

# 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

<b>1 Prelude</b>	<b>1</b>
1.1 Abstract . . . . .	1
1.2 Einleitung . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 Rasterisierung . . . . .	3
2.1.1 Beschränktheit . . . . .	4
2.2 Path Tracer . . . . .	5
2.2.0.1 Funktionsweise . . . . .	5
2.2.0.2 Monte-Carlo-Integration . . . . .	6
2.2.0.3 DirectX Raytracing . . . . .	6
2.2.1 RenderGraph . . . . .	8
2.3 Blue Noise . . . . .	9
2.3.1 Eigenschaften . . . . .	9
2.3.1.1 Uniformität . . . . .	9
2.3.1.2 Isotropie . . . . .	10
2.3.1.3 Niedrige Frequenzen . . . . .	10
2.3.1.4 Kachelung . . . . .	11
2.4 Quasi-Zufallsfolgen . . . . .	13
2.4.1 Einleitung . . . . .	13
2.4.2 Low Discrepancy Sequenzen . . . . .	13
2.4.3 Quasi Monte-Carlo Methode . . . . .	13
2.4.4 1-Dimension . . . . .	13
2.4.5 2-Dimensionen . . . . .	14
2.4.6 Dither Texturen und quasi-zufällige Folgen . . . . .	14
2.5 Simulated Annealing . . . . .	15
2.5.0.1 Allgemein . . . . .	15
2.5.1 Abkühlfunktion . . . . .	17
2.5.1.1 Hajek . . . . .	17
2.5.1.2 Linear . . . . .	17
2.5.1.3 Exponential . . . . .	17
2.5.1.4 Inverse . . . . .	18
2.5.1.5 Energieverlauf . . . . .	19
<b>3 Temporaler Algorithmus</b>	<b>20</b>
3.1 Blue Noise Dithering Sampling . . . . .	21
3.2 A Posteriori . . . . .	21
3.2.1 Theoretische Grundlage . . . . .	21
3.2.2 Praktische Durchführung . . . . .	23
3.3 Sorting . . . . .	23
3.4 Retargeting . . . . .	28
3.5 Rechenaufwand . . . . .	32





# 1. Prelude

## 1.1 Abstract

Die Bildberechnung durch hardwareunterstützte Strahlverfolgung 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 kleinerer 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 den aktuellen Übergang in Echtzeitanwendungen, indem es in einem konventionellen Spiel die (teilweise) konventionelle Bilderzeugung mit neuen Technologien des *Real-Time Raytracing* ausstauscht.

Abschnitt 2.1 beschreibt die bisherige, konventionelle Herangehensweise und zeigt die Limitierungen Dessen 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 [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 3.3 und Retargeting 3.4, lassen sich unsere Pixel im Bildraum so korrelieren, das eine zeitlich stabile Fehlerverteilungen entsteht. Dabei machen wir uns die Erkenntnisse aus 2.4 zu nutze um nur eine Textur anstatt Mehreren zu nutzen aber dennoch denselben Effekt zu haben. 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 [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.

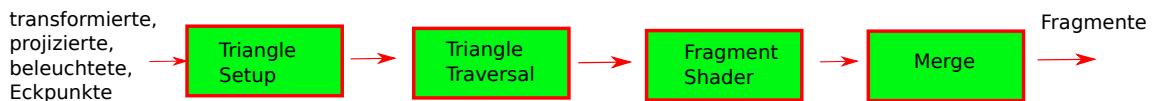


Abbildung 2.1: Ablauf der Rasterisierung

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

Ihre bisherige weite Verbreitung hatte die Rasterisierung der Objektorientierung zu verdanken: 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. Ihr großer Nachteil bzw. Beschränktheit liegt jedoch genau an dieser Objektorientierung. 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. So können Effekte, welche diese Sekundärstrahlen involvieren, entweder nicht oder nur (unzureichend befriedigend) dargestellt werden (z.B. Schatten, Spiegelungen).

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 erste image-ordered Bilderstellungen. Aktuelle Bemühungen gehen nun daran Raytracing und Rasterisierung zu kombinieren. [Barré-Brisebois et al., 2019] stellte mit dem Spiel *PICA PICA* eine solche Rendering-Pipeline vor, welche mithilfe von Path Tracing 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

Bei der Bilderzeugung, ausgehend von Szenen, welche viel Geometrie beinhalten bzw. bei Szenen die generelle BRDF's verwenden eignet sich der Path Tracer. Der Path Tracer ist in Hinsicht der Beleuchtung komplett. Deshalb lässt sich damit *Global Illumination* erreichen. Der hier verwendete Path Tracer in [Benty et al., 2018] verwendet eine klassische Umsetzung.

Der Path Tracer 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  $\epsilon$  von  $x'$  nach  $x$ . Beeinflusst wird der Energiefluss auch durch die bidirektionale Verteilungsfunktion  $\rho$ , 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 Licht zu einem Anderen ist die Summe des ausgestrahlten Lichts und das ausgestrahlte Licht zu  $x$  von allen anderen Oberflächen.

Ausgehend von der Rendergleichung Funktionsweise lässt sich die vollständige Transportgleichung Funktionsweise des Path Tracer 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.

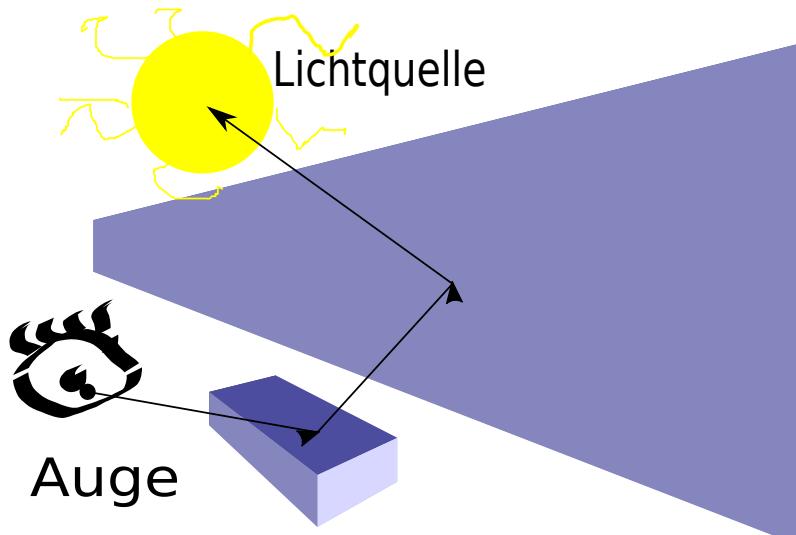


Abbildung 2.2: Grundkonzept path tracer

### 2.2.0.2 Monte-Carlo-Integration

Mit der Monte Carlo Integration approximieren wir die Rendergleichung.

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 IntegralFunktionsweise approximiert. Die Dichtefunktion  $\rho(x_i)$  beschreibt 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] Konvergenzrate, unabhängig von der Dimension unseres Tracers.  $O(N^{-\frac{1}{2}})$ . Sie ist robust, das heißt Exaktheit hängt nur vom ungenauesten Parameter ab. Eine Variante des Verfahrens, Monte Carlo Quadratur, wird mit quasi zufälligen Sequenzen Quasi-Zufallsfolgen, welche eine niedrige Abweichung aufweisen, durchgeführt. Laufzeit quasi-Monte Carlo  $O((\log N)^k N^{-1})$ . Um die Konvergenzrate zu steigern liegen eine Reihe von Varianzreduktionsmethoden vor.

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

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

Die Varianz Monte-Carlo-Integration ist ein quadratischer Fehler.

### 2.2.0.3 DirectX Raytracing

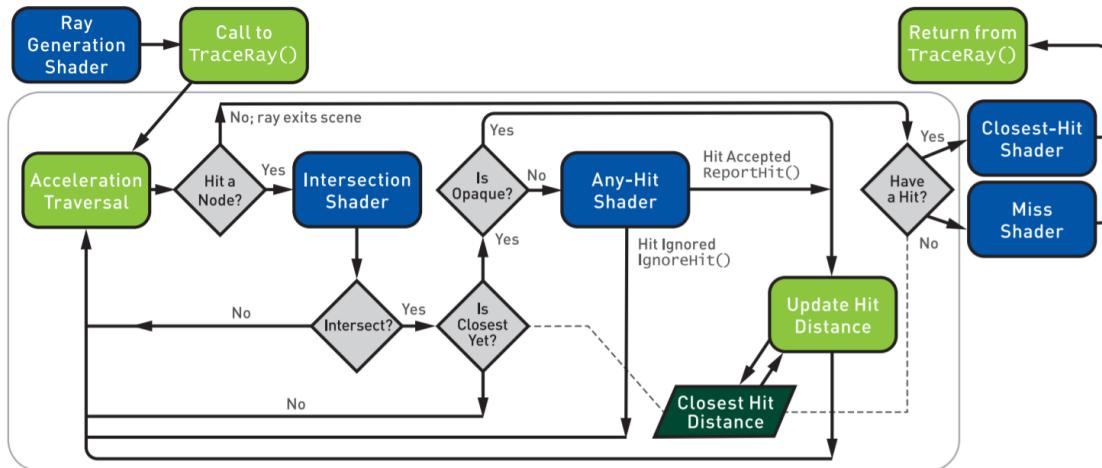


Abbildung 2.3: DirectX Raytracing Pipeline aus [Haines and Akenine-Möller, 2019]

In 2.3 lässt sich der Beginn (Generierung eines Strahles) der neuen Pipeline durch den programmierbaren **Ray Generation shader** erkennen.

---

**Algorithm 1** 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* Typ können wir einen struct mit Informationen jedem einzelnen Strahl mitgeben.

---

**Algorithm 2** beispielhafter payload

---

```

1: struct RayPayload =
2: float4 color, uint32 seed, uint32 depth
3: ;

```

---

Diese Methode **TraceRay()** kann auch innerhalb der anderen Shader zum weiteren verschießen von Strahlen verwendet werden. Beispiel beim Verschiessen vom Schattenstrahl: Flags RAY\_FLAG\_ACCEPT\_FIRST\_HIT\_AND\_END\_SEARCH, RAY\_FLAG\_SKIP\_CLOSEST\_HIT\_SHADER setzen, um unnötige Shadingberechnungen und weitere Schnittpunktberechnungen zu umgehen und mit einem Bit als payload die Sichtbarkeit zur Lichtquelle mitgeben.

Mit diesem beispielhaften payload können wir die Farbe akkumulieren, unseren verwendeten seed verwenden um z.B eine weiteren Strahlenschuss in einem Any-Hit Shader zu verwirklichen, solange die mit übergebene Rekursionstiefe 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 3** 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. Im Folgenden Path Tracing Algorithmus nochmal vereinfacht in Pseudocode dargestellt, wobei die entsprechenden shader im jeweiligen Codeabschnitt markiert sind.

---

#### Algorithm 4 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

```

---

### 2.2.1 RenderGraph

**ToDo**

(Give short introduction in the used rndergraph with its steps)

## 2.3 Blue Noise

[Ulichney, 1988] Ulichney gibt eine Einführung zu *Dithering mit blue noise*. Darunter ist ein abbilden beliebiger Grauwerte zu einer Menge von blue noise verteilten Schwarz- und Weißwerten zu verstehen. Somit kann ein für das menschliche Auge gutes Resultat von Grauwerten entstehen, indem nur Schwarz-/ Weißpixel verwendet werden. Denn das menschliche Auge tendiert dazu, benachbarte Pixel verschwimmen zu lassen und einen Farbwert aus diesen zu generieren. Hat man also einen Grauwert  $p \in [0, 1]$  und will Diesen mit Schwarz-/Weißpixeln approximieren vergleicht man diesen Wert  $p$  mit den Werten aus der Textur und gibt Schwarz (falls  $\text{Wert aus der Textur} \leq p$ ) oder Weiß (falls  $\text{Wert aus der Textur} > p$ ) aus.

### 2.3.1 Eigenschaften

Die in [Games, 2017] vorgestellten blue noise Texturen und ihre Eigenschaften geben Aufschluss über ihre Wirksamkeit. Deshalb werden im Folgenden, die dort bereit gestellten Texturen verwendet, welche anhand des in [Ulichney, 1993a] vorgestellten Algorithmus erstellt wurden. Die korrespondierenden Spektren wurden mit Hilfe von [JCr, 2018] erstellt.

#### 2.3.1.1 Uniformität

Wie bereits erwähnt, entsteht der neue Grauwert anhand einer Mittlung über mehrere benachbarte Pixel. Aufgrund dessen muss für die Wahrscheinlichkeitsfunktion, dass ein schwarzer Pixel bei der Generierung ausgegeben wird ( $p \in [0, 1]$ ) gelten:

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

Die Uniformität(lat. *uniformitas*-Einförmigkeit) garantiert uns dieses Verhalten  $\forall p \in [0, 1]$ . 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,

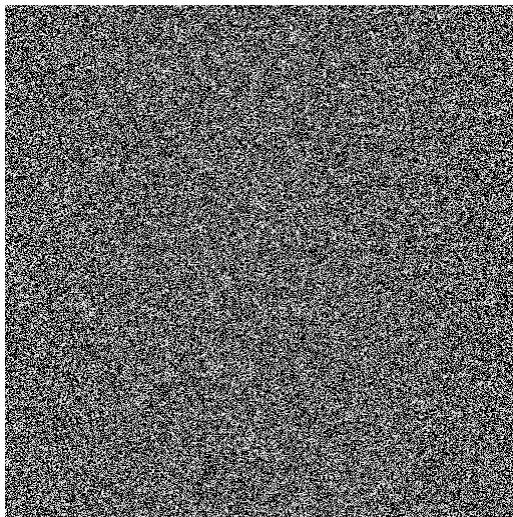


Abbildung 2.4:  $512^2$  white noise Textur

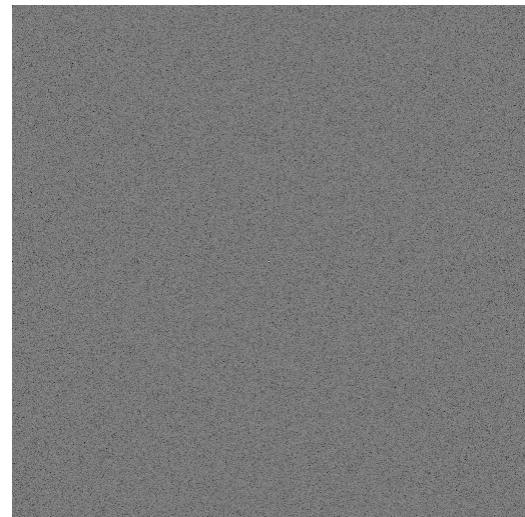


Abbildung 2.5: Amplitudenspektrum  
 $512^2$  white noise Textur

ergibt sich eine typische Amplitudendichte. Zufällig verteilt, über alle Frequenzen hinweg. Die folgende Rotationssymmetrie lässt sich [Kiencke and Jäkel, 2009] erklären, da wir die

Dimension der Phase des Signals nicht betrachten. Allerdings lassen sich noch deutlich ähnlichfarbende Pixelverbünde erkennen.i.e niedere Frequenzen in der Frequenzdomäne erkennen.

### 2.3.1.2 Isotropie

Die Isotropie(altgr. *isos*-gleich und *tropos*-Richtung) einer blue noise Textur wird ausgenutzt. Dabei haben wir in allen Dimensionen (in dieser Arbeit werden Texturen mit zwei benutzt) die Unabhängigkeit einer Eigenschaft. Um uns dies an einem Gegenbeispiel klar zu machen, schauen wir uns das Bayer-Pattern, sowie seine korrespondierende Amplitudendichte an.

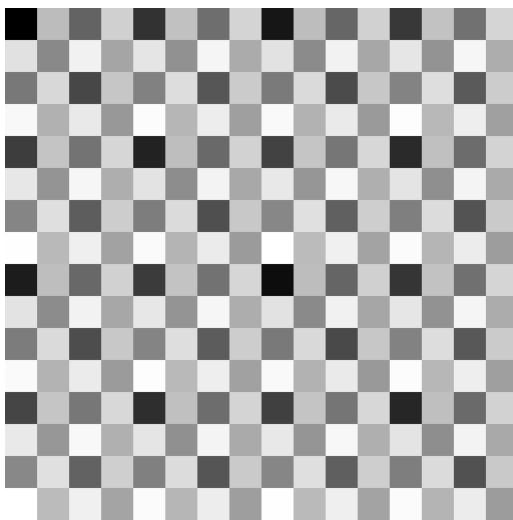


Abbildung 2.6:  $512^2$  bayer pattern Textur

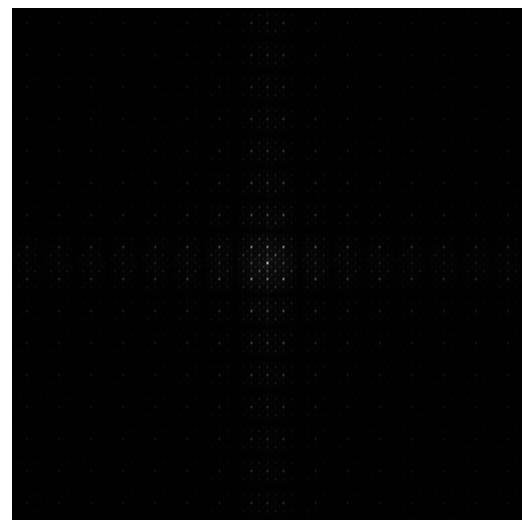
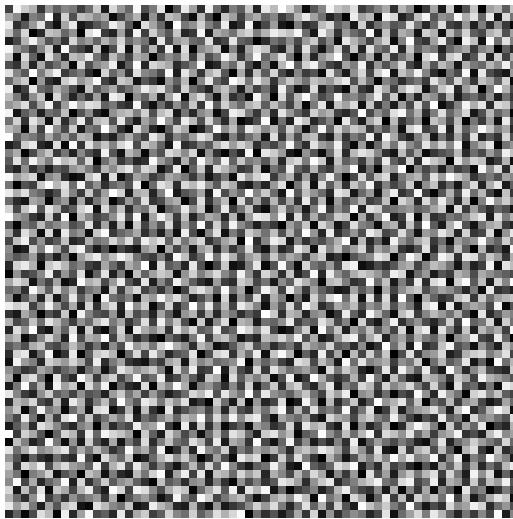
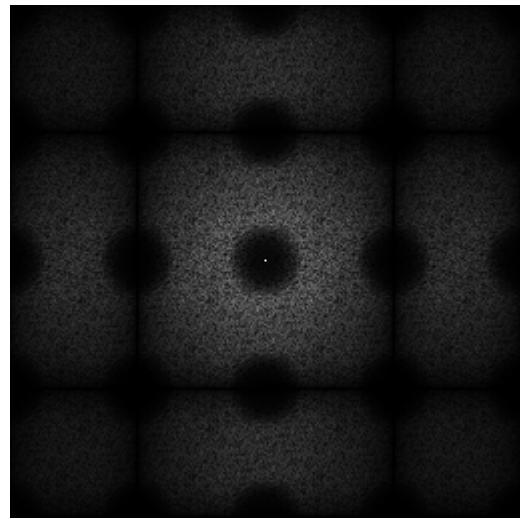


Abbildung 2.7: Amplitudendichte  $512^2$  bayer pattern Textur

In der Frequenzdomäne ist zu erkennen, das 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(Isotropie!) Verteilung. Durch dieses Bayer Pattern entstehen unbefriedigende Artefakte in Echtzeitanwendungen, so in (aktuellen) Spielen [Wronski, 2016] zu sehen.

### 2.3.1.3 Niedrige Frequenzen

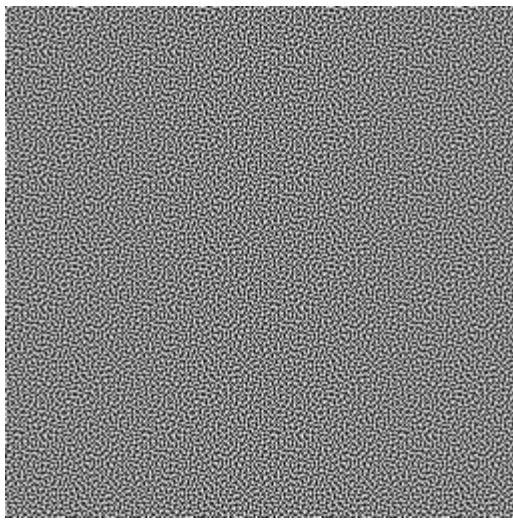
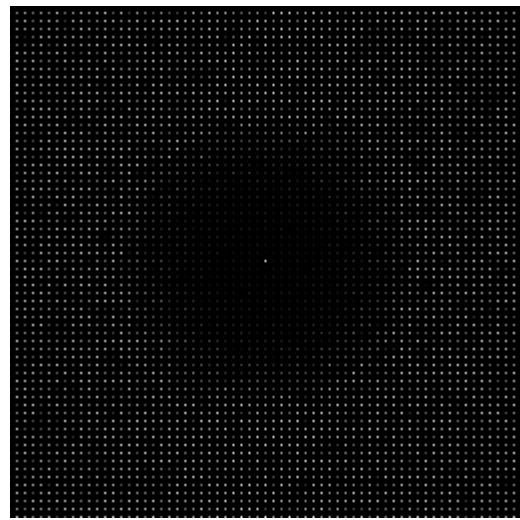
Niedrige Frequenzen sind in einer blue noise sehr wenig bis gar nicht vertreten. Dies ist an dem schwarzen Ring innerhalb der Amplitudendichte zu erkennenNiedrige Frequenzen. Oder in der Zeitdomäne, an dem Abhandensein von gleichfarbigen Pixelbündeln. Genau dies wollten wir erreichen. Außerdem haben wir aus den vorherigen Beispielen gesehen: Wir wollen eine Uniformität Uniformität und eine gleichmäßige Verteilung in allen Richtungen.

Abbildung 2.8:  $512^2$  blue noise TexturAbbildung 2.9: Fourier Spektrum  $512^2$  blue noise Textur

Wie in der Abbildung zu sehen ist, haben wir hier die erwünschte Rotationssymmetrie(Isotropie). Außerdem ist die Uniformität wie bei der white noise (bloß ausschließlich bei höheren Frequenzen) zu erkennen.

#### 2.3.1.4 Kachelung

Bayer Pattern sowie auch white noise lassen sich einfach zur Echtzeit berechnen. Bei blue noise texturen sieht das hingegen anders aus. Für diese Art von Textur müssen wir vor Start der Anwendung eine Vorberechnung machen. Daher stellt sich nun die Frage, wie groß (welche Auflösung) die Textur haben sollte. 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 2.10:  $512^2$  gekachelte Textur von 64x64Abbildung 2.11: Fourier Spektrum  $512^2$  4-fach getiled 64x64 Textur

In Kachelung lässt sich der blue noise Charakter wieder anhand des wenigen niederen Frequenzanteils erkennen. Wiederholungen sind in der Zeitdomäne schwerer zu erkennen,

wohingegen man bereits bekannten Effekt von Bayer Pattern im Frequenzbereich Isotropie, also Wiederholungen, erkennen kann. Jedoch weniger deutlich, weshalb man diese Kacheln als eine gute Approximation für eine entsprechend größere Textur halten kann. Vorallem in Anbetracht der bereits angesprochenen Performancevorteile.

## 2.4 Quasi-Zufallsfolgen

[Owen, 1998] [Heitz et al., 2019]

### 2.4.1 Einleitung

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. Dabei behalten wir die Eigenschaft einer zufälligen Folge, den gesamten Platz gleichmäßig auszufüllen. Diese Eigenschaften erinnern uns an die besprochenen Eigenschaften bei Blue Noise. Im Folgenden wird für uns der zweidimensionale Fall wichtig sein, weswegen wir vom ein- über zum zweidimensionalen schauen werden.

### 2.4.2 Low Discrepancy Sequenzen

Aus dem Vorhandensein einer low discrepancy Sequenz folgt dass auch alle Subsequenzen derartiger Gestalt sind: Gemessen am Anteil der Punkte innerhalb einer (Sub-)sequenz.

$$\sup \left| \frac{|s_1 \dots s_n \cap [c, d]|}{N} - \frac{d - c}{b - a} \right| \quad (2.7)$$

Jedes  $x \in s_n$  fällt mit der annähernd gleichen Wahrscheinlichkeit in das Subintervall  $[c, d]$ . Wäre es die selbe Wahrscheinlichkeit hätten wir die Uniformität.

### 2.4.3 Quasi Monte-Carlo Methode

Diese Methode macht sich die Low Discrepancy Sequenzen Folgen zu Nutze im Gegensatz zu der ursprünglichen Monte-Carlo Methode 2.2, welche auf pseudozufälligen Zahlen basiert. Im Gegensatz zu ihr haben wir hier eine schnellere Konvergenz  $O(\frac{1}{N})$ . Wichtig für das weitere Verständnis ist die Randomisierung einer von Grunde her deterministische Low Discrepancy Sequenz. Ein Verfahren, das zufällige Shiften, bildet eine neue Sequenz  $y_i$  aus  $x_i$  durch eine komponenteweise Addition mit einem zufälligen Wert  $w$ .

### 2.4.4 1-Dimension

Diese Arbeit betrachtet 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, und genau das wollen wir, so wählen wir  $\alpha = \Phi$  wobei  $\Phi \approx 1.618033$  den goldenen Schnitt bezeichnet. Wie [Roberts, 2018] gezeigt wird, ist diese Form von Sequenz die beste für das 1-Dimension.

### 2.4.5 2-Dimensionen

Für mehrere Dimensionen, hier zwei, kombiniert man in gängigen Methoden einfach zwei Quasi Monte-Carlo Methode eindimensionale Sequenzen. Für unsere Zwecke untersuchen wir hier die Generalisierung des bereits zuvor beschriebenen goldenen Schnitts Quasi Monte-Carlo Methode, wie hier [Krcadinac, 2006] beschrieben. Die sogenannte Plastische Zahl in ist die Lösung der Gleichung 2-Dimensionen

$$x^3 - x - 1 = 0 \quad (2.9)$$

Die Lösung dieser Gleichung lässt sich über die Padovan und Perrin Sequenz definieren. Damit erhalten wir Plastische Zahl  $\Phi$ :

$$\Phi = \frac{(9 - \sqrt{69})^{1/3} + (9 + \sqrt{69})^{1/3}}{2^{1/3}3^{2/3}} \approx 1.32471795 \quad (2.10)$$

[Padovan, 2002] [Piezas and Weisstein, ] Folgende Gleichung ist auch einfach erweiterbar für höhere Dimensionen.

$$t_n = n\alpha(\text{mod}1), n = 1, 2, 3, .. \alpha = \left(\frac{1}{\Phi_d}, \frac{1}{\Phi_d^2}\right), \quad (2.11)$$

Dabei ist  $\Phi_d$  der goldene Schnitt.  $\Phi_d^2$  ist Lösung der 2-Dimensionen obigen Gleichung.

[Roberts, 2018]

```

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

```

### 2.4.6 Dither Texturen und quasi-zufällige Folgen

## 2.5 Simulated Annealing

### 2.5.0.1 Allgemein

Im Kapitel Retargeting wird eine vorberechnete Textur verwendet. Diese speichert eine Permutation, die unsere blue noise Textur vom frame t in eine blue noise Textur vom frame t+1 umwandelt. Diese Permutation wird dann auf die Startwerte angewandt bevor das nächste frame t+1 gerendert wird. Dadurch werden die blue noise Umverteilungen der Sorting Phase akkumuliert und die optische Aufwertung erst richtig sichtbar. Die retarget Textur wird mit Hilfe von *Simulated Annealing* [Eric Heitz, 2019] berechnet. Angelehnt an metallurgischen Aufheizen und anschließenden Abkühlen wollen wir eine approximativ optimale Lösung finden: Permutiere Pixel der blue noise Textur von frame t bis Sie sehr ähnlich verteilt sind wie die Pixel der blue noise Textur von frame t+1. Dabei ist die Lokalität der Vertauschungen, welche wir bereits in der Sorting Phase verwendet haben, wichtig. Wir übernehmen hier die Lokalitätsbestimmung aus der Arbeit [Eric Heitz, 2019](Radius r = 6). Von einem hohen Energilevel (dither textur zum Zeitpunkt t = 0) wollen wir durch kontrolliertes Abkühlen in ein Niederes gelangen(dither textur zum Teitpunkt t = 1). Zugriff auf die dither Textur berechnet sich für weitere Zeitpunkte wie hier 6 beschrieben.

Die Funktion nach der optimiert wird ist an die Formel aus[Georgiev and Fajardo, 2016] angelehnt.

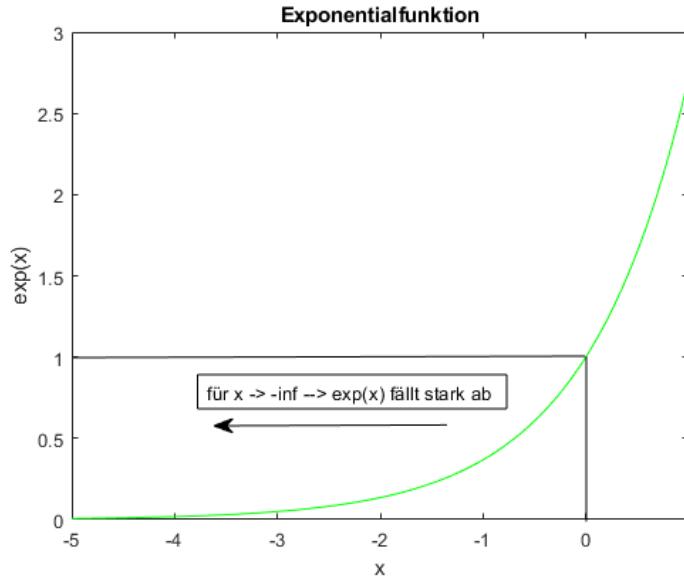
$$E(M) = \sum_{p \neq q} E(p, q) = \sum_{p \neq q} e^{-\frac{\|p_i - q_i\|^2}{\sigma_i^2} - \frac{\|p_s - q_s\|^d/2}{\sigma_s^2}} \quad (2.12)$$

Wähle nach [Ulichney, 1993a]  $\sigma_i = 2.1$  und  $\sigma_s = 1$  Zu den Pixeln p,q beschreibt  $p_i$  und  $q_i$  ihre jeweiligen Koordinaten. Und  $p_s$  und  $q_s$  sind ihre d-dimensionalen Samplewerte.

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

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

(a) Akzeptanzwahrscheinlichkeitsfunktion



(b) exponentialfunktion

Abbildung 2.12: Die Akzeptanzwahrscheinlichkeitsfunktion  $P(\text{Energie}(s), \text{Energie}(s_{\text{new}}), \text{temperature})$

---

**Algorithm 5 Simulated Annealing** finde eine globale Lösung nahe am Maximum

---

```

1: initialisiere Startzustand  $s = s_0$ 
2: initialisiere Starttemperatur  $T_0$ 
3: for  $i=1 \dots \text{maxSteps}$  do
4:   update Temperatur  $t_i$  anhand des Abkühlplanes
5:   //Radius für Nachbarschaftssuche ist auf 6 festgesetzt
6:    $s_{\text{neu}} \leftarrow \text{Nachbarzustand}(s)$ 
7:    $\text{energy}\Delta = \text{energy}(s_{\text{neu}}) - \text{energy}(s)$ 
8:   if  $\text{energy}\Delta < 0$  then
9:      $s = s_{\text{new}}$ 
10:   else
11:     if  $P(\text{Energie}(s), \text{Energie}(s_{\text{new}}), \text{temperature}) \geq \text{random}(0,1)$  then
12:        $s = s_{\text{new}}$ 
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 der Energiezustand des Nachbarn höher als der aktuelle ist. Mit absteigender Temperatur erkennt man in der Abbildung 2.12b 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 Allgemein 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  $\alpha$  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  $\alpha \in [0.8; 0.99]$

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

### 2.5.1.4 Inverse

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

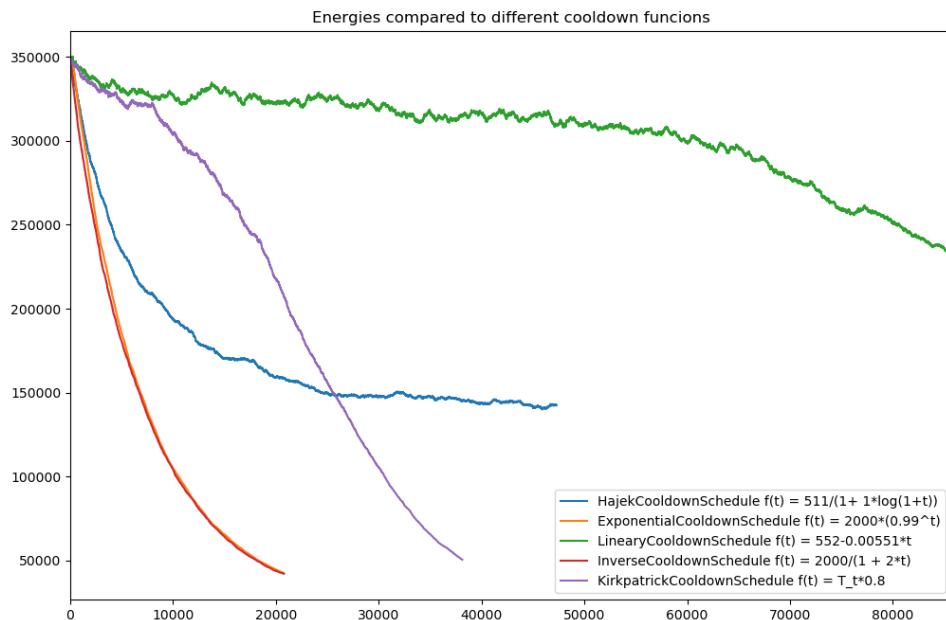


Abbildung 2.13: Vergleich von Abkühlfunktionen mit gesetzten Parametern

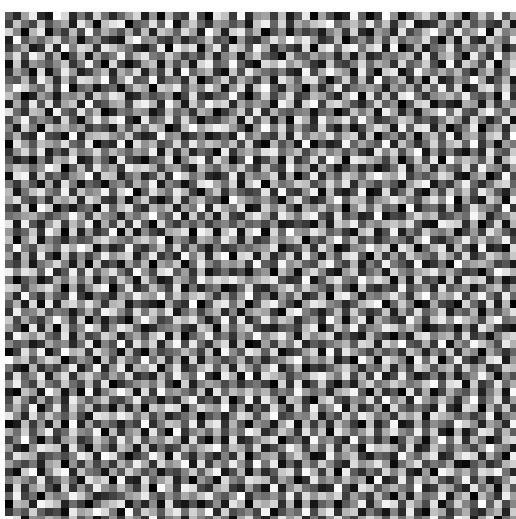


Abbildung 2.14: Blue noise Textur  
64x64

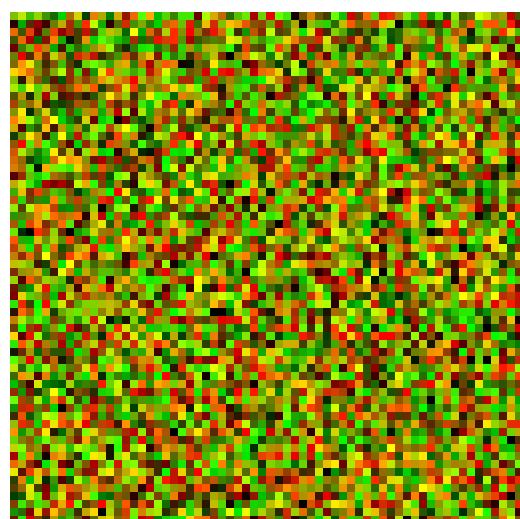


Abbildung 2.15: Permutation; gespeichert in R,G-Channel einer PNG

### 2.5.1.5 Energieverlauf

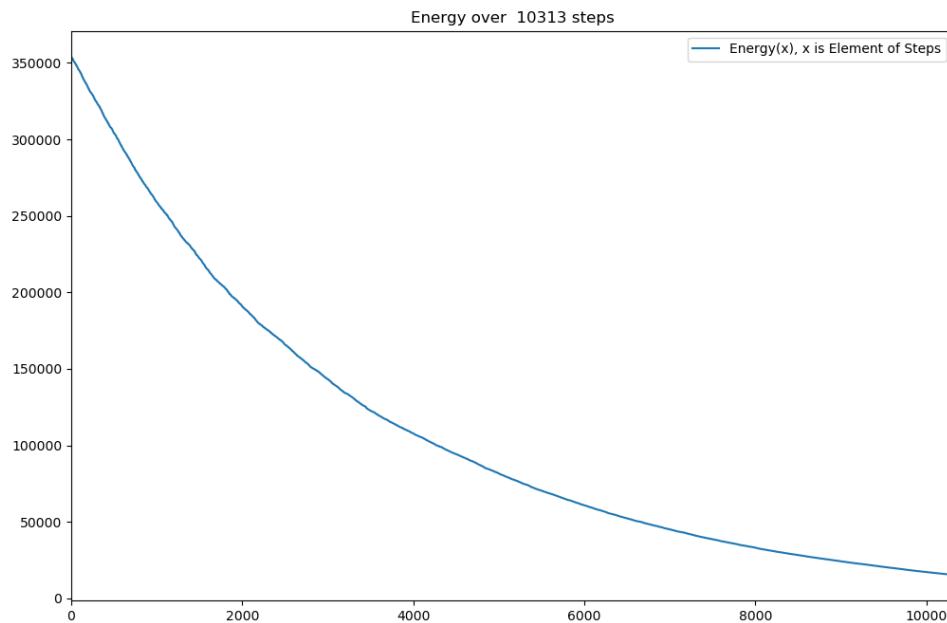


Abbildung 2.16: Energieverlauf beim Simulated Annealing

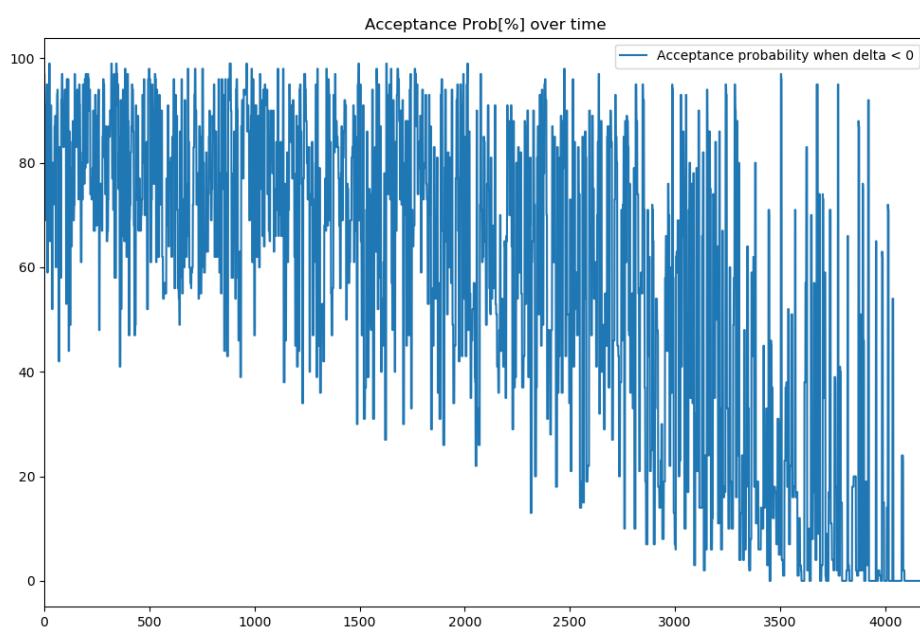


Abbildung 2.17: Akzeptanzfunktionsverlauf bei negativen Energiedeltas

### 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.

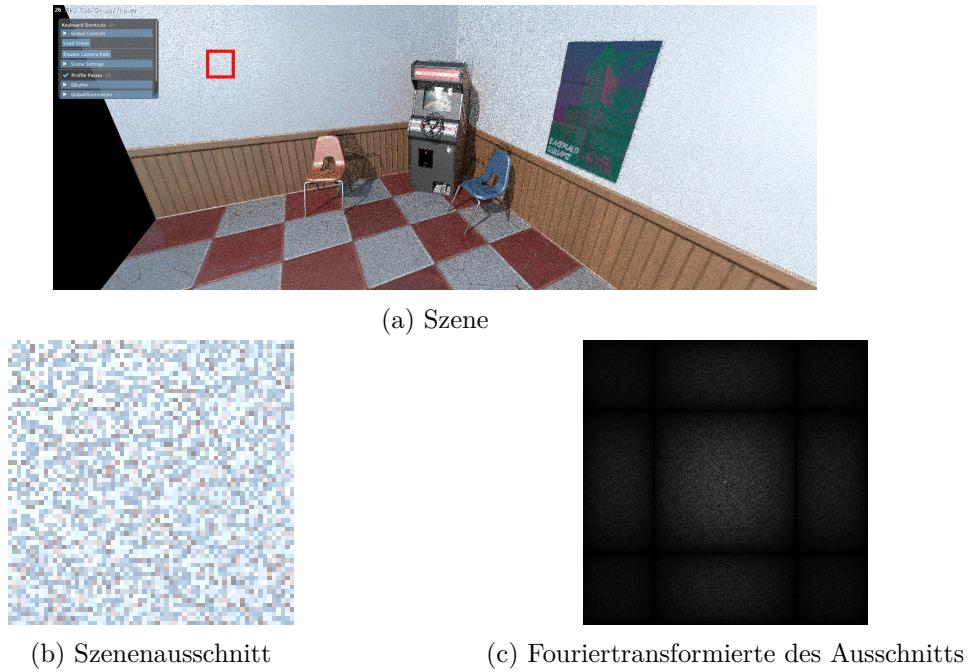


Abbildung 3.1: Weißes Rauschen

### 3.1 Blue Noise Dithering Sampling

Betrachten wir Verfahren wie das Low Discrepancy Sequenzen Verfahren, 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 Fehlverteilungen im Bildraum schaffen hiernach einen besseren optischen Eindruck für das menschliche Auge. Die Arbeit [Georgiev and Fajardo, 2016] präsentiert zum konventionellen zufälligen Shiften 2.4.2 eine Alternative, die den gewählten Offset anhand einer über das Fenster gekachelten Blue Noise textur wählt. Im Folgenden wird auf das BNDS (blue noise dithered sampling) und deren *a priori* Eigenschaften eingegangen um anschließend die A Posteriori Eigenschaften des Temporalen Algorithmus besser zu verstehen.

Die Optimalität der 1-Dimensionalität der Sample Anzahl geht aus den Ausführungen in [Eric Heitz, 2019, S.3] hervor und hat mit der abnehmenden Bildraumkorrelation der Samples mit gleichzeitig steigender Anzahl zu tun.

Die hohe Dimensionalität der verwendeten Blue Noise Textur bleibt ein offenes Problem [Peters, 2016]. Deshalb ist eine niedrige Anzahl von Dimensionalität günstig. für das BNDS günstig 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 unserem Path Tracer basierend auf [Benty et al., 2018], so ergibt es sich zur Anschaufung wie folgt:



(a) Szenenausschnitt

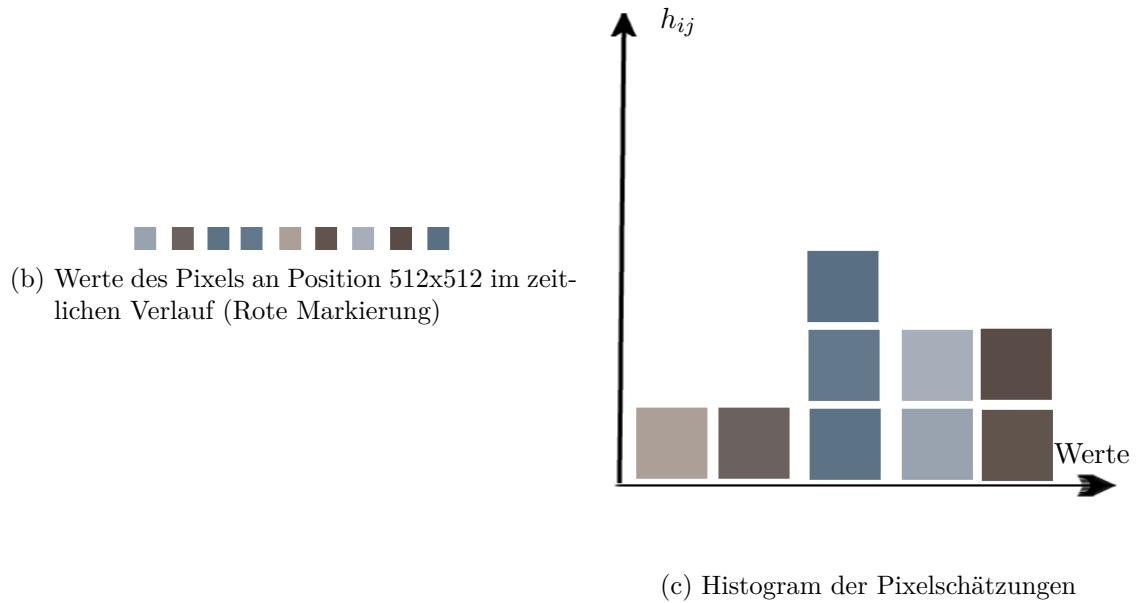


Abbildung 3.2: Pixelwerte an Position 512x512(grüne Markierung) in aufeinanderfolgenden Zeitschritten

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 unser 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:  $retarget_t(i,j) = retarget_0(i + \alpha t, j + \beta t) + (\alpha t, \beta t)$
- 

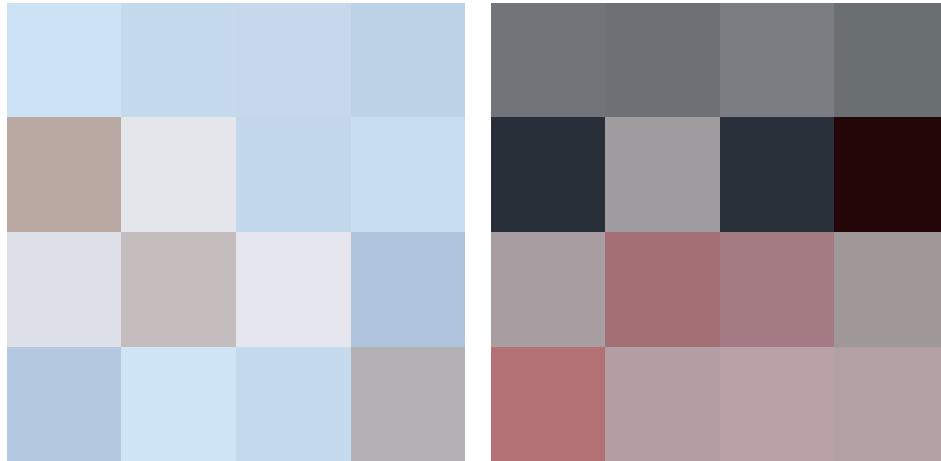


Abbildung 3.3: Pixelblöcke bei (in-)homogenen Flächen

## 3.3 Sorting

In diesem Schritt wollen wir nun die Untersuchungen aus A Posteriori durchführen. Nach dem Rendern eines  $Frames_t$  (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: blueNoise_t = 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.

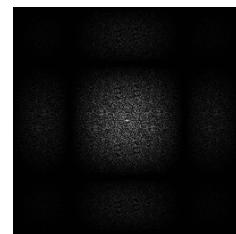
Die APosteriori Erkenntnisse zu der inversen Funktion 3.2 garantieren uns nach dem Um-sortieren die entsprechenden Seeds in einer ebenfalls blue noise verteilten Struktur zu erhalten.



(a) Szene



(b) Szenenausschnitt



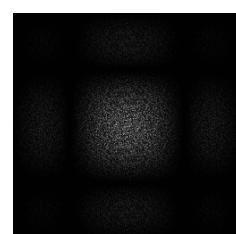
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.4: Zeitpunkt  $t=1$ 

(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

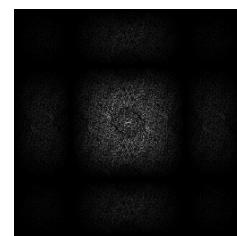
Abbildung 3.5: Zeitpunkt  $t=2$



(a) Szene



(b) Szenenausschnitt



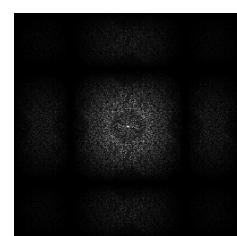
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.6: Zeitpunkt  $t=3$ 

(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

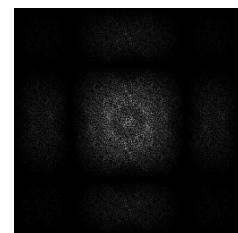
Abbildung 3.7: Zeitpunkt  $t=4$



(a) Szene



(b) Szenenausschnitt

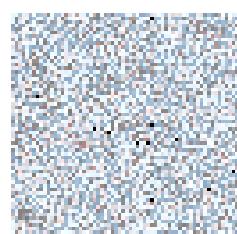


(c) Fouriertransformierte des Ausschnitts

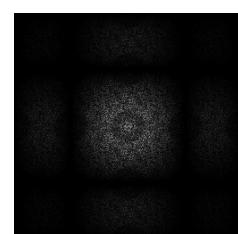
Abbildung 3.8: Zeitpunkt t=5



(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

Abbildung 3.9: Zeitpunkt t=5

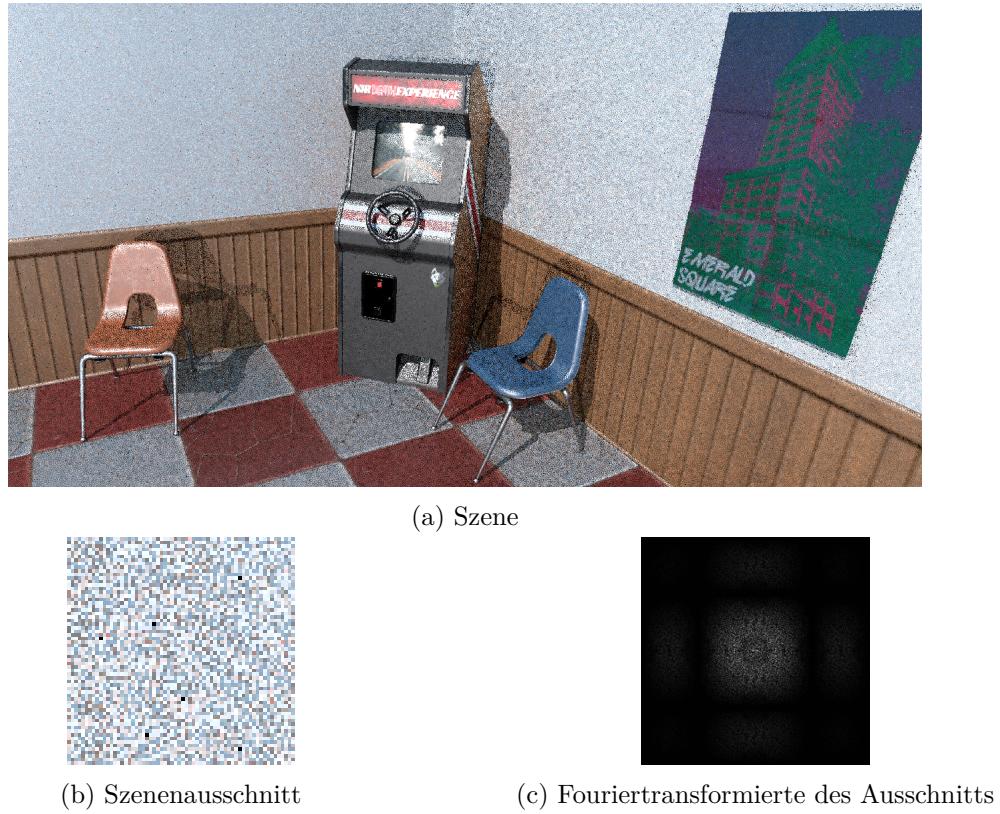


Abbildung 3.10: Zeitpunkt t=5

### 3.4 Retargeting

[Eric Heitz, 2019]

Zu Grunde liegender Sinn dieses Schrittes: Vertauschen der Anfangswerte, die verteilt sind wie  $BlueNoise_t$ , sodass Sie verteilt sind wie die  $BlueNoise_{t+1}$ . Aufgrund dessen haben wir eine Aufsummierung der blue noise Fehlerverteilungen über viele Frames.

---

#### Algorithm 8 Retargeting Schritt t Vor Rendern Frame t+1 nach Sortier Schritt

---

```

1: //permutation indices from precomputed texture
2: retagettex = retagettex[calcCorrectOffset(incomingbluenoisetexture)];
3: List<PixelPermutation> L = retagettex
4: for i = 1 .. numberofPixelsPerBlock do
5:   retargetedSeeds(L.getNewIndices()) = incomingSeeds(L.getOldIndices());
6: end for

```

---

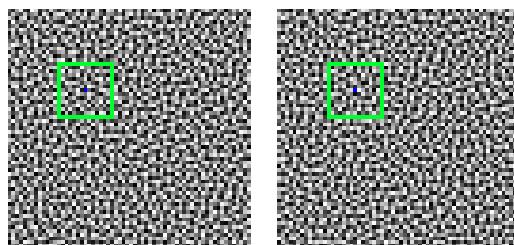


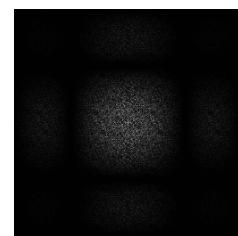
Abbildung 3.11: Permutation



(a) Szene



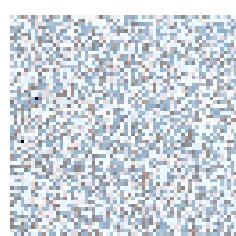
(b) Szenenausschnitt



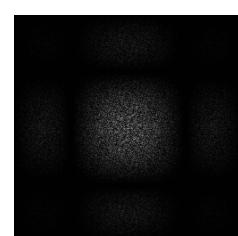
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.12: Zeitpunkt  $t=1$ 

(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

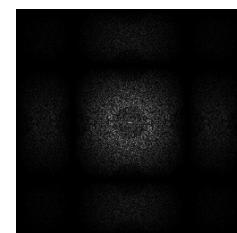
Abbildung 3.13: Zeitpunkt  $t=2$



(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

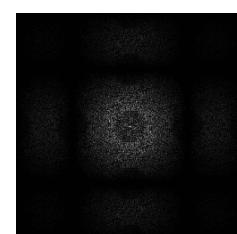
Abbildung 3.14: Zeitpunkt t=3



(a) Szene



(b) Szenenausschnitt

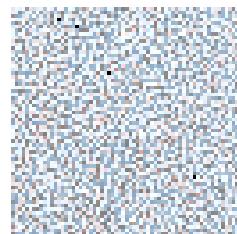


(c) Fouriertransformierte des Ausschnitts

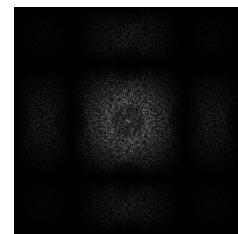
Abbildung 3.15: Zeitpunkt t=4



(a) Szene



(b) Szenenausschnitt



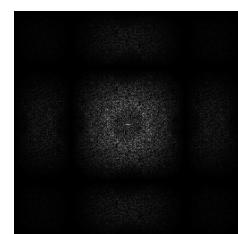
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.16: Zeitpunkt  $t=5$ 

(a) Szene

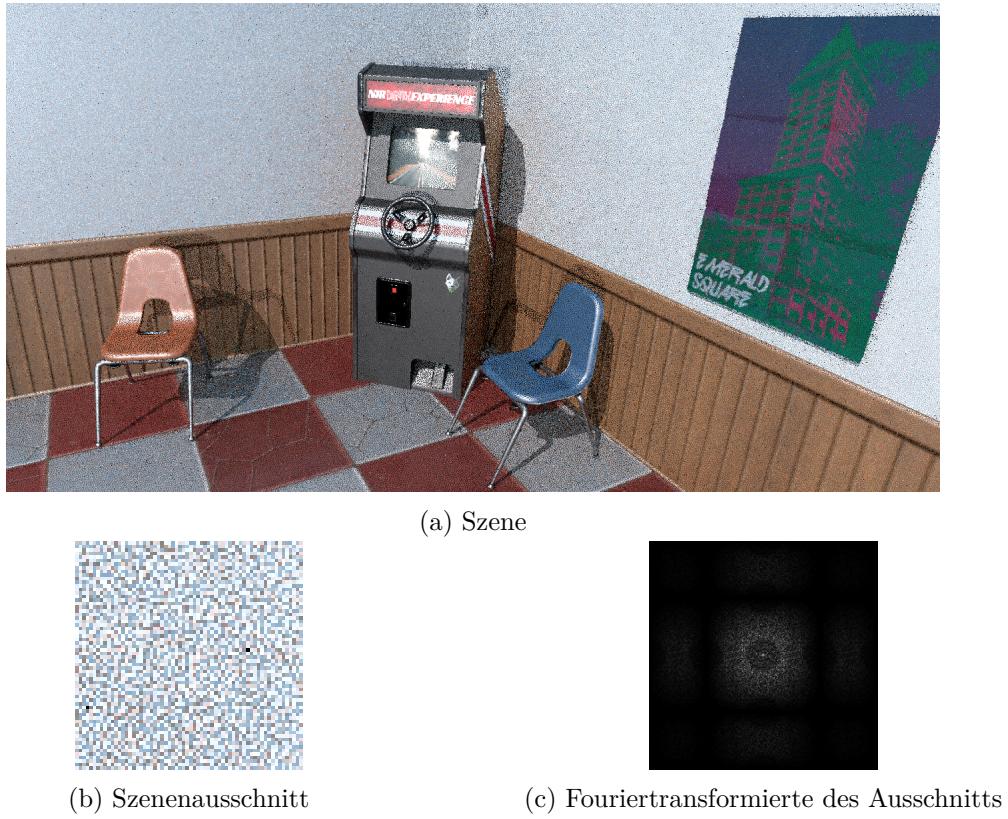


(b) Szenenausschnitt



(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.
- [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.
- [Heitz et al., 2019] Heitz, E., Belcour, L., Ostromoukhov, V., Coeurjolly, D., and Iehl, J.-C. (2019). A Low-Discrepancy Sampler that Distributes Monte Carlo Errors as a Blue Noise in Screen Space. In *SIGGRAPH’19 Talks*, Los Angeles, United States. ACM.
- [Kiencke and Jäkel, 2009] Kiencke, U. and Jäkel, H. (2009). *Signale und Systeme*. Oldenbourg Verlag.

- [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.
- [Owen, 1998] Owen, A. B. (1998). Scrambling sobol’and niederreiter–xing points. *Journal of complexity*, 14(4):466–489.
- [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.
- [Piezas and Weisstein, ] Piezas, Tito III; van Lamoen, F. and Weisstein, E. W. Plastic constant.
- [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.
- [Wronski, 2016] Wronski, B. (2016). Dithering part 1-5. blogpost.



# **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 13. Februar 2020

(Jonas Heinle)