

# 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: Prof. Dr. Hartmut Prautzsch  
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 Zufall . . . . .	3
2.1.1 Pseudozufallszahlen . . . . .	3
2.1.2 Uniforme Wahrscheinlichkeitsverteilung . . . . .	3
2.1.3 Quasi-Zufallsfolgen . . . . .	3
2.1.3.1 Goldener Schnitt . . . . .	4
2.1.3.2 1-Dimension . . . . .	4
2.1.3.3 2-Dimensionen . . . . .	4
2.2 Simulated Annealing . . . . .	6
2.2.0.1 Algorithmik . . . . .	6
2.2.1 Abkühlfunktion . . . . .	9
2.2.2 Visualisierung . . . . .	11
2.3 Rasterisierung . . . . .	14
2.3.1 Beschränktheit . . . . .	16
2.4 Path Tracer . . . . .	18
2.4.1 Monte-Carlo-Integration . . . . .	18
2.4.2 Funktionsweise . . . . .	19
2.4.3 DirectX Raytracing . . . . .	20
2.4.4 Temporale Lösungsansätze . . . . .	22
2.5 A Posteriori . . . . .	23
2.5.1 Praktische Durchführung . . . . .	24
2.6 Blue Noise . . . . .	26
2.6.1 Eigenschaften . . . . .	26
2.6.1.1 Uniformität . . . . .	26
2.6.1.2 Niedrige Frequenzen . . . . .	27
2.6.1.3 Isotropie . . . . .	27
2.6.1.4 Kachelung . . . . .	28
2.7 Dithering Sampling . . . . .	29
2.8 Render Graph . . . . .	30
<b>3 Temporaler Algorithmus</b>	<b>31</b>
3.1 Sorting . . . . .	33
3.1.0.1 Blockgröße . . . . .	34
3.2 Retargeting . . . . .	38
3.3 Temporales Projizieren . . . . .	42
3.4 Rechenaufwand und Speicherbedarf . . . . .	46

<b>4 Implementierung</b>	<b>48</b>
4.0.1 Falcor . . . . .	48
4.0.2 Anfangswerte . . . . .	48
4.0.3 Simulated Annealing . . . . .	48
4.0.3.1 FreeImage . . . . .	48
4.0.3.2 Visualisierungen . . . . .	48
<b>5 Zukünftige Arbeit</b>	<b>49</b>
<b>Literaturverzeichnis</b>	<b>50</b>



# 1. Prelude

## 1.1 Abstract

Die Bildberechnung durch hardwareunterstützte Strahlenverfolgung mit dazugehörigen Techniken gewinnt gegenwärtig in der Echtzeitcomputergrafik an Bedeutung. Trotz dieser neuen Hardwareunterstützung gibt es eine Limitierung bei der Anzahl verschossener Strahlen. Einhergehend mit dieser Limitierung sind wenige Strahlenpfade mit dementsprechend geringer Strahlentiefe. Bereits frühere Arbeiten haben, um dem so entstehenden Bildrauschen entgegenzuwirken, die Blue Noise Fehlerumverteilungen miteinbezogen. Diese Einbeziehung wurde aus der Kenntnis der Kontrastsensitivität des menschlichen Auges und des hohen Kontrastes einer Blue Noise getroffen. Damit lässt sich eine Steigerung der wahrnehmbaren Bildqualität erzielen. Diese Arbeit erläutert einen zeitlich stabilen Algorithmus aufgrund dieser Technik. Wir wollen eine Fehlerumverteilung direkt im Bildraum anwenden, um so eine entsprechend korrelierte Pixelfolge zu erhalten. Der Algorithmus erreicht dies im Vergleich zu einer Lösung ohne Fehlerumverteilungen im Bildraum ohne signifikant mehr Rechenaufwand und Speicherbedarf.

## 1.2 Einleitung

Das *q2vkpt*-Projekt [Sch19] zeigt beispielhaft den aktuellen Übergang in Echtzeitanwendungen, indem es in einem konventionellen Spiel die (teilweise) konventionelle Bilderzeugung mit neuen Technologien des *Real-Time Raytracing* austauscht.

Abschnitt 2.3 beschreibt die bisherige, konventionelle Herangehensweise und zeigt deren Limitierungen auf. Diese Limitierungen führen uns zu einem Ansatz, der im

Abschnitt 2.4 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 [GF16] 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 2.6 über Blue Noise, in welchem wir uns die Theorie aneignen und ihre Funktionsweise auf die Steigerung der Bildqualität genau anschauen. Dabei liefert uns [Pet16] eine Blue Noise Textur, welche wir im

Kapitel 3 in einem temporalen Algorithmus, vorgestellt in [EH19], verwenden können. In zwei zusätzlichen Schritten, dem Sorting und Retargeting, lassen sich unsere Pixel im Bildraum so korrelieren, dass eine zeitlich stabile Fehlerverteilungen entsteht. Diese Arbeit stellt einen zusätzlichen Schritt in Abschnitt 3.3: das temporale Projizieren vor, um die zeitliche Stabilität zu steigern. Dabei machen wir uns die Erkenntnisse aus Abschnitt 2.1.3 zu nutze, um nur eine Textur zu nutzen ohne jedoch auf den Effekt von mehreren durchwechselnden Texturen verzichten zu müssen. Neben der vorberechneten Blue Noise Textur verwenden wir eine weitere *Retarget*-Textur, welche wir erhalten, indem ein Optimierungsproblem mit Hilfe der im

Abschnitt 2.2 vorgestellten Technik, dem Simulated Annealing, gelöst wird.

## 2. Grundlagen

Um den temporalen Algorithmus für eine zeitlich stabile Blue Noise Fehlerverteilung im Bildraum umzusetzen, werden wir uns die zugrundliegende Bildsynthesetechnik, den Path Tracer, anschauen. Dazu benötigen wir Kenntnisse aus der Wahrscheinlichkeitstheorie, welche uns zusätzliche Einblicke auf von uns gewählte Texturzugriffe verschafft. Zugriffe auf vorberechnete Texturen, welche wir zum Teil in einer eigenen Umsetzung des Algorithmus: Simulted Annealing erstellen.

### 2.1 Zufall

#### 2.1.1 Pseudozufallszahlen

In dieser Arbeit machen wir Gebrauch von deterministisch erzeugten, zufällig erscheinenden Zahlensequenzen (Abschnitt 4 für Umsetzung). Sie erscheinen zufällig, verletzen jedoch gewisse Aspekte echter Zufallszahlen. So werden von einem Generator solcher pseudozufälligen Zahlen, bei selben Anfangswert, gleiche Zahlensequenzen erzeugt. Für unsere Anforderungen langt der Pseudozufall mit seiner überzeugenden Schnelligkeit des Generierungsprozesses.

#### 2.1.2 Uniforme Wahrscheinlichkeitsverteilung

An einigen Stellen machen wir uns die Uniformität von Wahrscheinlichkeitsverteilungen zu Nutze. Dabei ist vereinfacht gemeint: Eine Grundmenge  $G$ , die Elemente im Intervall  $[a,b]$  besitzt, die auf diesem Intervall gleichverteilt sind, wird bei einem (pseudo)zufälligen Zugriff jedes Element gleichwahrscheinlich auswerfen.

#### 2.1.3 Quasi-Zufallsfolgen

Um ungewollten strukturellen Artefakten entgegenzuwirken, wird empfohlen (so auch in [Pet16]) mehrere verschiedene Blue Noise Texturen in ein Array zu laden und diese in aufeinanderfolgenden Zeitpunkten randomisiert zu verwenden. Wir führen hier Kenntnisse über Quasi-Zufallsfolgen ein (siehe auch [Rob18]) um aus den zuvor (nach Abbildung 2.6.1.4) genannten Gründen nur eine kleine Blue Noise Textur verwenden zu können.

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

$$L_x(\delta) = f_\delta(x) \quad (2.1)$$

minimieren (siehe Abschnitt 2.6.1.2) und dabei die Uniformität (siehe Abschnitt 2.6.1.1) erhalten. Beide Eigenschaften haben wir bereits in dem Blue Noise Abschnitt behandelt. Auf quasi-zufällige Zahlenfolgen angewandt bedeutet das: Wir benutzen den gesamten Raum an Zufallszahlen *uniform* und hohe Frequenzen vermeiden Regionen, aus den viele Punkte kommen, die wieder punktarme Regionen zur Folge hätten. Im Folgenden wird für uns das Ziel sein, den quasi-zufälligen Zugriff auf eine Textur mit zwei Dimensionen (siehe Abschnitt 2.1.3.2) zu verstehen und nähern uns dabei über den Fall der Eindimensionalität (siehe Abschnitt 2.1.3.1). Ein Weg quasi-Zufallsfolgen zu beschreiben sind die zugrundeliegenden Parameter. Wir werden uns hier Folgen anschauen, die als Basisparameter den goldenen Schnitt (Gleichung 2.2) verwenden.

### 2.1.3.1 Goldener Schnitt

Der goldene Schnitt und die Generalisierung zur plastischen Zahl (Gleichung 2.6) ist mit-samt ihren Eigenschaften bereits früh beschrieben worden [Pad02]. Als wichtiges Seitenverhältnis in der Architektur konnte Sie durch verschiedene Arbeiten ihren Eingang in die Mathematik finden [Krc06].

$$\Phi_1^2 - \Phi_1 - 1 = 0 \implies \Phi_1 \approx 1.6180339887 \quad (2.2)$$

### 2.1.3.2 1-Dimension

Wir benutzen Rekurrenz Sequenzen, basierend auf irrationalem Bruchrechnen der Form

$$R_1(\alpha) : t_n = s_0 + n\alpha(\text{mod}1); n = 1, 2, 3, \dots \quad (2.3)$$

wobei  $\alpha \in \mathbb{I}$  und das  $(\text{mod } 1)$  einen "*toroidally shift*" bezeichnet. Konkret bedeutet das, dass man für jedes neu berechnete  $t_n$  den Bereich vor dem Dezimalpunkt abschneidet und den Bereich danach weiterführt. Will man mit dieser Formel eine Sequenz mit möglichst geringer Abweichung schaffen, so wählen wir den goldenen Schnitt 2.2.

### 2.1.3.3 2-Dimensionen

Da uns die Generalisierung des goldenen Schnittes auf die Lösung der Gleichung 2.4 führt

$$x^{d+1} = x + 1 \quad (2.4)$$

ist das Lösen der kubischen Gleichung 2.5

$$x^3 - x - 1 = 0 \quad (2.5)$$

für den zweidimensionalen Fall nötig. Die Generalisierung und Erweiterung des goldenen Schnittes wurde bereits ausgiebig erforscht [Krc06].

Die sogenannte Plastische Zahl in ist die Lösung der Gleichung 2.5

$$\Phi_2 \approx 1.32471795724$$
(2.6)

Die eindimensionale Rekurrenzsequenz 2.3 ist einfach erweiterbar für höhere Dimensionen.

$$t_n = n\alpha(\text{mod}1), n = 1, 2, 3, .. \alpha = \left(\frac{1}{\Phi_d}, \frac{1}{\Phi_d^2}\right),$$
(2.7)

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

```

1 float g = 1.32471795724474602596; //Plastische Zahl
2 int a1 = (1.0/g) * blue_noise_mask_width * frame_count;
3 int a2 = (1.0/(g*g)) * blue_noise_mask_height * frame_count;
4 int2 offset = (a1,a2);
5 int2 new_index = offset + old_index;
6 new_index.x = new_index.x % blue_noise_mask_width; //toroidally shifted
7 new_index.y = new_index.y % blue_noise_mask_height; //toroidally shifted

```

## 2.2 Simulated Annealing

Für unseren temporalen Algorithmus (siehe Abschnitt 3) gibt es einen wichtigen Retargeting Schritt. In diesem Schritt wird eine vorberechnete Textur verwendet. Diese speichert eine Permutation, die unsere Blue Noise Textur vom Bild  $t$  in eine Blue Noise Textur von Bild  $t+1$  umwandelt. Diese Permutation wird dann auf die Startwerte angewandt, bevor das nächste Bild  $t+1$  erzeugt wird (siehe auch Übersicht 2.26). Dadurch werden die Blue Noise Umverteilungen der Sorting Phase akkumuliert. All diese Vorberechnungen sind möglich, da wir mit „nur“ quasi-zufälligen Sequenzen (siehe Abschnitt 2.1.3) arbeiten. Das andauernde Permutieren von Pixeln bis zu einem Punkt, an dem ein Bild aussieht wie das andere, ist ein klassisches TSP, wofür es aktuell keine effiziente optimale Lösungsmethode gibt. Da wir nur an einer sehr guten Lösung, nahe dem globalen Optimum, interessiert sind, greifen wir wie in [EH19] vorgeschlagen auf das heuristische Approximationsverfahren, dem Simulated Annealing, zu. Aus der Metallurgie kommend fand das Simulated Annealing seine Verwendung in der Algorithmik. So fand man ursprünglich bei erhitztem Metall heraus, dass man durch einen kontrollierten Abkühlprozess den einzelnen Atomen ausreichend Zeit geben kann eine feste Ordnung und damit feste Kristalle zu bilden. Diese Ordnung entspricht der physikalischen Vorstellung eines energetisch günstigsten Zustandes.

### 2.2.0.1 Algorithmik

Angelehnt an die Erreichung eines energetisch günstigsten Zustandes definieren wir zuerst die Energiefunktion für unsere Textur als pixelweisen Unterschied der sich in Abkühlung befindlichen bereits permutierten Textur und der  $Textur_{t+1}$  (siehe Abbildung 2.1).  $Textur_{t+1}$  ist durch quasi Zufall (siehe Abschnitt 2.1.3) bereits bekannt. Mit Erkenntnissen aus[GF16] ergibt sich

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

Abbildung 2.1: Blue Noise Textur mit Dimension  $N$ ; Pixel  $p$  von abkühlende  $Textur_{t=0}$ ; Pixel  $q$  von  $Textur_{t=1}$

Da wir in jedem Schritt nur eine Permutation anwenden, vereinfacht sich unsere Energiefunktion 2.1 zu

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

Abbildung 2.2: Zustand  $s_{previous}$  ohne angewandte Permutation

Mit dem definierten Ziel, die Energiefunktion 2.1 zu minimieren und damit einen energieärmeren Zustand anzunehmen, wenden wir in jedem Schritt einer Permutation an. Das physikalische Vorbild, das metallurgische Abkühlen, orientiert sich an der Boltzmann-Statistik 2.3.

$$\exp\left(-\frac{E_j}{k_B * T}\right)$$

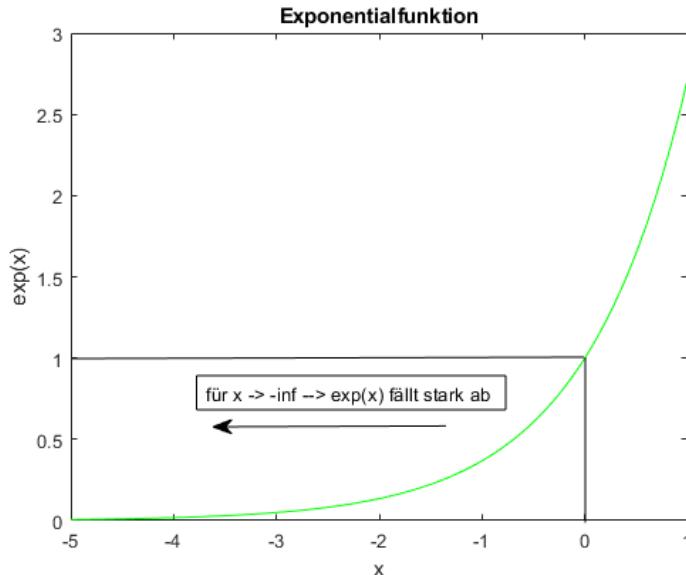
Abbildung 2.3: Boltzmann-Statistik

\*  $k_B$  Boltzmann-Konstante: Hat Dimension Energie/Temperatur; verbindet als Proportionalitätsfaktor Zustände mit ihrer Entropie

Dadurch lässt sich physikalisch die Wahrscheinlichkeit einen Energiezustand  $E_j$  anzutreffen formulieren ( $E_j \propto$  Gleichung 2.3) [KGV83]. Mit diesem physikalischen Vorbild untersuchen wir unsere Akzeptanzwahrscheinlichkeitsfunktion für einen neuen Energiezustand:

$$P = \exp\left(-\frac{\Delta_{energy}}{temperature}\right) \quad (2.8)$$

- (a) Akzeptanzwahrscheinlichkeitsfunktion; Abhängig von Energie und aktueller Temperatur



(b) Günstige Eigenschaft der Exponentialfunktion

Abbildung 2.4

Die günstigen Eigenschaften der Exponentialfunktion (siehe Abbildung 2.4b) als Wahrscheinlichkeitsakzeptanzfunktion sind vielfältig und bereits in weiterführender Literatur wie [KGV83],[VLA87] gut belegt. Eine der Eigenschaften ist in der obigen Abbildung 2.4b dargestellt. Das Argument der Funktion 2.4b hat einen Divisor Temperatur und einen Divedienten  $\Delta_{energy}$ . Mit absteigender Temperatur erkennt man in der Abbildung 2.4b 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 Weise zu einem höheren negativen Exponenten und damit eine geringere Akzeptanz als Energiezustände, die nur bisschen darüber liegen. Die Wahl des Abkühlvorgangs (also das Update der Temperatur über die Zeit) ist problemspezifisch [KGV83, 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. Diese Vorgänge habe ich im Abschnitt 2.2.1 untersucht.

Wir stellen fest, dass sich durch die physikalische Analogie der Akzeptanzwahrscheinlichkeitsfunktion 2.8 zur Boltzmann-Statistik 2.3 Erkenntnisse des zweiten Hauptsatz der Thermodynamik übertragen. Dieser besagt vereinfacht, dass mit Energiezunahme eine Zunahme an möglichen Zustandsübergängen einhergeht.

Nun haben wir alle Begriffe zusammen, um einen ersten Blick auf den Algorithmus zu werfen.

---

### Algorithm 1 Simulated Annealing

---

```

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 Abkühlplans
5:   //Radius für Nachbarschaftssuche ist auf 6 festgesetzt
6:    $s_{\text{neu}} \leftarrow \text{Nachbarzustand}(s)$  //wende hier die Permutation an!
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  $s$ ;

```

---

Für die abzuspeichernde Permutation gilt Folgendes. Als Startzustand  $s_0$  definieren wir eine Permutation, die alle Elemente auf sich selbst abbildet. Um von einem Zustand  $s$  zu einem neuem Zustand  $s_{\text{new}}$  zu kommen, definieren wir eine Nachbarschaftsfunktion  $\text{Nachbarzustand}()$ . Diese kann zwei Elemente genau dann vertauschen, wenn Sie in einem gegenseitigen Radius  $r = 6$  erreichbar sind (folgend der Empfehlung aus [EH19, S.7]) Dabei vertauschen wir in jedem Schritt ein Pixelpaar. Die Wahrscheinlichkeitsfunktion zur neuen Zustandsannahme  $P(\text{Energie}(s), \text{Energie}(s_{\text{new}}))$  beschreibt, ob wir den neu gewählten Zustand  $s_{\text{new}}$  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.

### 2.2.1 Abkühlfunktion

Für die Wahl unserer Abkühlfunktion bieten sich einige Möglichkeiten (siehe Abbildung 2.5). Im Folgenden wird auf die verschiedenen möglichen Abkühlfunktionen ([Sci20]) und ihre Eigenschaften sowie die Wahl interner Parameter(z.B. Starttemperatur, Gleichgewicht) eingegangen. Denn diese Funktion trägt maßgeblich mit ihrem Konvergenzverhalten zur Effizienz des Abkühlvorgangs bei. So beeinflusst Sie auch unsere wichtige Wahrscheinlichkeitsakzeptanzfunktion 2.8. Nach [KGV83] wählen wir die Anfangstemperatur  $T_0$  derart, dass anfangs jede Neue generierte Lösung akzeptiert wird. Außerdem werden wir einen Zustand des Quasiequilibriums (Gleichgewicht) definieren. Für einige Abkühlvorgänge wird es sinnvoll sein, erst nach einer bestimmten Anzahl von erfolgreichen Zustandsübergängen die Temperatur zu senken. Dazu beim jeweiligen Vorgang mehr.

#### Hajek

$$f(t) = T_0 \log(1 + t) \quad (2.9)$$

In [Haj88] haben wir eine Abkühlfunktion gegeben, welche durch ihre Eigenschaft, stets gegen das globale Maximum zu konvergieren, unter allen Anderen heraussticht. In Abbildung 2.5 angedeutet und in weiteren Beobachtungen bestätigt hat sich allerdings auch ihre sehr langsame Konvergenz. Sie hat sich daher für diese Aufgabenstellung als nicht nützliche Abkühlfunktion herausgestellt.

#### Linear

$$f(t) = T_0 - \mu * t \quad (2.10)$$

Typische Werte für  $\alpha$  liegen zwischen 0.8 and 0.99. Wie man in Abbildung 2.5 erkennen kann, ist das Problem der linearen Abkühlung die extreme Langsamkeit. Anstatt nur am Anfang schlechtere Energiezustände zuzulassen, um lokale Minima zu verlassen, geht der Algorithmus durch diesen Abkühlvorgang in ein bloßes randomisiertes Tauschen von Pixeln über.

#### Exponential

$$f(t) = T_0 * pow(\alpha, t) \quad (2.11)$$

Ist nach [KGV83] eine für viele Fälle zutreffende und zu wählende Abkühlfunktion. Wobei  $\alpha \in [0.8; 0.99]$ .

Wie in Abbildung 2.5 zu erkennen, haben wir hier im Vergleich zu den vorherigen Vorgängen eine deutlich schnellere Abkühlung. Jedoch lässt sich hier das andere Extrem, im Vergleich zum bloßen randomisierten Vertauschen von Pixelpaaren, erkennen: Wir verharren viel zu kurz in einem Temperaturzustand, geraten daher schnell in einen *greedy* Zustand und manche Bildbereiche bleiben in einem lokalen Minima hängen.

#### Inverse

$$f(t) = T_0 / (1 + alpha * step) \quad (2.12)$$

Wie in Abbildung 2.5 zu erkennen, haben wir hier im Vergleich zu den ersten beiden Vorgängen eine deutlich schnellere Abkühlung. Hat jedoch das selbe Problem wie die exponentielle 2.11 Variante.

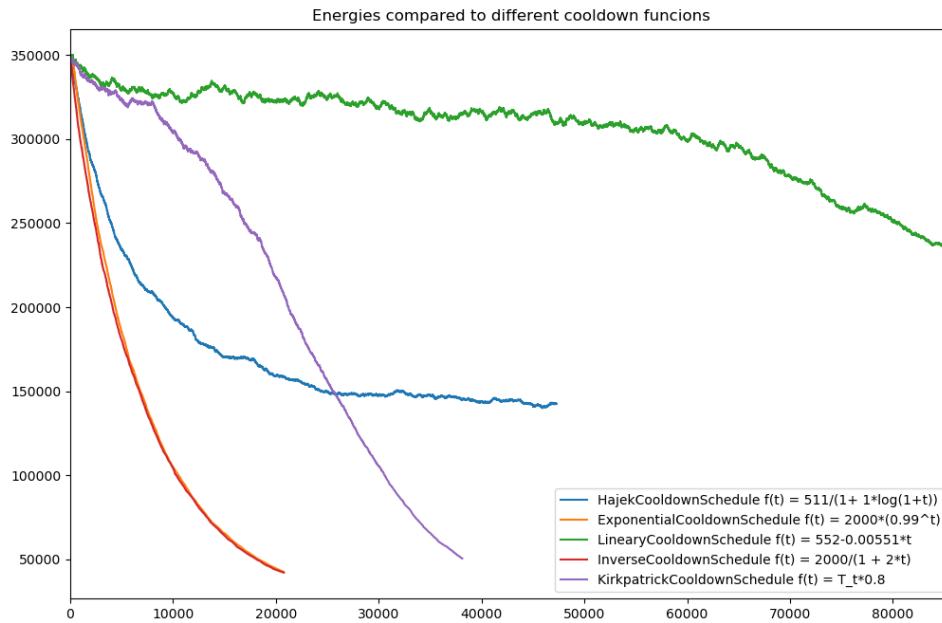


Abbildung 2.5: Vergleich von Abkühlfunktionen mit gesetzten Parametern alle mit 100000 Schritten; auf x-Achse sind erfolgreiche Schritte nach Wahrscheinlichkeitsakzeptanzfunktion aufgetragen

**Kirkpatrick** Wir haben uns bei unserem Optimierungsproblem für einen Abkühlvorgang, der in [KGV83] beschrieben wird, entschieden und in danach benannt. Die Abkühlfunktion hat hierbei exponentiellen Charakter.

$$f(t) = T_0 * \text{pow}(\alpha, t) \quad (2.13)$$

Wobei wieder  $\alpha \in [0.8; 0.99]$  und nach Auflösung der Textur zu wählen ist. Ein anderer Parameter, der nach Auflösung der Textur zu wählen ist, ist das Quasi-Gleichgewicht. Mit dem Quasi-Gleichgewicht lässt sich erreichen, dass jeder Bildabschnitt vor dem jeweiligen Abkühlen jede Temperatur durchläuft und dabei jeden Bildabschnitt davor bewahrt in einem lokalen Minima zu verharren. So lässt sich in einem direkten Vergleich mit dem bloßen exponentiellen Abkühlen in Abbildung 2.5 erkennen, das wir anfangs langsamer abkühlen und immer wieder auch höhere Energiezustände bewusst zulassen.

### 2.2.2 Visualisierung

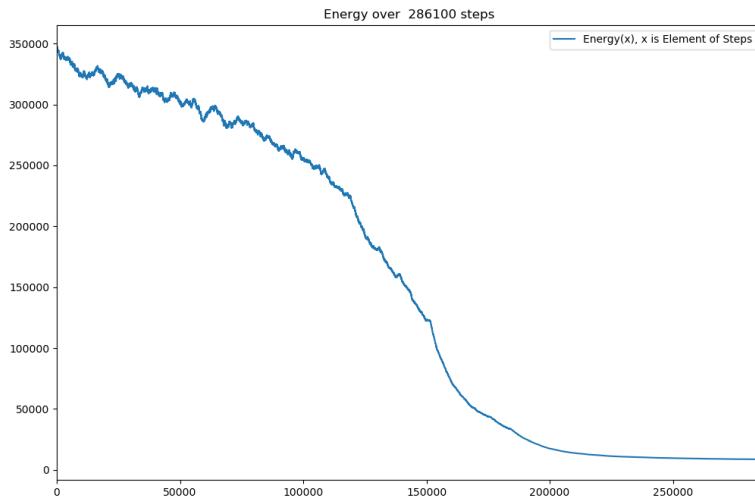


Abbildung 2.6: Energieverlauf beim Simulated Annealing

Die Abbildung 2.6 verdeutlicht, dass wir durch unser Abkühlen insgesamt die Energiefunktion 2.1 minimieren. Dabei akzeptieren wir in Abhängigkeit unserer Schritte, anfangs sehr häufig und am Ende immer seltener, neben günstigeren Zuständen, auch energetische Höherwertige.

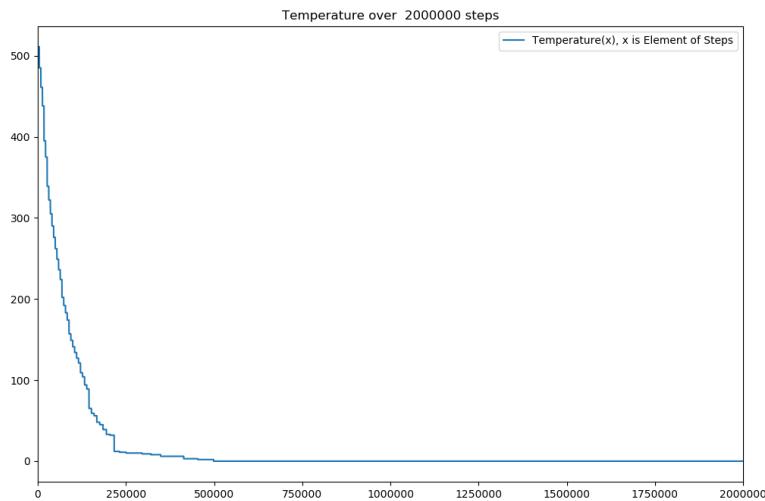


Abbildung 2.7: Temperaturverlauf

Wir wählen die Starttemperatur (siehe Abbildung 2.7) folgendermaßen, dass anfangs alle Permutationen akzeptiert werden. Dazu muss die Akzeptanzwahrscheinlichkeitsfunktion 2.8 auch die energetisch ungünstigste Permutation akzeptieren. Minimiere Argument der Exponentialfunktion 2.4b. In unserem konkreten Fall: Maximales  $\Delta_{Energy}$  bei einem 8-Bit Graustufenbild  $2^*255 = 510$ . Jeweilige Anpassungen müssen bei anderer Auflösung vorgenommen werden.

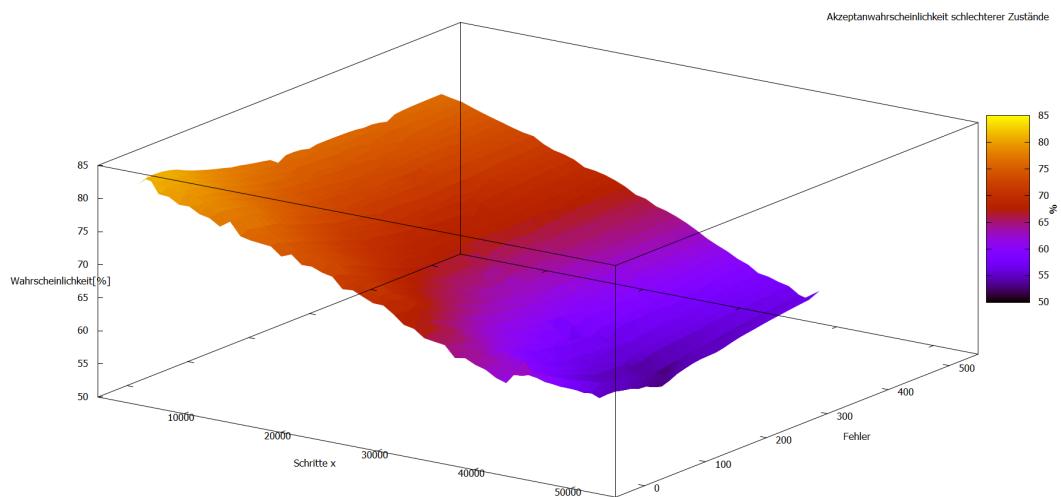


Abbildung 2.8: Akzeptanzfunktionsverlauf bei negativen Energiedeltas

Abbildung 2.8 visualisiert die Akzeptanz (Dimension Prozent) im Verlauf des Algorithmus (Schritte) bezogen auf die mögliche Verschlechterung bzw. den wachsenden Fehler. Anfangs werden sehr viele und sehr große Fehler akzeptiert, wohingegen im weiteren Verlauf vor allem im höheren Fehlerbereich der blaue Bauch auf eine schlechtere Akzeptanz höherer Fehler hindeutet.

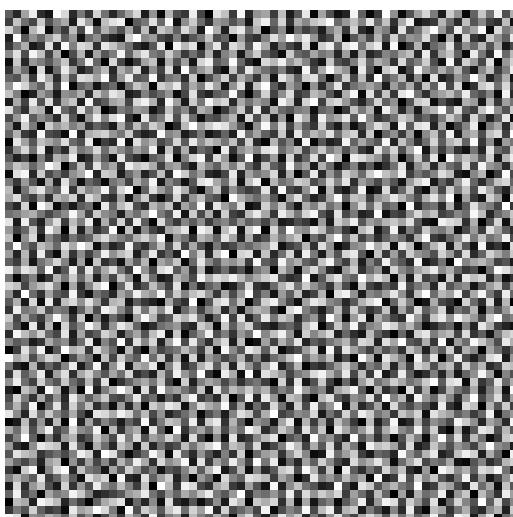


Abbildung 2.9: Blue noise Textur 64x64

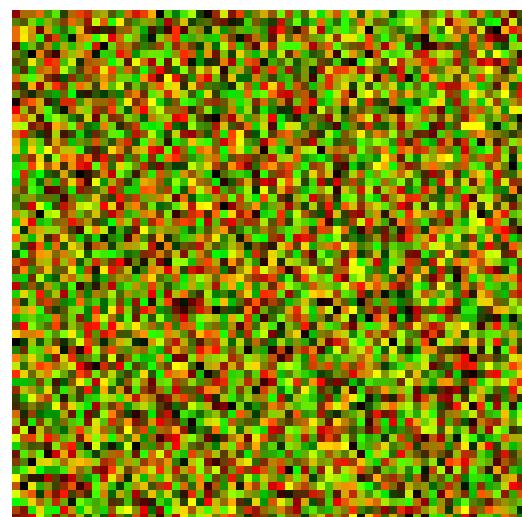


Abbildung 2.10: Permutation; gespeichert in R,G-Channel einer PNG

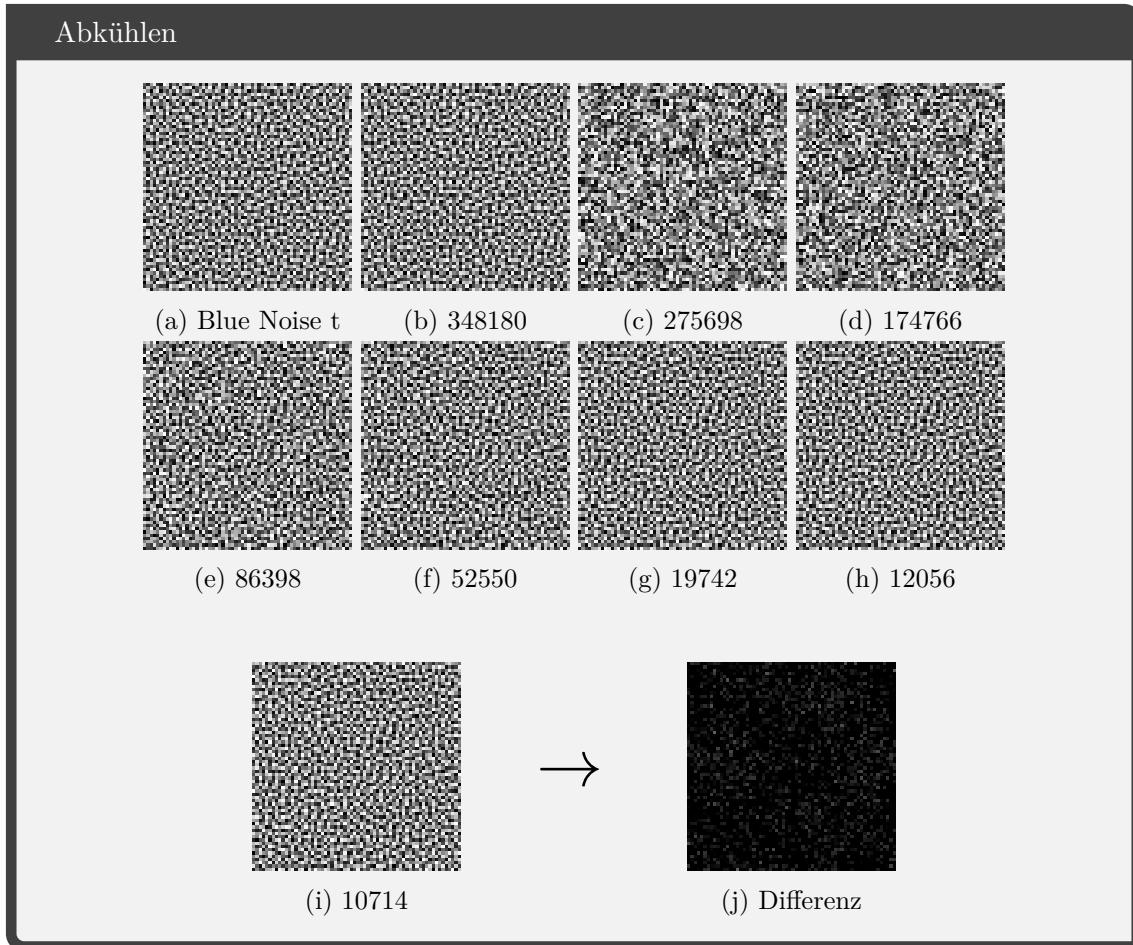


Abbildung 2.11: Der Prozess des Abkühlens mit jeweiliger Auswertung der Energiefunktion 2.1

Nachdem wir in Bild

2.11a unsere Blue Noise Textur zum Zeitpunkt  $t=0$  haben, können wir in dem Bild 2.11c typische Clusterbildungen erkennen, die die Folge von uniformen randomisierten Vertauschungen sind. Diese hatten wir bereits im Abschnitt 2.6 und 2.1.3 besprochen. Diese Vertauschungen sind die Folge der hohen Temperatur, welche wiederum zur Folge haben, dass alle Vertauschungen von der Funktion 2.8 angenommen werden. Die Bilder

2.11c-2.11e sind die Folge der abnehmenden Temperatur und der daraus folgenden geringeren Akzeptanz schlechterer, energiereicherer Zustände. Bessere Zustände werden allerdings immer angenommen und daher die immer bessere Blue Noise Verteilung. Die bessere Verteilung ist eben über die gesamte Textur zu erkennen. Mit abnehmender Energie in den Bildern

2.11g-2.11i lassen wir sehr wenige/keine schlechteren Zustände mehr zu und gelangen durch lokale Permutationen zu einer immer exakteren Verteilung.

2.11j Pixelweise Differenz mit Zieltextur; Durchschnittlicher Fehler/Pixel von  $\approx 2,6$

## 2.3 Rasterisierung

Die Rasterisierungspipeline und deren Hardwarebeschleunigung war das bisherige sehr effiziente „state-of-the-art“ Bilderzeugungsverfahren. Mit der hardwareunterstützten Strahlenverfolgung scheint es sich nun zu aktuell zu verschieben [BBHW<sup>+</sup>19]. Zu Beginn der Rasterisierung haben wir die Eckpunkte der bereits verarbeiteten, transformierten, projizierten Geometrie mit möglichen Beleuchtungsinformationen aus den vorherigen Berechnungen vorliegen (weiterführende Literatur zu der modernen Renderingpipeline [AMHH08]). Mit Hilfe der Rasterisierung wird nun die Farbe jedes einzelnen Pixels bestimmt [Ras20]. 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.

### Algorithm 2 Rasterisierungsalgorithmus

```

1: procedure RASTERISIERUNG(Dreiecke)           ▷ Pipelining der Dreiecke
2:   for dreieck ∈ Dreiecke do
3:     projiereEckpunkteInBild();
4:     for (x,y) ∈ image do
5:       if (x,y) im projizierten Dreieck enthalten then
6:         färbe Pixel mit dreieck.farbe;
7:       end if
8:     end for
9:   end for
10: end procedure
```

Die bereits angesprochene Effizienz hat der Rasterisierungsalgorithmus 2 der rechengünstigen Operation des Projizierens und der einfachen Umsetzung eines Pixelzugehörigkeits- tests für ein Dreieck zu verdanken. Des Weiteren lassen sich verschiedene sehr effiziente Verbesserungen vornehmen, z.B. lassen sich Bounding-Boxen verwenden, womit sich die untersuchten Pixel für jedes Dreieck (stark) reduzieren lassen.

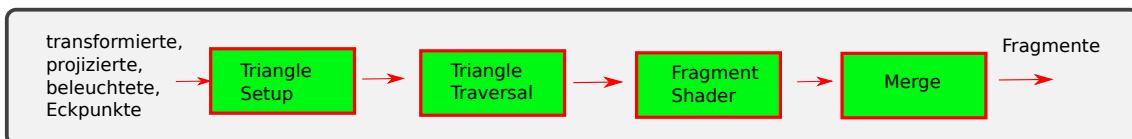


Abbildung 2.12: Ablauf der Rasterisierung

Zuallererst befindet man sich im Ablauf der Rasterisierung (siehe Abbildung 2.12) beim *Triangle Setup*. Unbeeinflussbar vom Programmierer werden hier Daten berechnet, welche zur Pixeleinfärbung benötigt werden. Beim darauffolgenden Schritt, dem *Triangle Traversal*, werden die *Fragmente* erzeugt, indem diejenigen Pixel bestimmt werden, welche innerhalb des Dreiecks liegen. Als freiprogrammierbare Shadereinheit können im *Fragment Shader* vom Programmierer weitere Berechnungen vorgenommen werden. Dazu zählt

eine pro Pixel Beleuchtungsberechnung (Phong Shading). 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 Depth-Werten, welche wir als Tiefe beim Viewport Transform gespeichert haben, gibt es hier Zugang zum Depth-Buffer. Dieser Depth-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 Depth-Buffer und auch der color Buffer mit der Fragmentfarbe aktualisiert. Falls nicht, passiert nichts und das nächste Primitiv bzw. Fragment wird betrachtet. Eine Möglichkeit ist die Ausgabe, bestehend aus Lichten, Normalen, Tiefenwert, Farbe (sog. *GBuffer*) in mehrere verschiedene *render targets* zu schreiben. Wir werden zur Beschleunigung der globalen Beleuchtung durch einen Path Tracer einen solchen durch Rasterisierung berechneten *GBuffer*(siehe auch Abschnitt 2.26) verwenden.

### 2.3.1 Beschränktheit

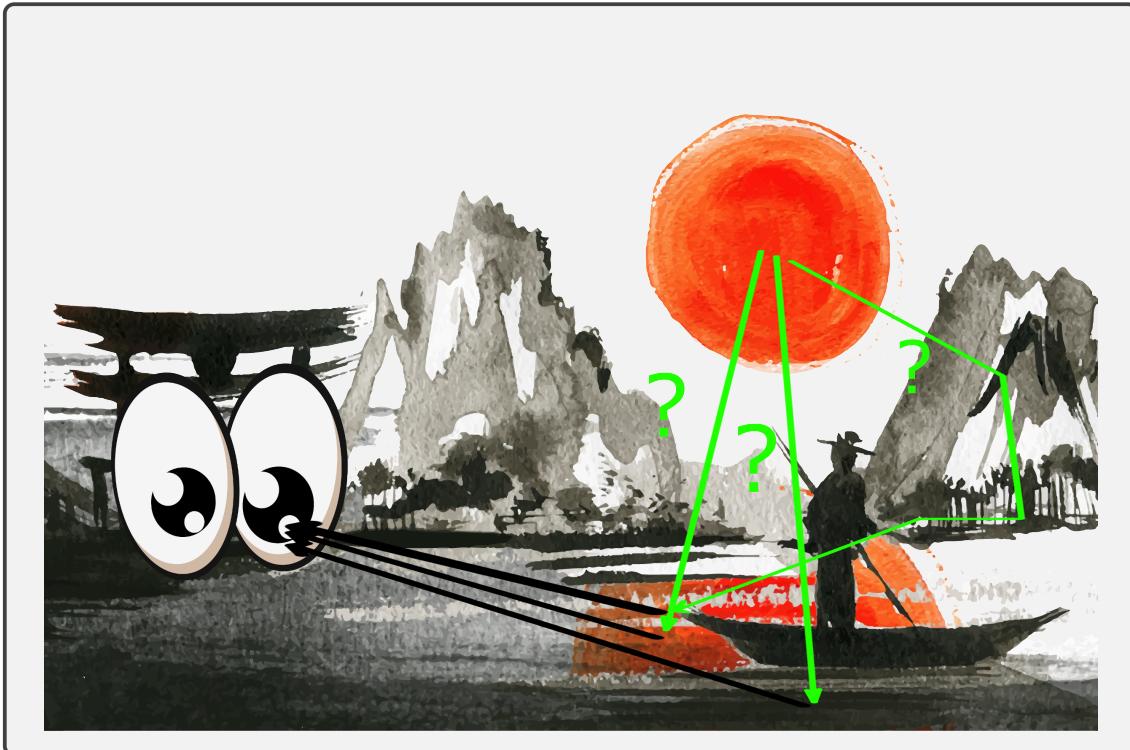


Abbildung 2.13: Ablauf der Rasterisierung

Ihre bisherige weite Verbreitung hatte die Rasterisierung der Objektorientierung zu verdecken: massives paralleles Arbeiten, Ignorieren von (großen) leeren Bereichen und Ausnutzen von Cache Kohärenzen gehören zu den Eigenschaften, welche die enorme effiziente, schnelle Abarbeitung bzw. (relativ) geringe aufzuwendende Rechenleistung begründen. Jedoch liegt in ihr auch die Crux. Die Abbildung der Farbe eines Geometrie/Dreiecks auf einen Pixel simuliert den physikalischen Lichttransport nicht korrekt! Die physikalische Optik lehrt uns das Verfolgen von weiteren (sekundären) Strahlen (siehe grüne Pfeile in Abbildung 2.13) abseits des Primärstrahls, der von Sichtebene zum Objekt verläuft und durch die Rasterisierung im Gegensatz zu den Sekundärstrahlen abgedeckt wird. Abbildung 2.13 verdeutlicht das Problem der Objektorientierung und deren Problem mit Sekundärstrahlen. So können Effekte, welche diese Sekundärstrahlen involvieren, entweder nicht oder nur (unzureichend befriedigend) dargestellt werden (Spiegelungen, Schatten und Pfade mit größerer Pfadlänge).

Die enormen Leistungsanforderungen von Technologien, welche diesen physikalisch korrekten Lichttransport möglich machen, haben sie bisher für Echtzeitanwendungen ausgeschlossen und führten zu nicht physikalisch korrekten Approximationstechniken. In heutigen modernen Grafikprogrammierschnittstellen (Vulkan, DirectX) jedoch befindet sich Raytracing-Funktionalität, welche auf Hardwareseite unterstützt wird. Diese Unterstützung erlaubt neuerdings effizientere image-ordered Bilderstellungen in Echtzeit. Aktuelle Bemühungen gehen nun daran Strahlenerzeugung und Rasterisierung zu kombinieren. [BBHW<sup>+</sup>19] stellte mit dem Spiel *PICA PICA* eine solche Rendering-Pipeline vor, welche mithilfe von Path Tracing (siehe Abschnitt 2.4) arbeitet. Dabei wird der G-Buffer (Texturen speichern Position, Normalen, Belichtung eines Bildes) noch über Rasterisierung berechnet. Direkten Schatten kann man durch rechengünstigere Approximationstechniken 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 temporalen Algorithmus (siehe Kapitel 3), der die visuelle Qualität nicht durch Verschießen von mehr Strahlen, sondern durch eine zeitlich stabile Blue Noise Fehlerverteilungen im Bildraum erreicht.

## 2.4 Path Tracer

Ein verbreiteter Ansatz zur physikalisch korrekten Simulation des Lichttransports ist die Pfadverfolgung. Im Folgenden wird auf die Funktionweise, welche wir zur globalen Beleuchtung innerhalb des Render Graphen (siehe Abbildung 2.26) benutzen, eingegangen. Die vollständige Beschreibung des Lichttransports in einer Szene hebt bisherige Limitierungen der Rasterisierung auf.

Um die Lichtverteilung in einer Szene vollständig zu beschreiben, langt folgende Gleichung 2.14 [Kaj86]

$$L(x, \omega) = L_\epsilon(x, \omega) + \int_{\Omega^+} f_r(\omega_i, x, \omega) L_i(x, \omega_i) * \cos(\delta_i) d\omega_i \quad (2.14)$$

Die Strahldichte eines Punktes  $x$  in Richtung  $\omega$  wird dabei maßgeblich von der grundlegend ausgesendeten Strahlung, dem Emissionsterm  $L_\epsilon(x, \omega)$  sowie dem Integral über die positive Hemisphäre (für reine Reflexion; komplettes Sphärenintegral für zusätzliche Transmissionen) bestimmt. Dabei bestimmt das Integral jedwedes auf den Punkt  $x$  einfalldendes Licht aus der Richtung  $\omega_i$  und berechnet anhand der Reflektanzverteilungsfunktion  $f_r$  seinen jeweiligen reflektierten Anteil in Richtung  $\omega$ . Da die Berechnung eines Integrals über eine Hemisphäre unpraktikabel ist, wird zur photorealistischen Bildsynthese in aktuellen Anwendungen [Pat19] eine Monte-Carlo Integration angewandt.

### 2.4.1 Monte-Carlo-Integration

Mit der Monte Carlo Integration approximieren wir die Rendergleichung 2.14 bei gegebener Dimensionalität  $n$  des Renderintegrals (siehe auch [DS02])

$$\int_a^b f(x) dx \approx \frac{b-a}{N} \sum_{i=1}^N f(x_i) \quad (2.15)$$

Dabei wird das Integral 2.14 approximiert, indem wir die Funktion  $f(x)$  an  $N$  zufälligen, uniform-verteilten Stellen auswerten.

### 2.4.2 Funktionsweise

Das Path Tracing ist eine Methode, die die Lichtverteilung durch Approximation des Hemisphären- mit Hilfe eines Monte-Carlo Integrals 2.15 löst.

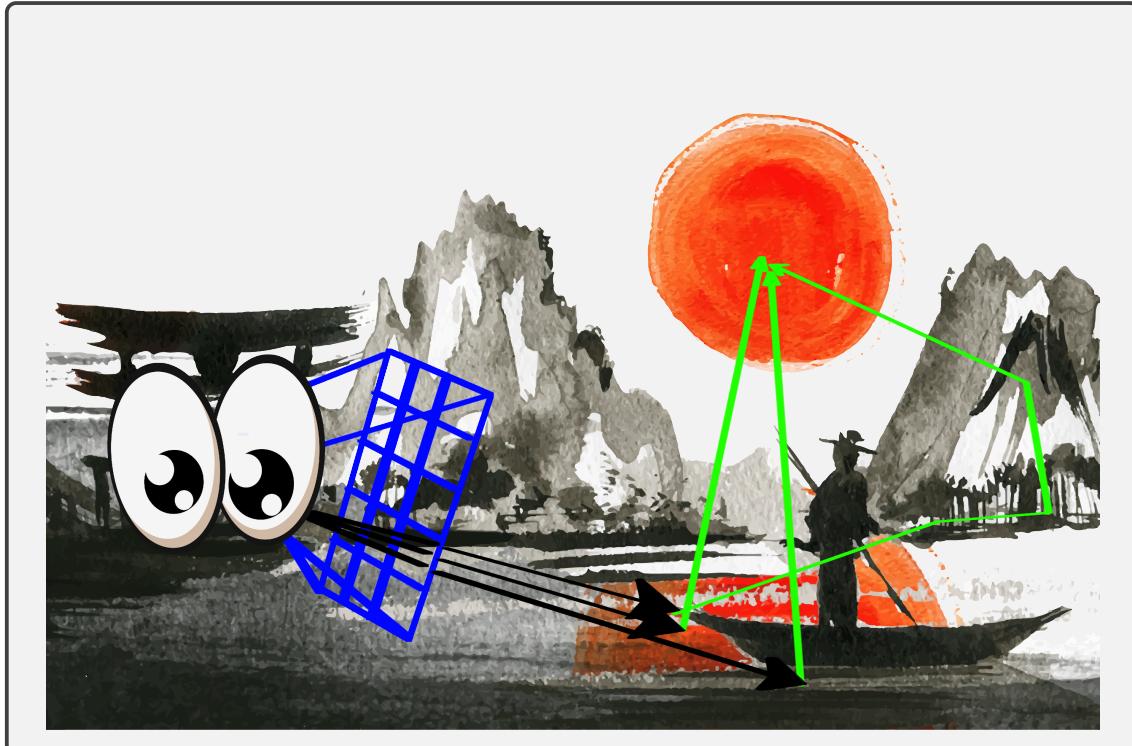


Abbildung 2.14: Sichtebene (blau), Sichtstrahlen(schwarz), Sekundärstrahlen(Schatten,Spiegelung,..) in grün

Zur Näherung werden hier nun für jeden Pixel mehrere Strahlen verschossen, welche jeweils nur einen Folgestrahl haben. Daher entstehen mit dieser Methode pro Pixel mehrere einzelne Lichtpfade. Der hier verwendete Path Tracer wurde mit dem Real-Time Rendering Framework Falcor [BYF<sup>+</sup>18] umgesetzt. Die pro Pixel verschossenen zufälligen Strahlen ergeben hierbei den *Samplewert*. Mehrere unkorrelierte zufällige Folgen ergeben somit die schlussendliche Pixelfarbe. Der Fehler, der bei der zugrundeliegenden Monte-Carlo Integration (siehe Gleichung 2.15) entsteht, wird klassischerweise über Varianzreduktionsmethoden wie *Importance Sampling* angegangen. Das Ziel hierbei ist es, die Fehlerverteilung im Bildraum zu beeinflussen, da diese für die wahrnehmbare visuelle Qualität des Bildes verantwortlich ist. Diese Arbeit wird von diesem Vorgehen abweichen und direkt im Bildraum eine zeitlich stabile Blue Noise Fehlerverteilung durch korrelierte Folgen erreichen. Die positive Auswirkung von Blue Noise Verteilungen auf die visuelle Qualität wurde bereits ausgiebig erforscht [Uli88].

### 2.4.3 DirectX Raytracing

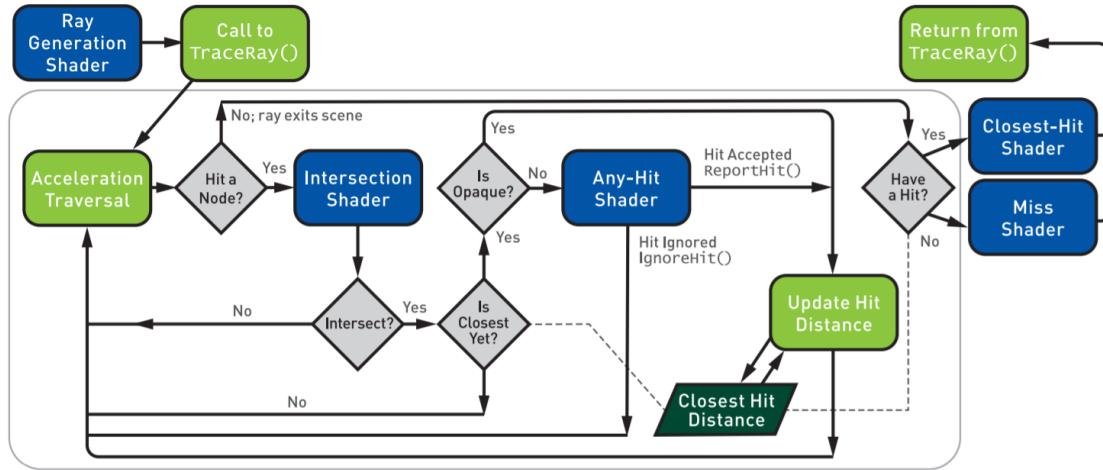


Abbildung 2.15: DirectX Raytracing Pipeline aus [HAM19]

Das hardwareunterstützte Raytracing erhielt Einzug in moderne Programmierschnittstellen (DirectX, Vulkan) und wird hier anhand von DirectX erläutert, welche auch für die *Global Illumination* innerhalb des Render Graphs 2.26 benutzt wurde.

In Abbildung 2.15 und Algorithmus 6 lässt sich der Beginn (Generierung eines Strahles) der neuen Pipeline durch den programmierbaren **Ray Generation shader** erkennen.

---

#### Algorithm 3 Beispielhafter minimalistischer Ray Generation Shader

---

```

1: [shader("raygeneration")]
2: launchIndex = DispatchRaysIndex().xy;
3: for (int i = 0; i < numberRays;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 unserem Strahl unter Anderem unsere Szene inklusive Beschleunigungsstruktur, rayflags (beeinflussten Transparenz, Culling, Abbruch)[Ray19] und einen payload. Mit dem *payload<sub>t</sub>* können wir einen struct mit Informationen jedem einzelnen Strahl mitgeben.

**Algorithm 4** beispielhafter payload

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

Diese Methode **TraceRay()** kann auch innerhalb der anderen Shader zum weiteren verschießen von Strahlen verwendet werden. So beispielweise beim Verschießen eines Schattenstrahls mit flags RAY\_FLAG\_ACCEPT\_FIRST\_HIT\_AND\_END\_SEARCH, RAY\_FLAG\_SKIP\_CLOSEST\_HIT\_SHADER setzen, um unnötige Beleuchtungsberechnungen und weitere Schnittpunktberechnungen zu umgehen und mit einem Bit als payload die Sichtbarkeit zur Lichtquelle mitzugeben. Mit diesem beispielhaften payload können wir die Farbe akkumulieren, unsere Anfangswerte 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 ausschließlich aus 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 5** 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 Algorithmus 6 wird nochmal vereinfacht die Funktionsweise eines Path Tracers erläutert, wobei die entsprechenden programmierbaren Shader von DirectX im jeweiligen Codeabschnitt markiert sind.

---

**Algorithm 6** 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

```

---

#### 2.4.4 Temporale Lösungsansätze

Das Problem der globalen Beleuchtung durch physikalisch basierter Monte-Carlo Integration und gleichzeitiges Erreichen der Echtzeitanforderung von  $30 \frac{\text{Bilder}}{\text{s}}$  ist ein bekanntes Problem. Dazugehörige bekannte Lösungsansätze ziehen bereits temporale Lösungsansätze z.B. temporales Akkumulieren [SKW<sup>+</sup>17] in Betracht.

Eine klassisches Formulierung [UE414] haben wir beispielhaft wie im Folgenden angewandt:

---

**Algorithm 7** Beispielhafte Akkumulation

---

```

1: Texture2D current_frame;
2: RWTexture2D accumulation_buffer;
3: float4 current_color = current_frame[pixel_pos];
4: float4 prev_color = accumulation_buffer[pixel_pos];
5: accumulation_buffer[pixel_pos] =
6: (frame_count * prev_color + current_color) / (frame_count + 1);

```

---

Diese klassische Formulierung verletzt unsere Annahme für die Quantilfunktion 2.17 in den A Posteriori-Bedingungen des zugrundeliegenden Algorithmus. Denn durch diese Akkumulation bestimmt nicht mehr allein der Anfangswert die Pixelfarbe!

## 2.5 A Posteriori

Um das in Abschnitt 2.7 vorgestellte *Dither Sampling* zu realisieren benutzen wir diese im Folgenden vorgestellten „nachträglichen“ Annahmen. A Posteriori sind Sie in dem Sinne, als das wir die Annahmen szenenabhängig machen und Sie anhand von bereits erstellten Pixelwerten formulieren. Damit werden Sie unabhängig vom Integranden der Formel 2.19.

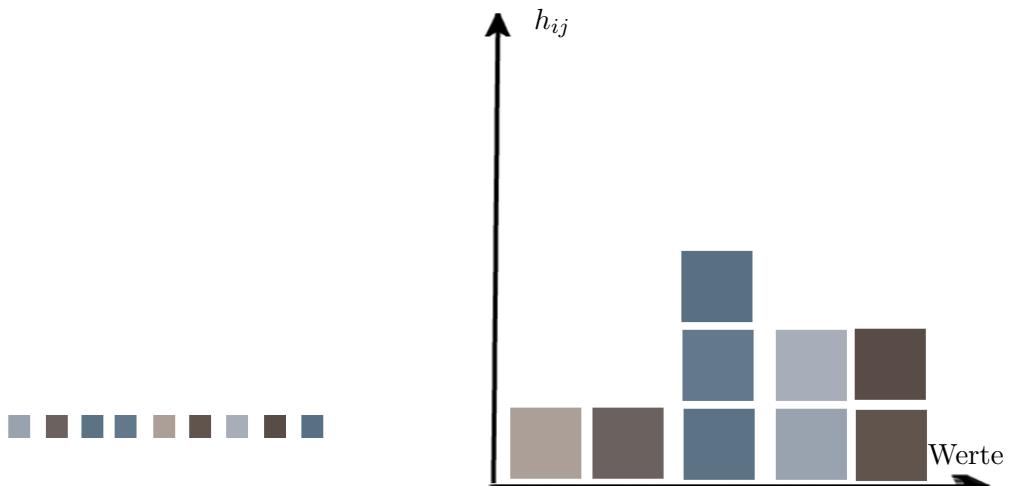
Im Kapitel über den Path Tracer haben wir gesehen, dass wir den Wert eines Pixels ( $i,j$ ) klassischerweise mit einem zufälligen Startwert durch eine Monte-Carlo Integration erhalten. Wir betrachten im Folgenden eine (theoretische) Menge aller möglichen Pixelwerte, welche durch alle möglichen Startwerte generiert wurde. In Abbildung 2.16 ist die Wahrscheinlichkeitsdichtefunktion  $h_{ij}$ , als eine Funktion über alle möglichen Werte  $I_{ij}$  eines Pixels ( $i,j$ ) aufgetragen.

$$H_{ij}([I_{Anfang}, I_{Ende}]) = \int_{I_{Anfang}}^{I_{Ende}} h_{ij} dI \quad (2.16)$$

Verfolgt man beispielhaft die Werte eines Pixels über neun Bilder bei unserem Path Tracer, so ergibt es sich zur Ansichtigkeit wie folgt:



(a) Szenenausschnitt



(b) Werte des Pixels im zeitlichen Verlauf (grüner Pfeil)

(c) Histogramm der Pixelschätzungen

Abbildung 2.16: Pixelwerte (grüne Markierung) in aufeinanderfolgenden Zeitschritten

Betrachten wir im Folgenden allerdings die theoretische, zur Echtzeit nicht umsetzbare

Menge aller möglichen Werte. Mit dieser Menge haben wir nun ein vollständiges Histogramm. Mit diesem Histogramm haben wir eine Wahrscheinlichkeitsfunktion der Pixelwerte  $I_{ij}$  an Stelle (i,j). Dies bedeutet wiederrum, dass das Erzeugen eines Pixelwertes nichts anderes bedeutet, als eine zufällige Wahl anhand der impliziten Wahrscheinlichkeitsdichtefunktion. Wir können folglich einem zufälligen Anfangswert einen konkreten Pixelwert zuteilen und eine umkehrbare Funktion definieren 2.17.

In Gleichung 2.17 lässt sich die Quantilfunktion  $H_{ij}^{-1}(x)$  erkennen. Sie verdeutlicht die Zuordnung eines Anfangswertes zu einem konkreten daraus entstandenen Wert eines erzeugten Pixels.

$$I_{ij} = H_{ij}^{-1}(x), x \in [0, 1] \quad (2.17)$$

**Fazit:** Betrachte nun blue noise verteilte Zahlenfolge  $d_{ij}$ , die zum Erzeugen eines Pixelwertes herangezogen wird (wir erhalten eine solche Zahlenfolge für jeden Pixel mit gekachelter Blue Noise) Textur über das Bild). Unsere Quantilfunktion  $H_{ij}^{-1}(x)$  ist monoton (da sie die Inverse einer Wahrscheinlichkeitsfunktion ist). Monotone Funktionen erhalten die Verteilung ihres Integranden (siehe vorherige Arbeiten zu „blue noise dithering sampling“ [EH19, Seite 3] [GF16]). Damit sind nun auch die daraus entstehenden Pixel wie eine Blue Noise verteilt!

Anmerkung: Da dies nur theoretisch möglich und gerade für Echtzeitanwendungen nicht umsetzbar ist, folgt eine praktikable Formulierung!

### 2.5.1 Praktische Durchführung

Die Berechnung des vollständigen Histogramms für jeden Pixel ist für eine Echtzeitanwendung zu kostenintensiv. Stattdessen könnte man auch die dadurch beanspruchte Rechenleistung auf z.B. mehrere Samples pro Pixel verteilen und dadurch eine Steigerung der Bildqualität erreichen! Stattdessen werden wir in dem temporalen Algorithmus das Histogramm mit dem vorherigen Bild approximieren. Die Approximation des Histogramms erfolgt dadurch mit dem  $Frame_t$  für  $Frame_{t+1}$ . Daher ist eine getroffene Annahme, um die gute Funktionalität des Algorithmus zu garantieren, eine nicht zu schnelle Bewegung der Kamera (siehe Abschnitt 7 um hier eine Verbesserung zu erzielen). Wir werden zur praktischen Durchführbarkeit die Anzahl an Pixelschätzung auf eine feste Zahl reduzieren (weitere Untersuchungen dazu siehe Abschnitt 3.1.0.1). Die Pixelwerte eines Pixels (i,j) werden wir in jedem Schritt durch seine benachbarten Pixelwerte approximieren. Diese Approximation macht bei kohärenten Bildbereichen Sinn. Diese Voraussetzung passt allerdings sowieso sehr gut zu der im Abschnitt 2.7 besprochenen nötigen Bildkohärenz um ein gutes Resultat im Dithering zu erreichen. Um die parallele Ausführbarkeit weiterhin zu steigern, werden wir außerdem dieses berechnete Histogramm eines Pixels für einen ganzen Block benutzen!

**Fazit:** Wir teilen das Bild in Blöcken auf, berechnen pro Block ein Histogramm und verwenden es als Schätzung für jeden Pixel.

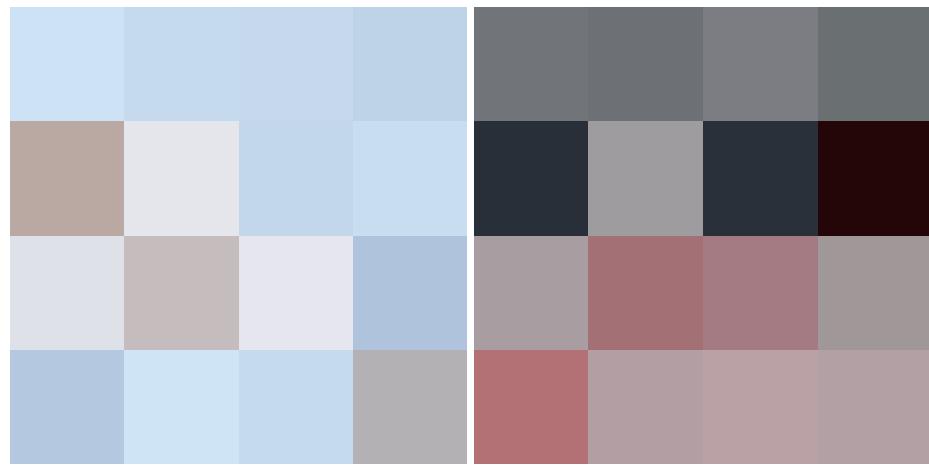


Abbildung 2.17: Pixelblöcke bei (in-)homogenen Flächen

In einer Gegenüberstellung eines (in-)homogenen Pixelblocks lässt sich die Notwendigkeit der Bildkohärenz erkennen. In Abbildung 2.17b sind die benachbarten Pixel eine gute Approximation des jeweiligen Pixels, wohingegen in Abbildung 2.17a die benachbarten Pixel dies nicht sind.

## 2.6 Blue Noise

Eine Ansammlung von Pixeln, deren Verteilung über den Raum einer blue noise entspricht, weist eine Reihe von Eigenschaften zur Steigerung der visuellen Qualität des Bildes auf [Uli88]

- Uniformität
- Isotropie
- Kachelung
- Niedrige Frequenzen

Im Folgenden wollen wir uns diese Eigenschaften genauer anschauen. Hierfür verwenden wir die in [Gam17] vorgestellten Texturen, welche anhand der *Void and Cluster*-Methode(siehe [Uli93]) erstellt wurden.

### 2.6.1 Eigenschaften

Um die Eigenschaften der Texturen zu untersuchen, wurden korrespondierende Spektren zu den Texturen mit Hilfe von [JCr18] erstellt und miteingebunden.

#### 2.6.1.1 Uniformität

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

$$P(n \leq p) = p \quad (2.18)$$

Die Uniformität(lat. *uniformitas*-Einförmigkeit) garantiert uns dieses Verhalten  $\forall p \in [0, 1]$ . Man könnte auch sagen, dass jeder Grauwert gleichwahrscheinlich auftreten soll. Die zugehörige konstante Wahrscheinlichkeitsdichte lässt sich einfach zur Echtzeit umsetzen mit Hilfe von (pseudo-)zufälligen Zahlen. Mit der erstellten(siehe [Whi19]) white noise Textur,

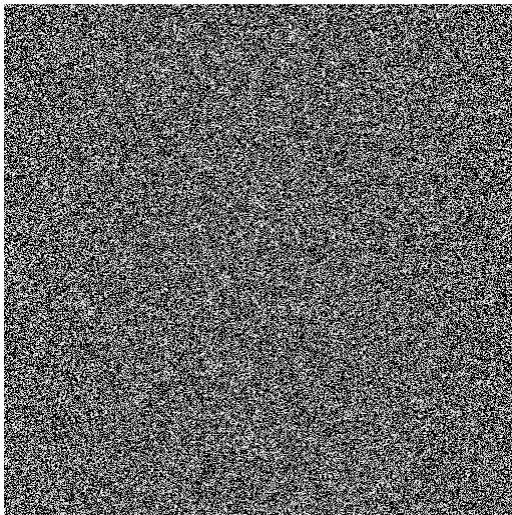


Abbildung 2.18: white noise Textur

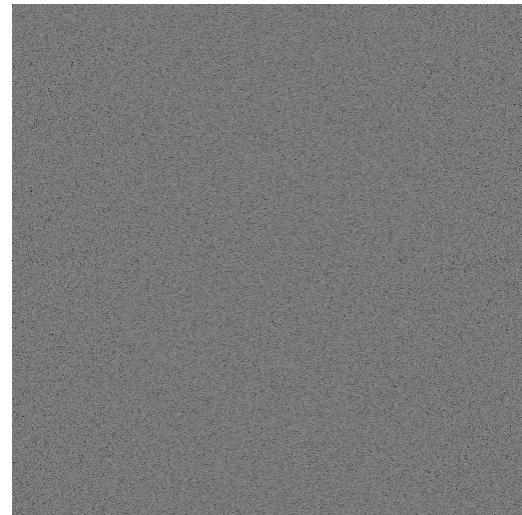


Abbildung 2.19: Amplitudenspektrum

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

### 2.6.1.2 Niedrige Frequenzen

Niedrige Frequenzen sind in einer blue noise sehr wenig bis gar nicht vertreten. Dies macht sich sowohl in der Zeitdomäne erkenntlich (Abbildung 2.20): keine erkennbaren gleichfarbigen Pixelverbünde innerhalb der Textur als auch in der Frequenzdomäne (Abbildung 2.21): der schwarze Ring innerhalb der Amplitudendichte deutet auf das Fehlen von niedrigen Frequenzen und dem hohen Unterschied benachbarter Pixel hin. Außerdem haben wir aus der vorherigen Eigenschaft der Uniformität gesehen: Wir wollen, dass alle Grauwerte gleichwahrscheinlich auftreten. Dies können wir am besten im Frequenzspektrum (siehe 2.21) beobachten. Hier sind alle hohen Frequenzen wie beim weißen Rauschen 2.19 gleich stark vertreten.

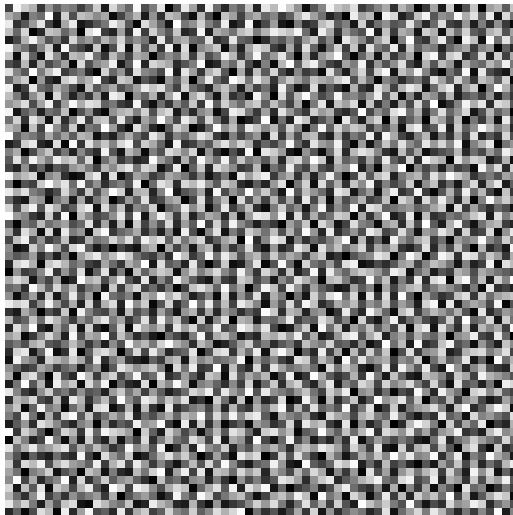


Abbildung 2.20:  $512^2$  blue noise Textur

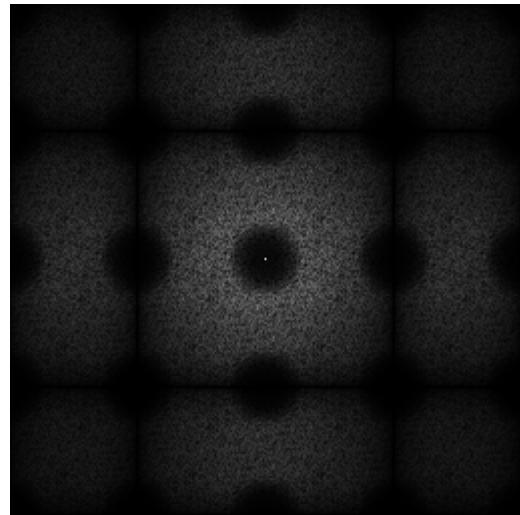


Abbildung 2.21: Fourier Spektrum  $512^2$  blue noise Textur

Auch zu erkennen ist die weitere Eigenschaft der Isotropie.

### 2.6.1.3 Isotropie

Die Isotropie (altgr. *isos*-gleich und *tropos*-Richtung) einer blue noise Textur bietet eine weitere wichtige Eigenschaft: in alle Richtungen eine gleiche Verteilung. Dabei haben wir in allen Dimensionen die Unabhängigkeit einer Eigenschaft. Um uns dies an einem Gegenbeispiel klar zu machen, schauen wir uns das Bayer-Pattern an. Dieses Pattern erfüllt sowohl die Eigenschaft der Niedrigen Frequenz (siehe Abschnitt 2.6.1.2) als auch die der Uniformität, jedoch nicht Jene der Isotropie. Zu erkennen ist dies an den sich wiederholenden Strukturen in der Abbildung 2.22.

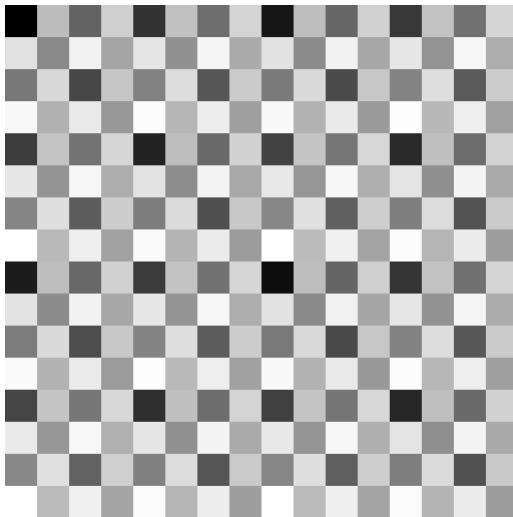


Abbildung 2.22:  $512^2$  bayer pattern Textur

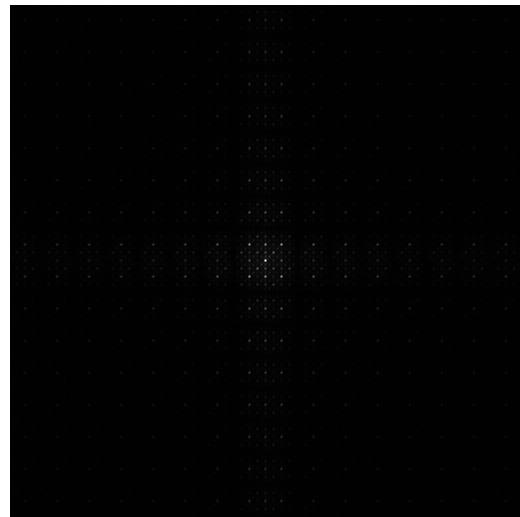


Abbildung 2.23: Amplitudendichte  $512^2$  bayer pattern Textur

In der Frequenzdomäne (Abbildung 2.23) ist zu erkennen, dass die Amplitudendichte in einzelnen Punkten organisiert ist. Diese lassen sich durch die vorhandenen Richtungen der Textur erklären. Speziell in zwei Richtungen ist eine sich wiederholende Pixelsequenz zu erkennen.

#### 2.6.1.4 Kachelung

Im Gegensatz zu dem bayer pattern oder der white noise, die sich einfach zur Echtzeit berechnen lassen, sind blue noise verteilte Texturen im Erstellungsaufwand, der mit Anzahl der Dimensionen und Größe der Textur schnell anwächst, deutlich höher (siehe auch [Pet16]). Es empfiehlt sich daher für den temporalen Algorithmus in Kapitel 3 eine kleinere Textur zu verwenden. Dies hat außerdem den Vorteil, eine bessere Ausnutzung des kleinen aber schnellen Cachespeichers zu gewährleisten. Aufgrund des Aufbaus von aktueller Grafikhardware (siehe [Kra18]), wollen wir diese Textur soweit oben wie möglich in der Cachehierarchie halten.(L1 96KByte, L2 6MByte).

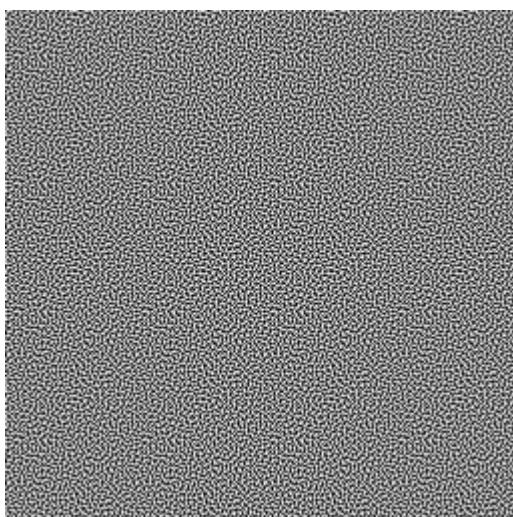


Abbildung 2.24:  $512^2$  gekachelte Textur von  $64^2$

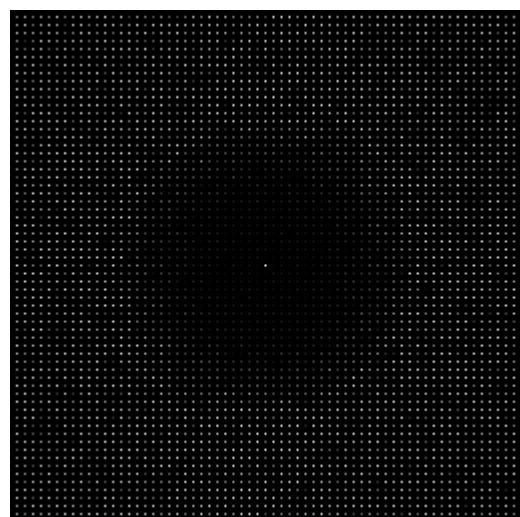


Abbildung 2.25: Fourier Spektrum

Betrachtet man die Abbildung 2.6.1.4 lässt sich der blue noise Charakter anhand des wenigen niederen Frequenzanteils erkennen. Wiederholungen sind in der Zeitdomäne schwerer zu erkennen. Obwohl Sie in der Frequenzdomäne erkennbar sind und unsere Isotropie (teilweise) aufheben. Um dieses Problem zu umgehen, werden wir mit Hilfe von Quasi-Zufallsfolgen auf diese Textur zugreifen und die Isotropie (teilweise) wiederherstellen. Dafür können wir hier folgendes Fazit ziehen: Regelmäßige Kachelung einer kleineren Blue Noise Textur über das gesamte Bild liefert eine gute Blue Noise Verteilung bei vertretbaren Rechenaufwand und guter Cacheausnutzung! Mit einer Größe von 16,1KB und der korrespondierenden Retarget-Textur (siehe Abschnitt 2.2 über die Erstellung dieser Textur) Größe von 12,1KB passen Beide in den 96KB großen L1 Cache [Kra18]. Wohingegen eine korrespondierende 512<sup>2</sup>-Textur mit einer Größe von 1MB nur in L2-Cache passt.

## 2.7 Dithering Sampling

Nachdem wir die Eigenschaften der Blue Noise kennengelernt haben, können wir zusammen mit dem Verständnis über den Path Tracer und der zugrundeliegenden Monte-Carlo Integration (siehe Gleichung 2.15) das „dithered sampling“ verstehen. *Dithering* ist das bewusste Einbringen eines Rauschens, um entstehende Quantisierungsfehler zu randomisieren. Klassischerweise wird eine zweidimensionale Blue Noise Textur verwendet, um mit einer darauf aufbauenden Schwellenwertbildung dieses Rauschen in ein Bild zu bringen. Hier wollen wir durch Dithering die Verteilung des entstehenden Monte-Carlo Integrationsfehlers (Rauschen) verändern.

Grundlage des n-dimensionalen Path Tracers werden sowohl eine Blue Noise-Verteilung als auch Anfangswerte  $s_n$  mit d-Dimensionen. Damit konkretisiert sich die Monte-Carlo Integration (siehe Gleichung 2.15) mit Integrand f zu folgender Gleichung:

$$\frac{1}{N} \sum_{n=0}^{N-1} f(s_n) \quad (2.19)$$

Mit dem Ziel, eine blue noise Fehlerverteilung im Bildraum zu erreichen, hat sich bereits die Arbeit [GF16] beschäftigt. Mit im Vorraus („a priori“) blue noise korrelierten Zahlenfolgen  $s_n$  hoffte man ebenso korrelierte Pixelwerte nach der Integration zu erhalten. Der temporale Algorithmus im Kapitel 3 führt A Posteriori Formulierungen (arbeitet auf bereits erzeugt Bilddaten) und mit ihr eine inverse Funktion 2.17, welche (approximative) garantierte korrelationerhaltene Integranden liefert! Neben der Eigenschaft der Verteilungsbeibehaltung der Integranden zeigt die frühere Arbeit von [EH19, Seite 3] eine weitere wichtige Voraussetzung des blue noise sampling: Die Bildraumkohärenz. Wie auch beim konventionellen Prozess des Dithering so sollten für ein gutes Ergebnis zwei benachbarte Pixel einen ähnlichen Wert aufweisen [Uli88].

## 2.8 Render Graph

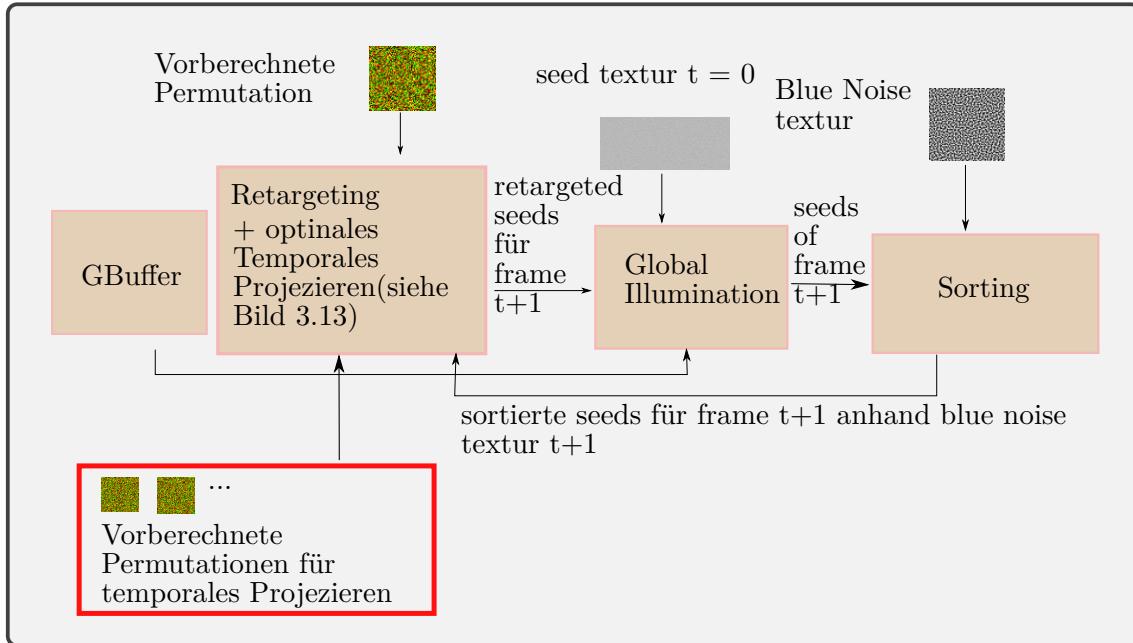


Abbildung 2.26: Render Graph

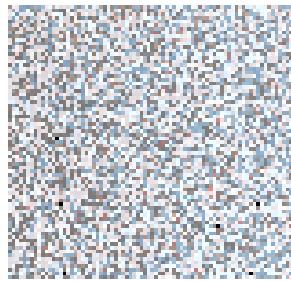
Für jeden Bilderzeugungsvorgang werden die in Abbildung 2.26 angegebenen Schritte von links nach rechts durchlaufen. Zu Beginn wird die Berechnung eines *GBuffers* vorgenommen und berechnete Normalen, Positionen etc. an die Beleuchtungsstufe weitergegeben. Der darauffolgende Retarget-Schritt permultiert die Anfangswerte anhand einer vorberechneten Textur, die sortiert sind wie  $BlueNoise_t$ , zu einer Verteilung von  $BlueNoise_{t+1}$ . Bei aktivierter temporaler Projizierung (siehe 3.3) werden Kamerabewegungen für die Permutation der Anfangswerte berücksichtigt. Die so umsortierten Anfangswerte werden an den Path Tracer übergeben. Das darauffolgend synthetisierte Bild wird in der Sortierphase zusammen mit einer vorberechneten Blue Noise Textur verwendet um die Anfangswerte zu einer Blue Noise Verteilung zu sortieren. Diese sortierten Anfangswerte bilden im nächsten Durchlauf die Eingabe des Permutierens zur nächsten Blue Noise Textur.

### 3. Temporaler Algorithmus

Mit Hilfe des zuvor in Kapitel 2 erworbenen Wissens stellen wir in diesem Kapitel einen temporalen Algorithmus, basierend auf der Arbeit von [EH19], vor. Dieser erreicht eine zeitlich stabile Blue Noise Fehlerverteilung innerhalb weniger Bilder im Bildraum. Dabei wird in dieser Arbeit im Speziellen auf den Einsatz innerhalb Echtzeitanwendungen eingegangen. Der Algorithmus arbeitet mit drei separaten Schritten: In einem ersten Schritt sortiert 3.1 der Algorithmus anhand unserer getroffenen A Posteriori Annahmen die Anfangswerte unseres Path Tracer derart um, sodass die im nächsten Bilderzeugungsschritt entstehenden Pixel Blue Noise verteilt sind. Die Notwendigkeit des nächsten Schrittes, das Retargeting, ergibt sich aus der Tatsache, dass wir in jedem Bilderzeugungsschritt eine neue blue noise Textur verwenden (theoretisch, praktisch benutzen wir Quasi-Zufallsfolgen). Ohne die angewandte Permutation innerhalb des Schrittes würde sich durch die Randomisierung die jeweilige gewonnene blue noise Verteilung nicht auf den nächsten Bilderzeugungsprozess übertragen. Der Algorithmus, bestehend aus zwei grundsätzlichen Schritten, das Umsortieren und Permutieren, verlangt folgende Vorbedingung: benachbarte Pixel müssen annähernd einen ähnlichen Wert haben (siehe Abschnitt 2.7 und Abschnitt 2.5). 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. Aufgrund dieses Problems nehmen wir eine Idee zur Verbesserung des Algorithmus auf (siehe [EH19]) und führen damit einen neuen temporalen Projektionsschritt in Abschnitt 7 ein. Damit wird sich die zeitliche Stabilität erhöhen.



(a) Szene



(b) Szenenausschnitt



(c) Fouriertransformierte des Ausschnitts

Abbildung 3.1: Ausgangssituation, erstes erzeugte Bild

In Abbildung 3.1 sehen wir die erste Ausgabe des Path Tracer aufgrund zufälliger Anfangswerte (erzeugt mit Mersenne-Twister). Dies ist der Startzustand für unseren Algorithmus. Wie bereits in Abschnitt 2.6 ausführlich besprochen, lassen sich anhand der Szene, des Szenenausschnitts und dem korrespondierenden Spektrum die typischen Eigenschaften einer white noise ablesen (Clusterbildung im Zeitbereich, gleichmäßige Amplitudendichte im Frequenzbereich). Nehmen wir diese Ausgabe des Path Tracers, so lassen sich mit den erzeugten Pixeln nachträgliche Annahmen (siehe Abschnitt 2.5) für ihn formulieren.

### 3.1 Sorting

In diesem Abschnitt machen wir uns die getroffenen A Posteriori Annahmen zu Nutze. Nach dem Erzeugen von  $Bild_t$  (und vor dem Erzeugen von  $Bild_{t+1}$ ) nehmen wir die Sortierung anhand der Pixelwerte von  $Bild_t$  vor.

Die Pixel innerhalb eines Blocks (z.B. Blockgröße = 64, Auswirkung von verschiedenen Blockgrößen in Abbildung 3.2) werden anhand ihrer Intensität in einer aufsteigenden Liste sortiert und damit zu einem Histogramm. Dieses generierte Histogramm wird für jeden Pixel innerhalb des Blocks verwendet. Die sortierte Liste als Histogramm zu benutzen ist zulässig. Denn beim Benutzen der inversen Funktion 2.17 machen wir nichts Anderes als zufällige Zahlen auf ein bestimmtes Wahrscheinlichkeitsquantil, das einer gewissen Sortierung entspricht, abzubilden. Deshalb können wir die Pixelintensitäten innerhalb eines Blocks anhand ihrer Indizes (entspricht den Wahrscheinlichkeitsquantilen bei der Wahrscheinlichkeitsfunktion) sortieren!

---

#### Algorithm 8 Sortier Schritt t

---

```

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: //sort the two lists by means of intensities
8: sort_by_means_of_value(framePixelsIntensities);
9: sort_by_means_of_value(noiseIntensities);
10:
11: //corresponding indices are also sorted by means of values
12: for i = 1..BLOCKSIZE do
13:   sortedSeeds(noiseIntensities.getIndex(i)) =
14:     incomingSeeds(framePixelIntensities.getIndex(i));
15: end for
16: //now the seeds are sorted following a blue noise distribution

```

---

Das Verwenden der anhand Pixel- und blue noise Werten sortierten Indizes in Zeile 13,14 des Algorithmus 8 ist die praktische Anwendung der zuvor eingeführten Quantilfunktion 2.17. Denn die Anfangswerte  $x \in [0, 1]$  der Quantilfunktion 2.17 werden hier einem eindeutigen sortierten Intensitätswert zugeteilt. Das Abbilden der Pixelwerte auf Werte einer Blue Noise Textur garantiert nun die entsprechende Korrelation der Anfangswerte zueinander (siehe auch Abschnitt 2.7). Die Annahme der Lokalität, also der Homogenität (in der Abbildung 2.17 demonstriert) einer Fläche innerhalb eines Blocks, erlaubt die Approximation des Histogramms 2.16 eines Pixels anhand der Intensitäten aller anderen Pixel innerhalb des Blocks. Wird der Block zu groß gewählt, so wird die Approximation zu rechenintensiv, die parallele Ausführbarkeit und die Homogenitätseigenschaft (Bildkohärenz) gehen verloren, wohingegen zu kleine Blockgrößen nur eine sehr vage und ungenaue Approximation des Histogramms 2.16c liefern.

Hierbei muss noch eine wichtige Anmerkung gemacht werden. Die Fehlerverteilung der Pixelwerte im Bildraum konvergiert auf diese Weise nicht perfekt zu einer blue noise Verteilung, denn wir wechseln in jedem Bild die verwendeten blue noise Texturen (theoretisch, praktischerweise werden wir hier eine Textur verwenden und mit Erkenntnissen aus Abschnitt 2.1.3 quasi-zufällig zugreifen um einen solchen Effekt zu erreichen).

### 3.1.0.1 Blockgröße

Für den Sortierschritt gibt es ein Abwägung zu treffen: Hohe Blockgröße  $B$  bedeutet eine bessere Approximation des Histogramms (siehe Bild 2.16c). Denn man erhöht die Anzahl an Pixelschätzungen für jeden einzelnen Pixel. Allerdings geht im Gegenzug die Raum-Zeit Kohärenz verloren. Der Algorithmus wird anfälliger für Szenen, die keine größeren homogenen Flächen aufweisen.

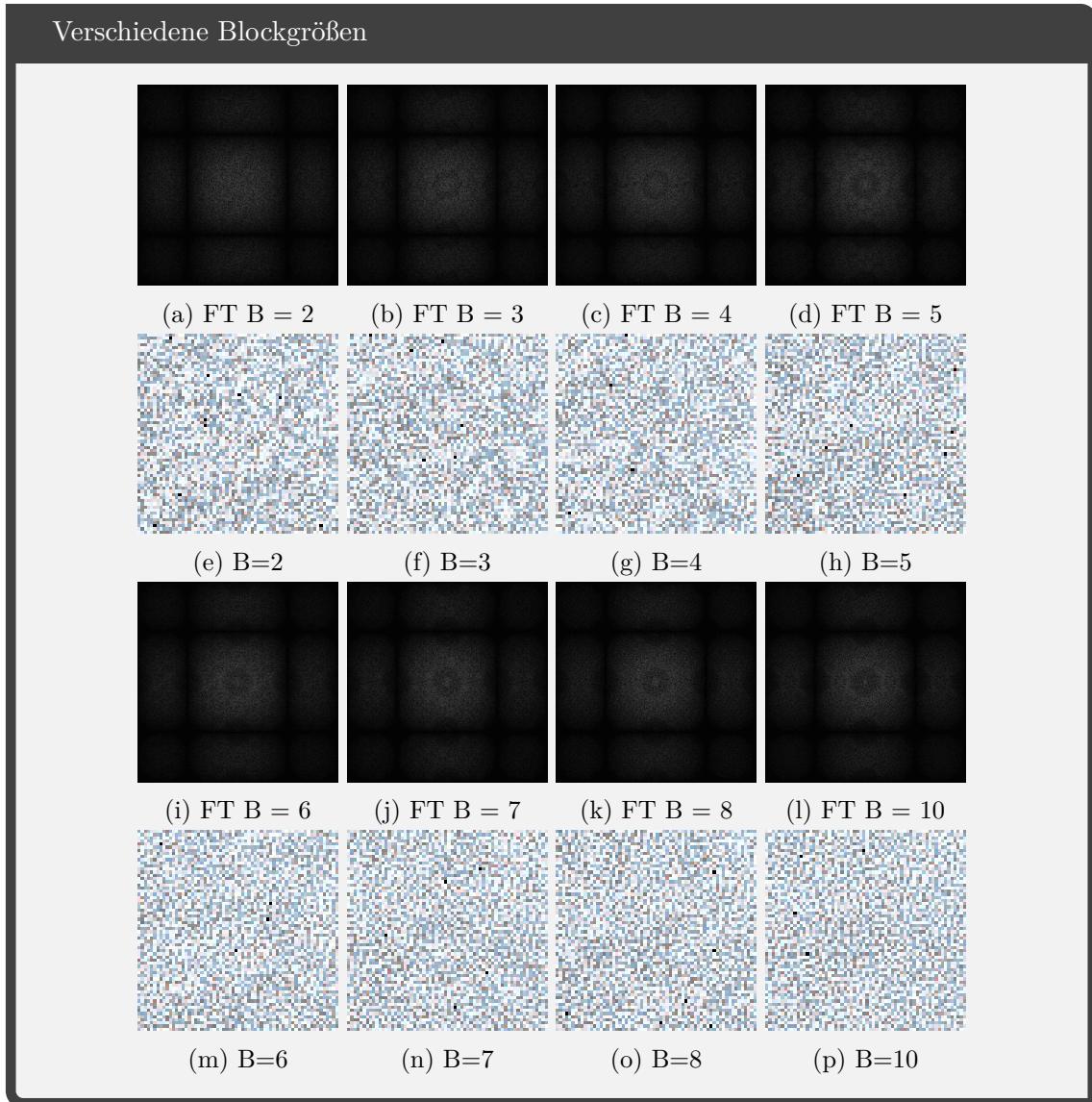


Abbildung 3.2: Verschiedene Blockgrößen  $B$  bei selben homogenen Szenausschnitt; nur Sortierschritt

Generell lässt sich aus den Untersuchungen in Abbildung 3.2 schließen, dass eine geringe Blockgröße des Sortierschrittes ein Fehlen bzw. eine starke Abschwächung der Blue Noise Fehlerverteilung im Bildraum zur Folge hat. Erklärung: Das Histogramm (siehe Abbildung 2.16c), Wahrscheinlichkeitsfunktion aller möglichen Pixelwerte, wird mit zu wenigen Werten approximiert. Diese sehr vage Approximation hat zur Folge, dass die Sortierung einer randomisierten Folge entspricht. Deshalb sind im Bildraum viele Cluster und die white noise Eigenschaften in den Spektren zu erkennen. Unsere Untersuchungen ergeben, dass eine Blockgröße  $B < 4$  vermieden werden sollte. Ab einer Blockgröße  $B = 4$  hat man bereits eine rechengünstige benutzbare Approximation. Eine gute Abschätzung ergibt sich

bei uns mit einer Blockgröße von  $B \approx 8/10$ . Mit dieser Blockgröße hat man immernoch eine gute, effiziente, parallele Ausführbarkeit garantiert und liefert eine generelle, gute Approximation der Pixelwerte.

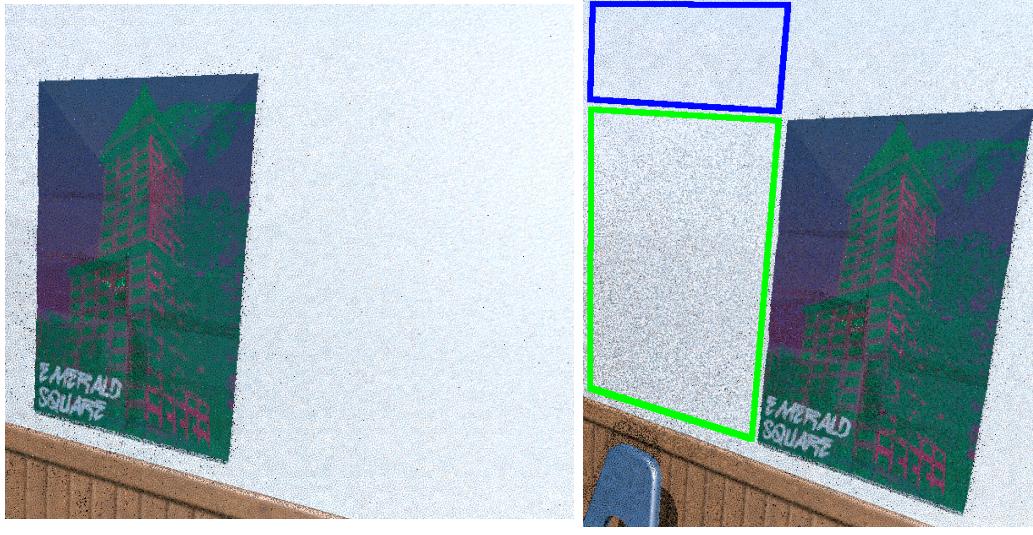


Abbildung 3.3: Wir sortieren unsere Anfangswerte und erhalten durch die Pixelwertschätzungen in der Umgebung eine gute Approximation im ersten Bild 3.3a; Verschieben wir die Anfangswertetextur ohne weiter zu sortieren, so werden nun Anfangswerte, die anhand der Pixelwertschätzung des Posters verwendet wurden, nun zur Pixelfarbgabeung für die Wand verwendet (grünes Rechteck). Man kann sehen, dass hier keine blue noise Eigenschaft mehr zu erkennen ist, da die Pixelwertschätzungen des Posters nicht auf die Wand übertragbar sind! Hingegen kann man im blauen Rechteck eine blue noise Eigenschaft erkennen, da die verwendeten Anfangswerte anhand einer ähnlich farbigen Oberfläche sortiert wurden.



Abbildung 3.4: Szene vor eingeschalteten Sortieren, Ausschnitt(grüner Kasten wird über mehrere Bilder nach Aktivierung des Sortierens verfolgt)

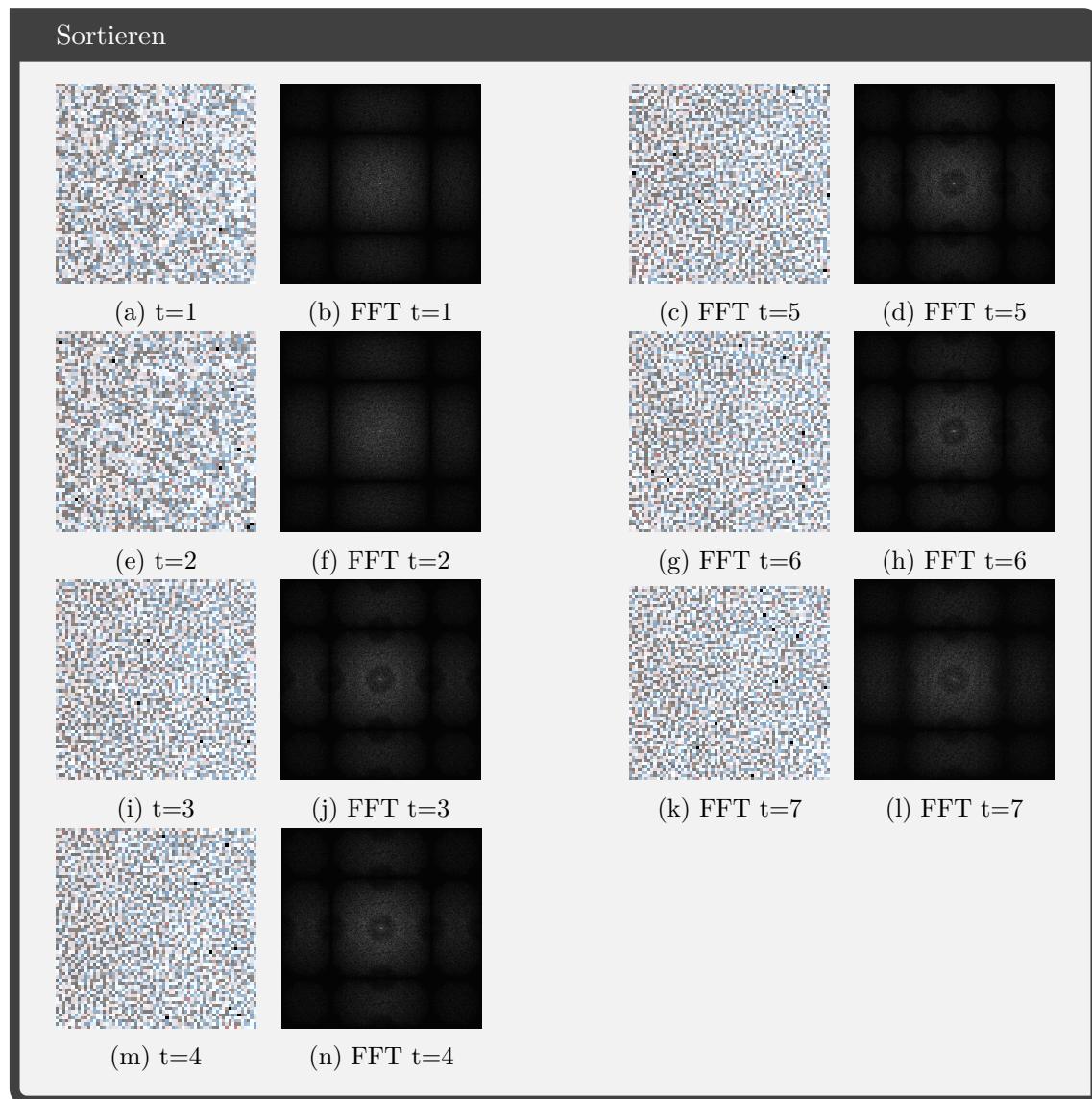


Abbildung 3.5: Sorting ab t=1 aktiviert



Abbildung 3.6: Szene

Die Bildreihe zeigt die ersten sieben erstellten Bilder mit ausschließlichem Sortieren( $B=8$ ) ohne Retargeting.

- t=1-2 Die ersten beiden erzeugten Bilder lassen keine blue noise Eigenschaften erkennen. Das typische weiße Rauschen ist zu erkennen und hat einen unbefriedigenden optischen Eindruck.
- t=3 Ab dem dritten Bild können wir eine Blue Noise Eigenschaft im Bild durch ein bloßes Sortieren erkennen. Die daraus resultierende Steigerung der optisch wahrnehmbaren Qualität, wie bereits in Arbeiten wie [Uli88] besprochen, lässt sich gut erkennen.
- t=4-7 Der blue noise Charakter kann auch über die darauffolgenden Bilder erhalten werden. Allerdings ist der Charakter noch nicht stark genug ausgeprägt. Hier lässt sich der zuvor beschriebene Effekt erkennen: Durch die Benutzung einer neuen blue noise Textur in jedem Bild akkumulieren Verbesserungen nicht so gut und das Spektrum hat einen mehr verschwommenen Charakter ohne hohen Kontrast.

Diese Erkenntnisse veranlassen uns einen zusätzlichen Schritt nach dem Sortieren (vor der erneuten Bilderzeugung) einzuführen: Das Retargeting.

### 3.2 Retargeting

Bevor die Anfangswerte zur globalen Beleuchtung benutzt werden (siehe auch Render Graph), durchlaufen sie nach dem Sortieren (siehe 3.1) das Retargeting. Der Sinn liegt hierbei beim Vertauschen der Anfangswerte, sodass Sie verteilt sind wie  $BlueNoise_{t+1}$ . Aufgrund dessen ergibt sich eine bessere Aufsummierung der Blue Noise Fehlerverteilungen über die ersten paar Bilder.

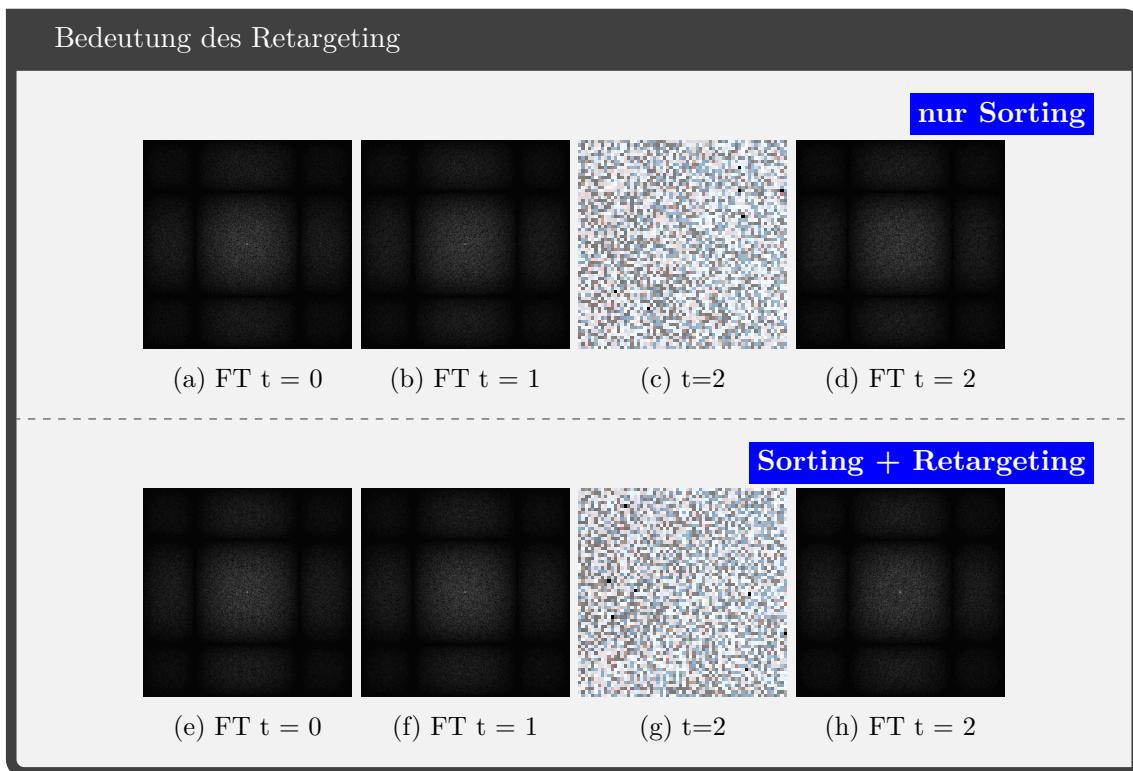


Abbildung 3.7: Vergleich erste Bilder Sorting( $B=2$ ) mit bzw. ohne Retargeting

Die Abbildung 3.7 verdeutlicht die Bedeutung des Retargeting Schrittes. Sie zeigt dieselben Ausschnitte eines homogenen Szenenausschnitts bzw. die korrespondierende Spektra über die ersten drei Bilder. Die erste Reihe mit den Bildern

- a.) - d.) zeigt auch nach dem dritten Bild keine Blue Noise Eigenschaften. Dies liegt an der geringen Blockgröße des Sortierschrittes (in Abschnitt 3.1.0.1 bereits besprochen). Das Histogramm, Wahrscheinlichkeitsfunktion aller möglichen Pixelwerte, wird mit nur vier Werten approximiert. Diese sehr vage Approximation hat zur Folge, dass die Sortierung einer randomisierten Folge entspricht. Deshalb sind im Bildraum (siehe Bild 3.7c) viele Cluster und die white noise Eigenschaften (siehe Bild 2.6.1.1 im Spektrum zu erkennen).
- e.) - h.) Benutzt man hingegen wie in diesen Bildern zu dem vagen Sortierschritt mit Blockgröße  $B=2$  noch den Retargetingschritt, so akkumulieren sich die Umverteilungen aus dem Sortierschritt besser. Trotz der geringen Blockgröße, die nicht reicht, damit nach einem Bild Blue Noise Eigenschaften auftreten und aufgrund der Nutzung einer neuen Textur (siehe auch Abschnitt 2.1.3) innerhalb eines jeden Bilderstellungsvorgangs wieder verschwimmen würden, können wir bereits im dritten Bild 2.6 Eigenschaften erkennen. Im Bildraum (Bild 3.7g) sind die Pixelverbünde verschwunden (=hohe Frequenzen). Dies macht sich im Frequenzraum (siehe Bild 3.7h) erkenntlich, indem niedrige Frequenzen weniger vertreten sind als bei dem bloßen Sortierschritt (Bild 3.7d)

**Anmerkung:** Das gewünschte Ergebnis stellt sich allerdings nur bei einem Sortierschritt mit ausreichender Blockgröße und anschließenden Retargeting ein (wie bei unserer Screenshotreihe).

An sich ist der Schritt eine bloße Anwendung einer Permutation wie im Algorithmus 9.

---

**Algorithm 9 Retargeting Schritt**

---

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

---

Da die Berechnung der Permutation „on-the-fly“ zu rechenintensiv ist, verwenden wir eine durch Simulated Annealing berechnete Textur. Wir greifen auf unsere beiden vorberechneten Texturen quasi-zufällig (siehe 2.1.3) zu. Dabei wird der Permutationswert auf die jeweilige Position des Anfangswertes addiert. Dabei kann ein jeder Pixel in einem Umfeld (Radius  $r = 6$ ) permutiert werden (siehe 3.8).

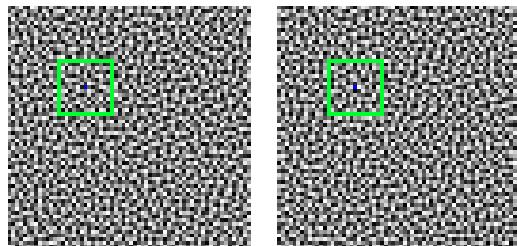


Abbildung 3.8: Permutation



Abbildung 3.9: Szene

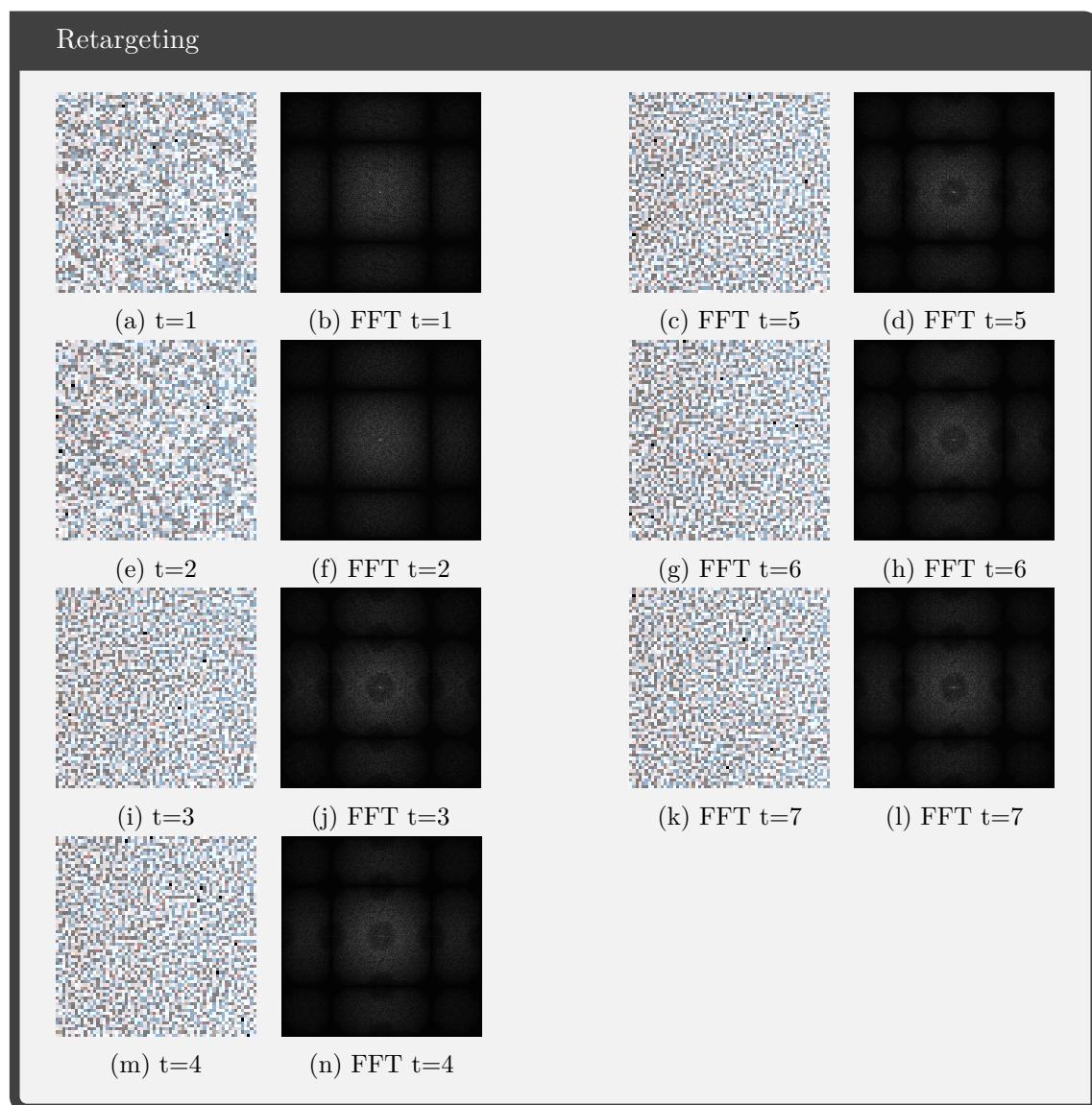


Abbildung 3.10: Retargeting ab t=1 aktiviert



Abbildung 3.11: Szene

$t=1-2$  Die ersten beiden erzeugten Bilder lassen keine blue noise Eigenschaften erkennen. Das typische weiße Rauschen ist zu erkennen und hat einen unbefriedigenden optischen Eindruck.

$t=3$  Ab dem dritten Bild können wir eine Blue Noise Eigenschaft im Bild durch ein bloßes Sortieren erkennen. Die daraus resultierende Steigerung der optisch wahrnehmbaren Qualität, wie bereits in Arbeiten wie [Uli88] besprochen, lässt sich gut erkennen.

$t=4-7$  In den darauffolgenden Schritten ist eine zeitlich stabilere blue noise Verteilung zu erkennen. Die Schwankungen zwischen den darauf folgenden Bildern wird geringer durch die vorherige Umsortierung zur nächsten blue noise Textur.

Zur Verdeutlichung der Bedeutung des zusätzlichen Retargeting zum vorherigen Sortierschritt:

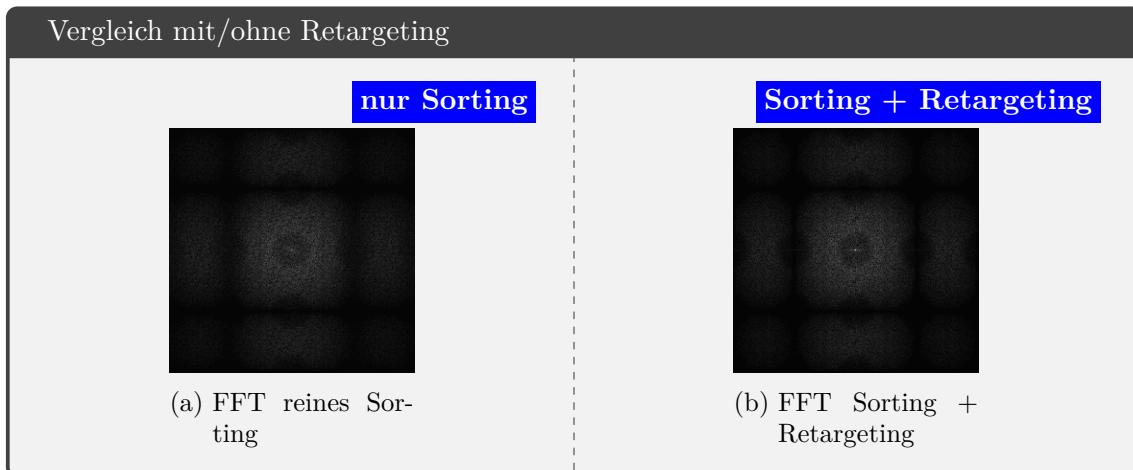


Abbildung 3.12: Vergleich des jeweilig siebten Bildes reines Sorting und zusätzliches Retargeting

Über die Zeit lässt sich der zusätzliche Permutationsschritt gut verdeutlichen. Das Spektrum des siebten gerenderten Bildes hat beim bloßen Sortieren (Bild 3.12a) keinen so starken Kontrast wie das Spektrum mit zusätzlichem Retargeting (Bild 3.12b). Die blue noise Umverteilungen akkumulieren nicht so gut und führen zu dem verwischten Charakter.

### 3.3 Temporales Projizieren

Bisherige temporale Lösungsansätze zur Steigerung der optisch wahrnehmbaren Qualität (siehe 2.4.4) lieferten gute Ergebnisse. Wir nehmen hier diesen Ansatz auf, um den Algorithmus noch zeitlich stabiler zu machen.

Die Arbeit [EH19, S.9/10] motiviert ein temporales Projizieren. Dabei ist zu beachten, dass um die A Posteriori Annahmen und die Vorbedingungen der Quantilfunktion 2.17 für unseren Algorithmus zu erfüllen, wir eine erneute Permutation brauchen! Denn durch die Permutation haben wir wieder garantiert, dass je ein Anfangswert  $x \in [0, 1]$  auf je ein Pixelfarbwert abgebildet wird. Da unser Algorithmus davon ausgeht, dass zwei aufeinanderfolgende Bilder gleiche Pixelwerte besitzen, erhoffen wir uns durch ein temporales Projizieren eine Verbesserung bei der Blue Noise Verteilung falls sich z.B. durch Kamerabewegung die Farbgebung der Pixel zwischen Ihnen ändert. Die temporale Projektion, welche wir hier anwenden, baut auf aktuelle verbreitete Techniken des TAA auf [INS18].

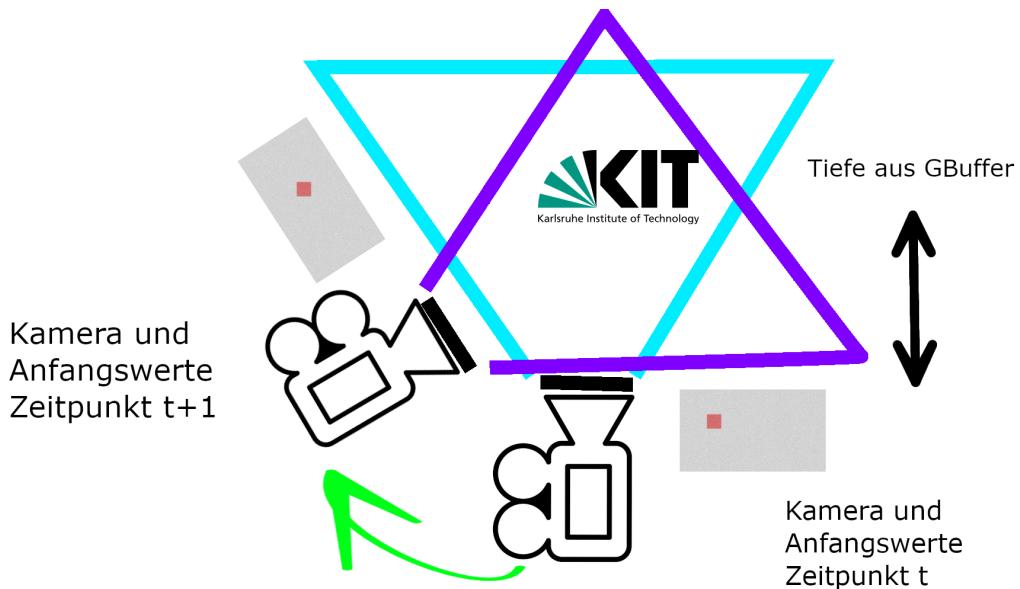


Abbildung 3.13: Übersicht temporales Projizieren

Wir benutzen die berechneten Tiefenwerte aus dem GBuffer (siehe auch unserer Render Graph) und die jeweiligen View-Projektion-Matrizen der Kameras, um herauszufinden, welche Koordinaten der jeweilige Anfangswert von Bild  $t$  im Bild  $t+1$  haben würde aufgrund der Drehung.

Wir berechnen nun anhand der aktuell berechneten projizierten Position den jeweiligen Verschiebungsvektor jedes einzelnen Pixels und damit der neuen Position des jeweiligen Anfangswertes. Eine anschließende Mittelung über das gesamt Bild verschafft uns einen durchschnittlichen Bewegungsvektor. Dieser fließt nun in die Auswahl einer extra berechneten Retarget-Textur, die diese Bewegung berücksichtigt. In einem zusätzlichen Vorberechnungsschritt wurden eine Vielzahl von Bewegungsvektoren auf die Blue Noise  $Textur_t$  angewandt, bevor die Permutation zur  $Textur_{t+1}$  berechnet und abgespeichert wurde. So mit haben wir eine Vielzahl von neuen Retarget-Texturen geschaffen, die eine jeweilige zweidimensionale Verschiebung berücksichtigen. Wir haben eine jeweilige zweidimensionale Verschiebung von je vier Pixeln in negativer sowie positiver Richtung berücksichtigt. Das ergibt in Summe 81 vorberechnete Texturen für diesen Schritt. In unseren Untersuchungen

hat sich auch bei schnellen, ruckartige Kamerabewegungen diese Anzahl an Texturen als ausreichend erwiesen.

Mit diesem Ansatz haben wir auch eine erneute Garantie einer Permutation, welche wir bereits bei dem Retargeting erreicht haben.

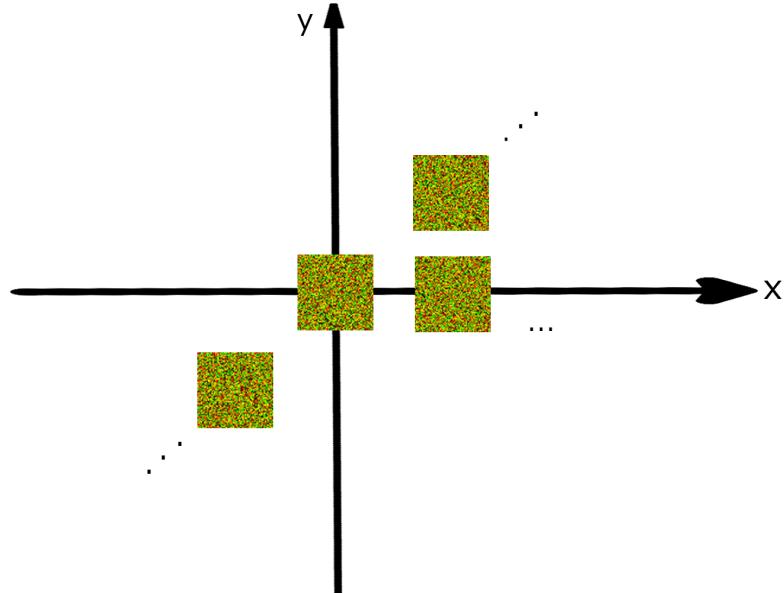


Abbildung 3.14: Für über das gesamte Bild gemittelte mögliche 2D-Verschiebungen eines Pixels zwischen zwei aufeinanderfolgenden Bildern je eine separate Retarget-Textur

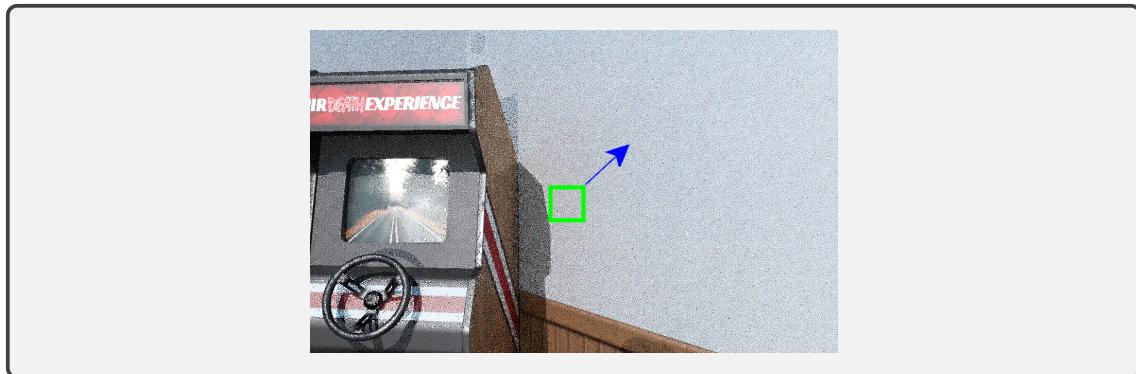


Abbildung 3.15: Ausschnitt(grüner Kasten) verfolgt bei Bewegungsvektor(blauer Pfeil)

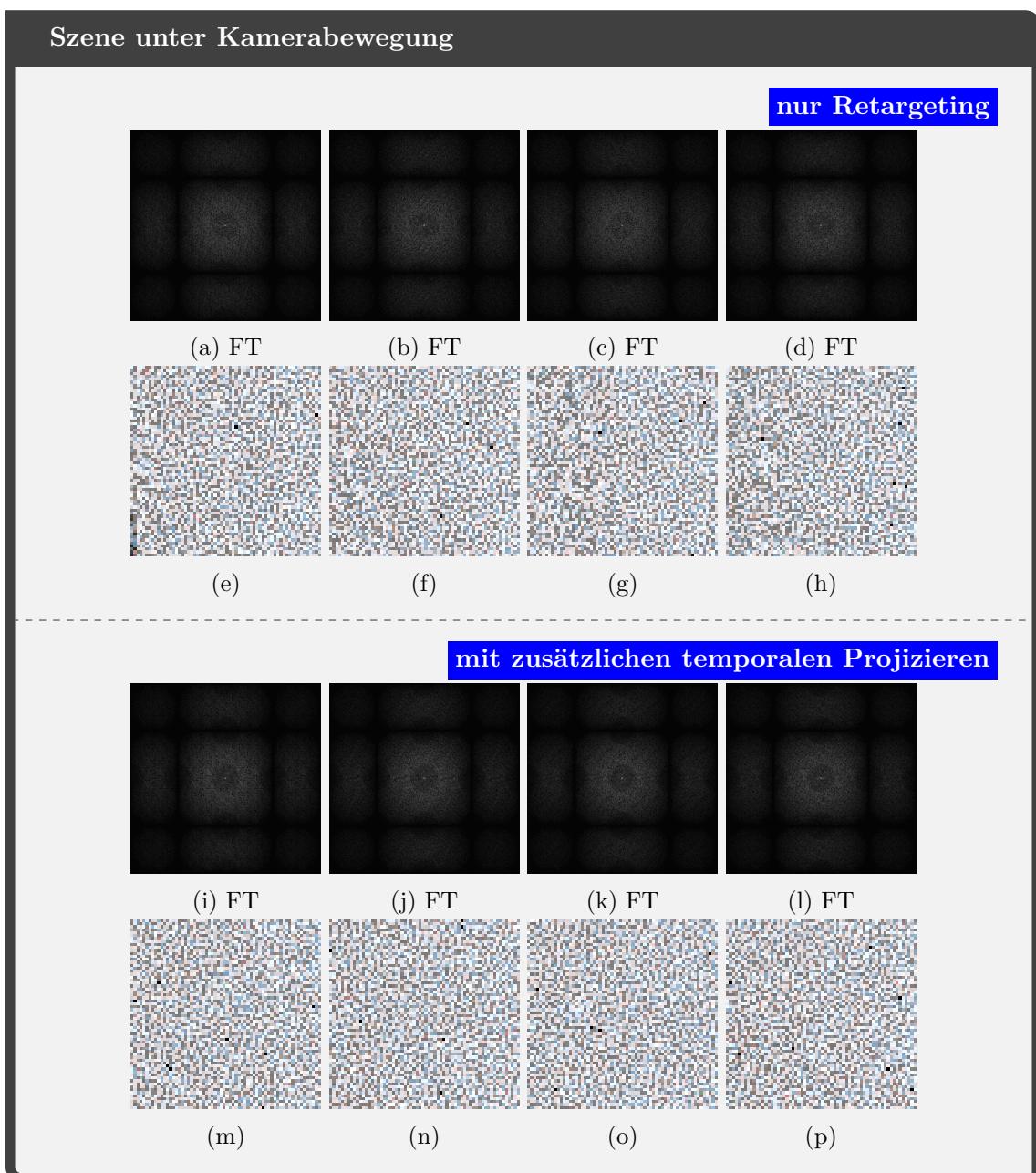


Abbildung 3.16: Erste beiden Reihen: kein temporales Projizieren; letzte beiden Reihen:  
Retargeting mit zusätzlichem temporalem Projizieren

nur Retargeting Bei reinem Retargeting sieht man in der linken Hälfte des Ausschnitts Artefakte, die unsere Blue Noise Verteilung stören. Da zwischen den Bildern die Anfangswerte nicht projiziert werden, nutzt die Bilderzeugung die anhand des vorherigen Bildes sortierten Anfangswerte. Die Sortierung allerdings wurde an den vorherigen Pixelintensitäten gemacht. Da der Ausschnitt an einem inhomogenen Bildübergang gemacht wurde, passen die neuen Pixelwerte nicht, um unsere A Posteriori(siehe Abschnitt 2.5) Bedingung zu erfüllen. Die Anfangswerte wurden zuvor anhand völlig anderer Pixelwerte sortiert.

Projizieren Bei dem zusätzlichen Projizieren wird die Bewegung der Kamera auf die Anfangswerte mit angewandt. Somit erhalten wir auch in inhomogenen Bildübergängen gute Approximationen der Pixelwerte anhand der am vorherigen Bild sortierten Anfangswerte.

### 3.4 Rechenaufwand und Speicherbedarf

Da unsere Ressourcen beschränkt sind und trotz Hardwarebeschleunigung immer noch viel Rechenzeit eines Frames auf die globale Beleuchtung entfällt, ist es von 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 (+ temporales Projizieren) sowohl auf CPU als auf GPU Seite im niedrigen Prozentbereich. Dabei wurde hier eine Blockgröße (siehe Abschnitt 3.1.0.1) von  $B = 64$  verwendet. Bei einer kleineren Blockgröße von  $B = 16$ , welche bereits gute Ergebnisse liefert, reduziert sich der Aufwand auf ein Viertel.

Rechenaufwand		
Pipeline		
Stufen	Rechenzeit(ms/%) CPU	Rechenzeit(ms/%) GPU
<b>Gesamt</b>	29.55/100%	19.04/100%
GBuffer	06.48/21.91%	01.30/6,83%
Retargeting(+ optionale temporale Reprojektion)	01.12/3.8%	00.11/0.57%
GGXGlobalIllumination	21.20/71.74%	15.51/81,46%
Sorting( $B=64$ )	00.75/2,53%	02.12/11,13%
<b>Nicht in Gesamt</b>		
Sorting( $B=16$ )	00.75	00.55

Vorberechnungen		
Vorberechnung	Rechenzeit(m)	Fehler(raw/%)
Retargeting	17,07	$10714 \approx 3.07\%$
Retargeting	35,36	$9444 \approx 2,71\%$
Temporale Projizieren	1382( $\approx 23$ h)	$\approx 10000$

Abbildung 3.17: Rechenzeiten die auf die einzelnen Stages fallen

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

\* Fehler: Anteil orientiert sich am Anfangsfehler von 348180 → Summe von pixelweise Differenz Blue Noise( $t=0$ ) und Blue Noise( $t=1$ )

Um zu einem Ergebnis wie in Figur 2.11 (siehe Abschnitt Simulated Annealing) zu kommen, braucht man  $\approx 17$  Minuten. Diese Berechnung liefert ein gutes Ergebnis bei einem akzeptablen Rechenaufwand. Für einen Fehler von 9444 (minimale Verbesserung) sind bereits  $\approx 35$  Minuten nötig.

Für das temporale Projizieren benutzen wir 81 solcher modifizierten Retarget-Texturen. Dementsprechend länger braucht die Vorberechnung. **Anmerkung:** Der Prozess lässt sich leicht parallelisieren und damit (sehr) stark Beschleunigen. Wir haben hier davon abgesehen, da uns eine einmaliger Durchlauf genügt.

Die langen Ausführungszeiten auf CPU-Seite bei der GBuffer- und GGXGlobalIlluminationsberechnung sind dabei mit dem Warten auf Resultate der GPU Seite zu erklären.

Wir speichern unsere Anfangswerte in einer 1920x1080 32-Bit-Textur. Die Blue Noise Textur innerhalb des Sorting Schrittes hat eine Auflösung von 64x64 32-Bit. Die Retarget-Textur benutzt eine 64x64 16-Bit Textur. Das temporale Projizieren benutzt 81 solcher modifizierten Retarget-Texturen.

Speicherbedarf	
Pipelinstage	Speicherbedarf/KB
Sorting	16,1
Retargeting	12,1
Temporale Reprojektion	980
Anfangswerte(Seeds)	7920
<b>Gesamt</b>	<b>8928,2</b>

Abbildung 3.18: Speicherdarf

Die Texturen für das Retargeting sowie für das temporale Projizieren wurden unkomprimiert verwendet. Dabei würde dies ein weiteres hohes Speichereinsparpotenzial bieten. Bei den 8 Bit pro Farbkanal stehen uns 256 Werte zum Abspeichern bereit. Dabei benutzen wir im Retarget-Schritt nur sieben und dem Projizieren nur zehn.

# 4. Implementierung

Zur Umsetzung der bereits eingeführten Ansätze und Algorithmen benutzen wir folgende Frameworks/Bibliotheken.

## 4.0.1 Falcor

Zur Umsetzung des Path Tracer benutzen wir das Framework Falcor [BYF<sup>+</sup>18]. Dieses Framework bietet bereits eine minimalistische Implementierung eines Path Tracer mit den neuen DirectX12 2.15 Shadereinheiten, die wir für unsere Zwecke angepasst haben. Die hier benutzte Version 3.2.1 generiert threads, die eine falsche ID besitzen ( $y$ -Komponente  $< 0$ ). Die falschen ID's führen bei einem Texturzugriff auf nicht erlaubten Speicherzugriff und damit zu schwarzen Bildbereichen. Diese schwarzen/falschen Bildpixel zerstören die geforderte Bedingung einer Permutation in unserem temporalen Algorithmus! Falls diese threads mit falscher ID nicht abgefangen werden würden, würde sich der Fehler auf das gesamte Bild ausbreiten.

Mit Falcor benutzen wir das Texturenformat BGRA8Unorm. Laden wir also eine Textur haben wir es mit 8-bit Informationen pro Kanal, die im Intervall [0,1] liegen, zu tun (lineare Abbildung → 255 wird auf 1 , 0 und 0 abgebildet [Ima]).

## 4.0.2 Anfangswerte

Für unsere vorberechnete Anfangswerttextur benötigen wir pro Eintrag zufällig generierte 32-bit Zahlen. Die Umsetzung der Zufallszahlen mit Mersenne-Twister [Mer] sowie mit der WangHash-Methode [wan] führten zu guten Ergebnissen.

## 4.0.3 Simulated Annealing

Als VS2019 Projekt mit ImGui [Img] als Benutzerschnittstelle umgesetzt.

### 4.0.3.1 FreeImage

Als Bibliothek [Fre] zum Laden und Speichern von Texturen benutzt. Man sollte unbedingt darauf achten, dass Bilder von links unten beginnend indiziert werden.

### 4.0.3.2 Visualisierungen

Die 2D-Schaubilder wurden mit einem Wrapper für Matplotlib [mat] erstellt. Für 3D-Schaubilder jedoch aufgrund aktueller bugs nicht. Dafür haben wir die anfallenden Datentripel in einer Textdatei abgespeichert und mit [gnu] visualisiert.

## 5. Zukünftige Arbeit

Während unseren Arbeiten zu Blue Noise Fehlerverteilungen im Bildraum ergaben sich weitergehende, lohnenswerte Untersuchungen, die jedoch den Umfang dieser Arbeit überstiegen hätten.

Zukünftiges	
Untersuchungen	Beschreibung
Temporales Projizieren und höhere Dimensionalität des Path Tracer	Wir haben den hier eingeführten Schritt des temporalen Projizierens anhand niederdimensionalen Rendering Integralen getestet. Die Funktionalität mit höheren Dimensionen würde den Schritten des Sorting und Retargeting entsprechen.
Ausbauen GUI bei Retargeting-Textur Berechnung	Die Berechnungen der Retargeting/Projektion-Texturen sowie der Visualisierungen lassen sich bereits simultan in separaten Threads mit benutzerspezifischen Eingaben berechnen. Allerdings sind die für den Nutzer wählbaren Parameter beschränkt. Ein weiterer Ausbau steigert die Wiederverwertbarkeit und vereinfacht andere zukünftige Arbeiten.
Parallelisieren der Texturberechnung temporales Projizieren	Der Vorgang lässt sich leicht parallelisieren und man könnte ihn dementsprechend stark beschleunigen. Wir haben hier erstmal davon abgesehen, da es eine einmalige Berechnung ist.

# Literaturverzeichnis

- [AMHH08] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-time rendering*. AK Peters/CRC Press, 2008.
- [BBHW<sup>+</sup>19] C. Barré-Brisebois, H. Halén, G. Wihlidal, A. Lauritzen, J. Bekkers, T. Stachowiak, and J. Andersson, *Hybrid Rendering for Real-Time Ray Tracing*. Berkeley, CA: Apress, 2019, pp. 437–473. [Online]. Available: [https://doi.org/10.1007/978-1-4842-4427-2\\_25](https://doi.org/10.1007/978-1-4842-4427-2_25)
- [BYF<sup>+</sup>18] N. Benty, K.-H. Yao, T. Foley, M. Oakes, C. Lavelle, and C. Wyman, “The Falcor rendering framework,” 05 2018, <https://github.com/NVIDIAGameWorks/Falcor>. [Online]. Available: <https://github.com/NVIDIAGameWorks/Falcor>
- [DS02] G. Drettakis and H.-P. Seidel, “Efficient multidimensional sampling,” vol. 21, pp. 1–8, 2002.
- [EH19] L. B. Eric Heitz, “Distributing monte carlo errors as a blue noise in screen space by permuting pixel seeds between frames,” vol. 38, pp. 1–10, 2019. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-02158423/document>
- [Fre] “Freeimage,” etablierte; ausführlich getestete lib zum Laden/ Speichern von Bildern. [Online]. Available: <http://freeimage.sourceforge.net/>
- [Gam17] E. Games, “The problem with 3d blue noise,” 2017, Blogpost. [Online]. Available: <http://momentsingraphics.de/3DBlueNoise.html>
- [GF16] I. Georgiev and M. Fajardo, “Blue-noise dithered sampling,” in *ACM SIGGRAPH 2016 Talks*. ACM, 2016, p. 35.
- [gnu] “gnuplot,” very good for visualizing 3d-data. [Online]. Available: <http://www.gnuplot.info/>
- [Haj88] B. Hajek, “Cooling schedules for optimal annealing,” *Mathematics of operations research*, vol. 13, no. 2, pp. 311–329, 1988.
- [HAM19] E. Haines and T. Akenine-Möller, Eds., *Ray Tracing Gems*. Apress, 2019, <http://raytracinggems.com>.
- [Ima] “Imageformat,” erklärt was es mit dem BGRA8Unorm-Format, das in Falcor benutzt wird, zu tun hat. [Online]. Available: [https://www.khronos.org/opengl/wiki/Image\\_Format](https://www.khronos.org/opengl/wiki/Image_Format)
- [Img] “Imgui,” als Benutzerschnittstelle zur Erstellung der benötigten Texturen. [Online]. Available: <https://github.com/ocornut/imgui>
- [INS18] “Temporal reprojection anti-aliasing in inside,” 2018, iNSIDE at GDC. [Online]. Available: <https://youtu.be/2XXS5UyNjjU>
- [JCr18] “Jcrystal,” 2018, performing fft on images. [Online]. Available: <http://www.jcrystal.com/products/ftlse/index.htm>

- [Kaj86] J. T. Kajiya, “The rendering equation,” in *ACM SIGGRAPH computer graphics*, vol. 20, no. 4. ACM, 1986, pp. 143–150.
- [KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: <https://science.sciencemag.org/content/220/4598/671>
- [Kra18] B. Kraft, “Aufbau turing architektur,” blogpost, 2018. [Online]. Available: <https://www.heise.de/newsticker/meldung/GeForce-RTX-Nvidia-verraet-mehr-zur-Technik-von-Turing-4165369.html>
- [Krc06] V. Krcadinac, “A new generalization of the golden ratio,” *Fibonacci Quarterly*, vol. 44, no. 4, p. 335, 2006.
- [mat] “matplotlib,” good for 2D-plots; 3d plots to buggy! [Online]. Available: <https://github.com/lava/matplotlib-cpp>
- [Mer] “random numbers through mersenne twister,” abgerufen 09.03.2020. [Online]. Available: <https://de.wikipedia.org/wiki/Mersenne-Twister>
- [Pad02] R. Padovan, “Dom hans van der laan and the plastic number,” *Nexus IV: Architecture and Mathematics*, pp. 181–193, 2002. [Online]. Available: <http://www.nexusjournal.com/conferences/N2002-Padovan.html>
- [Pat19] 2019, 2019 siggraph course. [Online]. Available: <https://jo.dreggn.org/path-tracing-in-production/2019/ptp-part2.pdf>
- [Pet16] C. Peters, “Free blue noise textures,” blogpost, 2016. [Online]. Available: <http://momentsingraphics.de/>
- [Ras20] 2020. [Online]. Available: <https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation?url=https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation>
- [Ray19] “Rayflag enumeration,” 2019, control behaviour of a traced ray. [Online]. Available: [https://docs.microsoft.com/en-us/windows/win32/direct3d12/ray\\_flag](https://docs.microsoft.com/en-us/windows/win32/direct3d12/ray_flag)
- [Rob18] M. Roberts, “The unreasonable effectiveness of quasirandom sequences,” <http://extremelearning.com.au/unreasonable-effectiveness-of-quasirandom-sequences/>, 2018.
- [Sch19] C. Schied, “Real-time path tracing and denoising in quake 2.” Game Developer Conference, 2019. [Online]. Available: <https://www.gdcvault.com/play/1026185/>
- [Sci20] “Cooling schedule,” 2020, very good overview and further explanation of kirkpatrick. [Online]. Available: <https://www.sciencedirect.com/topics/computer-science/cooling-schedule>
- [SKW<sup>+</sup>17] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and M. Salvi, “Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination,” in *Proceedings of High Performance Graphics*, 2017, pp. 1–12.
- [UE414] “High-quality temporal supersampling,” 2014, brian Karris. 2014. [Online]. Available: [https://www.youtube.com/redirect?q=https%3A%2F%2Fde45xmedrsdbp.cloudfront.net%2FResources%2Ffiles%2FTemporalAA\\_small-71938806.pptx&redir\\_token=gVWkz9-V68sJl5FFM8oGqOJpD818MTU4MzE1NzkzMUAxNTgzMDcxNTMx&v=yNQ47MY-Eo0&event=video\\_description](https://www.youtube.com/redirect?q=https%3A%2F%2Fde45xmedrsdbp.cloudfront.net%2FResources%2Ffiles%2FTemporalAA_small-71938806.pptx&redir_token=gVWkz9-V68sJl5FFM8oGqOJpD818MTU4MzE1NzkzMUAxNTgzMDcxNTMx&v=yNQ47MY-Eo0&event=video_description)
- [Uli88] R. A. Ulichney, “Dithering with blue noise,” *Proceedings of the IEEE*, vol. 76, no. 1, pp. 56–79, Jan 1988.
- [Uli93] R. A. Ulichney, “Void-and-cluster method for dither array generation,” in *Human Vision, Visual Processing, and Digital Display IV*, vol. 1913. International Society for Optics and Photonics, 1993, pp. 332–343.

- [VLA87] P. J. Van Laarhoven and E. H. Aarts, “Simulated annealing,” in *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.
- [wan] “Quick and easy gpu random numbers in d3d11,” blogpost about random numbers; abgerufen 09.03.2020. [Online]. Available: <http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/>
- [Whi19] “Whitenoisegenerator,” <https://www.cssmatic.com/noise-texture>, 2019, accessed: 24.11.2019.

# **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 11. März 2020

(Jonas Heinle)