweiten Stufe werden die Farbwerte durch ein Schattierungsverfahren, in dem Bereich wo Schattierung generiert werden soll, interpoliert.



(Grafik aus Nischwitz CB, *K12*, 2011)

Des Weiteren ergibt die Position von Schatten und Licht einen Eindruck von Form und Position des dargestellten 3D-Objektes. So kann durch die Positionierung eines Schattens direkt unter dem 3D-Objekt der Eindruck erweckt werden, dass das Objekt auf dem Boden steht. Wäre der Schatten versetzt und nicht direkt mit dem Objekt verbunden oder gar kein Schatten vorhanden, so würde der Eindruck erweckt werden, dass das Objekt "schwebt", d.h. dass es sich nicht auf dem Boden befindet.





(Grafik ausNischwitz CB, *K14*, 2011)

In Unity werden 2D-Objekte vom Licht ignoriert. Daher ist es wichtig ein System zu erstellen was die Schattengenerierung ermöglicht.

1.4 Aufbau der Arbeit

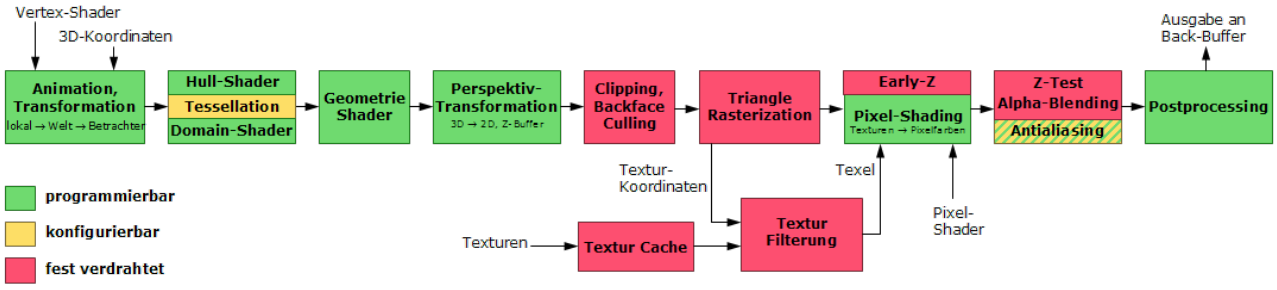
Die vorliegende Arbeit teilt sich in fünf Kapitel ein. Das erste Kapitel beinhaltet die Einleitung, Motivation, Gliederung und Zielstellung. Das zweite Kapitel erläutert die grundlegende Funktionsweise von Shadern und wie diese in Unity arbeiten. Das dritte Kapitel zeigt die Anforderungen und die ersten Konzepte der Methoden auf. Im vierten Kapitel wird jeweils ein Proof of Concept zweier Methoden vorgestellt. Im fünften Kapitel ist die Evaluation zu lesen, in welcher erläutert wird, ob die Konzepte und deren prototypischen Implementierungen die gewünschten Ergebnisse erbringen.

2. Grundlagen

2.1 Rendering-Pipeline

Eine Rendering-Pipeline(laut deacedic), oder auch Computergrafik-Pipeline beschreibt in der Computergrafik eine Modellvorstellung davon, an welchen Punkten eine Grafikkarte 3D-Szenen auf dem Bildschirm zu rendern hat.
Diese Punkte an denen es zu Rendern gilt sind sowohl von der Hardware als auch von der Software abhängig. Des Weiteren spielen auch die Grafikeinstellungen eine Rolle.
Zum Bearbeiten von Rendering-Pipelines werden Grafik APIs verwenden, wie zum Beispiel OpenGL.

Das Anwenden, beziehungsweise Bearbeiten einer Grafik-Pipeline findet beim Echtzeitrendern statt. In den meisten gängigen Grafikkarten ist die Rendering-Pipeline bereits fest eingebaut.
Die Punkte der Pipeline laufen in der Regel parallel ab. Dennoch werden diese blockiert, bis der letzte Punkt einer Reihe abgeschlossen wurde.



(Grafik-Pipeline von deacedic)

2.2 Shader in Unity

Für die Nutzung eines Shaders in Unity sind drei Komponenten notwendig. Ein GameObject, ein Material und der Shader.

Ein 3D-Model ist im Grunde eine Gruppe von Vertices, die Gruppen von Dreiecken bilden. Ein Vertex enthält folgende Informationen: Koordinaten (Position), Farbe, die Richtung in die das Vertex zeigt (normal) und gegebene Koordinaten der Texturen (UV Position).

Ein Material trägt einen Shader, welcher angibt wie das gegebene 3D-Model gerendert werden soll. Ist kein Material vorhanden benutzt Unity ein Default Material.
Ohne ein Material kann ein 3D-Model nicht gerendert werden. In dem Material sind Informationen des Shaders enthalten und können, sofern vorhanden, unterschiedliche Werte des Shaders beeinflussen.

Dadurch kann ein Shader auf unterschiedlichen Materialien unterschiedliche Renderergebnisse erzielen.

2.3 Aufbau eines Shaders in Unity

In Unity werden zwei Arten von Shadern unterstützt. Zum einem der Surface-Shader und zum anderen der Vertex-/Fragment-Shader.

Der grundlegende Aufbau beider Shader gestaltet sich identisch, und zwar wie folgt:

```
Shader "MyShader"
{
    Properties
    {
        // Die Eigenschaften des Shaders
        // -Die Texturen
        // -Die Farben
        // -Die Parameter
    }

    SubShader
    {
        // Der eigentliche Code des Shaders
        // - surface shader
        //    oder
        // - vertex/fragment Shader
    }
}
```

Es ist möglich (und empfohlen) mehrere SubShader hintereinander zu erstellen. Die SubShader enthalten den Eigentlichen Code, welcher der GPU Befehle erteilt wie etwas zu rendern ist.

Unity führt einen SubShader nach dem anderen aus und prüft jeweils ob die GPU den nötigen Anforderungen des SubShaders entspricht, also ob der auszuführende Code mit der zur Verfügung stehenden Grafikkarte kompatibel ist.

Dies ist notwendig, um das gewünschte Ergebnis auf unterschiedlichen Plattformen (Windows-PC, Smartphones, etc.) zu erzielen ohne redundante Shader verwenden zu müssen.

2.3.1 Der Properties Bereich

In dem Properties (Eigenschaften)-Bereich sind alle notwendigen Informationen die der Shader benötigt enthalten.

Dieser Bereich ähnelt dem eines Unity MonoBehavior-Scripts, sofern alle Angaben im Inspektor-Bereich angezeigt werden und angepasst/verändert werden können.

Der Unterschied zu einem Script ist, dass alle Änderungen im Inspektor zu den Properties permanent sind, wenn diese im Play Mode geändert werden. Daher sollte hier Vorsicht geboten sein.

Der Aufbau des Properties-Bereiches gestaltet sich wie folgt:

```
Properties
{
    _MyTexture ("My texture", 2D) = "white/black/grey" {}
    _MyNormalMap ("My normal map", 2D) = "bump" {}  // Grey

    _MyInt ("My integer", Int) = 2
    _MyFloat ("My float", Float) = 1.5
    _MyRange ("My range", Range(0.0, 1.0)) = 0.5

    _MyColor ("My colour", Color) = (R, G, B, A)
    _MyVector ("My Vector4", Vector) = (x, y, z, w)
}
```

`_MyTexture` enthält die Information für die Textur. Der Wert „2D“ ist notwendig, da dieser der Indikator dafür ist, dass es sich um eine Textur handelt.

Es kann mit den Farben Weiß, Schwarz und Grau (white/black/grey) initialisiert werden.

Dasselbe gilt für `_MYNormalMap`“ welches die NormalMap enthält. Hier kann zusätzlich der Wert „bump“ angegeben werden um zu definieren, dass die Textur als NormalMap genutzt wird.

Dadurch gibt Unity diesem Parameter automatisch den Farbwert #808080. Dieser Wert repräsentiert keinen Bumpwert und ist somit sozusagen flach.

`_MyInt`, `_MyFloat` und `_MyRange` sind Werte, die später von Funktionen genutzt werden können.

„`_MyColor`“ trägt den Wert der Farbe und setzt sich aus vier Einzelwerten zusammen: Rot-Anteil, Grün-Anteil, Blau-Anteil und dem Alpha-Anteil(Transparenz).

„`_MyVector`“ trägt die Koordinaten im dreidimensionalen Raum und benötigt ebenfalls vier Werte: x-Position, y-Position, z-Position und den Wert w. Dieser wird für Scaling und Normalization, sowie für Berechnungen mit anderen 4x1 Matrizen benötigt.

All diese Werte können im Inspektor gesehen und verändert werden. Sie sind äquivalent zu public-Variablen in einem C#-Script. Die angegebenen Variablen können nun im SubShader genutzt werden.

2.3.2 Der SubShader Bereich

Im SubShader-Bereich werden die definierten Werte des Properties-Bereichs genutzt. Jedoch müssen diese im SubShader Bereich noch definiert werden, damit Unity mit diesen Werten arbeiten kann. Dies ist wie folgt aufgebaut:

```
SubShader
{
    // Code of the shader
    // ...
    sampler2D _MyTexture;
    sampler2D _MyNormalMap;

    int _MyInt;
    float _MyFloat;
    float _MyRange;
    half4 _MyColor;
    float4 _MyVector;

    // Code of the shader
    // ...
}
```

Texturen werden dem Typ sampler2D zugewiesen um als Texturen erkannt zu werden.

Für die Farben ist half4(32bit) und für Vektoren float4(16bit) anzugeben.

Es ist in diesem Bereich darauf zu achten, dass die gegebenen Bezeichnungen für die Werte im Properties-Bereich mit den hier angegebenen übereinstimmen.

Die im Shader gegebenen Datentypen können, anders als in C#, von darauffolgenden Datentypen abweichen ohne eine Fehlermeldung aufzurufen.

Daher sollte hier immer genau beachtet werden was angegeben wird.

Dies kann jedoch zum Vorteil genutzt werden, ohne die Datentypen konvertieren zu müssen und zwar dadurch, dass Unity überflüssige Werte ignoriert.

2.3.3 Die Rendering Order

Der Shader wird für jeden Pixel im Bild aufgerufen, daher ist auf die Performance sehr zu achten.

Dadurch dass der Shader in der GPU verarbeitet wird, ist nur eine bestimmte Anzahl von Befehlen in ihm möglich.

Die Sprache der Shader ist Cg/HLSL, welche der Programmiersprache C ähnlich sind.

Der Aufbau eines Shaders sieht wie folgt aus:

```
SubShader
{
    Tags
    {
        "Queue" = "Geometry"
        "RenderType" = "Opaque"
    }
    CGPROGRAM
    // Cg / HLSL code of the shader
    // ...
    ENDCG
}
```

Vor dem eigentlichen Code können sogenannte „Tags“ angegeben werden.

Tags werden genutzt um Unity bestimmte Eigenschaften des Shaders mitzuteilen. In diesem Beispiel wird das Tag „Queue“ genutzt um anzugeben, in welcher Reihenfolge gerendert werden soll.

„Rendertype“ gibt an, in welcher Art das 3D-Model gerendert werden soll.

Der eigentliche Cg/HLSL-Code ist im CGPROGRAM anzugeben. Dieser wird durch den Befehl CGPROGRAM initialisiert und durch den Befehl EndCG beendet.

Die GPU rendert Polygone anhand der Entfernung zur Kamera (ZTest). In der Regel werden Polygone, welche weiter weg sind zuerst gezeichnet, während nähere Polygone später gezeichnet werden und die vorigen überschreiben.

Dies kann jedoch durch das Tag „Queue“ beeinflusst werden. Diesem können positive Integer-Werte zugewiesen werden, welche die Rendering Order beeinflussen. Es gilt je höher der Wert, desto später wird etwas gerendert.

Es gibt bestehende „mnemonic“-Labels welche alternativ genutzt werden können. Diese enthalten einen spezifischen Integer-Wert. Folgende mnemonic-Labels sind von Unity vordefiniert:

- Background (int-Wert = 1000): Wird vor allen anderen gerendert. Genutzt für Hintergründe oder eine Skybox.
- Geometry (int-Wert = 2000): Das Standardlabel für die meisten undurchsichtigen Objekte.
- Transparent (int-Wert = 3000): Das Standardlabel für durchsichtige Objekte, wie z.B. Glas.
- Overlay (int-Wert = 4000): Das Standardlabel für Effekte oder GUI-Elemente. Wird zuletzt gerendert.

Diese mnemonic-Werte können auch angepasst werden, indem sie mit einem Integer verrechnet werden, wie zum Beispiel: Geometry+1 was dem Wert 2001 entspricht.

Damit lässt sich die Rendering Order beeinflussen um besondere Effekte zu kreieren, wie zum Beispiel das Darstellen von bestimmten Objekten an einem bestimmten Punkt.

Konkret könnte dies zum Aufzeigen von Quest-Gegenständen oder besonderen Objekten verwendet werden. Spiele wie Assassin‘s Creed nutzen diese Art des Renderns, um in bestimmten Darstellungsmodi ein Objekt hervorzuheben.

Es werden jedoch unter anderem eine Outline-Funktion, sowie eine flatcolored-Funktion in diesem Fall zusätzlich genutzt.

2.3.4 Der ZTest

Der ZTest ist eine Funktion, welche die GPU automatisch ausführt. Dabei wird geprüft, welche Pixel verdeckt sind und somit nicht gezeichnet werden. Er nutzt einen Tiefenpuffer, welcher dieselbe Größe wie der Bildschirm, auf dem gerendert wird besitzt. Wenn ein Pixel von einem anderen Pixel verdeckt wird, so wird er verworfen. Dadurch werden redundante Pixel unabhängig von der Rendering Order entfernt.

2.3.5 Der Surface Shader

Ein Surface Shader wird genutzt, wenn Licht simuliert werden soll. Er führt die Berechnungen im Hintergrund aus und es können Eigenschaften wie albedo, normals, reflexion und glow in einer Funktion namens „surf“ angegeben werden. Diese werden einem lighting model übergeben, welches die RGB-Werte für jeden Pixel zur Verfügung stellt.

Der Cg-Code sieht in diesem Fall folgendermaßen aus:

```
CGPROGRAM
// Uses the Lambertian lighting model
#pragma surface surf Lambert

sampler2D _MainTexture; // The input texture

struct Input {
    float2 uv_MainTexture;
};

void surf (Input IN, inout SurfaceOutput o) {
    o.Albedo = tex2D (_MainTexture, IN.uv_MainTexture).rgb;
}
ENDCG
```

Hier wird die Textur `_MainTexture` erstellt und in der surf-Funktion als eine albedo-Eigenschaft des Materials zugewiesen. Es wird ein standard Lambertian lighting genutzt, um die Reflektion des Lichts zu simulieren. In diesem Fall wäre dies ein diffuse Shader.

2.3.6 Der Vertex- und Fragment-Shader

Vertex-/Fragment-Shader entsprechen den Stufen der Renderpipeline bzw. des Grafiktreibers. Es ist keine eingebaute Funktion vorhanden, wie Licht behandelt werden soll.

In der vert-Funktion wird die Geometry-Komponente des Objektes beeinflusst. Die Informationen, die ein Vertex beinhaltet werden hier verändert.

Im der frag-Funktion werden die RGB-Werte des Vertex bestimmt. Dies ist von Nutzen für besondere 3D- oder 2D-Effekte.

Der Code für ein Vertex-/Fragment-Shader kann wie folgt aussehen:

```
Pass {
    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    struct vertInput {
        float4 pos : POSITION;
    };

    struct vertOutput {
        float4 pos : SV_POSITION;
    };

    vertOutput vert(vertInput input) {
        vertOutput o;
        o.pos = float4 UnityObjectToClipPos(float3 pos) ;
        return o;
    }

    half4 frag(vertOutput output) : COLOR {
        return half4(0.0, 1.0, 0.0, 1.0);
    }
    ENDCG
}
```

Hier werden die Vertices in der vert-Funktion von ihren ursprünglichen 3D-Koordinaten in 2D-Koordinaten konvertiert. Der Befehl `float4 UnityObjectToClipPos(float3 pos)` führt die gewünschte Berechnung dafür aus.

In der frag-Funktion wird jedem Pixel eine Farbe zugewiesen, in diesem Fall grün.

3. Anforderungen und Spezifikationen

3.1 Anforderungen

Die Darstellung von 3D-Grafiken als Pixel Art ist durch verschiedene Techniken möglich, wie z.B. durch das nutzen von Shadern oder bearbeiten von 3D-Grafiken im Vorfeld(Prefabs).
Die Technik des Erstellens und Benutzen eines Shaders gestaltet sich wie folgt:
Ein Shader braucht die nötigen Funktionen um das angezeigte 3D-Objekt in einem verpixelten Zustand darzustellen. Des Weiteren muss das 3D-Objekt flach dargestellt werden.
Diese Darstellungsform wird häufig in Toon-/Cel-Shadern genutzt. Oft wird für einen Toon-Shader zusätzlich eine Outline-Funktion erstellt, jedoch ist diese für das Erstellen eines pixelierten 3D-Objektes nicht zwingend von Nöten, da oft Pixel-Art-Assets keine Outline besitzen.

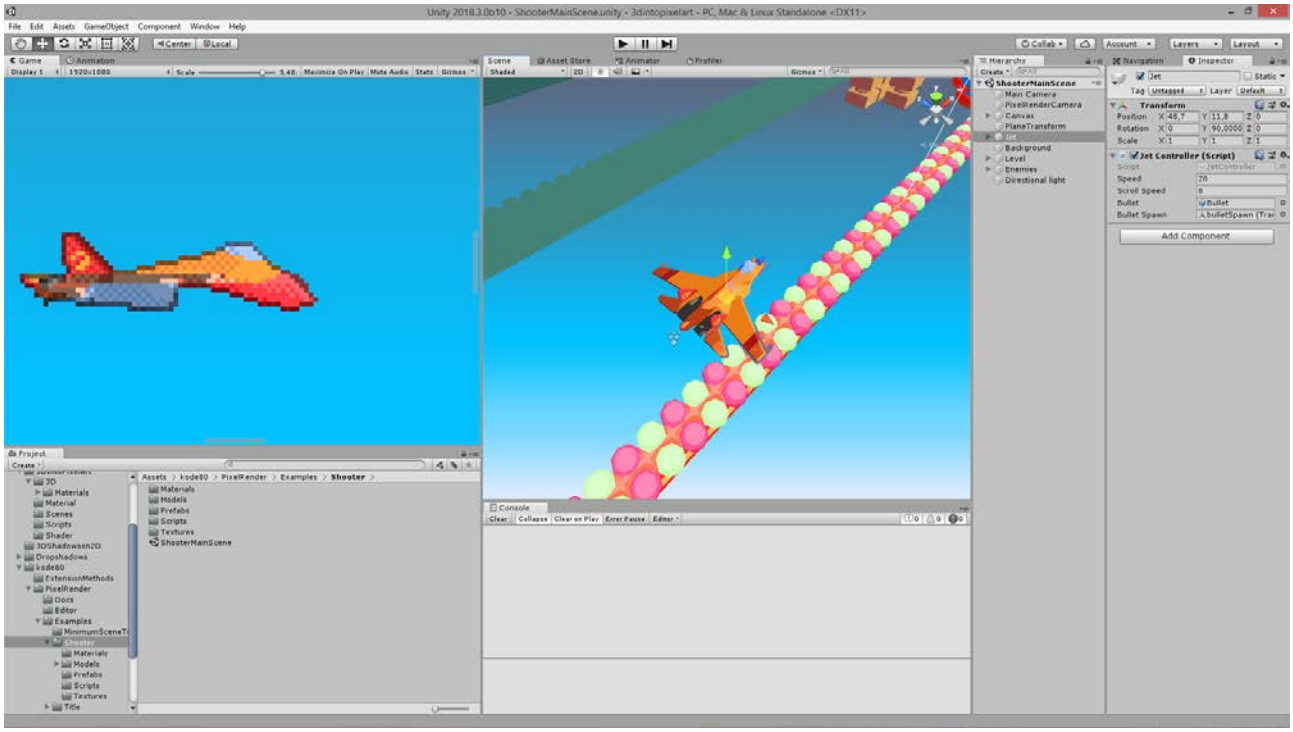
Bei der Technik von bereits vorbereiteten 2D-Assets müssen 3D-Objekte in ein 2D-Format umgeändert werden. Dies kann durch das Überzeichnen per Hand erreicht werden oder durch ein Programm, dass im Vorfeld erstellt werden muss, welches die 3D-Objekte in Pixel-Art konvertiert.

Schattierungen sollen dynamisch dargestellt werden und abhängig von der Lichtquelle sein. Diese sollen einfach an mehrere Objekte gehangen werden können, ohne die zuvor erstellte Arbeit anpassen zu müssen.

3.2 Vergleiche von bereits bestehenden Lösungen 3D-Pixel-Art

3.2.1 Pixelation während der Laufzeit:

Ein bereits bestehendes System dieser Art ist eine Erweiterung für Unity namens "PixelRender 1.1 " von Kodex80. In dieser Erweiterung werden 3D-Objekte als Pixel-Art dargestellt. Der Effekt entsteht durch eine nahe Kamera, durch welche die Objekte pixeliert werden und kein Antialiasing stattfindet.
Des Weiteren wird ein dither-Effekt im Schattenbereich angezeigt.
Dieser Farbverlauf wird im Shader PixelartShader generiert.
Als Textur ist eine Farbpalette angehängt aus welcher die Farben herausgelesen werden. An der Kamera ist ein Script, welches eine Outline erstellt.
Durch die Kombination wirken die 3D-Objekte wie Pixelgrafiken.



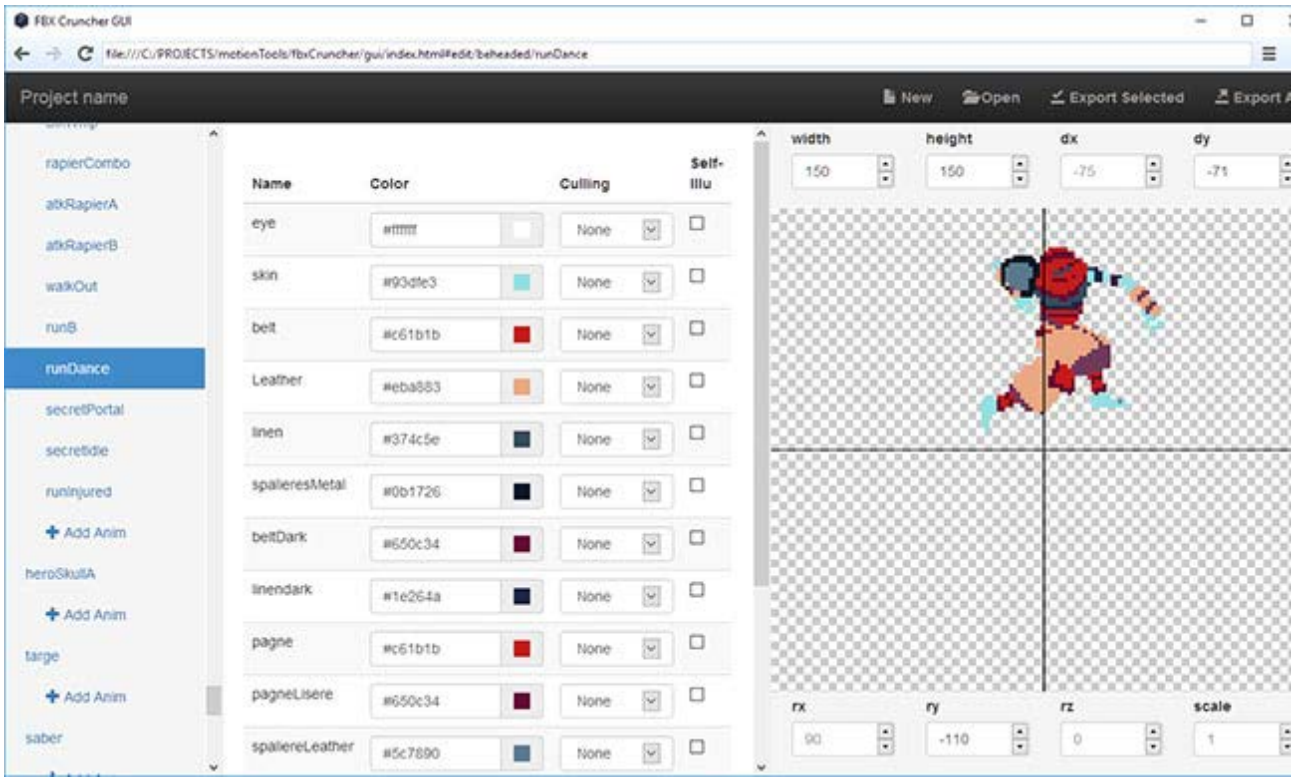
(Beispiel anhand von Kodex80`s Pixelrenderer)

3.2.2 Vorgeränderte Pixelgrafik:

Eine andere Möglichkeit aus 3D-Objekten 2D-Sprites/-Pixelgrafiken zu erstellen, ist die 3D-Grafik zu rendern und auf das gewünschte Format umzuwandeln. Donkey Kong Country von Square, veröffentlicht am 21.11.1994, hat diese Technik angewendet. Square hat pre-rendered 3D-24-Bit-Grafiken in 2D-16-Bit-Grafiken umgewandelt.

Eine neue Art der Rotoskopie entwickelte Thomas Vasseur. Er war für das Erstellen der Art-Assets des Spiels DeadCells zuständig. In einem Artikel beschreibt er seine Vorgehensweise und anhand daran wurde ein ähnliches Konzept erstellt.

In seinem Artikel beschreibt er, dass er einfache 3D-Objekte mit einfarbigen Texturen erstellt hat. Ebenfalls wurden Animationen für das 3D-Objekt erstellt.
Diese 3D-Objekte wurden dann in ein speziell dafür geschriebenes Programm importiert, welches das 3D-Model auf 50 Pixel reduziert und die Farben angepasst hat.



(Spezielles Programm für DeadCells, um 3D-Objekte in 2D-Sprites umzuwandeln.)

Die Animationen des 3D-Objektes wurden daraufhin auf Keyframes reduziert mittels Interpolation vor und nach diesen. Dadurch wurden die Animationen zu pose-to-pose Animationen.

Weiterhin werden aus dem speziellen Programm anschließend die Animationen als einzelne Frames exportiert.

Das Programm exportiert aber nicht nur das sichtbare Sprite, sondern auch eine dazugehörige Normalmap. Dadurch kann mithilfe eines Cel-Shaders Volumen simuliert werden.

Diese Arbeitsweise ist effizient und erlaubt das schnelle Anpassen von Animationen, da lediglich die Animationen des 3D-Objektes angepasst werden müssen und nicht eine ganze Sequenz neu gezeichnet werden muss.

Ebenfalls kann das 3D-Objekt an sich verändert werden oder sogar seine Assets für unterschiedliche 3D-Objekte genutzt werden. Dies wäre bei von Hand gezeichneten 2D-Sprites nicht möglich.

Diese Vorgehensweise lässt sich in Unity mithilfe von Scripten imitieren.

Es wird ein 3D-Objekt erstellt mit einer simplen Animation. Das 3D-Objekt wird exportiert und in Unity importiert. Der Animationscontroller wird erstellt und der Animationsclip hinzugefügt.

Ein Script, welches daraus einzelne Sprites kreiert, muss durch die Kamera das 3D-Model „fotografieren“ und das daraus entstehende Bild in ein Spritesheet einfügen. Des Weiteren sollte eine Normalmap erstellt werden, damit Licht auf die Sprites wirkt und darauf Schatten wirft.

2.3.1 Schattierung durch 3D-Modelle

Ein 3D-Objekt in Unity wirft automatisch Schatten und dadurch kann ein simulierter 2D-Schatten gewonnen werden.

Hierfür werden zwei Layers erstellt an denen jeweils Objekte liegen die Schatten werfen oder Schatten erhalten. Die Layers werden hier „Caster“ und „Receiver“ genannt.
Objekte die Schatten werfen liegen auf dem Caster-Layer und jene die den Schatten erhalten liegen im Receiver Layer.
Für den Receiver wird eine Plane erstellt, welche als ein Hintergrund gehandhabt wird.

Da es sich hier um ein simuliertes Schattierungssystem handelt, werden Sprites benötigt. Für jedes Sprite wird ein 3D-Objekt erstellt, welches sich hinter dem Sprite in der z-Achse befindet.

Es ist darauf zu achten, dass alle 3D-Objekte denselben z-Wert für die Position haben, da die Schatten ansonsten nicht vereint sind und Lücken bleiben oder sich Schatten zu hoch oder zu niedrig befinden.

Dabei kann es auch zu Unterschieden der Schattierung kommen, was meist nicht wünschenswert ist.

Die 3D-Objekte sollen nicht gerendert werden. Der Mesh-Renderer wird benötigt, da nur Schatten geworfen werden sollen.

Diese 3D-Objekte kommen in den Caster-Layer.

Die Objekte im Caster-Layer sollen cast shadows aktiviert haben und receive shadows deaktiviert.

Die bereits erwähnte Plane dient als unser Hintergrund und Schatten sollen darauf fallen. Daher wird diesem Objekt der Receiver-Layer zugewiesen.

In diesem Objekt sollte in den Eigenschaften cast shadows deaktiviert und receive shadows aktiviert sein.

Die Plane wird als child-Objekt an die main-Kamera gehängt und die Größe an der Kamera „viewpoint“ wird angepasst. Je nachdem wie dunkel die Farbe des Materials ist, welches auf die Plane kommt, desto stärker ist der Kontrast zwischen Licht und Schatten.

Es wird ein point light erstellt und zwischen den Caster und Receiver positioniert. Das Licht sollte von den Caster-Objekten verdeckt werden.

Die Culling Mask Option im Licht sollte alles deaktivieren außer Caster und Receiver.

Eine neue Kamera wird erstellt und als child an die main-Kamera gehangen. Diese neue Kamera sollte in den Werten genau gleich sein wie die main-Kamera.

Die Optionen hierbei sollten bei der Culling Mask ebenfalls alles deaktivieren außer Caster und Receiver.

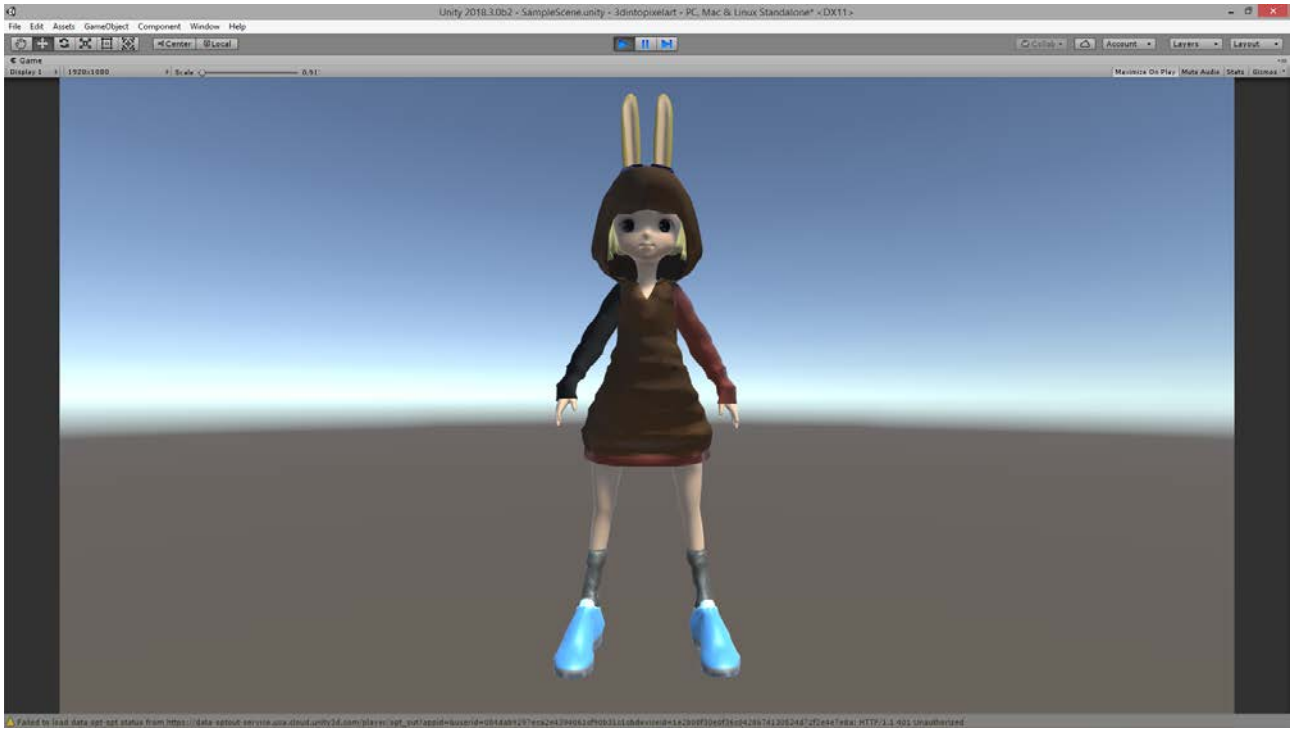
Des Weiteren wird eine Render-Textur benötigt, welche an die Kamera gehängt wird.

Die gerenderte Textur, welche von der child-Kamera erstellt wurde, kann nun auf die main-Kamera mithilfe eines Schaders geblendet werden.

Anschließend wird in der Option Culling Mask der main-Kamera alles aktiviert außer Caster und Receiver.

4. Implementierung

4.1 Erstellung eines Pixel-Art-Shader



(Normales Rendering)

Ein Shader besteht aus folgenden drei Komponenten: einem Script, einem Material und dem Shader selbst.
Der erstellte Shader simuliert die Reduktion der Pixelanzahl. Es wird ein Image-Effekt-Shader erstellt und ein Script namens „Pixelate“.

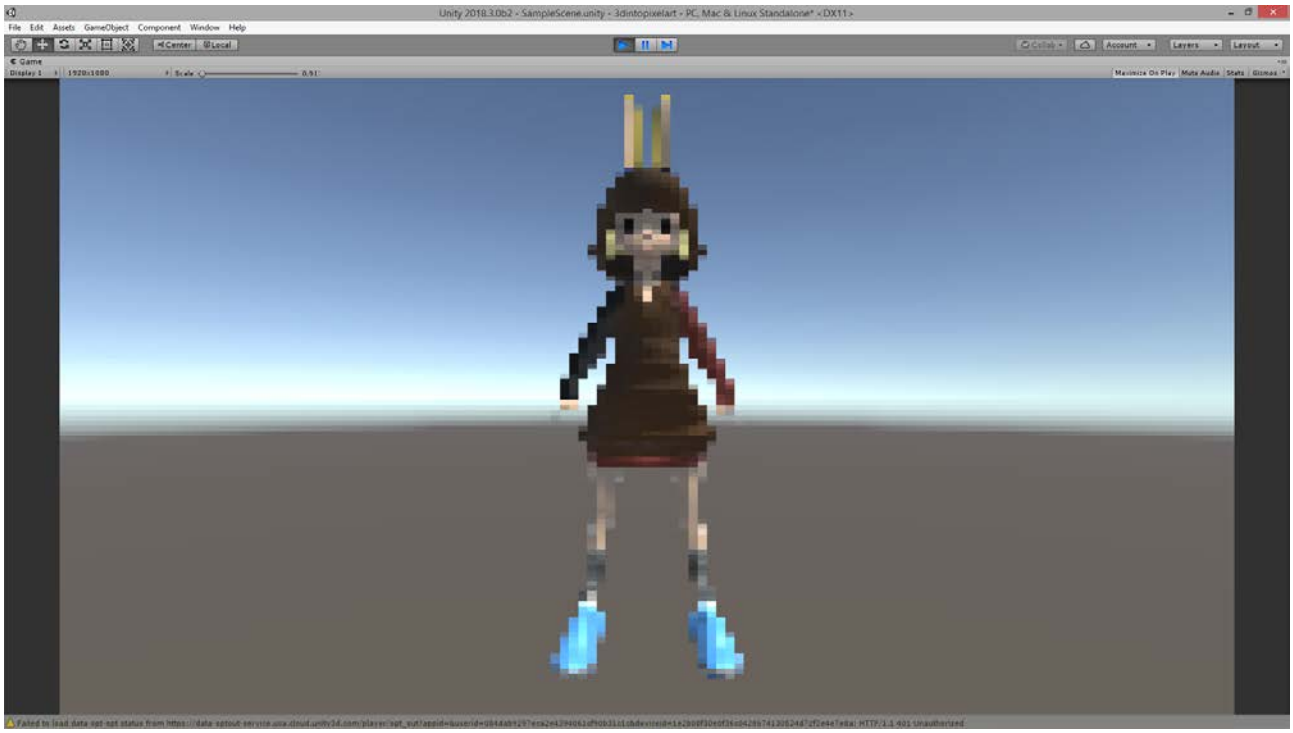
Es wird ein Material erstellt, welches den Shader hält.

```
public class Pixelate : MonoBehaviour
{
    public Material PixelateMaterial;



    private void OnRenderImage(RenderTexture source, RenderTexture destination)
    {
        Graphics.Blit(source, destination, PixelateMaterial);
    }
}
```

Das Script wird auf die Kamera gezogen. Das Material wird dem Script zugewiesen. Es wird ein invertiertes Bild dergestellt.

Innerhalb des ImageShader wird im fragment shader folgendes eingegeben:

```
fixed4 frag (v2f i) : SV_Target
            {
                        float2 uv = i.uv;
                        uv.x *= _PixelWidth;
                        uv.y *= _PixelHeight;
                        uv.x = round(uv.x);
                        uv.y = round(uv.y);
                        uv.x /= _PixelWidth;
                        uv.y /= _PixelHeight;

                fixed4 col = tex2D(_MainTex, uv);

                return col;
            }
```



(Verpixeltes Rendering 128x128 Pixels)

Es ist ein verpixeltes 3D-Objekt zu sehen.
Das jedoch reicht noch nicht aus, um die Illusion zu generieren, dass es sich hier um ein 2D-Objekt handelt.
Dafür wird noch ein zusätzlicher Shader benötigt, ein sogenannter Cel-Shader (oder auch Toonshader).

Beim Cel-Shading werden 3D-Objekte so gerendert, als ob es im Stil eines Zeichentrickfilms (eng. Cartoon) gezeichnet wäre.

Beim Cel-Shading werden die Schattierungen des 3D-Objektes, die normalerweise weich verlaufen in scharfkantige wenige Helligkeitsstufen konvertiert.
Diese sind meist in drei oder vier verschiedenen Abstufungen vorhanden: Weiß, ein helles Grau, ein dunkleres Grau und eventuell ein noch dunkleres Grau.
Auf konventionelle Texturen wird oft verzichtet und stattdessen eine einfarbige Textur benutzt.

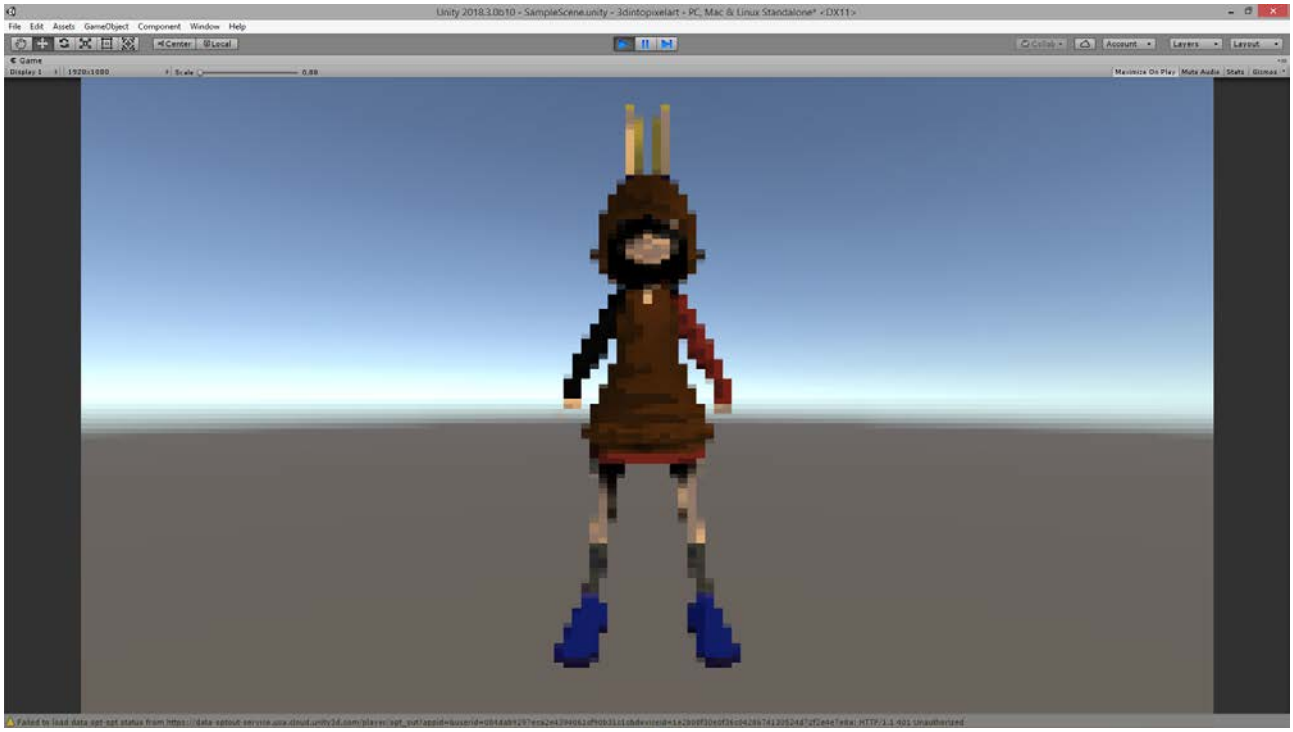
Der Lichtwert wird aus dem Skalarprodukt der Polygonnormalen und der Richtung des Lichtes berechnet.
Es ergibt sich ein normalisierter scaler, der die Position einer ramp-Textur auf der x-Achse angibt.

Eine ramp-Textur besteht aus mindestens zwei Farben, welche von einem dunklen Ton zu einem hellen verlaufen.
Es ist darauf zu achten, dass die Kanten scharf sind und nicht weich ineinander übergehen.
Anhand des normalisierten Wertes kann die gewünschte Schattierung angezeigt werden, da ein Punkt zwischen 0 und 1 ausgewählt wird. Hierbei ist 0 die äußerste rechte Seite der ramp-Textur und 1 die äußerste linke Seite.

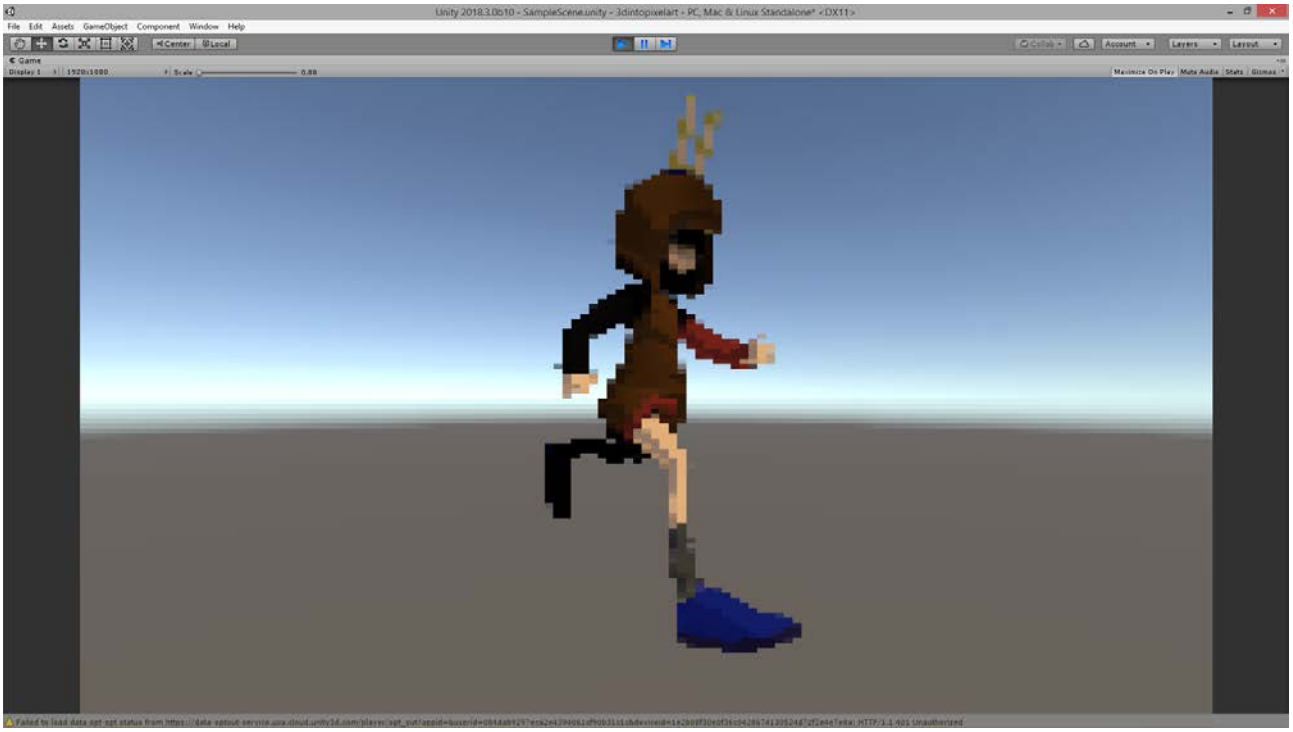
Dadurch kann der Shader erkennen, welche Polygone weniger Licht abbekommen und dadurch dunkler gerendert werden und welche mehr Licht abbekommen, und dadurch heller gerendert werden.

Anschließend wird die Schattenfarbe noch mit der albedo-Textur multipliziert. Nun werden scharfe Kanten zwischen den Schatten gerendert und es ergibt sich ein Cel-Shading-Effekt.

```
struct vertexInput
	{
		float4 vrtx : POSITION;
		float3 texCoord : TEXCOORD0;
		float3 nrml : NORMAL;
	};

	struct vertexOutput
	{
		float4 pos : SV_POSITION;
		float3 texCoord : TEXCOORD0;
		float3 nrml : NORMAL;
		LIGHTING_COORDS(1,2)
	};

	vertexOutput vert(vertexInput input)
	{
		vertexOutput output;


		output.pos = UnityObjectToClipPos(input.vrtx);
		float4 nrm4 = float4(input.nrml, 0.0);
		output.nrml = normalize(mul(nrm4, unity_WorldToObject).xyz);

		output.texCoord = input.texCoord;

		TRANSFER_VERTEX_TO_FRAGMENT(output);
		return output;
	}

	float4 frag(vertexOutput input) : COLOR
	{

		float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);


		float ramp = clamp(dot(input.nrml, lightDir), 0, 1.0);
		float3 lighting = tex2D(_RampTex, float2(ramp, 0.5)).rgb;


		float4 albedo = tex2D(_MainTex, input.texCoord.xy);

		float attenuation = LIGHT_ATTENUATION(input); // shadow value
		float3 rgb = albedo.rgb * _LightColor0.rgb * lighting * _Color.rgb *
		attenuation;
		return float4(rgb, 1.0);
	}
```

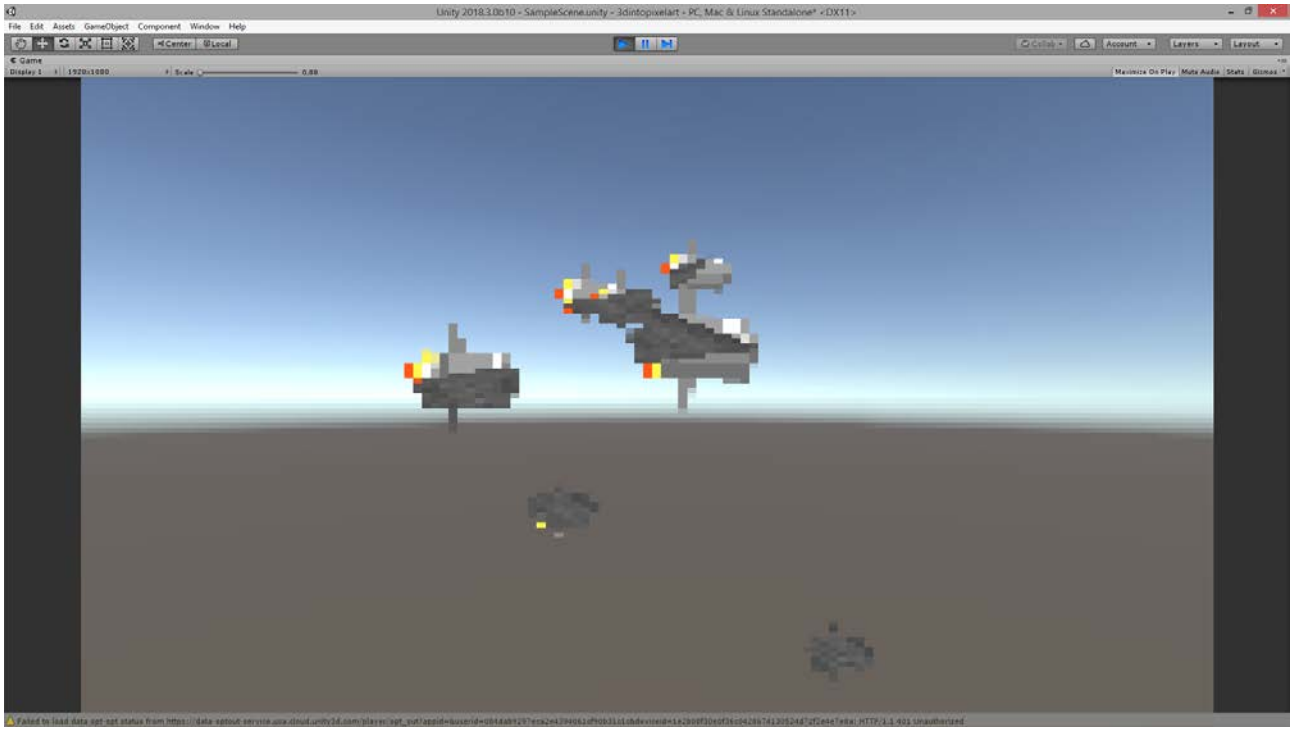


(Verpixeltes Rendering plus Cel-Shader)



(Verpixeltes Rendering plus Cel-Shading in Animation.)

Diese Shader lassen sich auch auf Partikeleffekte und andere Effekte übertragen.
Da der Shader zur Pixelation auf der Kamera liegt, wird jedes angezeigte Element im Bild automatisch verpixelt.
Daher muss lediglich der Cel-Schader auf den Effekt angehängt werden, um den Eindruck von Pixel-Art zu erwecken.
In diesem Beispiel wurden qualmende Raketen in einem Partikeleffekt erstellt.


(Verpixeltes Rendering plus Cel-Shading von Partikeleffekten.)

4.2 Schatten für 2D-Objekte

Schatteneffekte in Unity werden für 2D-Objekte, anders als für 3D-Objekte, nicht von Haus aus unterstützt. Eine Lichtquelle ignoriert das 2D-Objekt und das geworfene Licht dringt ungehindert hindurch.

In diesem Abschnitt werden unterschiedliche Lösungsansätze untersucht, die es ermöglichen einen Schatten für 2D-Objekte zu generieren.

Eine relativ einfache Lösung wäre ein zusätzliches 2D-Objekt unter dem bereits bestehenden zu erstellen, einen sogenannter DropShadow.

Hierbei ist zu beachten das der Schatten dieselbe Form besitzt und dem Hauptobjekt folgt.

Dies kann erreicht werden, indem das Objekt in einer konstanten dunklen Farbe mit einem Bildbearbeitungsprogramm erstellt und als child-Objekt an unser gewünschtes Objekt, welches einen Schatten haben soll angehängt wird. Jedoch kommt hierbei das Problem auf, dass bei jedem 2D-Objekt ein erneutes "Schatten"-Objekt erstellt werden muss.

Daher ist zu bedenken, dass ein Script und ein Shader die Arbeit dafür übernehmen können.

Das Script muss folgende Funktionen erfüllen:

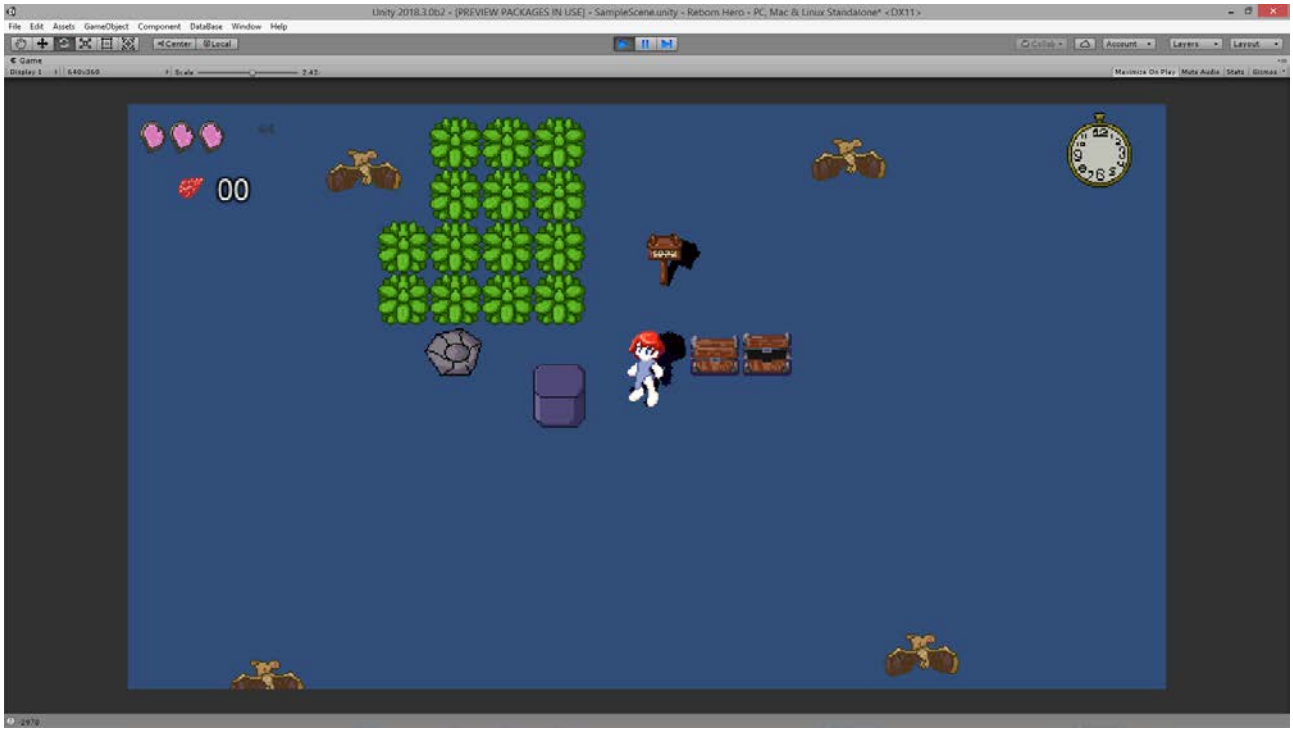
– Das Objekt, welches ein Schatten haben soll kopieren und an das Objekt als ein child-Objekt hängen

– Das neue Objekt in der Position versetzten, damit die Illusion eines Schattens entsteht

Der Shader muss folgende Funktion erfüllen:

– Das neue Objekt in einer konstanten Farbe darstellen

```
    void Start()
    {


        shadowObj.transform.localPosition = Offset;
        shadowObj.transform.localRotation = Quaternion.identity;
        shadowObj.transform.localScale = scale;

        spriteRenderer.sprite = SpriteR.sprite;
        spriteRenderer.material = material;
        spriteRenderer.sortingLayerName = SpriteR.sortingLayerName;
        spriteRenderer.sortingOrder = SpriteR.sortingOrder - 1;
    }


    void Update()
    {
        shadowObj.transform.localPosition = Offset;
        spriteRenderer.sortingOrder = SpriteR.sortingOrder - 1;



    }
}
```



(DropShadow Stationär)

Es wird nun ein Schatten in der Form des 2D-Objektes angezeigt, dennoch kann diese Methode

noch verbessert werden. Der Schatten ist momentan stationär, daher wird eine Methode herausgearbeitet in der sich der Schatten dynamisch bewegt. Dies könnte ermöglicht werden indem ein Gameobjekt erstellt wird, welches je nach Winkel zum schattenwerfenden Objekt die Rotation des Schattens beeinflusst. Dadurch könnte für Tag- und Nachtzyklen oder für stationäre Lichtquellen eine bessere Illusion des Schattens erzeugt werden.

Es wird zuerst ein Versuch unternommen, in dem der Winkel von der Lichtquelle mit dem des 2D-Objektes verrechnet wird. Dafür muss dem 2D-Objekt der Winkel der Lichtquelle bekannt sein.

Es wird ein GameObject als Lichtquelle in dem Script hinzugefügt. Die Lichtquelle in diesem Beispiel ist eine Sonne, welche sich um das Level bewegt. Die Lichtquelle ist ein child-Objekt von einem GameObject, welches langsam rotiert. Es wird nun der Rotationswinkel des parent-Objekts der Lichtquelle mit dem des Schattens verrechnet.

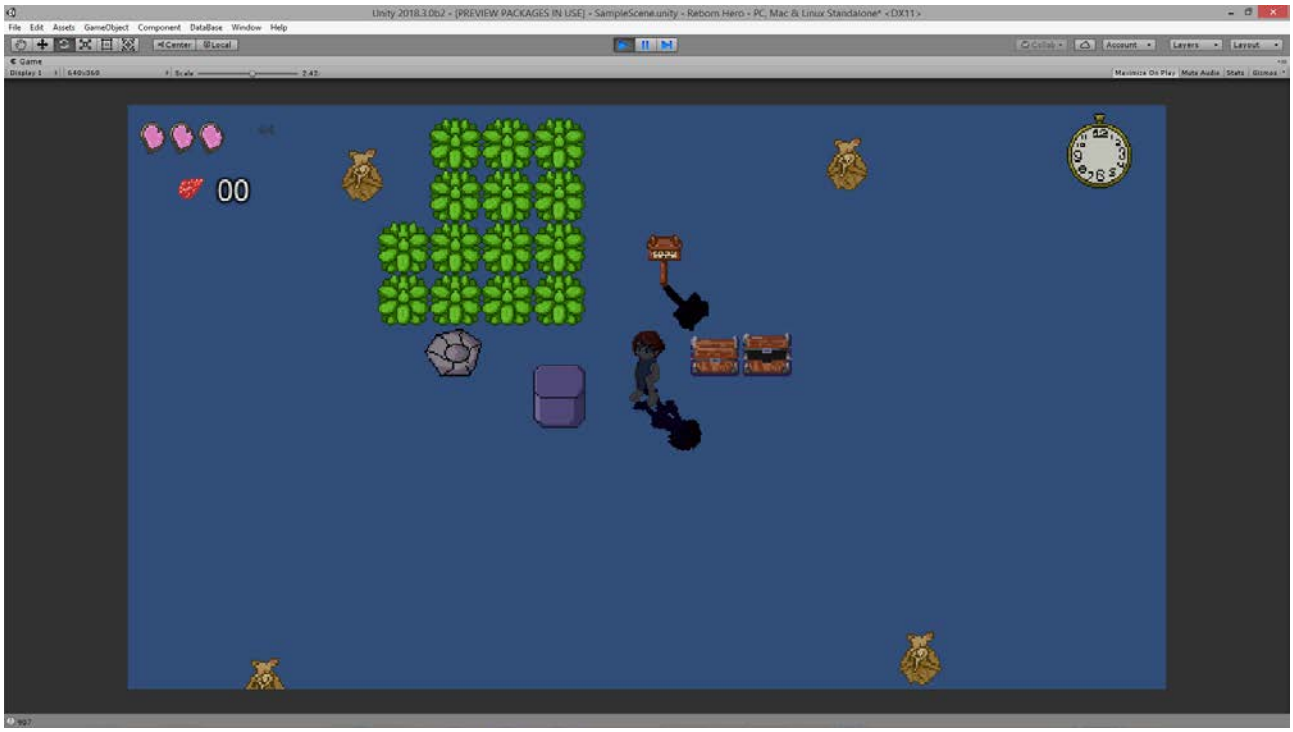
```
shadowObj.transform.eulerAngles =- (-(lightsource.transform.eulerAngles+
    correction));
```

Der Schatten rotiert nun im selben Winkel wie die Lichtquelle steht. Dies kann aber auch erweitert werden für stationäre Lichtquellen. Hier müsste das parent-Objekt der Lichtquelle lediglich zum 2D-Objekt zeigen und in die Richtung dessen rotieren.

```
 if ((lightsourceStationar.transform.position - this.transform.position).sqrMagnitude < 2 *
2)
        {
            Vector3 targetPos = player.position;
            Vector3 lightPos = lightsourceStationar.transform.position;
            targetPos.x = targetPos.x - lightPos.x;
            targetPos.y = targetPos.y - lightPos.y;
            float angle = Mathf.Atan2(targetPos.y, targetPos.x) * Mathf.Rad2Deg;

            lightsourceStationar.transform.rotation = Quaternion.Euler(new Vector3(0, 0,
            angle + offset));
            shadowObj.transform.eulerAngles =- (lightsourceStationar.transform.eulerAngles+
            correction);
```

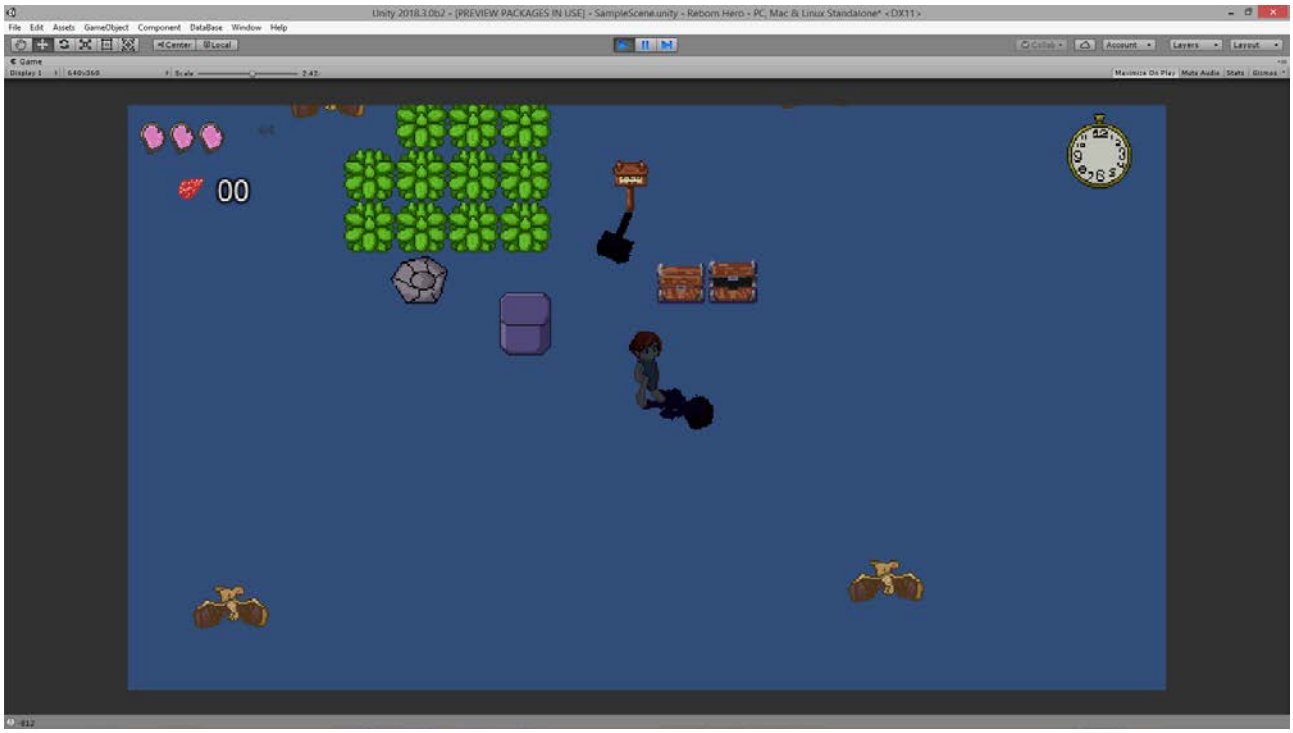
Der Schatten rotiert nun um das 2D-Objekt, jedoch wird dabei die Blickrichtung des Schattens nicht berücksichtigt. Daher wird an festgelegten Rotationspunkten das Sprite des Schattens gespiegelt, d.h. es erfolgt ein harter Übergang beim Wechseln von einer Blickrichtung zu einer anderen.

```
if (shadowObj.transform.eulerAngles.z >= 305 && shadowObj.transform.eulerAngles.z
<= 310)
         {
            scale.x = -1f;
        }else if(shadowObj.transform.eulerAngles.z <= 120 &&
      shadowObj.transform.eulerAngles.z >= 115)
        {

            scale.x = 1f;
        }
```



(Sonne als Lichtquelle, Schatten rotiert um das parent-Objekt)

Der Schatten auf dem Sprite wird durch ein externes Script erstellt, welches das Sprite mit der Farbe der Lichtquelle kombiniert. Der Lichtquelle kann zu bestimmten Positionswerten eine bestimmte Farbe zugewiesen werden, welche dadurch Einfluss auf das Sprite hat.



(Stationäre Lichtquelle beleuchtet den Spieler, während die Sonne das Schild beleuchtet)

Das stationäre Licht wird aufgerufen, wenn der Spieler sich innerhalb zweier Unity-Längeneinheiten befindet. Es wird die Sonne als Lichtquelle vorgezogen. Dies dient als Demonstrationzweck um verschiedene Lichtquellen zu veranschaulichen, wenn ein Sprite z.B. eine Höhle betritt und die Sonne verdeckt wird. Es besteht jedoch ein kleiner Fehler in der Art und Weise wie das stationäre Licht den Schatten wirft. Die Richtung in welche der Schatten fällt, ist auf der x-Achse invertiert. Auf der y-Achse wird diese jedoch richtig angezeigt.

Weiterhin würde die Illusion des Schattens verbessert werden, wenn das Scaling des Schattens anhand der Entfernung zu dem 2D-Objekt und der Lichtquelle angepasst wird.

Diese Art der Nutzung eines Schattens lässt jedoch einige Probleme offen, wie zum Beispiel, dass wenn das 2D-Objekt animiert ist es immer denselben Schatten darstellt.

Dieses Problem könnte gelöst werden, indem ein Script dynamisch das Sprite für den Schatten, anhand des momentan angezeigten Sprites des 2D-Objektes verändert.

Hierbei müsste das Script wissen, welcher Frame des 2D-Objektes momentan angezeigt wird und anhand dessen den Sprite des Schattens austauschen. Dafür wird der Name des Frames benötigt.

Die benötigten Parameter werden gesetzt und nach dem Namen des momentanen Frames wird gesucht:

```
public void SpriteUpdater()
    {
        string number = characterAnimationListener.AnimationNumber;



        m_CurrentClipInfo = anim.GetCurrentAnimatorClipInfo(0);
        clipname = m_CurrentClipInfo[0].clip.name;
        Debug.Log("clipname" + clipname);
      }
}
```

Es wird nun in der Konsole der Name des momentan angezeigten Frames angezeigt.

Die Klassen `CharacterAnimationListener` und `CharachterMovementModel` sind lediglich dafür zuständig das 2D-Objekt zu finden und zu bewegen.

Als nächstes müssen die Sprites ausgetauscht werden.

Es soll ein anderer Frame gewählt werden, wenn der Name des Frames sich verändert. Wenn also der Frame in der Animation sich verändert, soll derselbe Frame für den Schatten gewählt werden. Erreicht wird dies dadurch, dass der Name des Frames ausgelesen und in ein lesbares Format umgeändert wird. Dieses wird anschließend genutzt um den Namen des gewünschten Frames zu erhalten.

Dies wäre soweit benutzbar, jedoch ist eine Erweiterung vorteilhaft. In dieser soll es egal sein, welcher Spritesheet für den Schatten genutzt wird.

Durch diese Methode müsste jeder Frame den identischen Namen besitzen wie der für welchen er ersetzt werden soll. Um das zu beheben kann eine von Unity nützliche Funktion angewandt werden.

Immer wenn ein Spritesheet mit mehreren Sprites in Unity angelegt wird, schneidet Unity nach automatisch oder nach Gridsize-Angabe die Sprites aus und erstellt automatisch Namen für die nun

neuen Sprites.

Die Namen sind die des Spritesheets plus einer Nummer. Diese Nummer kann isoliert werden und für das Finden des richtigen Sprites genutzt werden.

```
public static int IntParseFast(string value)
    {
        int result = 0;
        for (int i = 0; i < value.Length; i++)
        {
            char letter = value[i];
            result = 10 * result + (letter - (n));  //n ist die anzahl der
            Characters in dem String
        }
        return result;
  }
```

In dieser Funktion wird der String des Framenamens auf n Zeichen gekürzt. Es bleibt eine Zahl übrig, jedoch liegt diese noch als String vor. Deswegen wird in dieser Funktion ebenfalls aus dem String ein Integer-Wert erstellt.

Dieser Integer-Wert kann nun als Indikator für das Sprite, welches getauscht werden soll genutzt werden.

Die SpriteUpdater-Funktion wird um folgendes erweitert:

```
int oldint = myInt;
myInt = IntParseFast(number);

Debug.Log("oldclipname" + oldClipName);

        if (myInt == oldint)
        {
            return;
        }
        else
        {


            if(clipname != oldClipName)
            {

                transform.Find("Sprite").GetComponent<SpriteRenderer>().sprite =
                sprites[myInt];
                Debug.Log("Ungleich");
                oldClipName = clipname;

            }
            else
            {

                Debug.Log("Gleich");
            }

    }
```

Ein neuer Parameter namens „myInt“ wurde hinzugefügt. Dieser enthält die Information des konvertierten Strings. Es wird nun geprüft ob der Frame sich verändert hat. Wenn dies der Fall ist, soll das Sprite ausgetauscht werden.

Dies funktioniert jedoch nur bedingt. Es ist eine Verzögerung zu erkennen welche unerwünscht ist.

Die Verzögerung könnte durch die von dem Script benötigte Zeit zum parsen des Namensstrings verursacht werden.

Dieses Problem könnte behoben werden, indem alle Sprites auf dem Spritesheet für den Schatten um eine Position nach hinten verschoben werden. Da nun jedoch immer ein falsches Sprite angezeigt wird, soll die SpriteUpdate-Funktion eine Verzögerung erhalten.

```
IEnumerator CreateSwapDelay(float SwapDelay)
    {
        yield return new WaitForSeconds(SwapDelay);
        transform.Find("Sprite").GetComponent<SpriteRenderer>().sprite =
        sprites[myInt];
        oldClipName = clipname;
  }
```

In der Delay-Funktion wird nun das Sprite geändert. Die Zeit für den Austausch ist in dem float-Wert „SwapDelay“ vertreten und wird abgewartet. Für diesen Fall funktionierte der Wert 0.14.

Die SpriteUpdater Funktion ändert sich wie folgt:

```
 public void SpriteUpdater()
    {
        string number = characterAnimationListener.AnimationNumber;

        int oldint = myInt;
        myInt = IntParseFast(number);
        m_CurrentClipInfo = anim.GetCurrentAnimatorClipInfo(0);
        clipname = m_CurrentClipInfo[0].clip.name;
        Debug.Log("clipname" + clipname);
        Debug.Log("oldclipname" + oldClipName);
        if (myInt == oldint)
        {
            return;
        }
        else
        {
            if(clipname != oldClipName)
            {


                Debug.Log("Ungleich");
                oldClipName = clipname;

            }
            else
            {
            StartCoroutine(CreateSwapDelay(SwapDelay));
                Debug.Log("Gleich");
            }

        }

  }
```

Der Sprite des Schattens wird nun zum richtigen Zeitpunkt ausgetauscht, jedoch entstehen merkliche Fehler, wenn das 2D-Objekt bewegt wird.

Durch die Verzögerung in der Funktion `CreateSwapDelay` wird ein falsches Sprite angezeigt. Um dies zu beheben wird vermutet, dass das originale Spritesheet anstatt des verschobenen wieder benutzt werden kann, in dem Moment, wenn das 2D-Objekt bewegt wird. In der Update-Funktion wird folgendes ergänzt:

```
m_movementDirection = cMModel.GetDirection();

        if (clipname == "PlayerLeftDown" && !swapAble)
        {
            transform.Find("Sprite").GetComponent<SpriteRenderer>().sprite =
            StartSprites[0];
            Debug.Log("0");
            oldClipName = clipname;
            swapAble = true;
                    return;
        }
        else if (clipname == "PlayerRightDown" && !swapAble)
        {
            transform.Find("Sprite").GetComponent<SpriteRenderer>().sprite =
            StartSprites[2];
            Debug.Log("2");
            oldClipName = clipname;
            swapAble = true;
                    return;
        }
        else if...
```

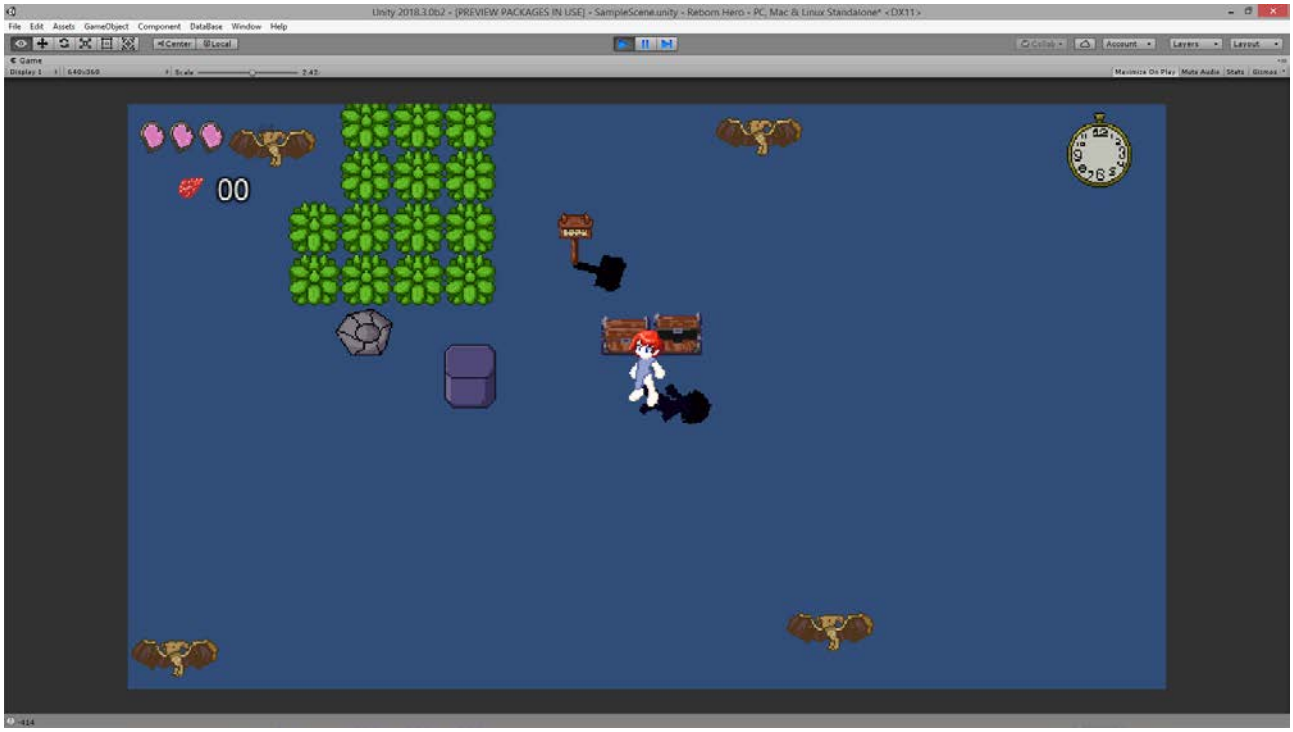
Es wurde zuerst versucht anhand des eingegebenen KeyCode und dem Wert der movementDirection den Spriteaustausch zu erwirken.

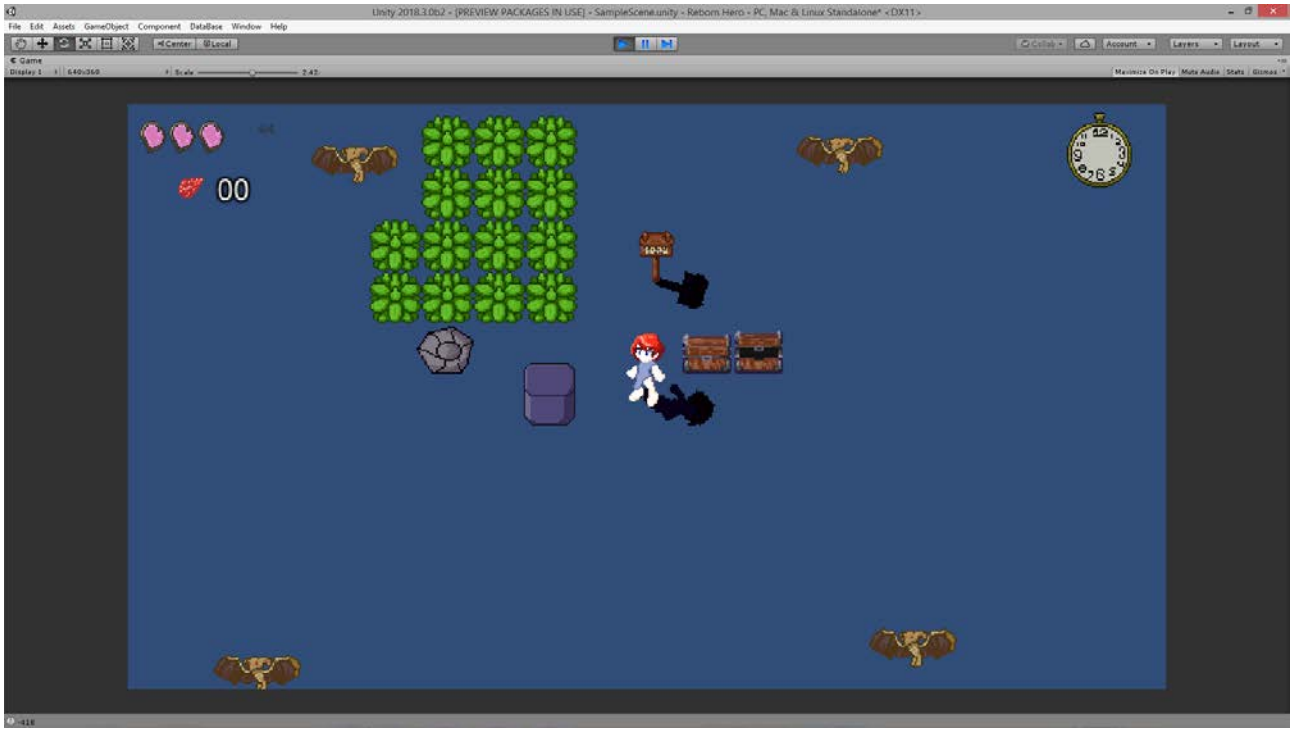
Die movementDirection ist nötig, da hier anstatt den Sprite anhand der ermittelten Nummer zu tauschen spezifisch ein Sprite als Input genutzt wird.

Jedoch hat diese Herangehensweise nicht funktioniert, der Spriteaustausch fand nicht statt.

Eine alternative Idee war, direkt anhand des Namens des momentan abgespielten clips im Animator zu tauschen. Dies führte dazu das durchgehend dasselbe Sprite angezeigt wurde, weswegen ein bool namens `swapAble` hinzugefügt wurde.

In der SpriteUpdater-Funktion wurde die Zeile „`swapAble = false`“ hinzugefügt. Dies verhindert das Aufrufen der falschen Methode.


(Animierter Schatten 1)


(Animierter Schatten 2)

Dennoch kann mit diesem Script jedes beliebige 2D-Objekt anhand eines „MainObject“ animiert werden, ohne Animator und Animationsclips zu erstellen. Daher kann dieses Script auch beispielsweise um Equipment, Skins, Kleidung, Environment-Sprites und mehr erweitert werden.

Die DropShadow-Variante ist eine einfache und gutaussehende Variante um Schatten für 2D-Objekte zu generieren.

Das Script erfüllt seine gedachte Aufgabe nicht zufriedenstellend. Dennoch sind die Einsatzmöglichkeiten, wenn es optimiert wird, vielseitig.

5. Evaluation

5.1 Auswertung der theoretischen Methoden

Rotoskopie in der Art wie es DeadCells vorgemacht hat ist sehr beeindruckend und würde bei der Erstellung von 2D-Assets jede Menge Zeit ersparen. Jedoch bedarf es einem speziellen Programm oder Script um denselben Anforderungen gerecht zu werden wird.

Das Bearbeiten von Animationen ist in der Regel bei 2D-Assets ein langwieriger Prozess und bringt großen Aufwand mit sich, da selbst bei nur geringen Veränderungen meist alle Frames angepasst werden müssen. Bei der oben genannten Technik wird dieses Problem umgangen. Es muss lediglich die 3D-Animation an den gewünschten Frames angepasst werden und anschließend durch das spezielle Programm erneut konvertiert werden.

Dies ermöglicht auch eine größere Varianz an möglichen Animationen, da diese schneller und kostengünstiger zu erstellen sind. Des Weiteren können bereits erstellte Animationen für mehrere Objekte genutzt werden, ohne diese neu erstellen zu müssen.

Auch können dadurch unterschiedliche Assets auf ein 3D-Objekt gezogen werden. Beispielsweise wird ein Schwert auf einem Objekt durch eine Axt ersetzt und erzeugt somit ein neues Asset, womit es ein Leichtes wird mit wenigen Grundbausteinen eine Vielzahl von unterschiedlichen 2D-Animationen zu erstellen.

Diese Methode hat eine Vielzahl von Vorteilen und es kann davon ausgegangen werden, dass sie in Zukunft des Öfteren von unterschiedlichen Game Development Studios genutzt wird, um kostengünstig und vereinfacht 2D-Spiele zu erstellen.

Das Erstellen von Schatten in einem 2D-Spiel mithilfe von 3D-Objekten ist nicht besonders reizvoll. Obwohl damit stationäre Elemente wie Wände und Bäume, als Caster von Schatten funktionieren können, bleibt dies bei animierten und beweglichen Objekten aus. Der Aufwand hierfür ist ebenfalls groß. Für jedes 2D-Objekt muss ein 3D-Caster erstellt werden. Daher ist diese Methode nur bedingt nutzbar. In einem zufällig generierten Dungeoncrawler, wo alle Elemente bereits als Prefabs vorhanden sind, könnte dies nützlich sein. Für Außenwelten und von Hand geschaffenen Levels jedoch ist diese Methode nicht wünschenswert. Dies ist daher-gehend nicht wünschenswert, dass Schatten und Lichtquellen statisch sind, da diese nicht während der Laufzeit, sondern vorher erstellt werden müssen. In einer Außenwelt wird oft ein Tag- und Nachtzyklus angewandt und ist somit von dynamischen Schatten abhängig. Des Weiteren sind Objekte wie Bäume oder Gras oft animiert und können von statischen Schatten nicht berücksichtigt werden.

Bei der Benutzung in von Hand platzierten Leveln müssten nach jeder Veränderung die Schatten und Lichter neu erstellt werden, was auf langer Sicht viel Zeit in Anspruch nehmen würde.

5.2 Auswertung des Pixel-Art Proof of Concept

Das Erstellen von Pixel-Art-Grafiken durch einen Verpixel-Shader plus einen Cel-Shader hat sehr zufriedenstellende Ergebnisse gebracht.

Nicht nur kann damit jedes beliebige 3D-Objekt, sondern ebenso tadellos Effekte als Pixel-Art-Grafiken darstellen werden.

Mit dieser Methode kann deutlich Zeit eingespart werden und externe Tools sind nicht von Nöten um die Grafiken zu erstellen. Ebenfalls sind Schattierungen automatisch vorhanden und müssen nicht nachgearbeitet oder zusätzliche Texturen erstellt werden.

Das Erstellen der Shaders und Scripts ist relativ einfach und nach einmaliger Erstellung können diese durchgehend weiterbenutzt werden.

Es vereinfacht das Erschaffen von Animationen, da einzelne Frames nicht von Hand gezeichnet werden müssen.

Weiterhin ist das Bearbeiten dieser Animationen vereinfacht, da lediglich an der gewünschten stelle der 3D-Animation eine Veränderung vorgenommen werden muss. im Gegensatz dazu stehen konventionelle 2D-Animation, bei denen meist alle Frames neu erstellt werden müssen.

Ebenfalls bietet sich mit dieser Methode an, eine erstellte Animation auf mehrere Objekte zu legen, was eine große Zeiteinsparung bedeutet.

Durch den Verpixelungs-Shader kann ebenfalls die gewünschte Verpixelung angepasst werden; Ob nun eine 32x32, eine 64x64 oder sonst beliebige Auflösung gewünscht ist.

Der größte Vorteil mit dieser Methode ist jedoch, dass nicht jedes Objekt einzeln bearbeitet werden muss, sondern die gesamte Szene als Pixel-Art dargestellt wird.

5.3 Auswertung der Schattierung bei 2D-Objekten Proof of Concept

Die Schattierung von 2D-Objekten hat noch Raum für Verbesserungen. Zum einen könnte noch ein Skalierungsfaktor hinzugefügt werden, bei welchem der Schatten je nach Abstand zu einer Lichtquelle größer oder kleiner wird.

Auch ist die Rotation des Schattens nicht zufriedenstellend. Obwohl die Rotation an sich funktioniert kann erkannt werden, dass das spiegeln des Schatten-Sprites zu ruckartig geschieht. Hier könnte verbessert werden, dass der Schatten sich langsam skaliert und dabei spiegelt. Jedoch muss beachtet werden, dass der Schatten nicht zu dünn oder klein wird.

Auch das Animieren des Schattens ist nicht zufriedenstellend. Es ist durchgehend eine Verzögerung vorhanden, obwohl auf dieses Problem eingegangen wurde. Die Animation gerät zu oft aus der Synchronisation mit dem parent-Objekt. Obwohl die Verzögerung auf ein Minumum reduziert werden konnte, fällt dieses Problem sofort auf.

Hierbei ist zu überlegen, ob eventuell ein Shader in Kombination mit einem Script, welches die momentane Animation des parent-Objektes ausliest die Übergabe handhaben könnte.

Trotz all dieser Mängel hat dieses Vorgehen Potential. Der Schatten kann auf jedes beliebige 2D-Objekt angewendet werden, ohne Anpassungen jedweder Natur vornehmen zu müssen. Durch Ausbesserung der vorhandenen Mängel, könnte dies ein mächtiges Tool für Unity werden.

6. Quellen

Shader Grundlagen:

https://gamedevelopment.tutsplus.com/tutorials/a-beginners-guide-to-coding-graphics-shaders--cms-23313 (zuletzt geöffnet 13.02.2019)

https://www.html5rocks.com/en/tutorials/webgl/shaders/(zuletzt geöffnet 13.02.2019)

https://docs.unity3d.com/Manual/SL-BuiltinFunctions.html(zuletzt geöffnet 13.02.2019)

https://digitalerr0r.wordpress.com/2015/09/02/unity-5-shader-programming-1-an-introduction-to-shaders/(zuletzt geöffnet 13.02.2019)

https://docs.unity3d.com/Manual/SL-BuiltinFunctions.html(zuletzt geöffnet 13.02.2019)

Grafics Pipeline:

https://docs.microsoft.com/en-us/windows/desktop/direct3d11/overviews-direct3d-11-graphics-pipeline(zuletzt geöffnet 13.02.2019)

http://deacademic.com/dic.nsf/dewiki/538458(zuletzt geöffnet 13.02.2019)

Kürzung: deacedic

Cel-Shading:

https://de.wikipedia.org/wiki/Cel_Shading(zuletzt geöffnet 13.02.2019)

https://lindenreid.wordpress.com/2017/12/19/cel-shader-with-outline-in-unity/(zuletzt geöffnet 13.02.2019)

Erstellen einer DeadCell Pipeline in Unity: https://www.youtube.com/watch?v=iNDRre6q98g (zuletzt geöffnet 13.02.2019)

Erklärung der DeadCell Pipeline: https://www.gamasutra.com/view/news/313026/Art_Design_Deep_Dive_Using_a_3D_pipeline_for_2D_animation_in_Dead_Cells.php (zuletzt geöffnet 13.02.2019)

Bücher:

A. Nischwitz et al., *Computergrafik und Bildverarbeitung*, DOI 10.1007/978-3-8348-8323-0_12, © Vieweg+Teubner Verlag | Springer Fachmedien Wiesbaden GmbH 2011

Kürzung : Nischwitz CB