

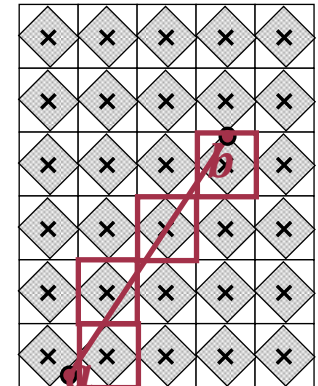
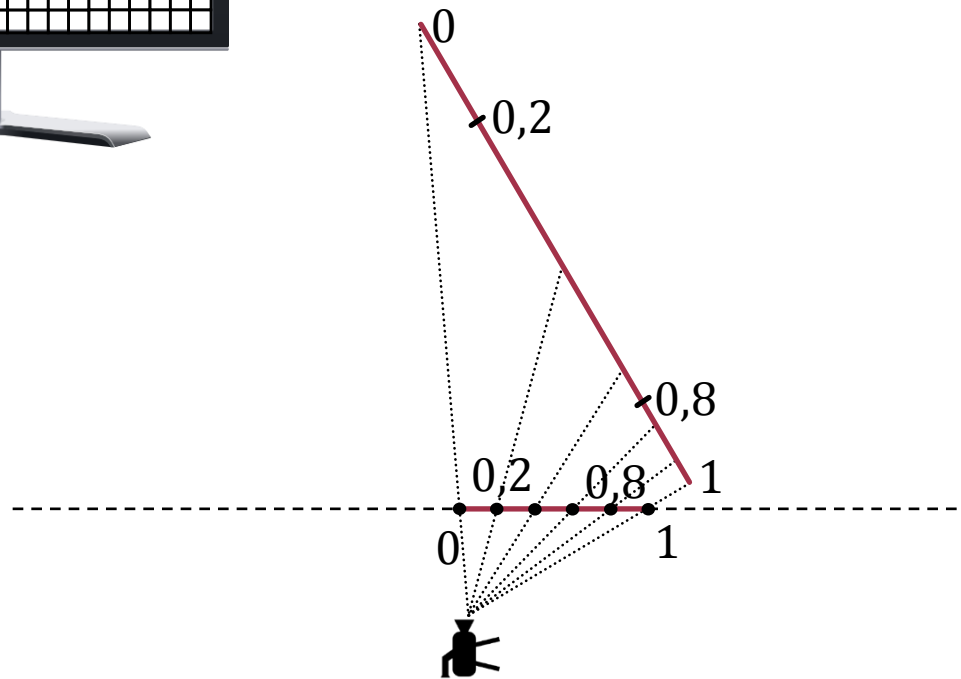
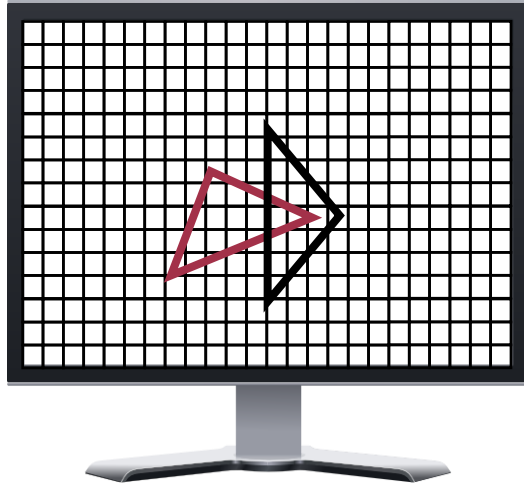
Computergrafik

Universität Osnabrück, Henning Wenke, 2012-06-12

Noch Kapitel X:

Rasterization

Was bisher geschah



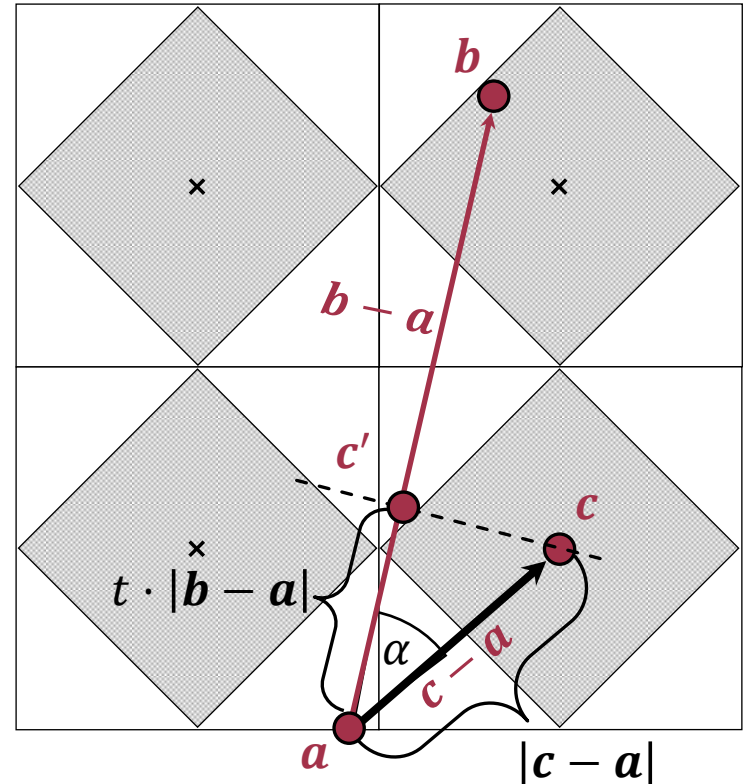
Parameter für Attribute Interpolation

- Fragmentzentrum liegt i.d.R. nicht genau auf Linie
- Deshalb wird Interpolationsparameter t in OpenGL für senkrechte Projektion c' von c auf die Linie ermittelt
- Gesucht: Linienparameter t für Fragment mit Zentrum $\mathbf{c}(x_f, y_f)$
- Es gilt:

- $\cos(\alpha) = \frac{(\mathbf{c}-\mathbf{a})(\mathbf{b}-\mathbf{a})}{|\mathbf{c}-\mathbf{a}||\mathbf{b}-\mathbf{a}|}$

- $\cos(\alpha) = \frac{t \cdot |\mathbf{b}-\mathbf{a}|}{|\mathbf{c}-\mathbf{a}|}$

- $\Rightarrow t = \frac{(\mathbf{c}-\mathbf{a})(\mathbf{b}-\mathbf{a})}{\|\mathbf{b}-\mathbf{a}\|^2}$



Attribute Interpolation

- Eckpunkte **a**, **b** haben Attribute at_a, at_b
- Interpoliere z linear
 - $z_c = (1 - t) \cdot z_a + t \cdot z_b$
- Attribute:
 - Interpoliere $\cdot z^{-1}$ (in w -Komponente) linear:
 - $\frac{at_c}{w_c} = \frac{(1-t) \cdot at_a}{w_a} + \frac{t \cdot at_b}{w_b}$
- Mit $\frac{1}{w_c} = \frac{(1-t)}{w_a} + \frac{t}{w_b}$ folgt:
 - $\frac{at_c}{\left(\frac{(1-t)}{w_a} + \frac{t}{w_b}\right)^{-1}} = \frac{(1-t) \cdot at_a}{w_a} + \frac{t \cdot at_b}{w_b}$
 - $\Rightarrow at_c = \frac{\frac{(1-t) \cdot at_a}{w_a} + \frac{t \cdot at_b}{w_b}}{\frac{(1-t)}{w_a} + \frac{t}{w_b}}$

Achtung:

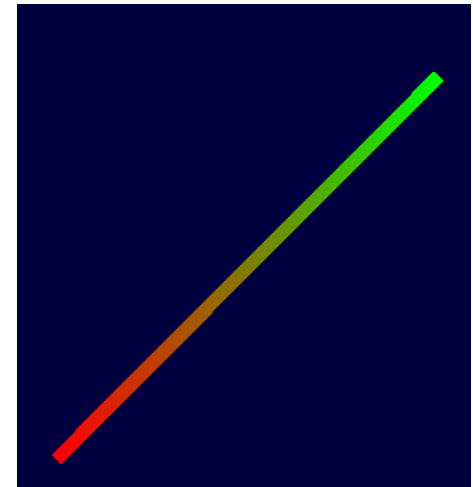
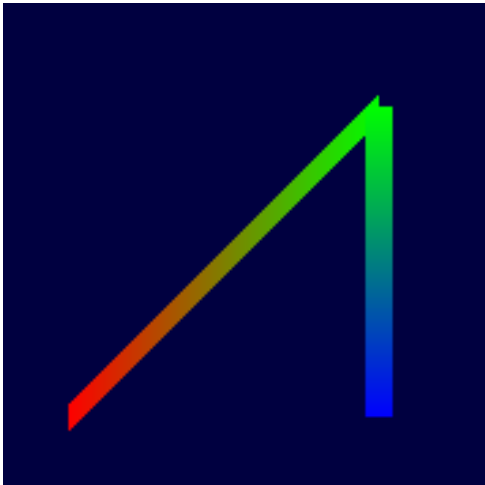
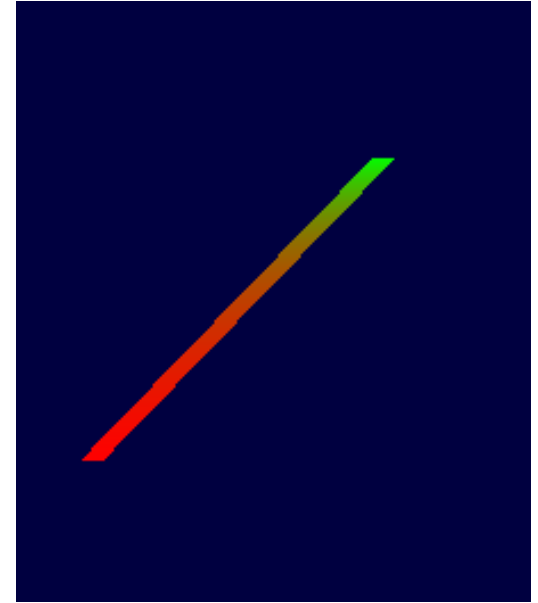
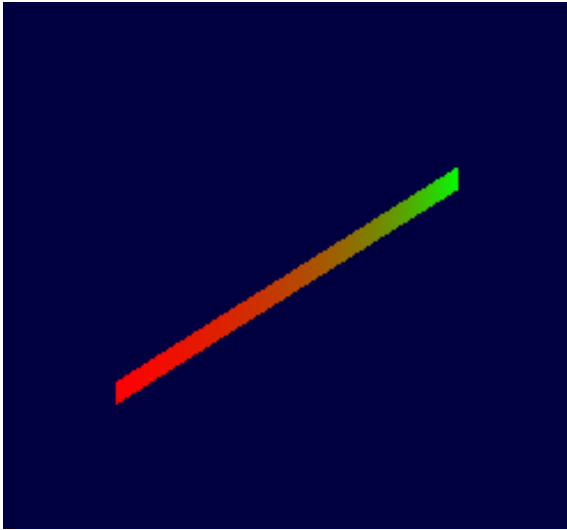
- $z_{Window} \propto \frac{1}{z_{cc}}$; Kann direkt interpoliert werden
- $w_{clip} \propto z_{cc}$, also muss $1/w$ interpoliert werden

OpenGL Line Rasterizer State

```
// Setzen Linienbreite in Pixeln, default: 1  
void glLineWidth(float size);
```

```
// Aktiviere/ Deaktiviere Glättung der Linien  
gl[En/Dis]able(GL_LINE_SMOOTH);
```

Demonstration



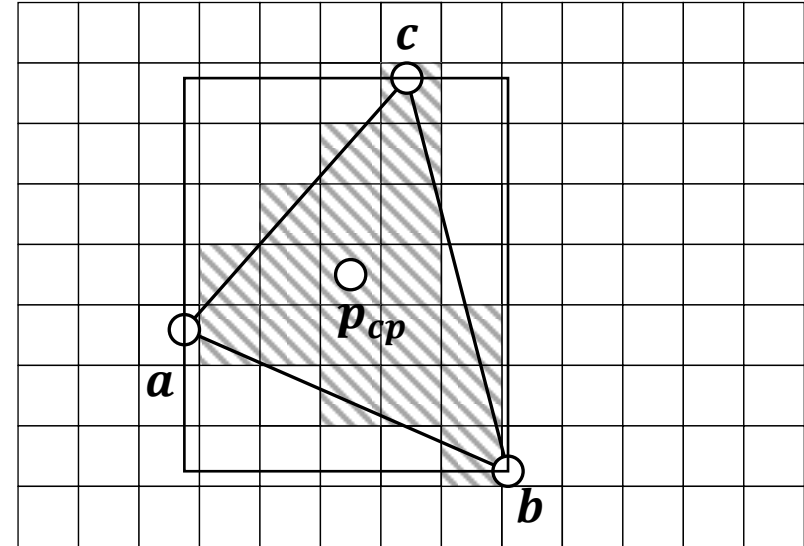
GL_LINES primär zum Debugging geeignet

X.4

Rastern von Dreiecken

Erzeugen der Fragments

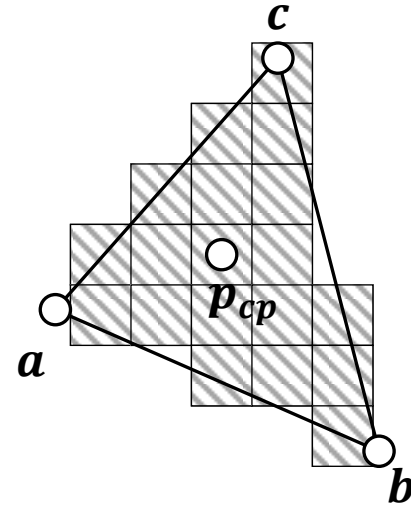
- Gegeben: Dreieck mit Eckpunkten a, b, c
- Bestimme „Kandidatenpixel“, z.B. durch rechteckigen Bereich um a, b, c und berechne deren Mittelpunkte p_{cp}
- $\forall p_{cp}$ berechne Baryzentrische Koordinaten α, β, γ , gemäß:
 - $p_{cp}(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$, mit:
 $\alpha + \beta + \gamma = 1$
- Falls $\alpha, \beta, \gamma \in [0,1]$
 - Erzeuge Fragment



Attribute Interpolation

- Eckpunkte ***a***, ***b***, ***c*** haben Attribute ***at_a***, ***at_b***, ***at_c***
- Interpoliere Daten mit Baryzentrischen Koordinaten des Fragment Mittelpunkts als Gewichten
- Z linear
 - $z_{cp} = \alpha \cdot z_a + \beta \cdot z_b + \gamma \cdot z_c$
- Interpoliere Attribute $\cdot z^{-1}$ linear:

- $$at_{cp} = \frac{\alpha \cdot \frac{at_a}{w_a} + \beta \cdot \frac{at_b}{w_b} + \gamma \cdot \frac{at_c}{w_c}}{\frac{\alpha}{w_a} + \frac{\beta}{w_b} + \frac{\gamma}{w_c}}$$

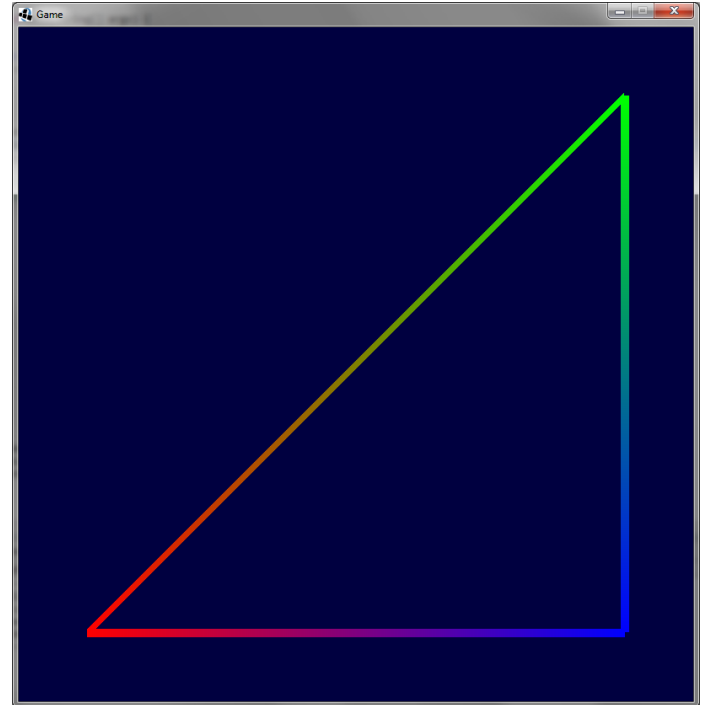
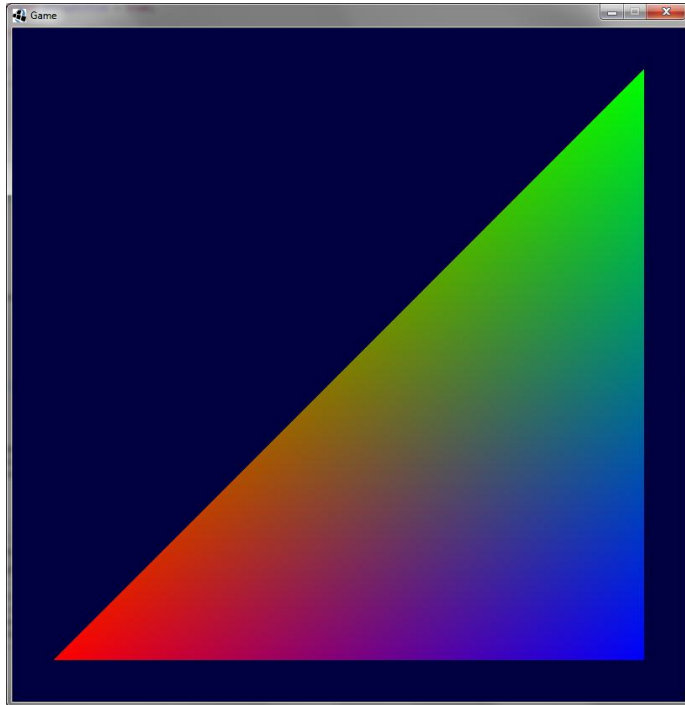


OpenGL Triangle Rasterizer State

```
// Festlegen der Art des Zeichnens der Dreiecke
// Erlaubte Konstanten für mode:
// GL_FILL   -> Zeichnet gefüllte Dreiecke (default)
// GL_LINE   -> Zeichnet Konturen der Dreiecke als GL_LINES
// GL_POINT  -> Zeichnet Eckpunkte der Dreiecke als GL_POINTS
void glPolygonMode(GL_FRONT_AND_BACK, int mode);
```

```
// Aktiviere / Deaktiviere Glättung der Dreiecke
gl[En/Dis]able(GL_POLYGON_SMOOTH);
```

Demonstration



Daten

IN
(pro Instanz)

1 Primitive, 1-3 Vertices

OUT
(pro Instanz)

(Anzahl der überlappten Pixel)
Fragments

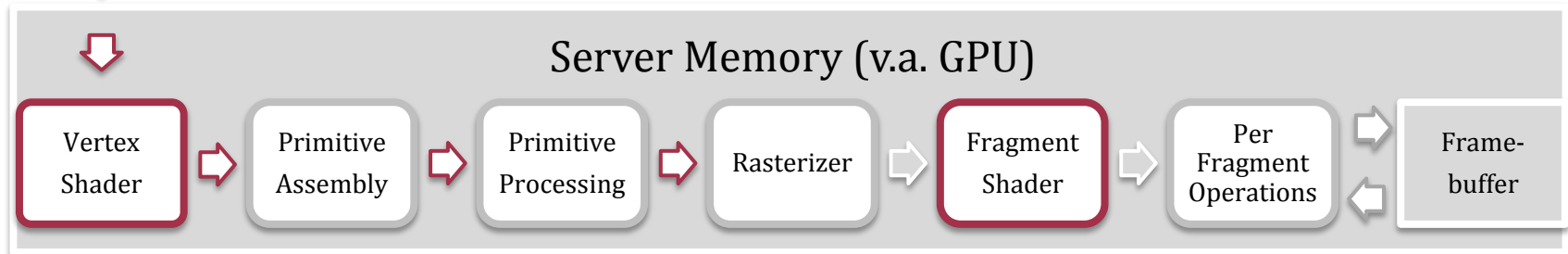
Kapitel XI:

Fragment Shader

OpenGL Graphics Pipeline

Client Memory (Unsere Java Applikation)

OpenGL Befehle

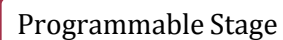


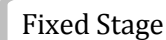
Legende

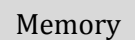
 Vertices

 Fragments

 Pixel Data

 Programmable Stage

 Fixed Stage

 Memory

Definition

- Fragment Shader ist in Shading Language, etwa GLSL, geschriebenes Programm
- Wird für jedes durch den Rasterizer erzeugte Fragment aufgerufen
- Verarbeitet dessen Daten
 - Oft: Berechnet aus Daten eine Fragmentfarbe
- Alle Instanzen unabhängig voneinander

Shader Paar: Vertex Lighting

Vertex Shader

```
#version 150 core
in vec3 normalMC;
in vec4 posMC;
uniform mat3 mc2wc_Normal;
uniform mat4 mc2wc_Pos, vp;
uniform vec3 inverseLightDir;

out float brightness;

void main() {
    gl_Position = vp*mc2wc_Pos*posMC;

    vec3 normalWC
        = mc2wc_Normal * normalMC;

    brightness = max(
        dot(normalWC, inverseLightDir),
        0.0
    );
}
```

Fragment Shader

```
#version 150 core

const vec3 color = vec3(1.0, 1.0, 1.0);

in float brightness;

out vec4 fragColor;

void main() {

    fragColor
        = vec4(color * brightness, 1.0);
}
```

Shader Paar: Fragment Lighting

```
// Vertex Shader
#version 150 core
in vec3 normalMC;
in vec4 posMC;
uniform mat3 mc2wc_Normal;
uniform mat4 mc2wc_Pos, vp;

// Was muss ausgegeben werden?
out vec3 normalWC;

void main() {

    gl_Position = vp*mc2wc_Pos * posMC;

    normalWC = mc2wc_Normal * normalMC;
}
```

```
// Fragment Shader
#version 150 core
const vec3 color = vec3(1.0, 1.0, 1.0);
uniform vec3 inverseLightDir;

in vec3 normalWC;

out vec4 fragColor;

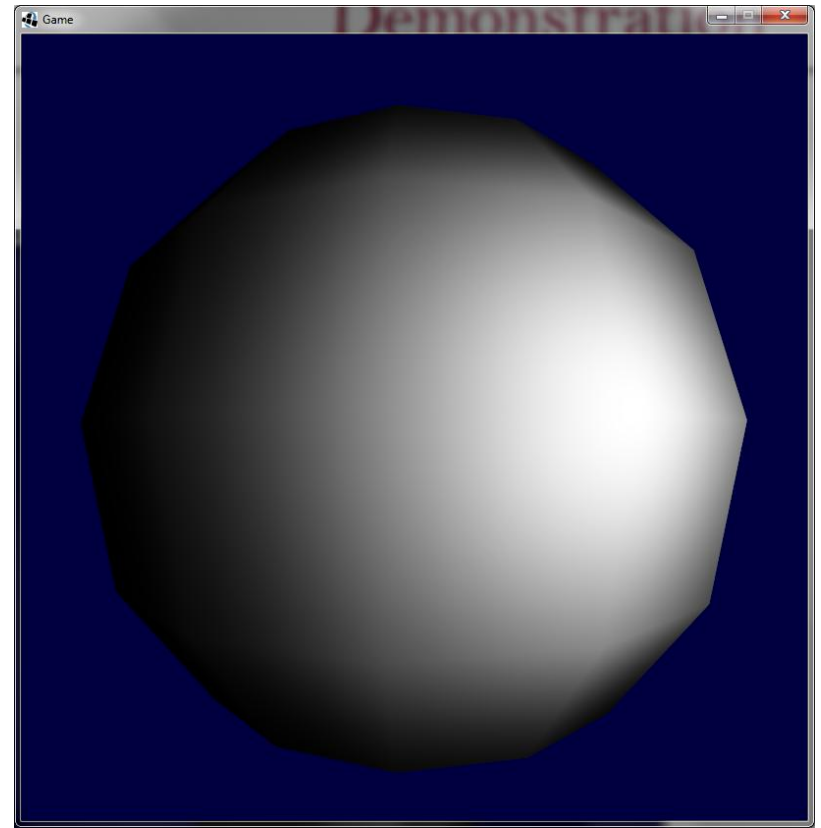
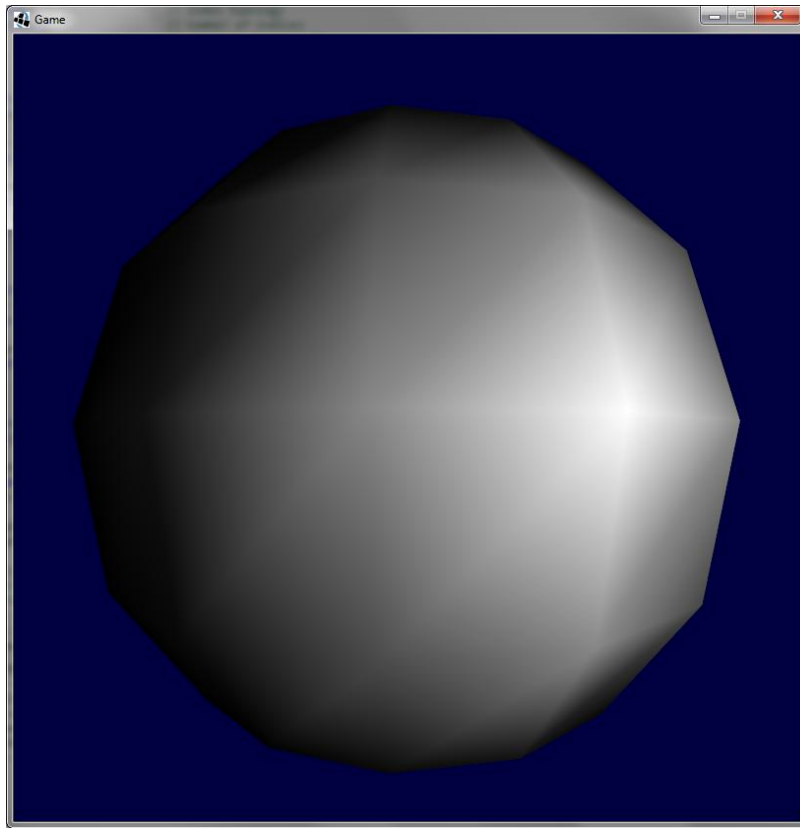
void main() {

    vec3 fragNormal = normalize(normalWC);

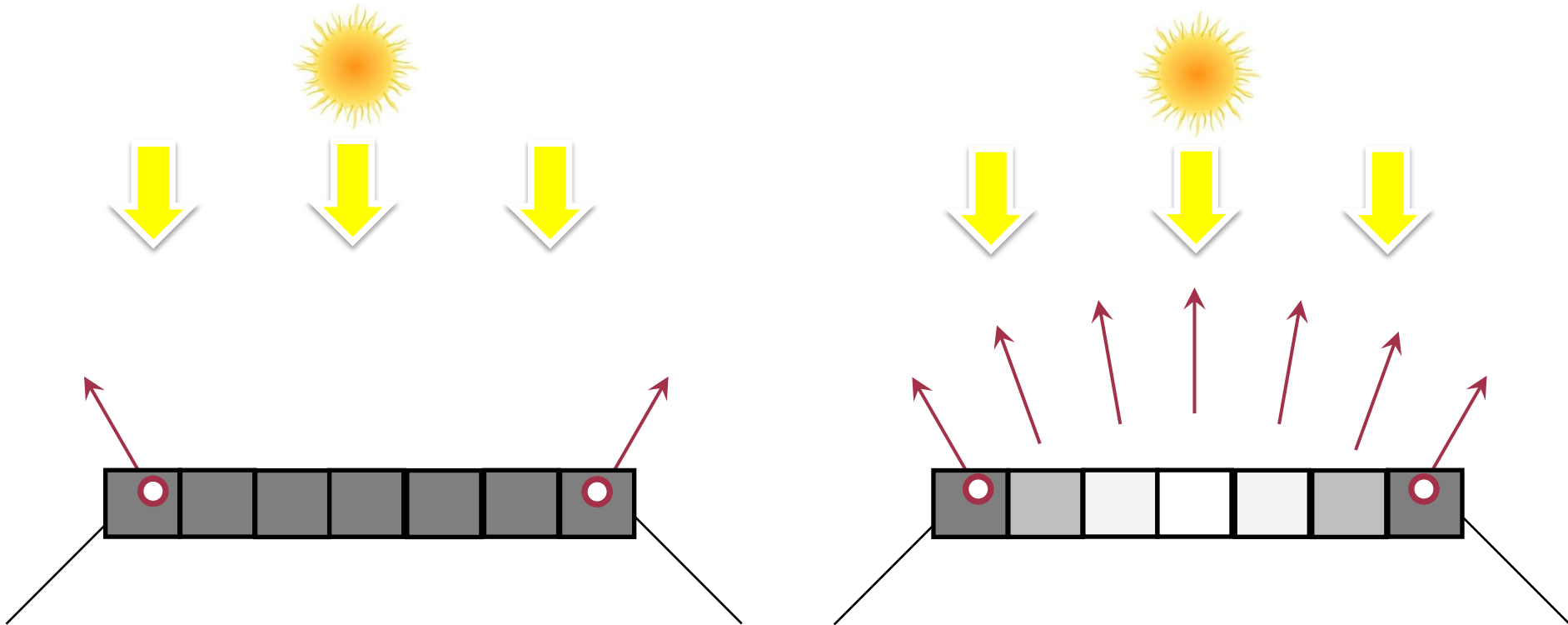
    brightness = max(
        dot(fragNormal, inverseLightDir),
        0.0
    );

    fragColor
        = vec4(color * brightness, 1.0);
}
```

Demonstration



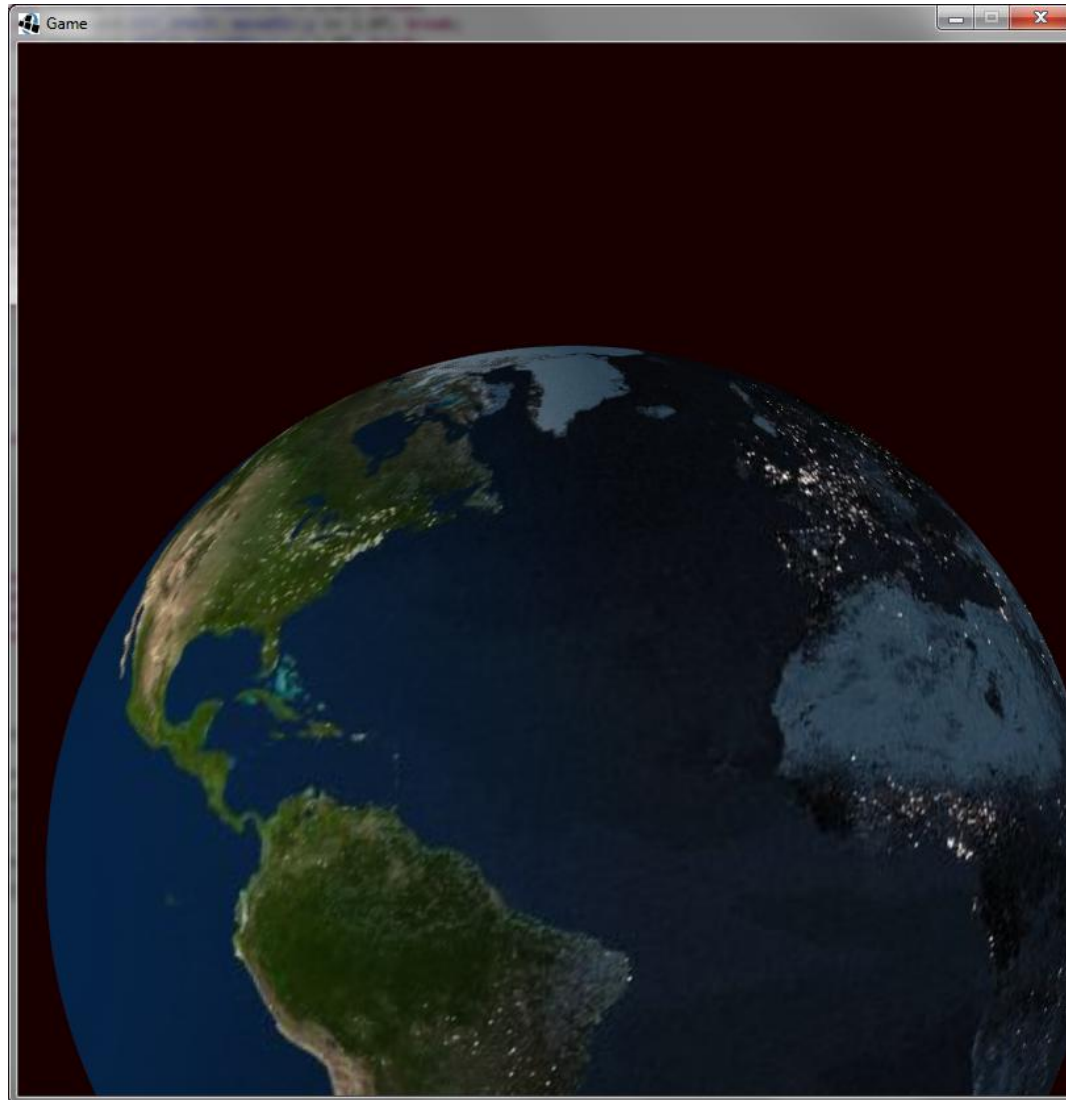
Farb- vs. Normaleninterpolation



VS & FS: Bildqualität & Performance

- Bildqualität besser bei Berechnungen im...
 - FS, da immer in “passender” Genauigkeit gerechnet wird
- Performance besser bei Berechnungen ...
 - Unklar
- Vertex Shader:
 - Typischerweise Vertexzahl << Fragmentzahl
 - Muss aber nicht!
 - Berechnung findet auf jeden Fall statt, da Pipeline Anfang
- Fragment Shader
 - Findet oft nicht statt: Clipping, Culling, Early-Z, Deferred Shading...
 - Fragments regelmäßig angeordnet und benachbarte greifen oft auf ähnliche / gleiche Daten zu
 - Ergibt günstigere Speicherzugriffsmuster
 - Deshalb identische Berechnungen im FS oft schneller als im VS

Demonstration: Tag-Nacht-Wechsel



Einige Build-In-Variablen

➤ **in** `vec4 gl_FragCoord;`

- `x, y`: Koordinaten des zum Fragment gehörigen Pixelzentrums
- `z`: Interpolierter Tiefenwert
- `w`: Interpolierte $1/w$ -Komponente aus Clip Coordinates

➤ **in** `vec2 gl_PointCoord;`

- Gibt Koordinate im KS eines `GL_POINTS`s an

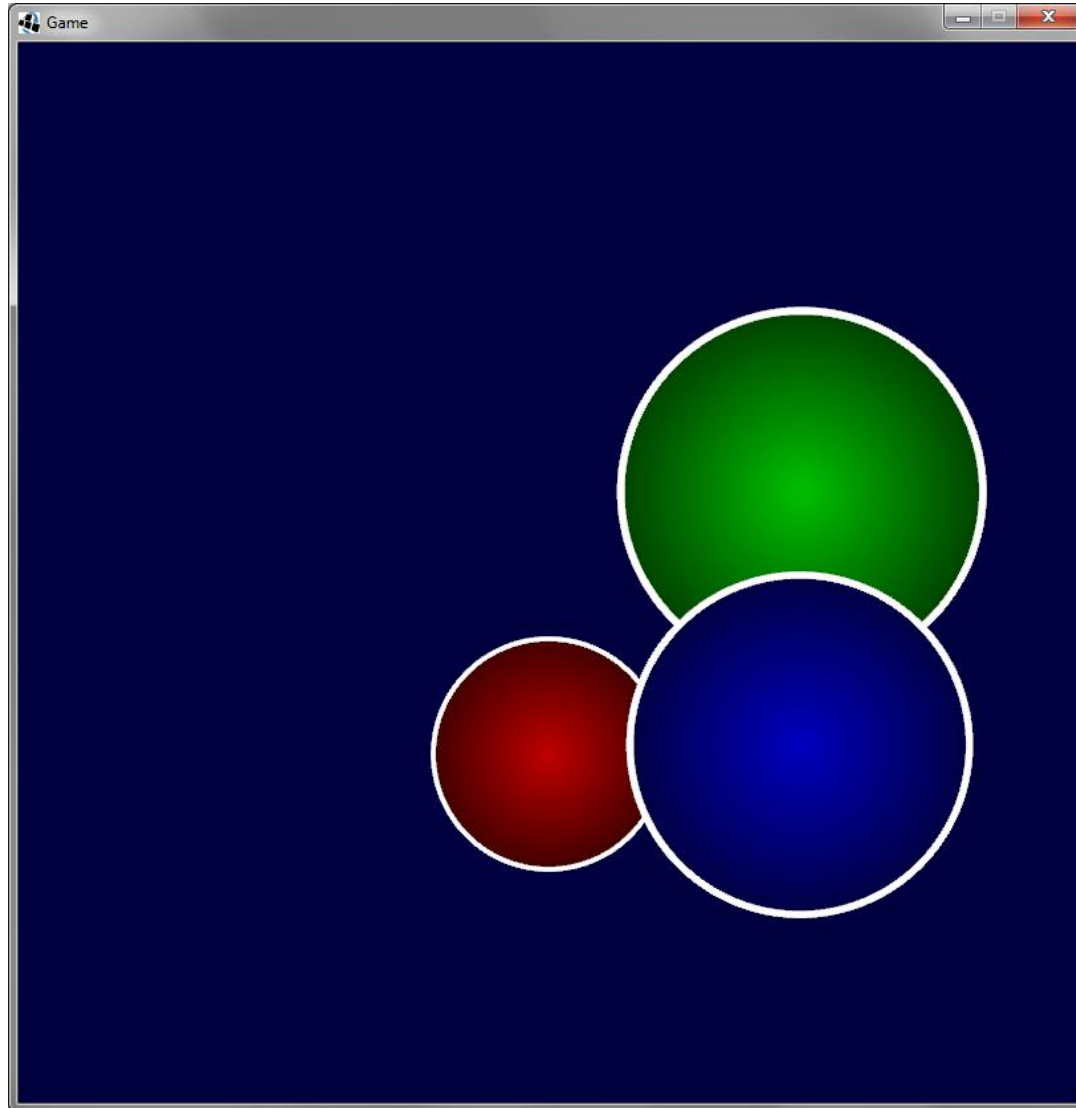
➤ **out** `float gl_FragDepth;`

- Erlaubt Verschieben des Fragments in `z`-Richtung
- Falls nicht geschrieben, wird `gl_FragCoord.z` verwendet

~~➤ **out** `vec4 gl_FragColor;`~~

➤ Sonstiges: Keyword `discard` verwirft Fragment

Demonstration: Shaded GL_POINTS



Zugehörige OpenGL Befehle

```
// Schreibt FS-Out Variable in bestimmten Colorbuffer  
void glBindFragDataLocation(  
    int    program,          // id des Program Objects  
    int    colorNumber,      // Index des zu schreibenden Buffers. Default: 0  
    String name              // Name der Variable  
);
```

```
// Beispiel: Schreibe Werte von fragColor in ColorBuffer 0 (Standard  
// Farbbuffer)  
glBindFragDataLocation(..., 0, "fragColor");
```

```
// Initialisierung eines (Fragment) Shader Objects wie bei VS Object,  
// außer:  
glCreateShader(GL_FRAGMENT_SHADER);
```

Fragen

- Kann FS die Position eines Fragments verändern?
 - X,y: Nein
 - Z, : Ja, Durch Schreiben von `gl_FragDepth`
- Muss FS eine Farbe ausgeben?
 - Nein, es existiert keine entsprechende Build-In Variable
- Kann FS die Farben aller für einen Pixel erzeugten Fragments mischen?
 - Nein, Zugriff nur auf Daten eines Fragments
- Kann FS direkt auf Pixel einwirken (z.B. einfärben)
 - Nein, da potentiell mehrere Fragments pro Pixel existieren können und Synchronisation unmöglich
- Kann FS mehrere selbstdefinierte Variablen, z.B. Normale und Objektfarbe, schreiben?
 - Ja. Müssen an unterschiedliche Colorbuffer angebunden werden

Daten

IN
(pro Instanz)

1 Fragment

OUT
(pro Instanz)

0 oder 1 Fragment

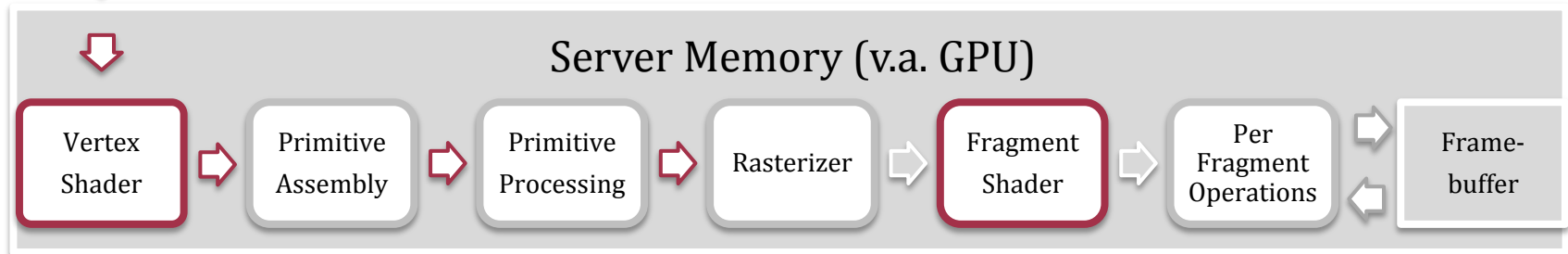
Kapitel XII:

Per Fragment Operations

OpenGL Graphics Pipeline

Client Memory (Unsere Java Applikation)

↓ OpenGL Befehle



Legende

↓ Vertices

→ Fragments

→ Pixel Data

Programmable Stage

Fixed Stage

Memory

Aufgaben

- Regeln den Einfluss der Fragments auf das jeweils korrespondierende Pixel
- Vergleicht immer 1 Fragment zur Zeit mit Pixel
- Außerdem: Einstellungen für den gesamten Framebuffer

IN
(pro Pixel, seriell)

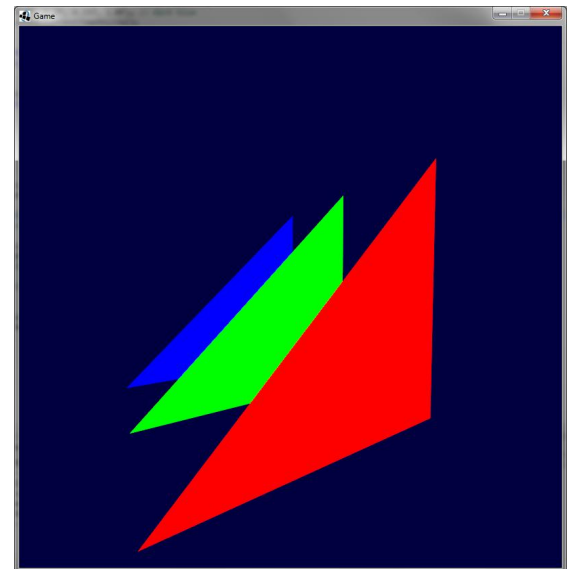
0...n Fragments

“OUT”
(pro Pixel, seriell)

Manipuliert maximal einen
Pixel

XII.1

Depth Test



Funktionsweise

- Blendet verdeckte Objekte aus
- Wird pro Pixel sequentiell für alle Fragments ausgewertet
- Setzt Pixel beispielsweise auf Wert des nächsten Fragments & verwirft alle anderen
- Wichtig: Reihenfolge zeitlich, nicht räumlich

Fragments



Zeit



Korrespondierender Pixel



Bedingung

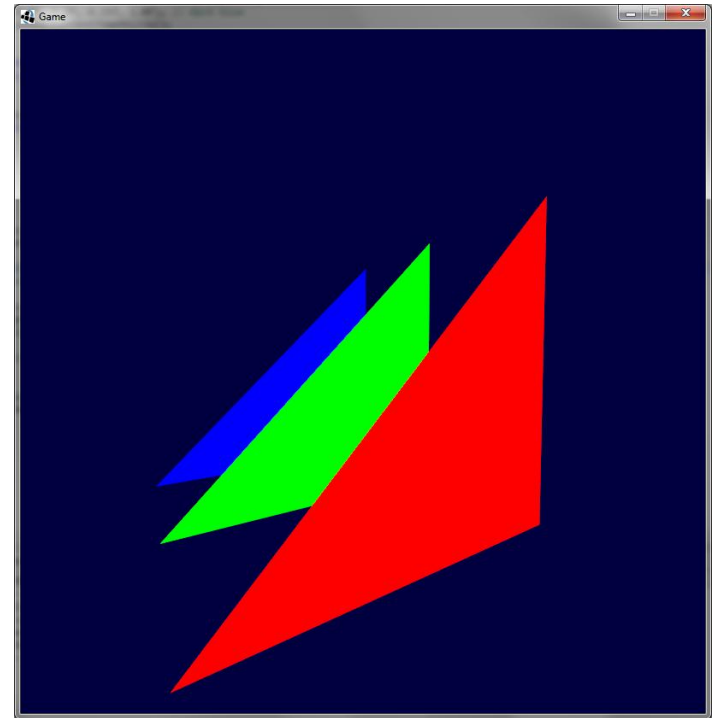
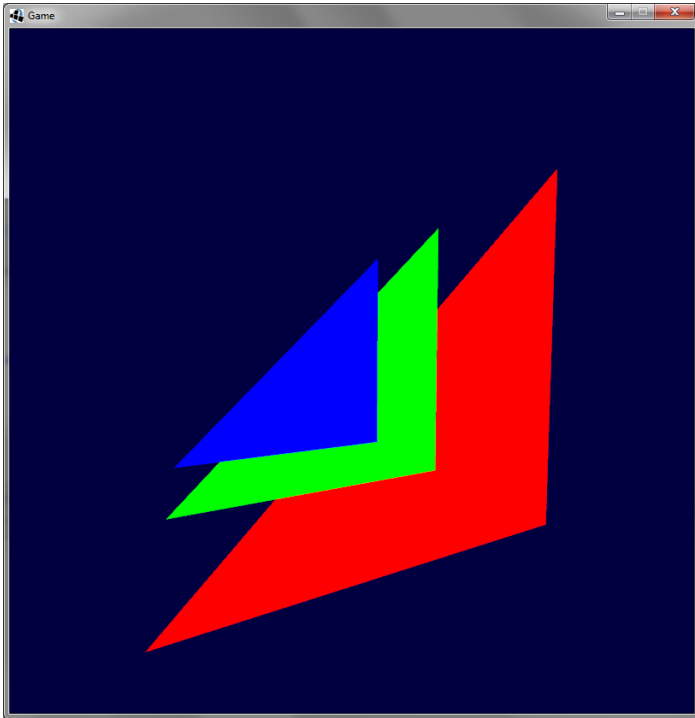


Zugehörige OpenGL Befehle

```
// Aktiviert / Deaktiviert Tiefentest  
gl[En/Dis]able(GL_DEPTH_TEST);
```

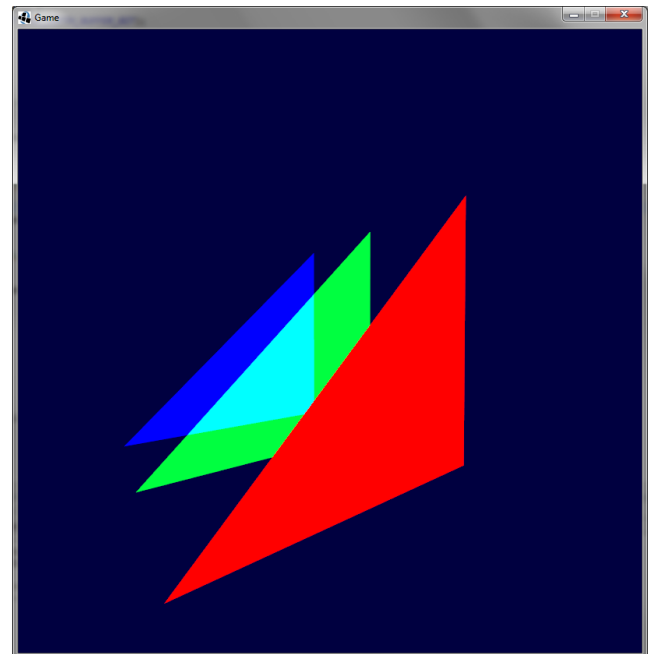
```
// Setzt Bedingung für den Tiefentest  
void glDepthFunc(  
    int func    // Bedingung, etwa...  
                //    GL_LESS, default, dichteres Fragment setzt sich durch  
                //    GL_GREATER,  
                //    GL_NOTEQUAL, GL_NEVER, GL_LEQUAL  
);
```

Demonstration



XII.2

Blending



Funktionsweise

- Angewandt auf Fragments, die vorherige Sichtbarkeitstests (Depth Test) passiert haben
- Wird pro Pixel sequentiell für alle Fragments ausgewertet
- Mischt bereits für Pixel eingetragene Farbe (dst) mit der des hereinkommenden Fragments (src)
- Berechnung festgelegt durch:
 - Gewichte für die Farben
 - Typischerweise eine Konstante oder src/dst Farb- bzw. Alphawerte
 - RGBA: Alpha Wert kann als Gewicht herangezogen werden
 - Funktion, mit der beide verknüpft werden

Zugehörige OpenGL Befehle I

```
// Aktiviert / Deaktiviert Blending  
gl[En/Dis]able(GL_BLEND);
```

```
// Setzt Gewichte für Verrechnung von Fragment- und Pixelfarbe  
// Mögliche Werte für sfactor und dfactor:  
//   GL_ONE, GL_ZERO  
//   GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR,  
//   GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR,  
//   GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA,  
//   GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA  
//   ...  
void glBlendFunc(  
    int sfactor, // Gewicht für hinzukommende Fragmentfarbe  
    int dfactor, // Gewicht für bereits eingetragene Pixelfarbe  
);
```

Zugehörige OpenGL Befehle II

```
// Setzt den Operator für Blendinggleichung
// Mögliche Werte für mode:
//   GL_FUNC_ADD: Addiere gewichtete Werte (diese Veranstaltung)
//   GL_FUNC_SUBTRACT,
//   GL_MAX,
//   ...
void glBlendEquation(
    int mode
);
```

α|B| Blend Equation (Gr. FINEC_ADD) :

Gewicht für
Fragments

Gewicht für
Pixel

Beispiel: $\sigma_{\text{geschohen}}: \mathbb{R}^{1 \times 4} \rightarrow \mathbb{R}^{1 \times 4}$ (grün: $(0 \ 1 \ 0 \ 1)$, also $\text{Dixal} \mapsto \text{tarbo}$)

Varianz Gewichtete: $\frac{1}{4} \cdot (0 \ 0 \ 1 \ 1) \cdot ((1 \ 1 \ 1 \ 1) = (0 \ 0 \ 1 \ 1)) : (0 \ 1 \ 0 \ 1)$

Perceptron: $(0 \ 0 \ 1 \ 1)$ \rightarrow $(0 \ 1 \ 0 \ 0)$

Demonstration

