Architektur und Programmierung von Grafik- und Koprozessoren

Rendering Algorithmen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

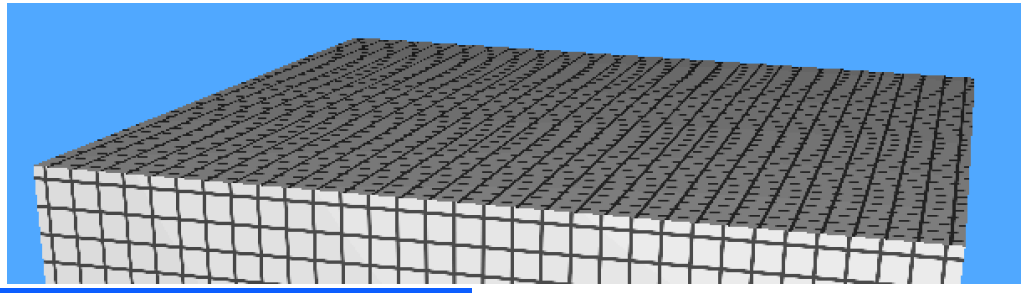
Aliasing und Sampling Theorie

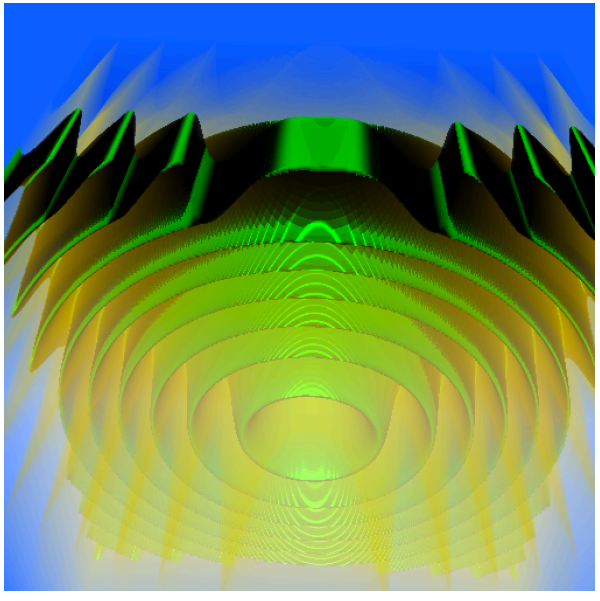
Aliasing und Sampling Theorie[1]

- *Aliasing* durch verschiedenste Quellen spielt in Computergrafik große Rolle.
- *Anti-Aliasing* integraler Teil von Hardware (GPUs).
  - Multi-Sample Anti-Aliasing.
  - Dedizierte Textur-Sampling Einheiten.
  - Spezialisierte Hardware für Mip-Mapping, anisotropisches Filtering etc.
- Daher *kurze* Einführung in Sampling Theorie, um zu verstehen, auf welchen Ebenen Anti-Aliasing nötig.

[1] vgl. z. B. Pharr, Jakob, Humphreys: Physically Based Rendering, 3rd ed. (2017)

Aliasing Quellen





Aliasing Quellen

- Aliasing in der Bildebene.
- Aliasing im Texturraum.
- Aliasing bei spiegelnden Oberflächen.
- Zeitliches Aliasing.
- Aliasing durch zu niedrige Farbauflösung.
- ...

Dirac Delta Verteilung und Shah Funktion

Dirac Delta Funktion:

$$\int \delta(x)dx = \begin{cases} 1, & \text{if } x = 0 \\ 0 & \text{sonst} \end{cases} \tag{4}$$

Für $f(x)$ stetig differenzierbar folgt $\int f(x)\delta(x)dx = f(0)$.

Shah Funktion (a.k.a. "Impulse Train"): unendliche Summe äquidistanter Delta Funktionen:

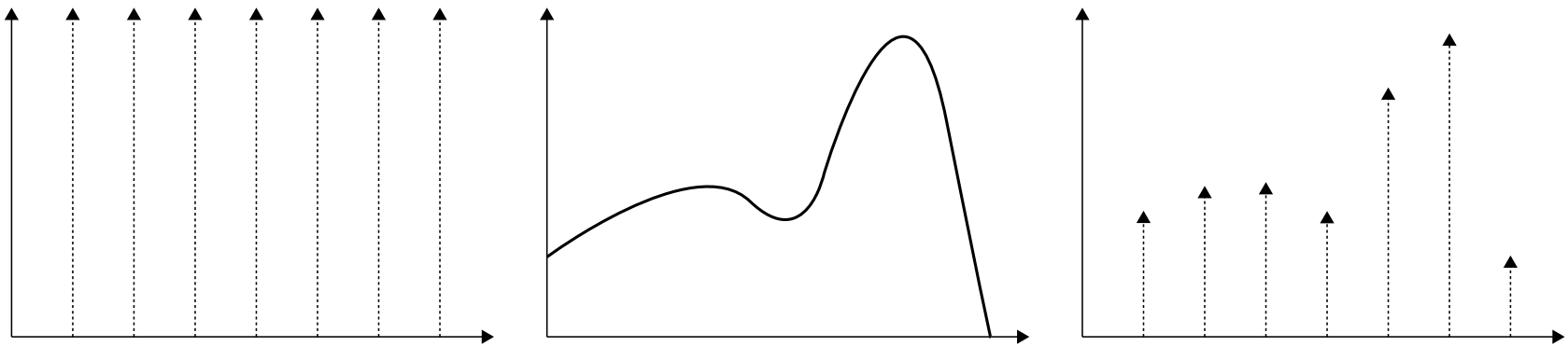
$$\text{Ш}_\Delta(x) = \Delta \sum_{i=-\infty}^{\infty} \delta(x - i\Delta), \tag{5}$$

wobei Δ die *Periode* der Shah Funktion.

Sampling und Rekonstruktion

Durch Multiplikation der stetigen, "zu samplenden" Funktion $f(x)$ mit der Shah Funktion ergibt sich eine unendliche Sequenz von *Sample Punkten*:

$$\text{Ш}_\Delta(x)f(x) = \Delta \sum_{i=-\infty}^{\infty} \delta(x - i\Delta)f(i\Delta). \tag{6}$$



Sampling und Rekonstruktion

Erhalte $\bar{f}(x)$ durch *Faltung* mit *Rekonstruktionsfilterfunktion* $r(x)$:

$$\bar{f}(x) = (\text{Ш}_\Delta(x) f(x)) * r(x), \tag{7}$$

wobei die Faltung zweier Funktionen $f(x)$ und $g(x)$ definiert ist als

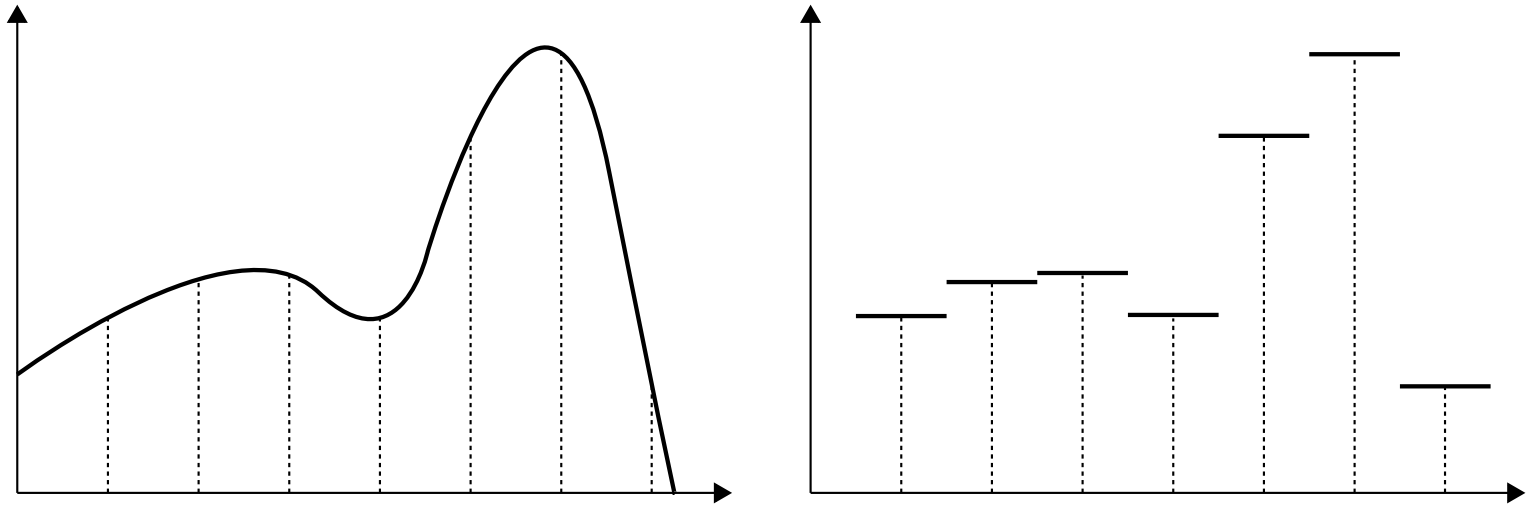
$$f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau) g(x - \tau) d\tau. \tag{8}$$

Bei der Faltung mit dem Rekonstruktionsfilter ergibt sich die gewichtete Summe:

$$\bar{f}(x) = \Delta \sum_{i=-\infty}^{\infty} f(i\Delta) r(x - i\Delta). \tag{9}$$

Sampling und Rekonstruktion

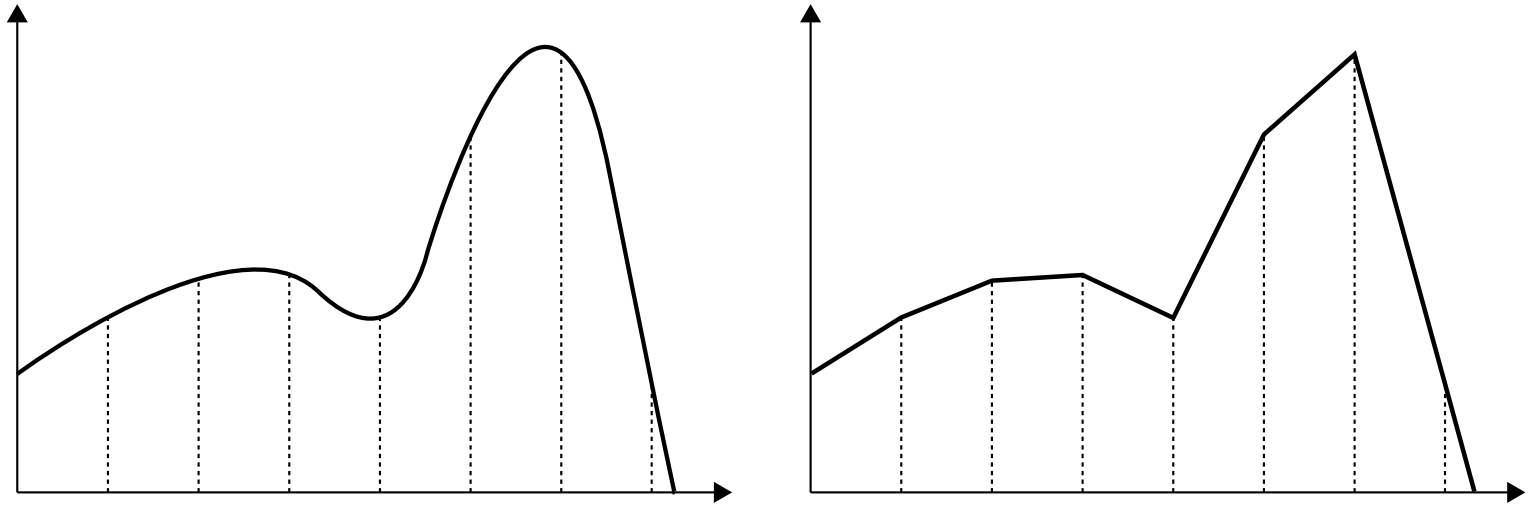
Rekonstruktion mit Box Filter



$$r(x) = \begin{cases} 1, & \text{if } |x| < \frac{1}{2} \\ 0 & \text{sonst} \end{cases} . \tag{10}$$

Sampling und Rekonstruktion

Stückweise lineare Rekonstruktion (Tent Filter)



$$r(x) = max(0, 1 - |x|) \tag{11}$$

Rekonstruktion

Exakt rekonstruierbare Signalfunktionen

Es gibt eine Klasse von Funktionen (*bandlimitierte* Funktionen), die, wenn man eine entsprechende Sampling Rate wählt, *exakt* aus der Sequenz von Sample Punkten rekonstruiert werden kann.

Fourier Transformation

$f(x)$ hat Repräsentation im *Ortsraum* sowie im *Frequenzraum*.

Fourier Transformation: Ortsraum $\Rightarrow$ Frequenzraum:

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi\omega x} dx, \tag{12}$$

inverse Fourier Transformation: Frequenzraum $\Rightarrow$ Ortsraum:

$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{i2\pi\omega x} d\omega. \tag{13}$$

(Dabei sind $e^{ix} = cos(x) + isin(x)$ und $i = \sqrt{-1}$.)
Idee: *Signale* lassen sich im Limit als gewichtete Summe von Sinusoiden repräsentieren.

Fourier Transformation

Orts- und Frequenzrepräsentation wichtiger Funktionen

Ortsraum	**Frequenzraum**
Konstante: $f(x) = 1$	Dirac Delta: $F(\omega) = \delta(\omega)$
Box: $f(x) = 1 \begin{cases} \text{if} & \lvert x \rvert < \frac{1}{2}, \\ 0 & \text{sonst} \end{cases}$	Sinc: $F(\omega) = \frac{sin(\pi\omega)}{\pi\omega}$
Shah: $f(x) = \Delta \sum_{i=-\infty}^{\infty} \delta(x - \Delta i)$	Shah: $F(\omega) = \frac{1}{\Delta} \sum_{i=-\infty}^{\infty} \delta(\omega - \frac{1}{\Delta})$

Tabelle: vgl. Pharr, Jakob, Humphreys: Physically Based Rendering, 3rd ed. (2017), p. 405.

Fourier Transformation

Wichtige Eigenschaft der Fourier Transformation:
Faltung in Ortsraum entspricht Multiplikation in Frequenzraum, und umgekehrt:

$$f(x)g(x) = F(\omega) * G(\omega), \tag{14}$$

sowie

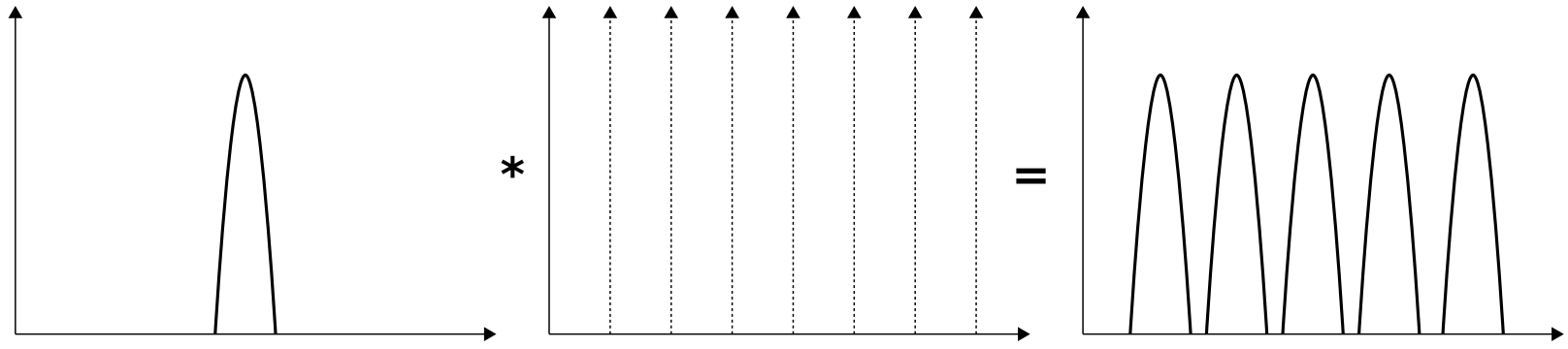
$$F(x)G(x) = f(\omega) * g(\omega). \tag{15}$$

Das Produkt von weiter oben: $\mathrm{III}_\Delta(x)f(x)$ entspricht also einer *Faltung* mit Periode $\frac{1}{\Delta}$ im Frequenzraum:

$$\mathrm{III}_\Delta(x)f(x) = \mathrm{III}_{\frac{1}{\Delta}}(\omega) * F(\omega). \tag{16}$$

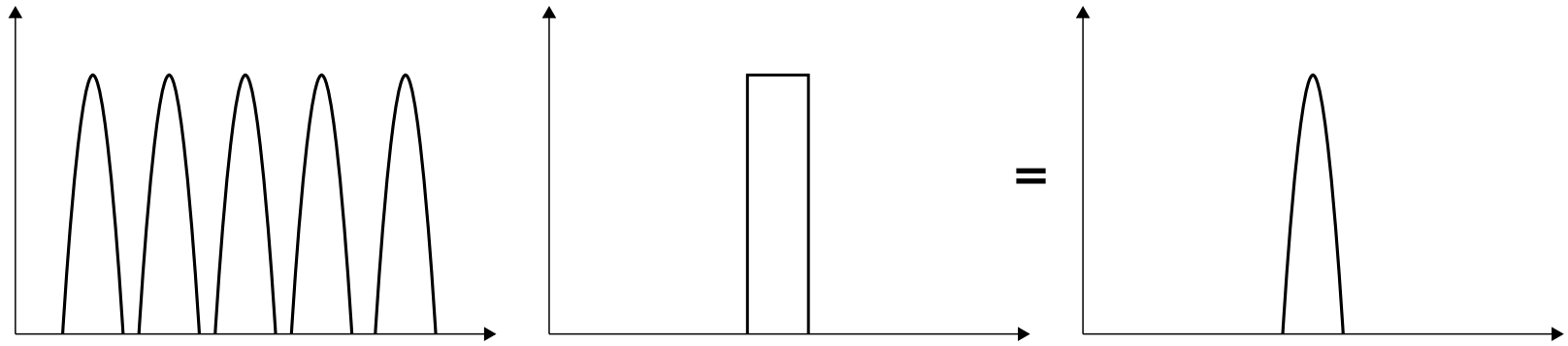
Bandlimitierte Signale

Faltung mit Shah Funktion in Frequenzraum: Signal wird unendlich oft *repliziert*, mit Periode der Shah Funktion.



Bandlimitierte Signale

Multiplikation mit Box Filter in Frequenzraum, um alle außer einer Kopie zu verwerfen.



Box mit Breite proportional zur Sampling Rate Δ:

$$r(x) = \begin{cases} \frac{1}{2\Delta} & |x| < \Delta \\ 0 & \text{sonst} \end{cases}. \tag{17}$$

Bandlimitierte Signale

Im Frequenzraum entspricht dieser Vorgang wegen o. g. Eigenschaften der Multiplikation mit der Shah Funktion $\mathrm{III}_\Delta(x)$ sowie einer Faltung mit dem *sinc* Rekonstruktionsfilter.

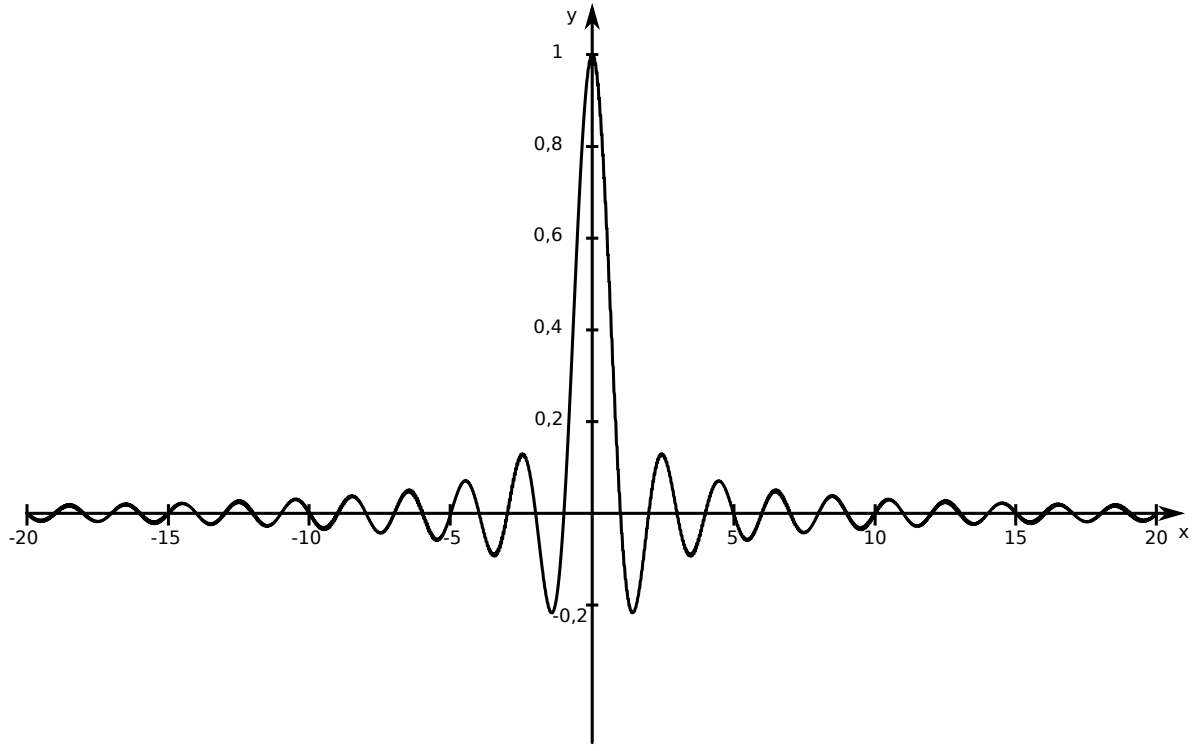


Abbildung: sinc Filter im Interval $[-20..20]$, vgl. Yvonne Percan, Untersuchung und Klassifikation des Fehlerverhaltens bei Direktem Volume Rendering, Diplomarbeit (2014)

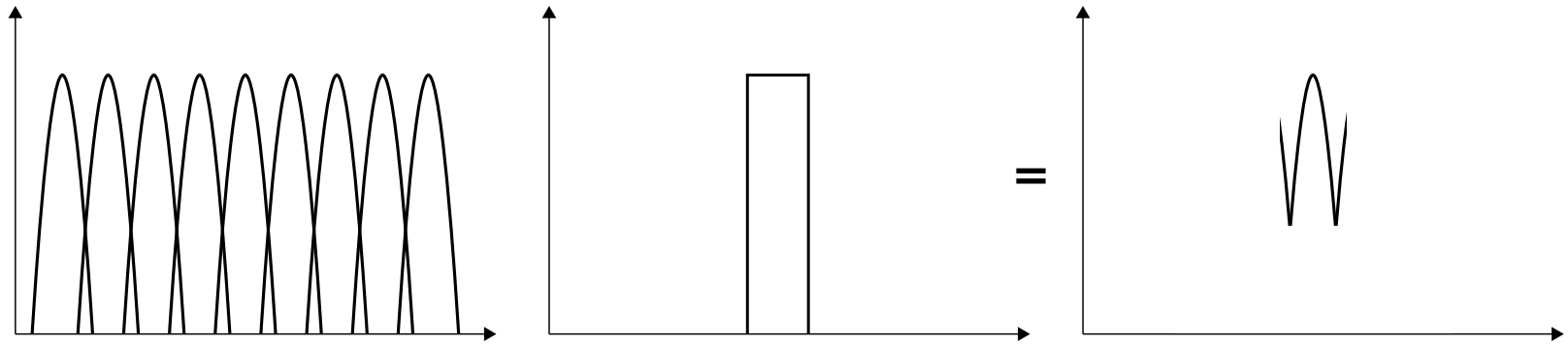
Bandlimitierte Signale

Problem bei dieser Betrachtung:

- *sinc* Filter hat unendliche Ausdehnung.
- Signal muss bandlimitiert sein.

Bandlimitierte Signale

Nicht-bandlimitierte Signale: Replikationen “überlappen”, es wird ein Teil einer anderen Frequenz mit rekonstruiert.



Frequenz *gibt sich als andere Frequenz aus* $\Rightarrow$ *Aliasing*.

Anti-Aliasing im Bildraum

- Nicht-bandlimitierte Signale $\Rightarrow$ niemals exakt rekonstruierbar. Aliasing, egal wie hoch die Sampling Rate.
- Sehr viele Samples: erhöhte Rechenzeit, Floating-Point Rundungsungenauigkeiten etc.
- Platziere so viele Samples, sodass optimal bzgl. *Konvergenz*.

Anti-Aliasing im Bildraum

Menschliches Auge weniger empfindlich ggü Rauschen. Weiche Schatten: links: 16 uniforme Samples, rechts: 16 zufällige Samples.

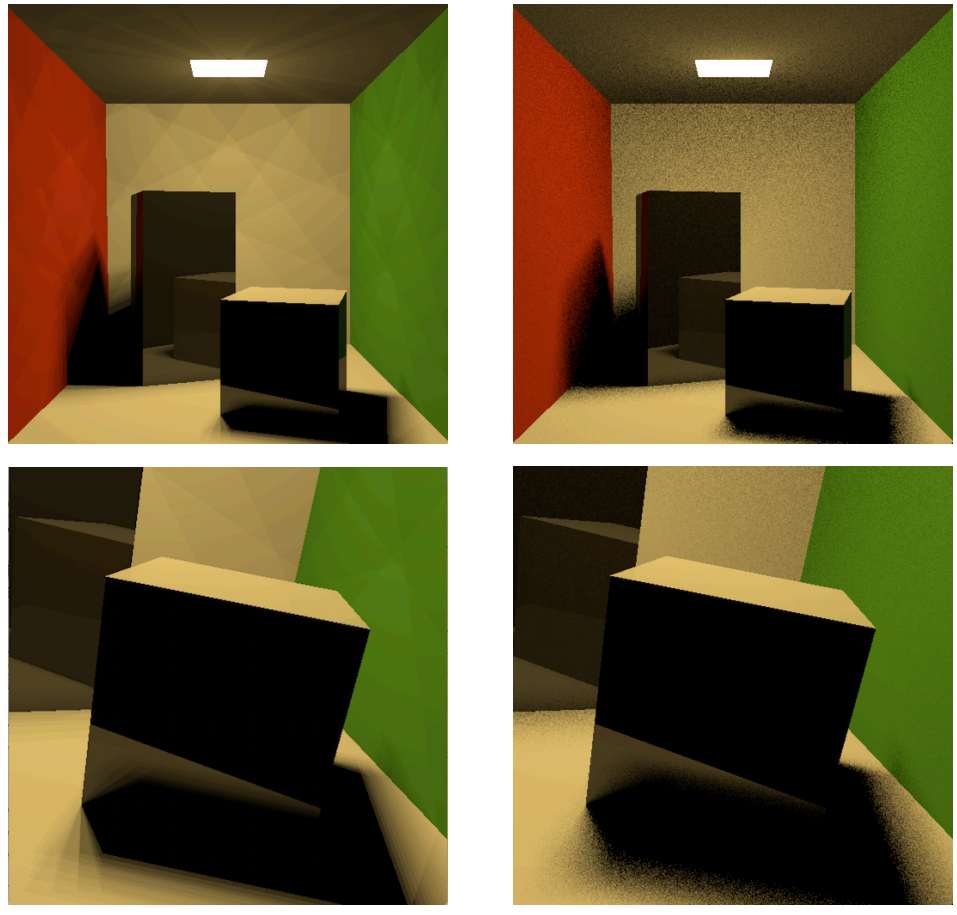


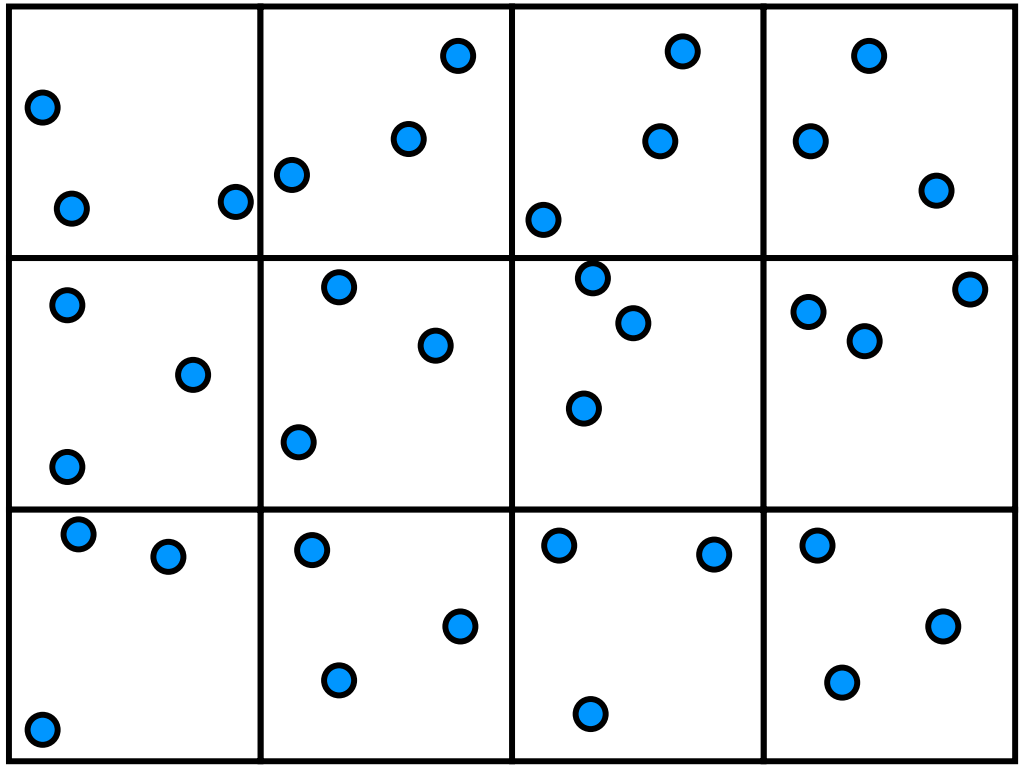
Tabelle: Cornell Box (https://www.graphics.cornell.edu/online/box/), Rendering: Stefan Zellmann

Anti-Aliasing im Bildraum

- Punkt-Samples an *diskreten* Stellen auf der Bildebene.
  - Pixel haben keine Fläche!
- Idee: Mehrere Sample “um Pixel herum”, gewichte jedes Sample mit Rekonstruktionsfilter (z. B. Tent oder Funktion höherer Ordung wie B-Spline).

Anti-Aliasing im Bildraum

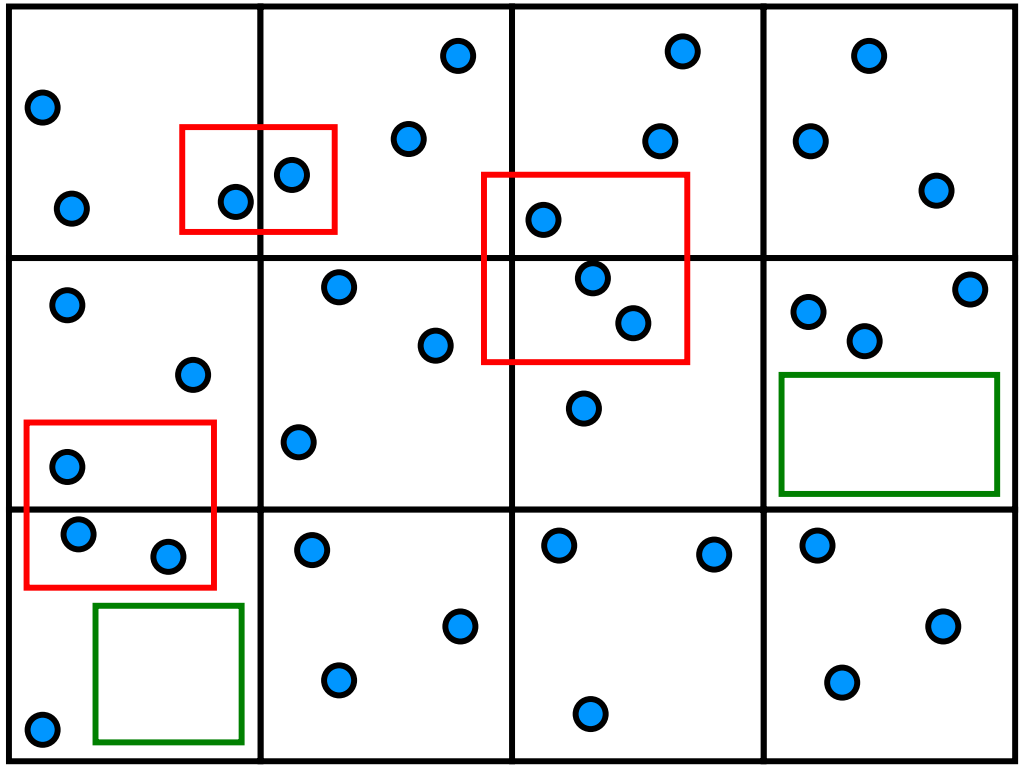
Jittered Sampling



Sample um Pixelposition herum, verschiebe um zufällig uniform verteiltes $\delta \in [0..\frac{1}{2})$.

Anti-Aliasing im Bildraum

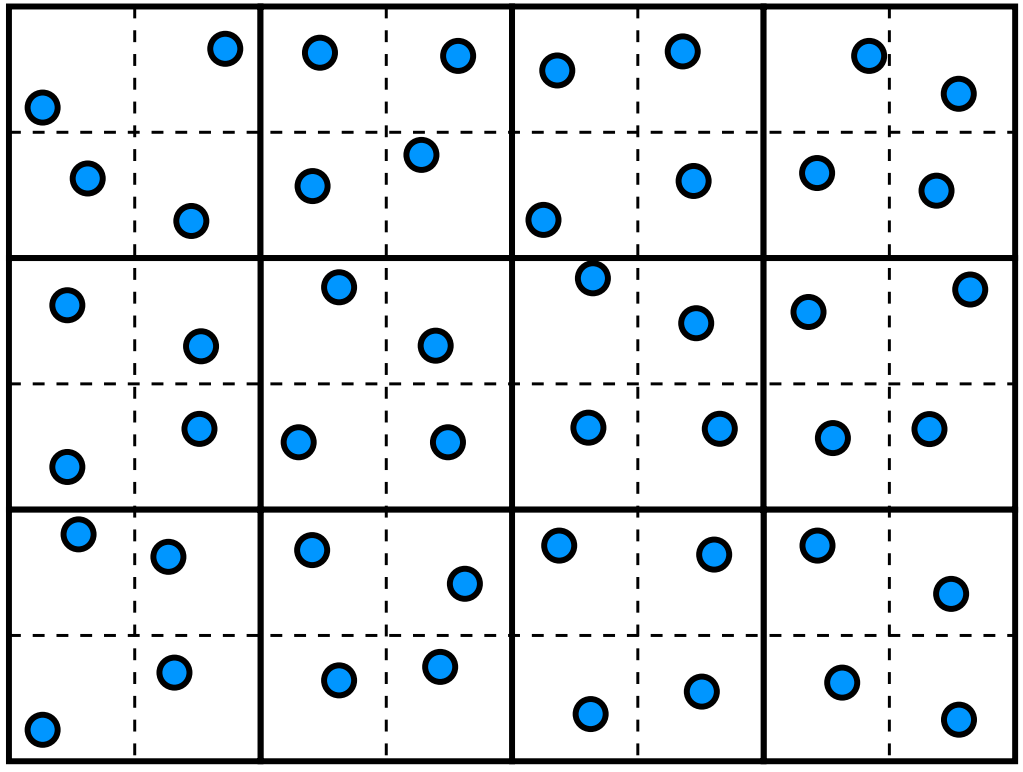
Jittered Sampling



Problem: Cluster-Bildung und unterrepräsentierte Bildregionen.

Anti-Aliasing im Bildraum

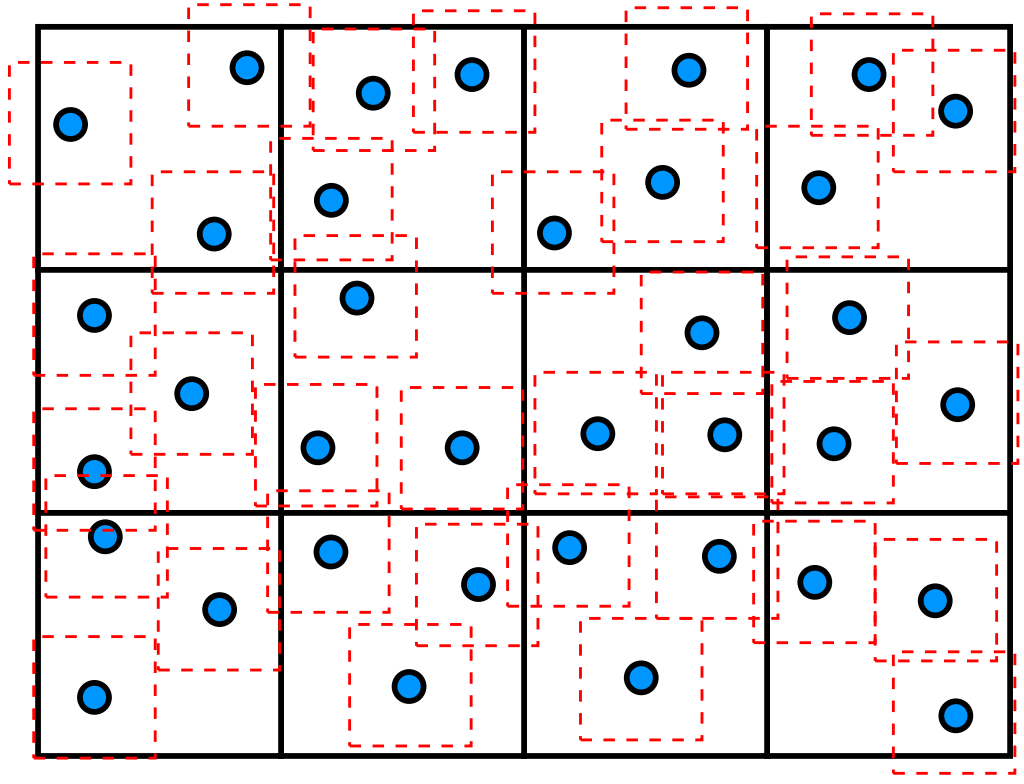
Stratified Sampling



Eine Lösungsstrategie: teile Pixel in *Strata* ein, sample innerhalb der Strata. Bessere Verteilung, jedoch andere Probleme (z. B. fixe Dimensionalität der Strata).

Anti-Aliasing im Bildraum

Low-Discrepancy Sampling



Informell: finde Sampling Positionen, die Überlapp der roten Boxen minimieren.

Anti-Aliasing im Bildraum

Diskrepanz

Sei B Familie von Polyedern b_i in $[0,1)^n$ (Linien, Rechtecke, Boxen mit Ursprung bei 0 und Kantenlänge < 1). Sei P eine Menge von N Sample Punkten innerhalb des Polyeders $[0,1)^n$.

Die Diskrepanz von P bzgl. B ist dann:

$$D(P,B) = \sup_{b \in B} \left| \frac{N_b}{N} - V(b) \right|, \tag{18}$$

wobei N_b die Anzahl Sample Punkte in b und $V(b)$ das Volumen von b.

Anti-Aliasing im Bildraum

Diskrepanz

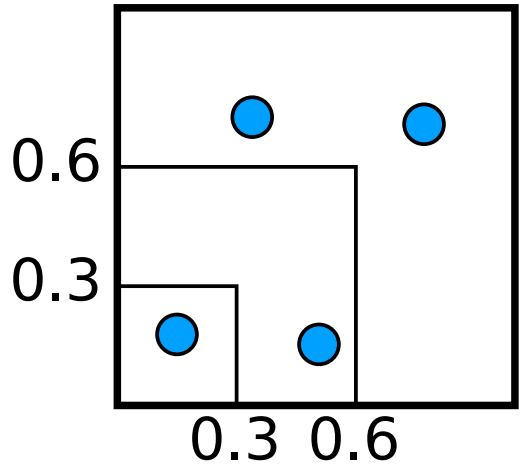


Abbildung: vgl. Pharr, Jakob, Humphreys: Physically Based Rendering, 3rd ed. (2017)

Punkte in $[0..1)^2$. Die Diskrepanz in der kleinen Box ($b_1 = [0, 0.3)$) beträgt $D = 0.25 - 0.09 = 0.16$, die Diskrepanz in der größeren Box ($b_2 = [0, 0.6)$) beträgt $D = 0.5 - 0.36 = 0.14$. Diskrepanz bzgl. P in $B = \{b_1, b_2, ..\}$ ist Maximum über alle D.

Anti-Aliasing im Bildraum

Bemerkung zur Diskrepanz

Offensichtlich haben uniform verteilte Sample Punkt Sequenzen die niedrigste Diskrepanz. Wir suchen Sequenzen mit niedriger Diskrepanz, die keine uniformen Sampling Artefakte mit sich bringen.

Anti-Aliasing im Bildraum

Low-Discrepancy Sequenzen

- “Quasi-Zufallszahlen”, entsprechen regelmäßigem Bildungsgesetz.
- Weiterführend. Beispiele sind:
  - Halton Sequenz.
  - Sobol Sequenz.

Multisample Anti-Aliasing auf GPUs

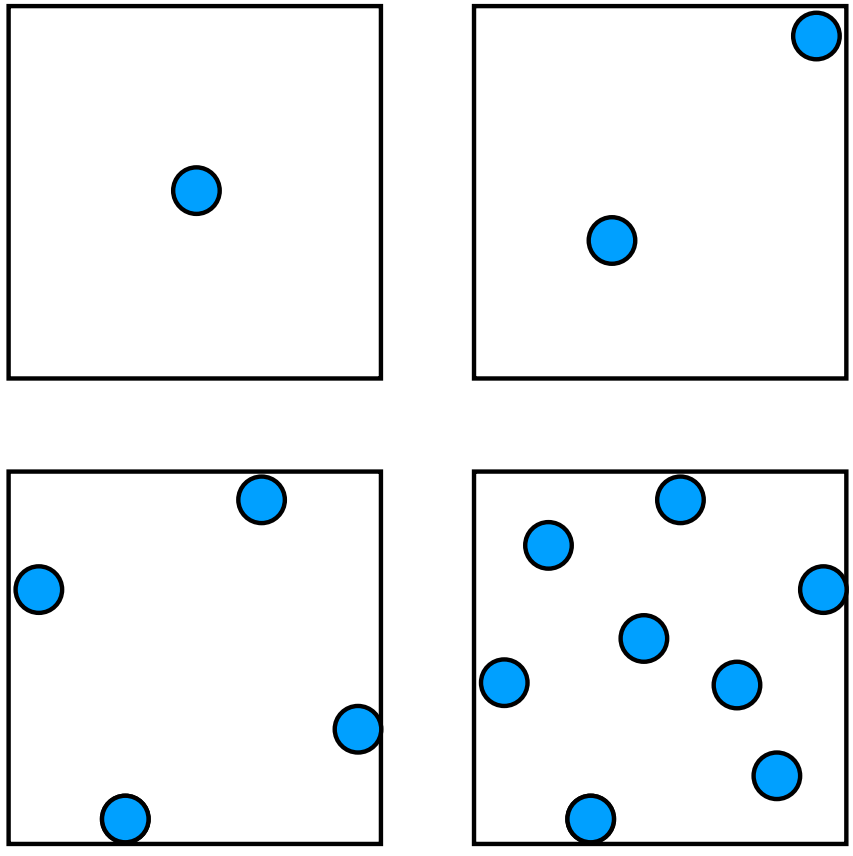


Abbildung: vereinfacht gemäß: EQAA Modes for AMD 6900 Series Graphics Cards, AMD Developer Relations (2011)

Multisample Anti-Aliasing auf GPUs

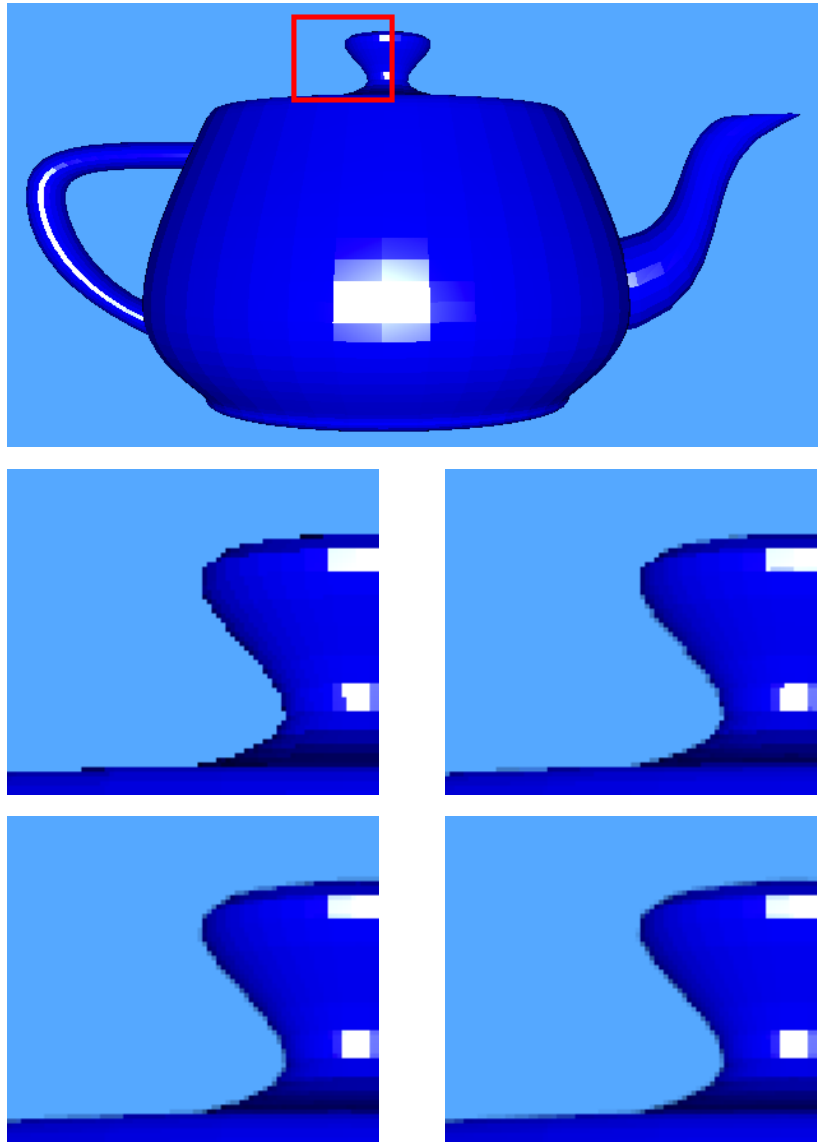
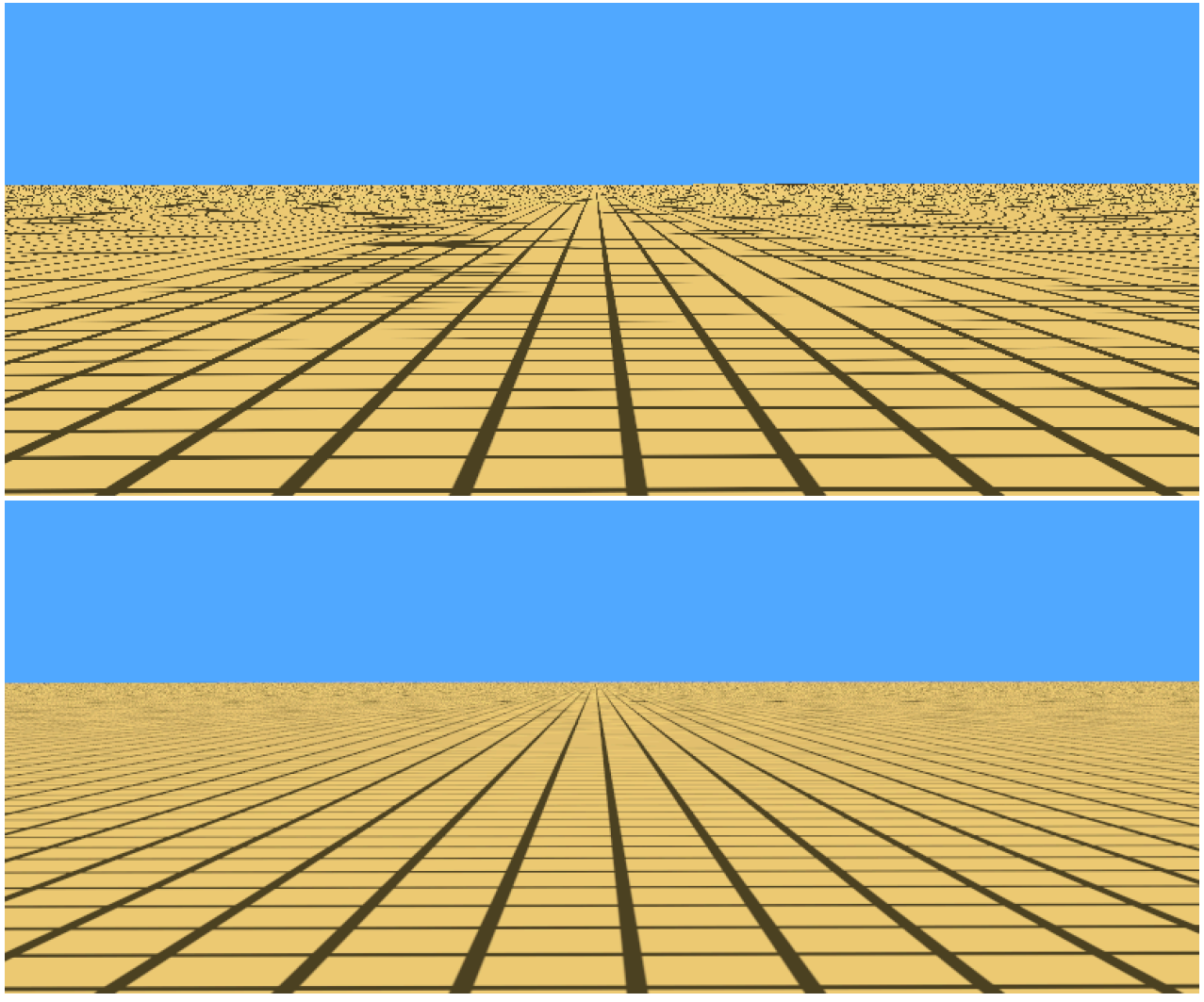


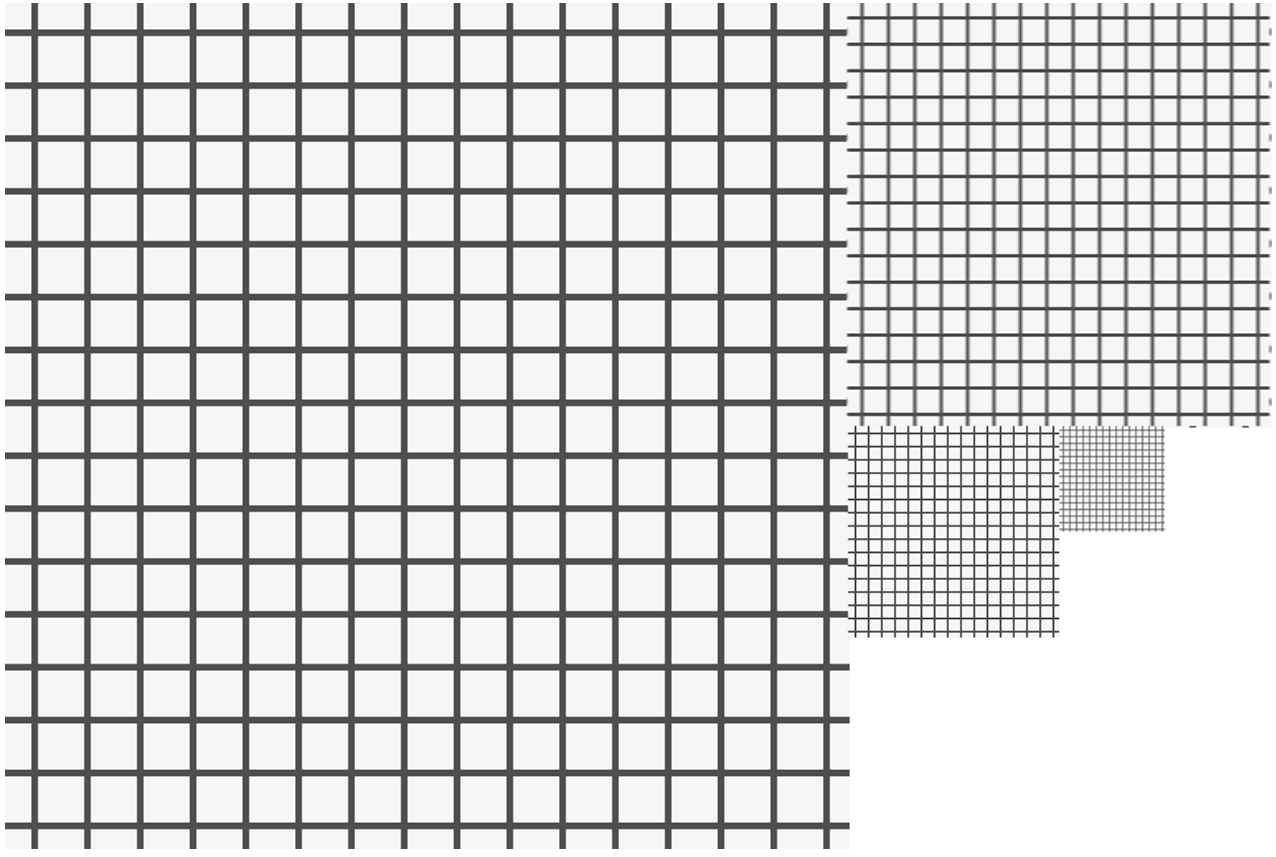
Tabelle: 1x, 2x, 4x und 8x Multisample Antialiasing (MSAA)

Textur Sampling auf GPUs



Textur Sampling auf GPUs

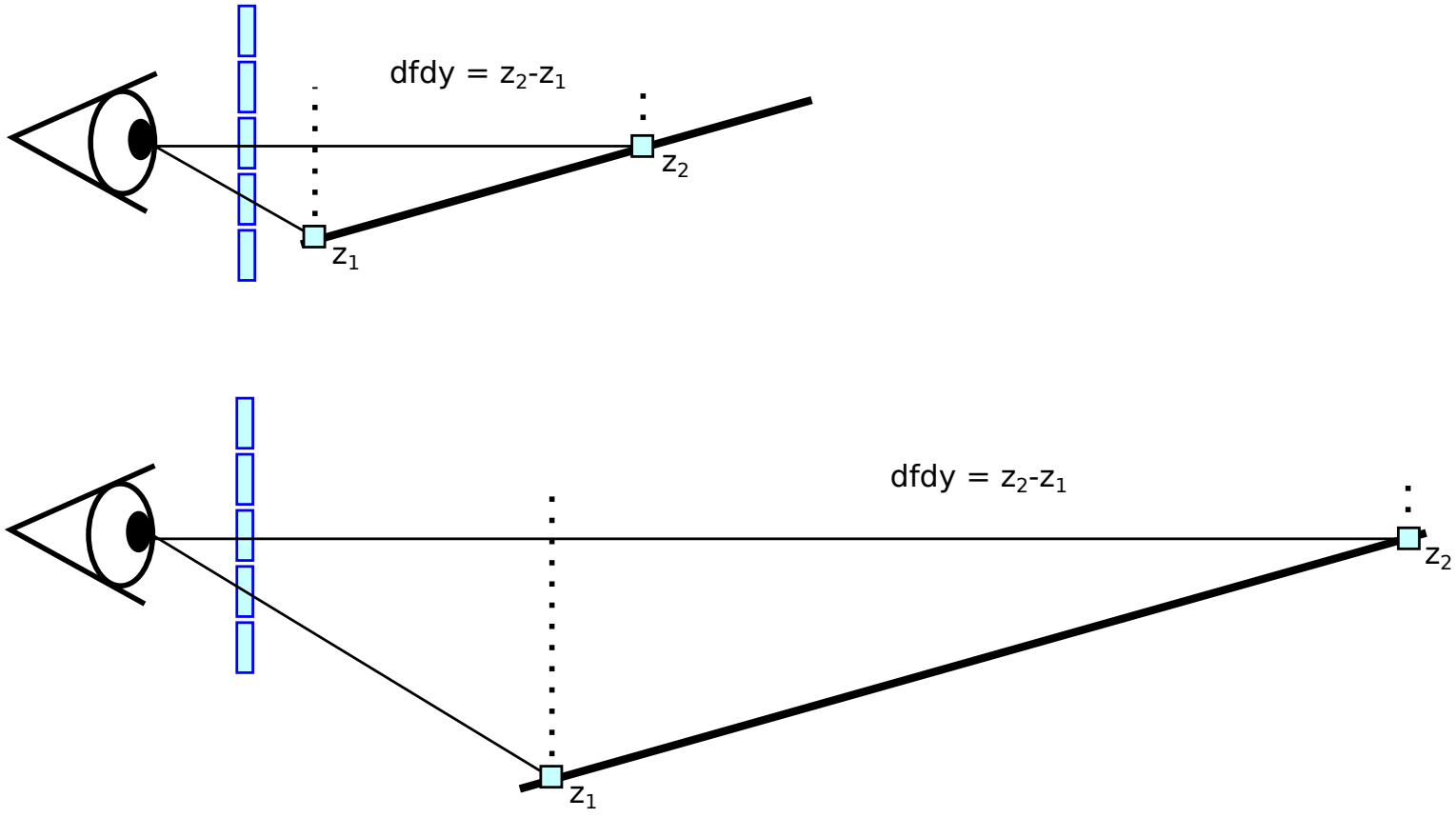
Mip-Mapping



Textur Sampling auf GPUs

Texture Derivatives

GPU *Raster Engines* wählen Mip-Map Level aufgrund der z-Distanz benachbarter Rasterpunkte. 1^{st}-order Derivatives $\Rightarrow$ Raster Engines verarbeiten 2×2 Bildschirmregionen ("Quads").



Drei Rendering Algorithmen

Auswahlkriterien

Warum genau diese drei Algorithmen?

Es gibt eine Vielzahl von Rendering Algorithmen. Wir sind an denjenigen Algorithmen interessiert, die derzeit produktiv zum Einsatz kommen. Wir konzentrieren uns auf Algorithmen, die heute am populärsten z. B. in Film- oder Spieleindustrie sind, oder beim Echtzeit Rendering für Wissenschaftliche Visualisierung zum Einsatz kommen.

Wir behandeln drei Algorithmen, die entweder in Grafik Hardware fest implementiert sind, oder die sich einfach mit Hilfe von *programmierbarer* Grafikhardware implementieren lassen.

Übersicht Rendering Algorithmen

Rasterisierung / z-Buffer Algorithmus

GPU Fixed-Function Pipeline, OpenGL und DirectX nutzen diesen Algorithmus.

g-Buffer Algorithmen ("Deferred Shading/Lighting")

Einfaches z-Buffering ist ineffizient, wenn viele Lichtquellen vorhanden. Erweiterung des z-Buffer Algorithmus, führe Beleuchtungsberechnung aus, *nachdem* Tiefentest entschieden ist ($\rightarrow$ viele Lichter).

Strahlverfolgung

Physikalisch plausibler als Rasterisierung - es wird ein Lichtpartikelmodell angenommen, bei dem sich Photonen entlang gerader Linien bewegen und dabei Phänomenen wie Absorption oder Streuung ausgesetzt sind.

Anmerkung zur Performanz von Rendering Algorithmen

Es gibt keine “per Konstruktion” langsamen Algorithmen, bestimmte Phänomene aber besser (oder schneller) mit bestimmten Algorithmen lösbar, z. B.:

- wenige Punktlichtquellen, lokale Beleuchtung: eher Rasterisierung.
- globale Beleuchtung, Scattering Phänomene, Flächenlichtquellen: eher Ray Tracing et al.
- viele Lichtquellen, Einsatz von Post Processing Verfahren (z. B. Bloom, Lens Flare, Screen Space Ambient Occlusion): g-Buffer.

Hardware Implementierung

Dedizierte Hardware Implementierung kann Algorithmen beschleunigen:

- Umgehung von General Purpose Befehlssatz.
- Speicherabstraktion ausgelegt auf Speicherzugriffsmuster, z. B. raumfüllende Kurven für 2D Texturen.
- Hardware Implementierung geht häufig mit dediziertem Befehlssatz für den Algorithmus einher (z. B. Register für 4-Komponenten Vektor Operationen) (denn nicht programmierbare Architekturen auf Dauer langweilig).

Popularität des Rasterisierungsalgorithmus rührt u. a. daher, dass GPUs speziell diesen Algorithmus in Hardware implementieren.

Weiteres Vorgehen

- Zunächst Erarbeiten einer formalen Beschreibung der drei Familien von Rendering Algorithmen.
- Darauf aufbauend Komplexitätsanalyse:
  - bzgl. #Polygone, #Pixel (Auflösung), sowie #Lichtquellen.
  - unter Annahme einer Turing Maschine (seriell, wahlfreier Zugriff auf Speicher).
  - später unter Annahme paralleler Architektur (PRAM).

Der Rasterisierungsalgorithmus

Rasterisierung

Im Abschnitt *Grundlegendes* dieses Vorlesungsteils haben wir die meisten Einzelkomponenten des Rasterisierungsalgorithmus schon kennengelernt. Wir wollen nun die Komponenten zusammenfügen und den Algorithmus formal analysieren.

Rasterisierung - Eingabedaten (1)

Liste von Vertices $V_i := \{\mathbf{p_i}, \vec{n_i}, TC_i, C_i, \cdots\}$

- Position $\mathbf{p_i} \in \mathbb{R}^3$

Weitere (ggf. benutzerdefinierte) Vertex Attribute, z. B.:

- Normalenvektor $\vec{n_i} \in \mathbb{R}^3$, wobei $\|\vec{n_i}\| = 1$
- Texturkoordinate $TC_i \in \mathbb{R}^2$
- Vertexfarben $\mathbf{C_i}$

Annahme:

Jeweils n aufeinanderfolgende Vertices $V_i, i \bmod n = [1, .., n-1, 0]$ sind koplanar bzgl. $\mathbf{p_i}$ (wir nehmen $n = 3$ an, also Dreiecke).

Rasterisierung - Eingabedaten (2)

Liste von Materialeigenschaften und Texturen

M_k

Ausführliche Behandlung in Vorlesungseinheit “Anwendungen: Photorealistische Computergrafik”.

Liste von Lichtquellen $L_j := \{P_j, I_j\}$

Position $P_j \in \mathbb{R}^3$

Lichtintensität $I_j(W/sr)$

Rasterisierung - Eingabedaten (3)

Beleuchtungsfunktion ("Shader")

$$
\begin{aligned}
f : \{\vec{\omega_i}, \vec{\omega_o}, M_k, L_j, ...\} &\rightarrow C, \\
C &\in \mathbb{R}^n_+, \\
\vec{\omega_i}, \vec{\omega_o} &\in \mathbb{R}^3
\end{aligned}
\tag{19}
$$

Ausführliche Behandlung in Vorlesungseinheit "Anwendungen: Photorealistische Computergrafik".
Häufig bidirektionale Reflektionsverteilungsfunktionen in Abhängigkeit von Lichtrichtungsvektoren $\vec{\omega_i}$ und $\vec{\omega_o}$ (kompliziertere Modelle sind möglich).
C ist i. Allg. Farbe oder spektrale Energieverteilung.

Rasterisierung - Eingabedaten (4)

Kameraparameter

Matrix $MV \in \mathbb{R}^{4\times4}$ (*extrinsische* Kameraparameter).
Matrix $PR \in \mathbb{R}^{4\times4}$ (*intrinsische* Kameraparameter).
Bildebene definiert durch Rechteck $VP = (x, y, w, h)$
(Ursprung + Breite und Höhe).

Extrinsische Parameter $\rightarrow$ Positionierung und Orientierung der Kamera bzgl. der Geometrie (affine Transformationen).

Intrinsische Parameter transkodieren, wie die *perspektivische Verzerrung* (der Effekt, dass entfernte Objekte kleiner erscheinen) anzuwenden ist. Hängen von Kameraeigenschaften wie Blendenöffnungswinkel, fokale Länge etc. ab.

Rasterisierung - Ausgabe

Rasterbild $I(VP)$, Dimensionen gemäß VP aus den Eingabedaten.

Rasterisierung Algorithmus 1

```
function Rasterisierung(V,M,L,f,MV,PR,VP)
    for all v1, v2, v3 ∈ V do
        Transformationen(v1,v2,v3,MV,PR)
        Clipping(v1,v2,v3,PR)
        T ←ErzeugeDreiecke(v1,v2,v3)
        for all t ∈ T do
            Fragmente ←ScanKonvertierung(t,VP)
            for all F ∈ Fragmente do
                for all Lj ∈ Lichtquellen do
                    Beleuchte(F,M,Lj,f)
                end for
                TiefenTest(F, I)
                AlphaBlending(F, I)
            end for
        end for
    end for
end function
```

Rasterisierung - Vertex Phase

function RASTERISIERUNG(V,M,L,f,MV,PR,VP)
    for all v_1, v_2, $v_3 \in V$ **do**
        TRANSFORMATIONEN(v,MV,PR)
        CLIPPING(v,PR)
        ...
    end for
end function

1. Wende Viewing- und Perspektivische auf jedes Vertex an.
2. Clippe Vertices am Sichtbaren Frustum.

Rasterisierung - Primitive Assembly

function RASTERISIERUNG(V,M,L,f,MV,PR,VP)

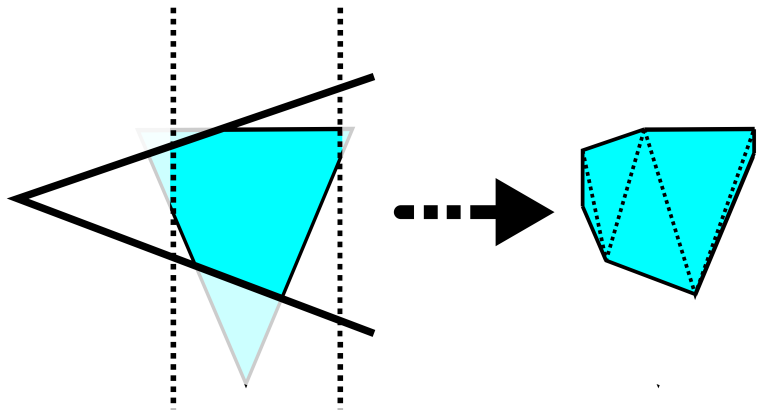
...

$T \leftarrow$ERZEUGEDREIECKE(v_1,v_2,v_3)

...

end function

3. Für jedes Eingabedreieck $\{v_1, v_2, v_3\}$, erzeuge Dreiecke, die sich *nach Clipping* ergeben.



Rasterisierung - Fragment Phase

```
function RASTERISIERUNG(V,M,L,f,MV,PR,VP)
    ...
    for all t ∈ T do
        Fragmente ←SCANKONVERTIERUNG(t,VP)
        for all f ∈ Fragmente do
            for all L_j ∈ Lichtquellen do
                BELEUCHTE(F,M,L_j,f)
            end for
            TIEFENTEST(F, I)
            ALPHABLENDING(F, I)
        end for
    end for
    ...
end function
```

4. Fragmente ergeben sich nach Scan Konvertierung.
5. Jedes Fragment wird beleuchtet, erst dann Tiefentest!

Rasterisierung Algorithmus 1 - Bemerkungen

- Manche Implementierungen führen Beleuchtungsberechnungen auf den Vertices durch und interpolieren später die resultierenden Farben.
- Der Algorithmus unterstützt sowohl opake als auch teiltransparente Geometrie. I. d. R. wird man den Tiefentest nur für opake, und Alpha-Blending nur für teiltransparente Fragmente ausführen.
- APIs (Direct3D, OpenGL, Vulkan) garantieren, dass Reihenfolge während Fragment Phase Eingabereihenfolge entspricht.

Rasterisierung Algorithmus 1 - Worst-Case Komplexität

Vorabüberlegungen

- Aus der *Primitive Assembly Phase* ergeben sich durch Clipping bis zu vier Dreiecke. Die Schleife über alle Dreiecke T lässt sich also durch $O(4)$ abschätzen.
- Die Anzahl an Fragmenten, die nach Scan Konvertierung *eines* Dreiecks entstehen, ist durch $O(VP)$ beschränkt, wobei VP die Bildschirmauflösung bezeichnet.
- Wir nehmen an, dass es für die Subroutinen Transformationen und Clipping, Beleuchte, TiefenTest, AlphaBlending und ErzeugeDreiecke jeweils $O(1)$ Algorithmen gibt. ScanKonvertierung ist durch $O(VP)$ beschränkt.

Rasterisierung - Worst-Case Komplexität

Es ergibt sich also im schlechtesten Fall für den Algorithmus Rasterisierung (1) die Komplexität:

$$O(V) \times O(4) \times (O(VP) \times O(L) + O(VP)) = O(V \times VP \times L) \quad (20)$$

Wir wollen im folgenden vereinfachend annehmen, dass die Anzahl Lichtquellen L konstant ist. Damit ergibt sich die Laufzeitkomplexität

$$O(V \times VP). \quad (21)$$

Wenn wir uns mit "Deferred Shading" beschäftigen, werden wir wieder von variabel vielen Lichtquellen ausgehen.