

Zeitlich stabile blue noise Fehlerverteilung im Bildraum für Echtzeitanwendungen

Bachelorarbeit von

Jonas Heinle

An der Fakultät für Informatik
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

Bearbeitungszeitraum: 12. November 2019 - 11. März 2020
Erstgutachter: Prof. Dr.-Ing. Carsten Dachsbacher
Zweitgutachter: ?
Betreuernder Mitarbeiter: M.Sc. Emanuel Schrade

Inhaltsverzeichnis

1 Prelude	1
1.1 Abstract	1
1.2 Einleitung	2
2 Grundlagen	3
2.1 Rasterisierung	3
2.1.1 Beschränktheit	4
2.2 Path Tracer	6
2.2.0.1 Funktionsweise	6
2.2.0.2 Monte-Carlo-Integration	7
2.2.0.3 DirectX Raytracing	8
2.2.1 Render Graph	10
2.3 Blue Noise	11
2.3.1 Eigenschaften	11
2.3.1.1 Uniformität	11
2.3.1.2 Niedrige Frequenzen	12
2.3.1.3 Isotropie	12
2.3.1.4 Kachelung	13
2.4 Quasi-Zufallsfolgen	15
2.4.1 Einleitung	15
2.4.2 Goldener Schnitt	15
2.4.3 1-Dimension	15
2.4.4 2-Dimensionen	15
2.5 Simulated Annealing	17
2.5.0.1 Allgemein	17
2.5.1 Abkühlfunktion	19
2.5.1.1 Hajek	19
2.5.1.2 Linear	19
2.5.1.3 Exponential	20
2.5.1.4 Inverse	20
2.5.1.5 Energieverlauf	21
3 Temporaler Algorithmus	24
3.1 Blue Noise Dithering Sampling	25
3.2 A Posteriori	26
3.2.1 Theoretische Grundlage	26
3.2.2 Praktische Durchführung	27
3.3 Sorting	28
3.4 Retargeting	33
3.5 Rechenaufwand	37
Literaturverzeichnis	38

1. Prelude

1.1 Abstract

Die Bildberechnung durch hardwareunterstützte Strahlenverfolgung und dazugehörige Techniken gewinnen gegenwärtig in der Echtzeitcomputergrafik an Bedeutung. Trotz dieser neuen Hardwareunterstützung entfällt nur wenig Rechenzeit auf die Berechnung eines einzelnen Bildes. Einhergehend zu dieser kurzen Rechenzeit sind wiederrum weniger Pfade mit dementsprechend geringerer Länge. Bereits frühere Arbeiten haben, um den so entstehenden Bildrauschen entgegenzuwirken, die blue noise Fehlerverteilungen miteinbezogen und deren Bedeutung in der Steigerung der wahrnehmbaren Bildqualität hervorgehoben und verdeutlicht. Diese Arbeit erläutert einen zeitlich stabilen Algorithmus aufgrund dieser Technik. Im Gegensatz zu vorhergehenden Ansätzen wollen wir direkt im Bildraum eine Fehlerumverteilung anwenden, um so eine entsprechend korrelierte Pixelfolge zu erhalten. All dies erreicht der Algorithmus ohne signifikanten Mehraufwand.

1.2 Einleitung

Das *q2vkpt*-Projekt(siehe [Schied, 2019]) zeigt beispielhaft den aktuellen Übergang in Echtzeitanwendungen, indem es in einem konventionellen Spiel die (teilweise) konventionelle Bilderzeugung mit neuen Technologien des *Real-Time Raytracing* austauscht.

Abschnitt 2.1 beschreibt die bisherige, konventionelle Herangehensweise und zeigt deren Limitierungen auf. Diese Limitierungen führen uns zu einem Ansatz, der im

Abschnitt 2.2 besprochen wird. Hiermit lassen sich optische Phänomene, so z.B. Schatten, Spiegelungen *korrekt* darstellen. Diese Technik wird durch die neue Hardwareunterstützung für Echtzeitanwendungen zugänglich, wenn auch mit deutlichen Leistungseinschränkungen. Aktuelle Entwicklungen wie in [Georgiev and Fajardo, 2016] haben sich in Bezug auf diese Technik mit Blue Noise dither masks beschäftigt und ihre Nützlichkeit in Steigerung der visuellen Qualität, bei geringer verfügbarer verbleibender Rechenzeit, gezeigt. Diese Ergebnisse motivieren den

Abschnitt 2.3 über blue noise in welchem wir uns die Theorie aneignen und ihre Funktionsweise auf die Steigerung der Bildqualität genau anschauen. Dabei liefert uns [Peters, 2016] eine blue noise Textur, welche wir im

Kapitel 3 in einem temporalen Algorithmus, vorgestellt in [Eric Heitz, 2019], verwenden können. In zwei zusätzlichen Schritten, dem Sorting und Retargeting, lassen sich unsere Pixel im Bildraum so korrelieren, dass eine zeitlich stabile Fehlerverteilungen entsteht. Dabei machen wir uns die Erkenntnisse aus 2.4 zu nutze, um nur eine Textur zu nutzen ohne jedoch auf den Effekt von mehreren durchwechselnden Texturen verzichten zu müssen. Neben der vorberechneten blue noise Texture verwenden wir eine weitere *Retarget*-Textur, welche wir erhalten, indem ein Optimierungsproblem mit Hilfe der im

Abschnitt 2.5 vorgestellten Technik, dem Simulated Annealing, gelöst wird.

2. Grundlagen

2.1 Rasterisierung

Die Rasterisierung spielt in konventionellen Bilderzeugungsverfahren eine große Rolle. Zu Beginn der Rasterisierung haben wir die Eckpunkte der bereits verarbeiteten, transformierten, projizierten Geometrie mit möglichen Beleuchtungsinformationen aus den vorherigen Berechnungen vorliegen (weiterführende Literatur zu der modernen Renderingpipeline [Akenine-Moller et al., 2008]). Mit Hilfe der Rasterisierung wird nun die Farbe jedes einzelnen Pixels bestimmt. Es ist also die Aufgabe der Rasterisierung herauszufinden, welche Geometrie welchen Pixel zu welchen Anteil bedeckt und wie die Shading Informationen zur Farbgebung des Pixels beitragen. Aufgrund dieser Vorgehensweise spricht man auch von einem objektbasierten Bilderzeugungsverfahren.

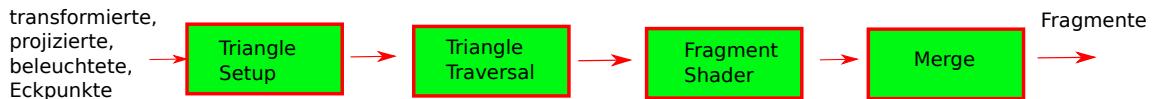


Abbildung 2.1: Ablauf der Rasterisierung

Zuallererst befindet man sich im Ablauf der Rasterisierung beim *Triangle Setup*. Unbeeinflussbar vom Programmierer werden hier Daten berechnet, welche zur Pixeleinfärbung benötigt werden. So werden viele zuvor per Eckpunkt berechnete Werte interpoliert (Beleuchtung, Tiefe). Beim darauffolgenden *Triangle Traversal* werden die wichtigen Fragmente erzeugt. Dieser Schritt bestimmt diejenigen Pixel, welche innerhalb des Dreiecks liegen und erzeugt darauf hin die Fragmente für dieses Dreieck anhand der zuvor berechneten/interpolierten per Dreieck Informationen. Als freiprogrammierbare Shadereinheit können im *Fragment Shader* vom Programmierer weitere Berechnungen vorgenommen werden. Dazu zählt eine pro Pixel Beleuchtungsberechnung (Phong Shading). Im darauffolgenden Schritt wird mit dem Z-Buffer auf Sichtbarkeit geprüft. Die Ausgabe kann in mehrere verschiedene *render targets* geschrieben und somit ein *GBuffer* erzeugt werden. In einem ersten Schritt speichert man Informationen über das Material/Position vom Objekt in verschiedene *render targets*. In einem zweiten Durchlauf kann man nun die Beleuchtung und einige andere Effekte sehr effektiv berechnen. Das abschließende nicht komplett freiprogrammierbare, aber hoch konfigurierbare *Merging* hat eine besondere Aufgabe beim Abspeichern der Farbe für jeden Pixel im color Buffer. Zur Bestimmung der aktuellen Farbe wird nun auch das Problem der Sichtbarkeit von Objekten angegangen. Zu den Z-Werten, welche wir als Tiefe

beim Viewport Transform gespeichert haben, gibt es hier Zugang zum Depth/Z-Buffer. Dieser Z-Buffer speichert anfangs überall den Wert inf. Beim Durchlauf der Geometrie wird nun jeweils für jeden Pixel, der die Geometrie bedeckt der color und depth buffer wie folgt aktualisiert: Ist der verglichene Tiefenwert des vom Objekt erzeugten Fragment kleiner als der Wert im Tiefenbuffer für den betroffenen Pixel, so schreibt er diesen Tiefenwert in den Z-Buffer und auch der color Buffer mit der Fragmentfarbe aktualisiert. Falls nicht passiert nichts und das nächste Primitiv bzw. Fragment wird betrachtet (Szene ohne semitransparente Objekte!). Haben wir semitransparente Objekte, so müssen wir zuerst die Szene wie beschrieben ohne diese Primitive zeichnen, alle semitransparenten Primitive nach ihrer Tiefe ordnen und in dieser Reihenfolge zu dem zuvor gerenderten Bild hinzufügen. Damit haben wir auch unsere Projektion vollzogen, welche wir zuvor vorbereitet haben (Weglassen der z-Komponente).

2.1.1 Beschränktheit

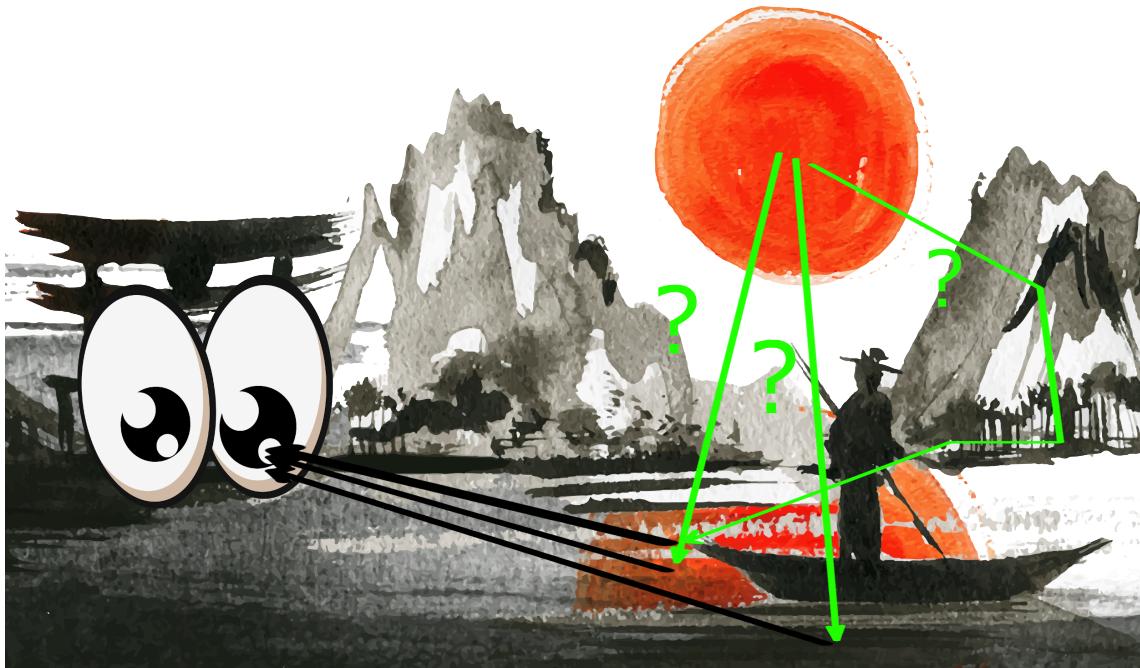


Abbildung 2.2: Ablauf der Rasterisierung

Ihre bisherige weite Verbreitung hatte die Rasterisierung der Objektorientierung zu verdecken: massives paralleles Arbeiten, Ignorieren von (großen) leeren Bereichen und Ausnutzen von Cachekohärenzen gehören zu den Eigenschaften, welche die enorme effiziente, schnelle Abarbeitung bzw. (relativ) geringe aufzuwendende Rechenleistung begründen. Jedoch liegt in ihr auch die Crux. Die Abbildung der Farbe eines Geometrie/Dreiecks auf einen Pixel simuliert den physikalischen Lichttransport nicht korrekt! Die physikalische Optik lehrt uns das Verfolgen von weiteren (sekundären) Strahlen abseits des Primärstrahls, der von Sichtebene zum Objekt verläuft und durch die Rasterisierung im Gegensatz zu den Sekundärstrahlen abgedeckt wird. Abbildung 2.2 verdeutlicht das Problem der Objektorientierung und deren Problem mit Sekundärstrahlen. So können Effekte, welche diese Sekundärstrahlen involvieren, entweder nicht oder nur (unzureichend befriedigend) dargestellt werden (so in 2.2 Spiegelungen, Schatten und Pfade mit größerer Pfadlänge).

Die enormen Leistungsanforderungen von Technologien, welche diesen physikalisch korrekten Lichttransport möglich machen, haben Sie bisher für Echtzeitanwendungen ausgeschlossen. In heutigen modernen Grafikprogrammierschnittstellen (Vulkan, DirectX) jedoch befindet sich Raytracing-Funktionalität, welche auf Hardwareseite unterstützt wird. Diese Unterstützung erlaubt neuerdings effizientere image-ordered Bilderstellungen in Echtzeit. Aktuelle Bemühungen gehen nun daran Strahlenerzeugung und Rasterisierung zu kombinieren. [Barré-Brisebois et al., 2019] stellte mit dem Spiel *PICA PICA* eine solche Rendering-Pipeline vor, welche mithilfe von Path Tracing2.2 arbeitet. Dabei wird der G-Buffer (Texturen die Position, Normalen, Belichtung eines Bildes speichern) noch über Rasterisierung berechnet. Direkten Schatten kann man rastern oder durch das Verschießen von Strahlen bekommen. Diese Option verspricht eine Anpassungsfähigkeit der Pipeline nach Leistungsfähigkeit der Hardware. Ähnlich können nun Reflexionen, Global Illumination, Ambient Occlusion und Transmission durch Verschießen von Strahlen oder auf Compute Shader ausgeführt werden (wieder je nach Hardwareleistung). Einzig direkte Beleuchtung sowie Post-Processing Effekte laufen nur über Compute-Shader.

Wir wollen diesen Ansatz in dieser Arbeit aufnehmen. Berechnung des *GBuffer's* mit Hilfe von Rasterisierung und globale Beleuchtung durch einen Path Tracer erreichen. Da trotz hardwarebeschleunigtes Strahlenverschießen unsere Anzahl an Strahlen beschränkt ist, beschäftigen wir uns innerhalb dieser Arbeit mit einem Temporaler Algorithmus, der die visuelle Qualität nicht durch Verschießen von mehr Strahlen, sondern durch eine zeitlich stabile Blue Noise Fehlerverteilungen im Bildraum erreicht.

2.2 Path Tracer

2.2.0.1 Funktionsweise

In offline Produktionen bereits fest etabliert [Dis, 2020] gewinnt die Technik der Bilderzeugung durch neue Hardwareunterstützung für Echtzeitanwendungen an Aufmerksamkeit[Schied, 2019]. Ähnlich zur Strahlen- wird bei der Pfadverfolgung anstatt vom Objekt die Bilderzeugung ausgehend vom Betrachter angesetzt (siehe Abbildung 2.3). So wird die Farbgebung eines Pixels vom Betrachter, über das betrachtete Objekt bis hin zur Lichtquelle zurückverfolgt.

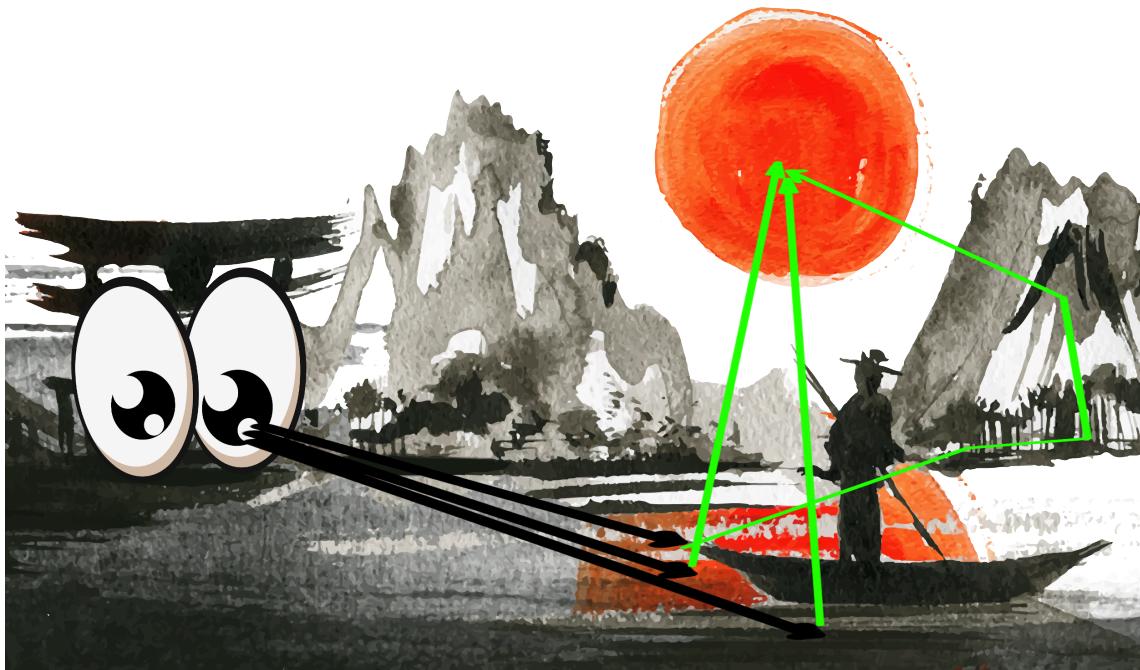


Abbildung 2.3: Grundkonzept Strahlverfolgung

Klassischerweise werden hierbei pro Pixel mehrere zufällige Strahlen verschossen, welche jeder für sich einen *Samplewert* ergibt und zur Farbgebung des Pixels beiträgt. Mehrere unkorrelierte zufällige Folgen ergeben somit die schlussendliche Pixelfarbe. Der Fehler, der bei der zugrundeliegenden Monte-Carlo Integration 2.3 entsteht, wird klassischerweise über Varianzreduktionsmethoden 2.4 wie *Importance Sampling* angegangen. ([referenz für importance sampling](#)) Das Ziel hierbei ist es die Fehlerverteilung im Bildraum zu beeinflussen, da diese für die wahrnehmbare visuelle Qualität des Bildes verantwortlich ist. Diese Arbeit wird vom diesen Vorgehen abweichen und direkt im Bildraum eine zeitlich stabile Blue Noise Fehlerverteilung durch korrelierte Folgen erreichen. Die positive Auswirkung von Blue Noise Verteilungen auf die visuelle Qualität wurde bereits ausgiebig erforscht [Ulichney, 1988].

ToDo

Der Path Tracer ist in Hinsicht der Beleuchtung komplett. Deshalb verwenden wir den Path Tracer um innerhalb unseres 2.5 die *Global Illumination* zu erreichen. Der hier verwendete Path Tracer in [Bentley et al., 2018] beruht auf Erkenntnisse der Lösung der allgemeinen Rendergleichung.Funktionsweise

$$I(x, x') = g(x, x') * \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'' dx'') \right] \quad (2.1)$$

Sie beschreibt den Energietransport I von einem Punkt x' zu einem Punkt x . Dabei ist ein maßgebender Faktor der Geometrieterm g , der die relative Lage der beiden Punkte zueinander im Raum beschreibt. Ein weiterer Faktor ist die Abstrahlung ϵ von x' nach x . Beeinflusst wird der Energiefluss auch durch die bidirektionale Verteilungsfunktion ρ , welche Aufschluss über das einfallende Licht von einem Punkt x'' über x' zu x gibt.

Die Schlussfolgerung aus dieser Gleichung Funktionsweise ist: Die transportierte Intensität von einem Punkt zu einem Anderen ist die Summe des ausgestrahlten Lichts und das reflektierte Licht zu x von allen anderen Oberflächen x .

Ausgehend von der Rendergleichung Funktionsweise lässt sich die vollständige TransportgleichungFunktionsweise beschreiben. Wie von [Marschner and Shirley, 2009] beschrieben wird ausgehend von der vollständigen Transportgleichung Funktionsweise

$$L_s(k_0) = L_e(k_0) + \int_{all(k_i)} \rho(k_i, k_0) * L_f(k_i) * \cos(\theta_i) d\theta_i \quad (2.2)$$

der vollständige Lichttransport beschrieben. Man kann deutlich die Ähnlichkeit zu Funktionsweise erkennen. Wir haben den Emissionsterm, die relative Lage der Punkte zueinander und die bidirektionale Verteilungsfunktion welche den Energietransport beeinflussen.

2.2.0.2 Monte-Carlo-Integration

Mit der Monte Carlo Integration approximieren wir die Rendergleichung.2.1 Bei gegebener Dimensionalität n des Renderintegrals und der Wahrscheinlichkeitsdichtefunktion $\rho(x_i)$ [Drettakis and Seidel, 2002]

$$E\left[\frac{1}{k} \sum_{i=1}^k \frac{f(X_i)}{\rho(X_i)}\right] = \int_{[0,1]^n} f(x) dx \quad (2.3)$$

Dabei wird das n-dimensionale Integral2.2 approximiert. Die Dichtefunktion $\rho(x_i)$ deutet an, dass hierbei die Stichproben auch nicht-uniform genommen werden können. Varianzreduktionsmethoden machen sich diese Dichtefunktion zu Nutze um ein besseres Ergebnis zu bekommen [Caflisch, 1998]. Die Konvergenzrate ist unabhängig von der Dimension unse- res Path Tracer $O(N^{-\frac{1}{2}})$ und ist robust, das heißt Exaktheit hängt nur vom ungenauesten Parameter ab. Eine Variante des Verfahrens, die Monte Carlo Quadratur, wird mit qua- si zufälligen Sequenzen Quasi-Zufallsfolgen, welche eine niedrige Abweichung aufweisen, durchgeführt. Um die Konvergenzrate zu steigern liegen eine Reihe von Varianzreduktionsmethoden vor. Jedoch werden wird hier ein Temporaler Algorithmus und damit eine direkte Umverteilung im Bildraum verwendet.

Abseits dieser herkömmlichen Strategien zeigen wir hier die Steigerung der visuellen Qualität durch blue noise Fehlerverteilung im Bildraum.

$$V[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2 \quad (2.4)$$

2.2.0.3 DirectX Raytracing

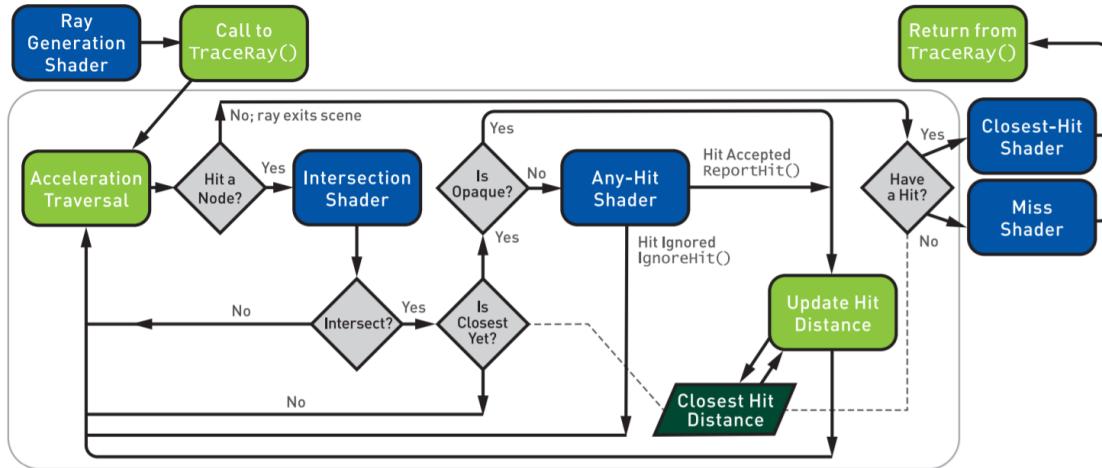


Abbildung 2.4: DirectX Raytracing Pipeline aus [Haines and Akenine-Möller, 2019]

Das hardwareunterstützte Raytracing erhielt Einzug in moderne Programmierschnittstellen(DirectX, Vulkan) und wird hier anhand von DirectX erläutert, welche auch für die *Global Illumination* innerhalb des Render Graphs 2.5 benutzt wurde.

Im Folgenden Algorithmus 1 wird nochmal vereinfacht die Funktionsweise eines Path Tracers erläutert, wobei die entsprechenden programmierbaren Shader von DirectX im jeweiligen Codeabschnitt markiert sind.

Algorithm 1 Path Tracing Algorithmus

```

1: procedure TRACE PATH(BVH)                                ▷ verfolge Pfad durch Szene
2:   for (x,y) ∈ frame do
3:     strahl = verschießeStrahlInPixel(x,y); // ray generation shader
4:     for blatt = bekommeBVHBlatt() do
5:       schnittpunkt = schneideGeometrie(strahl, blatt); //Intersection shader
6:       if schnittpunkt ≤ nähesterSchnittpunkt then
7:         aktualisiereNähestenSchnittpunkt();
8:       end if
9:     end for
10:    if Schnittpunkt gefunden then
11:      frame(x,y) = gebeFarbe(strahl,nähesterSchnittpunkt); //closest-hit shader
12:    else
13:      frame(x,y) = Umgebungskarte(x,y); //miss shader
14:    end if
15:   end for
16: end procedure
  
```

In Abbildung 2.4 und Algorithmus 1 lässt sich der Beginn (Generierung eines Strahles) der neuen Pipeline durch den programmierbaren **Ray Generation shader** erkennen.

Algorithm 2 Beispielhafter minimalistischer Ray Generation Shader

```

1: [shader("raygeneration")]
2: launchIndex = DispatchRaysIndex().xy;
3: for (int i = 0; i < numberOfrays;i++) do
4:     float shadowRayMult = TraceRay(gRtScene, RAY_FLAG_ACCEPT_FIRST_HIT_
    _AND_END_SEARCH | RAY_FLAG_SKIP_CLOSEST_HIT_SHADER, 0xFF,
    0, hitProgramCount, 0, ray, payload);
5:     float indirectRayColor = TraceRay(gRtScene, 0, 0xFF, 1, hitProgramCount, 1,
    rayColor, payload);
6:     color = shadowRayMult * shadingColor + computeindirectLighting(indirectRayColor);
7: end for
8: output[id] = color;

```

Mit Hilfe der Methode **TraceRay()** werden dann zur Beleuchtungsberechnung die Strahlen verschossen. Damit diese Methode richtig arbeiten kann übergeben wir neben unseren Strahl unter Anderem unsere Szene inklusive Beschleunigungsstruktur, rayflags (beeinflussen Transparaenz, Culling, Abbruch)[Ray, 2019] und einen payload. Mit dem *payload* können wir einen struct mit Informationen jedem einzelnen Strahl mitgeben.

Algorithm 3 beispielhafter payload

```
1: struct RayPayload = float4 color, uint32 seed, uint32 depth;
```

Diese Methode **TraceRay()** kann auch innerhalb der anderen Shader zum weiteren verschießen von Strahlen verwendet werden. So beispielweise beim Verschießen eines Schattenstrahls mit flags RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH, RAY_FLAG_SKIP_CLOSEST_HIT_SHADER setzen, um unnötige Beleuchtungsberechnungen und weitere Schnittpunktberechnungen zu umgehen und mit einem Bit als payload die Sichtbarkeit zur Lichtquelle mitzugeben. Mit diesem beispielhaften payload können wir die Farbe akkumulieren, unsere seeds verwenden um z.B eine weiteren Strahlenschuss in einem Any-Hit Shader zu verwirklichen, solange die mit übergebene Rekursionsstiefe in unserem payload eingehalten wird.

Intersection shader führt die Schnittberechnungen durch. Haben wir eine Szene, welche aus ausschließlich Dreiecken besteht, können wir die auf Hardware standardmäßig gelieferte Implementierung übernehmen. Optionale Berechnungen für andere Geometrie können hier implementiert werden. Bei einem gefundenen nächsten Schnittpunkt einer durchsichtigen Oberfläche wird der *Any-hit shader* aufgerufen. **Any-hit shaders** erlauben klassische *Discards* oder informieren über einen korrekten Schnitt. So können wir z.B. einen Alpha Test durchführen.

Algorithm 4 Any-Hit shader

```

1: [shader("anyhit")]
2: if (!alphaTest) then
3:     IgnoreHit();
4: end if

```

Der **Closest-hit shader** berechnet den Schnittpunkt des Strahls mit der Geometrie der Szene, die dem Strahlursprung am nächsten ist. Mit der Kennzeichnung [shader("closesthit")] wird die Hauptmethode zur dessen Ausführung markiert. An dieser Stelle bietet es sich

an die Shading Farbe mit der Schnittpunktinformation zu aktualisieren und/oder um eine Rekursionstiefe weiter zu gehen einen weiteren Strahl zu verschießen. Der **miss shader** wird immer dann ausgeführt, wenn ein Strahl die Szenengeometrie nicht schneidet. Kann also für das Nachschauen in einer Environment Map verwendet werden.

2.2.1 Render Graph

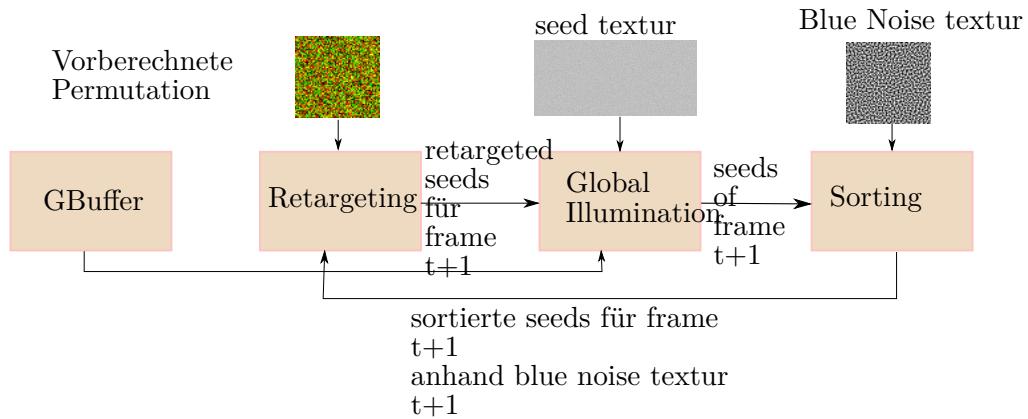


Abbildung 2.5: Unser Render Graph

2.3 Blue Noise

Eine Ansammlung von Pixeln, deren Verteilung über den Raum einer Blue Noise entspricht, weist eine Reihe von Eigenschaften

- Uniformität
- Isotropie
- Kachelung
- Niedrige Frequenzen

zur Steigerung der visuellen Qualität des Bildes auf. [Ulichney, 1988] Im Folgenden wollen wir uns diese Eigenschaften genauer anschauen. Hierfür verwenden wir die in [Games, 2017] vorgestellten Texturen, welche anhand der *Void and Cluster*-Methode[Ulichney, 1993a] erstellt wurden.

2.3.1 Eigenschaften

Um die Eigenschaften der Texturen zu untersuchen wurden korrespondierende Spektren zu den Texturen mit Hilfe von [JCr, 2018] erstellt und miteingebunden.

2.3.1.1 Uniformität

Für die Wahrscheinlichkeitsfunktion, dass ein Pixel mit Grauwert p bei der Generierung ausgeben wird ($p \in [0, 1]$) muss gelten:

$$P(n \leq p) = p \quad (2.5)$$

Die Uniformität(lat. *uniformitas*-Einförmigkeit) garantiert uns dieses Verhalten $\forall p \in [0, 1]$. Man könnte auch sagen, dass jeder Grauwert gleichwahrscheinlich auftreten soll. Die zugehörige konstante Wahrscheinlichkeitsdichte lässt sich einfach zur Echtzeit umsetzen mit Hilfe von (pseudo-)zufälligen Zahlen. Mit der in [Whi, 2019] erstellten white noise Textur,

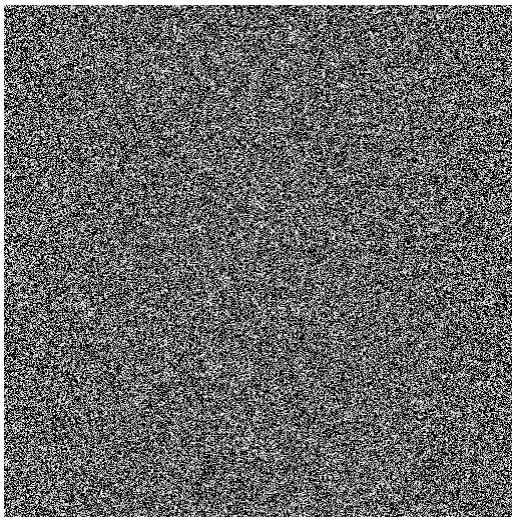


Abbildung 2.6: white noise Textur

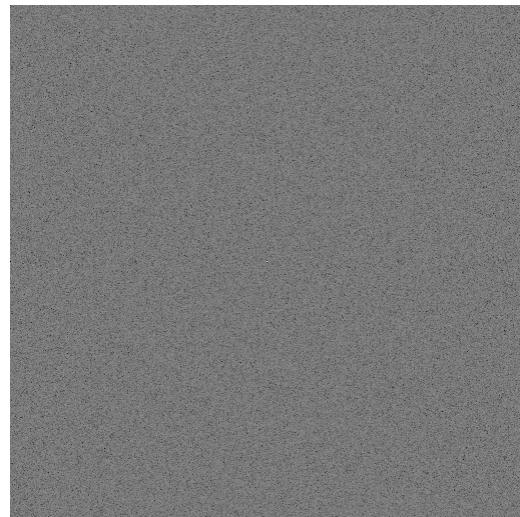


Abbildung 2.7: Amplitudenspektrum

ergibt sich eine typische Amplitudendichte. Zufällig verteilt, über alle Frequenzen hinweg. Mit diesem weißen Rauschen arbeitet man klassischerweise in einem Path Tracer. Diese Uniformität garantiert dabei die Unkorreliertheit der Pixelfolgen. Wir merken also, dass diese Eigenschaft alleine nicht ausreicht und uns zur Niedrige Frequenzen führt.

2.3.1.2 Niedrige Frequenzen

Niedrige Frequenzen sind in einer blue noise sehr wenig bis gar nicht vertreten. Dies macht sich sowohl in der Zeitdomäne erkenntlich: keine erkennbaren gleichfarbigen Pixelverbündete innerhalb der Textur als auch in der Frequenzdomäne: der schwarze Ring innerhalb der Amplitudendichte deutet auf das Fehlen von niederen Frequenzen und dem hohen Unterschied benachbarter Pixel hin. Außerdem haben wir aus der vorherigen Eigenschaft der Uniformität gesehen: Wir wollen das alle Grauwerte gleichwahrscheinlich auftreten. Dies können wir am besten im Frequenzspektrum 2.3.1.2 beobachten. Hier sind alle hohen Frequenzen wie beim weißen Rauschen 2.3.1.1 gleich stark vertreten.

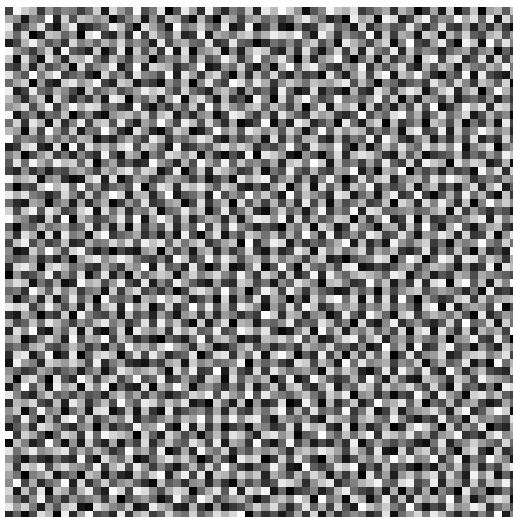


Abbildung 2.8: 512^2 blue noise Textur

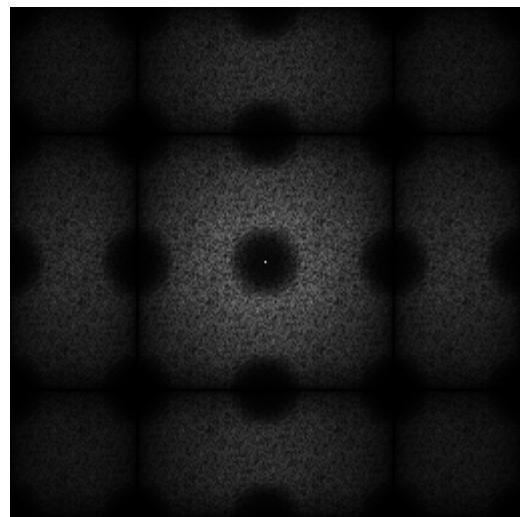


Abbildung 2.9: Fourier Spektrum 512^2 blue noise Textur

Auch zu erkennen ist die weitere Eigenschaft der Isotropie.

2.3.1.3 Isotropie

Die Isotropie(altgr. *isos*-gleich und *tropos*-Richtung) einer blue noise Textur bietet eine weitere wichtige Eigenschaft. Dabei haben wir in allen Dimensionen die Unabhängigkeit einer Eigenschaft. Um uns dies an einem Gegenbeispiel klar zu machen, schauen wir uns das Bayer-Pattern an. Dieses Pattern erfüllt sowohl die Eigenschaft der Niedrigen Frequenz 2.3.1.2 als auch die der Uniformität, jedoch nicht Jene der Isotropie. Zu erkennen ist dies an den sich wiederholenden Strukturen in der Abbildung 2.3.1.3.

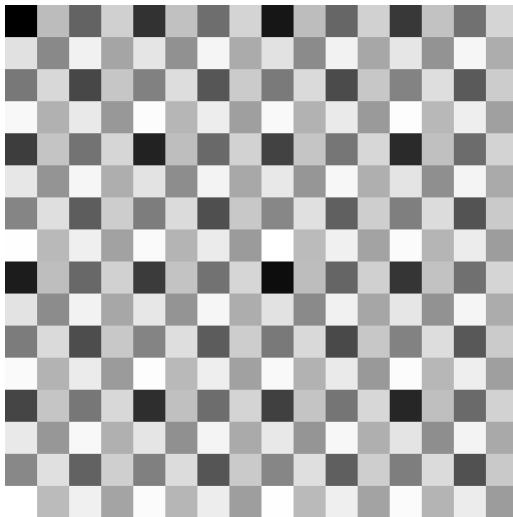


Abbildung 2.10: 512^2 bayer pattern Textur

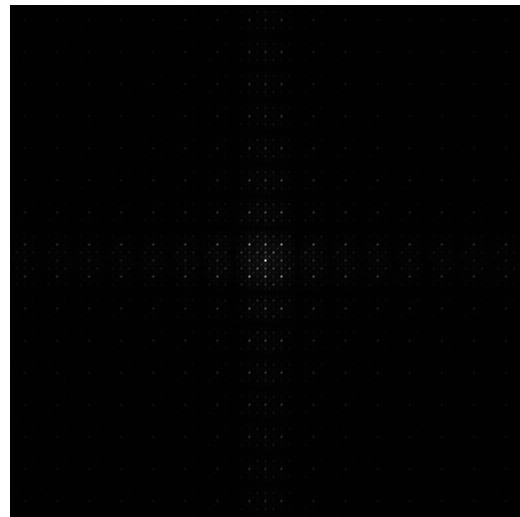


Abbildung 2.11: Amplitudendichte 512^2 bayer pattern Textur

In der Frequenzdomäne 2.11 ist zu erkennen, dass die Amplitudendichte in einzelnen Punkten organisiert ist. Diese lassen sich durch die vorhandenen Richtungen der Textur erklären. Speziell in zwei Richtungen ist eine sich wiederholende Pixelsequenz zu erkennen. Allerdings wollen wir in alle Richtungen eine gleiche (isotrope!) Verteilung.

2.3.1.4 Kachelung

Im Gegensatz zu dem bayer pattern oder der white noise, die sich einfach zur Echtzeit berechnen lassen, sind blue noise verteilte Texturen im Erstellungsaufwand, der mit Anzahl der Dimensionen und Größe der Textur schnell anwächst, deutlich höher [Peters, 2016]. Es empfiehlt sich daher für den Temporaler Algorithmus eine kleinere Textur zu verwenden. Dies hat außerdem den Vorteil, eine bessere Ausnutzung des kleinen aber schnellen Cachespeichers zu gewährleisten. Aufgrund des Aufbaus von aktueller Grafikhardware [Kraft, 2018] wollen wir diese Textur soweit oben wie möglich in der Cachehierarchie halten.(L1 96KByte, L2 6MByte).

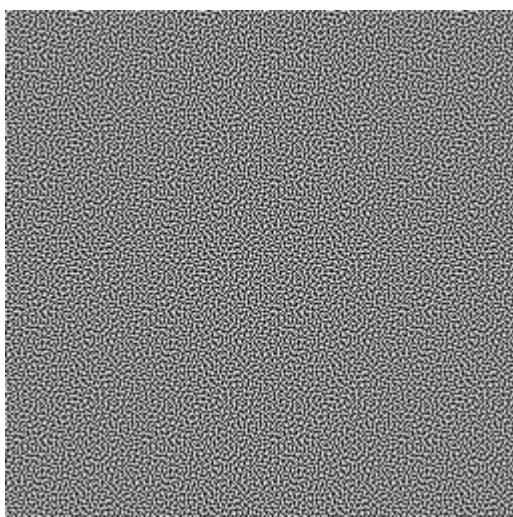


Abbildung 2.12: 512^2 gekachelte Textur von 64^2

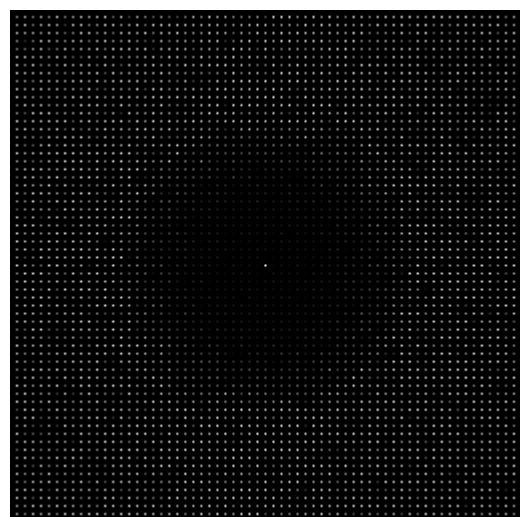


Abbildung 2.13: Fourier Spektrum

Betrachtet man die Abbildung 2.3.1.4 lässt sich der blue noise Charakter anhand des wenigen niederen Frequenzanteils erkennen. Wiederholungen sind in der Zeitdomäne schwerer zu erkennen. Obwohl Sie in der Frequenzdomäne erkennbar sind und unsere Isotropie (teilweise) aufheben. Um dieses Problem zu umgehen werden wir mit Hilfe von Quasi-Zufallsfolgen auf diese Textur zugreifen und die Isotropie wiederherstellen. Daher können wir hier folgendes Fazit ziehen: Regelmäßige Kachelung einer kleineren Blue Noise Textur über das gesamte Bild liefert eine gute Blue Noise Verteilung bei vertretbaren Rechenaufwand und guter Cacheausnutzung!

2.4 Quasi-Zufallsfolgen

2.4.1 Einleitung

Um ungewollten Artefakten entgegenzuwirken wird empfohlen (so auch in [Peters, 2016]) mehrere verschiedene Blue Noise Texturen in ein Array zu laden und diese abwechselnd, randomisiert zu verwenden. Wir führen hier Kenntnisse über Quasi-Zufallsfolgen ein [Roberts, 2018] um aus den zuvor (nach Abbildung 2.3.1.4) genannten Gründen nur eine kleine Blue Noise Textur verwenden zu können.

Quasi-zufällige Sequenzen mit niedriger Abweichung sind deterministisch erzeugte Sequenzen, welche die Likelihood-Funktion der Clusterbildung

$$L_x(\delta) = f_\delta(x) \quad (2.6)$$

minimieren (siehe Abschnitt 2.3.1.2) und dabei die (siehe Abschnitt Uniformität) erhalten. Beide Eigenschaften haben wir bereits bei dem Blue Noise Abschnitt behandelt. Auf Quasi-zufällige Zahlenfolgen angewandt heißt das: Wir benutzen den gesamten Raum an Zufallszahlen *uniform* und hohe Frequenzen vermeiden Regionen, aus den viele Punkte kommen, die wieder punktarme Regionen zur Folge hätten. Im Folgenden wird für uns das Ziel sein, den quasi-zufälligen Zugriff auf eine Textur mit 1-Dimension zu verstehen und nähern uns dabei über den Fall der Goldener Schnitt. Ein Weg quasi-Zufallsfolgen zu beschreiben sind die zugrundeliegenden Parameter. Wir werden uns hier Folgen anschauen, die als Basisparameter den goldenen Schnitt 2.7 verwenden.

2.4.2 Goldener Schnitt

Der goldene Schnitt und die Generalisierung zur plastischen Zahl 2.11 ist mitsamt ihren Eigenschaften bereits früh beschrieben worden [Padovan, 2002]. Als wichtiges Seitenverhältnis in der Architektur konnte Sie durch verschiedene Arbeiten ihren Eingang in die Mathematik finden [Krcadinac, 2006].

$$\Phi_1 = \frac{1 + \sqrt{5}}{2} \approx 1.6180339887 \quad (2.7)$$

2.4.3 1-Dimension

Wir benutzen Rekurrenz Sequenzen, basierend auf irrationalem Bruchrechnen der Form

$$R_1(\alpha) : t_n = s_0 + n\alpha(\text{mod}1); n = 1, 2, 3, \dots \quad (2.8)$$

wobei $\alpha \in \mathbb{I}$ und das $(\text{mod} 1)$ einen "*toroidally shift*" bezeichnet. Will man mit dieser Formel eine Sequenz mit möglichst geringer Abweichung schaffen so wählen wir den goldenen Schnitt 2.7 $\alpha = \Phi_1$ (siehe [Roberts, 2018]).

2.4.4 2-Dimensionen

Da uns die Generalisierung des goldenen Schnittes auf die Lösung der Gleichung 2.9 führt

$$x^{d+1} = x + 1 \quad (2.9)$$

ist das Lösen der kubischen Gleichung 2.10

$$x^3 - x - 1 = 0 \quad (2.10)$$

für den zweidimensionalen Fall nötig. Die Generalisierung und Erweiterung des goldenen Schnittes wurde bereits ausgiebig erforscht [Krcadinac, 2006].

Die sogenannte Plastische Zahl in ist die Lösung der Gleichung 2.10

$$\Phi_2 \approx 1.32471795724 \quad (2.11)$$

Die eindimensionale Rekurrenzsequenz 2.8 ist einfach erweiterbar für höhere Dimensionen.

$$t_n = n\alpha(\text{mod}1), n = 1, 2, 3, .. \alpha = \left(\frac{1}{\Phi_d}, \frac{1}{\Phi_d^2}\right), \quad (2.12)$$

Für den Texturzugriff in unserem Shader bei dem temporalen Algorithmus 3 werden wir also wie folgt vorgehen:

```

1   float g = 1.32471795724474602596; //Plastische Zahl
2   float a1 = (1.0/g) * frame_count;
3   float a2 = (1.0/(g*g)) * frame_count;
4   x[n] = (0.5 + a1*n) % 1; //toroidally shifted
5   y[n] = (0.5 + a2*n) % 1; //toroidally shifted

```

2.5 Simulated Annealing

Für unseren temporalen Algorithmus (Abschnitt 3) gibt es einen wichtigen Retargeting Schritt. In diesem Schritt wird eine vorberechnete Textur verwendet. Diese speichert eine Permutation, die unsere Blue Noise Textur vom Bild t in eine blue noise Textur von Bild $t+1$ umwandelt. Diese Permutation wird dann auf die Startwerte angewandt bevor das nächste Bild $t+1$ gerendert wird (siehe auch Übersicht Unser Render Graph). Dadurch werden die blue noise Umverteilungen der Sorting Phase akkumuliert. All diese Vorberechnungen sind möglich, da wir mit „nur“ quasi-zufälligen Sequenzen (siehe Abschnitt 2.4) arbeiten. Das andauernde Permutieren von Pixeln bis zu einem Punkt, an dem ein Bild aussieht wie das Andere ist ein klassisches TSP, wofür es aktuell keine effiziente optimale Lösungsmethode gibt. Da wir nur an einer sehr guten Lösung, nahe dem globalen Optimum, interessiert sind greifen wir wie in [Eric Heitz, 2019] vorgeschlagen auf das heuristische Approximationsverfahren, dem Simulated Annealing, zu.

2.5.0.1 Allgemein

Angelehnt an metallurgischem Aufheizen und dem sich anschließenden Abkühlen wollen wir eine approximativ optimale Lösung finden. Wir haben also eine zu Anfang hohe Temperatur, welche durch eine Abkühlfunktionen (siehe Abschnitt 2.5.1 für Weiteres) verringert wird. Wir definieren die Energie als pixelweisen Unterschied der sich in Abkühlung befindlichen bereits permutierten Textur und der $Textur_{t+1}$ (siehe Gleichung 2.14). $Textur_{t+1}$ ist durch quasi Zufall (siehe Abschnitt 2.4) bereits bekannt. Mit Erkenntnissen aus [Georgiev and Fajardo, 2016] ergibt sich

$$E(SA) = \sum_{p \neq q} E(p, q) = \sum_{\forall i \in [0, N-1]} \|p_i - q_i\|$$

Abbildung 2.14: Blue Noise Textur mit Dimension N ; Pixel p von abkühlende Textur; Pixel q von Zieltextur

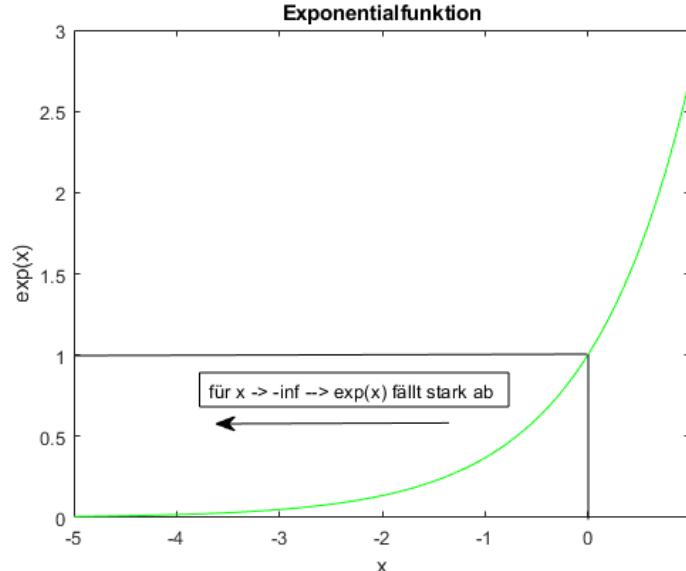
Mit dem definierten Ziel die Energiefunktion 2.14 zu minimieren wenden wir in jedem Schritt eine Permutation an und entscheiden anhand einer Akzeptanzwahrscheinlichkeitsfunktion 2.13 ob wir diese Permutation behalten. Da wir in jedem Schritt nur eine Permutation anwenden vereinfacht sich unsere Energiefunktion zu

$$E(SA) = E(s_{previous}) + \|p_i - q_i\| + \|p_{i+permutation} - q_{i+permutation}\|$$

Abbildung 2.15

$$P = e^{-(energy(s_{neu}) - energy(s))/temperature} \quad (2.13)$$

(a) Akzeptanzwahrscheinlichkeitsfunktion



(b) exponentialfunktion

Abbildung 2.16

Die günstigen Eigenschaften der exponentialfunktion für das Schmelzen sind vielfältig [Van Laarhoven and Aarts, 1987]. Eine ist die Konvergenz der Funktion für $\lim_{n \rightarrow -\infty} \exp(-x)$.

Algorithm 5 Simulated Annealing finde eine globale Lösung nahe am Maximum

```

1: initialisiere Startzustand  $s = s_0$ 
2: initialisiere Starttemperatur  $T_0$ 
3: for i=1...maxSteps do
4:   update Temperatur  $t_i$  anhand des Abkühlplanes
5:   //Radius für Nachbarschaftssuche ist auf 6 festgesetzt
6:    $s_{neu} \leftarrow$  Nachbarzustand( $s$ )
7:    $energy\Delta = energy(s_{neu}) - energy(s)$ 
8:   if  $energy\Delta < 0$  then
9:      $s = s_{new}$ 
10:   else
11:     if  $P(Energie(s), Energie(s_{new}), temperature) \geq random(0,1)$  then
12:        $s = s_{new}$ 
13:     end if
14:   end if
15: end for
16: return Endzustand  $s$ ;

```

Die Wahl der Energiefunktion ist angelehnt an die Formulierung der Wahrscheinlichkeitsakzeptanzfunktion in [Kirkpatrick et al., 1983]. Diese wird bedeutet für positive Energy-deltas, d.h. wenn der Energiezustand des Nachbarn höher als der aktuelle ist. Mit

absteigender Temperatur erkennt man in der Abbildung 2.16b eine ebenfalls abnehmende Wahrscheinlichkeit der Akzeptanz. Dies führt zu dem gewünschten Verhalten, energiehöhere Zustände zuzulassen, um somit lokale Maxima zu verlassen. Dies geschieht bei höheren Temperaturen häufiger wohingegen bei niederen Temperaturen ein gefundenes Maxima seltener verlassen wird. Höhere Deltas führen passender Weiße zu einem höheren negativen Exponenten und damit eine geringere Akzeptanz als Energiezustände, die nur bisschen darüber liegen. Die Wahl des Abkühlvorgangs (also das Updaten der Temperatur über die Zeit) ist problemspezifisch [Kirkpatrick et al., 1983, S. 9]. Dabei muss der Abkühlvorgang derart gewählt werden, sodass kein bloßer Greedy-Algorithmus entsteht und man in einem lokalen Maxima stecken bleibt aber auch kein wahlloses Vertauschen entsteht.

Als Startzustand s_0 definieren wir eine Permutation, die alle Elemente auf sich selbst abbildet. Um von einem Zustand s zu einem neuem Zustand s_{new} zu kommen, definieren wir eine Nachbarschaftsfunktion $Nachbarzustand()$. Diese kann zwei Elemente genau dann vertauschen, wenn Sie in einem gegenseitigen Radius $r = 6$ erreichbar sind. Dabei vertauschen wir in jedem Schritt ein Pixelpaar. Die Wahrscheinlichkeitsfunktion zur neuen Zustandsannahme $P(Energie(s), Energie(s_{new}))$ beschreibt, ob wir den neu gewählten Zustand s_{new} übernehmen. Dabei wird klassischerweise die Akzeptanz von Zuständen mit höherer Energie immer kleiner.(bzw. die Toleranz gegenüber größeren Fehlern im Bezug zur Zeit). Die allgemeine Akzeptanz von Zuständen mit höherer Energie ist dabei von fundamentaler Bedeutung. Somit verlassen wir möglicherweise nur lokale Maxima. Die zu minimierende Energiefunktion Blue Noise Textur mit Dimension N; Pixel p von abkühlende Textur; Pixel q von Zieltextur betrachtet dabei zwei

2.5.1 Abkühlfunktion

Für die Wahl unserer Abkühlfunktion bieten sich einige Optionen.[coo, 2019] Im Folgenden wird auf die verschiedenen möglichen Abkühlfunktionen und ihre Eigenschaften sowie die Wahl interner Parameter(z.B. Starttemperatur) eingegangen. Denn diese Funktion trägt maßgeblich mit ihrem Konvergenzverhalten zur Effizienz des Abkühlvorgangs bei. Nach [Kirkpatrick et al., 1983] wählen wir die Anfangstemperatur T_0 derart, dass anfangs jede Neue generierte Lösung akzeptiert(bzw. nahe 1) wird. Außerdem haben wir einen Zustand des Quasiequilibrium zu definieren [Sci, 2020]. Für jeden zu gehenden Temperaturschritt ist nach einer Abfolge von einer festen Anzahl erfolgreicher Zustandsübergänge das Quasiequilibrium erreicht. Bemerkung: Mit abnehmender Wahrscheinlichkeit steigt die Dauer der hierfür erforderlichen Iterationen. Dafür verwende eine feste Zahl.

2.5.1.1 Hajek

$$f(t) = T_0 \log(1 + t) \quad (2.14)$$

In [Hajek, 1988] haben wir eine Abkühlfunktion gegeben, welche durch ihre Eigenschaft, stets gegen das globale Maximum zu konvergieren, unter allen Anderen heraussticht. Jedoch passiert das asymptotisch extrem langsam. Auch in unseren Fall hat es sich als zu langsam herausgestellt. Leicht aus dem energetischen "Gleichgewicht zu bringen. Erreicht dann nicht das globale Minimum.

2.5.1.2 Linear

$$f(t) = T_0 - \mu * t \quad (2.15)$$

Typische Werte für α liegen zwischen 0.8 and 0.99.[Kirkpatrick et al., 1983].

2.5.1.3 Exponential

Ist nach [Kirkpatrick et al., 1983] eine für viele Fälle zutreffende und zu wählende Abkühlfunktion. Wobei $\alpha \in [0.8; 0.99]$

$$f(t) = T_0 * \text{pow}(\alpha, t) \quad (2.16)$$

2.5.1.4 Inverse

$$f(t) = T_0 / (1 + \text{alpha} * \text{step}) \quad (2.17)$$

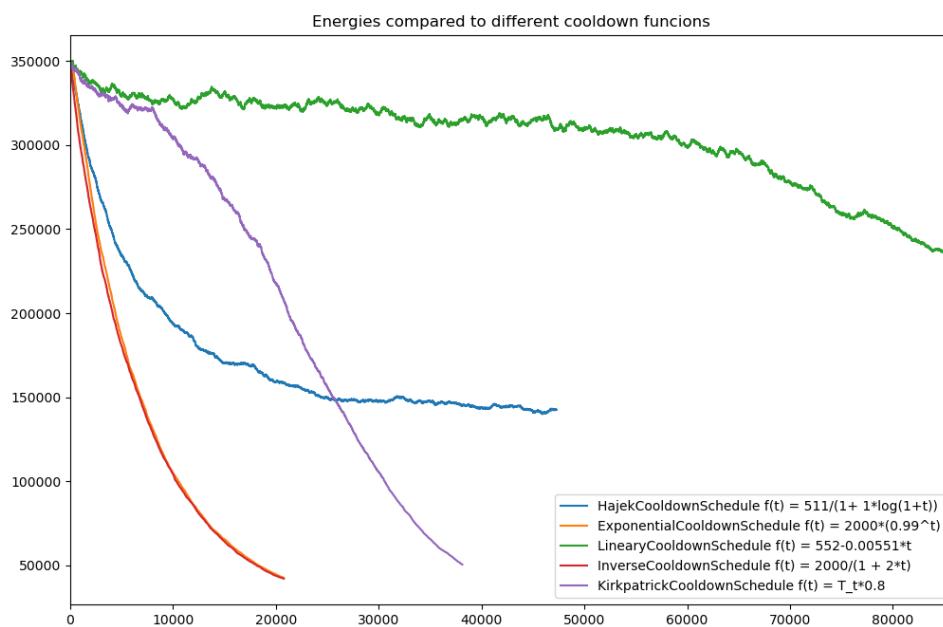


Abbildung 2.17: Vergleich von Abkühlfunktionen mit gesetzten Parametern

To Do

(Hier noch Resumee über die einzelnen Abkühlfunktionen einfügen)

2.5.1.5 Energieverlauf

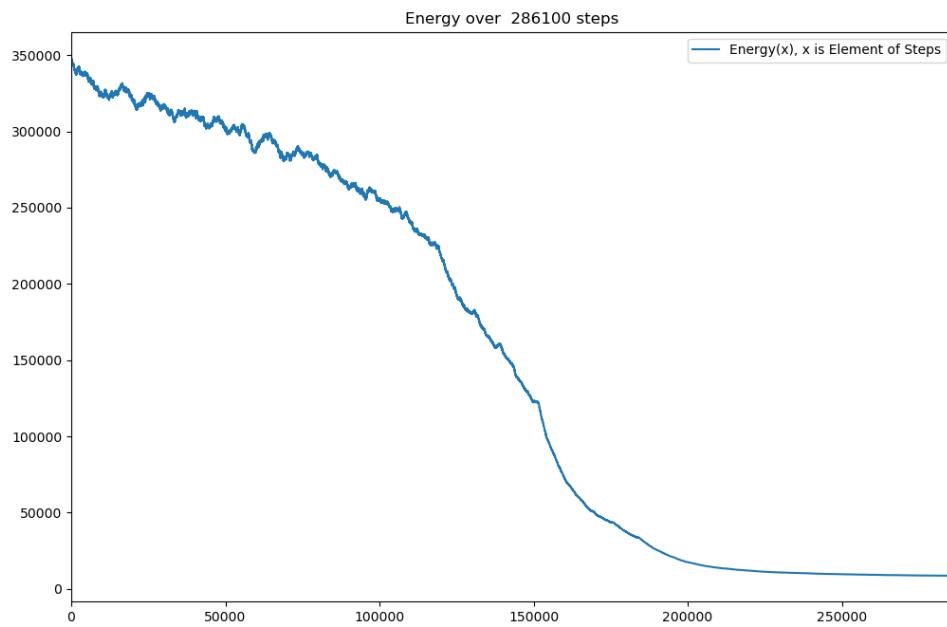


Abbildung 2.18: Energieverlauf beim Simulated Annealing

(Erläutern)

ToDo

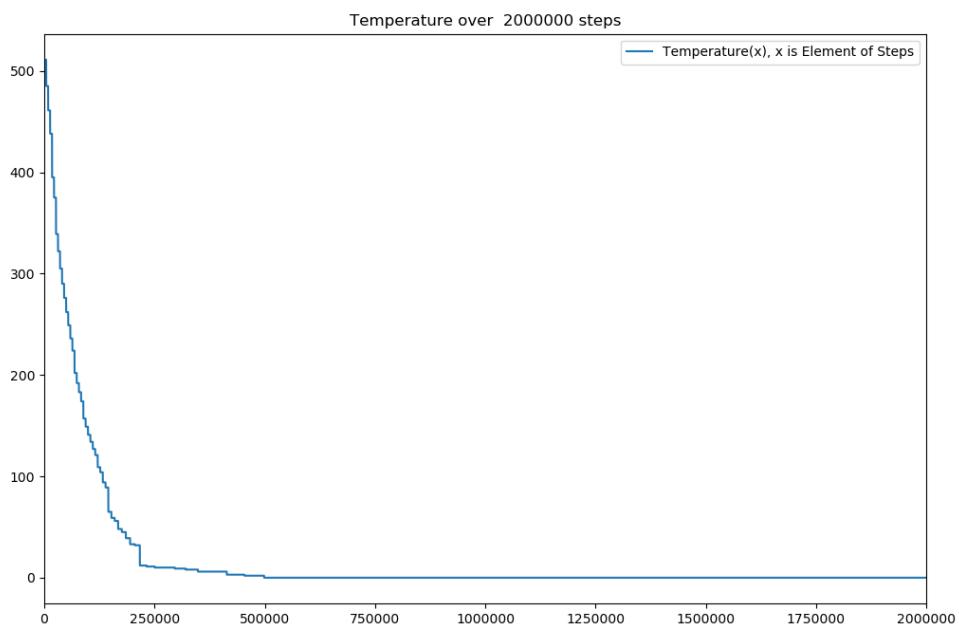


Abbildung 2.19: Temperaturverlauf

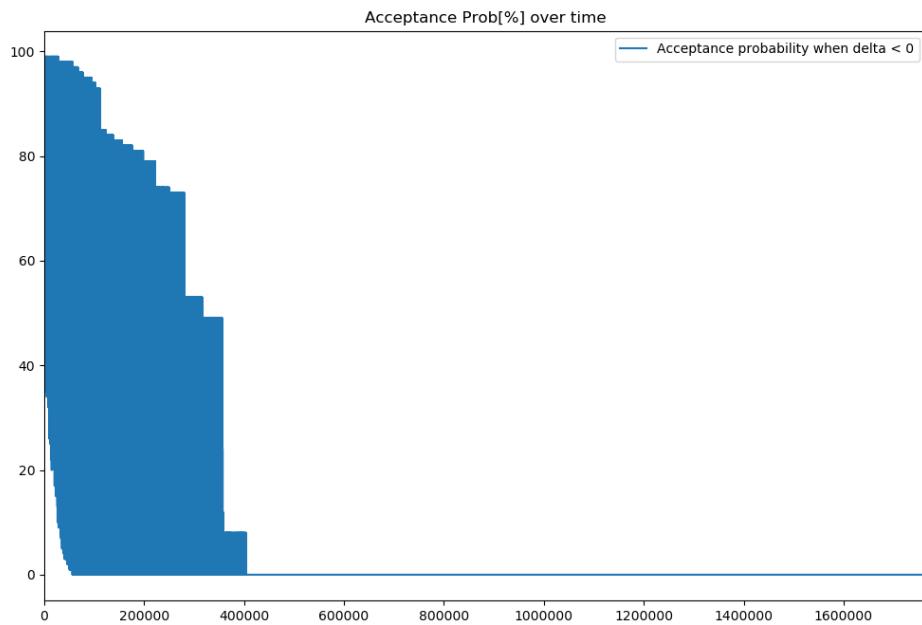


Abbildung 2.20: Akzeptanzfunktionsverlauf bei negativen Energiedeltas

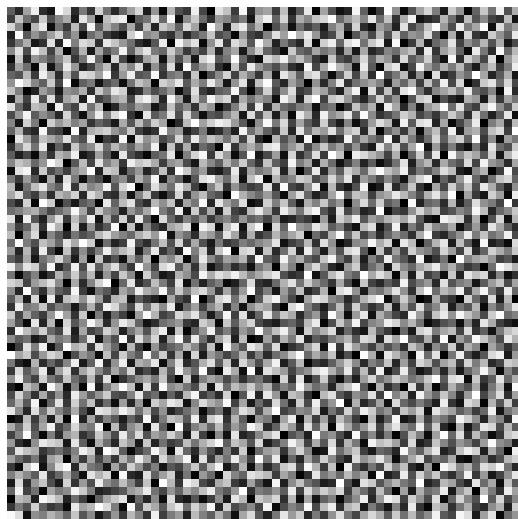


Abbildung 2.21: Blue noise Textur
64x64

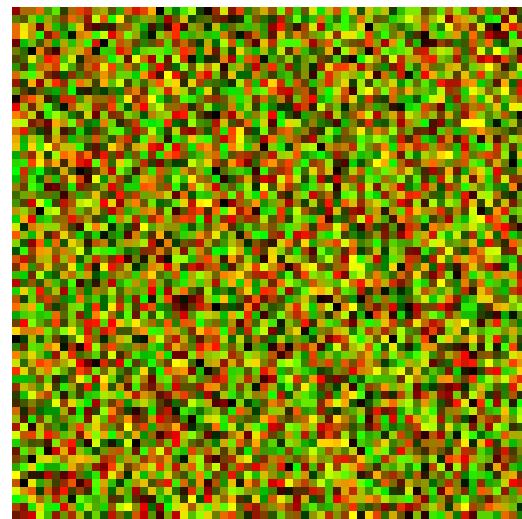


Abbildung 2.22: Permutation; gespeichert in R,G-Channel einer PNG

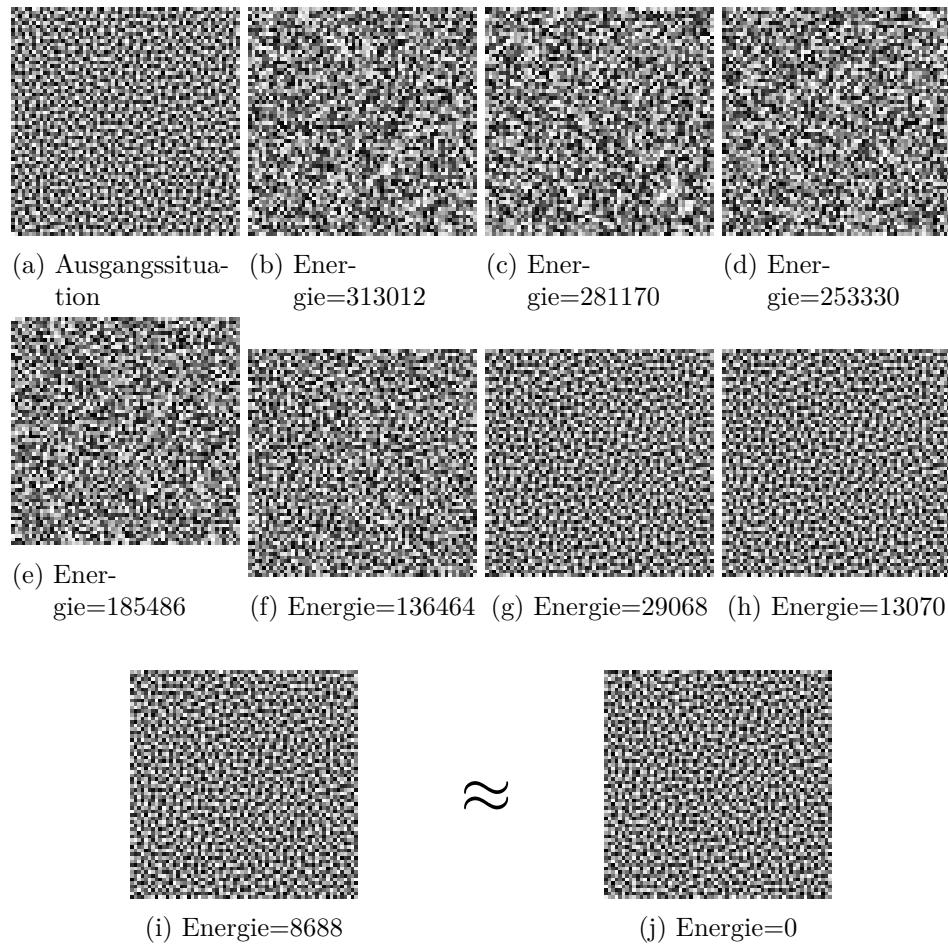


Abbildung 2.23: The process of annealing

3. Temporaler Algorithmus

In diesem Abschnitt wird auf den in [Eric Heitz, 2019] vorgestellten, temporalen Algorithmus eingegangen. Dieser besteht grundsätzlich aus dem Sorting sowie den Retargeting. Es sollte unbedingt beachtet werden, dass folgende Annahmen getroffen wurden: Der Algorithmus arbeitet Blockweise auf den Pixeln und erwartet, dass benachbarte Pixel innerhalb dieses Blockes den selben Wert haben. Da wir einen temporalen Algorithmus haben, soll diese Annahme auch über mehrere gerenderte Bilder hinweg gelten. Es sollte also beachtet werden, dass der Algorithmus z.B. nicht für Objektkanten oder ruckartige Bewegungen (der Kamera oder Objekte) ausgelegt ist. Des Weiteren gehen wir aus den *A Posteriori Eigenschaften* gewonnen Einsicht, dass die Wahl unserer Anfangswerte des Path Tracer unsere Fehlerverteilung im Bildraum beeinflusst, aus. Somit werden wir ein Umsortieren unserer Anfangswerte anhand einer blue noise Textur vornehmen, um so auch für die gerenderten Farbwerte der Pixel eine blue noise Fehlerverteilung zu erhalten.

[Peters, 2016] empfiehlt die Benutzung von 64^2 8-bit Texturen. Eine Benutzung in der Hinsicht, alle 64 bereitgestellten Varianten in ein Array zu laden, jedes Frame ein neues Zufälliges zu verwenden und mit einem zufälligen Offset drauf zuzugreifen. Die Database von Texturen [Peters, 2016] enthält für die empfohlene Auflösung jeweils Varianten mit einer unterschiedlicher Anzahl von Kanälen. Wir wählen die Anzahl der Kanäle anhand der Anzahl der Dimensionen, die gleichzeitig blue noise verteilt werden sollen. Für unsere Variante reicht ein Channel einer Textur. Die gewonnenen Einsicht bei Quasi-Zufallsfolgen erlaubt uns eine Textur zu verwenden (und nicht eine Vielzahl von Texturen in ein Array zu laden) und auf diese mit einem entsprechenden Offset zuzugreifen um entsprechenden Artfakten (Abrisskanten bei Bewegung, wiederholende Strukturen) vorzubeugen. Abbildung 3.1 zeigt uns eine Szene mit zufällig gewählten Seeds und den daraus folgenden weißen Rauschen. Der Szenausschnitt wurde bewusst an einem homogenen Abschnitt gewählt.

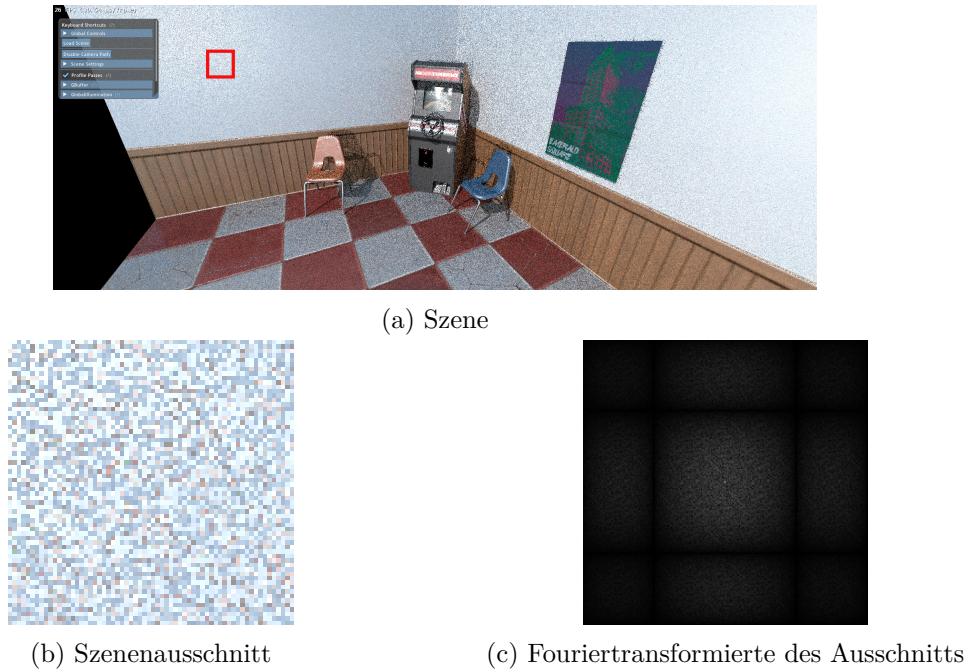


Abbildung 3.1: Wei es Rauschen

3.1 Blue Noise Dithering Sampling

so bekommen wir klassische unkorrelierte Pixelwerte. Resultat hiervon sind typische white noise Fehlerverteilungen. Erkenntnisse aus [Ulichney, 1993b] bringen die Vorteile von anderen Fehlerverteilungen zur Geltung. Blue noise verteilte Fehlerverteilungen im Bildraum schaffen hiernach einen besseren optischen Eindruck f r das menschliche Auge. Die Arbeit [Georgiev and Fajardo, 2016] pr sentiert zum konventionellen gewhlten Offset anhand einer 脿ber das Fenster gekachelten Blue Noise textur w hlt. Im Folgenden wird auf das BNDS (blue noise dithered sampling) und deren *a priori* Eigenschaften eingegangen um anschlie end die A Posteriori Eigenschaften des Temporalen Algorithmus besser zu verstehen.

Die Optimalit t der 1-Dimensionalit t der Sample Anzahl geht aus den Ausf hrungen in [Eric Heitz, 2019, S.3] hervor und hat mit der abnehmenden Bildraumkorrelation der Samples mit gleichzeitig steigender Anzahl zu tun.

Die hohe Dimensionalit t der verwendeten Blue Noise Textur bleibt ein offenes Problem [Peters, 2016]. Deshalb ist eine niedrige Anzahl von Dimensionalit t g nstig. f r das BNDS g nstig und der temporale Algorithmus verwendet eine 1-D Textur.

3.2 A Posteriori

Der nun behandelte temporale Algorithmus von [Eric Heitz, 2019] beruht im Gegensatz zu [Georgiev and Fajardo, 2016] auf *nachträglichen* Annahmen. Welches zur Folge hat, dass die Dimension unseres Path Tracers Path Tracer, sowie die Stichprobenanzahl einhergehen mit der Verteilung der Integrationsfehler als blue noise im Bildraum. Die zu Grunde liegenden Annahmen sollen nun im Folgenden untersucht werden.

3.2.1 Theoretische Grundlage

Im Kapitel Path Tracer haben wir gesehen, dass wir den Wert eines Pixels (i,j) klassischerweise mit einem zufälligen Startwert durch eine Monte-Carlo Integration erhalten. Wir betrachten im Folgenden eine (theoretische) Menge von allen möglichen Werten eines Pixels, welche durch alle möglichen Startwerte generiert wurde. In Theoretische Grundlage ist die Wahrscheinlichkeitsdichtefunktion h_{ij} aufgetragen, als eine Funktion über alle möglichen Werte I_{ij} eines Pixels (i,j) .

$$H_{ij}([I_{Anfang}, I_{Ende}]) = \int_{I_{Anfang}}^{I_{Ende}} h_{ij} dI \quad (3.1)$$

Verfolgt man beispielhaft die Werte eines Pixels über 9 frames bei unseren Path Tracer basierend auf [Benty et al., 2018], so ergibt es sich zur Anschaufung wie folgt:



(a) Szenenausschnitt

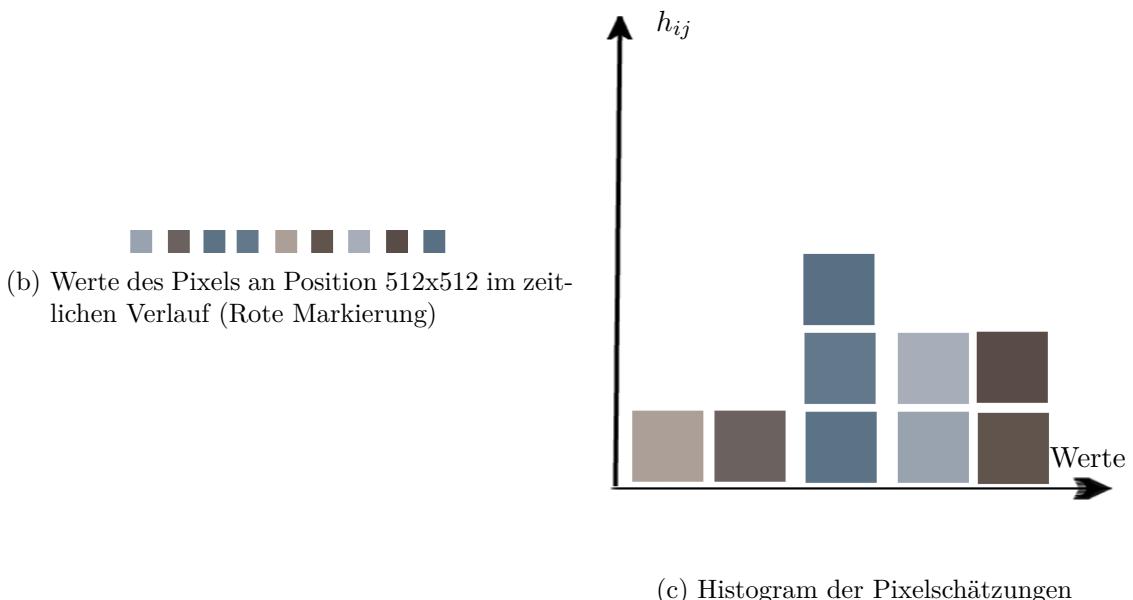


Abbildung 3.2: Pixelwerte an Position 512x512(grüne Markierung) in aufeinanderfolgenden Zeitschritten

Allerdings betrachten wir die theoretische Menge aller Werte. Demnach können wir das Rendern eines jeden konkreten Pixels als die Wahl eines Wertes anhand der Wahrscheinlichkeitsdichtefunktion sehen.

Daraus lässt sich die Gleichbedeutung zweier Aussagen begründen: Das Rendern des Pixels (i,j) und das Wählen eines Pixelwertes I_{ij} von unserer zuvor formulierten Wahrscheinlichkeitsdichtefunktion h_{ij} .

$$I_{ij} = H_{ij}^{-1}(x), x \in [0, 1] \quad (3.2)$$

Nun betrachte man die Werte für x in Theoretische Grundlage als im Bildraum blue noise verteilte Zahlen. Daraus folgt, dass die resultierenden Integrationsfehler auch als blue noise im Bildraum verteilt sind.

3.2.2 Praktische Durchführung

Die Berechnung des vollständigen Histogramms ist für eine Echtzeitanwendung zu kostenintensiv. Stattdessen könnte man auch die dadurch beanspruchte Rechenleistung auf z.B. mehrere Samples pro Pixel verteilen. Stattdessen werden wir in dem temporalen Algorithmus von [Eric Heitz, 2019] das Histogramm mit dem vorherigen Frame approximieren. Bereits vorherige Arbeiten [Schied et al., 2018] haben die Wirksamkeit eines solchen temporalen Ansatzes (Zugriff auf das vorherige Frame) gezeigt. Die Approximation des Histogramms erfolgt dadurch mit dem $Frame_t$ für $Frame_{t+1}$, indem umliegende Pixel in das Histogramm aufgenommen werden. Offensichtliche Konsequenzen dieser blockweisen Verarbeitung sind schlechte blue noise Fehlerverteilungen im Bildraum bei sich stark ändernden Bildausschnitten (so z.B. bei Objektkanten), da dort die Annahme, dass eine ähnliche Oberfläche zur Farbgebung beiträgt verletzt wird.

Algorithm 6 Benutzung unserer zwei vorberechneten Texturen: Blue Noise und Retarget

- 1: $bluenoise_t(i,j) = bluenoise_0(i + \alpha t, j + \beta t);$
 - 2: $retarget_t(i,j) = retarget_0(i + \alpha t, j + \beta t) + (\alpha t, \beta t)$
-

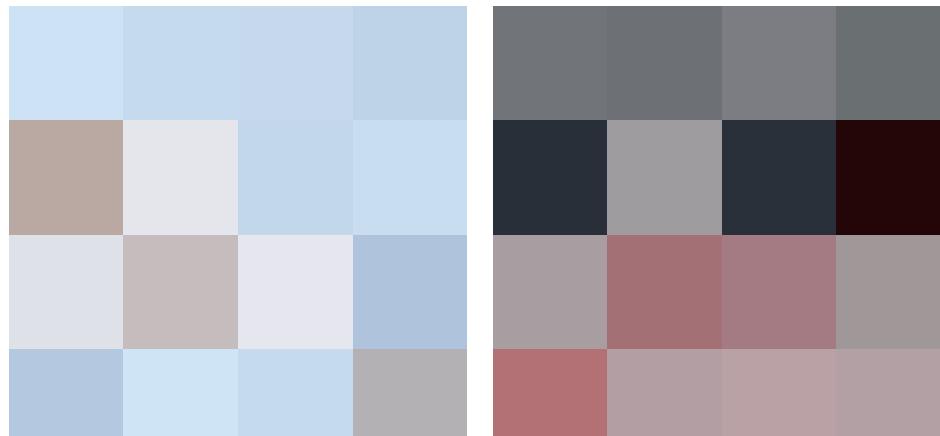


Abbildung 3.3: Pixelblöcke bei (in-)homogenen Flächen

3.3 Sorting

In diesem Schritt wollen wir nun die Untersuchungen aus A Posteriori durchführen. Nach dem Rendern eines $Frame_t$ (vor dem Rendern von $Frame_{t+1}$) approximieren wir das Histogramm der Pixelwerte anhand der Pixelwerte von $Frame_t$. Dabei betrachten wir eine Anzahl Pixel pro BLOCK(=16 nach [Eric Heitz, 2019]) in der unmittelbaren Nachbarschaft und nehmen diese wie anfangs erwähnt als Schätzung des Histogramms.

[Eric Heitz, 2019]

Algorithm 7 Sortier Schritt t nach dem Rendern von Frame t und vor dem Rendern von Frame t+1

```

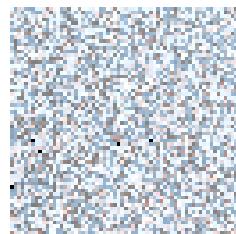
1: pixel consists of value,index;
2: List framePixelsIntensities, noiseIntensities;
3: assert(sizeof(framePixelsIntensities) == BLOCKSIZE);
4: assert(sizeof(noiseIntensities) == BLOCKSIZE);
5: List L  $\leftarrow$  pixels of frame t in block;
6:
7: //init lists
8: initList(framePixelsIntensities, pixelIntensity(L));
9: blueNoiset = calcCorrectOffset(incomingbluenoisetexture);
10: initList(noiseIntensities, pixelIntensity(blueNoiset));
11:
12: //sort the two lists by means of intensities
13: sort(framePixelsIntensities);
14: Sort(noiseIntensities);
15:
16: //now we reorder our seeds hence the sorted lists
17: for i = 1..BLOCKSIZE do
18:   sortedSeeds(noiseIntensities.getIndex(i)) = incomingSeeds(framePixelIntensities.getIndex(i))
19: end for
```

Hierbei muss noch eine wichtige Anmerkung gemacht werden. Die Fehlerverteilung der Pixelwerte im Bildraum konvergiert auf diese Weise nicht zu einer blue noise Verteilung, denn wir wechseln in jedem Frame die verwendeten blue noise Texturen(theoretisch! praktischerweise werden wir hier eine Textur verwenden und mit Erkenntnissen aus Quasi-Zufallsfolgen) quasi-zufällig zugreifen um so einen solchen Effekt zu erreichen). Dieser Schritt alleine reicht also nicht für den erwünschten Effekt zu erreichen.

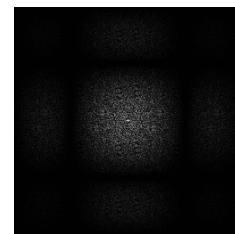
Die APosteriori Erkenntnisse zu der inversen Funktion3.2 garantieren uns nach dem Um-sortieren die entsprechenden Seeds in einer ebenfalls blue noise verteilten Struktur zu erhalten.



(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

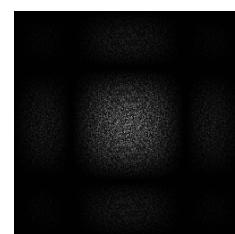
Abbildung 3.4: Zeitpunkt t=1



(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

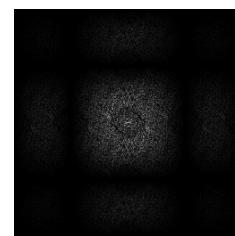
Abbildung 3.5: Zeitpunkt t=2



(a) Szene



(b) Szenenausschnitt



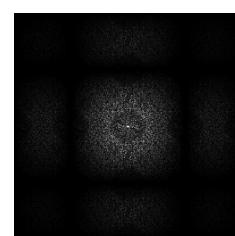
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.6: Zeitpunkt $t=3$ 

(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

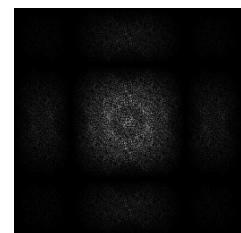
Abbildung 3.7: Zeitpunkt $t=4$



(a) Szene



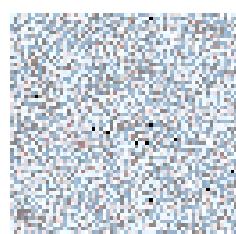
(b) Szenenausschnitt



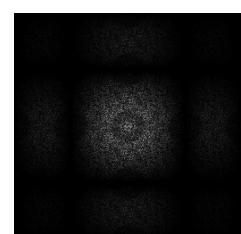
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.8: Zeitpunkt $t=5$ 

(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

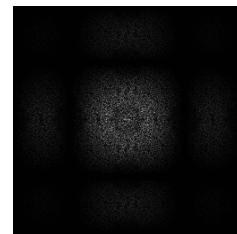
Abbildung 3.9: Zeitpunkt $t=5$



(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

Abbildung 3.10: Zeitpunkt t=5

3.4 Retargeting

Zu Grunde liegender Sinn dieses Schrittes: Vertauschen der Seeds, die verteilt sind wie $BlueNoise_t$, aufgrund des zuvor ausgeführten Sortierschrittes 3.3, sodass Sie verteilt sind wie die Textur $BlueNoise_{t+1}$. Aufgrund dessen haben wir eine Aufsummierung der blue noise Fehlerverteilungen über die ersten paar Frames(siehe Szene).

Algorithm 8 Retargeting Schritt

```
1: //permutation indices from precomputed texture
2: retagett = retarget_texture[calc_correct_offset()];
3: retargetedSeeds(old_id + retagett) = incomingSeeds(old_id);
```

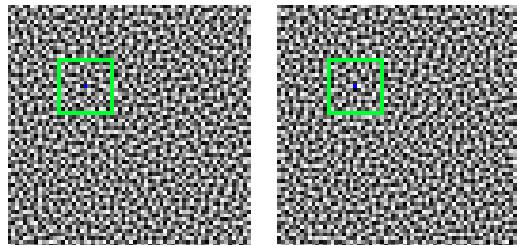


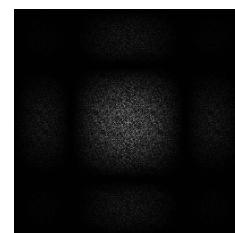
Abbildung 3.11: Permutation



(a) Szene



(b) Szenenausschnitt



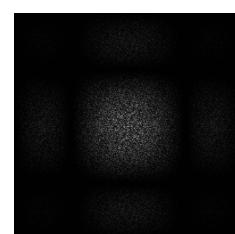
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.12: Zeitpunkt $t=1$ 

(a) Szene

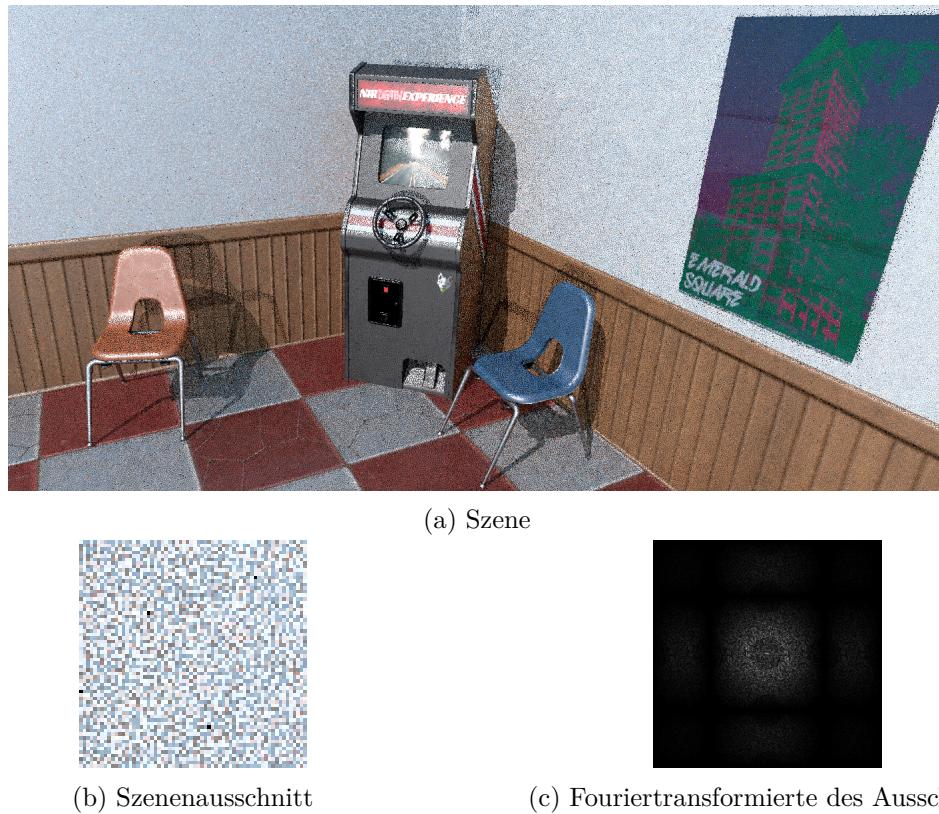
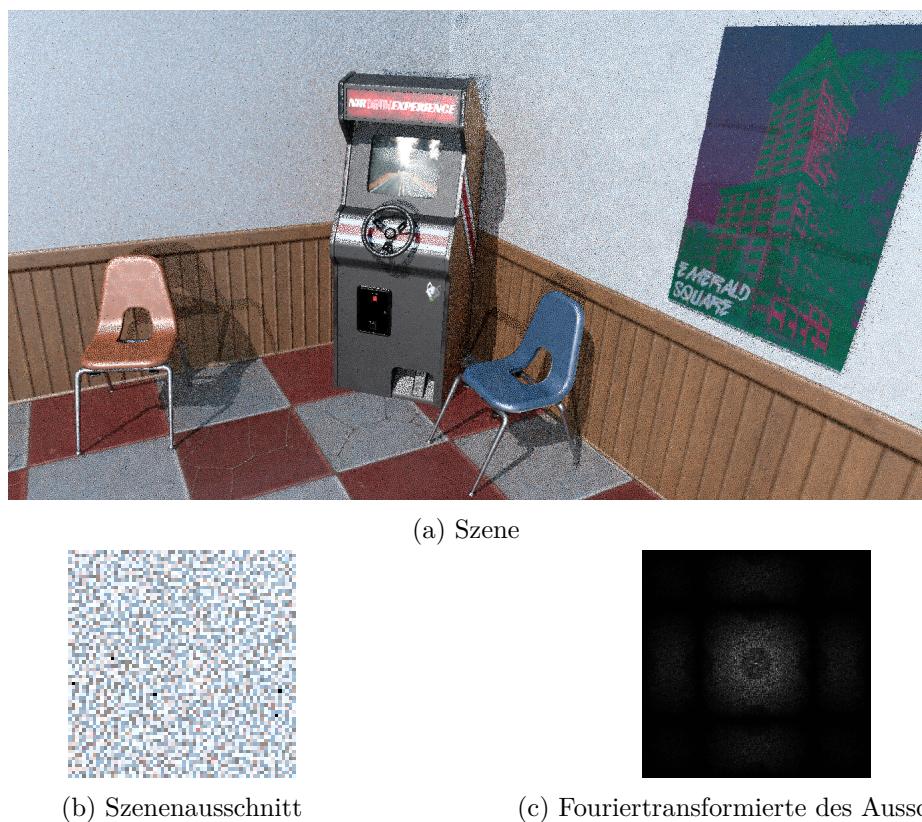


(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

Abbildung 3.13: Zeitpunkt $t=2$

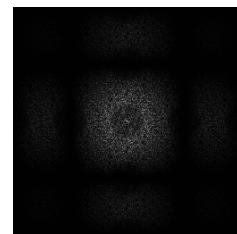
Abbildung 3.14: Zeitpunkt $t=3$ Abbildung 3.15: Zeitpunkt $t=4$



(a) Szene



(b) Szenenausschnitt



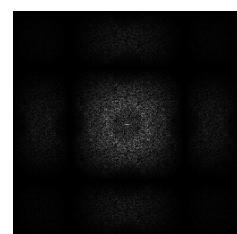
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.16: Zeitpunkt $t=5$ 

(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

Abbildung 3.17: Zeitpunkt $t=6$

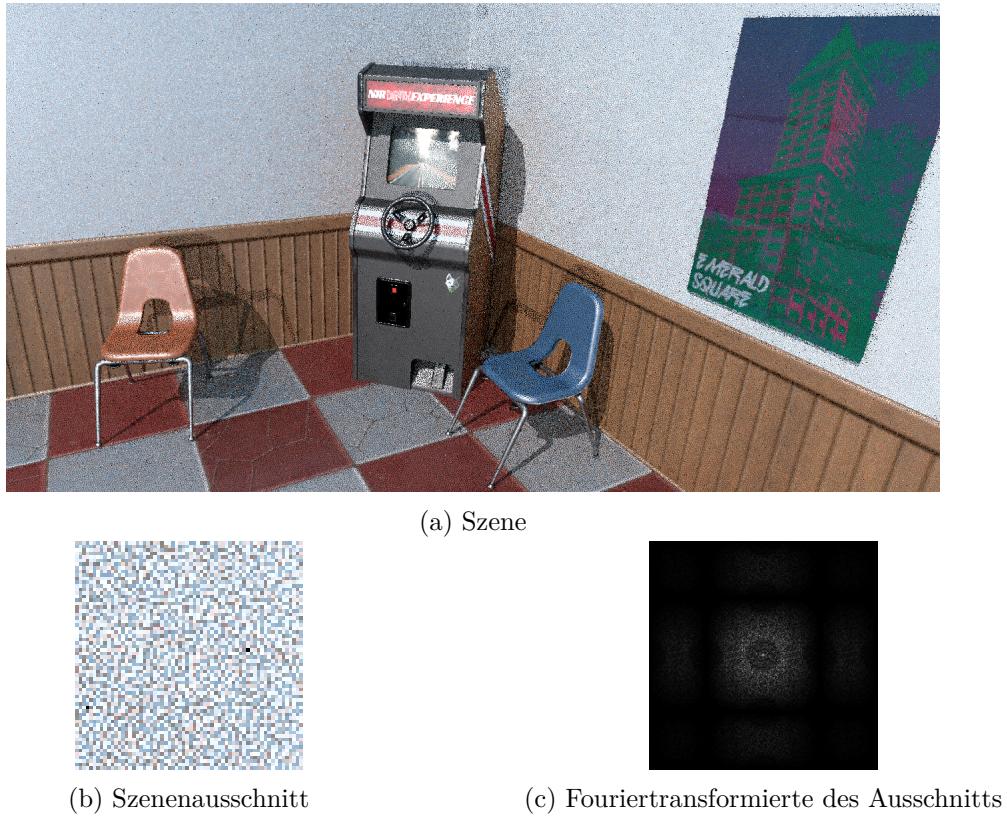


Abbildung 3.18: Zeitpunkt t=7

3.5 Rechenaufwand

Da unsere Ressourcen beschränkt sind und trotz Hardwarebeschleunigung immer noch viel Rechenzeit eines Frames auf die globale Beleuchtung entfällt, ist es von großer Bedeutung, dass unser temporaler Algorithmus keinen signifikanten zusätzlichen Aufwand schafft. Mit einem Großteil der Rechenzeit, der auf die Berechnung des GBuffer und der globalen Beleuchtung fällt sind wir hingegen mit den Schritten Sorting und Retargeting sowohl auf CPU als auf GPU Seite im niedrigen einstelligen Prozentbereich.

Pipelinstage	Rechenzeit(ms/%) CPU	Rechenzeit(ms/%) GPU
Gesamt	29.87/100%	17.76/100%
GBuffer	06.48/21.7%	01.30/7.32%
Retargeting	01.12/3.7%	00.33/1.9%
GGXGlobalIllumination	21.20/70.97%	15.51/87,33%
Sorting	00.94/3,14%	00.63/3,55%

Tabelle 3.1: Rechenzeiten die auf die einzelnen Stages fallen

* Hardware: AMD Ryzen 5 2600X, NVIDIA GeForce RTX 2060 SUPER

)

Literaturverzeichnis

- [JCr, 2018] (2018). Jcrystal. performing fft on images.
- [Ray, 2019] (2019). Rayflag enumeration. control behaviour of a traced ray.
- [coo, 2019] (2019). Simulated annealing cool down schedules. different cool down schedules.
- [Whi, 2019] (2019). Whitenoisegenerator. <https://www.cssmatic.com/noise-texture>. Accessed: 24.11.2019.
- [Sci, 2020] (2020). Cooling schedule. very good overview and further explanation of kirkpatrick.
- [Dis, 2020] (2020). Disneys guide to path tracing. very good overview and further explanation of path tracing.
- [Akenine-Moller et al., 2008] Akenine-Moller, T., Haines, E., and Hoffman, N. (2008). *Real-time rendering*. AK Peters/CRC Press.
- [Barré-Brisebois et al., 2019] Barré-Brisebois, C., Halén, H., Wihlidal, G., Lauritzen, A., Bekkers, J., Stachowiak, T., and Andersson, J. (2019). *Hybrid Rendering for Real-Time Ray Tracing*, pages 437–473. Apress, Berkeley, CA.
- [Benty et al., 2018] Benty, N., Yao, K.-H., Foley, T., Oakes, M., Lavelle, C., and Wyman, C. (2018). The Falcor rendering framework. <https://github.com/NVIDIAGameWorks/Falcor>.
- [Caflisch, 1998] Caflisch, R. E. (1998). Monte carlo and quasi-monte carlo methods. *Acta Numerica*, 7:1–49.
- [Drettakis and Seidel, 2002] Drettakis, G. and Seidel, H.-P. (2002). Efficient multidimensional sampling. 21:1–8.
- [Eric Heitz, 2019] Eric Heitz, L. B. (2019). Distributing monte carlo errors as a blue noise in screen space by permuting pixel seeds between frames. 38:1–10.
- [Games, 2017] Games, E. (2017). The problem with 3d blue noise. Blogpost.
- [Georgiev and Fajardo, 2016] Georgiev, I. and Fajardo, M. (2016). Blue-noise dithered sampling. In *ACM SIGGRAPH 2016 Talks*, page 35. ACM.
- [Haines and Akenine-Möller, 2019] Haines, E. and Akenine-Möller, T., editors (2019). *Ray Tracing Gems*. Apress. <http://raytracinggems.com>.
- [Hajek, 1988] Hajek, B. (1988). Cooling schedules for optimal annealing. *Mathematics of operations research*, 13(2):311–329.
- [Kirkpatrick et al., 1983] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.
- [Kraft, 2018] Kraft, B. (2018). Aufbau turing architektur. blogpost.

- [Krcadinac, 2006] Krcadinac, V. (2006). A new generalization of the golden ratio. *Fibonacci Quarterly*, 44(4):335.
- [Marschner and Shirley, 2009] Marschner, S. and Shirley, P. (2009). *Fundamentals of computer graphics*. CRC Press.
- [Padovan, 2002] Padovan, R. (2002). Dom hans van der laan and the plastic number. *Nexus IV: Architecture and Mathematics*, pages 181–193.
- [Peters, 2016] Peters, C. (2016). Free blue noise textures. blogpost.
- [Roberts, 2018] Roberts, M. (2018). The unreasonable effectiveness of quasirandom sequences. <http://extremelearning.com.au/unreasonable-effectiveness-of-quasirandom-sequences/>.
- [Schied, 2019] Schied, C. (2019). Real-time path tracing and denoising in quake 2. Game Developer Conference.
- [Schied et al., 2018] Schied, C., Peters, C., and Dachsbacher, C. (2018). Gradient estimation for real-time adaptive temporal filtering. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(2):24:1–24:16.
- [Ulichney, 1988] Ulichney, R. A. (1988). Dithering with blue noise. *Proceedings of the IEEE*, 76(1):56–79.
- [Ulichney, 1993a] Ulichney, R. A. (1993a). Void-and-cluster method for dither array generation. In *Human Vision, Visual Processing, and Digital Display IV*, volume 1913, pages 332–343. International Society for Optics and Photonics.
- [Ulichney, 1993b] Ulichney, R. A. (1993b). Void-and-cluster method for dither array generation. In Allebach, J. P. and Rogowitz, B. E., editors, *Human Vision, Visual Processing, and Digital Display IV*, volume 1913, pages 332 – 343. International Society for Optics and Photonics, SPIE.
- [Van Laarhoven and Aarts, 1987] Van Laarhoven, P. J. and Aarts, E. H. (1987). Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer.

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den 18. Februar 2020

(Jonas Heinle)