

Computergrafik SS 2010

Henning Wenke

Kapitel 21:

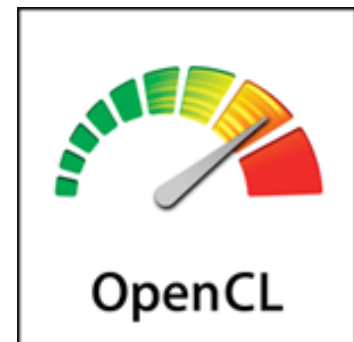
OpenGL 3.1

Einordnung



Über OpenGL

- API für Rastergrafik
 - Prozedural
 - Hardwarenah
 - Plattform-, Betriebssystem- und Sprachunabhängig
- Spezifikationen variieren im Funktionsumfang
 - Diese Veranstaltung: Version 3.1, Core Profile
- Implementation divergieren in:
 - Hardwarenutzung / Performance
 - Genauigkeit (etwas)
- Verwandte APIs:



Entwicklungsgeschichte



Vertex

- Mathematischer Punkt im Raum
- Oft „Eckpunkt“ einer geometrischen Figur
- Enthält oft weitere Eigenschaften an diesem Punkt, etwa:
 - Normale
 - Farbe
 - Texturkoordinaten
 - Geschwindigkeit
 - Beschleunigung
 - Materialdichte
 - ...

Primitive

- „Elementare Grafische Grundform“
- Besteht aus 1 - 3 Vertices
- Implizit topologischen Informationen
- Eventuell Ausdehnung

Fragment

- Vom Rasterizer aus Primitive für einen Pixel erzeugte Datenstruktur
- Enthält zunächst für diese Stelle interpolierte Daten der zugehörigen Vertices
- „Pixelvorstufe“

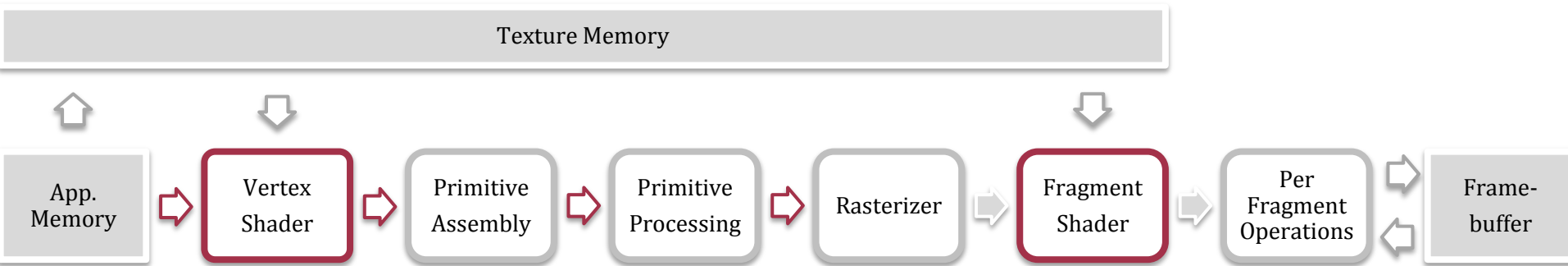
Uniform Data

- Nicht durch die Graphics Pipeline verarbeitet, konstant
- Global lesbar
- Enthält oft Informationen anderer Frequenz \times Matrix II als die Geometrie

Texture

- Ein- bis dreidimensionale Datenstruktur
- Texel ein- bis vierdimensional
- Enthält beliebige numerische Informationen
 - Farbe, Normale, ..., Dichte, Geschwindigkeit
 - Informationsdichte oft höher als Geometrieauflösung
- Zusätzlich Filterfunktionen

Graphics Processing Pipeline





Legende


 Vertices

 Fragments

 Pixel/Texture Data

 Programmable Stage

 Fixed Stage

 Memory

Vertex Shader

- Programm
- Wird unabhängig für jeden Vertex ausgeführt
- Verarbeitet dessen Daten

Shader Beispiel

```
#version 140

float getAB(float a, float b){
    return a * b;
}

void main() {

    float a = 5.0;
    a = getAB(10.0, 1.5);
    vec3 vectorA = vec3(1.0, 0.0, 0.0);
    vec3 vectorB = vec3(1.0, 1.0, 0.0);
    vec3 kp = cross(vectorA, vectorB);
    float sp = dot(vectorA, vectorB);
    vec3 kompMul = vectorA * vectorB;
    vec3 kompAdd = vectorA + vectorB;

    for(int i=0; i< 5; i++){
        if(a<1000)
            a *= 2;
    }
}
```

Vertex Shader Beispiel

```
#version 140

uniform mat4 mvpMatrix;
in vec4 vPosition;

void main() {

    gl_Position = mvpMatrix * vPosition;

}
```

Shader

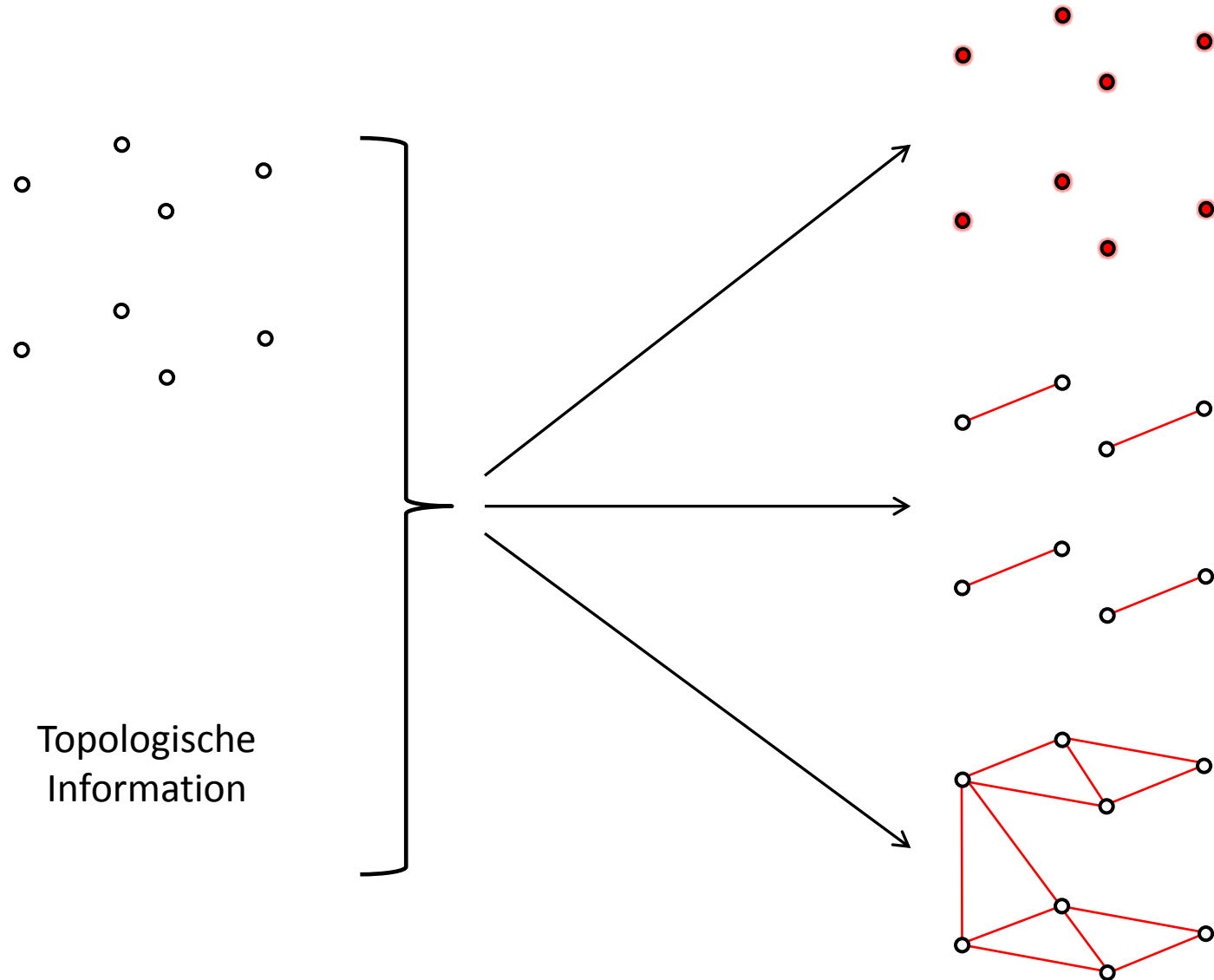
➤ Shader (Cook, 1984)

- Programm zur Beschreibung/Berechnung von Oberflächeneigenschaften
- Renderman Shading Language

➤ Shader (Hier)

- In einer Shadersprache (GLSL, HLSL, Cg, ...) geschriebenes Programm, welches auf einer GPU ausgeführt werden kann
- Prozedural
- Heute oft an C angelehnte Hochsprachen mit speziellen Vektor und Matrix Datentypen und Operatoren

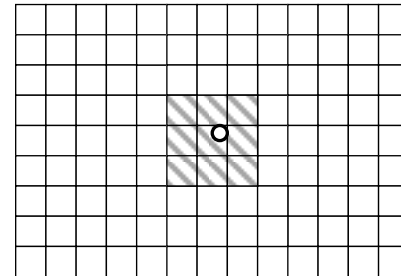
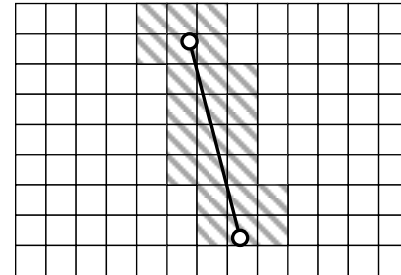
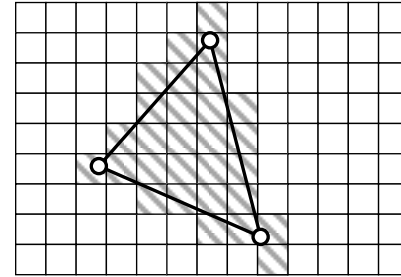
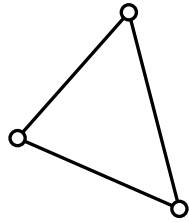
Primitive Assembly



Primitive Processing

- Operationen, die Informationen über ganzes Primitive benötigen
- Restliche Transformationen
 1. Clipping
 2. Perspective Division
 3. Viewport Transformation
 4. Culling

Rasterizer



Fragment Shader

```
// Vertex Shader, Folie 13
```

```
...  
in vec4 vPosition;  
in vec2 myTexCoords;  
out vec2 texCoords;  
  
void main() {  
  
    gl_Position = . . .  
    texCoords = textureCoords;  
  
}
```

```
// Fragment Shader
```

```
#version 140  
  
uniform sampler2D colors;  
  
in vec2 texCoords;  
  
out vec4 fragColor;  
  
void main() {  
  
    fragColor = texture(colors, texCoords);  
  
}
```

Per Fragment Operations

Regeln den Einfluss der Fragments auf das jeweils korrespondierende Pixel

Datenfluss

