Interaktive Computergrafik

Prof. Dr. Frank Steinicke
Human-Computer Interaction
Fachbereich Informatik
Universität Hamburg

Interaktive Computergrafik

Kapitel WebGL

Prof. Dr. Frank Steinicke
Human-Computer Interaction, Universität Hamburg

Interaktive Computergrafik

Kapitel WebGL

Einführung

Rückblick: Grafikkarte

- Grafikprozessor (*engl. Graphics Processing Unit, GPU*) der Grafikkarte übernimmt rechenintensive Aufgaben der ICG und entlastet Hauptprozessor
  = **hardwarebeschleunigtes Rendering**
- mittlerweile sind Grafikprozessoren nicht mehr fest verdrahtet, sondern wie CPUs flexibel **programmierbar**

Rückblick: Grafikkarte

- Wie können Hardwarebeschleunigung und Programmierbarkeit der Grafikkarte genutzt werden?

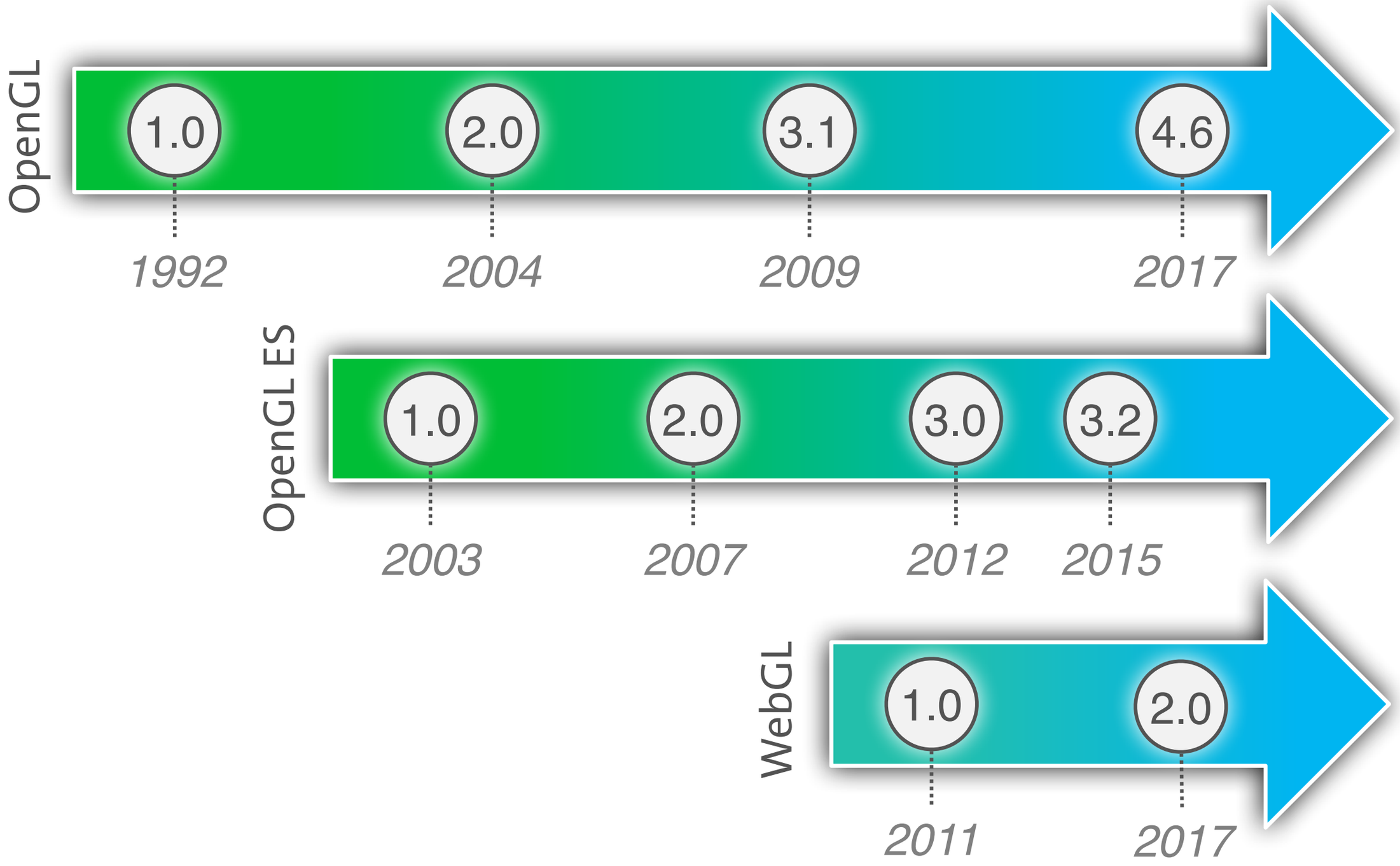
→ Schnittstelle (OpenGL / WebGL)

Web Graphics Library

- WebGL ist Spezifikation einer plattform-unabhängigen **Software-Schnittstelle für Grafik-Hardware**

➡ erlaubt hardwarebeschleunigtes Rendering von 2D- und 3D-Grafiken in Webbrowsern

Historie


OpenGL
1.0
2.0
3.1
4.6
1992
2004
2009
2017
OpenGL ES
1.0
2.0
3.0
3.2
2003
2007
2012
2015
WebGL
1.0
2.0
2011
2017

Historie (WebGL 1.0)

- 2009: Gründung der **„WebGL Working Group“** durch Khronos Group

  → Beteiligung von Mozilla, Apple, Google, AMD, Ericsson, Nvidia und Opera

- 2011: Release **WebGL 1.0**

  → Basis: OpenGL ES 2.0 (= reduzierte Version für eingebettete Systeme)

  → Unterstützt von Chrome, Firefox, *Safari, Opera*

Historie (WebGL 2.0)

- 2012: Start der Entwicklung von **WebGL 2.0**
- 2017: Release der finalen Version

→ Basis: OpenGL ES 3.0

→ Unterstützt von Chrome, Firefox, Opera

WebGL Vorteile

- keine **Installation** notwendig
- keine zusätzlichen **Libraries** erforderlich (z.B. für Behandlung von Nutzereingaben)
- keine speziellen **Systemanforderungen**

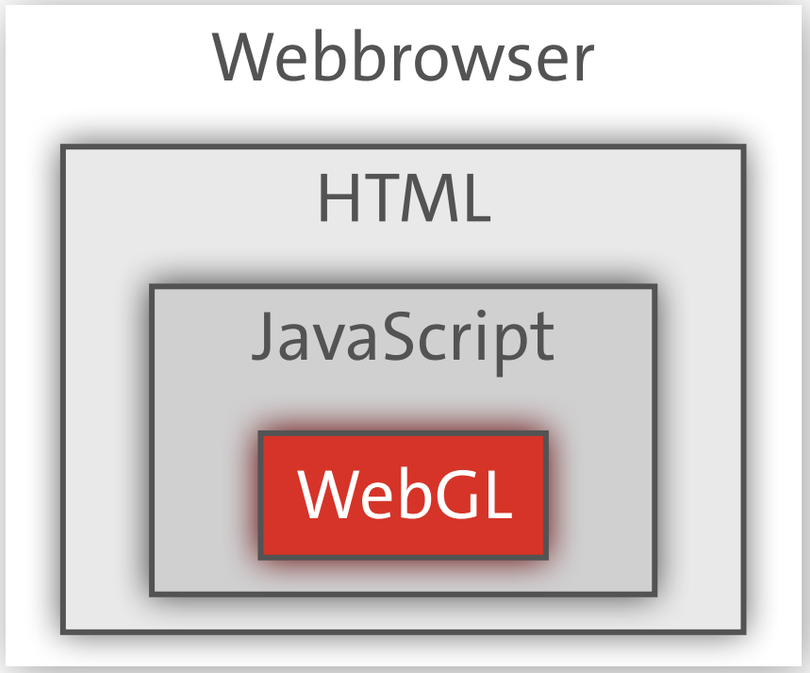
Einschränkungen

- WebGL unterstützt nicht alle OpenGL-Features, z.B.
  - Geometry Shader
  - Tessellation Shader
  - Compute Shader

WebGL

Prinzip

- Aufruf von JavaScript-Funktionen im HTML-Code
- Interpretation zur Laufzeit
- GPU Computing innerhalb von JavaScript Code → **hardwarebeschleunigte 3D-Grafiken im Browser**



```
<!DOCTYPE HTML>
<html>
  <head>
    <script>
      window.onload = function init() {
        const cv = document.getElementById("canvas");
        const gl = …
        …
        gl.drawArrays(gl.TRIANGLES, 0, 3);
      }
    </script>
  </head>
  <body>
    <canvas id="canvas" width="512" height="512">
    </canvas>
  </body>
</html>
```

HTML

- steht für **H**yper**t**ext **M**arkup **L**anguage
- beschreibt die **Struktur** von Webseiten
- nutzt **Tag-Paare** (Starttag/Endtag) zur Definition von inhaltlichen Blöcken (z.B. * Ein Text *)
- formt mit JavaScript und Cascading Style Sheets (*CSS*) **Basis des World Wide Web**

JavaScript

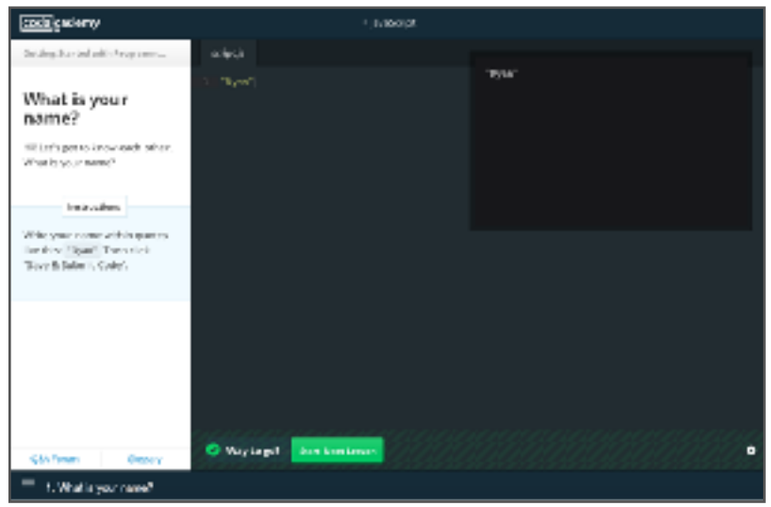
- ist eine plattformübergreifende, objektorientierte Skriptsprache
- ermöglicht **dynamisches HTML** in Webbrowsern (z.B. Reaktion auf Benutzereingaben)
- unterscheidet sich grundlegend von Java (z.B. dynamische Typisierung)

JavaScript
Einbindung in HTML

- Direkt in HTML-Code oder als **externe JavaScript-Datei**
- Externe Datei für größere Projekte bevorzugt, aufgrund besserer Übersichtlichkeit, Wartbarkeit und Wiederverwendbarkeit

```
<head>
  <script src="externalFile.js"></script>
</head>
```

JavaScript
Tutorials



- http://www.w3schools.com/js/default.asp
- http://wiki.selfhtml.org/wiki/JavaScript/Tutorials
- http://www.codecademy.com/learn/introduction-to-javascript

Interaktive Computergrafik

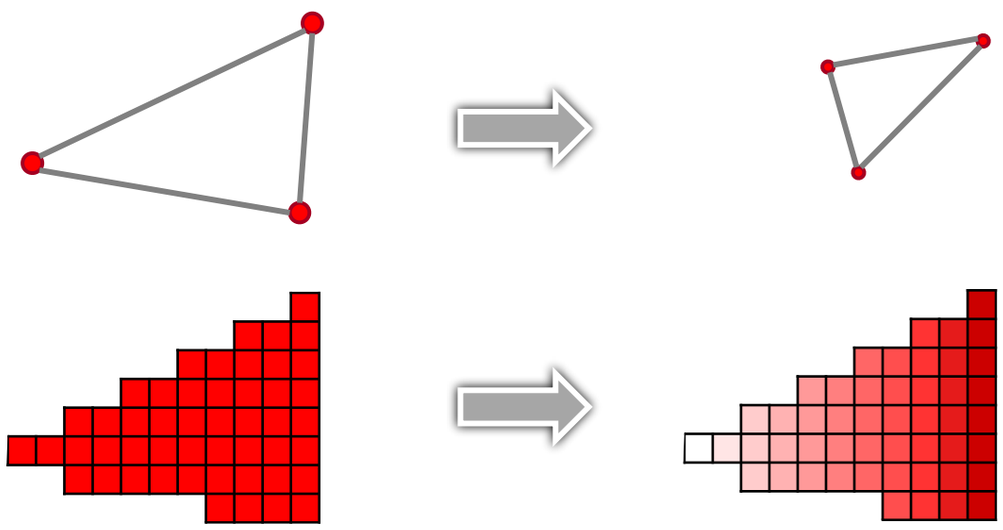
Kapitel WebGL

WebGL-Pipeline

Shader

Allgemein

- **Shading** = Veränderung einzelner Eckpunkte (*Vertices*) bzw. Pixel (*Fragmente*) innerhalb der Grafikpipeline

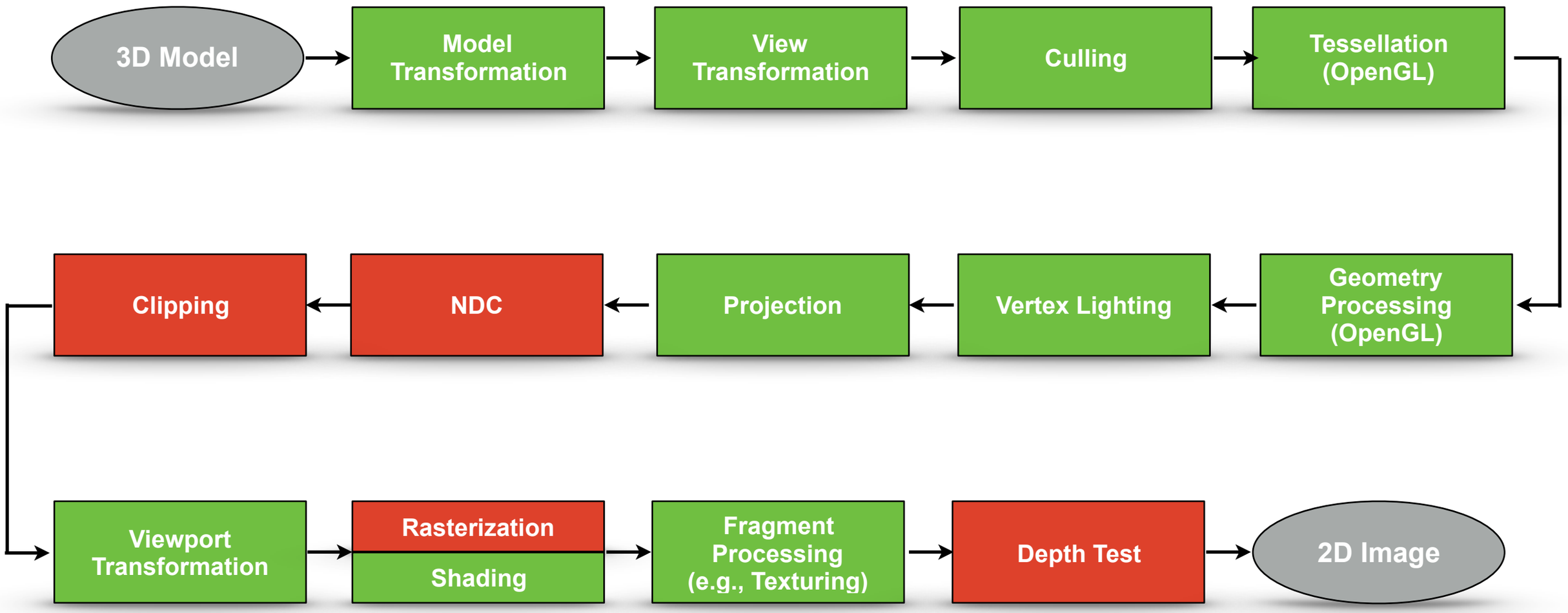


Shader

WebGL

- **Programmierbare** Shader, d.h. anwendungsspezifische, **frei gestaltbare** Programme, die die früheren vordefinierten Stufen der Rendering Pipeline ersetzen
- 2 Shadertypen in WebGL
  1. **Vertex Shader**
  2. **Fragment Shader**

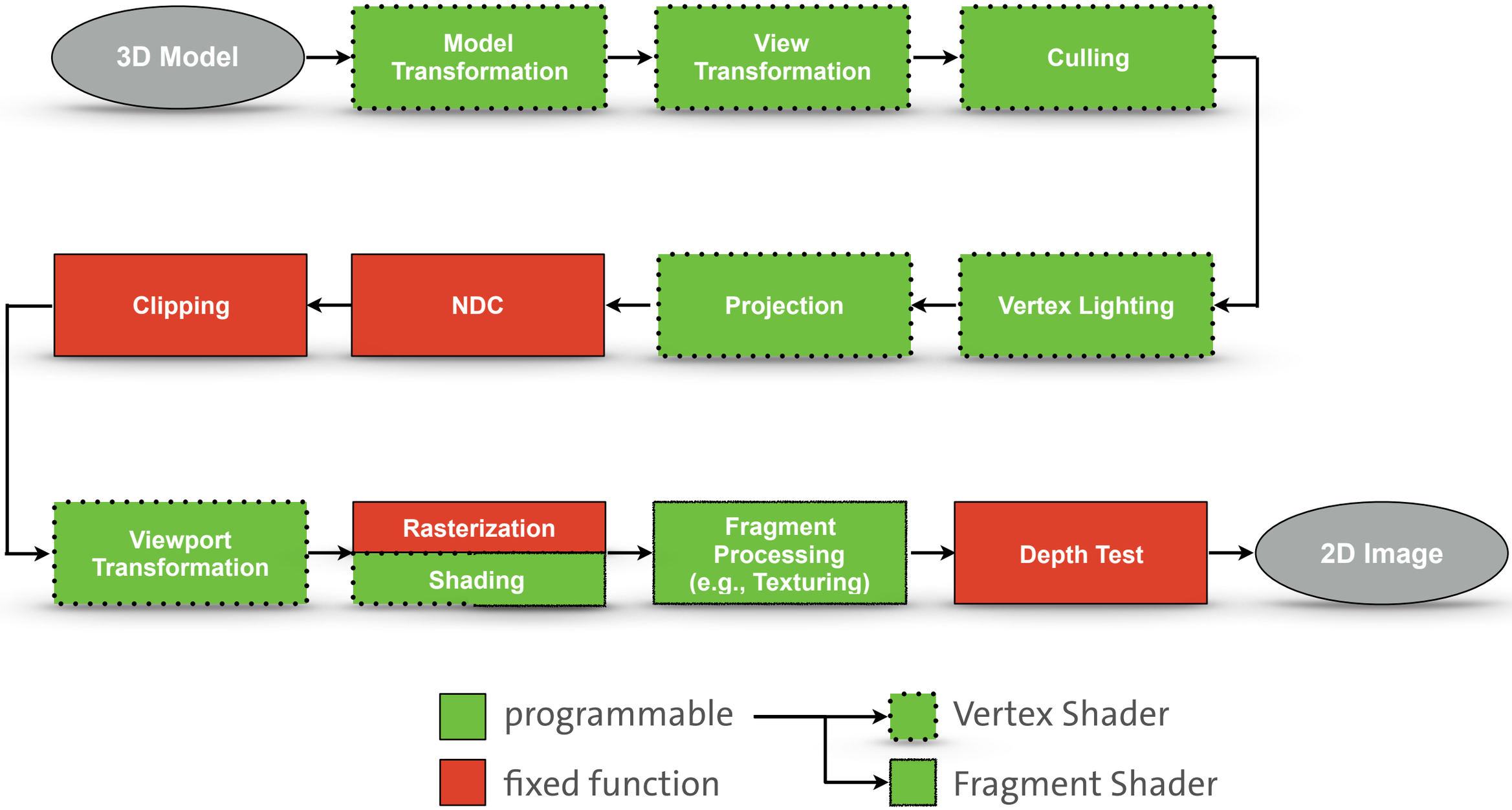
3D Rendering Pipeline



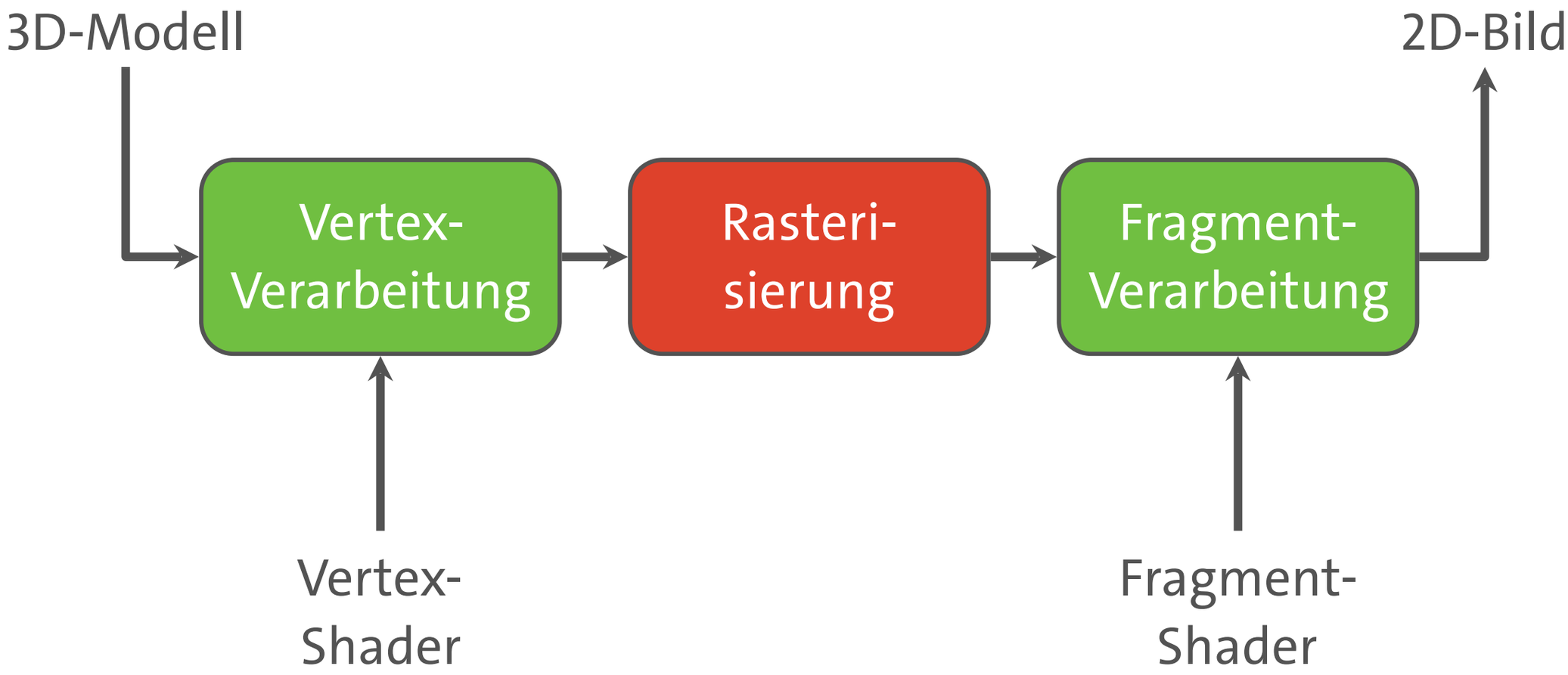
programmable

fixed function

WebGL-Pipeline



WebGL-Pipeline
(vereinfachte Version)



Was brauchen wir dafür?

Anwendungs-
programm

JavaScript + WebGL API (+ HTML)

Vorlesung + Übung

Vertex-
Shader

Fragment-
Shader

GLSL (+ HTML)
(OpenGL Shading Language)

Übung

GLSL?

index.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta charset="utf-8"/>
        <title>WebGL Example</title>

        <script id="vertex-shader" type="x-shader/x-vertex">#version 300 es
            in vec4 vPosition;

            void main()
            {
                gl_Position = vPosition;
            }
        </script>
        <script id="fragment-shader" type="x-shader/x-fragment">#version 300 es
            precision mediump float;
            out vec4 fColor;

            void main()
            {
                fColor = vec4(1.0, 0.0, 0.0, 1.0);
            }
        </script>

        <script type="text/javascript" src="common/initShaders.js">
```

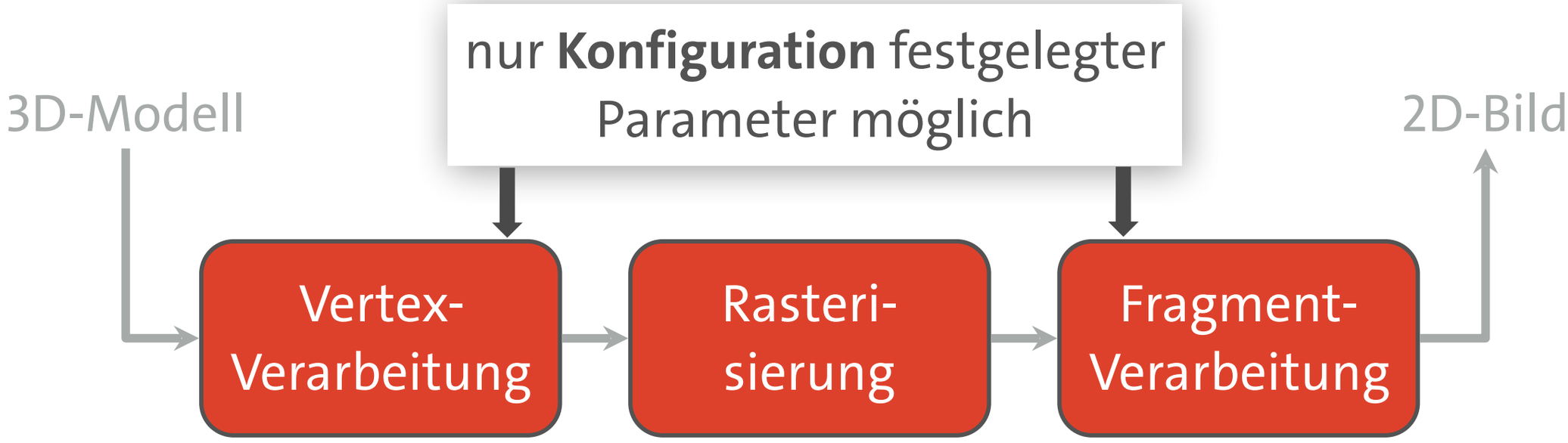
Interaktive Computergrafik

Kapitel WebGL

WebGL Paradigma

Traditionalles Paradigma

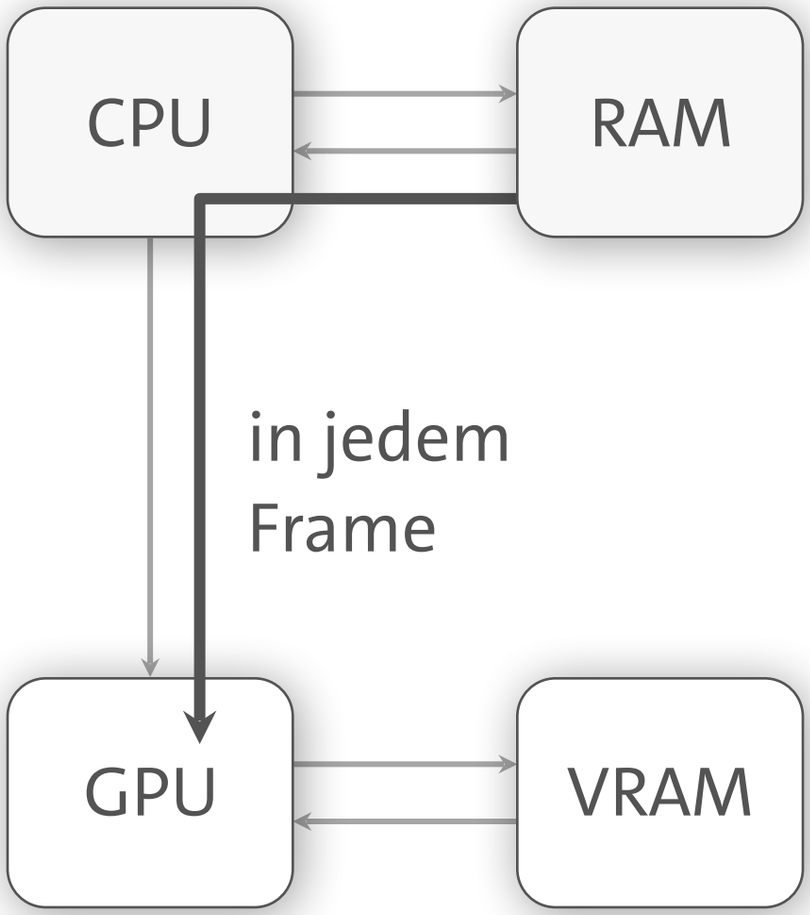
- „Fixed Function“



- seit OpenGL 3.1 (2009) aus Spezifikation entfernt → in WebGL direkt ausgelassen

Traditionalles Paradigma

„Direct mode rendering“



- Speicherung der Daten in Hauptspeicher
- 1 Funktionsaufruf pro Eckpunkt und Attribut
- ineffizient bei großer Anzahl an Eckpunkten

Paradigmenwechsel

Traditionelles Paradigma

Probleme:

- geringe Effizienz für große Szenen
- keine Flexibilität

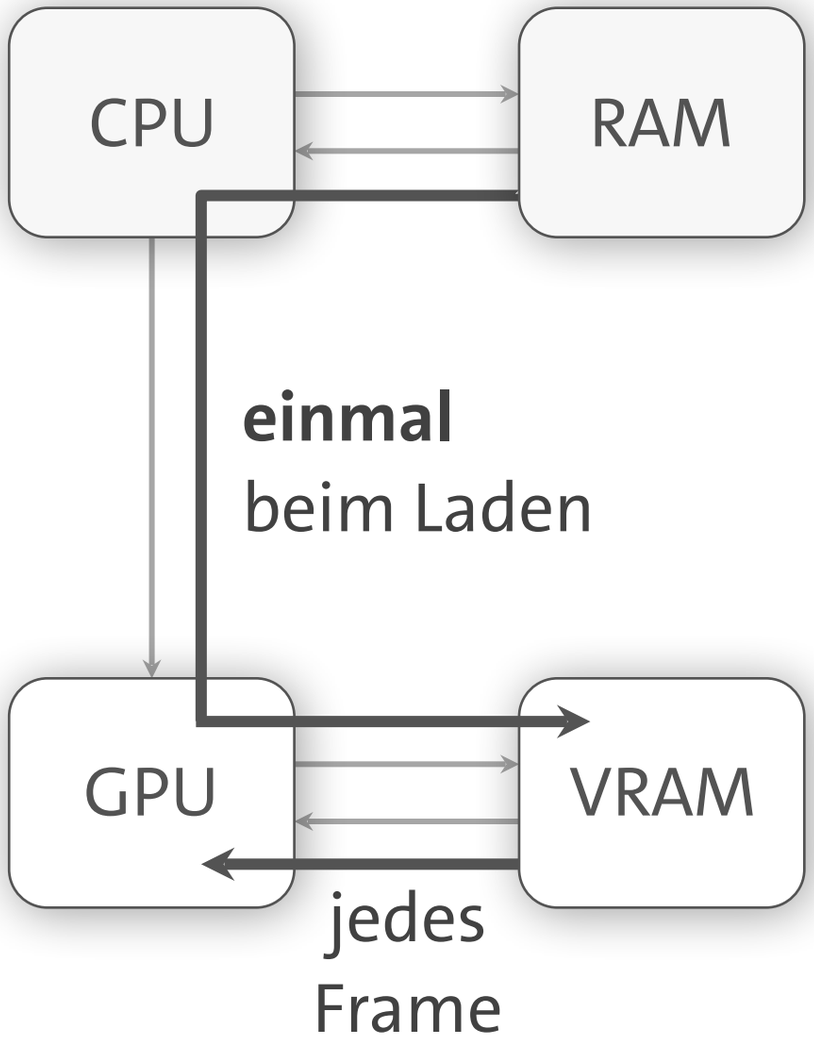
Neues Paradigma

Lösung:

- **Bufferobjekte**
  (Vertex Buffer Objects - VBOs)

Neues Paradigma

- Zentrales Konzept: **Bufferobjekte**

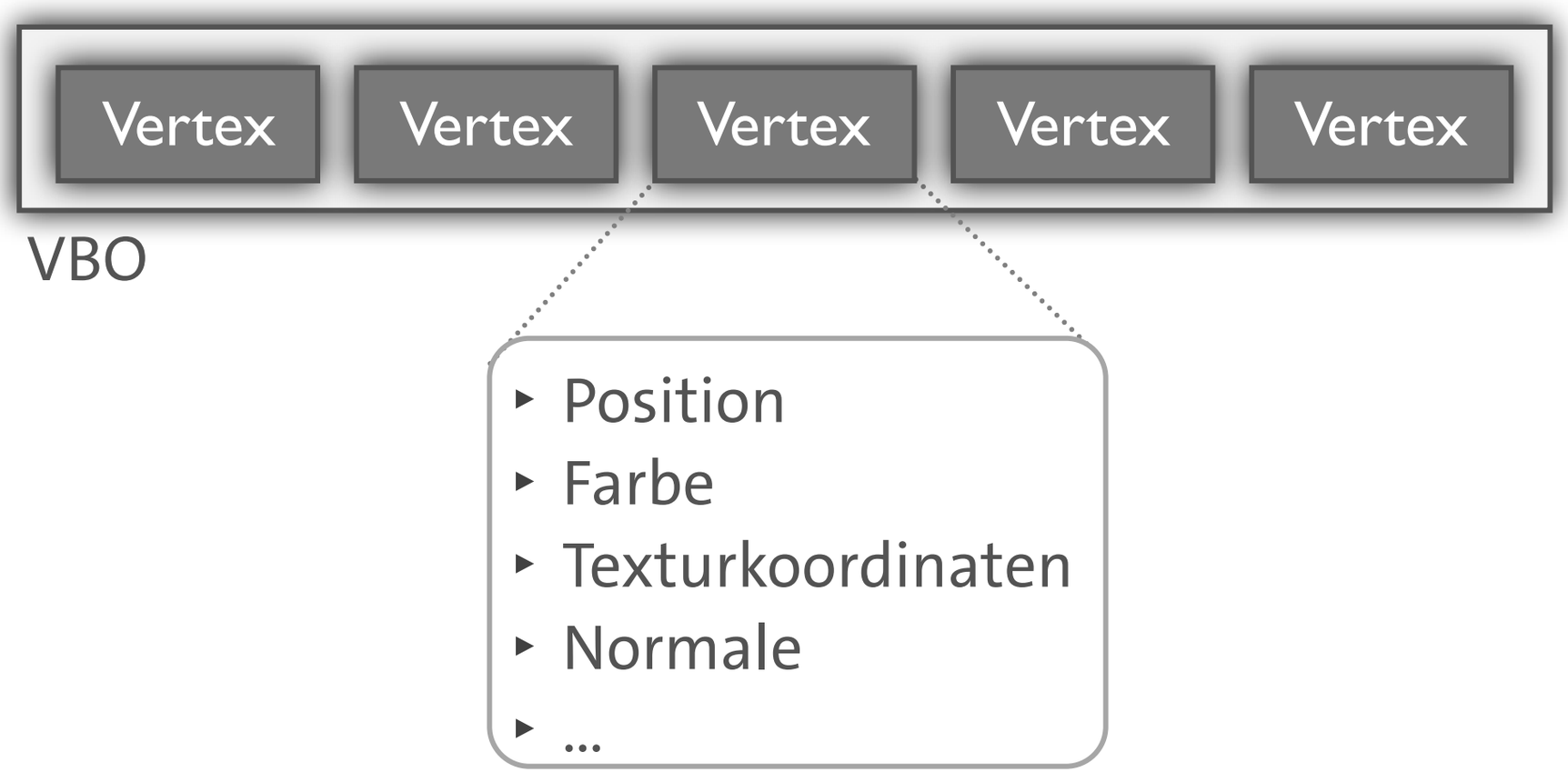


- Speicherung der Daten in Grafikspeicher
- Änderungen müssen von der CPU an die GPU weitergeleitet werden

Vertex Buffer Objects

Definition

- Eckpunkt eines Primitivs = **Vertex**
- Sammlung von Vertices = **VBO**



Vertex Buffer Objects

Layouts

- ein VBO pro Attribut

- ein VBO für alle Attribute (nacheinander)

- ein VBO für alle Attribute (abwechselnd)

Vertex Buffer Objects

Layouts

- Mischform

→ Format für alle Vertices in einem VBO gleich

→ statische und dynamische Attribute trennen

Vertex Buffer Objects

Interpretation

0.6	-0.6	-0.4	-0.4	0.7	0.2	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0

?

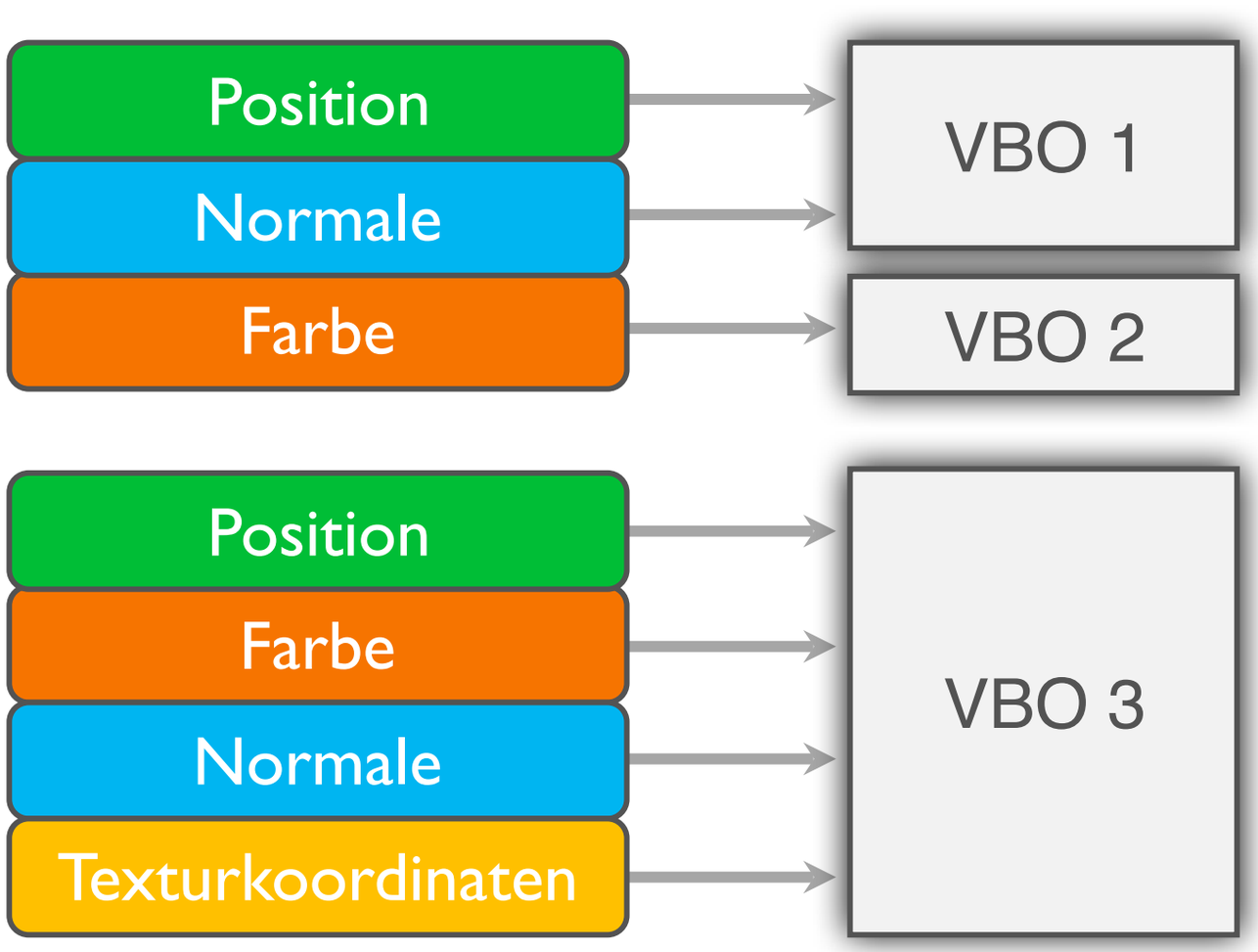
oder

oder

Vertex Buffer Objects

Interpretation

- Speicherung von Metadaten zu VBOs, damit Shader diese interpretieren können

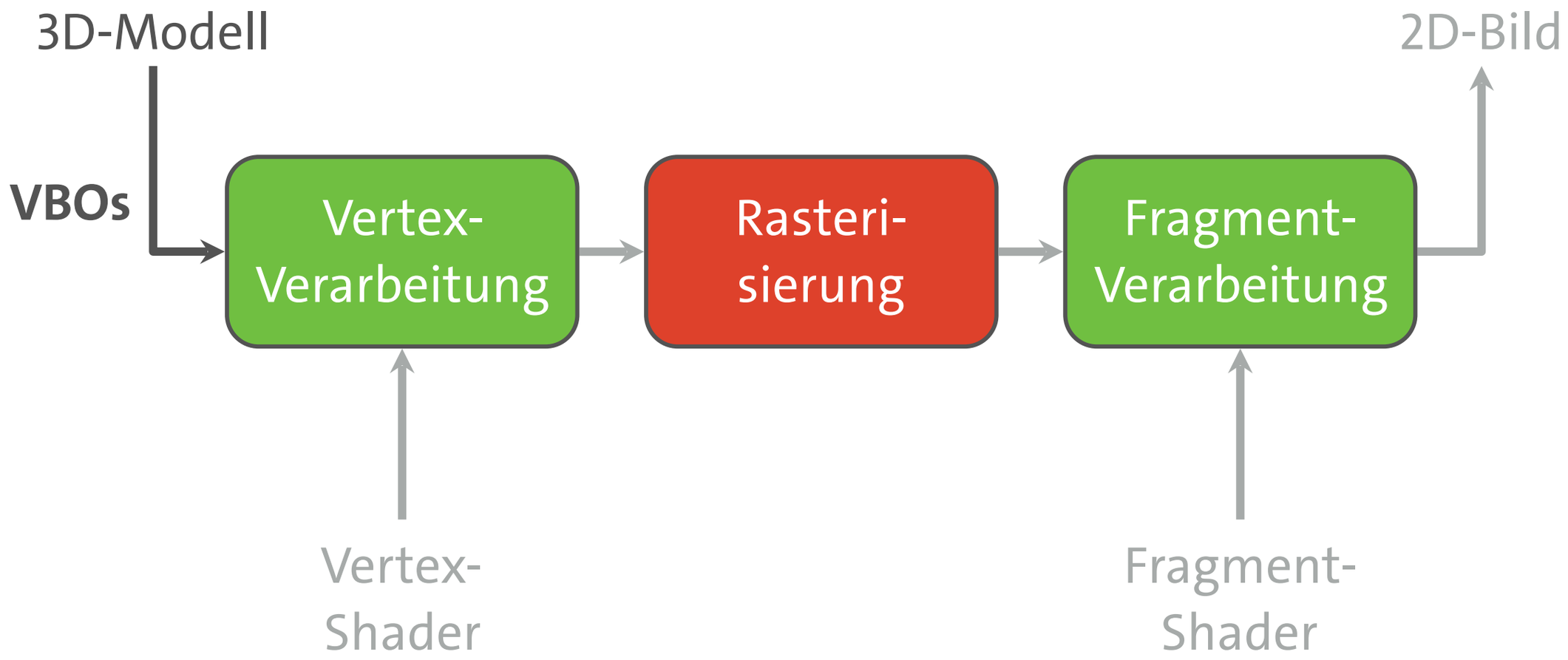


Vertex Buffer Objects

Interpretation

- Was steht in dem Array?
- Wo steht es in dem Array?
- Wieviele Komponenten hat ein Attribut?
- Welchen Typ hat ein Array-Element?

WebGL-Pipeline



Interaktive Computergrafik

Kapitel WebGL

Umsetzung

1. WebGL-Kontext erstellen
2. Zeichenfläche vorbereiten
3. Geometrie festlegen
4. Shader integrieren
5. VBOs anlegen
6. Daten in VBOs laden
7. VBOs mit Shadervariablen verknüpfen
8. Rendern

1. **WebGL-Kontext erstellen**
2. Zeichenfläche vorbereiten
3. Geometrie festlegen
4. Shader integrieren
5. VBOs anlegen
6. Daten in VBOs laden
7. VBOs mit Shadervariablen verknüpfen
8. Rendern

WebGL-Kontext erstellen

- **Canvas** („Leinwand“) = HTML5-Element, das als Container für Grafiken dient, die mittels JavaScript gezeichnet werden können

HTML

```
<canvas id="gl-canvas"
        width="512" height="256">

Dein Browser unterstützt kein WebGL.

</canvas>
```

WebGL-Kontext erstellen

- Canvas-Element weist zwei Kontexte zum Zeichnen auf:
  - **CanvasRenderingContext2D** für 2D-Zeichnungen
  - **WebGLRenderingContext** für (hardwarebeschleunigte) 2D- und 3D-Zeichnungen

WebGL-Kontext erstellen

JS

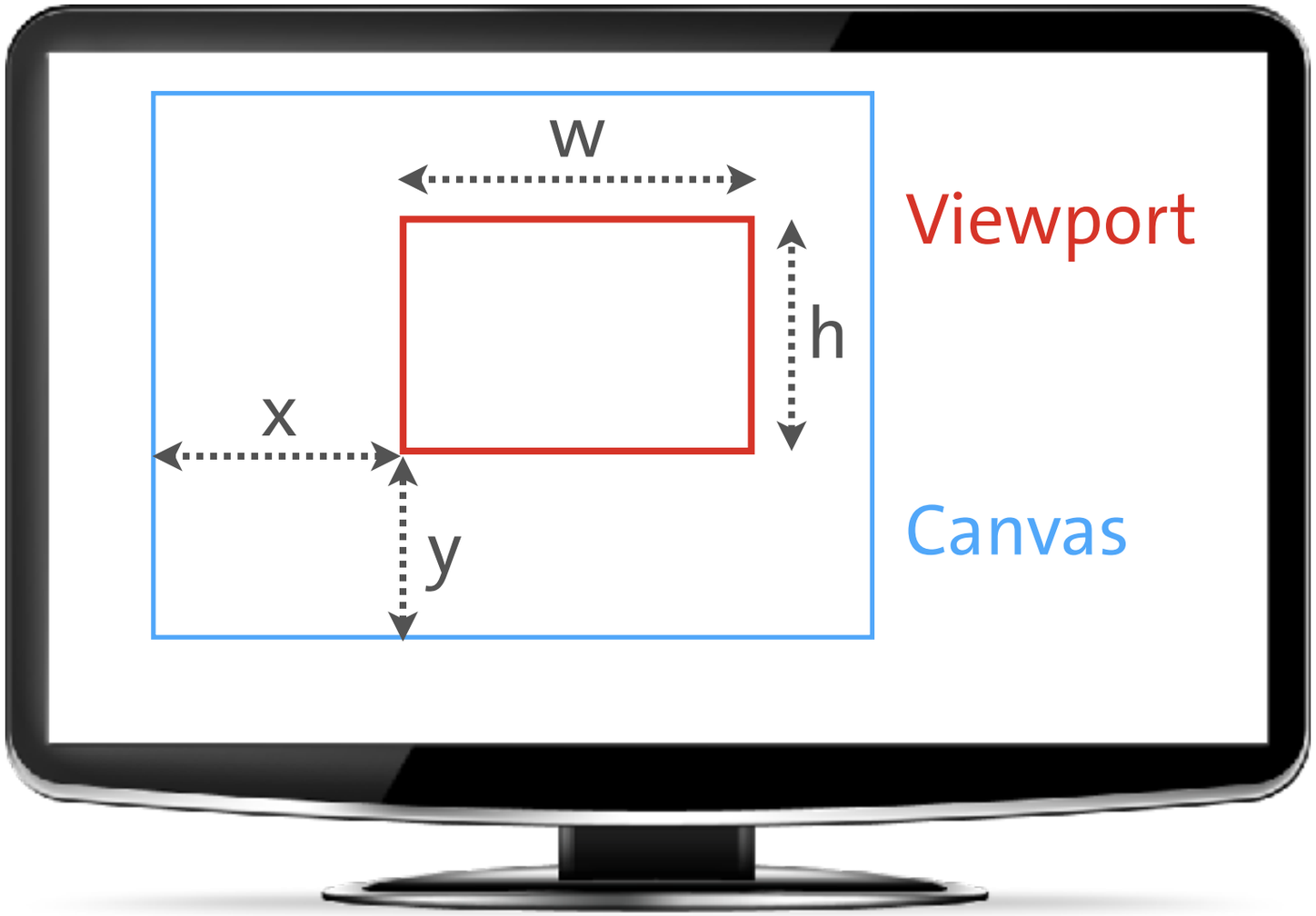
```
const canvas =
   document.getElementById("gl-canvas");

const gl = canvas.getContext("webgl2");
```

- WebGL-Kontext =

  JavaScript-Objekt, das alle WebGL-**Methoden**, -**Eigenschaften** und -**Konstanten** enthält

1. WebGL-Kontext erstellen
2. **Zeichenfläche vorbereiten**
3. Geometrie festlegen
4. Shader integrieren
5. VBOs anlegen
6. Daten in VBOs laden
7. VBOs mit Shadervariablen verknüpfen
8. Rendern

Zeichenfläche vorbereiten




```
gl.viewport(0, 0, canvas.width, canvas.height);
```

x y w h

Zeichenfläche vorbereiten

- Einmalig zur Initialisierung:

  Festlegung eines Standardwertes für Hintergrundfarbe der Zeichenfläche

```
gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

r g b a

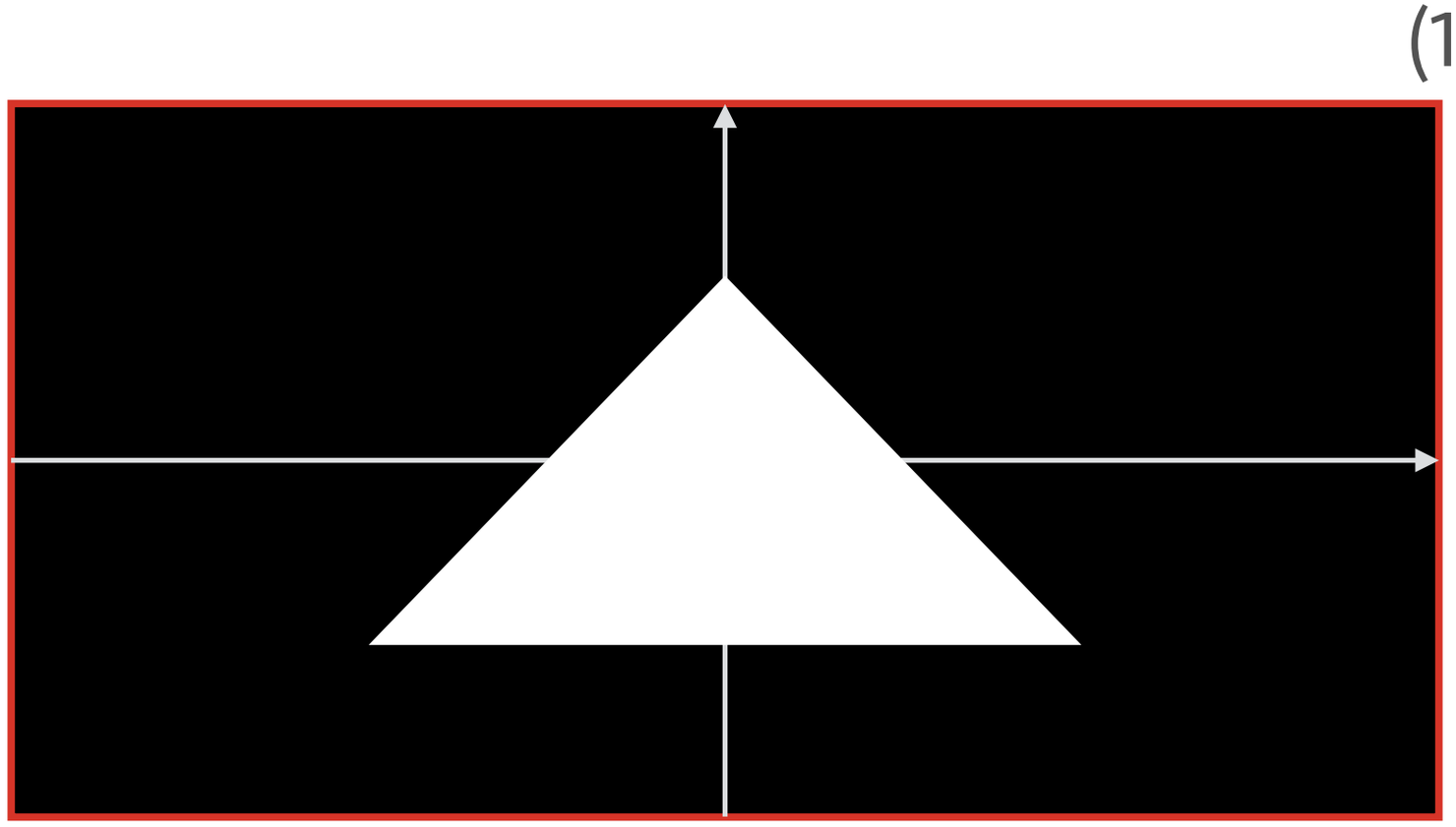
- Vor jedem Frame:

  Rücksetzen der Hintergrundfarbe

```
gl.clear(gl.COLOR_BUFFER_BIT);
```

1. WebGL-Kontext erstellen
2. Zeichenfläche vorbereiten
3. **Geometrie festlegen**
4. Shader integrieren
5. VBOs anlegen
6. Daten in VBOs laden
7. VBOs mit Shadervariablen verknüpfen
8. Rendern

Geometrie festlegen



(Vorsicht: per Default hat Koordinatensystem Ursprung in Mitte des Viewports; nicht links unten wie bei Viewport-Definition selbst!)

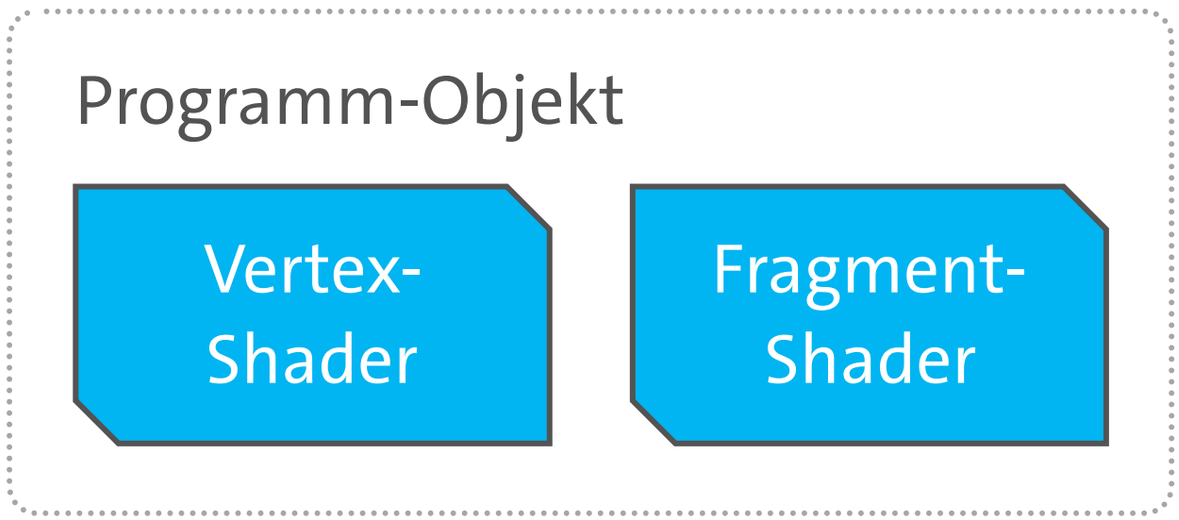
Geometrie festlegen

```
let positions = new Float32Array([
   -0.5, -0.5,      // (x0, y0)
    0.0,  0.5,      // (x1, y1)
    0.5, -0.5       // (x2, y2)
]);
```

1. WebGL-Kontext erstellen
2. Zeichenfläche vorbereiten
3. Geometrie festlegen
4. **Shader integrieren**
5. VBOs anlegen
6. Daten in VBOs laden
7. VBOs mit Shadervariablen verknüpfen
8. Rendern

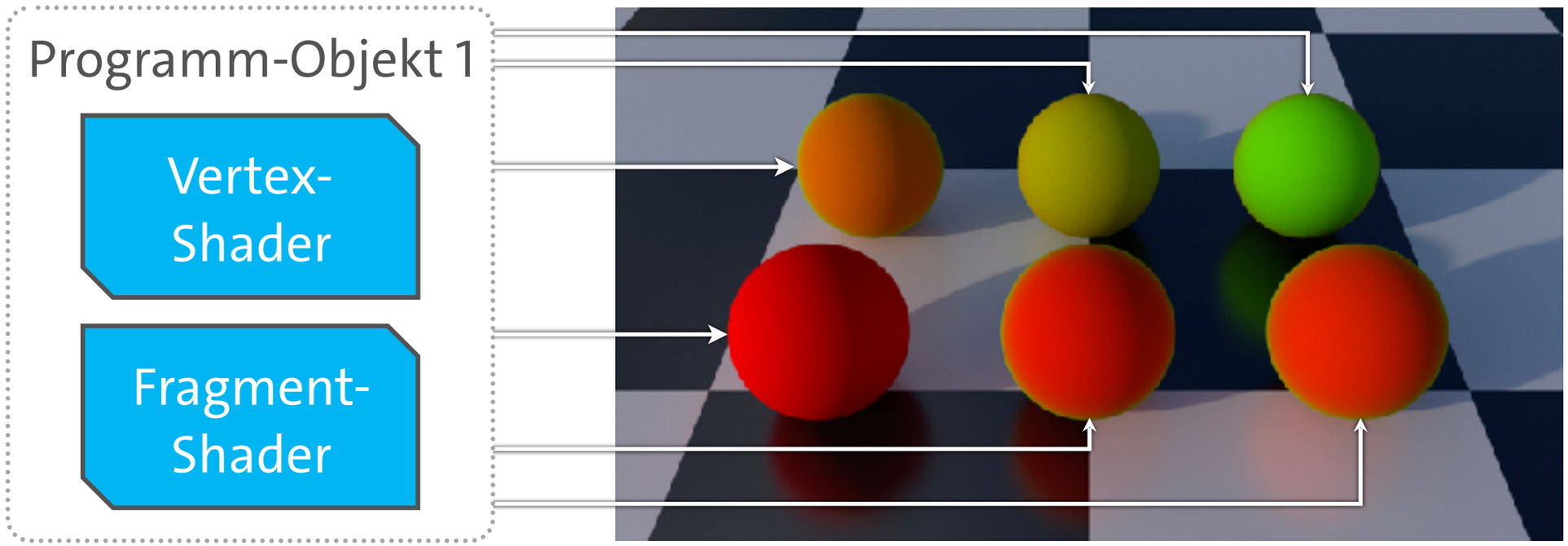
Shader integrieren

- Erinnerung: In WebGL nur 2 Shadertypen: **Vertex Shader** und **Fragment Shader**
- Jeweils 1 Vertex- und 1 Fragment-Shader bilden Programm-Objekt



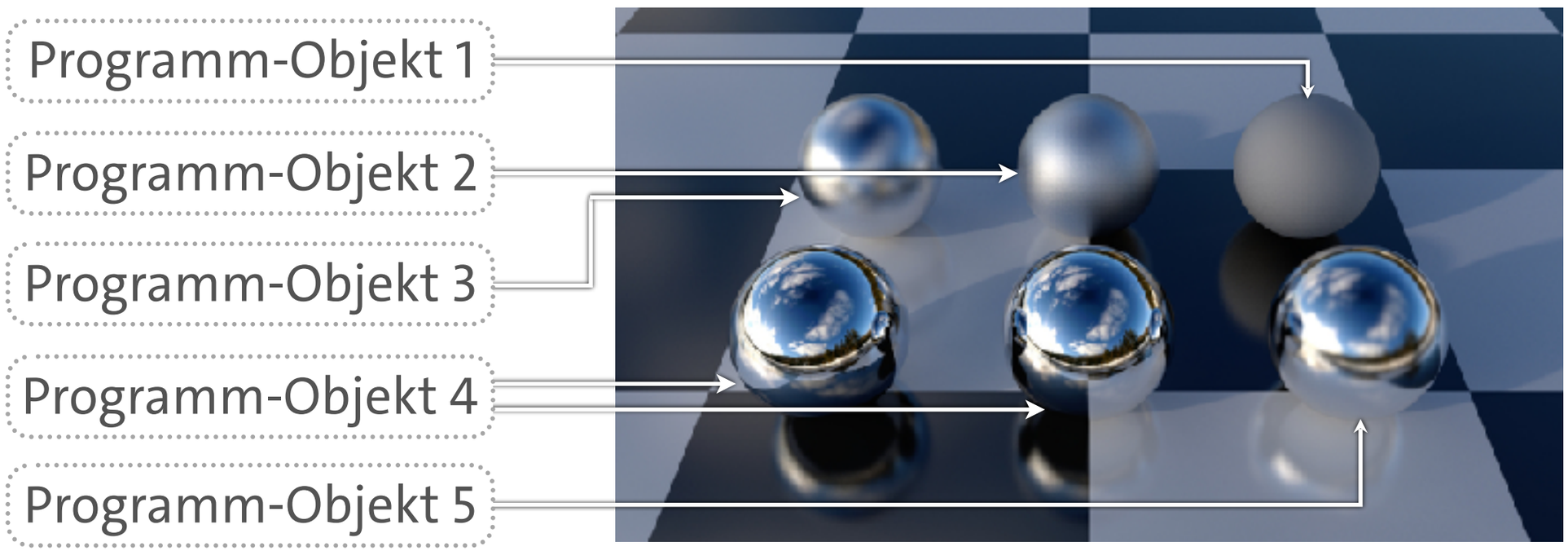
Shader integrieren

- Minimum: 1 Programm-Objekt (= Vertex + Fragment Shader) pro Anwendung

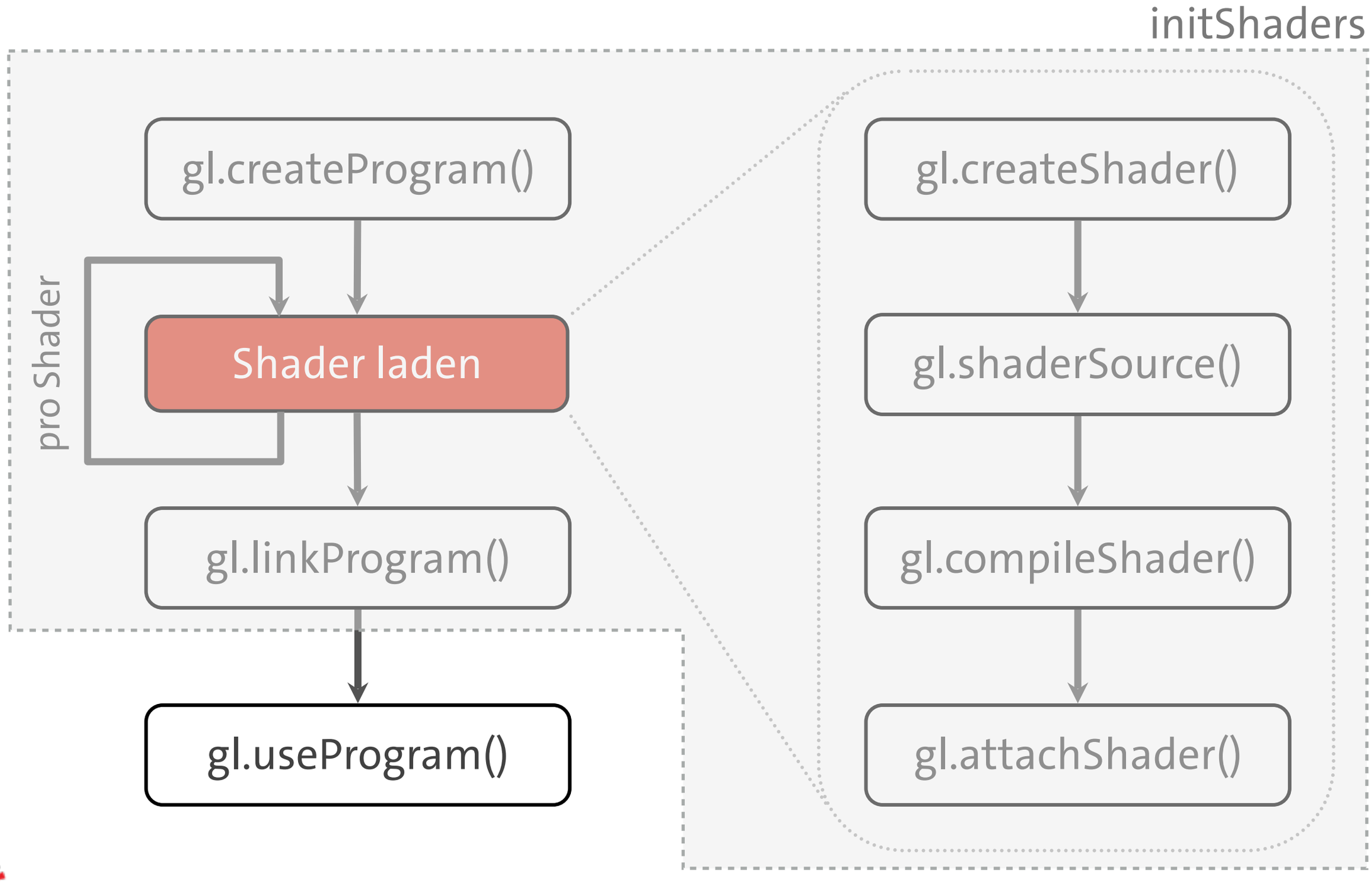


Shader integrieren

- Normalfall: viele Programm-Objekte in einer Anwendung (eins pro Material)



Shader integrieren



Shader integrieren

- Zur Vereinfachung: Auslagerung der Shaderintegration in Funktion ***initShaders*** (in externe Datei *initShaders.js*)

```
const program = initShaders(gl,
    "vertex-shader", "fragment-shader");

gl.useProgram(program);
```

1. WebGL-Kontext erstellen
2. Zeichenfläche vorbereiten
3. Geometrie festlegen
4. Shader integrieren
5. **VBOs anlegen**
6. Daten in VBOs laden
7. VBOs mit Shadervariablen verknüpfen
8. Rendern

VBOs anlegen

1. Neues Buffer-Objekt erzeugen
2. Dieses Objekt zum aktiven VBO machen

(alle folgenden Funktionen beziehen sich auf aktives VBO!)

```
const vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
```

VBOs anlegen

Was heißt aktiv?

- WebGL = **Zustandsautomat** → statt VBO bei jedem Funktionsaufruf zu übergeben, wird er einmal aktiv gesetzt und bis zur Änderung des aktiven VBOs genutzt
- Beispiel (in Pseudocode):

```
aktiviereVBO(vbo1);
rendereVBO;              // rendert vbo1
aktiviereVBO(vbo2);
rendereVBO;              // rendert vbo2
```

1. WebGL-Kontext erstellen
2. Zeichenfläche vorbereiten
3. Geometrie festlegen
4. Shader integrieren
5. VBOs anlegen
6. **Daten in VBOs laden**
7. VBOs mit Shadervariablen verknüpfen
8. Rendern

VBOs befüllen

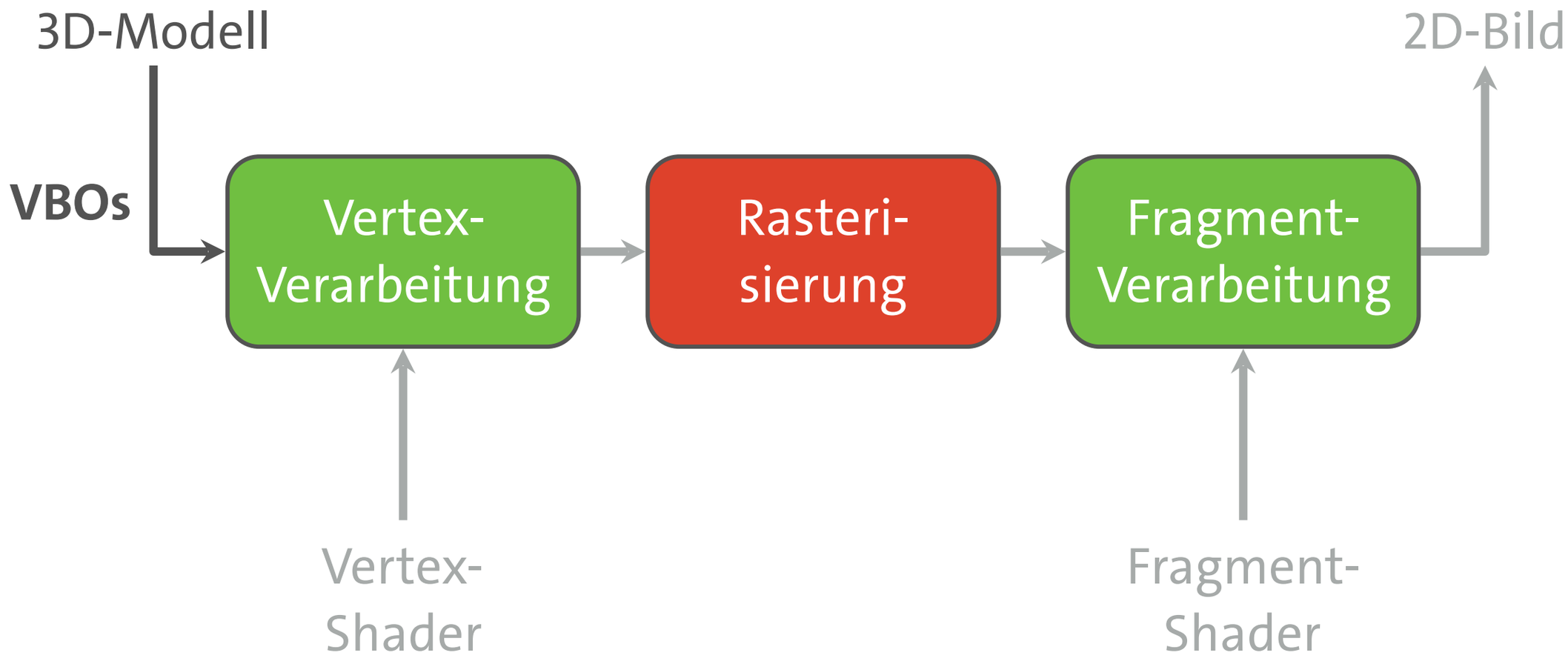
```
gl.bufferData(
    gl.ARRAY_BUFFER,
    positions,
    gl.STATIC_DRAW
);
```

Art des Bufferobjektes

Daten, mit denen der Speicher initialisiert wird

Hinweis auf Nutzung der Daten

VBOs befüllen

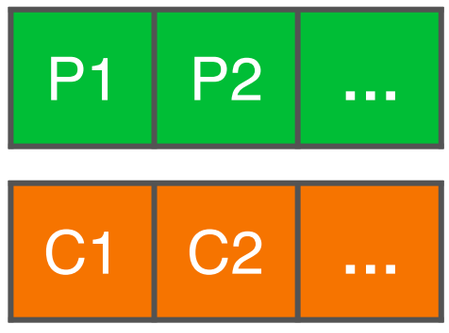


1. WebGL-Kontext erstellen
2. Zeichenfläche vorbereiten
3. Geometrie festlegen
4. Shader integrieren
5. VBOs anlegen
6. Daten in VBOs laden
7. **VBOs mit Shadervariablen verknüpfen**
8. Rendern

VBOs verknüpfen

Anwendungsprogramm
(in JavaScript)

VBOs



Vertex-Shader
(in GLSL)

```
<script id="vertex-shader"
type="x-shader/x-vertex">
    in vec4 vPosition;
    in vec4 vColor;
    ..
</script>
```

VBOs verknüpfen

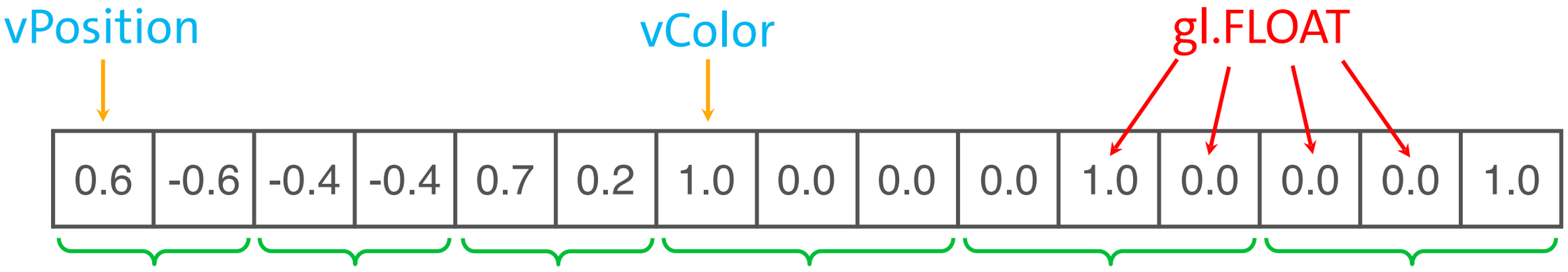
Erzeugung eines Index, der die Attribut-variable im Vertex Shader repräsentiert:

```
const vPosition = gl.getAttribLocation(
program, "vPosition");
```

Aktivierung des Vertexattributs:

```
gl.enableVertexAttribArray(vPosition);
```

VBOs verknüpfen



- Was steht in dem Array?
- Wo steht es in dem Array?
- Wieviele Komponenten hat ein Attribut?
- Welchen Typ hat ein Array-Element?

VBOs verknüpfen

Interpretation der Bufferobjekte festlegen:

```
gl.vertexAttribPointer(
    vPosition,              → Was?
    2,                      → Wieviele Komponenten?
    gl.FLOAT,               → Welcher Typ?
    false,                  → Normalisierung?
    0,                      → Welches VBO-Layout?
    0                       → Wo im VBO?
);
```

1. WebGL-Kontext erstellen
2. Zeichenfläche vorbereiten
3. Geometrie festlegen
4. Shader integrieren
5. VBOs anlegen
6. Daten in VBOs laden
7. VBOs mit Shadervariablen verknüpfen
8. **Rendern**

Rendern

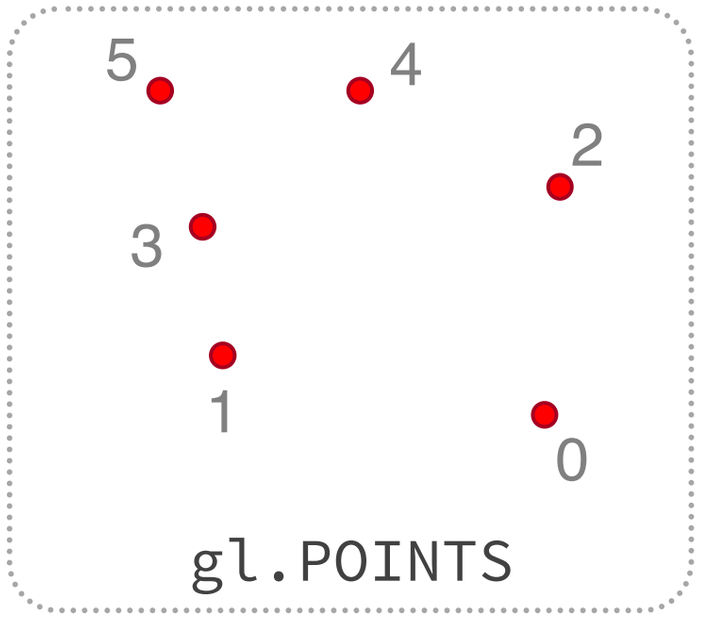
```
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Typ der zu rendern-den Primitiven

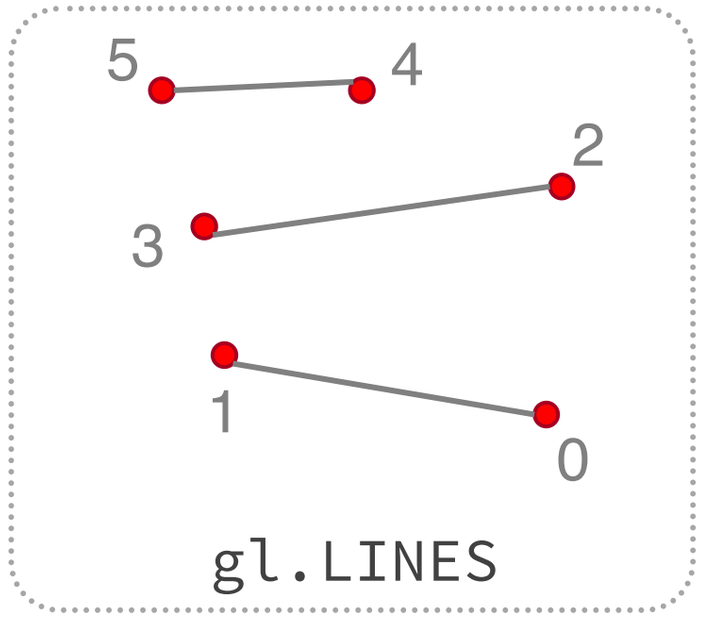
Index des ersten Vertex

Anzahl Vertices

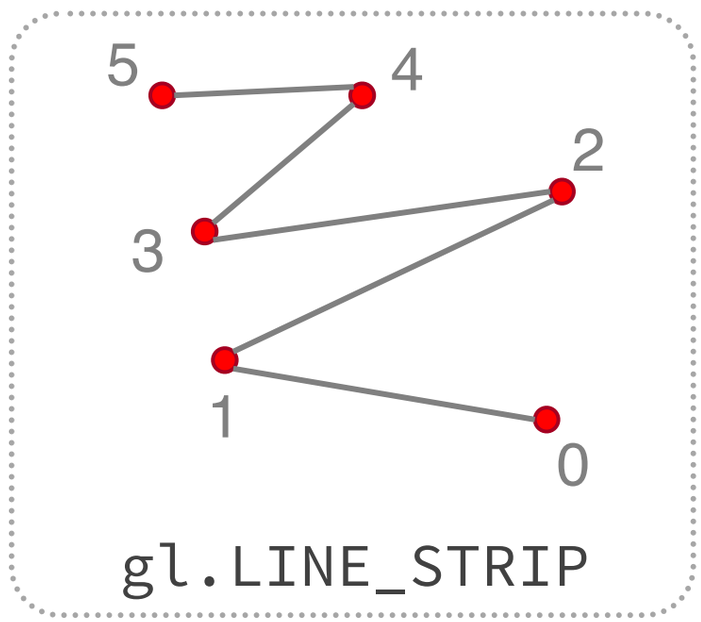
Rendern



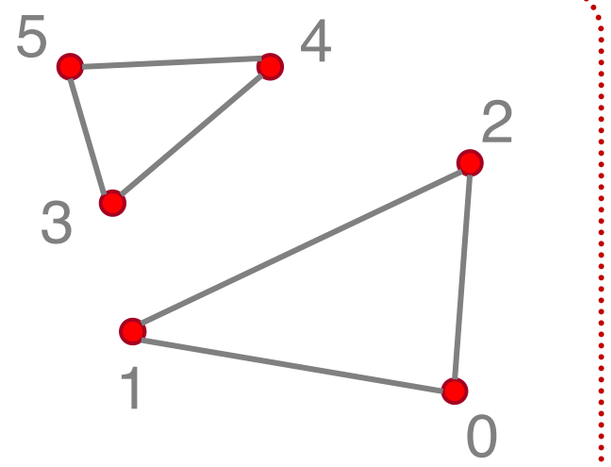
gl.POINTS



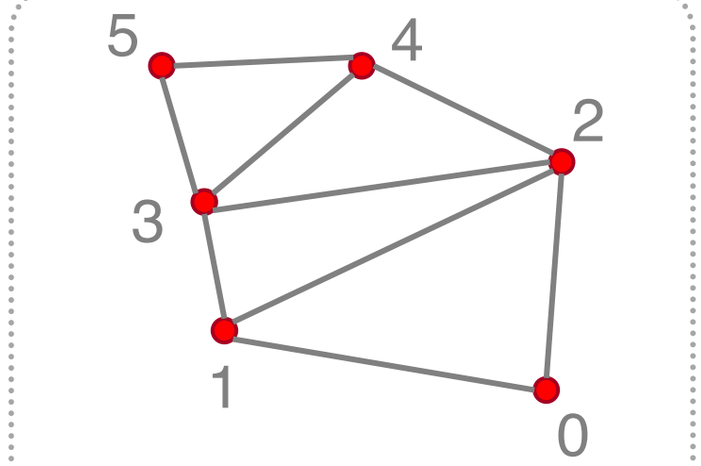
gl.LINES



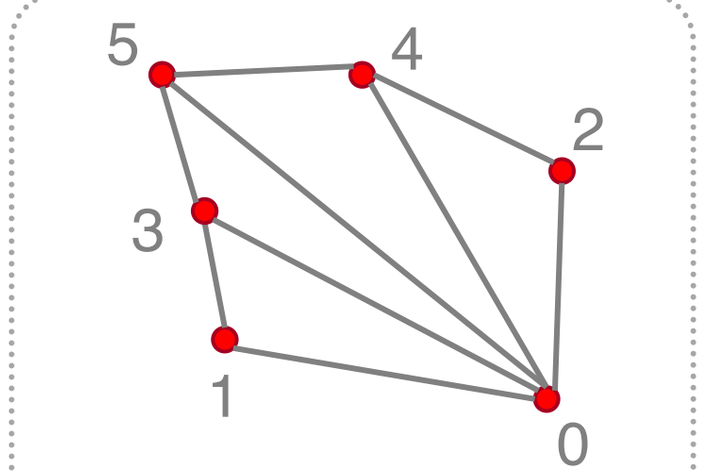
gl.LINE_STRIP



gl.TRIANGLES

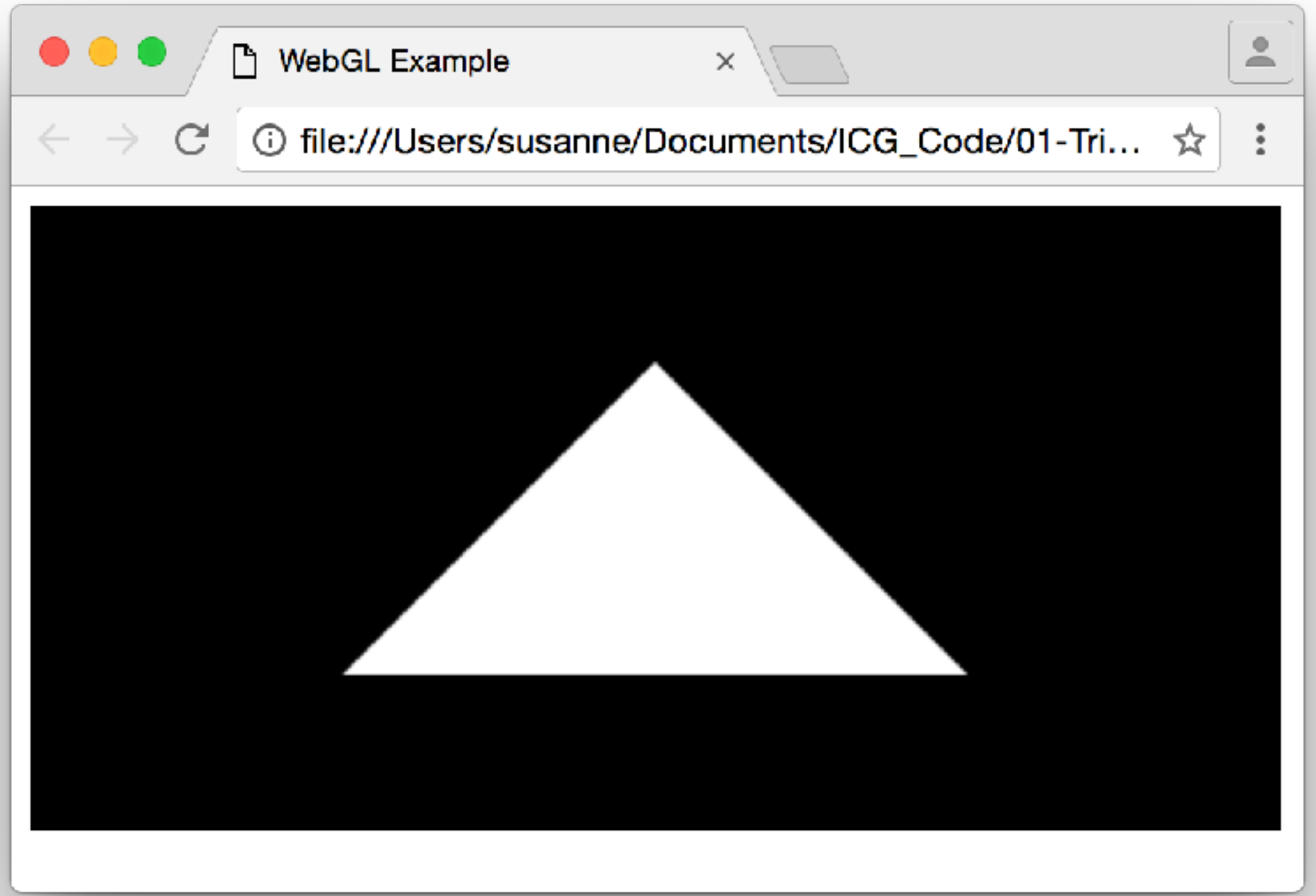


gl.TRIANGLE_STRIP

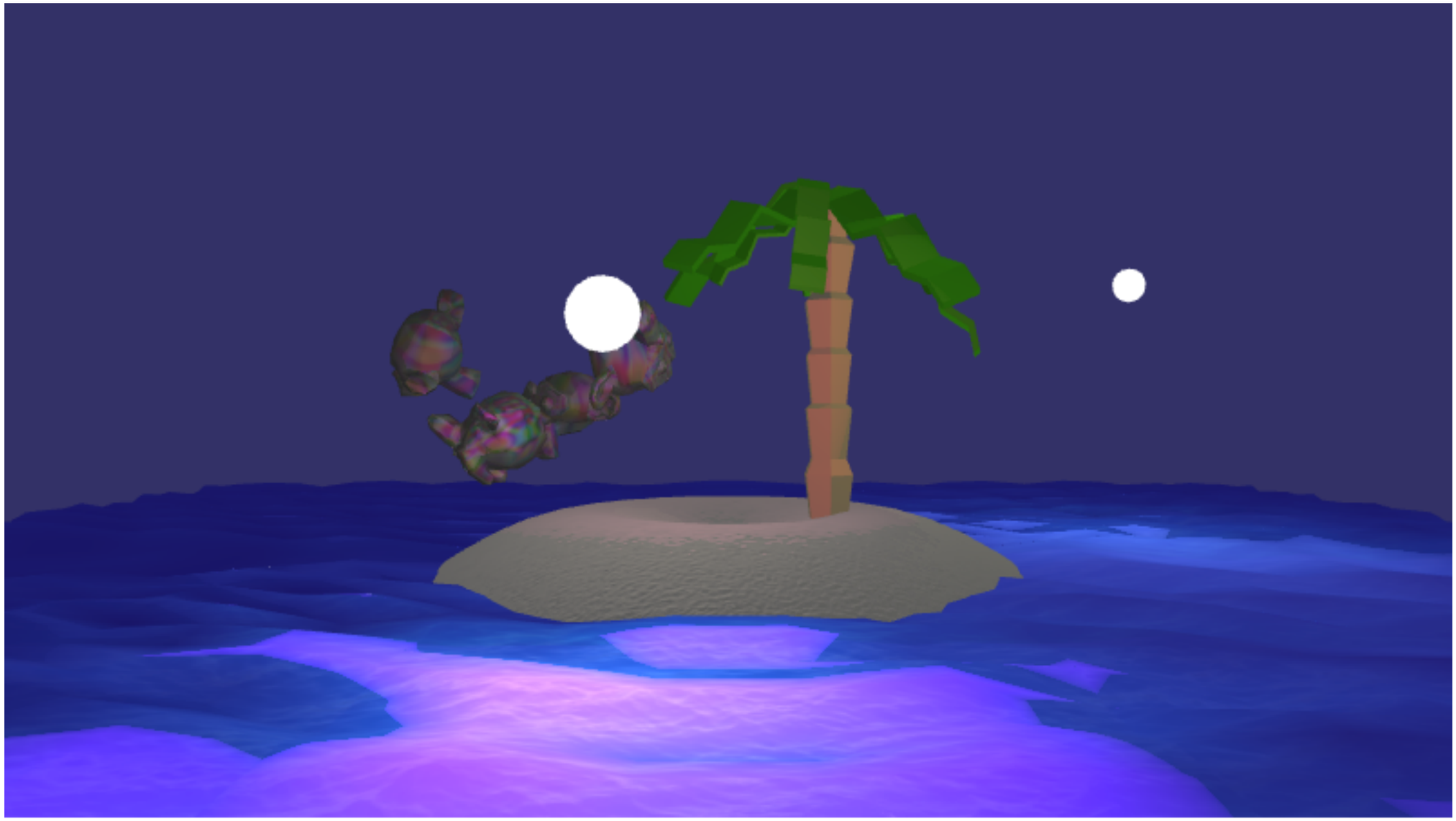


gl.TRIANGLE_FAN

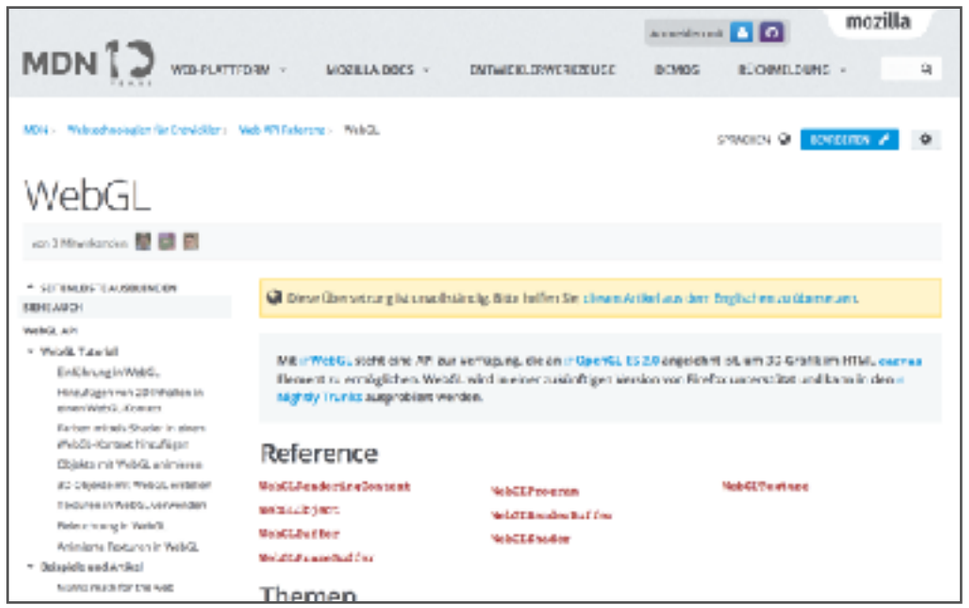
Ergebnis



Ergebnis in 12 Wochen



WebGL
Tutorials



- https://www.youtube.com/watch?v=tgVLb6fOVVc
- http://developer.mozilla.org/de/docs/Web/API/WebGL_API