

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

6. Dezember 2019

Inhaltsverzeichnis

1 Prelude	1
1.1 Abstract	1
1.2 Einleitung	2
2 Grundlagen	3
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	5
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	11
3 Temporaler Algorithmus	12
3.1 A Posteriori	12
3.1.1 Theoretische Grundlage	12
3.1.2 Praktische Durchführung	13
3.2 Sorting	13
3.3 Retargeting	14
3.4 Simulated Annealing	14
Literaturverzeichnis	16

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 [Jim14] mit white noise und bayer pattern gearbeitet wird.

Weitere Entwicklungen [GF16] haben sich mit blue noise dither masks 2.2 beschäftigt und ihre Nützlichkeit in Steigung der visuellen Qualität gezeigt. Hierbei lassen sich im Bildraum die entstehenden Monte-Carlo-Integrationsfehler 2.1 zwar nicht verringern, jedoch umverteilen.

Im Besonderen zeigte [Sch19] die Bedeutung von blue noise Fehlerverteilung im Bildraum für Echtzeitanwendungen mit neuer Raytracingtechnologie. Das vor dem temporalen Algorithmus erschienene Paper [HBO⁺19] zeigt eindrucksvoll die Mächtigkeit von Fehlerumverteilungen im Bildraum. Die Entwickler von INSIDE [Uli93a] 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 [BYF⁺18] verwendet eine klassische Umsetzung.

Der Path Tracer beruht auf Erkenntnisse der Lösung der allgemeinen Rendergleichung 2.1

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

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

Die Schlussfolgerung aus dieser Gleichung 2.1 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 2.1 lässt sich die vollständige Transportgleichung 2.2 des Path Tracer beschreiben. Wie in [MS09] beschrieben wird ausgehend von der vollständigen Transportgleichung 2.2

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

der vollständige Lichttransport beschrieben. Man kann deutlich die Ähnlichkeit zu 2.1 erkennen. Wir haben den Emissionsterm, die relative Lage der Punkte zueinander und die bidirektionale Verteilungsfunktion welche den Energietransport beeinflussen.

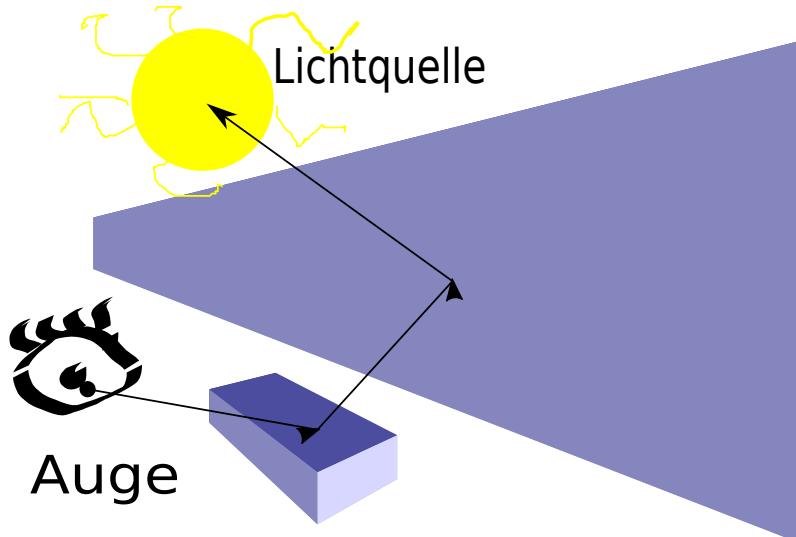


Abbildung 2.1: Grundkonzept path tracer

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 $\rho(x_i)$ [DS02]

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

Dabei wird das n -dimensionale Integral 2.1 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. [Caf98] 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 2.3, 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 2.4 ist ein quadratischer Fehler.



Abbildung 2.2: Szene mit Weißen Rauschen

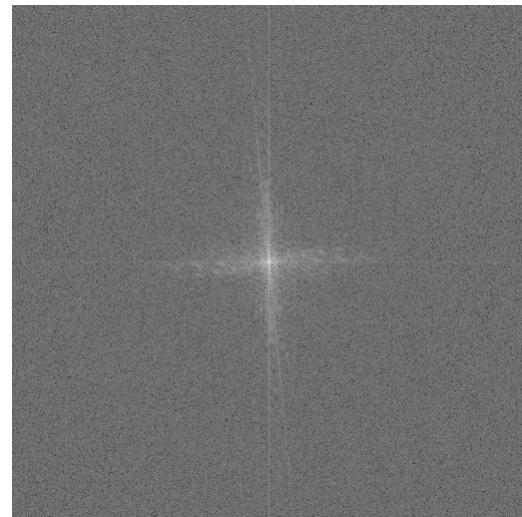


Abbildung 2.3: FFT des Ausschnitts

2.1.0.3 DirectX Raytracing

Für besseres Verständnis möchte ich hier eine Einführung in das Path Tracing[BYF⁺18] mit Hilfe der neuen DirectX-Schnittstelle geben.

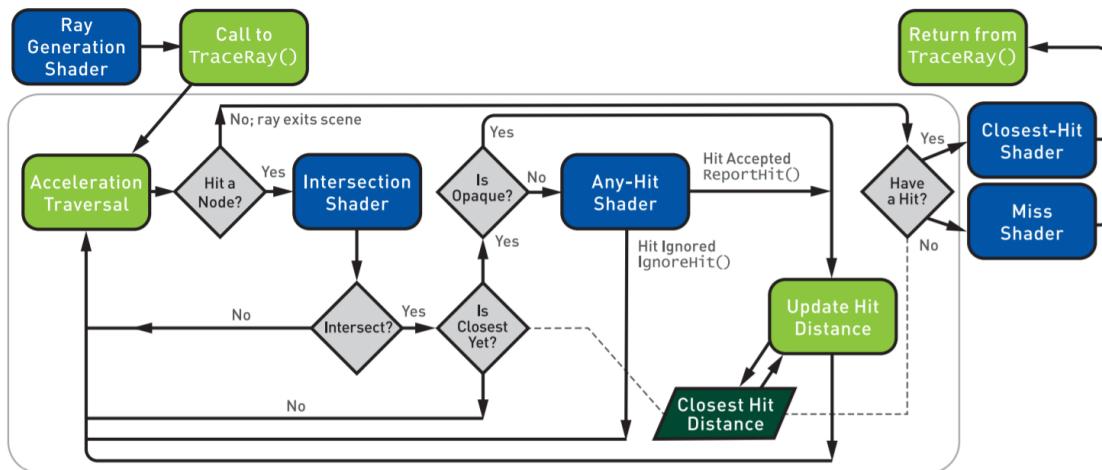


Abbildung 2.4: DirectX Raytracing Pipeline aus [HAM19]

In der obigen Übersicht lässt sich der Start der neuen Pipeline erkennen, der **Ray Generation shader**. Von hier aus werden Rays gestartet **TraceRay()**. **Intersection shader** führt die Schnittpunktberechnungen durch. **Any-hit shaders** erlauben klassische "Discards" oder informieren über einen korrekten Schnitt. Der **Closest-hit shader** führt den konkreten, nähsten Schnitt für jeden Strahl durch. 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 1 nochmal vereinfacht in Pseudocode dargestellt, wobei die entsprechenden shader im jeweiligen Codeabschnitt markiert sind.

Algorithm 1 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

[Uli88] 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 [Gam17] 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 [Uli93b] vorgestellten Algorithmus erstellt wurden. Die korrespondierenden Spektren wurden mit Hilfe von [JCr18] 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 ausgeben 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] erstellten white noise Textur,

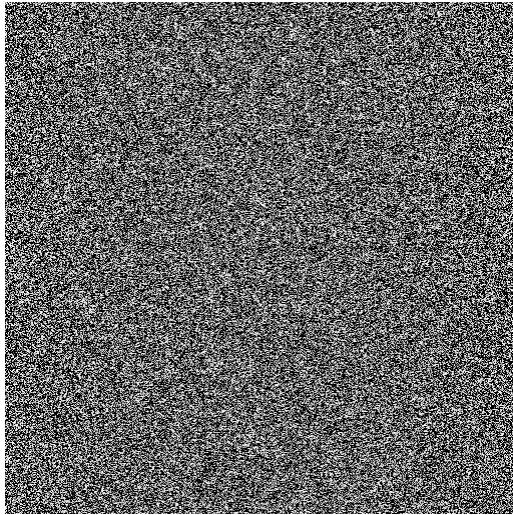


Abbildung 2.5: 512^2 white noise Textur

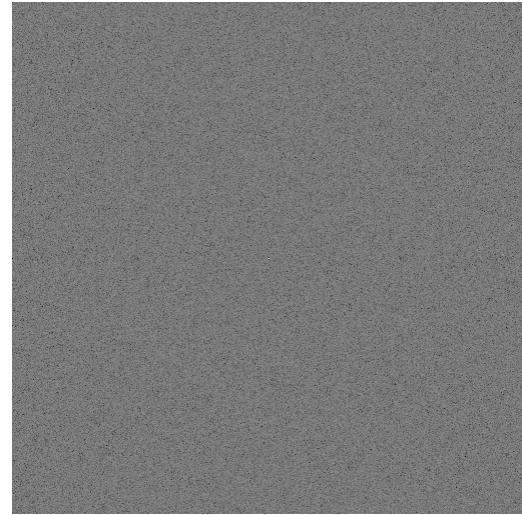


Abbildung 2.6: Amplitudenspektrum
 512^2 white noise Textur

ergibt sich eine typische Amplitudendichte. Zufällig verteilt, über alle Frequenzen hinweg. Die folgende Rotationssymmetrie lässt sich [KJ09] 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.

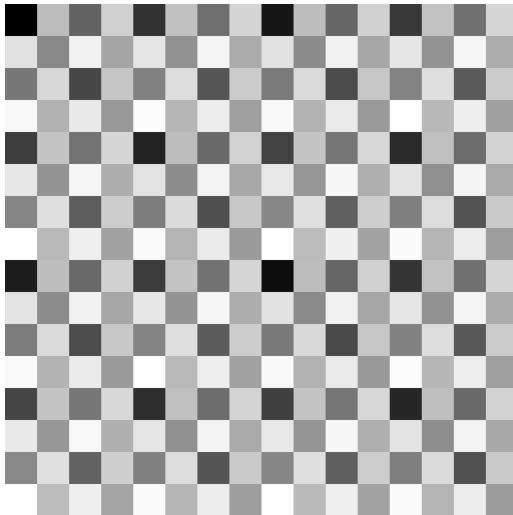


Abbildung 2.7: 512^2 bayer pattern Textur

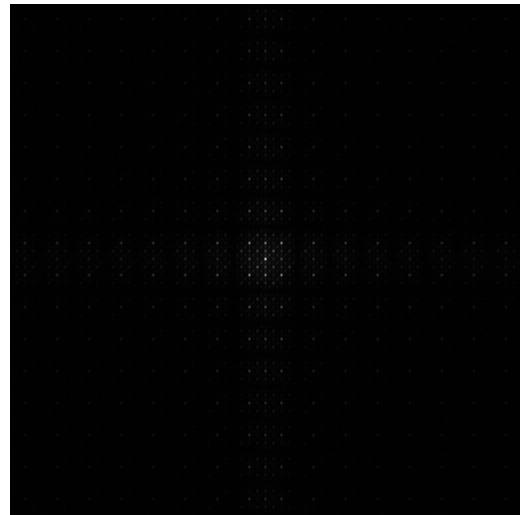
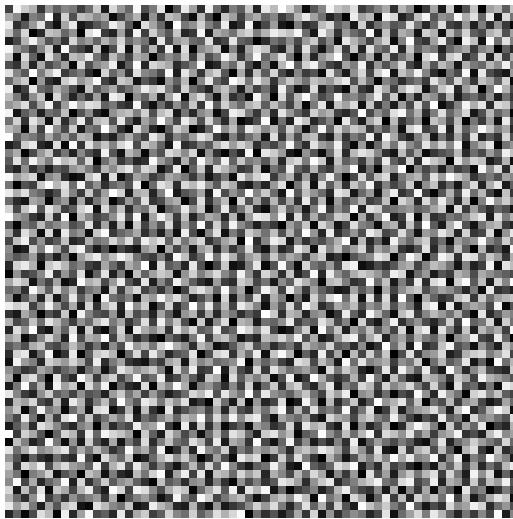
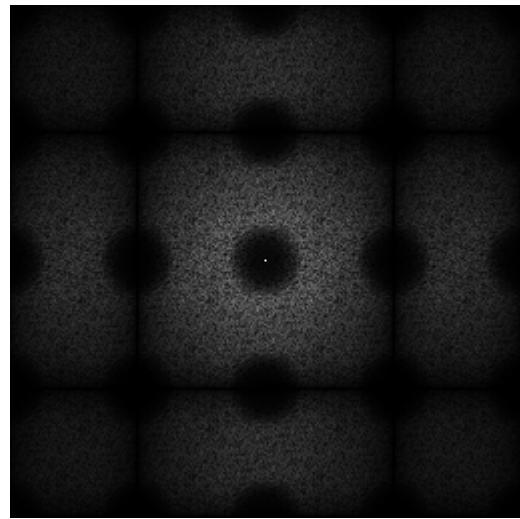


Abbildung 2.8: Amplitudendichte 512^2 bayer pattern Textur

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 [Wro16] zu sehen. Bieten allerdings eine sehr effiziente Verwendung, da sehr leistungssparende GPU Befehle.

2.2.1.3 Niedrige Frequenzen

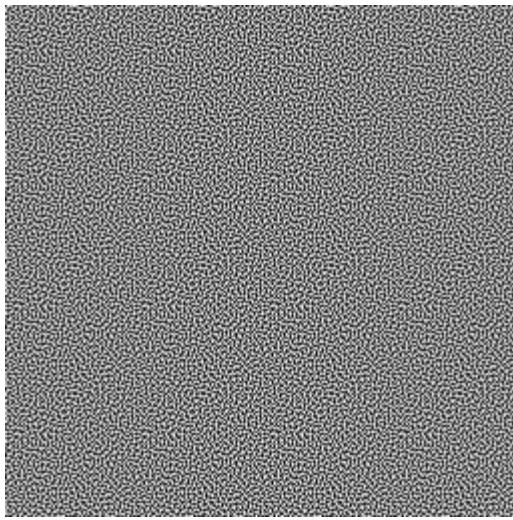
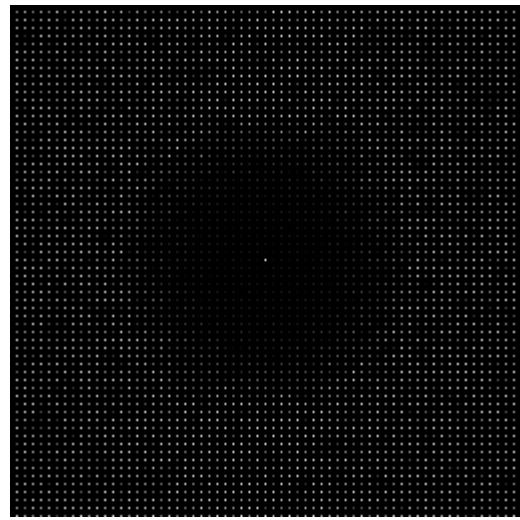
Niedrige Frequenzen sind in einer blue noise sehr wenig bis gar nicht vertreten. Dies ist an dem schwarzen Ring innerhalb der Amplitudendichte zu erkennen^{2.2.1.3}. Oder in der Zeitdomäne, an dem Abhandensein von gleichfarbigen Pixelbündeln. Genau dies wollten wir erreichen. Außerdem haben wir aus den vorherigen Beispielen gesehen: Wir wollen eine Uniformität 2.5 und eine gleichmäßige Verteilung in allen Richtungen.

Abbildung 2.9: 512^2 blue noise TexturAbbildung 2.10: 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 [Kra18] wollen wir diese Textur soweit oben wie möglich in der Cachehierarchie halten.(L1 96KByte, L2 6MByte).

Abbildung 2.11: 512^2 gekachelte Textur von 64x64Abbildung 2.12: Fourier Spektrum 512^2 4-fach getiled 64x64 Textur

In 2.2.1.4 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 2.2.1.2, 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

[Owe98] [HBO⁺19]

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 [Rob18] gezeigt wird, ist diese Form von Sequenz die beste für das ??.

2.3.3 2-Dimensionen

Für mehrere Dimensionen, hier zwei, kombiniert man in gängigen Methoden einfach zwei 2.3.1 eindimensionale Sequenzen. Für unsere Zwecke untersuchen wir hier die Generalisierung des bereits zuvor beschriebenen goldenen Schnitts 2.3.1, wie hier [Krc06] beschrieben. Die sogenannte Plastische Zahl in 2.3.3 ist die Lösung der Gleichung 2.9

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

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

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

[Pad02] [PW] Folgende Gleichung ist auch einfach erweiterbar für höhere Dimensionen.

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

Dabei ist Φ_d der goldene Schnitt. Φ_d^2 ist Lösung der 2.8 obigen Gleichung.

[Rob18] (**find the right place for this chapter**)

ToDo

```

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

3. Temporaler Algorithmus

In diesem Abschnitt wird auf den in [EH19] 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 *nachträglichen Eigenschaften* 3.1 gewonnenen 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.

[Pet16] 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 [Pet16] 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.

3.1 A Posteriori

Der nun behandelte temporale Algorithmus von [EH19] beruht im Gegensatz zu [GF16] auf *nachträglichen* Annahmen. Welches zur Folge hat, dass die Dimension unseres Path Tracers 2.1, 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 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 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 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 [EH19] das Histogramm mit dem vorherigen Frame approximieren. Bereits vorherige Arbeiten [SPD18] 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.

3.2 Sorting

In diesem Schritt wollen wir nun die Untersuchungen aus 3.1 durchführen. Nach dem Rendern eines $Frame_t$ (vor dem Rendern von $Frame_{t+1}$) approximieren wir das Histogramm der Pixelwerte anhand der Pixelwerte von $Frame_t$. Dabei betrachten wir Anzahl Pixel pro BLOCK in der unmittelbaren Nachbarschaft und nehmen diese wie anfangs erwähnt als Schätzung des Histogramms.

[EH19]

Algorithm 2 Sortier Schritt t nach dem Rendern von Frame t und vor dem Rendern von Frame t+1

```

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

```

Hierbei muss noch eine wichtige Anmerkung gemacht werden. Die Fehlerverteilung der Pixelwerte im Bildraum konvergiert auf diese Weise nicht zu einer blue noise Verteilung. 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.

3.3 Retargeting

[EH19]

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 3 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

```

3.4 Simulated Annealing

Im vorherigen Kapitel, dem Retargeting Schritt 3.3, 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 3.2 akkumuliert und die optische

Aufwertung erst richtig sichtbar. Die retarget Textur wird mit Hilfe von **simulated annealing** [EH19] 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 Phase 3.2 verwendet haben, wichtig.

Die Funktion nach der optimiert wird ist an die Formel aus [GF16] angelehnt.

$$E(M) = \sum_{p \neq q} E(p, q) = \sum_{p \neq q} e^{-\frac{\|p_i - q_i\|^2}{\sigma_i^2} - \frac{\|p_s - q_s\|^{d/2}}{\sigma_s^2}} \quad (3.3)$$

Wähle nach [Uli93b] $\sigma_i = 2.1$ 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.

Literaturverzeichnis

- [BYF⁺18] Nir Benty, Kai Hwa Yao, Tim Foley, Matthew Oakes, Conor Lavelle und Chris Wyman: *The Falcor Rendering Framework*, Mai 2018. <https://github.com/NVIDIAGameWorks/Falcor>, <https://github.com/NVIDIAGameWorks/Falcor>.
- [Caf98] Russel E. Caflisch: *Monte Carlo and quasi-Monte Carlo methods*. Acta Numerica, 7:1–49, 1998.
- [DS02] G. Drettakis und H. P. Seidel: *Efficient Multidimensional Sampling*. 21:1–8, 2002.
- [EH19] Laurent Belcour Eric Heitz: *Distributing Monte Carlo Errors as a Blue Noise in Screen Space by Permuting Pixel Seeds Between Frames*. 38:1–10, 2019. <https://hal.archives-ouvertes.fr/hal-02158423/document>.
- [Gam17] Epic Games: *The problem with 3d blue noise*, 2017. <http://momentsingraphics.de/3DBlueNoise.html>, Blogpost.
- [GF16] Iliyan Georgiev und Marcos Fajardo: *Blue-noise dithered sampling*. In: *ACM SIGGRAPH 2016 Talks*, Seite 35. ACM, 2016.
- [HAM19] Eric Haines und Tomas Akenine-Möller (Herausgeber): *Ray Tracing Gems*. Apress, 2019. <http://raytracinggems.com>.
- [HBO⁺19] Eric Heitz, Laurent Belcour, Victor Ostromoukhov, David Coeurjolly und Jean Claude Iehl: *A Low-Discrepancy Sampler that Distributes Monte Carlo Errors as a Blue Noise in Screen Space*. In: *SIGGRAPH’19 Talks*, Los Angeles, United States, Juli 2019. ACM. <https://hal.archives-ouvertes.fr/hal-02150657>.
- [JCr18] *jcrystal*, 2018. <http://www.jcrystal.com/products/ftlse/index.htm>, performing fft on images.
- [Jim14] Jorge Jimenez: *A new generalization of the golden ratio*. THE FIBONACCI ASSOCIATION, 2014. http://advances.realtimerendering.com/s2014/index.html#_NEXT_GENERATION_POST.
- [KJ09] Uwe Kiencke und Holger Jäkel: *Signale und Systeme*. Oldenbourg Verlag, 2009.
- [Kra18] Benjamin Kraft: *Aufbau Turing Architektur*. blog-post, 2018. <https://www.heise.de/newsticker/meldung/GeForce-RTX-Nvidia-verraet-mehr-zur-Technik-von-Turing-4165369.html>.
- [Krc06] Vedran Krcadinac: *A new generalization of the golden ratio*. Fibonacci Quarterly, 44(4):335, 2006.
- [MS09] Steve Marschner und Peter Shirley: *Fundamentals of computer graphics*. CRC Press, 2009.
- [Owe98] Art B Owen: *Scrambling Sobol’and Niederreiter–Xing points*. Journal of complexity, 14(4):466–489, 1998.
- [Pad02] Richard Padovan: *Dom Hans Van Der Laan and the Plastic Number*. Nexus IV: Architecture and Mathematics, Seiten 181–193, 2002. <http://www.nexusjournal.com/conferences/N2002-Padovan.html>.

- [Pet16] Christoph Peters: *Free blue noise textures.* blogpost, 2016. <http://momentsingraphics.de/BlueNoise.html#BayerMatrix>.
- [PW] Floor; Piezas, Tito III; van Lamoen und Eric W. Weisstein: *Plastic Constant.* <http://mathworld.wolfram.com/PlasticConstant.html>.
- [Rob18] Martin Roberts: *The unreasonable effectiveness of quasirandom sequences.*, <http://extremelearning.com.au/unreasonable-effectiveness-of-quasirandom-sequences/>, 2018.
- [Sch19] Christoph Schied: *Real-time path tracing and denoising in Quake 2.* Game Developer Conference, 2019. <https://www.gdcvault.com/play/1026185/>.
- [SPD18] Christoph Schied, Christoph Peters und Carsten Dachsbacher: *Gradient Estimation for Real-time Adaptive Temporal Filtering.* Proc. ACM Comput. Graph. Interact. Tech., 1(2):24:1–24:16, August 2018, ISSN 2577-6193. <http://doi.acm.org/10.1145/3233301>.
- [Uli88] R. A. Ulichney: *Dithering with blue noise.* Proceedings of the IEEE, 76(1):56–79, Jan 1988, ISSN 1558-2256.
- [Uli93a] Robert A. Ulichney: *Void-and-cluster method for dither array generation.* In: Jan P. Allebach und Bernice E. Rogowitz (Herausgeber): *Human Vision, Visual Processing, and Digital Display IV*, Band 1913, Seiten 332 – 343. International Society for Optics and Photonics, SPIE, 1993. <https://doi.org/10.1117/12.152707>.
- [Uli93b] Robert A Ulichney: *Void-and-cluster method for dither array generation.* In: *Human Vision, Visual Processing, and Digital Display IV*, Band 1913, Seiten 332–343. International Society for Optics and Photonics, 1993.
- [Whi] WhiteNoiseGenerator. <https://www.cssmatic.com/noise-texture>. Accessed: 24.11.2019.
- [Wro16] Bart Wronski: *Dithering part 1-5.* blogpost, 2016. <https://bartwronski.com/2016/10/30/dithering-part-one-simple-quantization/>.

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 6. Dezember 2019

(Jonas Heinle)