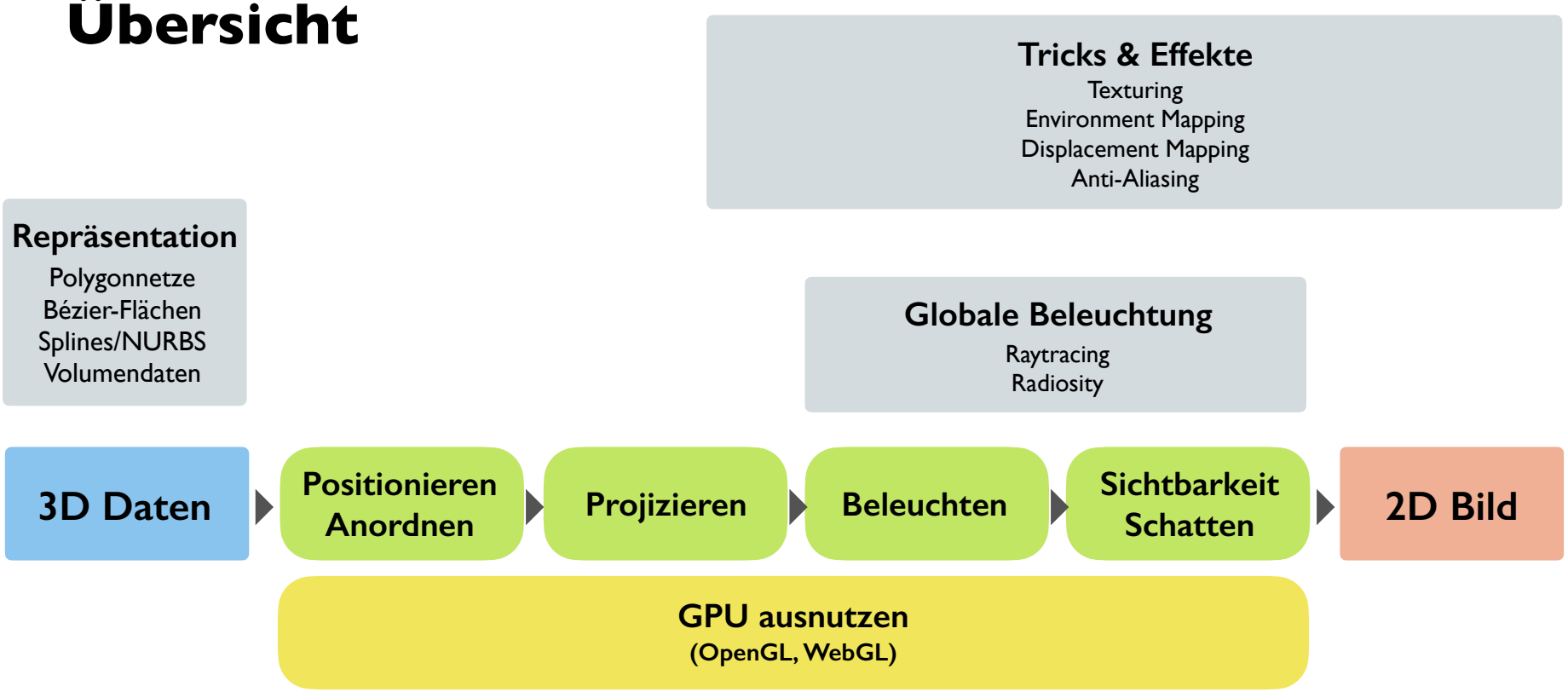
Vorlesung
Computergrafik

Media Informatics
Graphics & Geometric Computing
Prof. Dr. M. Campen

Übersicht


Tricks & Effekte
Texturing
Environment Mapping
Displacement Mapping
Anti-Aliasing
Repräsentation
Polygonnetze
Bézier-Flächen
Splines/NURBS
Volumendaten
Globale Beleuchtung
Raytracing
Radiosity
3D Daten
Positionieren Anordnen
Projizieren
Beleuchten
Sichtbarkeit Schatten
2D Bild
GPU ausnutzen
(OpenGL, WebGL)

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreiecke*”, ggf. “*Linien*”)
- Lichtquellen
- Beleuchtungsparameter
- Transformationsmatrizen
- Projektionsmatrizen
- …

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreiecke*”, ggf. “*Linien*”)
- Pro-Vertex-Berechnungen
  - Transformation
  - Projektion
  - Beleuchtung (bei Flat oder Gouraud Shading)
- Rasterisierung
  - Pro Dreieck die enthaltenen Pixel bestimmen (“Fragments”)
- Pro-Fragment-Berechnungen
  - Beleuchtung (bei Phong Shading)
  - Interpolation der drei pro-Vertex-berechneten Werte (Beleuchtung, …)
  - Verdeckungstest (z-Buffer)

- Lichtquellen
- Beleuchtungsparameter
- Transformationsmatrizen
- Projektionsmatrizen
- …

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreiecke*”, ggf. “*Linien*”)
- Lichtquellen
- Beleuchtungsparameter
- Transformationsmatrizen
- Projektionsmatrizen
- …
- Pro-Vertex-Berechnungen
  - Transformation
  - Projektion
  - Beleuchtung (bei Flat oder Gouraud Shading)

unabhängig von anderen Vertices, daher trivial parallelisierbar

- Rasterisierung
  - Pro Dreieck die enthaltenen Pixel bestimmen (“Fragments”)
- Pro-Fragment-Berechnungen
  - Beleuchtung (bei Phong Shading)
  - Interpolation der drei pro-Vertex-berechneten
  - Verdeckungstest (z-Buffer)

unabhängig von anderen Fragments, daher trivial parallelisierbar

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreiecke*”, ggf. “*Linien*”)
- Lichtquellen
- Beleuchtungsparameter
- Transformationsmatrizen
- Projektionsmatrizen
- …

- Pro-Vertex-Berechnungen
  - Transformation
  - Projektion
  - Beleuchtung (bei Flat oder Gouraud Shading)
- Rasterisierung
  - Pro Dreieck die enthaltenen Pixel bestimmen (“Fragments”)
- Pro-Fragment-Berechnungen
  - Beleuchtung (bei Phong Shading)
  - Interpolation der drei pro-Vertex-berechneten Werte (Beleuchtung, …)
  - Verdeckungstest (z-Buffer)

Hardware-Implementierung in 3D-Grafikbeschleuniger-Chips (ca. 1982-2003)

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreiecke*”, ggf. “*Linien*”)
  - Lichtquellen
  - Beleuchtungsparameter
  - Transformationsmatrizen
  - Projektionsmatrizen
  - …

Diese Daten werden durch einen Treiber an den 3D-Grafikchip gesendet. Dieser wird angesprochen über APIs, wie OpenGL oder DirectX.

- Pro-Vertex-Berechnungen
  - Transformation
  - Projektion
  - Beleuchtung (bei Flat oder Gouraud Shading)
- Rasterisierung
  - Pro Dreieck die enthaltenen Pixel bestimmen (“Fragments”)
- Pro-Fragment-Berechnungen
  - Beleuchtung (bei Phong Shading)
  - Interpolation der drei pro-Vertex-berechneten Werte (Beleuchtung, …)
  - Verdeckungstest (z-Buffer)

Hardware-Implementierung in 3D-Grafikbeschleuniger-Chips (ca. 1982-2003)

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreiecke*”, ggf. “*Linien*”)
  - Lichtquellen
  - Beleuchtungsparameter
  - Transformationsmatrizen
  - Projektionsmatrizen
  - …

Diese Daten werden durch einen Treiber an den 3D-Grafikchip gesendet. Dieser wird angesprochen über APIs, wie OpenGL oder DirectX.

- Pro-Vertex-Berechnungen
  - Transformation
  - Projektion
  - Beleuchtung (bei Flat oder Gouraud Shading)
- Rasterisierung
  - Pro Dreieck die enthaltenen Pixel bestimmen (“Fragments”)
- Pro-Fragment-Berechnungen
  - Beleuchtung (bei Phong Shading)
  - Interpolation der drei pro-Vertex-berechneten Werte (Beleuchtung, …)
  - Verdeckungstest (z-Buffer)

seit ca. 2003:

Hardware-Implementierung in 3D-Grafikchips

Massiv-parallele Software-Impl. in 3D-Grafikchips

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreiecke*”, ggf. “*Linien*”)
  - Lichtquellen
  - Beleuchtungsparameter
  - Transformationsmatrizen
  - Projektionsmatrizen
  - …

Diese Daten werden durch einen Treiber an den 3D-Grafikchip gesendet. Dieser wird angesprochen über APIs, wie OpenGL oder DirectX.

- Pro-Vertex-Berechnungen
  - …
  - …
  - …
- Rasterisierung
  - Pro Dreieck die enthaltenen Pixel bestimmen (“Fragments”)
- Pro-Fragment-Berechnungen
  - …
  - …
  - Verdeckungstest (z-Buffer)

seit ca. 2003:

Hardware-Implementierung in 3D-Grafikchips

Massiv-parallele Software-Impl. in 3D-Grafikchips

Struktur

- Daten-Definition
  - Punkte (“*Vertices*”)
    - Positionen
    - Normalen
    - Farben/Materialien
  - Konnektivität (“*Dreieck*
  - Lichtquellen
  - Beleuchtungsparameter
  - Transformationsmatrizen
  - Projektionsmatrizen
- Pro-Vertex-Berechnunge
  - …
  - …
  - …
- Rasterisierung
  - Pro Dreieck die enthaltenen Pixel bestimmen (“Fragments”)
- Pro-Fragment-Berechnungen
  - …
  - …
  - Verdeckungstest (z-Buffer)

Diese Daten werden durch einen Treiber an den 3D-Grafikchip gesendet. Dieser wird angesprochen über APIs, wie OpenGL oder DirectX.

Der Code, der für die pro-Vertex- (*Vertex Shader*) und pro-Fragment-Berechnungen (*Fragment Shader*) massiv-parallel durch den Grafikchip ausgeführt werden soll, wird zuvor ebenfalls über solche APIs an den Chip geschickt.

seit ca. 2003:

Hardware-Implementierung in 3D-Grafikchips

Massiv-parallele Software-Impl. in 3D-Grafikchips

APIs

- OpenGL
  - plattform- und programmiersprachenübergreifend
  - Version 1.0-1.5: *Fixed Function Pipeline*
  - Version *2.0-…:* frei programmierbare *Shader (GLSL: OpenGL Shading Language)* zwischen fixen Teilen (Rasterisierung, z-Buffer, …)
  - OpenGL ES (*for Embedded Systems*, z.B. Smart Phones, Tablets, …)
    - schlanker; beschränkter Funktionsumfang
    - in modernen Webbrowsern unter dem Namen WebGL nutzbar
- Direct3D
  - Windows-/Xbox-spezifisch
  - Version 2.0-7.0: *Fixed Function Pipeline*
  - Version 8.0-…: frei programmierbare *Shader (HLSL: High-Level Shading Language)* zwischen fixen Teilen (Rasterisierung, z-Buffer, …)

OpenGL

- **Früher**: *Immediate Mode*
  - Vertices (und ggf. Normalen, Farben, etc.) werden one-by-one mit je einem API-Funktionsaufruf an den Grafikchip gesendet und direkt verarbeitet.
  - Im Code zu erkennen an glBegin(…), glEnd(…)
  - Ineffizient und überholt; seit OpenGL 3.0 (und in OpenGL ES / WebGL grundsätzlich) nicht unterstützt.

- **Heute**: *Vertex Buffer Objects (Retained Mode)*
  - Beliebig viele Daten (*Buffer*) werden mit einem API-Funktionsaufruf an den Treiber delegiert. Dieser kann diese (unabhängig vom Hauptprogrammablauf) zu geeigneter Zeit in den Grafikspeicher zur späteren Verwendung übertragen.
  - CPU und GPU können dadurch asynchron arbeiten.

WebGL (mit Vertex Buffer Objects)

```
const canvas = document.querySelector('#glcanvas');
const gl = canvas.getContext('webgl');

const positions = [
     0.0, 0.0, 0.0,
     1.0, 0.0, 0.0,
     1.0, 1.0, 0.0,
     0.0, 1.0, 0.0
];

const indices = [
    0,  1,  2,
    0,  2,  3
];

const positionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

const indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);
```

WebGL (mit Vertex Buffer Objects)

```
programInfo = createMyVeryOwnShaderProgram();

gl.useProgram(programInfo.program);

gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

const numComponents = 3;
const type = gl.FLOAT;
const normalize = false;
const stride = 0;
const offset = 0;
gl.vertexAttribPointer(
    programInfo.attribLocations.vertexPosition,
    numComponents,
    type,
    normalize,
    stride,
    offset);
gl.enableVertexAttribArray(
    programInfo.attribLocations.vertexPosition);
```

WebGL (mit Vertex Buffer Objects)

```
var modelViewMatrix = ...
var lightPosition = ...
var cAmbient = ...

gl.uniformMatrix4fv(
    programInfo.uniformLocations.modelViewMatrix,
    false,
    modelViewMatrix);

gl.uniform3fv(
    programInfo.uniformLocations.lightPosition,
    lightPosition);

gl.uniform1f(
    programInfo.uniformLocations.cAmbient,
    cAmbient);
```

WebGL (mit Vertex Buffer Objects)

```
gl.enable(gl.DEPTH_TEST);
gl.depthFunc(gl.LEQUAL);

gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

const type = gl.UNSIGNED_SHORT;
const offset = 0;
gl.drawElements(gl.TRIANGLES, indices.length   , type, offset);
```

WebGL (mit Vertex Buffer Objects)

```
function createMyVeryOwnShaderProgram(gl, vsSource, fsSource) {

    const vertexShader   = loadShader(gl, gl.VERTEX_SHADER, vsSource);
    const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER, fsSource);

    const shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert('Unable to initialize the shader program: ' + gl.getProgramInfoLog(shaderProgram));
      return null;
    }

    var programInfo = {
      program: shaderProgram,
      attribLocations: {
          vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition')
      },
      uniformLocations: {
          modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),
          lightVector: gl.getUniformLocation(shaderProgram, 'uLightPosition'),
          cAmbient: gl.getUniformLocation(shaderProgram, 'uCAmbient')
      },
    };

    return programInfo;
}
```

GLSL

```
// Vertex shader
const vsSource = `
    attribute vec4 aVertexPosition;

    uniform mat4 uModelViewMatrix;
    uniform vec3 uLightPosition;
    uniform float uCAmbient;

    varying vec3 vAmbientTerm;

    void main(void) {
        gl_Position = uModelViewMatrix * aVertexPosition;

        vAmbientTerm = uCAmbient * ...
    }
`;

// Fragment shader
const fsSource = `
    varying vec3 vAmbientTerm;

    void main(void) {
        gl_FragColor = vec4(vAmbientTerm + ..., 1.0);
    }
`;
```

GLSL

- **Datentypen:**

```
float,  int,  bool                  …

vec2,  vec3,  vec4                  2D, 3D, 4D floating point vector
ivec2, ivec3, ivec4                 2D, 3D, 4D integer vector
bvec2, bvec3, bvec4                 2D, 3D, 4D boolean vector

mat2, mat3, mat4                    2x2, 3x3, 4x4 floating point matrix
```

- **Input/Output-Variablenarten:**

```
uniform     Variable, konstant für alle Vertices/Fragments (z.B. Lichtposition);
            vom Host gegeben, im Vertex- und Fragment-Shader lesbar.

attribute   pro-Vertex Variable (z.B. Position, Normale, Farbe)
            vom Host gegeben, im Vertex-Shader lesbar.

varying     im Vertex-Shader schreibbar, im Fragment-Shader interpoliert lesbar.
```

Müssen immer global definiert werden (d.h. nicht innerhalb von main() oder anderer Funktion)

- **Built-In Variablen:**

```
gl_Position     vec4 für die finale Vertex-Position. Im Vertex Shader zu schreiben.

gl_FragColor    vec4 für die finale Fragment-Farbe. Im Fragment Shader zu schreiben.
gl_FragDepth    z-Wert des Fragments (für z-Buffer). Im Fragment Shader anpassbar.
```

GLSL

- **Built-In Funktionen:**

```
dot         Skalarprodukt
cross       Kreuzprodukt
normalize   Vektor auf Länge 1 normalisieren
clamp       Auf gegebenen Wertebereich beschränken
sin, cos, tan, pow, exp, log, max, min ...
```

- **Vektoren:**
  - **Initialisierung:**

```
vec4 n = vec4(1.0, 0.0, 0.5, -1.0);
```

  - **Zugriff:**

```
n[0] = n.x = n.r = n.s
n[1] = n.y = n.g = n.t
n[2] = n.z = n.b = n.p
n[3] = n.w = n.a = n.q
```

  - **Multi-Zugriff:**

```
n.rgba;    ist ein vec4 (= n)
n.rgb;     ist ein vec3
n.b;       ist ein float
n.yz;      ist ein vec2
n.xgza;    illegal
```

  - **Swizzling:**

```
vec3 v = vec3(1.0, 2.0, 3.0);
vec4 w = v.xzyy;      w: (1.0, 3.0, 2.0, 2.0)    (dabei xyzw, rgba, stpq nicht mischbar)
```

WebGL1

JavaScript:

```
canvas.getContext('webgl')
```

Shader:

```
attribute

varying (Vertex Shader)

varying (Fragment Shader)

texture1D/2D
```

```
precision mediump float
```
(am Anfang des FS)

WebGL2

```
canvas.getContext('webgl2')
```

```
#version 300 es
```
direkt(!) am Anfang

```
in

out (Vertex Shader)

in (Fragment Shader)

texture
```

Bis zum nächsten Mal!

Media Informatics
Graphics & Geometric Computing
Prof. Dr. M. Campen