h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Visual Computing

Grafische Objekte und deren Programmierung

Hochschule Darmstadt

Prof. Dr. Elke Hergenröther
Björn Frömmer
Prof. Dr. Benjamin Meyer

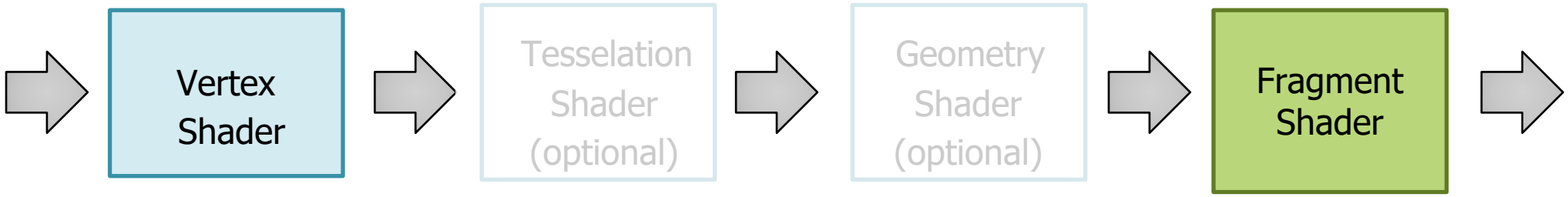
KAPITEL 8

Rendering Pipeline

Shaderpipeline (Wiederholung)

- Die hochgeladenen Daten können vor dem Rendern noch manipuliert werden.
- Die gewünschten Berechnungen werden in sog. **Shadern** implementiert, kleinen Programmfragmenten, die direkt auf der GPU ausgeführt werden können.

- Computersprache der Shaderprogramme: GLSL (Open**GL S**hading **L**anguage)
  - Alternativen: HLSL für reine Windows/DirectX-Entwicklung, Cg (von NVidia, 2012 eingestellt)

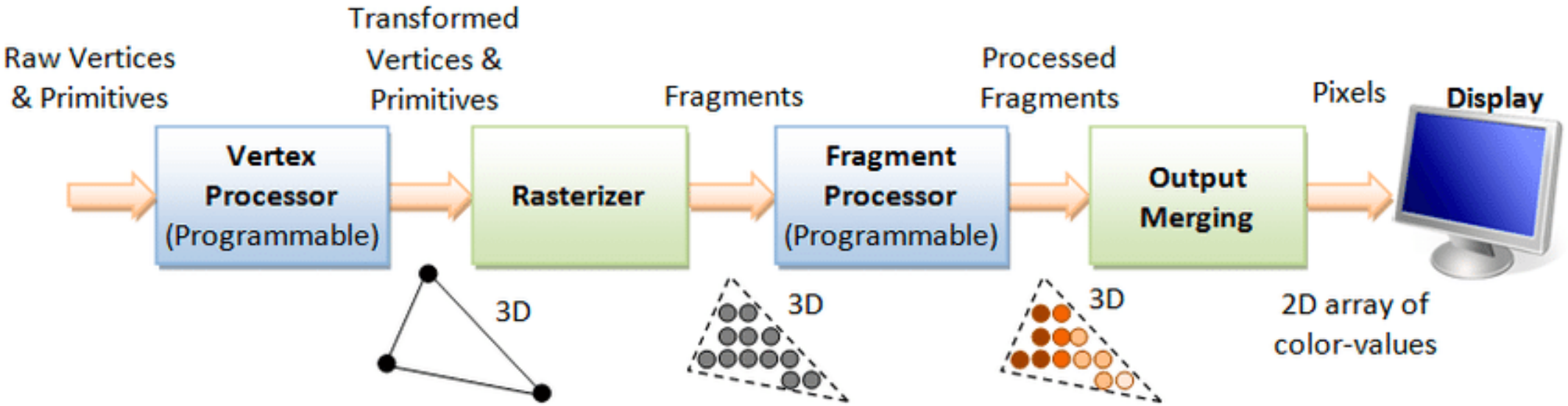


- Jeder Shader hat hierbei eine genau definierte Aufgabe:
  - **Vertex Shader:** Berechnet die **finale Position** jedes Vertex im Ausgabebild
  - **Fragment Shader:** Berechnet die **Farbe jedes Pixels** im Ausgabebild
    - (Tesselation Shader: Zerlegung der Objekte in feinere Dreiecksnetze)
    - (Geometry Shader: Veränderung der Geometrie-Daten zur Laufzeit)

Tradeoff Speicherplatz versus Performance

Rendering Pipeline

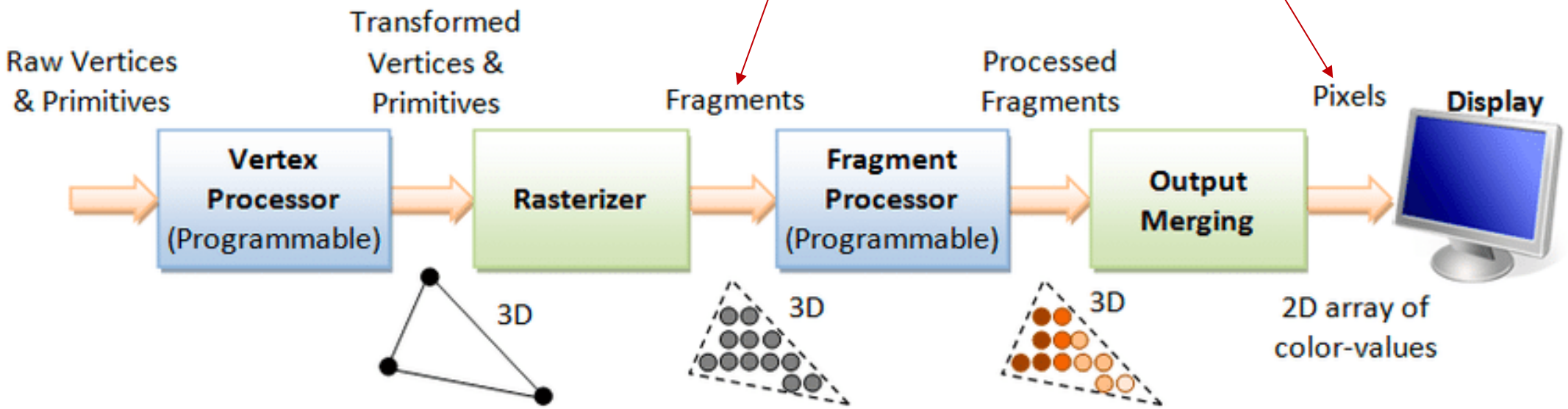
- Das Positionieren der Objekte in der Szene mittels der akkumulierten Transformationsmatrix und die Integration einer (virtuellen) Kamera sind die ersten Schritte einer ganzen Kette von Berechnungen
- Einzelne Berechnungen werden dabei automatisch auf der GPU ausgeführt oder können per OpenGL-Befehl aktiviert/deaktiviert werden.



Quelle: https://www.researchgate.net/publication/334129575_Efficient_Spatial_Anti-Aliasing_Rendering_for_Line_Joins_on_Vector_Maps

Rendering Pipeline

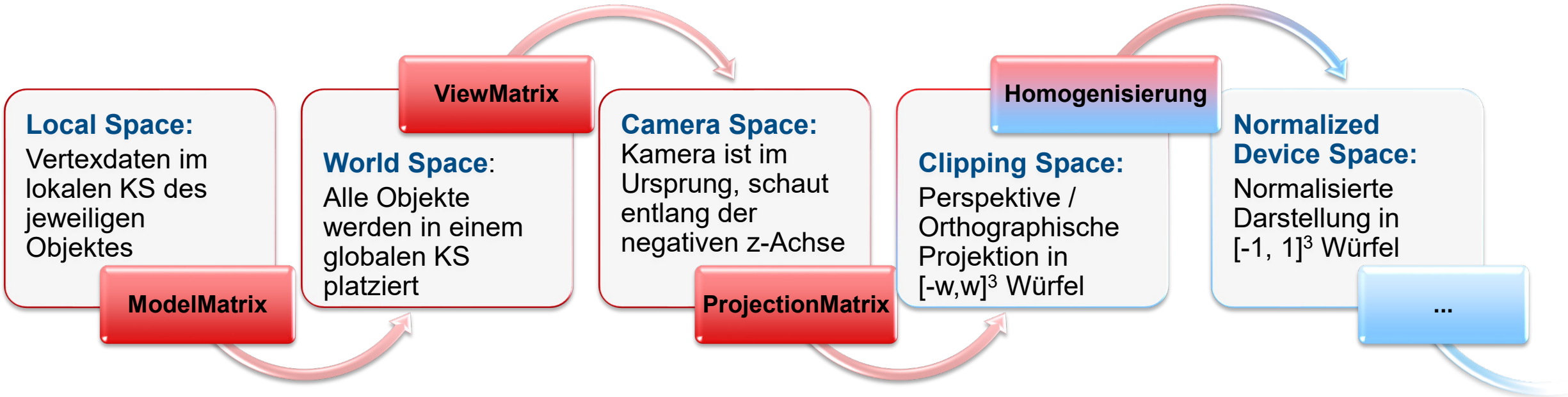
Unterschied zwischen Fragment und Pixel: Ein Pixel ist ein Bildschirmelement. Das Fragment ist die Vorstufe dazu. Manchmal können auch mehrere Fragmente ein Pixel ergeben (Transparenz) und nicht alle Fragmente werden zum Pixel (Z-Buffer).



Quelle: https://www.researchgate.net/publication/334129575_Efficient_Spatial_Anti-Aliasing_Rendering_for_Line_Joins_on_Vector_Maps

Rendering Pipeline

Homogenisierung kann manuell im Vertex Shader vorgenommen werden!

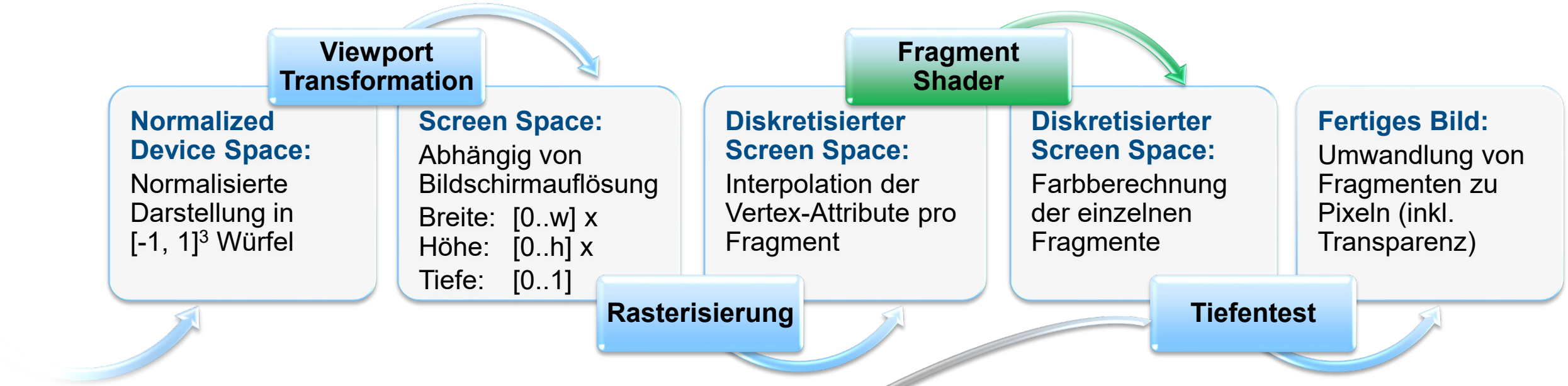


Selbstimplementierte Berechnungen im Vertex Shader!

Automatisierte Berechnungen auf der Grafikkarte!

KS = Koordinatensystem

Rendering Pipeline II

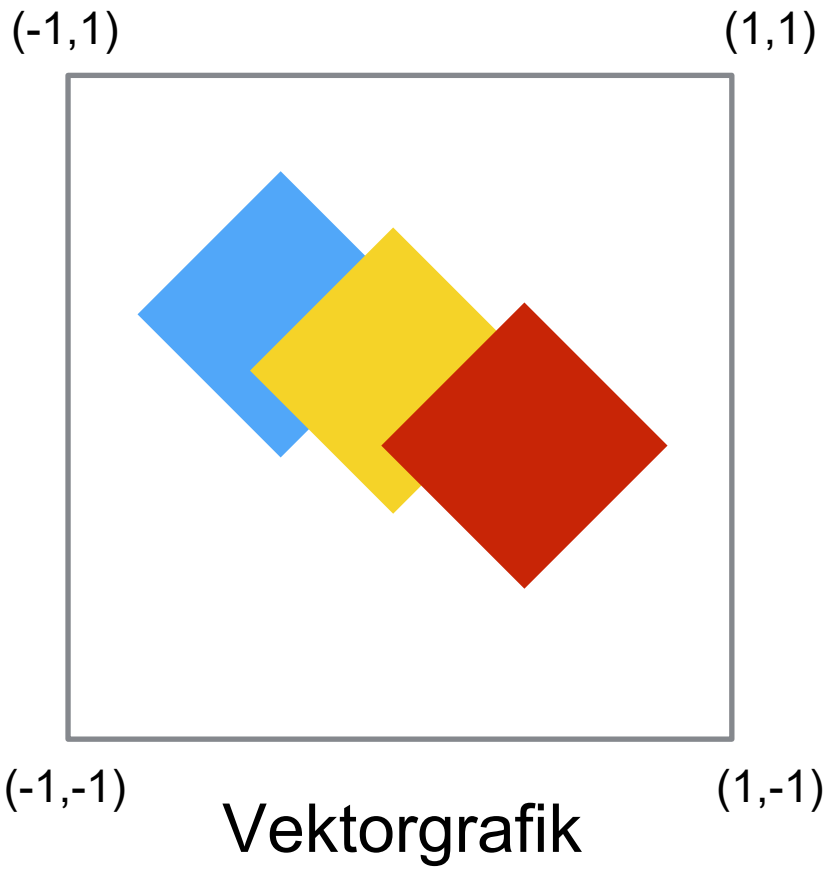


Als Optimierung kann u.U. der Tiefentest auch *vor* dem Fragment Shader durchgeführt werden

Automatisierte Berechnungen auf der Grafikkarte!

Viewport Transformation

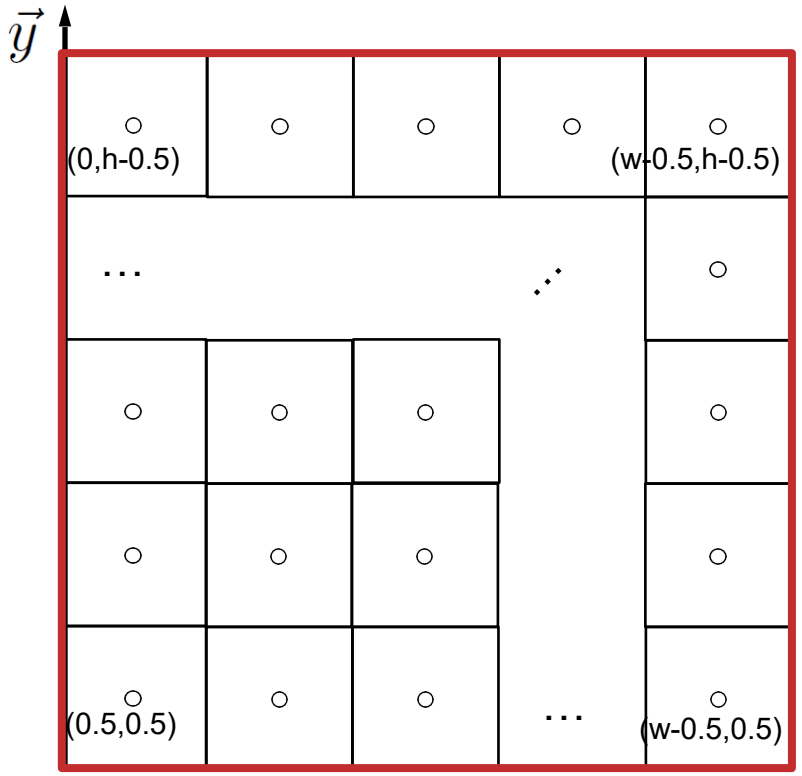
Viewport Transformation



Viewport Transformation

Viewport Transformation

- Bekannt: Bildschirmauflösung Breite (*w)* x Höhe (*h)*
- Wie würden Sie eine Abbildung vom Normalized Device Space ($[-1...1]^3$) in den Viewport [0...w] x [0...h] x [0 ...1] berechnen?



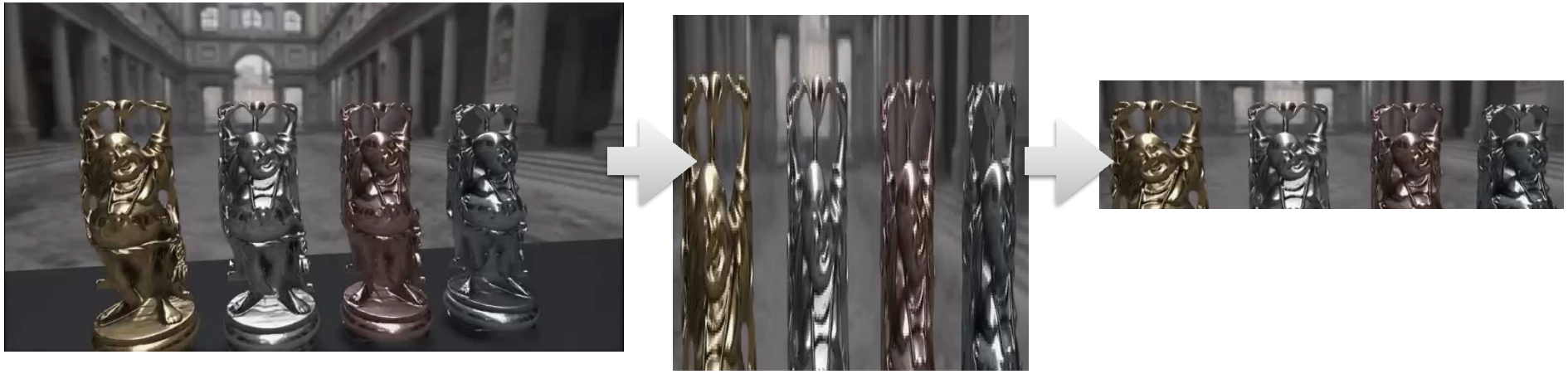
$$p'_x = \frac{1}{2}(p_x + 1) \cdot w$$
$$p'_y = \frac{1}{2}(p_y + 1) \cdot h$$

Der Tiefenwert *z* wird abgebildet auf [0..1]:

$$p'_z = \frac{1}{2}(p_z + 1)$$

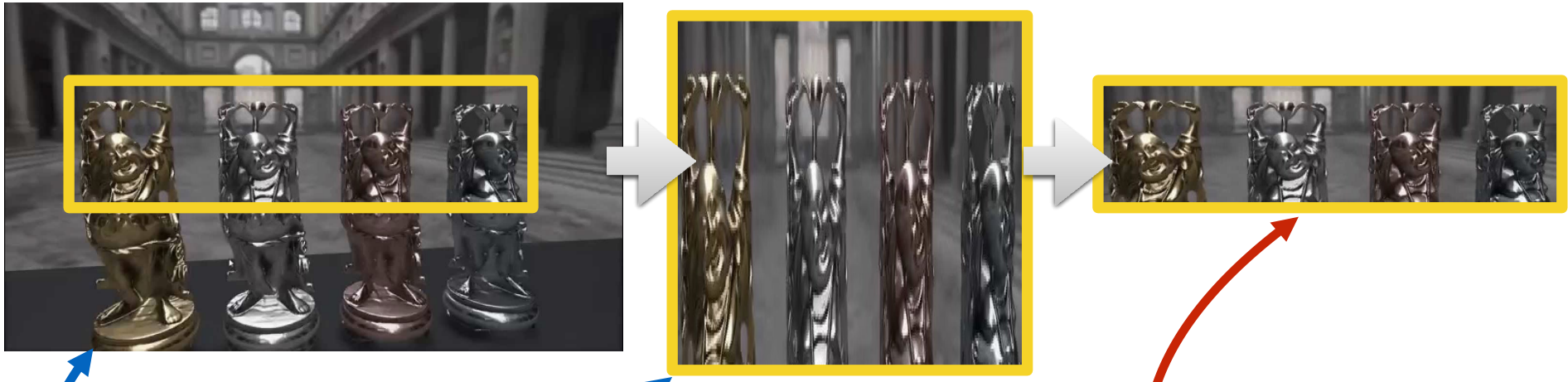
Viewport Transformation – Beispiel

- Was passiert hier?



Viewport Transformation – Beispiel

- Was passiert hier?



Definition des zu rendernden Volumens

Skalierung der Szene sodass dieses Volumen im Einheitswürfel $[-1,1]^3$ liegt

Viewport Transformation transformiert Vektorgrafik in Fensterkoordinaten

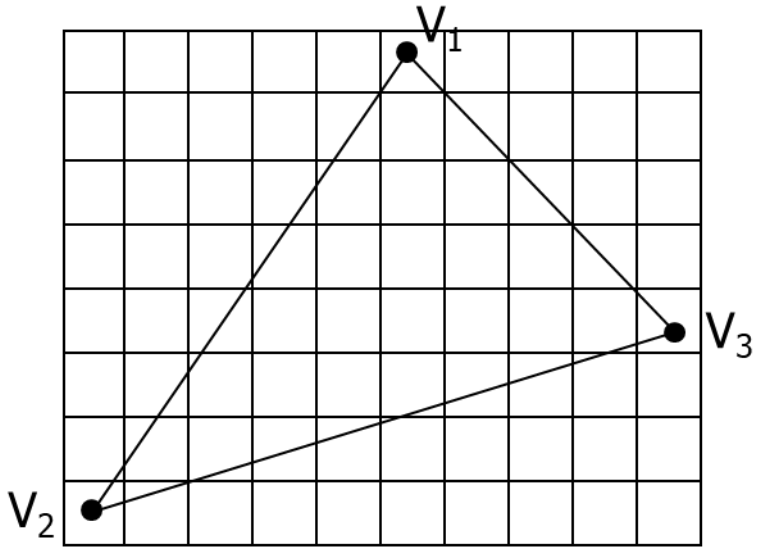
Auflösungsunabhängig!

Nicht auflösungsunabhängig!
Abhängig vom Ausgabemedium
(Laptop, HD Monitor, Smartphone, etc.)

Rasterisierung

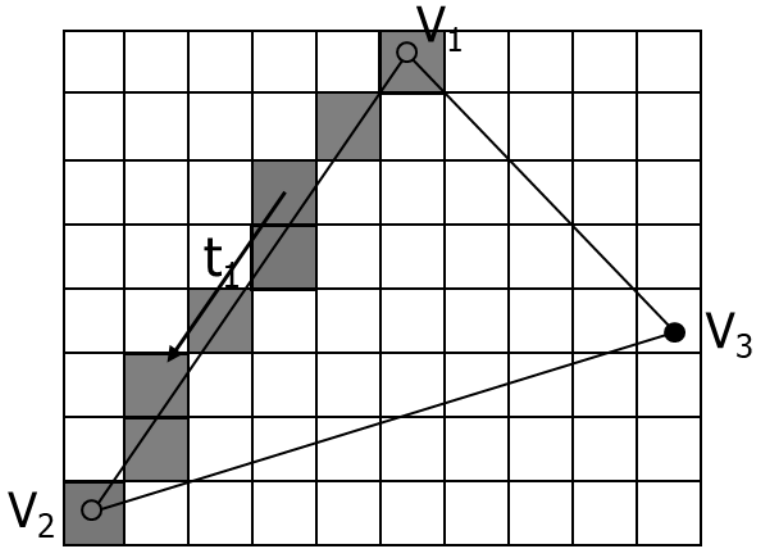
Rasterisierung

1. Schritt: Bestimmung der Pixel-Koordinaten der Vertices



Rasterisierung

1. Schritt: Bestimmung der Pixel-Koordinaten der Vertices
2. Schritt: Interpolation der Vertex-Attribute entlang der Dreieckskanten

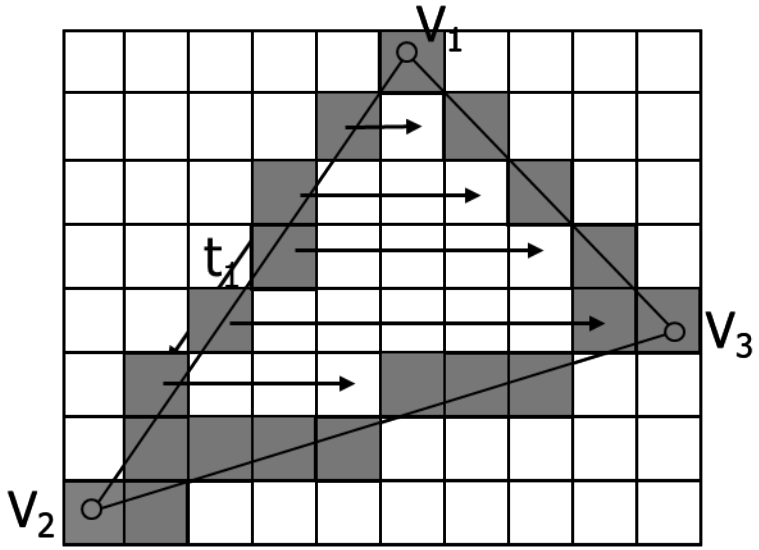


Interpolation zwischen V_1 und V_2:

- Zunächst muss die Strecke zwischen den beiden Vertices normiert werden.
- Der Parameter t_1 steht für die normierte Strecke und geht von 0 bis 1.
- Danach erfolgt eine lineare Interpolation abhängig von t_1.

Rasterisierung

1. Schritt: Bestimmung der Pixel-Koordinaten der Vertices
2. Schritt: Interpolation der Vertex-Attribute entlang der Dreieckskanten
3. Schritt: Interpolation der Pixelwerte entlang der einzelnen Rasterzeilen

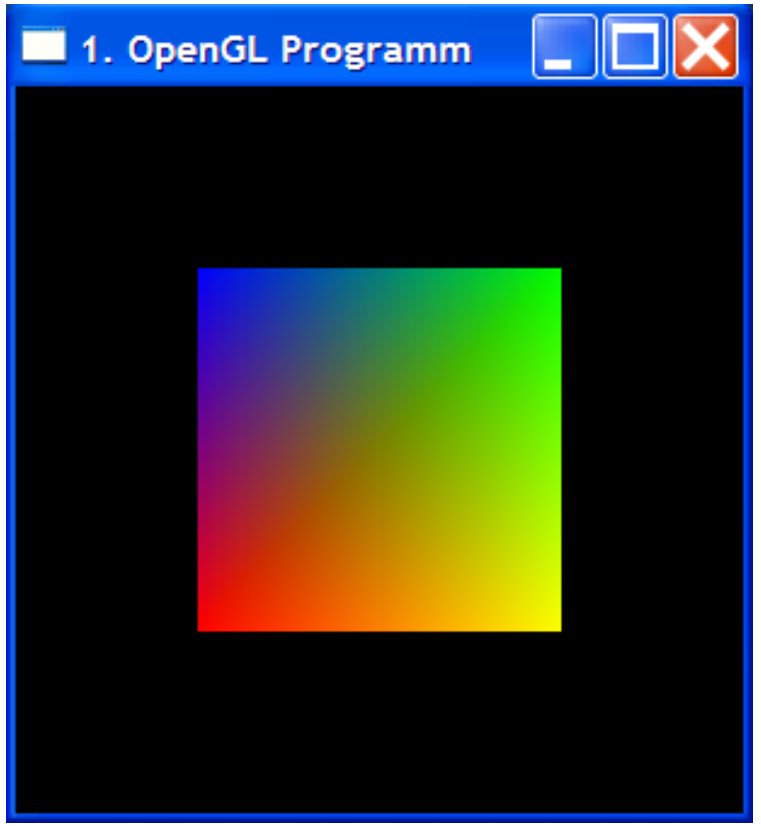


Visualisierung ohne Berücksichtigung von Licht

- Wie berechnet man den Farbverlauf innerhalb einer Linie oder eines Dreiecks, wenn jedem Punkt eine andere Farbe zugeordnet wurde?

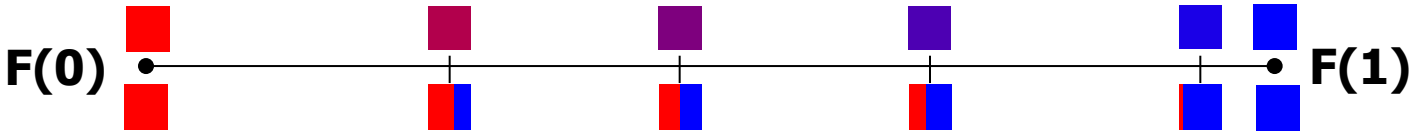
- **Lineare Interpolation:**
  Interpolationsverfahren für Linien

- **Bilineare Interpolation**:
  Interpolationsverfahren für Flächen



Lineare Farbinterpolation

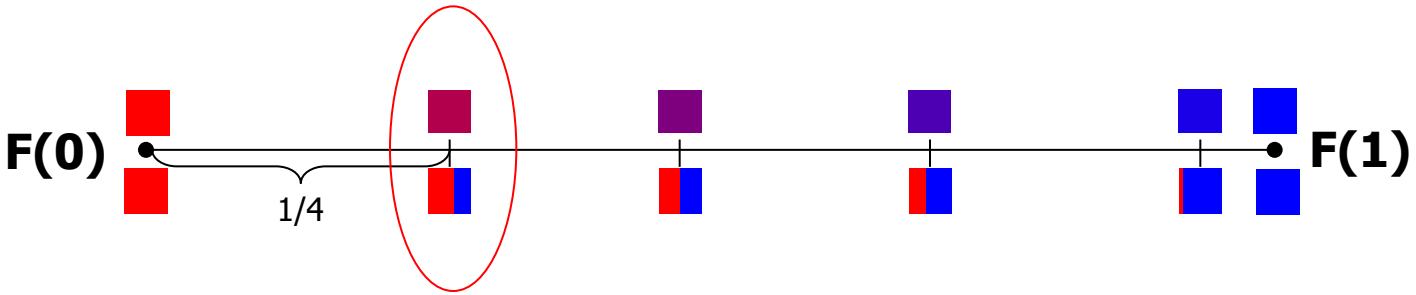
Verhältnis der „Farbanteile“ von F(0) und F(1)



$$F(t) = (1-t) \cdot F(0) + t \cdot F(1)$$

$$F(t) = (1-t) \cdot \begin{pmatrix} R(0) \\ G(0) \\ B(0) \end{pmatrix} + t \cdot \begin{pmatrix} R(1) \\ G(1) \\ B(1) \end{pmatrix}$$

Lineare Farbinterpolation – Beispiel



$$F(t) = (1-t)\cdot F(0) + t\cdot F(1)$$

$$F(t) = (1-\frac{1}{4})\cdot F(0) + \frac{1}{4}\cdot F(1)$$

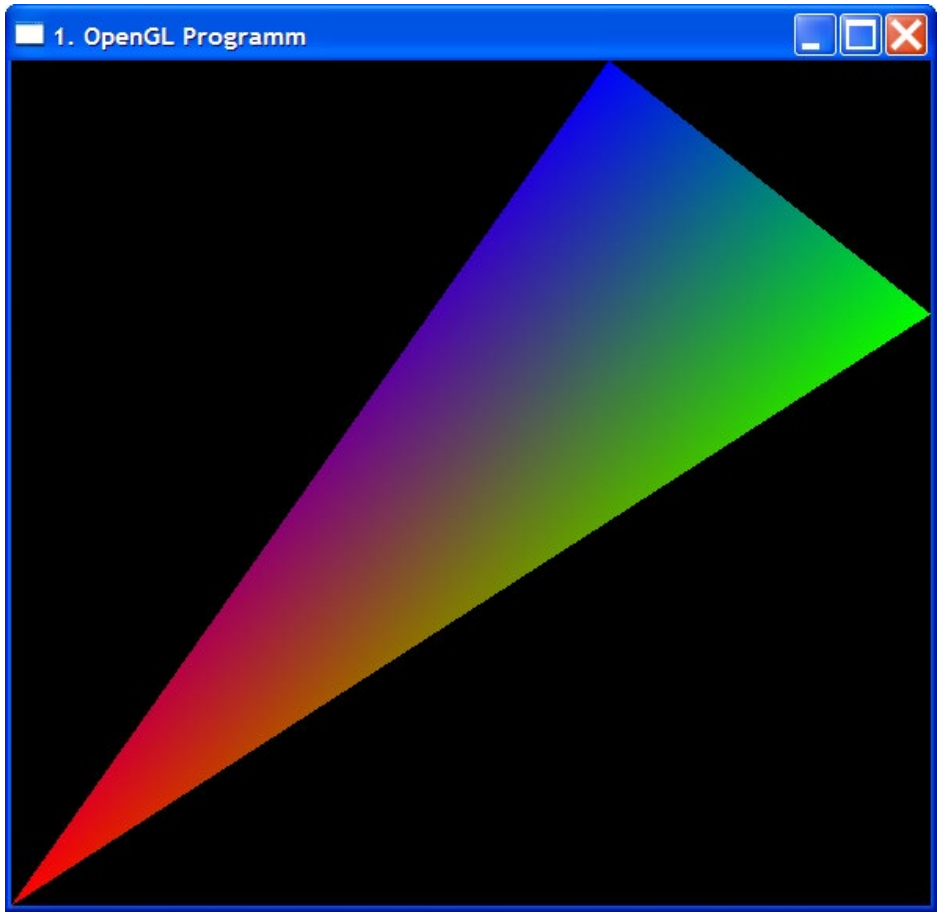
$$F(t) = \frac{3}{4}\cdot\begin{pmatrix}1\\0\\0\end{pmatrix} + \frac{1}{4}\cdot\begin{pmatrix}0\\0\\1\end{pmatrix} = \begin{pmatrix}\frac{3}{4}\\0\\\frac{1}{4}\end{pmatrix}$$

Auf die gleiche Weise werden alle Vertex-Attribute interpoliert!

Vertex-Shader Fragment Shader

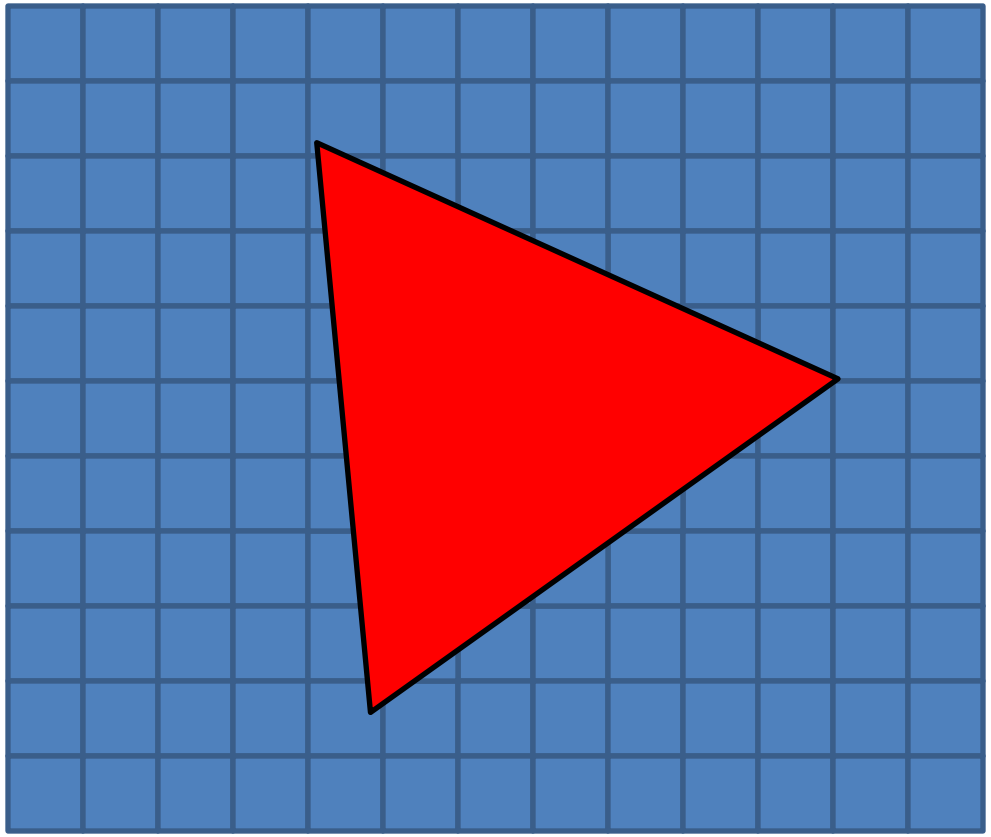
```
out TYPE name;  --->  in TYPE name;
```

Ergebnis einer bilinearen Farbinterpolation



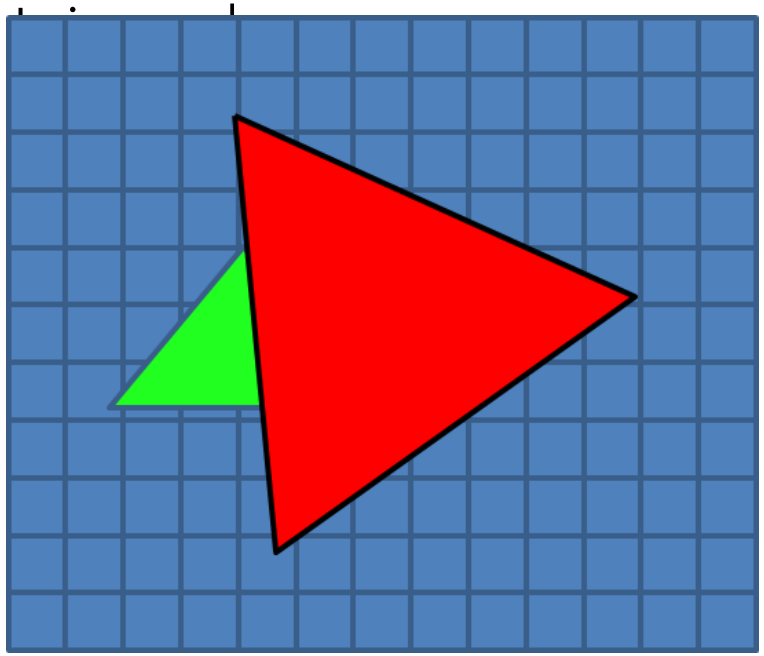
Tiefentest

Was ist sichtbar?

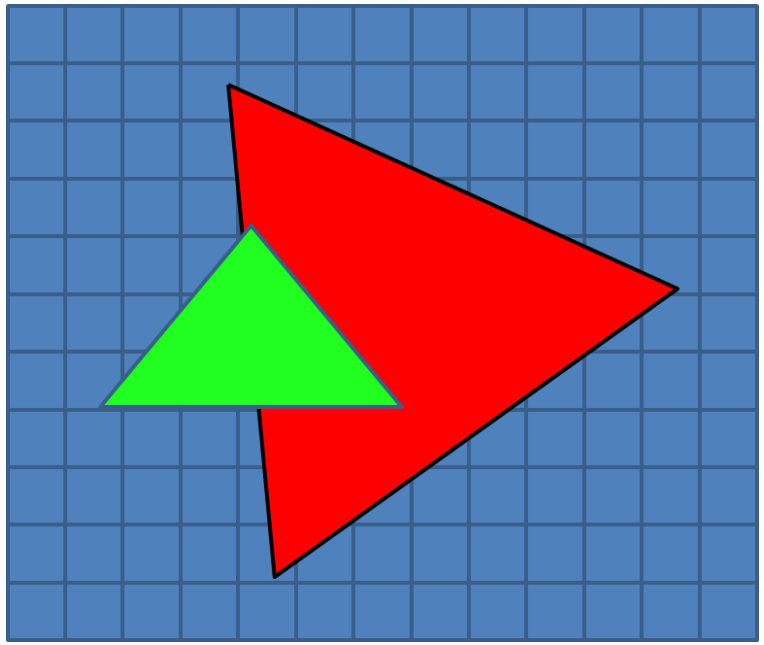


Was ist sichtbar?

- → Darstellung hängt von der Reihenfolge der Renderbefehle ab!
  - Dieses Vorgehen wird als **Painters Algorithm** bezeichnet, ähnlich einer klassischen



Grünes Dreieck zuerst gerendert

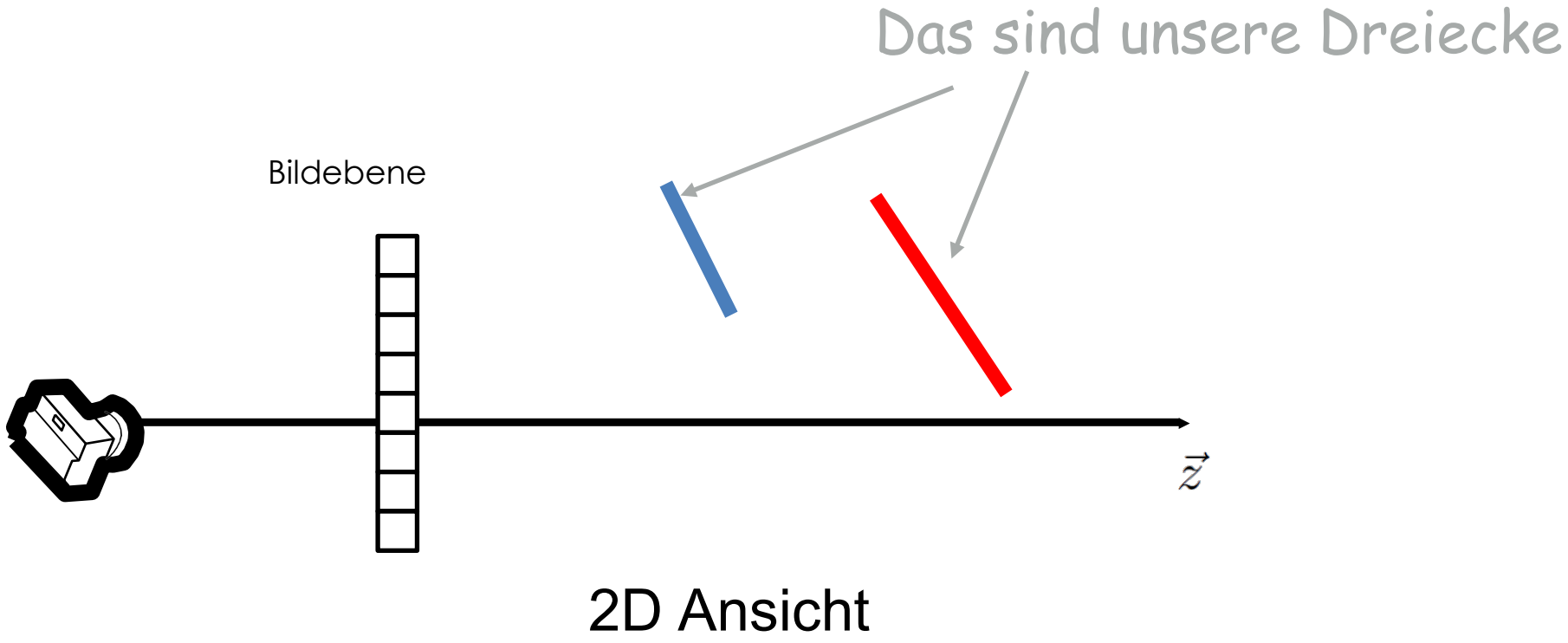


Rotes Dreieck zuerst gerendert

- **Besser:** Verdeckung abhängig vom Abstand des jeweiligen Pixels zur Kamera

Tiefentest mit z-Buffer

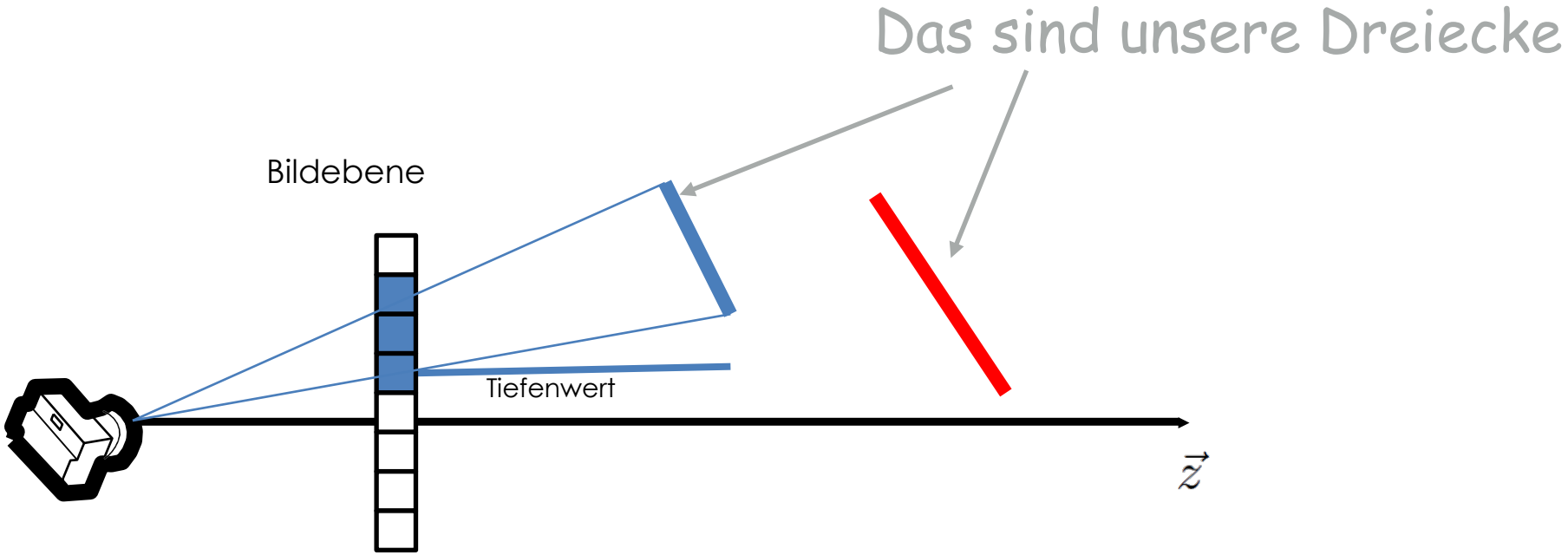
- **Tiefentest:** Sortierung der Fragmente nach Tiefenwert soll vermieden werden
- **Z-Buffer:** Daher wird pro Fragment die Entfernung zur Kamera (= Z-Koordinate) gespeichert



2D Ansicht

Tiefentest mit z-Buffer

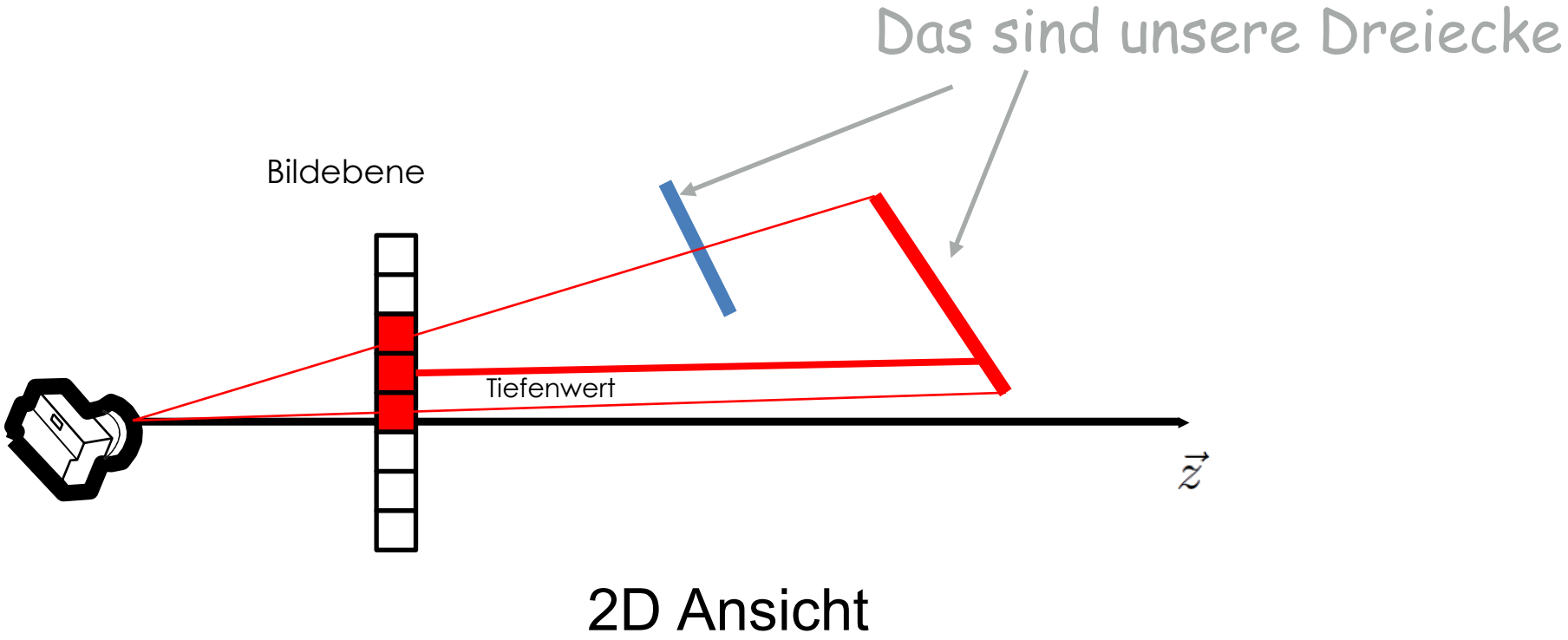
- **Tiefentest:** Sortierung der Fragmente nach Tiefenwert soll vermieden werden
- **Z-Buffer:** Daher wird pro Fragment die Entfernung zur Kamera (= Z-Koordinate) gespeichert



2D Ansicht

Tiefentest mit z-Buffer

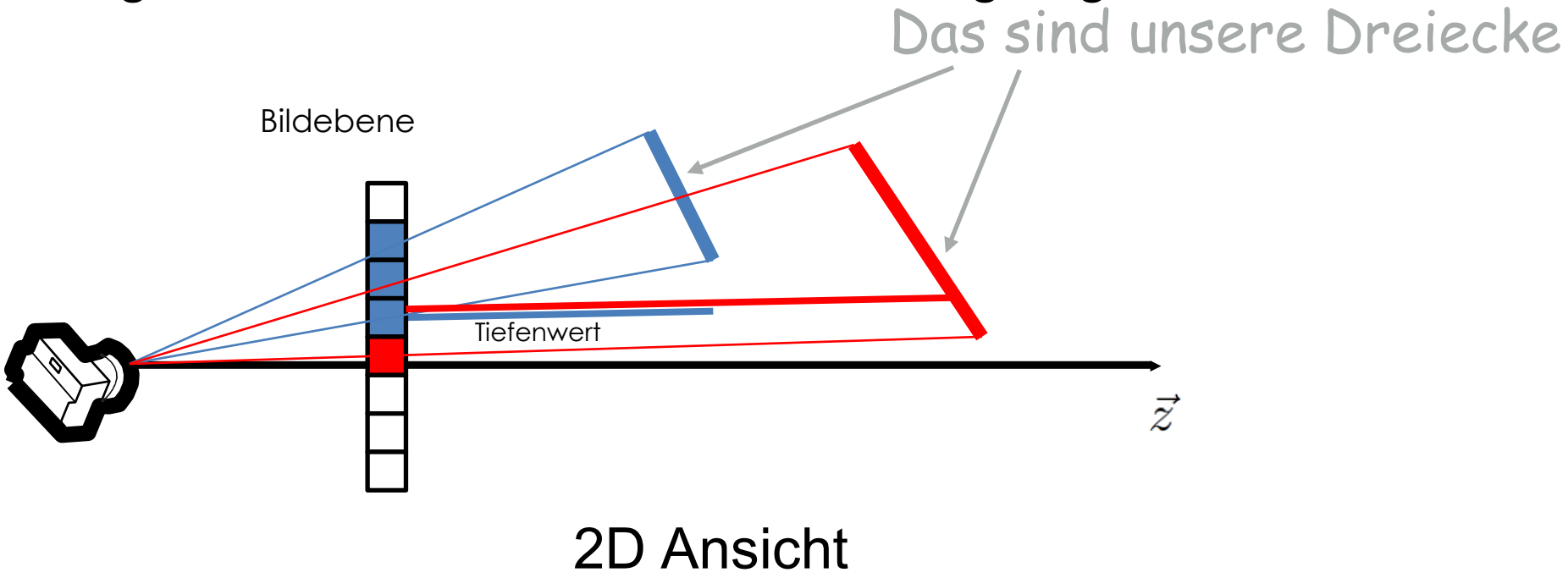
- **Tiefentest:** Sortierung der Fragmente nach Tiefenwert soll vermieden werden
- **Z-Buffer:** Daher wird pro Fragment die Entfernung zur Kamera (= Z-Koordinate) gespeichert



2D Ansicht

Tiefentest mit z-Buffer

- **Tiefentest:** Sortierung der Fragmente nach Tiefenwert soll vermieden werden
- **Z-Buffer:** Daher wird pro Fragment die Entfernung zur Kamera (= Z-Koordinate) gespeichert
- → **Aktualisierung des Pixels, wenn der neue Tiefenwert geringer ist!**



2D Ansicht

Tiefentest mit z-Buffer

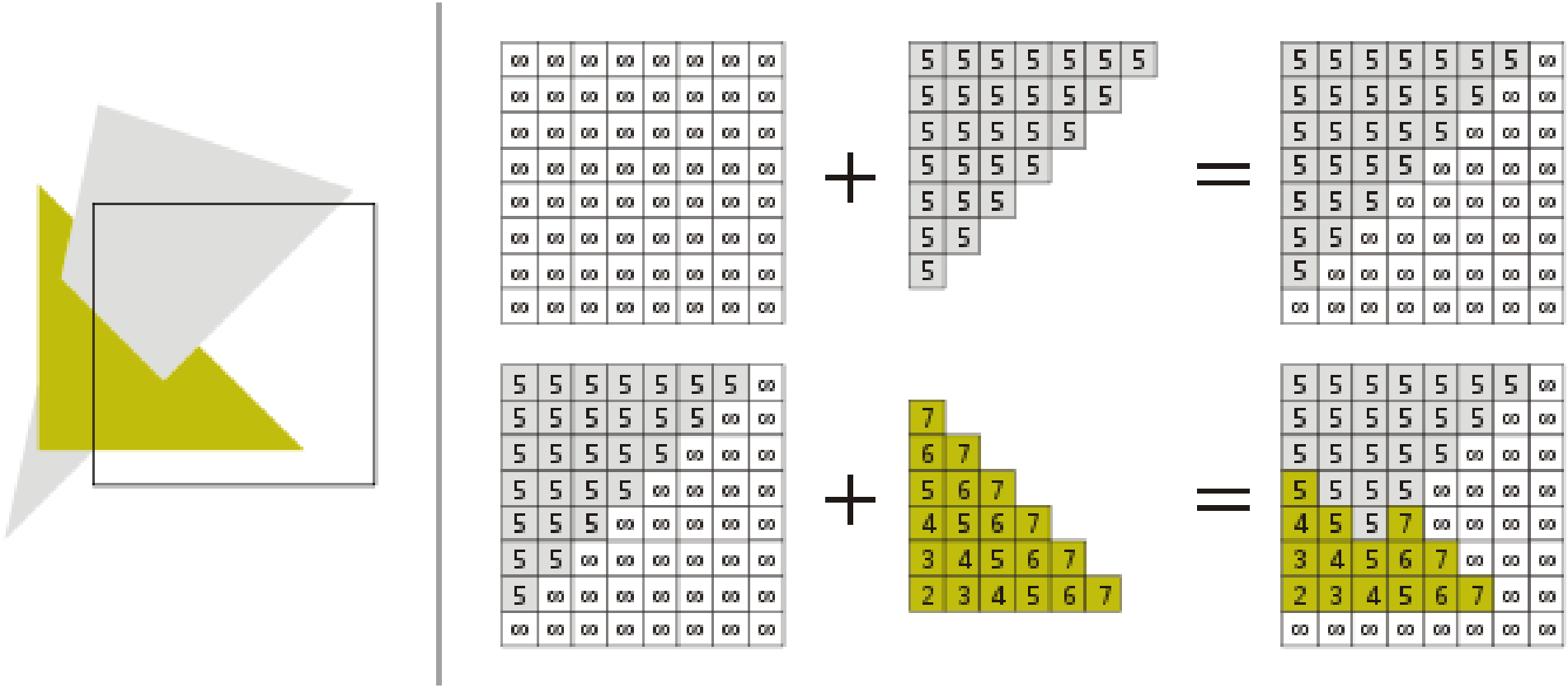


Bild aus Wikipedia: Prinzip des Z-Buffers am Beispiel zweier sich schneidender Polygone

Tiefentest in OpenGL

- OpenGL stellt verschiedene Optionen zum Tiefentest bereit
- **Es werden immer die z-Werte der Fragmente getestet!**
- Tiefentest aktivieren:
  - `glEnable(GL_DEPTH_TEST);`
    - ohne diese Zeile wird der Painters Algorithm verwendet!
- Entsprechende Tiefentest-Funktion auswählen
  - `glDepthFunc(function)`
    - Funktion: Verwendeter Tiefentest, z.B. `GL_LESS`, `GL_GREATER`, `GL_NEVER`, `GL_ALWAYS`, `GL_EQUAL`, …
    - Für den "normalen" Tiefentest: `GL_LESS`