UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Rendering von blickabhängigen Reflexionen auf der Grafikkarte

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Matthias Schnorr

Erstgutachter:	Prof. Dr. Stefan Müller (Institut für Computervisualistik, AG Computergrafik)
Zweitgutachter:	M.Sc. Bastian Krayer (Institut für Computervisualistik, AG Medizinische Visualisierung)

Koblenz, im März 2020

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

In der Computergrafik stellte die Berechnung von Reflexionen lange ein Problem dar. Doch mit der ständigen Weiterentwicklung der Hardware und Vorstellung neuer Verfahren ist eine realitätsnahe, echtzeitfähige (durchschnittlich 60 FPS) Berechnung von Reflexionen möglich.
In der folgenden Ausarbeitung werden verschiedene Reflexionsverfahren vorgestellt. Alle mathematischen und physikalischen Grundlagen werden gegeben, um die Algorithmen nachvollziehen zu können. Da eine Reflexion immer das Abtasten eines reflektierten Vektors bedeutet, werden zwei verschiedene Abtastungsverfahren für blickabhängige Reflexionen vorgestellt und anschließend implementiert. Zuletzt werden die Verfahren auf Basis von Qualität und Performance gegenübergestellt.

Abstract

In computer science, the rendering of reflections has long been a problem. But with the constant development of hardware and the introduction of new methods, a realistic, real-time (average of 60 FPS) calculation of reflections is possible.
In the following thesis different reflection methods are presented. All mathematical and physical basics are given in order to be able to reproduce the method. Since a reflection always means the sampling of a reflected vector, two different sampling methods for view-dependent reflections are introduced and then implemented. Finally, the methods are compared on the basis of quality and performance.

Inhaltsverzeichnis

1 Einleitung

Die folgende Abschlussarbeit erläutert verschiedene Verfahren zum korrekten Rendern von Reflexionen in Echtzeit. Dabei werden zwei unterschiedliche Abtastungsverfahren implementiert und verglichen.

1.1 Motivation

Reflexionen in der realen Welt sind überall zu jeder Tageszeit zu sehen – sei es auf einem Fluss oder in Fenstern (vgl. Abbildungen 1a und 1b). Dabei ist eine Reflexion nicht immer perfekt wie bei einem Spiegel, sondern oft verrauscht durch die Oberfläche oder auch verkrümmt, auch andere physikalische Effekte treten auf. Dementsprechend ist es erwünscht, dass qualitativ hochwertige Reflexionen auch in Grafikanwendungen wie beispielsweise Videospiele implementiert werden. Ein Beispiel für eine qualitativ gute Reflexion ist in Abbildung 1c zu sehen.
Echtzeitfähige Reflexionen waren in Grafikanwendungen die meiste Zeit zu rechenaufwändig und wurden daher selten verwendet. Die vorhandenen Verfahren waren entweder sehr verrauscht oder reflektierten nur gewisse Objekte in der Szene oder wurden nur auf einzelne kleine Flächen (z.B. Spiegel) angewandt. Aufgrund der Entwicklung der Hardware in den letzten Jahren können mehr Aufrufe parallel und mit größerem Speicher ausgeführt werden.

[0] `https://expertphotography.com/creative-reflections-in-street-photography/`, aufgerufen am 20.02.2020

(a) Ein Beispiel für Reflexionen im Alltag anhand von einem reflektierenden Fluss und reflektierenden Gebäuden, entnommen von [0]

(b) Ein Foto als Beispiel für Reflexionen und Fresnel Effekt im Alltag, entnommen von [0]

(c) Reflexion aus dem Videospiel „Satisfactory“

Abbildung 1: Beispiele für Reflexion in realer und digitaler Welt

1.2 Ziele

Das Hauptziel dieser Ausarbeitung ist es, ein effizientes und reproduzierbares Verfahren für blickabhängige Reflexionen zu erarbeiten. Außerdem soll ein Programm entwickelt werden, welches eine realitätsnahe, echtzeitfähige Qualität vorweisen soll.
Um einen besseren Einblick in die Thematik zu erhalten, werden mehrere Grundlagen und gängige Reflexionsverfahren in den nachfolgenden Kapiteln erläutert. Die anschließend näher ausgearbeiteten Abtastungsverfahren – View und Screen Space Abtastungen – werden in den darauf folgenden Abschnitten in ihrer Implementation beschrieben. Dies dient insbesondere der Reproduzierbarkeit der Testergebnisse, die im vorletzten Kapitel vorgestellt werden. Die anschließende Bewertung der Ergebnisse bezüglich ihrer Performance und Qualität dienen zur Überprüfung der gestellten Ziele. Außerdem wird auf die Grenzen der Verfahren und die möglichen Verbesserungen eingegangen.

2 Grundlagen und bisherige Arbeiten

Im folgenden Kapitel werden alle für diese Arbeit grundlegenden Verfahren zum Berechnen und Rendern von Reflexionen erläutert. Dazu gehört Grundwissen über die verwendeten Koordinatensysteme und deren aufgespannte Räume und eine Übersicht an bereits gängigen Reflexionsverfahren.

2.1 Räume

Die Computergrafik setzt verschiedene Räume (engl.: spaces) voraus, um beispielsweise die Beleuchtung zu berechnen. Um von dem Wertebereich der globalen Positionen auf die zugehörigen Pixelkoordinaten zu schließen, müssen mehrere Koordinatentransformationen angewendet werden. Positionen $\vec{p}$ werden dabei wie folgt umgerechnet[1] :

Local Space Die Koordinaten geben die Position und Ausrichtung des Objekts relativ zu seinem lokalen Ursprung an. Dieser dient als Input für den Vertex Shader und ist der aus der 3D-Datei gelesene Wert.

- Wertebereich: $]-\infty;\infty[$

World Space Die Koordinaten geben Position und Ausrichtung relativ zum globalen Ursprung der gesamten Szene an. Diese Positionen können miteinander verglichen werden, da die verschiedenen Modelle nun in demselben Koordinatensystem liegen.
Außerdem werden die Werte für Cubemap Reflexionen, welche in Abschnitt 3.1 näher beschrieben werden, in World Space berechnet.

- Wertebereich: $]-\infty;\infty[$
- Transformation: $\vec{p_w} = \mathbf{M}_{\text{model}} \cdot \vec{p_l}$

View Space Die Koordinaten sind relativ zur Kamera, wessen Position dadurch den Ursprung darstellt. Die Orientierung ist durch die Blickrichtung der Kamera gegeben. Dies entspricht der negierten z-Achse, da es sich um ein rechtshändiges Koordinatensystem handelt. Dementsprechend zeigt die x-Achse nach rechts und die y-Achse oberhalb der Kamera. Die Werte werden in View Space für das View Space Abtastungsverfahren transformiert, welches in Abschnitt 3.3 erklärt wird. Dieses Koordinatensystem reicht für die meisten Berechnungen (bspw. Phong) aus.

[1] `https://learnopengl.com/Getting-started/Coordinate-Systems` aufgerufen am: 20.02.2020

- Wertebereich: $]-\infty;\infty[$
- Transformation: $\vec{p_v} = \mathbf{M}_{\text{view}} \cdot \vec{p_w}$

Camera Space Camera Space wird auch Clip Space genannt, da ab diesem Space nur noch der relevante, sichtbare Teil der Szene, welcher sich innerhalb des View Frustums befindet, berechnet wird. Durch die Projektion liegt der Wertebereich zwischen der negativen und positiven homogenen Komponente $(-w; w)$. Außerdem ist durch die perspektivische Projektion die Linearität der z-Achse mit steigender Distanz zur Kamera nicht mehr gegeben.

- Wertebereich: $[-w; w]$
- Transformation: $\vec{p_c} = \mathbf{M}_{\text{projection}} \cdot \vec{p_v}$

Normalized Device Coordinate Space Durch das Teilen der homogenen Koordinaten wird die vierdimensionale Camera Space Position $\vec{p_c}$ auf die dreidimensionale Normalized Device Coordinate Space Position $\vec{p}_{ndc}$ projiziert. Der Wertebereich liegt dementsprechend zwischen -1 und 1 und ist von dem Ausgabegerät / Monitor unabhängig. Letzteres ist besonders wichtig, falls die Maps für SSR eine andere Auflösung besitzen als das Ausgabefenster.

- Wertebereich: $[-1; 1]$
- Transformation: $\vec{p_{ndc}} = \frac{\vec{p_c}.xyz}{\vec{p_c}.w}$

Screen Space Durch das Transformieren in den positiven Wertebereich können die Werte als Texturzugriffe verwendet werden. Dabei ist es wichtig, dass nur die xy-Koordinaten verwendet werden. Screen Space wird außer für SSR auch für Screen Space Ambient Occlusion (SSAO) und andere Post-Processing Effekte verwendet. Außerdem existiert ein Tiefenwert, welcher im Gegensatz zu anderen z-Werten nicht linear ist.

- Wertebereich: $[0; 1]$
- Transformation: $\vec{p_s} = 0.5 \cdot \vec{p_{ndc}} + 0.5$

Richtungen werden anders transferiert: Anstelle mit Matrix M multiplizieren (wie bei Positionen) wird mit der transponierten Inversen $(M^{-1})^T$ multipliziert.

2.2 Physikalische Reflexion

Um die Reflexion möglichst realistisch auf der Grafikkarte zu berechnen, muss zunächst die grundlegende Physik der Reflexion analysiert werden. Im Folgenden liegt der Fokus ausschließlich auf der Reflexion von dem Menschen wahrnehmbarem Lichtspektrum. Dessen Wellenlänge beträgt im Durchschnitt zwischen 380 nm und 740 nm. Wellenlängen oberhalb (Infrarot) und unterhalb (UV) dieses Spektrums und Wellen, die ein anderes Medium als Photonen verwenden (Schall), werden in dieser Arbeit nicht betrachtet.
Ohne Licht kann der Mensch seine Umgebung nicht visuell wahrnehmen. Selbst die Farben aller Objekte sind das Resultat aus der Reflexion des eingehenden Lichts. Wenn beispielsweise die weißen Sonnenstrahlen auf eine Mohnblume treffen, dann absorbiert die Blüte sämtliche Lichtstrahlen bis auf das rote Licht, welches reflektiert und vom Menschen wahrgenommen wird.

Eine perfekt spekulare Reflexion (mirror reflection) setzt eine zu 100% glatte Oberfläche voraus, wie in Abbildung 2 angenähert. Alle eingehenden Strahlen werden gleich reflektiert und dementsprechend wird die gesamte Energie während der Reflexion beibehalten. Da die physikalischen Eigenschaften entlang jeder Achse identisch sind, handelt es sich dabei um

[2] `https://www.fotocommunity.de/photo/spiegelung-auf-dem-see-sidlangen-carmen-stegenitz/18114199`, aufgerufen am: 20.02.2020

Abbildung 2: Beinahe perfekte Reflexion in einem See, entnommen von [2]

eine isotropische Reflexion (isos = "gleich", tropos = "Richtung").
In der Realität sind Materialoberflächen nie perfekt glatt und besitzen daher eine gewisse Unebenheit, wodurch die reflektierten Strahlen gestreut werden. Die daraus resultierenden glossy Reflexionen sind diffuser und verlieren einen Teil ihrer Energie, da nur gewisse Teile des Farbspektrums die Oberfläche verlassen. Sollten sich außerdem Unregelmäßigkeiten in der Oberfläche befinden, welche je nach Blickwinkel unterschiedlich reflektieren, ist eine gewisse Abhängigkeit zwischen den physikalischen Eigenschaften und dem Betrachterwinkel vorhanden und es handelt sich um eine anisotropische Reflexion. Eine Gegenüberstellung der verschiedenen Reflexionsarten ist in Abbildung 3 zu sehen.
In dieser Arbeit werden ausschließlich isotropische Reflexionen betrachtet, da das Beleuchtungsmodell nach Phong [4] keine anisotropischen Reflexionen unterstützt.

[3] `https://cs184.eecs.berkeley.edu/sp16/lecture/reflection/slide_022`, aufgerufen am: 20.02.2020

Abbildung 3: Eine isotropische und anisotropische Kugel im Vergleich, entnommen von [3]

2.3 Reflexion Mathematische Grundidee

Die grundlegende Mathematik der Verfahren lässt sich auf folgenden Ablauf reduzieren: Zunächst wird aus der Sicht der Kamera die Blickrichtung $\vec{v}$ ermittelt und anschließend an der Normalen $\vec{n}$ des reflektierenden Objekts reflektiert. [2] Die Berechnung des resultierenden Reflexionsvektors $\vec{r}$ ist in Gleichung 1 beschrieben.

$$\vec{r} = \vec{n} - 2.0 \cdot \vec{n} \circ \vec{v} \cdot \vec{v} \tag{1}$$

Der Vektor $\vec{r}$ gibt den reflektierten Punkt an, der anschließend auf der Oberfläche zu sehen ist. Es handelt sich dabei um kein komplett physikalisch korrektes Verfahren, da die Energieerhaltung nicht beachtet wird. Außerdem wird der Strahl entgegengesetzt des eigentlichen Pfades der Photonen verwendet.[6]

2.4 Fresnel

Wie auf Abbildung 4 zu sehen ist, wird je senkrechter der Betrachter auf die reflektierende Oberfläche schaut, desto weniger wird von der Reflexion wahrgenommen. Das Material gibt dabei den Faktor an, mit welchem das Licht eher reflektiert oder gebrochen wird.
Die Ursache für dieses physikalische Phänomen ist, dass das Licht, was mit einem steileren Winkel auf die Oberfläche trifft, eher durch das Objekt transmittiert wird (und dementsprechend nur seine eigene Farbe reflektiert) als Photonen, die mit einem flacheren Winkel auftreffen. Bei Letzterem wird weniger Energie benötigt, um den Strahl abzulenken, wodurch er eher reflektiert wird.

Um den oben genannten Effekt mathematisch anzunähern, wird die Schlicksche Näherungsformel aus 2 verwendet.[6]

$$R(\vec{n}, \vec{v}) = R_0 + (1 - R_0) \cdot (1 - \cos(\frac{\vec{n} \circ \vec{v}}{|\vec{n}| + |\vec{v}|}))^5 \tag{2}$$

$$R_0 = \frac{n_1 - n_2}{n_1 + n_2} \tag{3}$$

Dabei entsprechen $\vec{v}$ dem Sichtstrahl und $\vec{n}$ der Normalen. Bei dem Übergang zwischen den Medien (in der Regel Luft und das jeweilige Material) werden außerdem bereits gegebene Brechungsindizes benötigt, welche in der Gleichung n_1 und n_2 entsprechen. Findet dabei der Übergang zwischen Luft und einem Material statt, wird für n_1 ein Wert von 1 verwendet.

Abbildung 4: Ein Foto von einem Bergsee, bei dem der vordere Bereich des Wassers transparenter und der hintere Bereich reflektierender ist. Entnommen von [4]

2.5 Verfahren

Im folgenden Abschnitt werden verschiedene grundlegende Verfahren der Computergrafik zur Berechnung von Reflexionen näher beschrieben. Außerdem werden deren Vor- und Nachteile erläutert und anschließend bewertet.

2.5.1 Cubemapping (CM)

Bei dem Ansatz des Cubemappings wird die Szene sechs Mal aus verschiedenen Blickwinkeln gerendert. Die Position der Kamera wird dabei in die Mitte des spiegelnden Objekts platziert und bleibt für alle sechs Durchläufe konstant. Um die gesamte Umgebung abzudecken, wird der Öffnungswinkel der Kamera auf $90°$ gesetzt und ein quadratischer Viewport verwendet. Durch das sechsmalige Wiederholen wird ein Würfelnetz erstellt, bei dem jede einzelne Textur einer Fläche des Würfels entspricht. [8]
Die Reihenfolge ist dabei vorgegeben und in 5a visualisiert. In dem resultierenden Netz wird anschließend mit dem Reflexionsvektor $\vec{r}$ der Farbwert an der zugehörigen Stelle ermittelt und auf der Oberfläche dargestellt, wie in Abbildung 5b illustriert.
Je nachdem ob eine Veränderung der Szene durch dynamische Objekte

[4] `https://pixabay.com/de/photos/berg-see-natur-gewässer-schnee-3252838/`, aufgerufen am: 20.02.2020

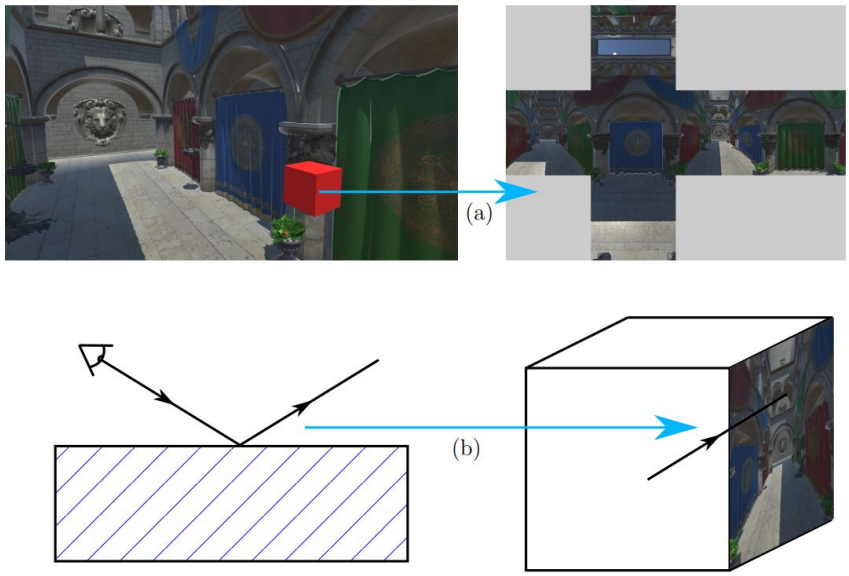


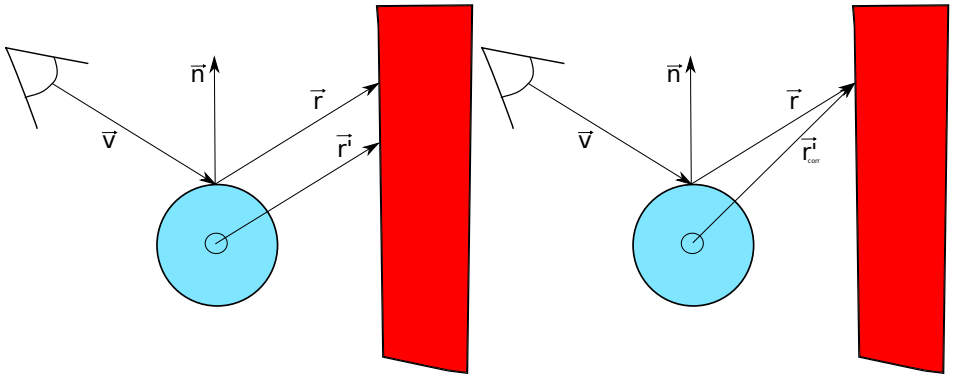
Abbildung 5: (a) zeigt die Generierung der Cubemap mit dem resultierenden Würfelnetz.
(b) zeigt wie die Berechnung des Reflexionsvektors für den Zugriff auf die Cubemap verwendet wird, entnommen von [5].

und / oder Beleuchtung zu erwarten ist, wird eine statische Cubemap vor dem Renderloop oder eine dynamische Cubemap innerhalb des Renderloops erstellt. Letztere erhöht abhängig von der Auflösung der Cubemap wesentlich den Aufwand der Berechnungen der Grafikkarte pro Frame.
Das Problem bei diversen reflektierenden Objekten und deren Auswirkung aufeinander kann durch Blending der jeweils betroffenen Cubemaps gelöst werden. Ein weiteres Problem stellt der einzelne Referenzpunkt der Cubemap dar, wodurch der Reflexionsvektor den Mittelpunkt der Oberfläche anstatt den korrekten Reflexionspunkt in der Mitte des Objekts als Startwert für die Abtastung verwendet, illustriert in Abbildung 6a. Außerdem kann sich der reflektierte Punkt nahezu unendlich weit weg von dem Referenzpunkt befinden, wodurch Cubemapping leere Pixel enthalten kann, was insbesondere in der Kombination mit anderen Verfahren problematisch ist. Diese Probleme können durch einen parallaktisch korrigierten Reflexionspunkt und Verwenden von Proxy Geometrie behoben werden.

2.5.2 Parallax Corrected Cubemapping (PCCM)

Wie bereits erwähnt, muss zunächst der Reflexionspunkt korrigiert werden, um einen korrekten Reflexionspunkt in der Mitte des Objekts als Start-

[5] `https://www.atoft.dev/files/dissertation-redacted.pdf` aufgerufen am: 20.02.2020



(a) Cubemapping visualisiert anhand von reflektierter Kugel (blau) und reflektiertem Objekt (rot).

(b) Parallax Corrected Cubemapping visualisiert mit gleichem Aufbau.

Abbildung 6: Die Visualisierung dient zur Veranschaulichung für den Unterschied durch eine korrigierte Parallaxe.

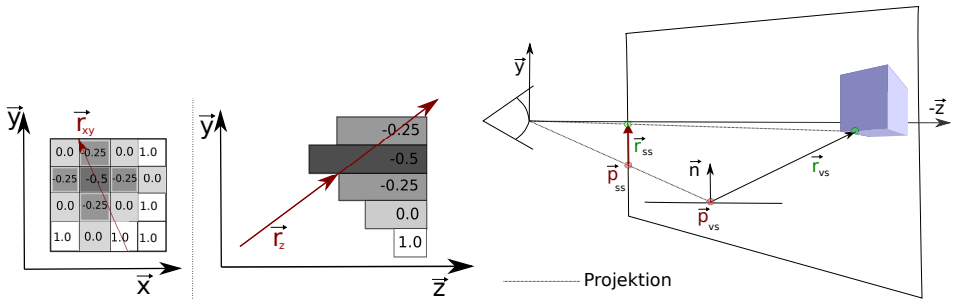
punkt zu verwenden. Diese Korrektur geschieht, indem entlang der negierten Normalen der Oberfläche der Referenzpunkt um die Hälfte der Länge der Geometrie verschoben wird und sich so in der Mitte des Objekts befindet. Dementsprechend muss jedoch der Reflexionsvektor ebenfalls korrigiert werden. [8] Mit dem korrigierten Reflexionsvektor kann dann auf die korrekte Position zugegriffen werden, wie in Abbildung 6b illustriert.

2.5.3 Raytracing

Raytracing (dt. Strahlenverfolgung) bezeichnet ein Verfahren zur Ermittlung von Sichtbarkeit. Es kann zur Darstellung einer Szene verwendet werden, welche mit Strahlen abgetastet werden. Dabei werden von der Kamera ausgehend mehrere Strahlen pro Pixel verschossen, die dann in der Szene verfolgt werden. Komplexere Beleuchtungsmodelle als Phong [4] lassen sich somit auch implementieren, nur sind diese nicht immer echtzeitfähig, da dieses Verfahren sehr aufwändig ist.
Für Reflexionen muss eine Rekursionstiefe gegeben sein, denn mit Raytracing ist es möglich z.B. zwei aufeinander gerichtete Spiegel zu rendern, wofür dann die Rekursionstiefe als Abbruchbedingung benötigt wird. Da das Raytracing auf der Idee der Strahlenverfolgung beruht, lassen sich Reflexionen besonders gut implementieren, benötigen allerdings einen deutlich höheren Rechenaufwand als andere gängige Reflexionsverfahren. [2]

2.5.4 Billboard Reflections

Da es oft zu aufwändig ist, komplexere Szenen mittels Raytracing komplett abzutasten, werden bei Billboard Reflections (BBR) die zu reflektie-



(a) Die Tiefenabtastung des SSR-Verfahrens in der Tiefenmap anhand des Reflektionsvektors $\vec{r}$.

(b) Die Projektion der View Space Position $\vec{p}_{vs}$ und Reflektionsvektor $\vec{r}_{vs}$ in Screen Space.

Abbildung 7: Abtastung des SSR-Verfahrens

renden Objekte durch Imposters oder Billboards ersetzt.[5] Der einfachste Weg dieser Umwandlung geschieht, indem jeweils eine Bounding Box um jedes Objekt erstellt wird und anschließend die Textur des Objekts auf das Quad unwrapped wird.
Um zu ermitteln, welche Farbe in der Reflexion zu sehen ist, wird ein Schnittpunkt zwischen dem Billboard und dem reflektierten Sichtstrahl berechnet und anschließend als uv-Parameter zum Zugreifen auf die Billboardtextur verwendet.
Während BBR eine simple und schnelle Alternative für Reflexionen in Echtzeit bietet, besitzt es mehrere Nachteile. Damit auch nicht quad-approximierbare Objekte (wie beispielsweise Kugeln) als Billboard verwendet werden können, muss die Bounding Box entweder eine runde Form annehmen, wodurch der Zugriff auf die Texturen fehlerhaft werden kann, oder die Textur des Quads enthält (transparente) Objekte, welche sich hinter dem Billboard befinden. Außerdem können Objekte, welche aus der Sicht der Kamera verdeckt sind, nicht in der Reflexion verwendet werden, da über den Sichtstrahl nur auf sichtbare Quads zugegriffen wird. Um leere Pixel an diesen Stellen auf der Oberfläche zu verhindern, kann auf eine Cubemap zurückgegriffen werden.

2.5.5 Screen Space Reflection (SSR)

Weil die Abtastung einer ganzen Szene für jeden Sichtstrahl einen großen performanten Aufwand bedeutet, wird bei der Screen Space Reflection (oder auch Realtime Local Reflections) nur die Sichtstrahlen für die reflektierenden Oberflächen berechnet.[3].
Um mehrfaches Rendern der reflektierten Objekte, wie bei dem Stencil Buffer Reflexions-Verfahren, zu verhindern, wird als Input eine Colormap, Tiefenmap und Normalmap aus der Sicht der Kamera erstellt und in einem

Frame Buffer Object (FBO) zwischengespeichert.[7]
Die Colormap wird verwendet, um die Farbe der Objekte in der Textur als Resultat auf die Oberfläche des reflektierenden Objekts zu rendern. Dementsprechend ist es notwendig, die betroffenen Objekte davor zu beleuchten. Die Normalmap ist notwendig, um den Strahl nach der bereits erwähnten Reflexionsformel 1 zu berechnen. Die Tiefenmap wird zum Abtasten des Strahls verwendet, vergleiche dazu Abbildung 7a. Ist das Objekt an der zugehörigen Stelle tiefer, wird der Strahl weiter verfolgt, sonst wird der Strahl entgegen seiner Richtung in kleineren Schritten abgetastet, vergleiche dazu Abbildung 7b.

3 Konzeption und Implementierung

Mehrere von den bereits erwähnten Verfahren werden in dem nächsten Abschnitt und deren Umsetzung näher erläutert. Dabei liegt der Fokus auf der grundlegenden Mathematik und die Implementierung durch Entwicklung eines Programmes auf der CPU und Grafikkarte.

3.1 Cubemapping

Als performantester Ansatz dient Cubemapping. Dabei wird über den Reflexionsvektor in World Space auf die zuvor erstellte Cubemap zugegriffen [6]. Die Position und Normale werden im Vertex Shader in World Space transferiert und anschließend wie in Listing 1 berechnet.

```
1  out vec4 FragColor;
2
3  in vec3 wsNormal;
4  in vec3 wsPosition;
5
6  uniform samplerCube cubeMap;
7  uniform mat4 vM; // viewMatrix
8
9  void main()
10 {
11     vec3 camPos = vec3(vM[0].w, vM[1].w, vM[2].w);
12     vec3 I = normalize(wsPosition - camPos);
13     vec3 R = reflect(I, normalize(wsNormal));
14     FragColor = vec4(texture(cubeMap, R).rgb, 1.0);
15 }
```

Listing 1: Simpler Cubemap Reflexion Fragmentshader

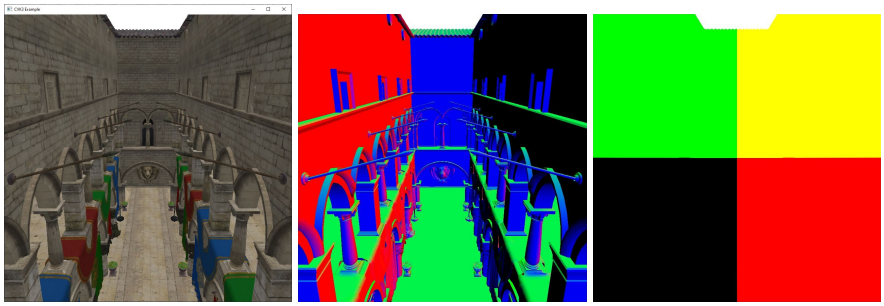
3.2 Screen Space Reflection

In den folgenden Unterkapiteln werden die Voraussetzungen, der Algorithmus, die Abtastung und die Einschränkung von SSR erläutert.

3.2.1 Voraussetzung / Input

Damit der eigentliche Algorithmus ausgeführt werden kann, müssen in dem vorherigen Renderpass drei Texturen aus der Sicht der Kamera beschrieben werden. Die Colormap ist eine RGBA Textur, welche die Farben

[6] `https://learnopengl.com/Advanced-OpenGL/Cubemaps` aufgerufen am: 20.02.2020



(a) Colormap **(b)** Normalmap **(c)** Positionmap

Abbildung 8: Die Komponenten des G-Buffer

der bereits beleuchteten Objekte aus der Sicht der Kamera enthält. Die Normalmap enthält im View Space der Kamera die Normalen, welche durch das jeweilige Objekt gegeben sind. Die Tiefenmap wird aus den jeweiligen Z-Koordinaten der Positionen der Objekte in der Szene aus der Sicht der Kamera erstellt. Die Tiefen werden von dem View Space in Screen Space transferiert, bevor sie in die Textur geschrieben werden. Außerdem muss der z-Wert anschließend linearisiert werden, was in Abschnitt 3.4.5 näher erläutert wird.
Außerdem werden die Breite und Höhe des Fensters für weitere Berechnungen benötigt.

3.2.2 SSR

Nachdem der G-Buffer, vergleiche dazu Abbildung 8, aus dem ersten Renderpass befüllt wurde, werden die Texturen an das Shaderset für den nächsten Renderpass übergeben. Zunächst wird die Blickrichtung ermittelt, indem im Vertex Shader die Position des aktuellen Pixels in View Space transformiert wird und dem Fragment Shader übergeben wird. Die Berechnung für die Blickrichtung erfolgt durch $Fragmentposition - Kameraposition$. Durch die Transformation in View Space entspricht die Kameraposition dem Ursprung (0.0, 0.0, 0.0) des lokalen Koordinatensystems. Daher reicht es aus, die Fragmentposition in View Space zu transformieren, um die Blickrichtung in View Space zu erhalten.
Die Blickrichtung wird anschließend an der Normalen aus der Normalmap reflektiert. Der daraus resultierende Reflexionsvektor befindet sich bereits in View Space, da die Normale und Blickrichtung auch bereits in View Space waren. Bei der anschließenden Transformation des Vektors in Screen Space ist es wichtig, dass die Transformation über die transponierte Inverse geschieht. Außerdem wird die Richtung nicht im letzten Schritt in einen positiven Wertebereich transferiert.
Nach der Projektion der relevanten Variablen beginnt das Raycasting des

Reflexionsvektors in der Tiefentextur. Ein Fragmentshader zur Screen Space Reflexion ist in Listing 2 gegeben.

```
1  out vec4 FragColor;
2
3  in vec3 wsNormal;
4  in vec3 wsPosition;
5
6  in vec3 vsNormal;
7  in vec3 vsPos;
8
9  uniform float tRes;
10 uniform sampler2D colorMap;
11 uniform sampler2D normalMap; // in viewMatrix space
12 uniform sampler2D positionMap; // in viewMatrix space
13
14 uniform mat4 viewMatrix
15
16 void main()
17 {
18     vec3 vsRefVec = reflect(normalize(vsPos.xyz), normalize(vsPos));
19     vec3 ssPos = VStoSS(vsPos.xyz);
20     vec3 ssVec = VStoSS(vsRefVec);
21
22     //Vektorlaenge zu Pixelbreite
23     ssVec = ssVec / max(ssVec.x, ssVec.y) / tRes;
24     fragColor = raycast(ssPos, ssVec);
25 }
```

Listing 2: Screen Space Reflexion Fragmentshader

Es ist wichtig, dass das Depth Attachment des ersten FBOs an den zweiten weitergegeben wird, da sonst das zu reflektierende Objekt nicht richtig in die Tiefe der gesamten Szene eingeordnet wird.
Das Resultat wird in einer Textur gespeichert und mittels additivem Blending mit der Colormap auf einem Screen Filling Quad gerendert.

3.2.3 Screen Space Abtastung

Die Abtastung des Strahls (ray casting) bietet sich durch mehrere Gegebenheiten an. Zunächst müssen die Werte der Positionmap und der reflektierte Sichtstrahl in Screen Space projiziert werden, was in Listing 3 in Form von Pseudocode veranschaulicht wird.

```
1 vec3 VStoSS(vec3 Pos)
2 {
3     vec4 vsP = vec4(Pos.xyz, 1.0);
4     vec4 csP = projectionMatrix * vec4(vsP.xyz, 1.0);
5     vec3 ndcsP = csP.xyz / csP.w;
6     vec3 ssP = 0.5 * ndcsP + 0.5;
7
8     return vec3(ssP);
9 }
```

Listing 3: Algorithmus für die Transformation von View zu Screen Space für Positionen

Nachdem die Tiefenwerte korrekt interpoliert wurden, was in Kapitel 3.4.5 näher erläutert wird, kann anschließend das Maximum aus Höhe und Breite der Textur ermittelt werden. Aus diesem Wert ergibt sich die Länge des Strahls, die pro Iteration auf die Referenzposition addiert wird. Um möglichst performant über die Textur zu iterieren, wird die Länge des Vektors auf $\frac{1}{\max(t.height,t.width)}$ gesetzt, wobei t.height und t.width jeweils der Höhe und Breite der Textur entspricht. Dadurch wird ermöglicht, dass der Vektor sich immer genau eine Pixelgröße pro Iteration fortbewegt. Dementsprechend wird der folgende Prozess bis zu $\max(t.height, t.width)$ ausgeführt.
Zunächst wird der z-Wert der aktuellen Position mit dem der Positionmap verglichen. Sollte die Differenz geringer als ein fester Delta Wert sein, ist eine Kollision zwischen Reflexionsvektor und einem Objekt in der Szene vorhanden. In dem Fall wird über die aktuelle Position, welche sich in Screen Space befindet, auf die Colormap zugegriffen und deren Inhalt als Fragcolor ausgegeben.
Sollte der z-Wert in der Positionmap größer sein (die Position in der Tiefenmap ist tiefer als die aktuelle Position) wird auf die aktuelle Position der Reflexionsvektor addiert und die Iteration fortgeführt. Falls die Tiefe der aktuellen Referenzposition größer ist als die in der Positionmap, bedeutet dies, dass der Vektor über die Kollision hinaus ist und durch eine bspw. binäre Suche die Abtastung erfolgen muss.
Sollte die Suche ohne Resultat abschließen, kann davon ausgegangen werden, dass an dieser Stelle der Himmel oder die Skybox zu sehen ist. Um zu verhindern, dass Texturzugriffe außerhalb des Bildschirms (out of Screen Space) stattfinden, wird die Suche in diesem Fall abgebrochen.
Der Ablauf des beschriebenen Algorithmus ist in Listing 4 näher beschrieben.

```
1 vec4 raycast(ssPos, ssVec)
2 {
3     while (count < tRes)
4     {
```

```
5           if(isOffScreen)
6           {
7               return texture(cubeMap,wsReflectionVector);
8       }
9       float rayDepth = ssPos.z;
10      float samp_Depth = VStoSS((texture(posMap, ssPos.xy)).xyz).z;
11      float delta = abs(samp_Depth - rayDepth);
12      if(rayDepth < samp_Depth)
13      {
14            ssPos += ssVec;
15      }
16      else
17      {
18          if (delta < minimumDelta)
19          {
20                FragColor = texture(colorMap, ssPos.xy);
21            }
22            else
23            {
24                ssVec /= 2.0;
25                ssPos = ssPos - ssVec;
26            }
27          }
28      }
29  }
```

Listing 4: Abtastung in Screen Space

Die passende Abtastungsgröße stellt einen wichtigen Bestandteil der Reflexion dar und ist einer der Unterschiede zwischen den Abtastungsverfahren. Sollte ein zu großer Wert gewählt werden, führt dies zu Fehlern in der Abtastung wie zum Beispiel das Wiederholen desselben Pixels entlang einer Linie. Diese Streifeneffekte können auch bei einer nicht-linearisierten Tiefe für Objekte, die sich von der spiegelnden Oberfläche weiter entfernt befinden, auftreten.
Bei einer zu kleinen ermittelten Abtastungsgröße werden die gleichen Pixel in den Maps mehrmals überprüft. Dadurch wird die Schleife zu oft ausgeführt, was in der Regel zu höherem Leistungsaufwand und zu verfrühtem Abbrechen der Schleife führt. Die Abtastung in Pixelgröße führt daher dazu, dass jedes Pixel maximal einmal getestet wird, ohne dass dasselbe Pixel mehrmals überprüft wird.

3.2.4 SSR Einschränkung

Auch wenn SSR ein vergleichbar performantes Reflexionsverfahren ist, besitzt es mehrere Grenz- bis Fehlerfälle. Da nur auf Texturen aus der Sicht der Kamera zugegriffen wird, können Objekte außerhalb der Sicht nicht in der Reflexion dargestellt werden. Dies betrifft Objekte, welche sich außerhalb des Frustums der Kamera befinden, aber auch welche, die von anderen Objekten verdeckt werden. Um „leere“ Pixel in der Reflexion zu verhindern, kann auf eine Cubemap zugegriffen werden. Dabei wird wie für die Berechnung der Reflexionen bei Cubemaps fortgeschritten.[9]
Alternativ ist es auch möglich, den Viewport kleiner als die Texturgröße im G-Buffer zu setzen. Dadurch beinhalten die Maps mehr Informationen als die Kamera, wodurch auch Objekte außerhalb des Sichtbereichs in der Reflexion gefunden werden können. Dies stellt jedoch eine eher mangelhafte Lösung dar, da um sicherzugehen, dass jedes Objekt der Szene in den Maps vorhanden ist, der Öffnungswinkel bei dem Beschreiben der Texturen $360°$ betragen muss. Der daraus resultierende Rechner- und Speicheraufwand überschreitet den Gewinn durch SSR, weshalb Cubemaps die effizientere Alternative darstellen.
Das Problem, das Objekte aus der Sicht der Kamera andere Objekte verdecken und dementsprechend nicht in der Colormap für SSR vorhanden sind, kann durch mehrfaches Rendern behoben werden. Dabei wird bei dem ersten Renderpass die Colormap (wie auch zuvor) mit allen front faces befüllt. Anschließend wird mit einer zweiten Tiefenmap, bei welcher die Tiefenwerte nicht zuvor modifiziert wurden, erneut gerendert. Die zweite Tiefenmap wird anschließend so geupdated, dass die neue Near-Plane der Position der ersten verdeckten Vertices entspricht.
Dieser Vorgang wird wiederholt, bis alle Schichten der sichtbaren Szene unabhängig von ihrer Tiefe in jeweils eine Colormap gerendert wurden. Dieser Algorithmus wird als Depth Peeling bezeichnet und vor allem bei Order Independent Transparency Verfahren verwendet. Für SSR behebt dieses Verfahren das Problem der Verdeckung durch andere Objekte sowie das Rendern von Flächen, die nur aus der Sicht der Reflexionsquelle deutlich zu sehen sind (beispielsweise Unterflächen oberhalb des reflektierenden Objekts).

3.3 View Space Abtastung

Der simpelste Ansatz für Screen Space Reflexionen ist die Werte der Maps und den Reflexionsvektor in View Space zu berechnen und dann die Abtastung entlang der invertierten Tiefenwerte vorzunehmen. Der Algorithmus ist in Pseudocode in Listing 5 beschrieben.
Zum Abtasten wird ein fester, frei gewählter Wert benötigt, ab welchem die metrische Distanz für eine Kollision hinreichend ist. Sollte eine Kollisi-

on gefunden werden, wird die aktuelle Position in Screen Space projiziert, um auf die Texturkoordinaten der Colormap zu schließen.[2] Die Vektorlänge muss je nach Szene angepasst werden, da sonst die Suche zu lange dauert oder keine Resultate erzielt.
Während das Ergebnis in optimalen Bedingungen und passend gewählten Konstanten dem von SSR sehr nahe kommt, ist dieses Verfahren anfällig für mehrere Probleme. Oft befinden sich Artefakte in der Reflexion, da der gewählte Mindestabstand für verschieden große Objekte stark variiert. Außerdem ist der Aufwand zur Berechnung in der Regel höher, da der Vektor entweder zu lang ist, was eine anschließende lange binäre Suche zur Folge haben kann, oder zu kurz ist, welches zu einer hohen Anzahl an Iterationen führt. Letzteres kann zu einem verfrühten Abbrechen der Suche führen, falls eine feste Anzahl an Iterationen gewählt ist.

```
1  void main()
2  {
3      vec3 vsPos = passPosition.xyz;
4      vec3 vsVec = reflect(normalize(vsPos.xyz), normalize(vsN));
5      fragColor = raycast(vsPos, vsVec);
6  }
7
8  vec4 raycast(vsPos, vsVec)
9  {
10     while (count < tRes)
11     {
12         if(isOffScreen)
13         {
14             return texture(cubeMap,wsReflectionVector);
15       }
16       float rayDepth = vsPos.z;
17       float sampledDepth = (texture(positionMap, vsPos.xy).xyz).z;
18       float delta = abs(sampledDepth - rayDepth);
19       if(rayDepth < sampledDepth)
20       {
21             vsPos += vsVec;
22       }
23       else
24       {
25             if (delta < minimumDelta)
26             {
27                return texture(colorMap, VStoSS(vsPos.xy));
28             }
29             else
30             {
```

```
31                      vsVec /= 2.0;
32                      vsPos = vsPos - vsVec;
33                  }
34              }
35          }
36  }
```

Listing 5: Setup + Abtastung in View Space

3.4 Implementation

In diesem Abschnitt wird die Umsetzung der in der Konzeption erwähnten Punkte erläutert. Insbesondere wird die Umsetzung der zuvor erläuterten Merkmale herausgearbeitet.

3.4.1 Setup

Zum Beleuchten der Szene wird ein Shader Storage Buffer Object (SSBO) erstellt und mit relevanten Materialeigenschaften (beispielsweise Diffusität und Spekularität) beschrieben und den Shadern übergeben. Die Berechnung der Beleuchtung wird nach Phong [4] berechnet und anschließend werden die resultierenden RGB Farben in dem SSBO gespeichert und für jede weitere Lichtquelle aufaddiert. Außerdem werden die drei Texturen für SSR in einem FBO gespeichert. Die verwendete Renderpipeline ist in Abbildung 9 illustriert.

3.4.2 Renderpipeline

Alle bereits erwähnten Reflexionsverfahren benötigen mehr als einen Renderaufruf, um die Reflexion darzustellen. Im folgenden Abschnitt wird die Renderpipeline für die implementierte Kombination von SSR und Cubemapping und deren einzelne Renderpasses näher beschrieben.

0. Renderpass Cubemap Berechnung Falls statische Cubemaps verwendet werden, reicht das sechsmalige Rendern mit einem Öffnungswinkel von 90° vor dem Renderloop aus. Dadurch wird ein einmaliges Aufrufen von Phong Buffered zum Beleuchten notwendig, da die Cubemap eine beleuchtete Textur voraussetzt.
Falls dynamische Cubemaps verwendet werden, ist es erforderlich, sechs Mal pro Renderaufruf (bzw. Frame) zu rendern, da sich deren Inhalt während der Laufzeit verändert.

1. Renderpass G-Buffer Alle Objekte in der Szene werden durch Phong Buffered aus der Sicht der Kamera beleuchtet und gerendert. Außerdem werden die Colormap, Normalmap und Positionmap befüllt und gespeichert.

2. Renderpass Screen Space Reflexion Die SSR wird in diesem Renderpass berechnet, wobei die Implementation des Algorithmus in den Kapiteln 3.4.3 bis 3.4.5 näher erläutert wird.

3. Renderpass Screen Filling Quad Die finale Textur, mit der das Screen Filling Quad beschrieben wird, ist das Resultat aus allen vorherigen Renderpasses. Die jeweiligen Texturen können mittels verschiedenen Blending Methoden kombiniert werden. Als Eingangstexturen zählen die Colormap von Phong Buffered, eine Fläche der Cubemap und das Ergebnis von der Screen Space Reflexion.
Um die verschiedenen Texturen zu kombinieren, existieren mehrere Blendingverfahren. Die intuitivste Methode ist die jeweiligen Farbwerte zu summieren (additives Blending). Da dies zu einem in der Summe helleren Bild führt und nicht die Werte (durch einen Alpha-Wert) gewichtet, ist diese Methode in der Regel nicht angebracht.
Bei dem binären Blending wird stets nur ein Farbwert aus den drei Texturen ermittelt und ohne den Einfluss der anderen verwendet. Als Gewichtung kann entweder ein Alpha-Wert oder eine boolesche Variable verwen-

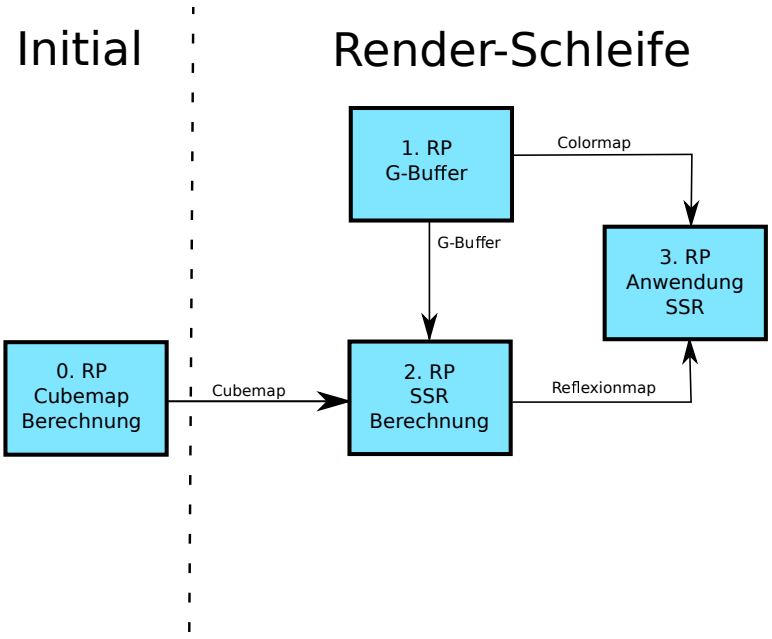


Abbildung 9: Renderpipeline

det werden. Dieses Verfahren bietet sich vor allem an, da meistens nur eine Textur für die Referenzposition zur Verfügung steht.
Für die Übergänge zwischen den Texturen ist ein multiplikatives Blending implementiert, bei dem statt einem Alpha-Wert die Positionen zum Interpolieren zwischen den Texturen verwendet werden. Je weiter das Resultat von SSR sich am Bildschirmrand befindet, desto transparenter wird das Resultat an den SFQ Shader übergeben. Dadurch ist am Rande der Textur ein fließender Übergang zu der Cubemap vorhanden.

3.4.3 Abtastung durch Ray Casting

Sollte der Reflexionsvektor sich in der Tiefe weiter als der eigentliche Schnittpunkt befinden, kann auf eine binäre Suche zurückgegriffen werden. Die Abtastung erfolgt, indem die aktuelle Position mit dem Schnittpunkt verglichen wird und der Vektor je nach Resultat auf die Position addiert oder subtrahiert wird. Anschließend wird der Vorgang mit der Hälfte der bisherigen Vektorlänge wiederholt, wenn die Position weiter als einen festgesetzten Delta Abstand zum Schnittpunkt hat. Dieser Ansatz ist aktuell weit verbreitet und wird beispielsweise auch bei Kollisionen verwendet.

3.4.4 Binäre Suche

Die Abtastung der bisherigen Verfahren kann durch verschiedene Verfahren in der Geschwindigkeit verbessert werden. Eine Option bietet die binäre Suche, ein „divide and conquer“Algorithmus, bei welchem für jeden Abtastungsschritt ermittelt wird, ob die Referenzposition sich weiter in der Tiefe befindet. Die Startposition ist dabei nicht der Reflexionspunkt, sondern die Mitte der abtastbaren Fläche.
Nach dem Vergleichen der Tiefenwerte wird der Vektor um einen gewissen Faktor in seiner Länge verringert und auf die aktuelle Position addiert. Der Ablauf des Algorithmus ist rekursiv in Listing 6 beschrieben.

```
1  vec3 binSearch(vec3 pos, vec3 dir)
2  {
3      vec2 s; //Screen Resolution
4      dir *= 10.0;
5      int sampleCount = max(int(s.x) / int(s.y)) / 10;
6      return binSearch(pos, dir, 0, sampleCount);
7  }
8
9  vec3 binSearch(vec3 pos, vec3 dir, int count, int sampleCount)
10 {
11     vec2 s; //Screen Resolution
12     dir *= 10.0;
13     int sampleCount = max(int(s.x) / int(x.y)) / 10;
```

```
14
15      float delta = abs(currentDepth - sampledDepth);
16      if(delta < 0.001f)
17      {
18        return texture2D(colorMap, pos.xy);
19      }
20      if (count < sampleCount && !isOffScreen)
21      {
22          float sampledDepth = linDepth(texture(depthTex, pos).z);
23          float currentDepth = linDepth(pos.z);
24          if(currentDepth > sampledDepth)
25          {
26              pos -= ssReflectionVector / 10.0;
27              return binSearch(pos, dir, count+1, sampleCount);
28          }
29          else
30          {
31              pos += ssReflectionVector / 10.0;
32              return binSearch(pos, dir, count+1, sampleCount);
33          }
34      }
35  }
```

Listing 6: Pseudocode für die Binäre Suche

3.4.5 Linearisierung der Tiefe

Zum Vergleichen der Tiefenwerte der Textur mit dem Reflexionsvektor kann nicht direkt das Depth Attachment des FBOs verwendet werden, da sich der Wertebereich zwischen 0 und 1 befindet und dadurch die Präzision mit steigender Distanz zum Betrachter abnimmt. Dies führt zu Linienartefakten in der Reflexionstextur entlang der Tiefe. Um dies zu verhindern, muss der Tiefenwert entweder in Viewspace in einer eigenen Tiefenmap gespeichert werden oder durch folgende Formel zurücklinearisiert werden:

$$z_{\text{lin}} = \frac{2 \cdot n}{f + n - z_{\text{ndc}} \cdot (f - n)} \tag{4}$$

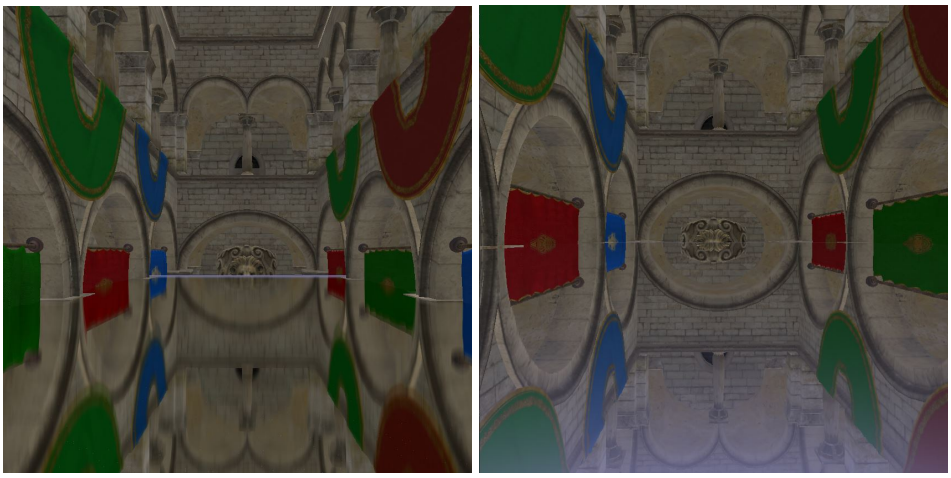
Dabei entsprechen n und f jeweils den Near und Far Werten. Die Gleichung erwartet z_{ndc} in Normalized Device Coordinate Space, kann aber auch mit angepassten Near und Far Werten in Screen Space verwendet werden. Die Implementierung der Formel ist mittels GLSL-Code in Listing 7 veranschaulicht.

```
1 float linDepth(float depthSample)
2 {
3   float zNear = 0.5;
4   float zFar = 5000.0;
5   depthSample = 2.0 * depthSample - 1.0;
6   float b = (zFar + zNear - depthSample * (zFar - zNear)
7   return  2.0 * zNear * zFar / b);
8 }
```

Listing 7: Linearisierung der Tiefe in NDCS

3.4.6 Effekte der Reflexion

Wie in Abschnitt 2.2 beschrieben, besitzt jede Oberfläche eine gewisse Unebenheit, wodurch eine Reflexion nie perfekt gespiegelt wird. Um diesen Effekt darzustellen, wird nach der Berechnung der Reflexion das Resultat geblurrt. Je weiter das gespiegelte Objekt von der reflektierenden Oberfläche entfernt ist, desto mehr entlang der Blickrichtung angrenzende Pixel werden aufaddiert. Um alle Werte gleich zu gewichten und den finalen Farbwert zwischen 0 und 1 zu normalisieren, wird durch die Anzahl der addierten Pixel geteilt.[7] Der Blur Effekt ist in Abbildung 10a zu sehen.
Des Weiteren wurde Fresnel, siehe dazu Kapitel 2.4, implementiert, zu sehen in Abbildung 10b.



(a) Die Reflexion wird so geblurrt, sodass die Objekte mit steigender Distanz unschärfer werden.

(b) Der blaue Farbton visualisiert, wie sehr durch Fresnel das Oberflächenmaterial zu sehen ist.

Abbildung 10: Die Textur wird nach der Reflexion mittels Fresnel und Blur weiter bearbeitet. Der blaue Farbton entspricht dabei dem Material, die spiegelnde Plane dem zu reflektierenden Objekt.

```
1  vec4 blur()
2  {
3    vec4 reflCol = vec4(0.0); //Reflected color
4    float f = ssPos.z;
5    float blurSize = 20.0 * f;
6    int counter = 0;
7    for(float i = -blurSize / 2.0; i < blurSize / 2.0; i+= 1.0)
8    {
9      vec2 texCoord = vec2(ssPos.x, ssPos.y + i / tRes );
10     reflCol += vec4(texture2D(colorMap, texCoord).xyz, 1.0);
11     counter++;
12   }
13   reflCol /= counter;
14   return reflCol;
15 }
```

Listing 8: Pseudocode für den Blur Effekt

4 Ergebnisse und Bewertung

In diesem Kapitel wird der in Kapitel 3 beschriebene Algorithmus in verschiedenen Szenen evaluiert. In den folgenden Unterkapiteln wird der Aufbau der Testszenarien beschrieben sowie deren Auswertung.

4.1 Aufbau der Testszenarien

Um SSR zu evaluieren, wurden Szenen verschiedener Auflösung gewählt, welche in Tabelle 1 beschrieben sind. Dabei variieren die Anzahl an Vertices von 25 284 bis 429 519 und die Anzahl an Dreiecken von 83 490 bis 143 173. Es wurde für jede Szene einmal in View Space und einmal in Screen Space abgetastet. Dabei wurden die FPS gemessen, um die Performance evaluieren zu können. Des Weiteren wurden Bilder von annähernd gleichen Kameraperspektiven aufgezeichnet, um die Qualität der Verfahren gegenüberzustellen. Benutzt wurde dazu ein Rechner mit folgender Hardware:

- GPU: NVIDIA Geforce 860M, VRAM: 4096 MB
- CPU: Intel i7-4710HQ 2.5 GHz
- RAM: 8 GB

4.2 Auswertung

In den folgenden Unterkapiteln werden Performance und Qualität der View Space und Screen Space Abtastung gegenübergestellt.

4.2.1 Performance

Um die Rechenleistung der Verfahren miteinander zu vergleichen, wurde deren Frames per Second (FPS) unter gleichen Voraussetzungen in drei verschiedenen Szenen mittels dem Programm Fraps [8] berechnet. Auffällig ist, dass VSA und SSR bei größeren Szenen wie bspw. der Fireplace Room

[7] https://casual-effects.com/data/

[8] `https://www.fraps.com/`, aufgerufen am 25.02.2020

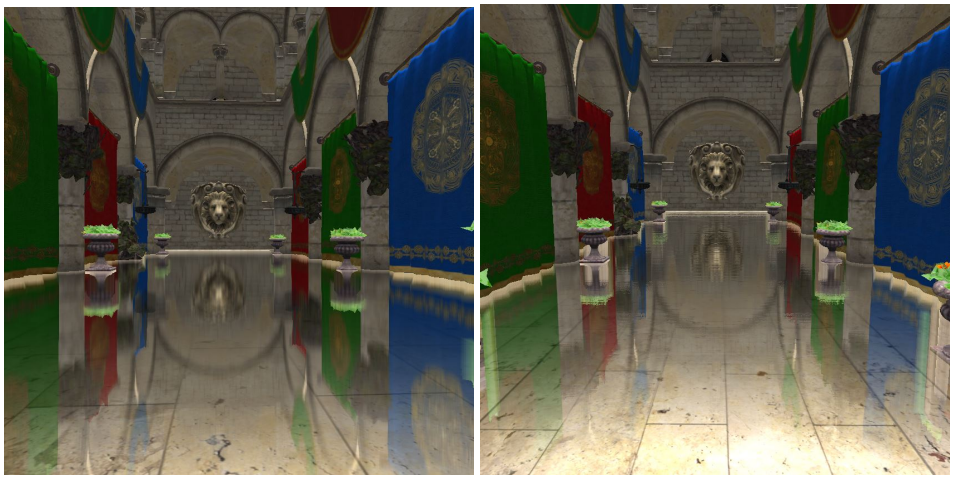
Szene	# Vertices	# Dreiecke
Sponza	153 635	135 394
Fireplace Room[7]	429 519	143 173
Sibenik Cathedral[7]	75 284	83 490

Tabelle 1: Aufbau der Testszenarien

Szene	Verfahren	FPS
Sponza	VSA	30
	SSR	70
Fireplace Room	VSA	20
	SSR	50
Cathedral	VSA	33
	SSR	70

Tabelle 2: Durchschnittliche FPS der Testszenarien

Szene weniger FPS haben als bei der Sibenik Cathedral Szene – beide Verfahren haben Einbußen um ca. 10-20 FPS.
Unabhängig von der Szene lässt sich sagen, dass SSR erheblich performanter als VSA abschneidet – bei der Fireplace Room Szene liegt der Faktor bei 2.5. Dies lässt sich aus der höheren Anzahl an Iterationen für die VSA verglichen zu SSR erklären.
Des Weiteren sinkt die FPS erheblich bei Bewegung der Kamera, wenn VSA aktiv ist. Das Minimum der FPS bei Bewegung beträgt dabei 11 FPS.
Der in Kapitel 3 implementierte Algorithmus ist mit SSR auch in großen Szenen echtzeitfähig (ca. 60 FPS).



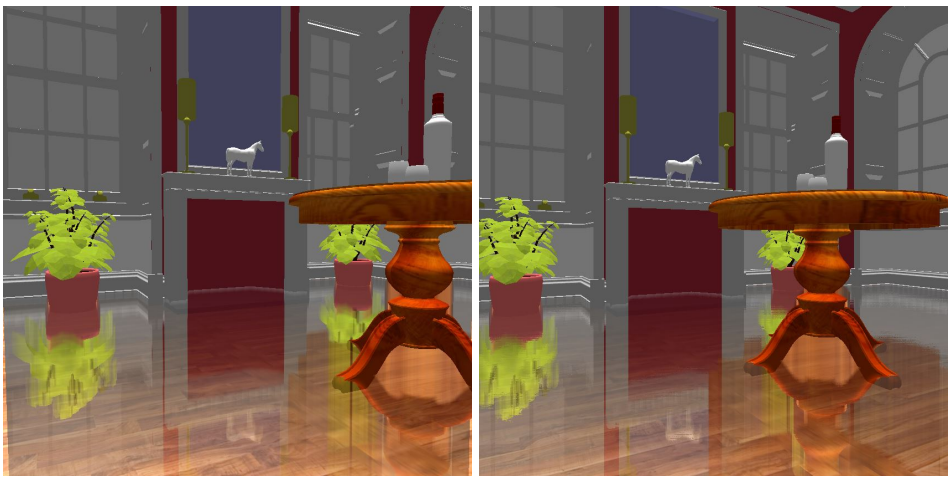
(a) Die View Space Abtastung zeigt eine Streckung entlang der Tiefe wie zum Beispiel in der Krümmung des Bogens oberhalb der Löwenbüste.

(b) Die vorherige Verzerrung ist durch das Linearisieren der Tiefe 3.4.5 behoben.

Abbildung 11: Gegenüberstellung von VSA und SSR in der Sponza Testszene

4.2.2 Qualität

In den Abbildungen 11a und 11b wird die Reflexion von beiden Verfahren zum besseren Vergleich gegenübergestellt. Obwohl beide Reflexionen geblurrt sind, enthält das Ergebnis von SSR mehr Details. Beispiele dafür sind die reflektierten Pflanzen und die Muster auf den farbigen Bannern. Die Linienartefakte, welche vor allem bei der Löwenbüste für SSR zu sehen sind, resultieren aus dem Verhältnis zwischen der Texturgröße und der Größe der Szene.



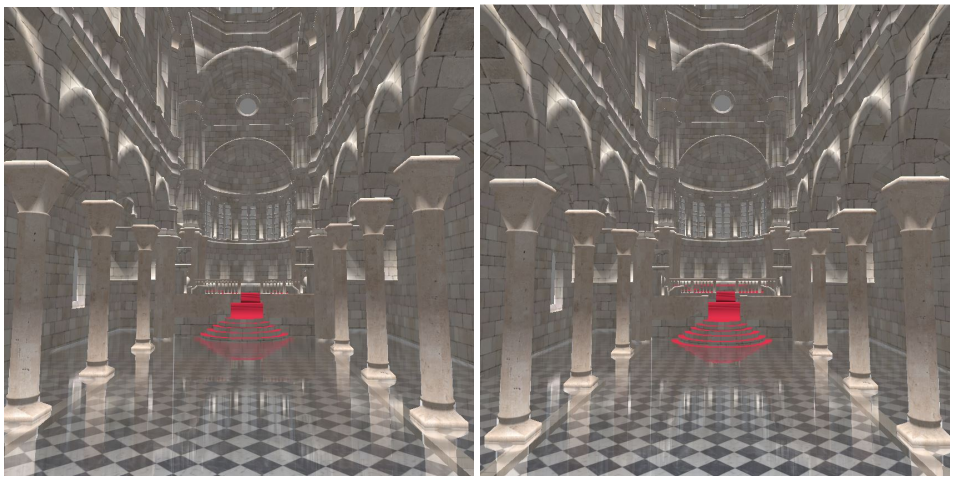
(a) Bei der VS Abtastung werden Teile der rechten Pflanze zu früh gefunden, wodurch Teile der Reflexion den Ast wiederholen.

(b) Bei der SSR Abtastung sind die meisten Artefakte aus VS behoben.

Abbildung 12: Gegenüberstellung von VSA und SSR in der Fireplace Room Testszene

Um den Unterschied zwischen den Abtastungsverfahren besser zu visualisieren, ist jeweils ein Ergebnisbild auf den Abbildungen 12a und 12b dargestellt. Während SSR die Linienartefakte aus der View Space Abtastung behebt, können beide Verfahren nur aus der Kamera wahrnehmbare Objekte reflektieren. Dieser Fall ist unterhalb des Tischs zu sehen, da weder die Unterseite, noch Objekte hinter dem Tisch in der Reflexion dargestellt werden.

Die Reflexionen der Sibenik Cathedral Szene in Abbildung 13 dienen zur Veranschaulichung mehrerer Effekte. Durch den Fresnel Effekt ist je nach Winkel zum Betrachter der Marmorboden stärker sichtbar. Außerdem sind die besonders hellen Regionen besser in der Reflexion zu sehen als die beispielsweise eher dunklen Wandtexturen.
Die jeweils in den Abbildungen 13a und 13b dargestellten Reflexionen ver-



(a) In der Reflexion der View Space Abtastung sind die Treppenstufen mit dem Teppich verschwommen.

(b) In der Reflexion des SSR Verfahrens sind die Stufen besser von dem Teppich zu unterscheiden. Außerdem sind die Artefakte um die reflektierten Säulen behoben.

Abbildung 13: Gegenüberstellung von VSA und SSR in der Sibenik Cathedral Testszene

deutlichen den Unterschied zwischen den Verfahren. Im Gegensatz zu SSR 13b tastet VSA entlang der Tiefe nicht so genau ab. Das kann darin resultieren, dass die roten Treppenstufen nicht korrekt reflektiert werden, wie in Abbildung 13a.
Um nun die Qualität bezogen auf Bildern aus der realen Welt zu vergleichen, wurden Fotos mit ähnlichen Bodeneigenschaften verwendet, die denen der Testszenen ähneln. Dazu wurden ein Foto von einem polierten Holzparkett sowie die Kathedrale von New Orleans verwendet, dargestellt in Abbildung 14.
Die Holzböden in Abbildungen 12b und 14a weisen beide Reflexionen der Umgebungsgeometrie auf – so sind beispielsweise in beiden Szenen die Fenster im Boden zu sehen. Da in der Fireplace Room Testszene keine Beleuchtung von außen erfolgt, sind die Fenster in der Reflexion deutlich dunkler als in der realen Szene. Die angewandten Fresnel und Blur Effekte tragen in der Fireplace Room Testzene deutlich zu einer realistischeren Reflexion bei. Allerdings beinhaltet Abbildung 14a auch Objekte in der Reflexion, die durch SSR nicht erfasst werden können – so ist beispielsweise die Decke des Badezimmers in der Reflexion zu sehen, die in SSR nicht sichtbar wäre.
Die spiegelnden Böden in Abbildungen 11b und 13b lassen sich mit dem Boden der Kathedrale in 14b vergleichen. Die verzerrte Reflexion des Löwenkopfes in Abbildung 11b ist ähnlich verzerrt wie die Reflexion des Al-

(a) Polierter Parkettboden, entnommen aus [9]

(b) Das Innere der Kathedrale in New Orleans, entnommen aus [10]

Abbildung 14: Vergleiche zu Testszenen bieten ein spiegelnder Parkettboden und die Kathedrale von New Orleans.

tars auf dem Boden in 14b. Dies lässt sich in beiden Fällen auf die hohe Distanz zwischen Betrachter und Reflexionspunkt zurückführen. Die gespiegelten hellen Objekte in Abbildung 13b geben realistische Reflexionen der Lichtquellen in Abbildung 14b wieder.

[9] `https://millenniumfloors.com.au/gallery/`, aufgerufen am 26.02.2020

[10] `http://www.asergeev.com/pictures/k/St_Louis_cathedral_New_Orleans_interior.htm`, aufgerufen am 26.02.2020

5 Fazit und Ausblick

Im folgenden Abschnitt wird eine Zusammenfassung dieser Arbeit gegeben, gefolgt von einem Ausblick auf zukünftige Projekte.

5.1 Fazit

Mit der vorangegangenen Bewertung der Ergebnisse lässt sich zeigen, dass die Qualität innerhalb des sichtbaren Bereichs sich realitätsnah verhält. Das Ziel der Echtzeitfähigkeit von durchschnittlich 60 FPS wurde trotz mittelmäßiger Hardware erzielt. Des Weiteren wurde gezeigt, dass das SSR Verfahren schneller und qualitativ bessere Ergebnisse liefert als das VSA Verfahren. Durch das Erläutern von Quellcode und den dabei verwendeten Verfahren ist die Reproduzierbarkeit gegeben.

5.2 Ausblick

Auch wenn das SSR Verfahren in Verbindung mit der Cubemap viele Probleme behebt, bleiben Probleme in der Tiefenabtastung. So würden beispielsweise transparente Objekte in der Colormap die Farben der Objekte hinter ihnen beinhalten, welche aus der Sicht der Kamera in der Regel nicht der aus der Sicht der spiegelnden Oberfläche entspricht. Um dies zu beheben ist eine Form von Depth Peeling notwendig, da ansonsten immer nur Farbe und Position des vordersten Pixel gespeichert wird. Außerdem muss der jeweilige Alpha-Wert in den Colormaps hinterlegt werden und die Abtastung erfolgt für mehrere Schichten. Des Weiteren kann mittels Depth Peeling die Einschränkung von SSR für nicht-sichtbare Unterflächen und verdeckte Objekte behoben werden. Jedoch sollte der daraus resultierende Rechenaufwand nicht unterschätzt werden, und so besser an einem leistungsstärkeren Rechner verwendet werden, als in Kapitel 4. Um an anisotropischen Oberflächen physikalisch korrekt zu reflektieren, ist ein photorealistisches Renderingverfahren (z.B. Bidirektionales Pathtracing) notwendig.[1]
Zu den heute bestehenden Verfahren werden in Zukunft einige hinzukommen, da durch die stetige Weiterentwicklung der Hardware immer mehr möglich wird – so geht der Trend zu echtzeitfähigem Raytracing für beispielsweise Videospiele [9].

[9] `https://www.nvidia.com/de-de/geforce/20-series/rtx/` hin

Literatur

[1] D. V. de Macedo and M. A. F. Rodrigues. Real-time dynamic reflections for realistic rendering of 3d scenes. *The Visual Computer*, 34(3):337–346, 2018.

[2] N. Kurachi. *The magic of computer graphics*. AK Peters/CRC Press, 2011.

[3] M. McGuire and M. Mara. Efficient gpu screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, 2014.

[4] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[5] V. Popescu, C. Mei, J. Dauble, and E. Sacks. Reflected-scene impostors for realistic reflections at interactive rates. In *Computer Graphics Forum*, volume 25, pages 313–322. Wiley Online Library, 2006.

[6] C. Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[7] G. Schmidt. Rendering view dependent reflections using the graphics card. 2015.

[8] L. Sébastien and A. Zanuttini. Local image-based lighting with parallax-corrected cubemaps. In *ACM SIGGRAPH 2012 Talks*, pages 1–1. 2012.

[9] M. Sobek. Real-time reflections in mafia 3 and beyond. Game Developers Conference, 2018.