

# 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

16. Januar 2020

# 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 Path Tracer . . . . .	3
2.1.0.1 Funktionsweise . . . . .	3
2.1.0.2 Monte-Carlo-Integration . . . . .	4
2.1.0.3 DirectX Raytracing . . . . .	4
2.2 Blue Noise . . . . .	7
2.2.1 Eigenschaften . . . . .	7
2.2.1.1 Uniformität . . . . .	7
2.2.1.2 Isotropie . . . . .	8
2.2.1.3 Niedrige Frequenzen . . . . .	8
2.2.1.4 Kachelung . . . . .	9
2.3 Quasi-Zufallsfolgen . . . . .	11
2.3.1 Einleitung . . . . .	11
2.3.2 1-Dimension . . . . .	11
2.3.3 2-Dimensionen . . . . .	11
2.3.4 Dither Texturen und quasi-zufällige Folgen . . . . .	12
2.4 Simulated Annealing . . . . .	12
<b>3 Temporaler Algorithmus</b>	<b>14</b>
3.1 A Posteriori . . . . .	15
3.1.1 Theoretische Grundlage . . . . .	15
3.1.2 Praktische Durchführung . . . . .	16
3.2 Sorting . . . . .	16
3.3 Retargeting . . . . .	17
<b>Literaturverzeichnis</b>	<b>21</b>

# 1. Prelude

## 1.1 Abstract

Die Strahlverfolgung und dazugehörige Techniken gewinnen gegenwärtig in der Echtzeitcomputergrafik an Bedeutung. Dabei haben bereits frühere Arbeiten die blue noise Fehlerverteilungen miteinbezogen und deren Bedeutung in der Steigerung der wahrnehmbaren Bildqualität hervorgehoben und verdeutlicht. Diese Arbeit wird diesen Stand aufnehmen und einen zeitlich stabilen Algorithmus erläutern. Ein Algorithmus, der mit Anzahl der Samples und Dimension des Tracers einhergeht. 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

Dithering ist in Echtzeitanwendungen bereits verbreitet. So zu sehen in dem Computerspiel *Call of Duty: Advanced Warfare*, wobei bereits im Post-Processing [Jimenez, 2014] mit white noise und bayer pattern gearbeitet wird.

Weitere Entwicklungen [Georgiev and Fajardo, 2016] haben sich mit blue noise dither masks Blue Noise beschäftigt und ihre Nützlichkeit in Steigung der visuellen Qualität gezeigt. Hierbei lassen sich im Bildraum die entstehenden Monte-Carlo-Integrationsfehler Path Tracer zwar nicht verringern, jedoch umverteilen.

Im Besonderen zeigte [Schied, 2019] die Bedeutung von blue noise Fehlerverteilung im Bildraum für Echtzeitanwendungen mit neuer Raytracingtechnologie. Hier werden bereits zum Sampling blue noise dither masks benutzt, über den Bildschirm "gekachelt" und über frames durchgewechselt. Das vor dem temporalen Algorithmus erschienene Paper [Heitz et al., 2019] zeigt eindrucksvoll die Mächtigkeit von Fehlerumverteilungen im Bildraum. Die Entwickler von INSIDE [Ulichney, 1993a] fanden auch bereits eine Anwendung für blue noise Fehlerverteilungen.

## 2. Grundlagen

### 2.1 Path Tracer

#### 2.1.0.1 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 in [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.

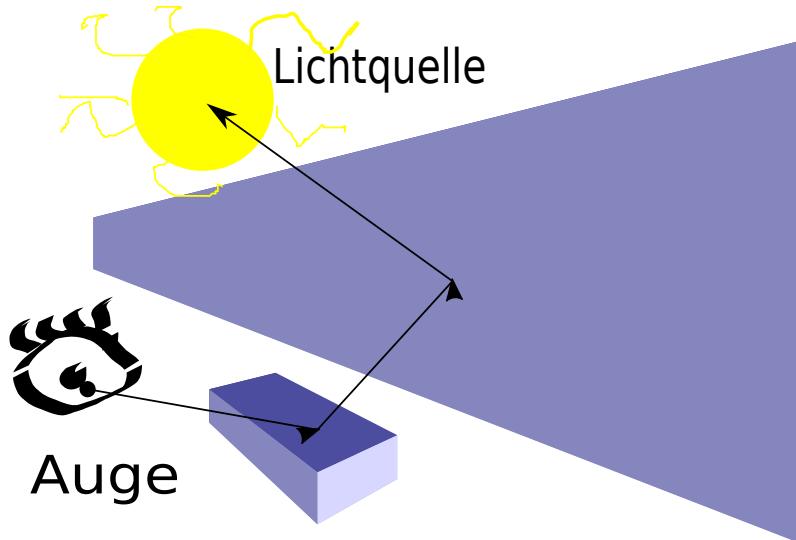


Abbildung 2.1: Grundkonzept path tracer

### 2.1.0.2 Monte-Carlo-Integration

Mit der Monte Carlo Integration approximieren wir die Rendergleichung.

Bei gegebener Dimensionalität  $n$  des Renderintegrals und der Wahrscheinlichkeitsdichte-funktion  $\rho(x_i)$  [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.

### 2.1.0.3 DirectX Raytracing

Für besseres Verständniss möchte ich hier eine Einführung in das Path Tracing[Bentley et al., 2018] mit Hilfe der neuen DirectX-Schnittstelle geben.

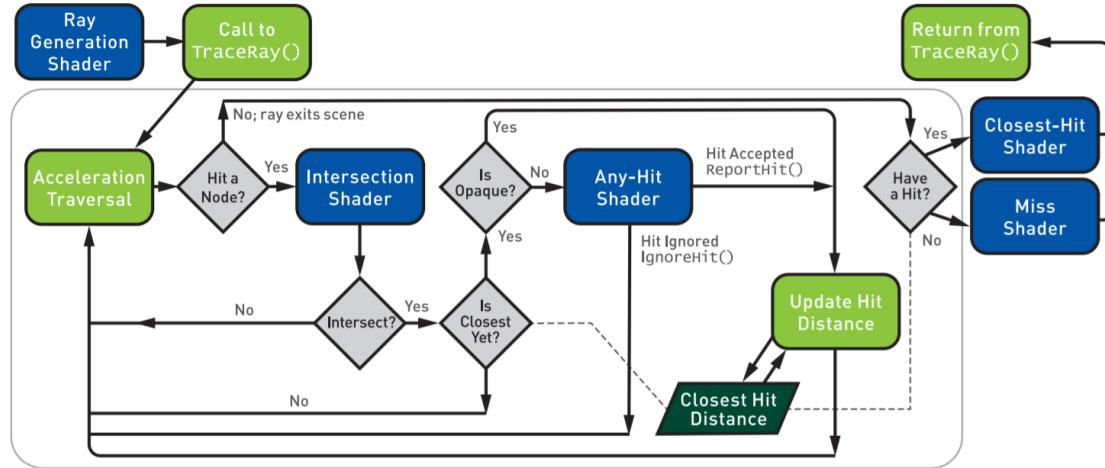


Abbildung 2.2: DirectX Raytracing Pipeline aus [Haines and Akenine-Möller, 2019]

In der obigen Übersicht lässt sich der Beginn (Startpunkt eines Strahles) der neuen Pipeline erkennen, der **Ray Generation shader**.

---

#### Algorithm 1 Wie man Strahlen verschießt

---

```

1: [shader("raygeneration")]
2: launchIndex = DispatchRaysIndex().xy;
3: for (int i = 0; i < numberOfrays;i++) do
4:   color = computedirectLighting(launchIndex, i) + computeindirectLighting(launchIndex, i);
5: end for
6: output[id] = color;
  
```

---

Mit Hilfe der Methode **TraceRay()** werden dann innerhalb der Methoden 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<sub>t</sub>* 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** führt den konkreten, nächsten Schnitt für jeden Strahl durch. 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 upzudaten 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 Was macht ein Path Tracer nochmal vereinfacht in Pseudocode dargestellt, wobei die entsprechenden shader im jeweiligen Codeabschnitt markiert sind.

---

**Algorithm 4** Was macht ein Path Tracer

---

```

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

---

## 2.2 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.

### 2.2.1 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, 1993b] vorgestellten Algorithmus erstellt wurden. Die korrespondierenden Spektren wurden mit Hilfe von [JCr, 2018] erstellt.

#### 2.2.1.1 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,

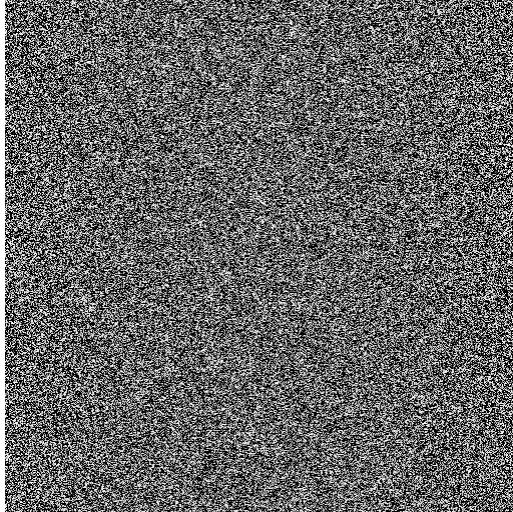


Abbildung 2.3:  $512^2$  white noise Textur

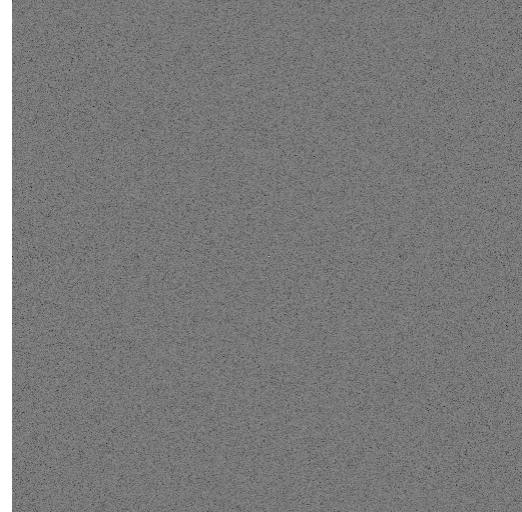


Abbildung 2.4: Amplitudenspektrum  
 $512^2$  white noise Textur

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.

### 2.2.1.2 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.

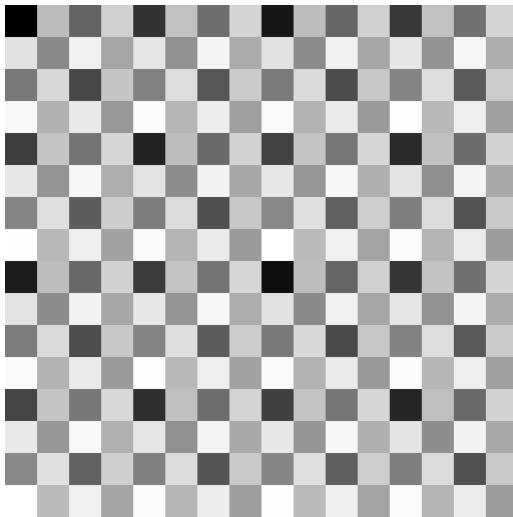


Abbildung 2.5:  $512^2$  bayer pattern Textur

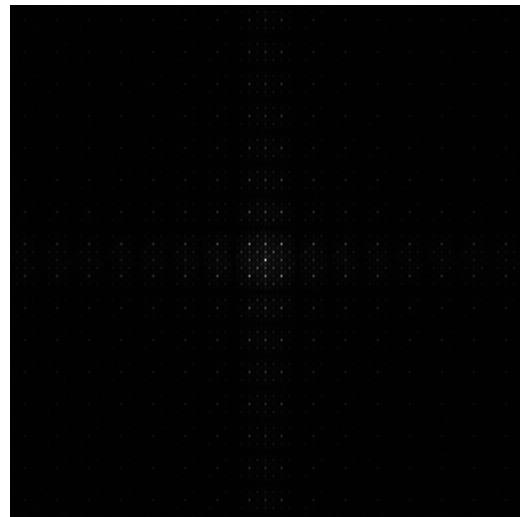
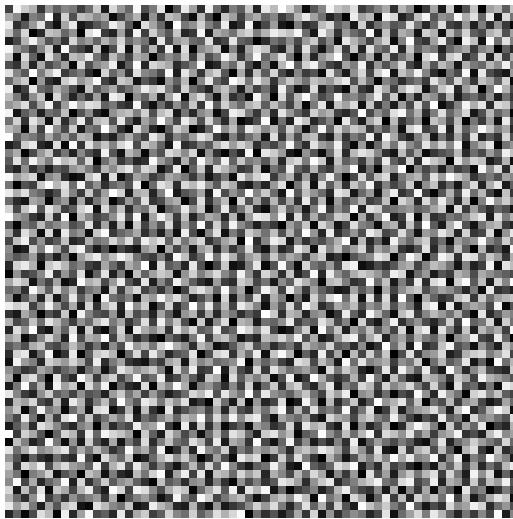
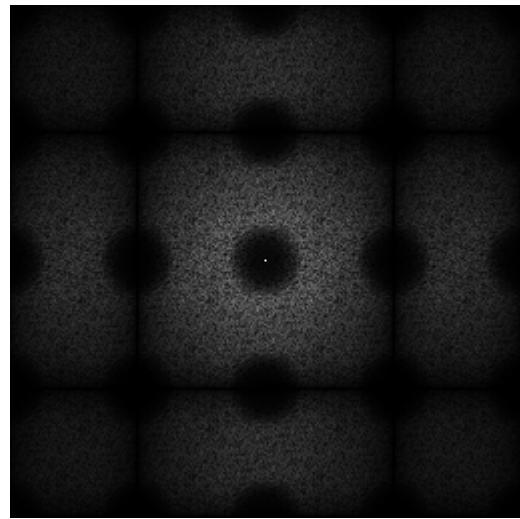


Abbildung 2.6: Amplitudendichte  $512^2$  bayer pattern Textur

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. Bieten allerdings eine sehr effiziente Verwendung, da sehr leistungssparende GPU Befehle.

### 2.2.1.3 Niedrige Frequenzen

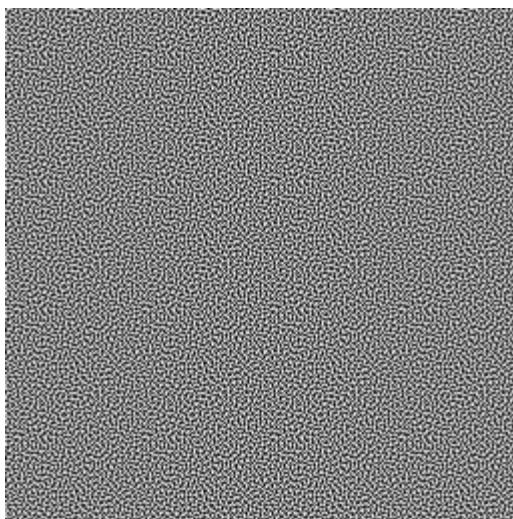
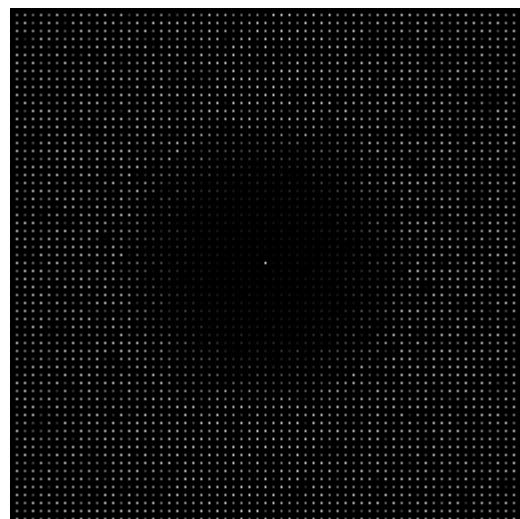
Niedrige Frequenzen sind in einer blue noise sehr wenig bis gar nicht vertreten. Dies ist an dem schwarzen Ring innerhalb der Amplitudendichte zu erkennen. Oder in der Zeitdomäne, an dem Abhandensein von gleichfarbigen Pixelbündeln. Genau dies wollten wir erreichen. Außerdem haben wir aus den vorherigen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

Abbildung 2.7:  $512^2$  blue noise TexturAbbildung 2.8: 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.

#### 2.2.1.4 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 2.9:  $512^2$  gekachelte Textur von  $64 \times 64$ Abbildung 2.10: Fourier Spektrum  $512^2$  4-fach getiled  $64 \times 64$  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.

## 2.3 Quasi-Zufallsfolgen

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

### 2.3.1 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.

### 2.3.2 1-Dimension

Diese Arbeit betrachtet Rekurrenz Sequenzen, basierend auf irrationalem Bruchrechnen der Form

$$R_1(\alpha) : t_n = s_0 + n\alpha(\text{mod}1); n = 1, 2, 3, \dots \quad (2.7)$$

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.

### 2.3.3 2-Dimensionen

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

$$x^3 - x - 1 = 0 \quad (2.8)$$

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

$$\Phi = \frac{(9 - \sqrt{69})^{1/3} + (9 + \sqrt{69})^{1/3}}{2^{1/3}3^{2/3}} \approx 1.32471795 \quad (2.9)$$

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

$$t_n = n\alpha(\text{mod}1), n = 1, 2, 3, .. \alpha = \left( \frac{1}{\Phi_d}, \frac{1}{\Phi_d^2} \right), \quad (2.10)$$

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

```

### 2.3.4 Dither Texturen und quasi-zufällige Folgen

## 2.4 Simulated Annealing

Im vorherigen Kapitel, dem Retargeting Schritt Retargeting, wird eine vorberechnete Retargeting-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 Umverteilung der Sorting Phase Sorting akkumuliert und die optische Aufwertung erst richtig sichtbar. Die retarget Textur wird mit Hilfe von **simulated annealing** [Eric Heitz, 2019] berechnet. Wir wollen somit 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 PhaseSorting verwendet haben, wichtig.

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

$$E(M) = \sum_{p \neq q} E(p, q) = \sum_{p \neq q} e^{-\frac{\|p_i - q_i\|^2}{\sigma_i^2} - \frac{\|p_s - q_s\|^{d/2}}{\sigma_s^2}} \quad (2.11)$$

Wähle nach [Ulichney, 1993b]  $\sigma_i = 2.1$  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.

---

#### Algorithm 5 Simulated Annealing finde sehr gute Lösung

---

```

1: initialisiere Startzustand  $s = s_0$ 
2: for i=1...maxSteps do
3:   //Radius für Nachbarschaftssuche ist auf 6 festgesetzt
4:    $s_{neu} \leftarrow$ Nachbarzustand(s)
5:   if P(Energie(s), Energie( $s_{new}$ )) $\geq$  random(0,1) then
6:      $s = s_{new}$ 
7:   end if
8: end for
9: return Endzustand s;
```

---

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  $P(\text{Energie}(s), \text{Energie}(s_{new}))$  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 ESimulated Annealing betrachtet dabei zwei

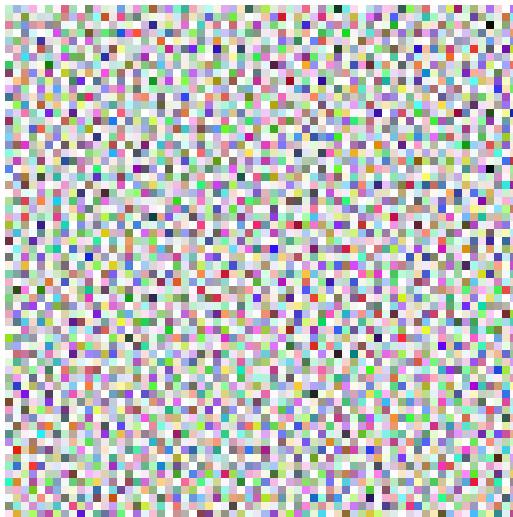


Abbildung 2.11: Blue noise Textur  
64x64

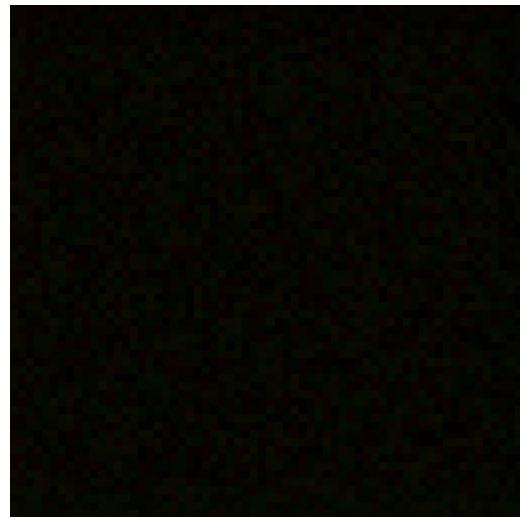


Abbildung 2.12: Permutation; gespeichert in R,G-Channel einer .png

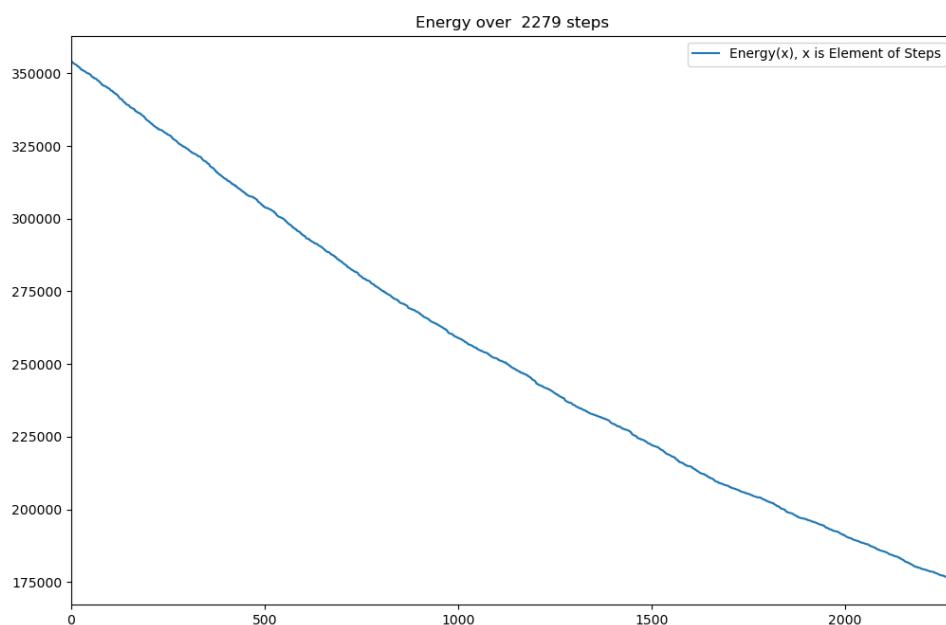


Abbildung 2.13: Energieverlauf beim Simulated Annealing

### 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.

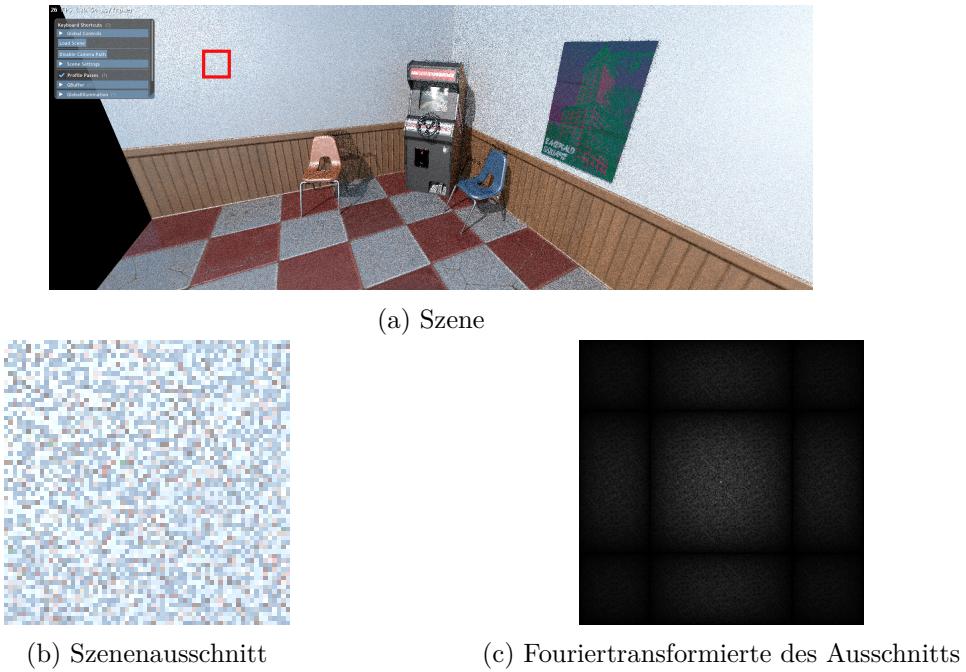


Abbildung 3.1: Weißes Rauschen

## 3.1 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 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.

### 3.1.1 Theoretische Grundlage

Im Kapitel über des 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 der Gleichung zur Abbildung 3.1 ist die Wahrscheinlichkeitsdichtefunktion  $h_{ij}$  aufgetragen, als eine Funktion über alle möglichen Werte  $I_{ij}$  eines Pixels  $(i,j)$ .

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

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 Abbildung 3.2 als im Bildraum blue noise verteilte Zahlen. Daraus folgt, dass die resultierenden Integrationsfehler auch als blue noise im Bildraum verteilt sind.

### 3.1.2 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:  $retarget_t(i,j) = retarget_0(i + \alpha t, j + \beta t) + (\alpha t, \beta t)$ 
```

---

## 3.2 Sorting

In diesem Schritt wollen wir nun die Untersuchungen aus A Posteriori durchführen. Nach dem Rendern eines  $Frame_{t+1}$  (vor dem Rendern von  $Frame_t$ ) approximieren wir das Histogramm der Pixelwerte anhand der Pixelwerte von  $Frame_t$ . Dabei betrachten wir Anzahl Pixel pro BLOCK 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(blueNoise_t));
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.

Wir wechseln in jedem Frame die verwendeten blue noise Texturen um gewissen Artefakten zu entgehen und andere temporale Algorithmen zu ermöglichen. Dieser Schritt alleine reicht also nicht für den erwünschten Effekt.

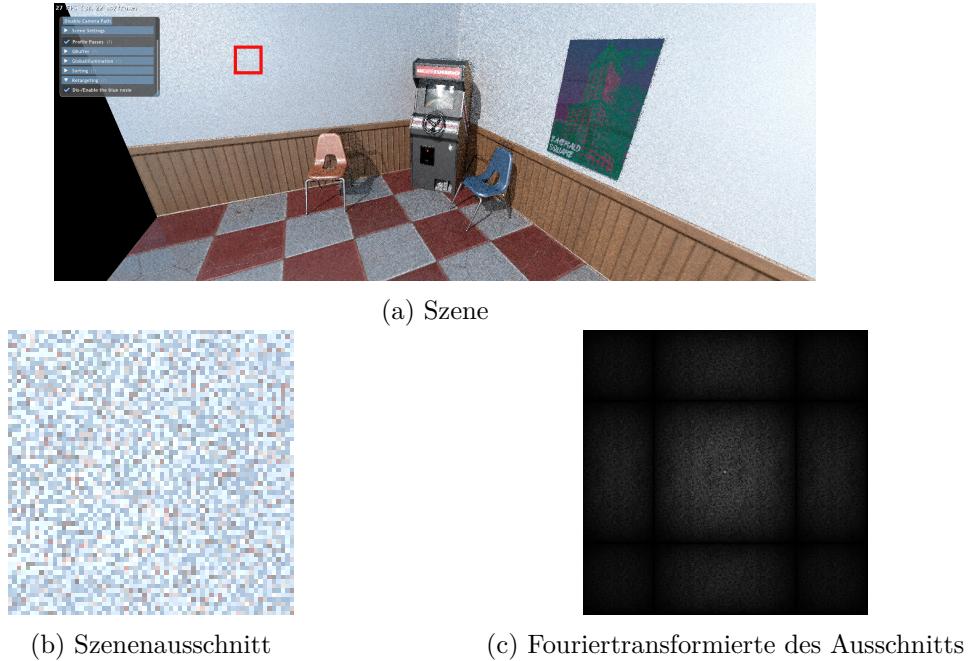


Abbildung 3.2: Path Tracer nur Sorting

### 3.3 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: retagett = retargettexture[calcCorrectOffset(incomingbluenoisetexture)];
3: List<PixelPermutation> L = retagett
4: for i = 1 .. numberOfPixelsPerBlock do
5:     retargetedSeeds(L.getNewIndices()) = incomingSeeds(L.getOldIndices());
6: end for

```

---

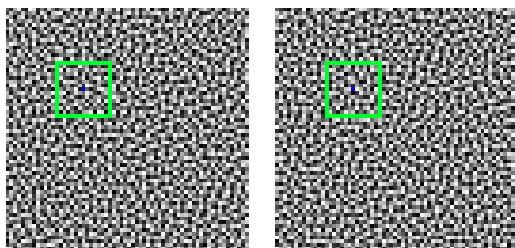
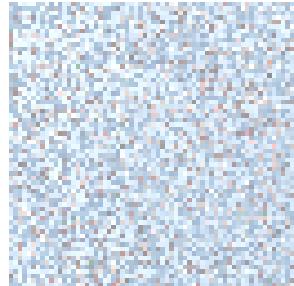


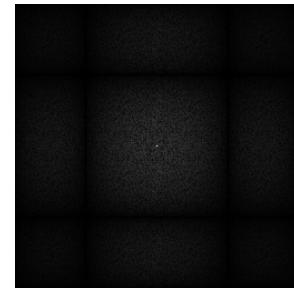
Abbildung 3.3: Permutation



(a) Szene



(b) Szenenausschnitt

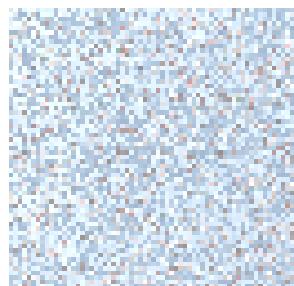


(c) Fouriertransformierte des Ausschnitts

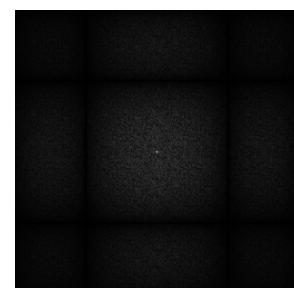
Abbildung 3.4: Zeitpunkt t=1



(a) Szene

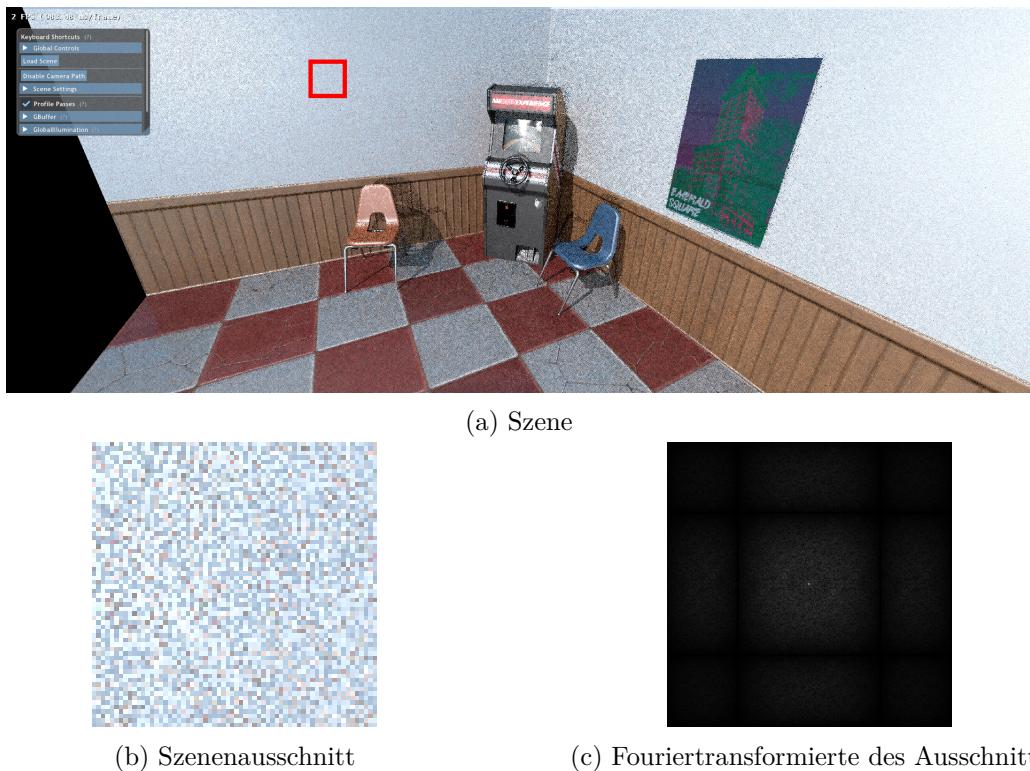
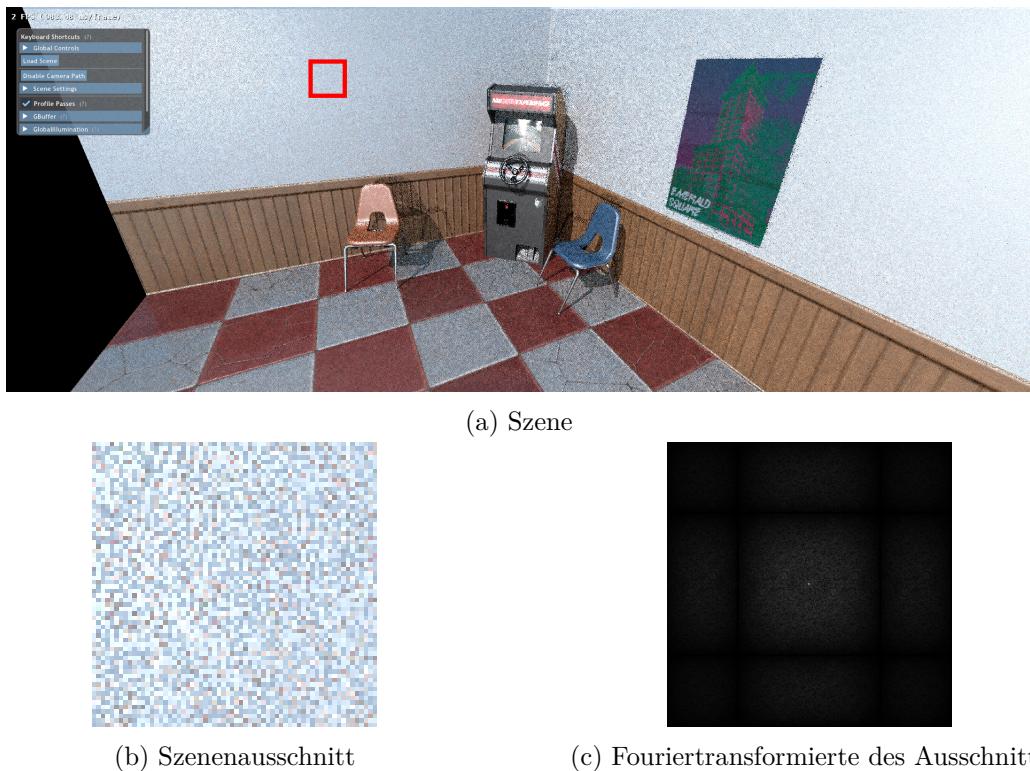


(b) Szenenausschnitt



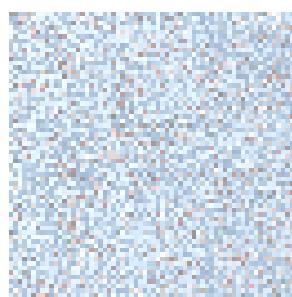
(c) Fouriertransformierte des Ausschnitts

Abbildung 3.5: Zeitpunkt t=2

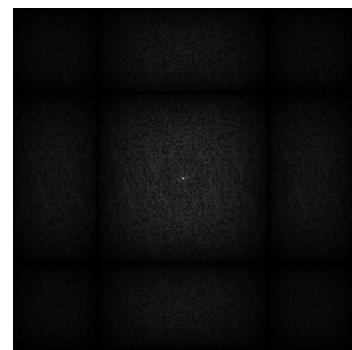
Abbildung 3.6: Zeitpunkt  $t=3$ Abbildung 3.7: Zeitpunkt  $t=4$



(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

Abbildung 3.8: Zeitpunkt t=5

)

# Literaturverzeichnis

- [JCr, 2018] (2018). jcrystal. performing fft on images.
- [Ray, 2019] (2019). Rayflag enumeration. control behaviour of a traced ray.
- [Whi, 2019] (2019). Whitenoisegenerato. <https://www.cssmatic.com/noise-texture>. Accessed: 24.11.2019.
- [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>.
- [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.
- [Jimenez, 2014] Jimenez, J. (2014). A new generalization of the golden ratio. THE FIBONACCI ASSOCIATION.
- [Kiencke and Jäkel, 2009] Kiencke, U. and Jäkel, H. (2009). *Signale und Systeme*. Oldenbourg Verlag.
- [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 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.
- [Ulichney, 1993b] Ulichney, R. A. (1993b). 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.
- [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 16. Januar 2020

(Jonas Heinle)