Compositing und Blending

Verwendet Folien von Ed Angel

(Professor Emeritus of Computer Science, University of New Mexico)

Ziele

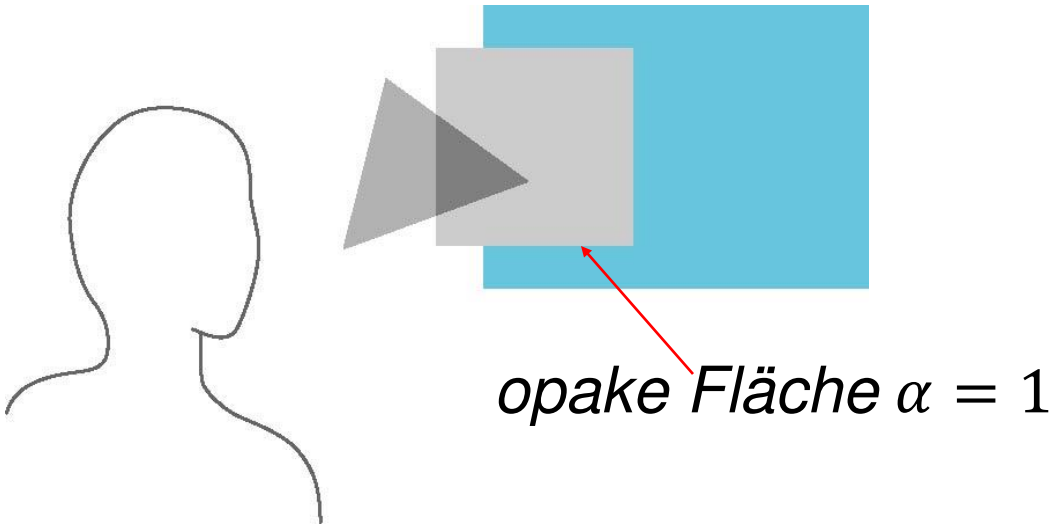
- Verwendung der A-Komponente in RGBA-Farben für
  - durchscheinende (transluzente) Oberflächen
  - Bild-Compositing
  - Antialiasing

- Korrekte Behandlung von Verdeckungseffekten

Opazität und Transparenz

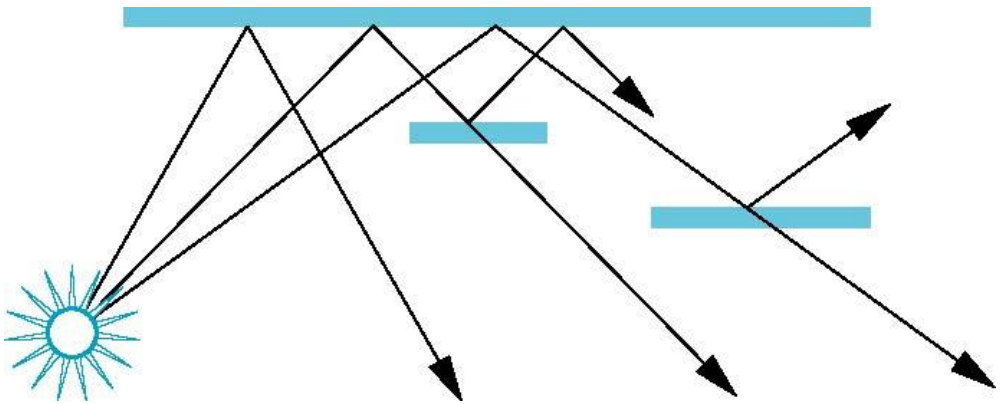
- Opake Oberflächen lassen kein Licht durch
- Transparente Oberflächen lassen alles Licht durch
- Durchscheinende Oberflächen lassen etwas Licht durch

translucency = 1 – opacity (α)



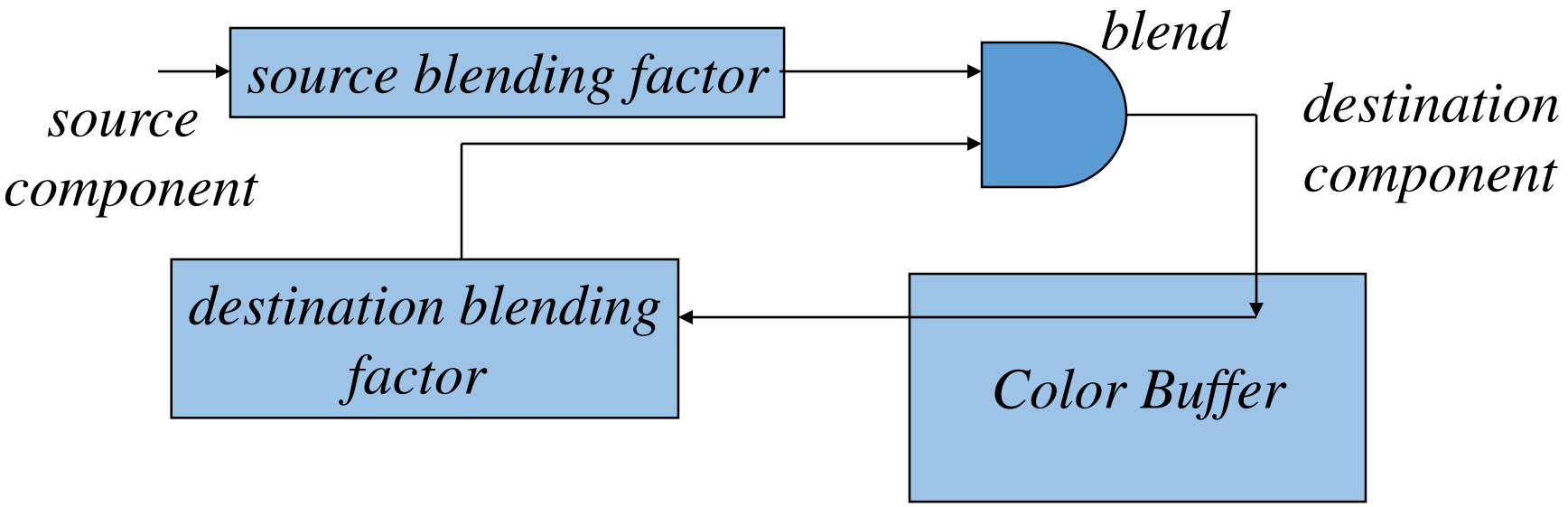
Physikalische Modelle

- Durchscheinende Materialien physikalisch korrekt zu behandeln ist schwer, weil
  - die innere Wechselwirkung zwischen Licht und Materie komplex ist
  - wir einen Pipeline-Renderer verwenden



Schreibmodell

- Verwende die A-Komponente von RGBA (oder RGBα)-Farbe um Opazität zu speichern
- Beim Rendern beziehen wir A ein, um die vorhandene und die neue Pixelfarbe zu mischen:



Blending-Gleichung

- Wir definieren Mischfaktoren für jede RGBA-Komponente:

$$\mathbf{s} = [s_r, s_g, s_b, s_\alpha]$$

$$\mathbf{d} = [d_r, d_g, d_b, d_\alpha]$$

Angenommen, die Quell- (source) und Ziel- (destination) Farben sind:

$$\mathbf{b} = [b_r, b_g, b_b, b_\alpha]$$

$$\mathbf{c} = [c_r, c_g, c_b, c_\alpha]$$

Die Mischfarbe ist dann

$$\mathbf{c}' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha]$$

OpenGL: Blending und Compositing

- Blending muss aktiviert und die Quell- und Zielfaktoren festgelegt werden:

```
glEnable(GL_BLEND)
glBlendFunc(source_factor,
            destination_factor)
```

- Nur bestimmte Faktoren werden unterstützt
  - `GL_ZERO, GL_ONE`
  - `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`
  - `GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA`
  - und ein paar andere

Beispiel

- Angenommen, wir fangen mit der opaken Hintergrundfarbe (R_0,G_0,B_0,1) an
  - Mit dieser Farbe wird die Zielfarbe initialisiert.
- Jetzt wollen wir ein transluzentes Polygon mit Farbe (R_1,G_1,B_1,α_1) darüberlegen
- `GL_SRC_ALPHA` und `GL_ONE_MINUS_SRC_ALPHA` als Quell- und Ziel-Mischfaktoren

$$R'_1 = \alpha_1 R_1 + (1 - \alpha_1) R_0, \ldots\ldots$$

- Dies ist der sog. **Over**-Operator [Porter+Duff 1984]

Quiz

- Welche Blending-Funktion für Lens Flare?

Zusätzliches Licht durch Reflexionen im Objektiv -> additiver Effekt

Daher: `glBlendFunc(GL_ONE, GL_ONE)`

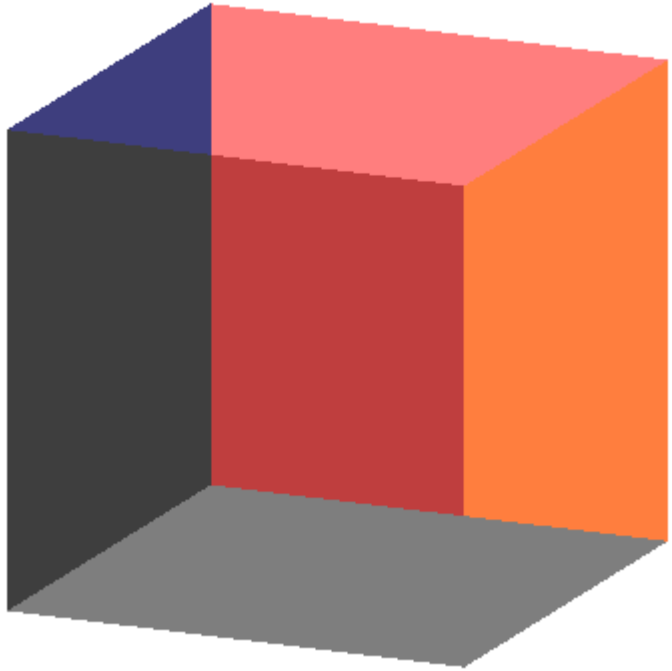
Gleichermaßen für Feuer, Irrlichter, usw.

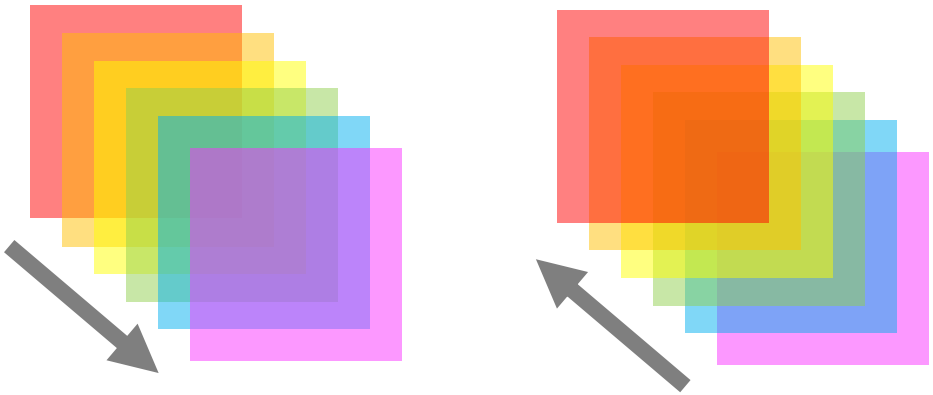
Sättigung (Clamping) und Genauigkeit

- Alle Komponenten (RGBA) sind gesättigt und bleiben im Bereich (0,1)
- Oft werden RGBA-Werte jedoch mit 8-Bit-Genauigkeit abgelegt.
  - Bei Mischung vieler Komponenten kann die Genauigkeit leiden
  - Beispiel: n Bilder zusammenaddieren
    - Teile alle Farbkomponenten durch n, um Sättigung zu vermeiden
    - Blending mit Quellfaktor = 1, Zielfaktor = 1
    - Aber Division durch n verringert Genauigkeit

Abhängigkeit von der Reihenfolge

- Ist dieses Bild korrekt?
  - Wahrscheinlich nicht
  - Polygone werden in der Reihenfolge gerendert, in der sie die Pipeline durchlaufen
  - Blending-Ergebnis hängt von der Reihenfolge ab





Korrektes Alpha-Blending

- Lösung: Sortiere Polygone vor dem Rendern

- **Back-to-front** Compositing ergibt [Porter+Duff 1984]

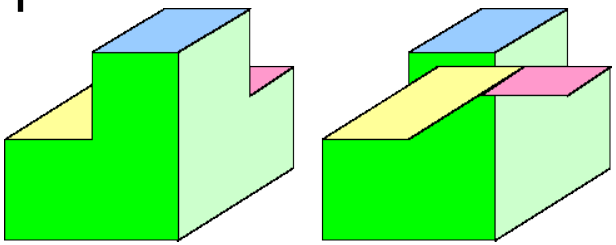
$$C_{total} = \sum_{i=1}^{n} C_i \, \alpha_i \prod_{\substack{k=1 \\ z_k < z_i}}^{n} (1 - \alpha_k)$$

- **Front-to-back**: verwende Alpha-Buffer, um Faktoren $(1 - \alpha_k)$ zu akkumulieren
- Front-to-back umständlicher, aber erlaubt Abbruch wenn Transluzenz zu klein wird (Vordergrund nahezu opak)
- Verwendung z.B. im texturbasierten Volumenrendering

Verdeckung (Hidden Surface Removal)

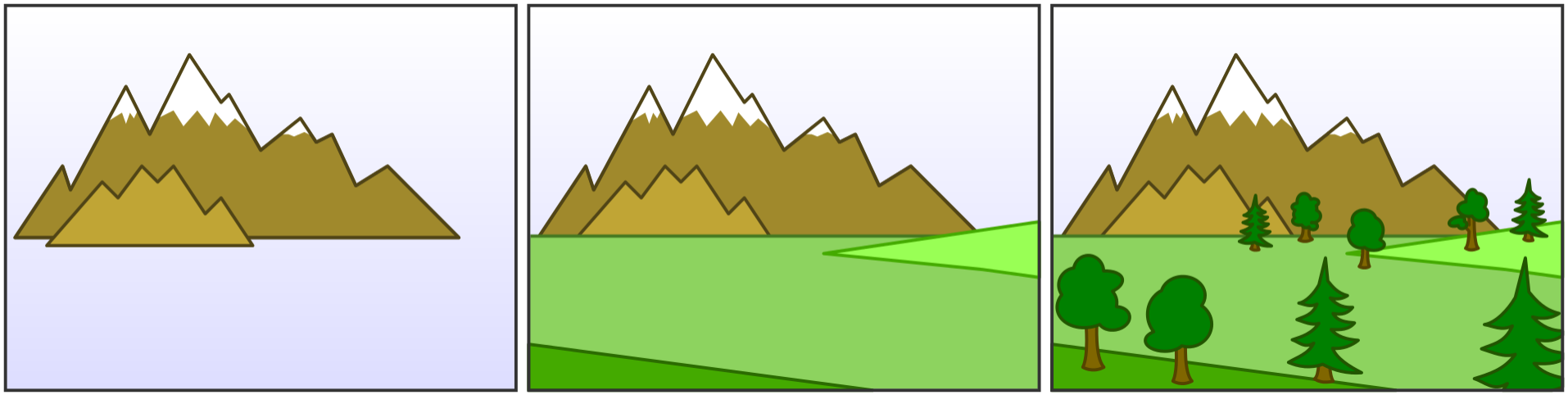
- Die Zeichenreihenfolge wirkt sich entscheidend auf das Resultat aus
- Wie lässt sich sicherstellen, dass der Vordergrund nicht vom Hintergrund verdeckt wird?

- **Backface Culling**
  - `glEnable(GL_CULL_FACE); glCullFace(GL_BACK);`
  - Zeichne Polygone nur von der Vorderseite
  - Generell gute Idee (spart ca. 50% Rechenleistung)
  - Kann früh in der Pipeline (während Primitive-Konstruktion) einfach durchgeführt werden
  - Löst das Verdeckungsproblem aber nur bei konvexen Objekten



Verdeckung (Hidden Surface Removal)

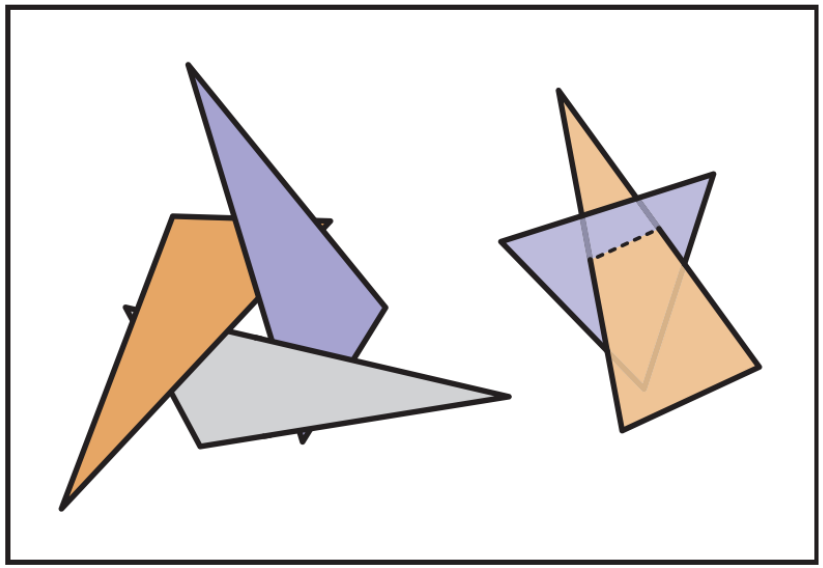
- Maleralgorithmus (Painter's Algorithm)
- Back-to-front-Rendering in der Regel mit rein opaken Objekten
- Erfordert Sortieren der Objekte

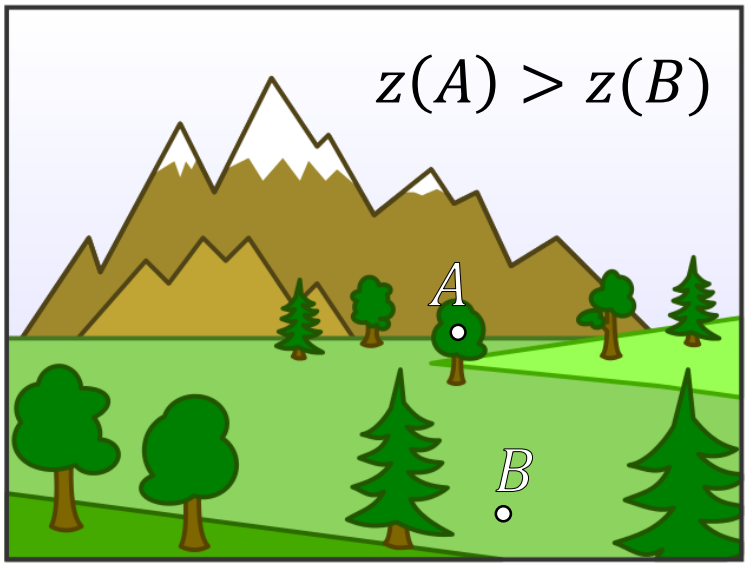


Probleme mit dem Sortieren

- Sortieren ist langsam... (jedenfalls zu langsam, um es sich einmal pro Frame leisten zu können)

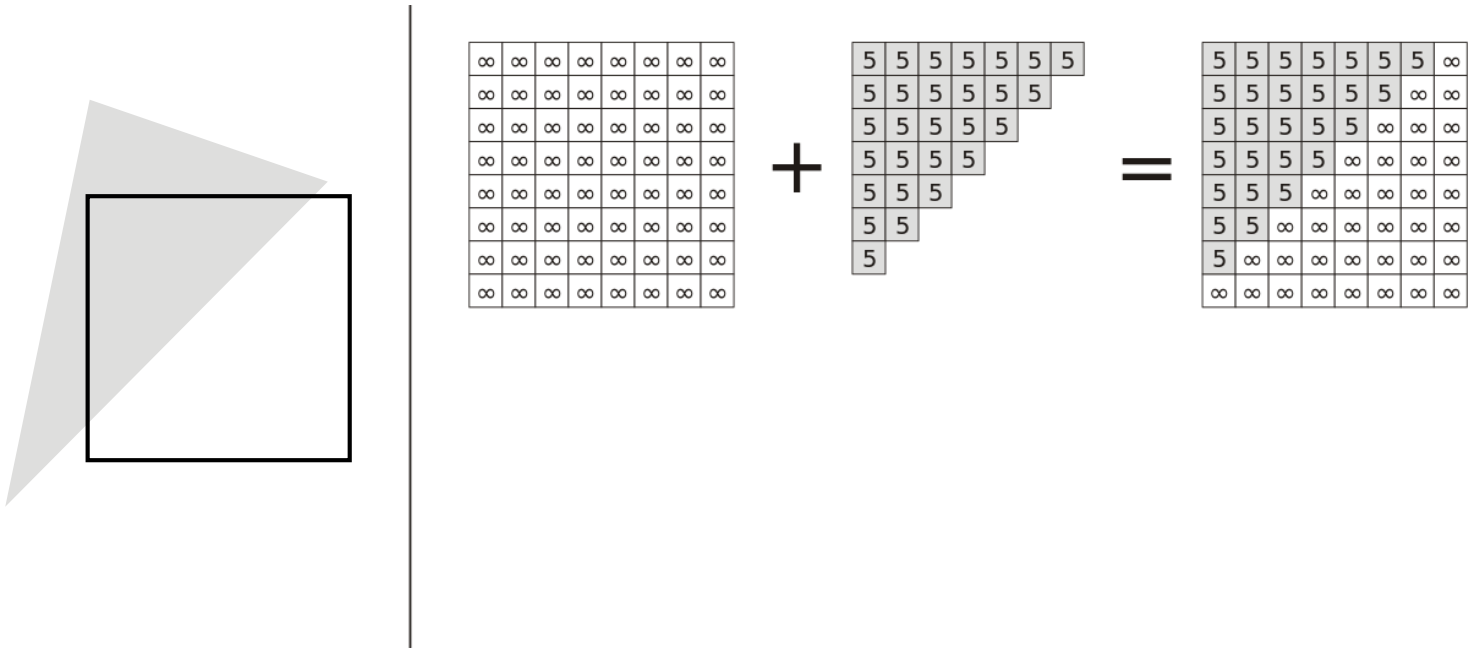
- Manchmal gibt es keine korrekte Reihenfolge
  - durchdringende Flächen
  - zyklische Verdeckung
  - große Polygone





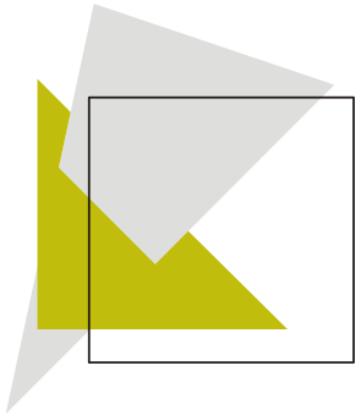
Z-Buffer

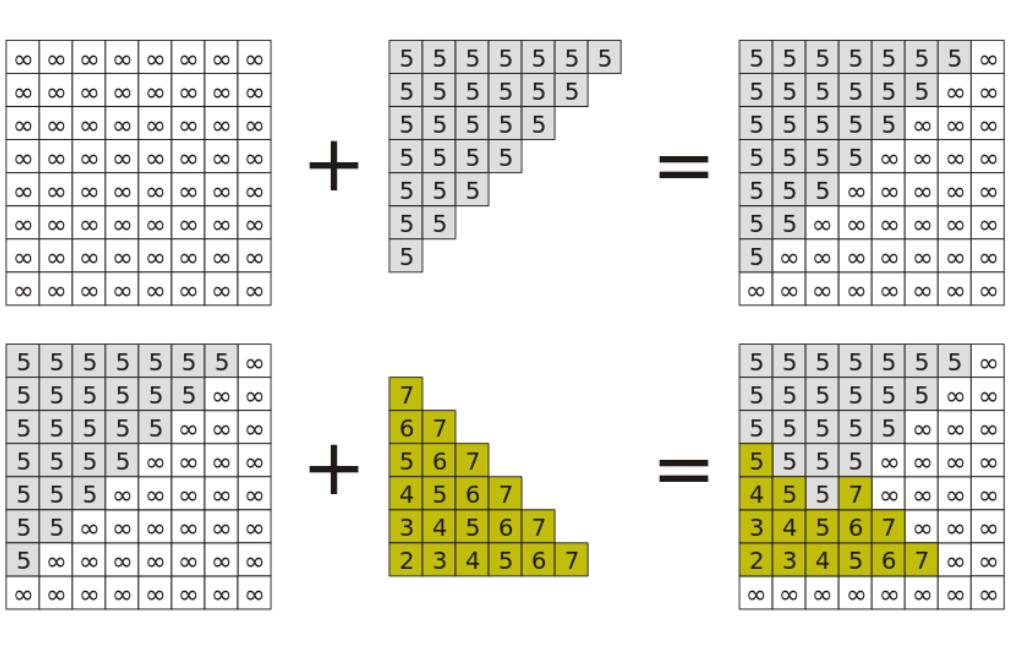
- Dissertation Wolfgang Straßer / TR Ed Catmull, 1974
- Speichere minimale Tiefe pro Pixel
- Bei neuem Fragment, überschreibe Pixel nur falls neue Tiefe geringer als alte Tiefe
- In diesem Fall speichere neue Tiefe



Z-Buffer

- Dissertation Wolfgang Straßer / TR Ed Catmull, 1974
- Speichere minimale Tiefe pro Pixel
- Bei neuem Fragment, überschreibe Pixel nur falls neue Tiefe geringer als alte Tiefe
- In diesem Fall speichere neue Tiefe





Z-Buffer / Depth Buffer in OpenGL

- Z-Test aktivieren:
```
glEnable(GL_DEPTH_TEST);
```

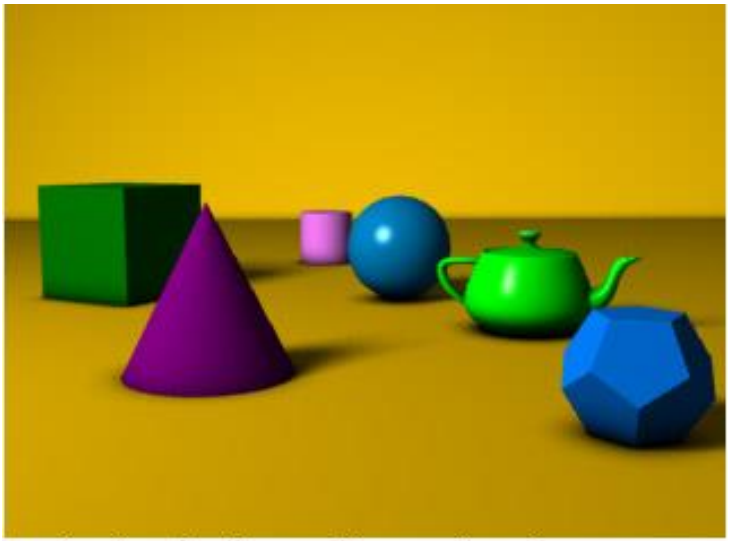
- Fragmente mit geringerer Tiefe sollen den Test bestehen
```
glDepthFunc(GL_LESS);   // GL_ALWAYS, GL_NOTEQUAL, ...
```

- Z-Buffer löschen (um das nächste Bild zu rendern):
```
glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
```

- Z-Buffer beim Rendern nicht updaten (z.B. zum Rendern transluzenter Objekte über eine vorhandene opake Szene):
```
glDepthMask(GL_FALSE);
```

Vorteile Z-Buffer

- Unabhängig von Szenenkomplexität
- Kann auch komplexe Durchdringung korrekt darstellen



A simple three dimensional scene

Z-buffer representation

Auflösungs-/Genauigkeitsprobleme Z-Buffer

- In der Praxis werden Tiefenwerte gern in vorzeichenlose Integers gespeichert (schneller Vergleich)
- Tiefenwerte $\{0,1,\dots,B-1\}$
  - $0 \rightarrow$ Near clipping plane
  - $B-1 \rightarrow$ Far clipping plane

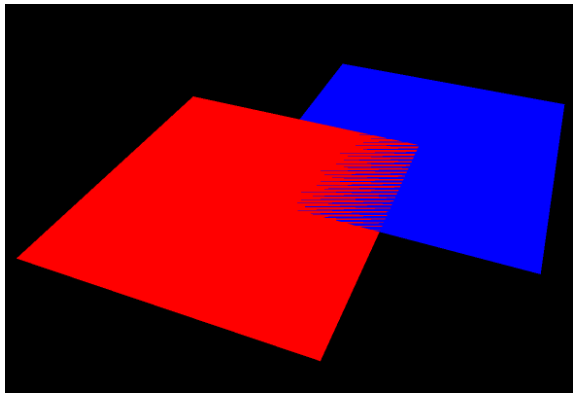
- Tiefe wird auf diskrete Werte gerundet
  - Jeder Wert deckt einen Wertebereich der Länge $\Delta z = \frac{f-n}{B}$ ab
- Bei b Bit Wordbreite für Z-Buffer-Werte ist $B = 2^b$
  - Minimiere $f-n$ für maximale Auflösung

Z-Buffer Auflösung

- Tiefenwerte können einen immensen Bereich abdecken
- Millimeter bis hunderte Kilometer

Z-Fighting

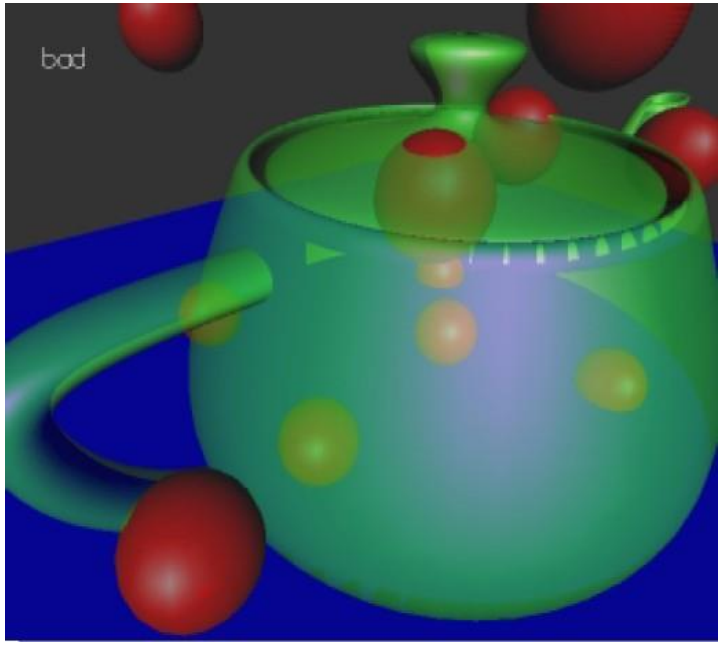
- Bei unzureichender Auflösung oder exakt koplanaren Oberflächen hängt es wieder von der Reihenfolge ab, wer gewinnt:

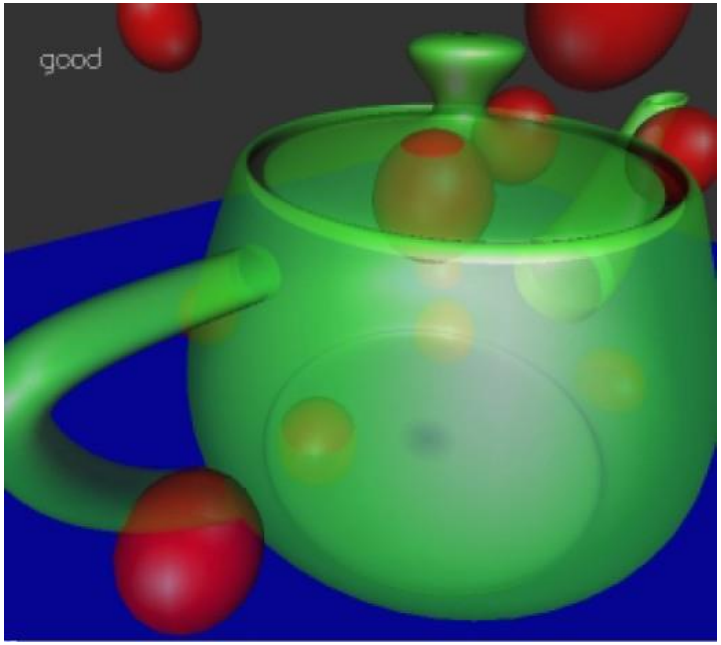


- Dies nennt man "Z-Fighting".
- Mögliche Abhilfen:
  - Versetze koplanare Polygone geringfügig
  - Möglichst hohe Auflösung (24-bit oder 32-bit integer)
  - Möglichst geringer Abstand zwischen near- und far-plane
  - Getrennte Z-Buffer für Vordergrund und Hintergrund

Order-Independent Transparency

- Wie kann ich sicherstellen, dass Transparenzeffekte korrekt dargestellt werden?
- Ohne zu sortieren





A-Buffer

(Accumulation Buffer)

- Entwickelt für Reyes Renderer (Lucasfilm/Pixar), ca. 1985
- Speichere Fragmentliste pro Pixel
- Zudem weitere Informationen für Antialiasing, Mikropolygone, ...
- Für Multisample-Antialiasing geeignet, jedoch sehr speicherintensiv (typ. 10GB für 1080p Rendering)
- Effektiv wird doch sortiert, nur an anderer Stelle

Depth peeling [Everitt 2001]

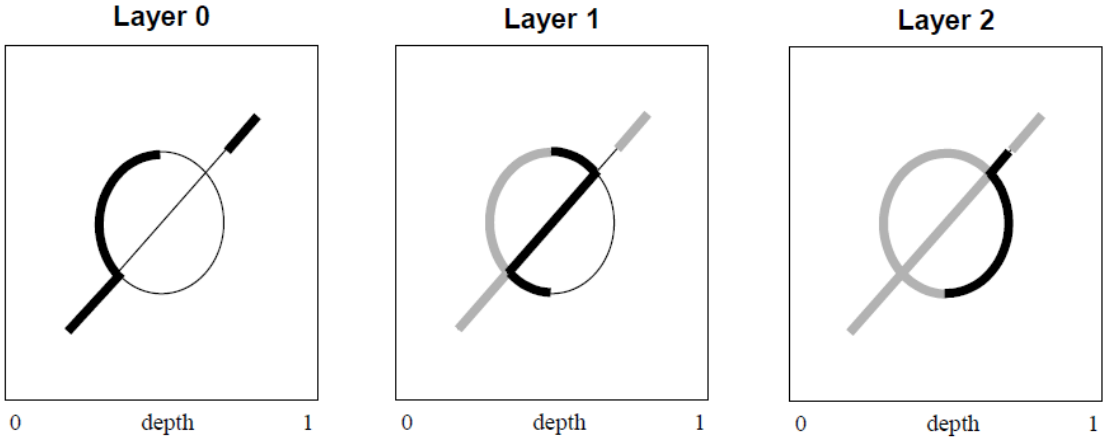
- Rendere Bild mehrere Male
- Verwende zwei Z-Buffer
  - Buffer 1: konventioneller Z-Buffer (maskiert; testet `GL_LESS`)
  - Buffer 2: statischer Buffer für Mindestdistanz (als Shader implementiert)

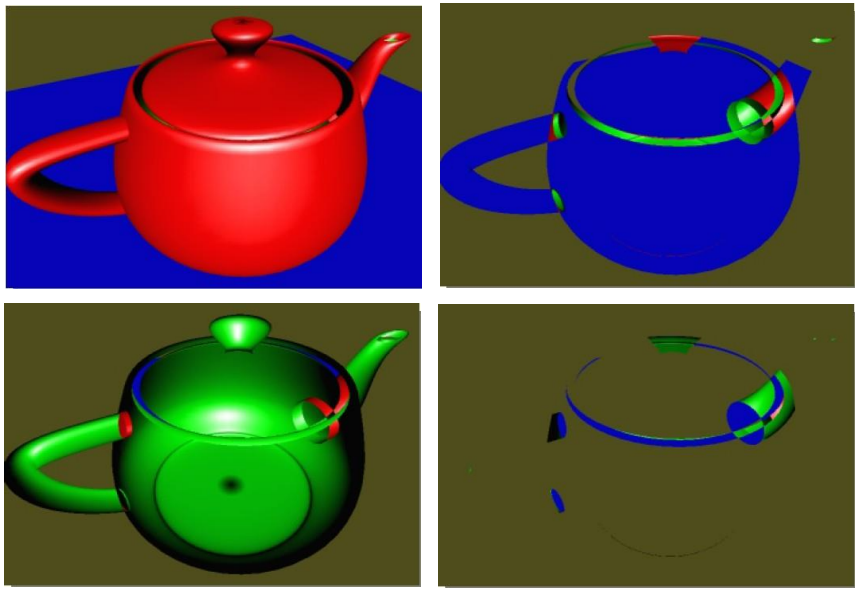
Fragment Shader:

```
uniform sampler2D depthTexture;
in vec3 pos;
void main() {
    if (pos.z <= texture(depthTexture, vec2(pos.x, pos.y)))
        discard; // Entspricht manuellem GL_GREATER Test pro Pixel
    // Rendere Fragment unter bestehendes Bild (“front-to-back” Regel)
}
```

Nach dem Renderpass: ersetze depthTexture durch letzten Z-Buffer und wiederhole

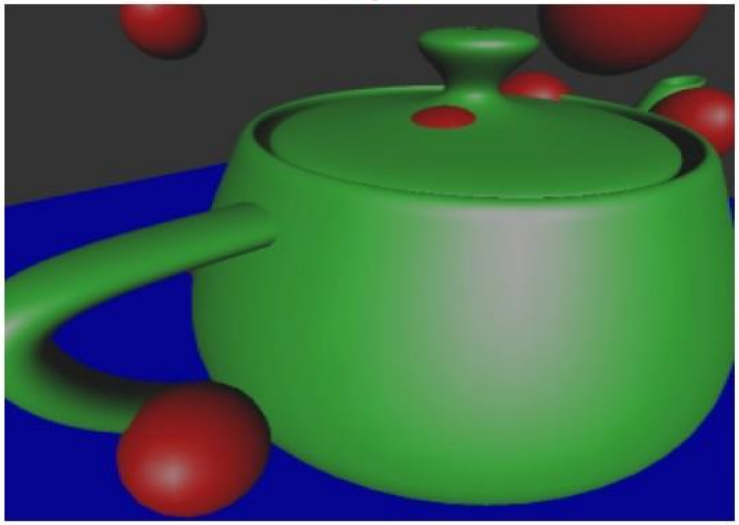
Depth Peeling



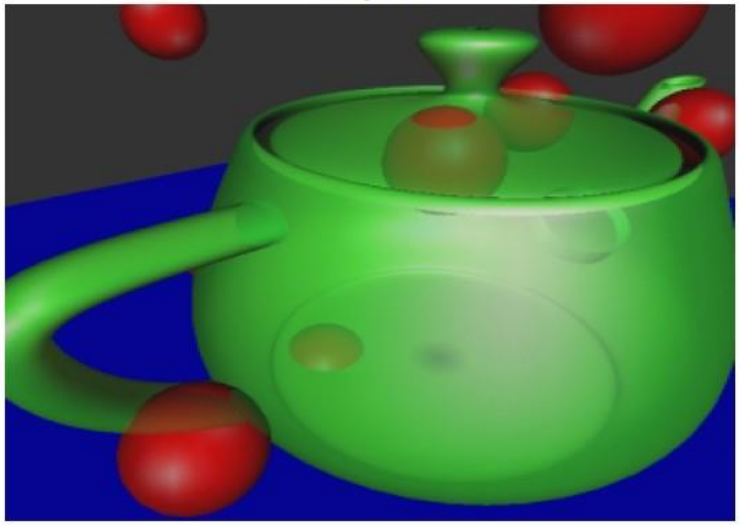


Depth Peeling

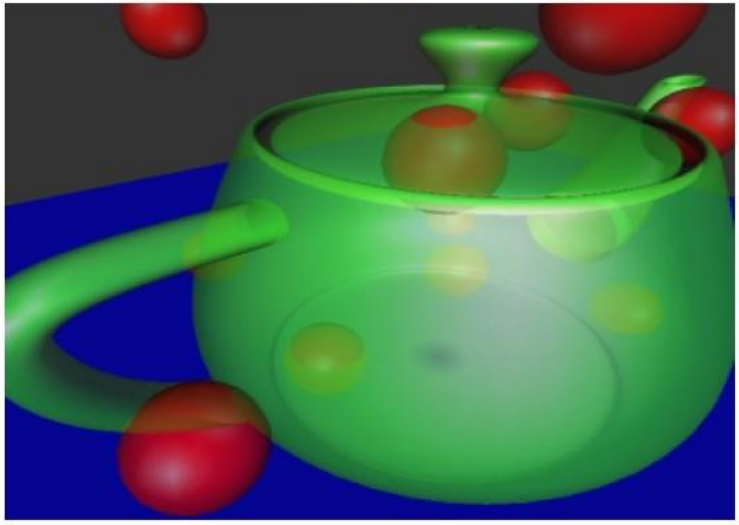
1 layer



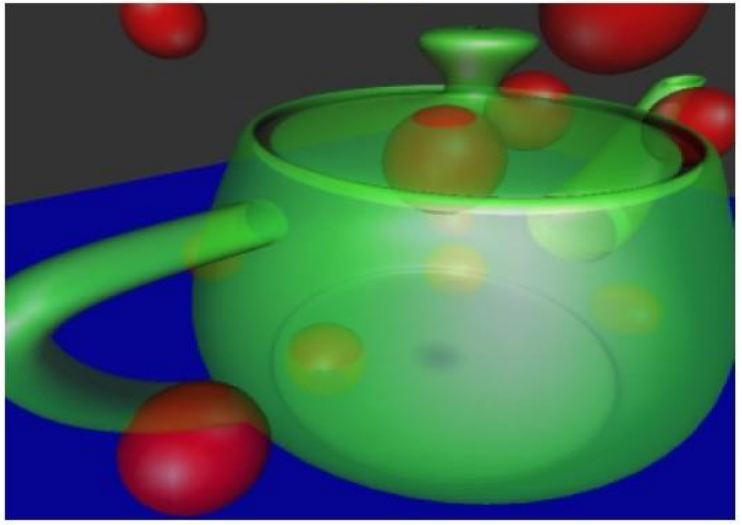
2 layers



3 layers



4 layers

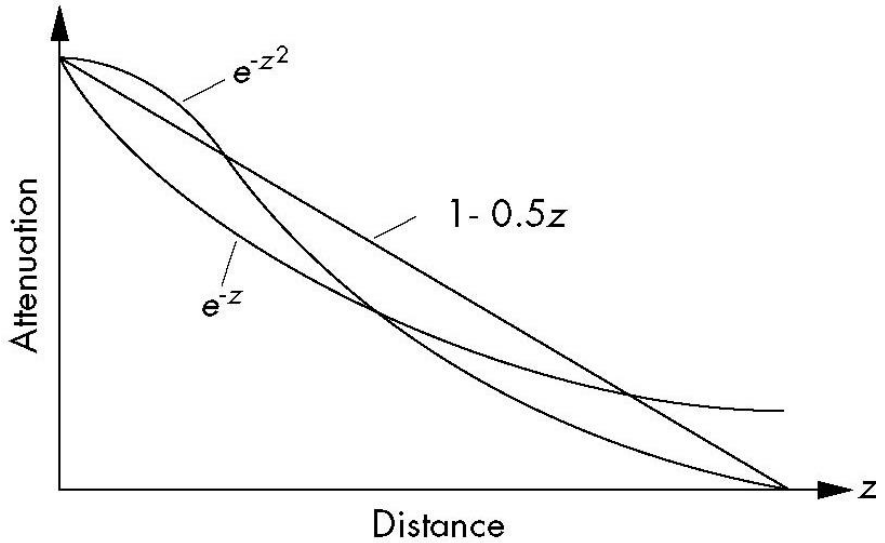


Nebel

- Mische mit fester Farbe; Faktor hängt von Tiefenwert ab
  - Simuliert einen Nebeleffekt (fog)
- Mische Quellfarbe C_s und Nebelfarbe C_f mittels

$$C_s' = f\, C_s + (1-f)\, C_f$$

- f ist der *fog factor*
  - Exponentiell (`GL_EXP`)
  - Gauß (`GL_EXP2`)
  - Linear (`GL_LINEAR`)
- `glEnable(GL_FOG);`



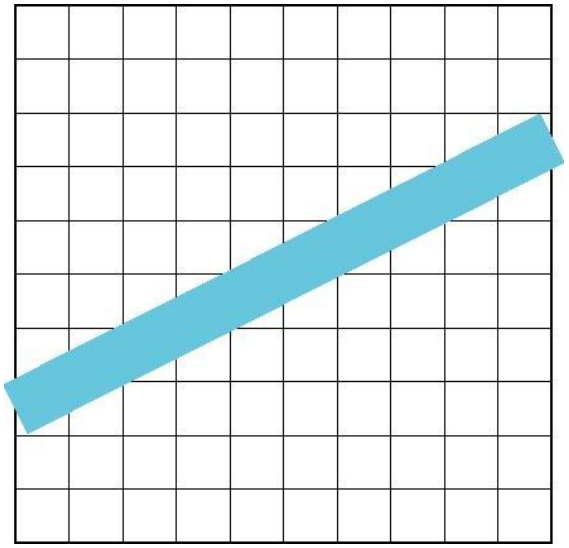
- Relikt aus der Fixed-Function-Pipeline; seit OpenGL 3.2 “deprecated”

Compositing und HTML

- In OpenGL (Desktop) hat die A-Komponente keine Auswirkung, außer wenn Blending aktiviert ist
- In WebGL hat jedes A außer 1.0 eine Auswirkung, weil WebGL mit dem HTML5 Canvas-Element zusammenarbeitet
- So können andere Anwendungen mit der WebGL-Grafik vermischt werden

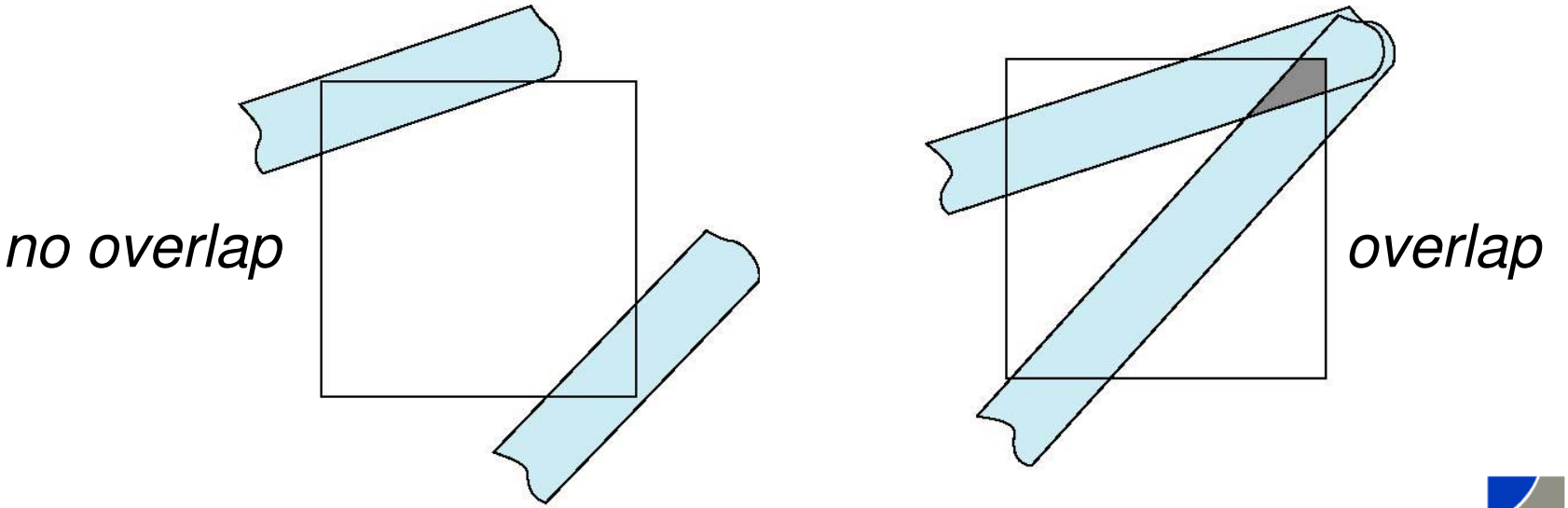
Linien-Aliasing

- Eine ideale Rasterlinie ist einen Pixel breit
- Alle Liniensegmente außer speziellen vertikalen und horizontalen Segmenten decken Pixel teilweise ab
- Einfache Algorithmen
  färben nur ganze Pixel ein
- Treppeneffekt oder Aliasing
- Ähnliche Probleme mit
  Polygonen



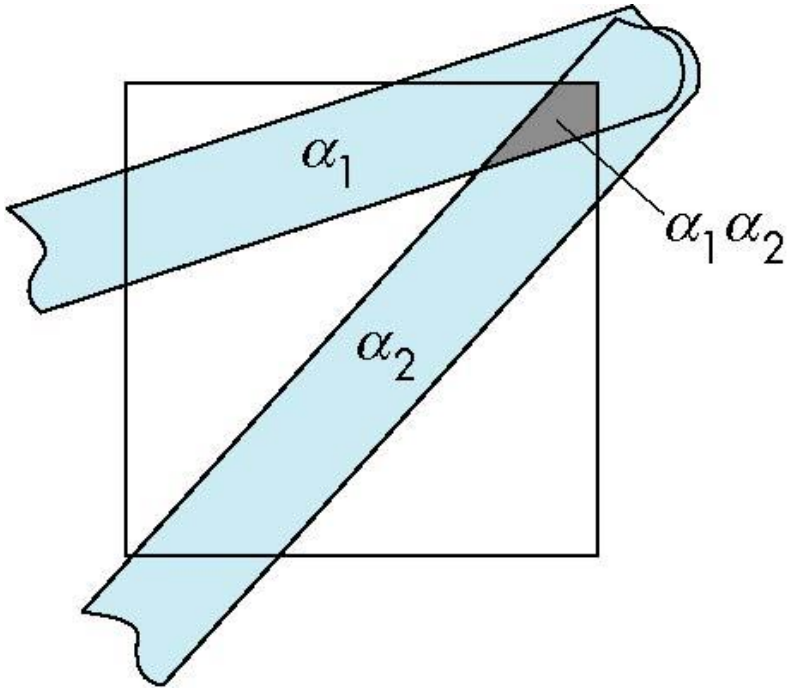
Antialiasing

- Versuche, einen Pixel teilweise einzufärben: addiere einen Bruchteil seiner Farbe zum Framebuffer
  - Bruchteil hängt davon ab, welcher Anteil des Pixels vom Fragment abgedeckt wird
  - Dieser Bruchteil hängt davon ab, ob sich Linien überlappen



Area Averaging

- Verwende Fläche $\alpha_1+\alpha_2-\alpha_1\alpha_2$ als Mischfaktor



OpenGL Antialiasing

- Noch sehr lückenhaft in WebGL
- Separat aktivierbar für Punkte, Linien, Polygone

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- In Desktop-OpenGL ist Antialiasing auf den meisten GPUs automatisch aktiviert