



BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Computergrafik II

Einführung in WebGL und Javascript

Bachelor Medieninformatik
Wintersemester 2011

Prof. Dr.-Ing. Hartmut Schirmacher
<http://schirmacher.beuth-hochschule.de>
hschirmacher@beuth-hochschule.de



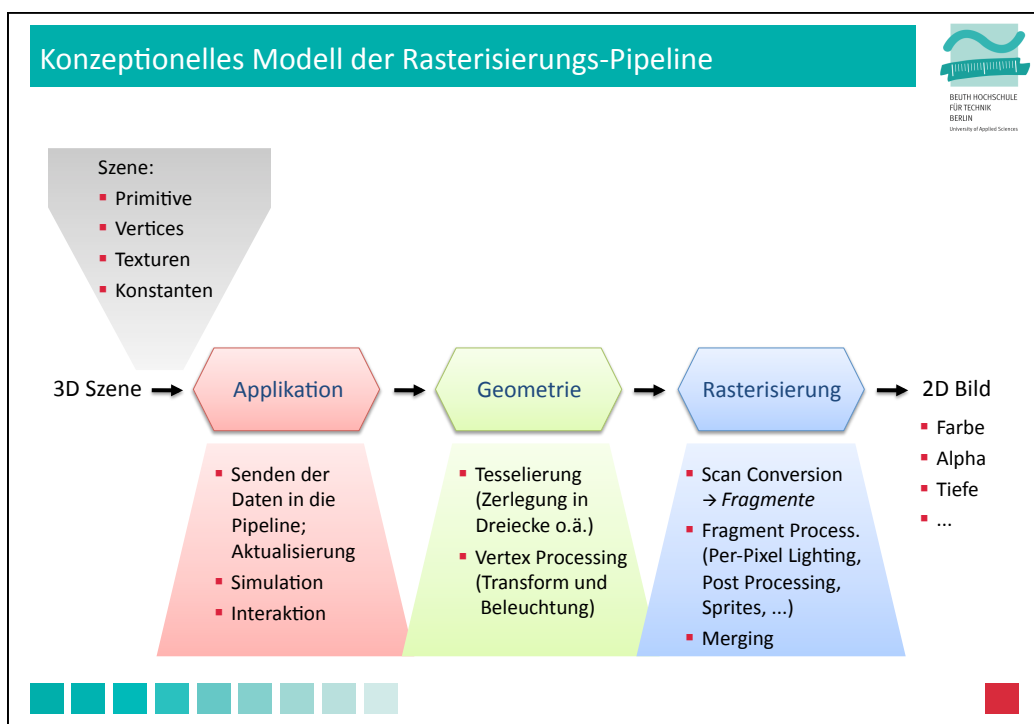
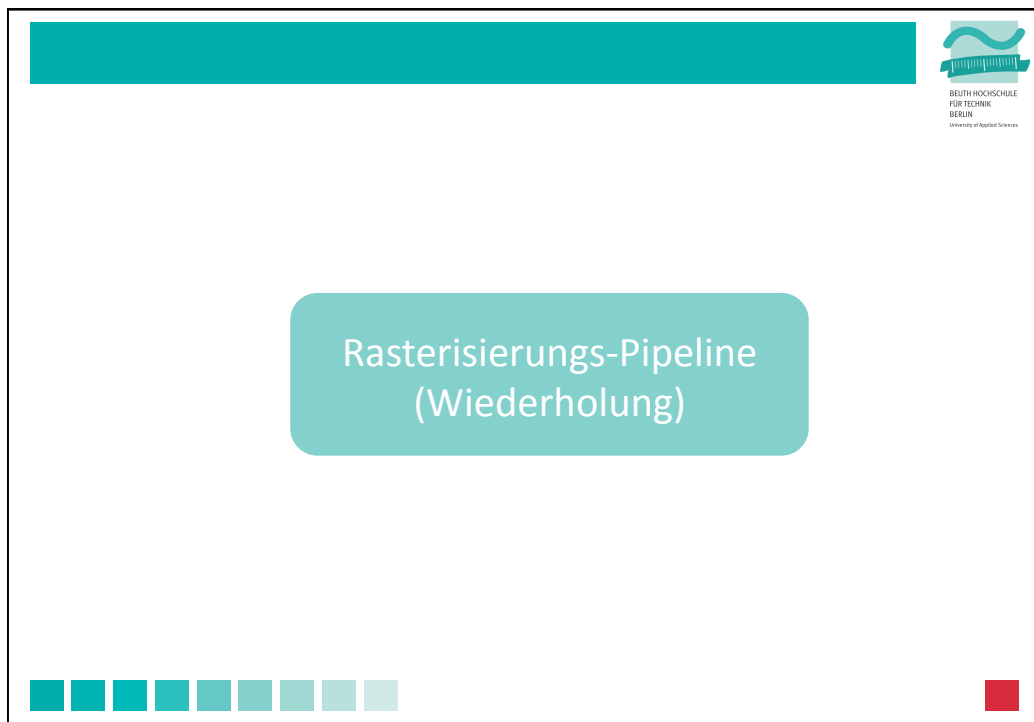
Gliederung

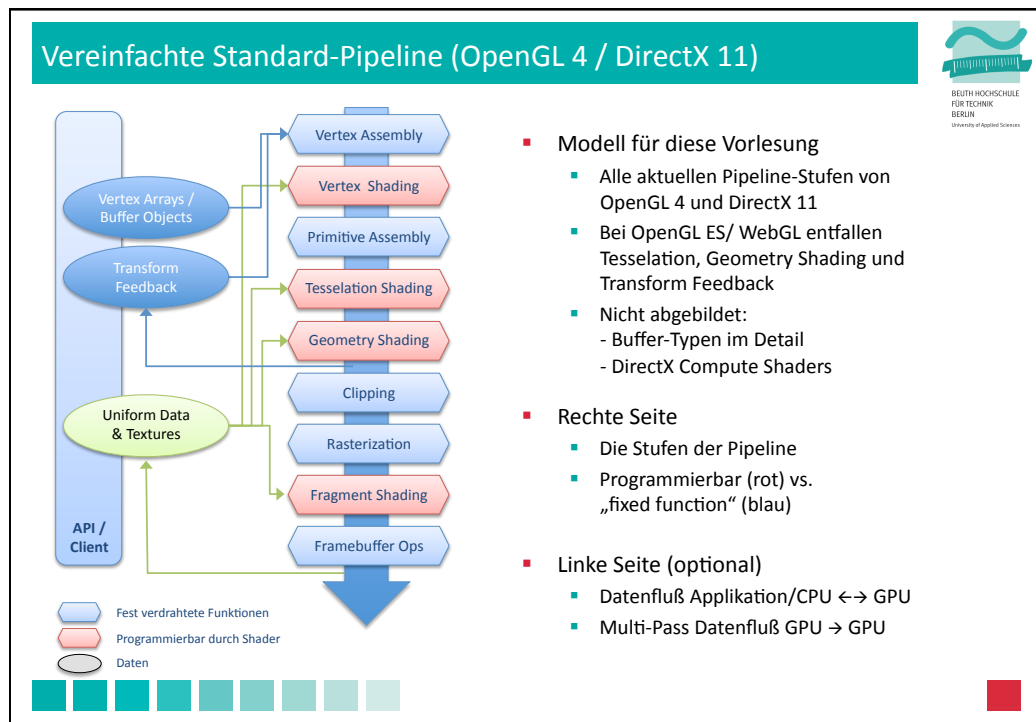


BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

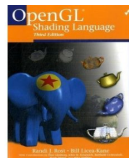
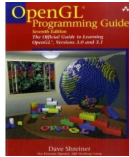
- Zielsystem, Pipeline und Shading Language
 - Wiederholung: Rasterisierungs-Pipeline
 - Zusammenspiel OpenGL und Shading Language
 - Einbettung von OpenGL in das Zielsystem
- Einführung in WebGL und Javascript
 - Ein einfaches Beispiel
 - Objektorientierung in Javascript
 - Beispiel mit Shadern, Szene, Shapes & Co







OpenGL – GLSL – OpenGL ES – WebGL



- OpenGL – "Open Graphics Library"
 - 1992 OpenGL 1.0 basierend auf IRIS GL von Silicon Graphics
 - Definiert / implementiert die verbreitete Grafik-Pipeline
 - Aktuell OpenGL 4.2 (2011)
- Open GL Shading Language (GLSL, GLSLang)
 - Hochsprache zur Spezifikation von Shader-Code
 - Seit 2004 Teil von OpenGL (2.0)
 - Aktuell GLSL 4.2 (an OpenGL-Versionsnummer angepaßt)
- OpenGL ES (Embedded Systems)
 - Speziell für mobile Geräte wie Smartphones entworfen
 - "Schlanke" Untermenge von OpenGL, kein OpenGL 1 erlaubt
 - Aktuell OpenGL ES 2.0 (2007), basiert auf OpenGL 2 und GLSL 1.2
- WebGL (HTML 5 / Browser)
 - Javascript-Schnittstelle zur Verwendung von OpenGL mit HTML 5 im Browser
 - Gleicher Sprachumfang wie OpenGL ES 2.0
 - Aktuell WebGL 1.0 (2011)



OpenGL 1



OpenGL 1

```

glClearColor(1, 1, 1, 1); // white
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
glOrtho(0, 100, 0, 100, -1, 1);
glShadeModel(GL_SMOOTH | GL_FLAT);
glLightfv(GL_LIGHT0, GL_POSITION, ...);
glEnable(GL_LIGHTING);
glMaterialfv(GL_FRONT, GL_SPECULAR, ...);
...
glBegin(GL_TRIANGLE_STRIP);
glColor3f(0, 0.5, 0); // dark green
glVertex2i(11, 31);
glVertex2i(37, 71);
glColor3f(0.5, 0, 0); // dark red
glVertex2i(91, 38);
glVertex2i(65, 71);
glEnd();
glFlush();

```

- **Starre Rendering-Modelle**
 - Entwickler kann aus vorgegebenen Beleuchtungs-Modellen wählen und diese *konfigurieren*
 - OpenGL setzt dann alle Berechnung intern um
- **Eingeschränkte Attribute**
 - Vor dem malen eines Vertex müssen Position und Farbe global festgelegt werden (ggf. noch Texturen und Texturkoordinaten)
 - Diese werden dann im Rahmen des konfigurierten Beleuchtungsmodells verarbeitet



OpenGL 1 vs. OpenGL 3 – Programmierbare Pipeline



OpenGL 1

```
glClearColor(1, 1, 1, 1); // white
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
glOrtho(0, 100, 0, 100, -1, 1);
glShadeModel(GL_SMOOTH | GL_FLAT);
glLightfv(GL_LIGHT0, GL_POSITION, ...);
glEnable(GL_LIGHTING);
glMaterialfv(GL_FRONT, GL_SPECULAR, ...);
...
glBegin(GL_TRIANGLE_STRIP);
glColor3f(0, 0.5, 0); // dark green
glVertex2i(11, 31);
glVertex2i(37, 71);
glColor3f(0.5, 0, 0); // dark red
glVertex2i(91, 38);
glVertex2i(65, 71);
glEnd();
glFlush();
```

OpenGL 3

```
glClearColor(1, 1, 1, 1); // white
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
glOrtho(0, 100, 0, 100, -1, 1);
glUseProgram(myProgram);

glVertexAttribPointer(..., vertices);
glVertexAttribPointer(..., colors);

glDrawArrays(GL_TRIANGLE_STRIP);
glFlush();
```



Wie kommt der Shader-Code in die Applikation / GPU?



OpenGL-Applikation (1)

```
// Install Shading Programs
void InstallPrograms () {

    p1 = glCreateProgram();
    glAttachShader(p1,vs);
    glAttachShader(p1,fs);

    p2 = glCreateProgram();
    ...
}

// Rendering
void renderScene() {
    ...
}
```

OpenGL – Compile Shader

```
vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs,...);
glCompileShader(vs);
```

OpenGL – Compile Shader

```
fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs,...);
glCompileShader(fs);
```

GLSL - Vertex Shader

```
// per-vertex attribute
in vec4 vertexPos;

// transform vertex coords into
// camera coordinate system
void main ()
{
    gl_Position = ... * vertexPos;
}
```

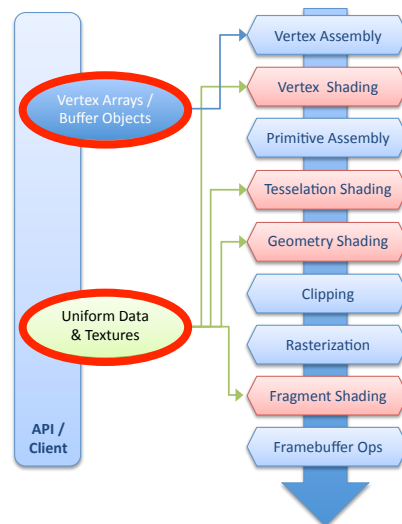
GLSL - Fragment Shader

```
// global "uniform" variables
uniform vec4 u_color1;
uniform vec4 u_color2;

// shade in two colors, depending
// on the pixel's x coordinate
void main()
{
    if (...)
    {
        gl_FragColor = u_color1;
    } else {
        gl_FragColor = u_color2;
    }
}
```



Wie kommen die Daten von der Applikation in die Shader (1)?



Wie kommen die Daten von der Applikation in die Shader (2)?

OpenGL-Applikation (2)

```

GLfloat vertices[] = ...;

void renderObject() {

    // specify vertex attributes (e.g. position)
    glVertexAttribPointer(i,...,vertices);
    glEnableVertexAttribArray(i);
    glBindAttribLocation(p1, i, "a_vertexPos");

    // link and activate program
    glLinkProgram(p1);
    glUseProgram(p1);

    // set global GLSL variables
    u1 = glGetUniformLocation(p1, "u_color1");
    glUniform4f(u1, 1.0,0.0,0.0,1.0);
    u2 = glGetUniformLocation(p1, "u_color2");
    glUniform4f(u2, 0.0,1.0,0.0,1.0);

    // issue draw commands
    glDrawArrays(GL_QUADS, ...);
}
  
```

Vertex Shader (GLSL)

```

// per-vertex attribute
attribute vec4 a_vertexPos;

// transform vertex coords into
// camera coordinate system
void main ()
{
    gl_Position = ... * a_vertexPos;
}
  
```

Fragment Shader (GLSL)

```



// global variables in shader
uniform vec4 u_color1;
uniform vec4 u_color2;

// shade in two colors, depending
// on the pixel's X coordinate
void main()
{
    if (...) {
        gl_FragColor = u_color1;
    } else {
        gl_FragColor = u_color2;
    }
}
  
```



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences



Einbettung von OpenGL in das Zielsystem



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Einbettung von OpenGL (1)

- Was ist OpenGL?
 - Ein offener Standard zur 3D-Grafik-Programmierung
 - Eine API zum Zugriff auf Funktionen der Grafikhart-Hardware
- Was ist OpenGL *nicht*?
 - Eine API zum Erzeugen und Verwalten von Fenstern und Bildschirmen
 - Eine API zur Erzeugung graphischer Benutzerschnittstellen (UIs)
- OpenGL muß in das Zielsystem *eingebettet* werden



Einbettung von OpenGL (2)



- Einbettung von OpenGL
 - unterschiedlichste Möglichkeiten, abhängig von
 - dem verwendeten Framework
 - der gewünschten Programmiersprache
 - Das Fenster-System verwaltet die *Displays* und muß die sog. *Rendering Surface* bereitstellen, also z.B. ein Desktop-Fenster oder ein Bereich auf einem Smartphone-Display.
 - Das Fenster-System kümmert sich auch um die *Synchronisation* der Ausgabe



Einbettung von OpenGL: C++ und Qt



C++ und Qt mit OpenGL (Desktop Cross-Plattform)

```
void init_gl() {
    QGLFormat format(QGL::DoubleBuffer | QGL::Rgb | QGL::DepthBuffer);
    QGLContext *context = new QGLContext(format);
    context->create();
    Viewer = new QGLViewer(context);
    ...
}

void Viewer::paint_gl() {
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

- Qt und QGL
 - Qt ist ein Cross-Plattform-Framework von Trolltech (z.Zt. Nokia) für C++ (und Java)
 - QGL ist die Anbindung von OpenGL an Qt
- QGLContext
 - **Kontext-Objekt** wird vom System angefordert, unter Angabe der gewünschten Daten im Framebuffer
 - Kapselt den Zustand des Grafiksystems in einem bestimmten Kontext; für mehrere Fenster können z.B. mehrere unabhängige Kontexte angefordert werden
- QGLViewer
 - Erlaubt es, die Methode **paint_gl()** zu überladen, um dort die **OpenGL-Befehle** zum malen der Szene abzusetzen
 - Nach dem Abarbeiten von **paint_gl()** führt QGL automatisch die Swap-Operation von Front- und Back-Buffer durch



Einbettung von OpenGL: Java und LWJGL



Java und LWJGL (Desktop Cross-Plattform)

```
import org.lwjgl.opengl.Display;

Display.create(new PixelFormat());
while (!Display.isCloseRequested()) {
    (...)
    // set background color / clear image and Z Buffer
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);

    // draw whatever you want...
    (...)

    // perform the buffer swap etc.
    Display.update();
}
```

- LWJGL (Lightweight Java Game Library): u.a. OpenGL-Anbindung für Java
 - Das **Display** ist die Verbindung zum Fenstersystem des Betriebssystems
 - Nachdem ein Display bzw. ein Rendering-Kontext erzeugt wurde, können **OpenGL-Befehle** abgesetzt werden
 - Am Ende wird explizit **Display.update()** aufgerufen, dies löst u.a. den Buffer Swap aus



Einbettung von OpenGL: Beispiel WebGL



HTML 5

```
<html>
<head>
<title>WebGL in HTML5</title>
<script type="text/javascript">
window.onload = function () {
    canvas = document.getElementById("myCanvas");
    gl = canvas.getContext("webgl");
    program = gl.createProgram();
    ...
    gl.viewport(0,0,canvas.width,canvas.height);
    ...
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    ...
}
</script>
</head>
<body>
<canvas id="myCanvas" width="500" height="500"></canvas>
<br>
Dies ist eine <i>ganz normale</i> HTML-Seite.
</body>
</html>
```

- Einbettung von WebGL
 - nicht direkt in ein natives Fenster
 - sondern in ein **<canvas>**-Element einer HTML-5-Seite



Einbettung von OpenGL: Beispiel WebGL



HTML 5 mit WebGL

```

<html>
<head>
<title>WebGL in HTML5</title>
<script type="text/javascript">
window.onload = function () {
  canvas = document.getElementById("myCanvas");
  gl = canvas.getContext("webgl");
  program = gl.createProgram();
  ...
  gl.viewport(0,0,canvas.width,canvas.height);
  ...
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  ...
}
</script>
</head>
<body>
<canvas id="myCanvas" width="500" height="500"></canvas>
<br>
Dies ist eine <i>ganz normale</i> HTML-Seite.
</body>
</html>

```


- Einbettung von WebGL
 - nicht direkt in ein natives Fenster
 - sondern in ein **<canvas>**-Element einer HTML-5-Seite
- Javascript-API
 - Startpunkt ist typischerweise ein Event-Handler wie z.B. `window.onload()`
- Zentraler Punkt ist das Erzeugen eines WebGL-Kontext-Objekts
 - **Kontext-Objekt** kapselt alle WebGL-Befehle
 - **OpenGL-Funktionen** aufrufen mit `gl.createProgram()`, `gl.viewport()`, `gl.clear()`, ... etc.
- Eine Seite kann *mehrere* WebGL-canvas-Elemente enthalten
 - Jede arbeitet dann unabhängig mit einem eigenen WebGL-Kontext-Objekt



Ein erstes Beispiel
in Javascript / WebGL



HTML Canvas-Element und WebGL Kontext



JavaScript Event Handler:
Wird beim Laden der Seite aufgerufen.

Referenzierte Javascript-Datei:
Lagert verwendete Funktionen aus.

1. Erzeuge WebGL-Rendering-Kontext:
Objekt, welches alle Befehle und den Zustand von WebGL für den ausgewählten Canvas kapselt.


HTML 5 Canvas-Element:
Definiert einen Grafik-Bereich (*rendering surface*) innerhalb der HTML-Seite.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>WebGL: Getting Started</title>
5 <script type="text/javascript" src="01-getting-started.js"></script>
6 <script>
7   window.onload = function () {
8     gl = initWebGL("myCanvas");
9     vs = "attribute vec4 vertexPosition; void main() { gl_Position = vertexPosition; }";
10    fs = "void main() { gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); } ";
11    program = makeProgram(gl, vs, fs);
12    buffer = makeBuffer(gl, program);
13    drawScene(gl, program, buffer);
14  };
15 </script>
16 </head>
17 <body>
18   <canvas id="myCanvas" width="300" height="300"></canvas>
19 </body>
20 </html>

```

Komponenten eines einfachen WebGL-Codes



Shader-Sourcecode:
Der GLSL-Code für zwei Minimal-Shader ist hier direkt als Text-String in Javascript angegeben.

2. Erzeuge WebGL program
Erzeugt aus dem Sourcecode der beiden Shader durch Kompilation und Linking ein *program*, welches es erlaubt, die Shader auf der GPU auszuführen.

3. Erzeuge buffer object mit Vertexdaten
Objekt lebt potenziell im Speicher der GPU und enthält die Daten aller gewünschten Vertex-Attribute (zunächst einmal nur die jeweilige 3D-Position)

4. Male die Szene
Initialisiere mit Hintergrundfarbe, aktiviere *program* und *buffer object*, gib Befehl zum Zeichnen der geometrischen Primitiven (z.B. TRIANGLES).

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>WebGL: Getting Started</title>
5 <script type="text/javascript" src="01-getting-started.js"></script>
6 <script>
7   window.onload = function () {
8     gl = initWebGL("myCanvas");
9     vs = "attribute vec4 vertexPosition; void main() { gl_Position = vertexPosition; }";
10    fs = "void main() { gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); } ";
11    program = makeProgram(gl, vs, fs);
12    buffer = makeBuffer(gl, program);
13    drawScene(gl, program, buffer);
14  };
15 </script>
16 </head>
17 <body>
18   <canvas id="myCanvas" width="300" height="300"></canvas>
19 </body>
20 </html>

```

WebGL-Initialisierung / Erzeugung des GL-Kontext-Objekts



```
initWebGL = function(canvasName) {
    var canvas = window.document.getElementById(canvasName);
    var gl = canvas.getContext("experimental-webgl");
    // error checks...
    return gl;
}
```

- Finde das DOM-Element zum gesuchten Canvas-Knoten
- Fordere einen WebGL-Kontext für dieses Canvas-Element an
 - Man könnte auch mit anderen Mitteln hineinmalen, aber in unserem Beispiel soll es mit WebGL geschehen
- Überprüfe, ob das funktioniert hat (`gl != null`)
 - Sonst unterstützt der Browser kein WebGL
 - Ausführlicheres Code-Beispiel in der Übung



Shader kompilieren und linken



```
makeProgram = function(gl, vs_source, fs_source) {
    var program = gl.createProgram();

    // compile and attach vertex shader
    vshader = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vshader, vs_source);
    gl.compileShader(vshader); // error checks...
    gl.attachShader(program, vshader);

    // compile and attach fragment shader
    fshader = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fshader, fs_source);
    gl.compileShader(fshader); // error checks...
    gl.attachShader(program, fshader);

    gl.linkProgram(program);
    return program;
}
```

- Kompilation der Shader-Sourcen
 - Jeder Shader hat maximal eine `main()`-Funktion, aber beliebige sonstige Funktionen
- Zuweisen der Shader zu einem Programm
 - Ein Shader pro Shader-Typ
- Es ist möglich, mehrere Programme zu erzeugen und mittels `gl.useProgram()` zwischen ihnen zu wechseln
- Kompilations-Fehler können abgefragt und dem Entwickler im Klartext ausgegeben werden (siehe Übungs-Code)



Vertex Buffer Object erzeugen

```
makeBuffer = function(gl, program) {
    var buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
    var vVertices = new Float32Array( [0.0,1.0, -1,-1,0, 1,-1,0 ] );
    gl.bufferData(gl.ARRAY_BUFFER, vVertices, gl.STATIC_DRAW);
    return buffer;
}
```

- Ein Vertex Buffer Object (VBO) ist ein konzeptionell ein Objekt, welches im Grafik-Speicher verweilt
- Für größere Effizienz wird das Objekt einmal in den Grafikspeicher geladen und dann möglichst oft lesend wiederverwendet
 - Hinweise für die Zugriffs-Optimierung: STATIC_DRAW, DYNAMIC_DRAW, ... geben an, ob die Daten des Objekt oft aktualisiert werden müssen, bzw. ob dieselben Daten oft wiederverwendet werden
 - Es existieren separate Funktionen, um Teile der Daten eines VBOs effizient zu aktualisieren



Szene mittels vorher definiertem VBO malen

```
drawScene = function(gl, program, buffer) {
    gl.clearColor(0,0,0,1);
    gl.enable(gl.DEPTH_TEST);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    gl.useProgram(program);

    var location = gl.getAttribLocation(program, "vertexPosition");
    if(location < 0) {
        window.console.log("vertexPosition not used!");
    } else {
        gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
        gl.vertexAttribPointer(location, 4, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(location);
        gl.drawArrays(gl.TRIANGLES, 0, 3);
    }
}
```

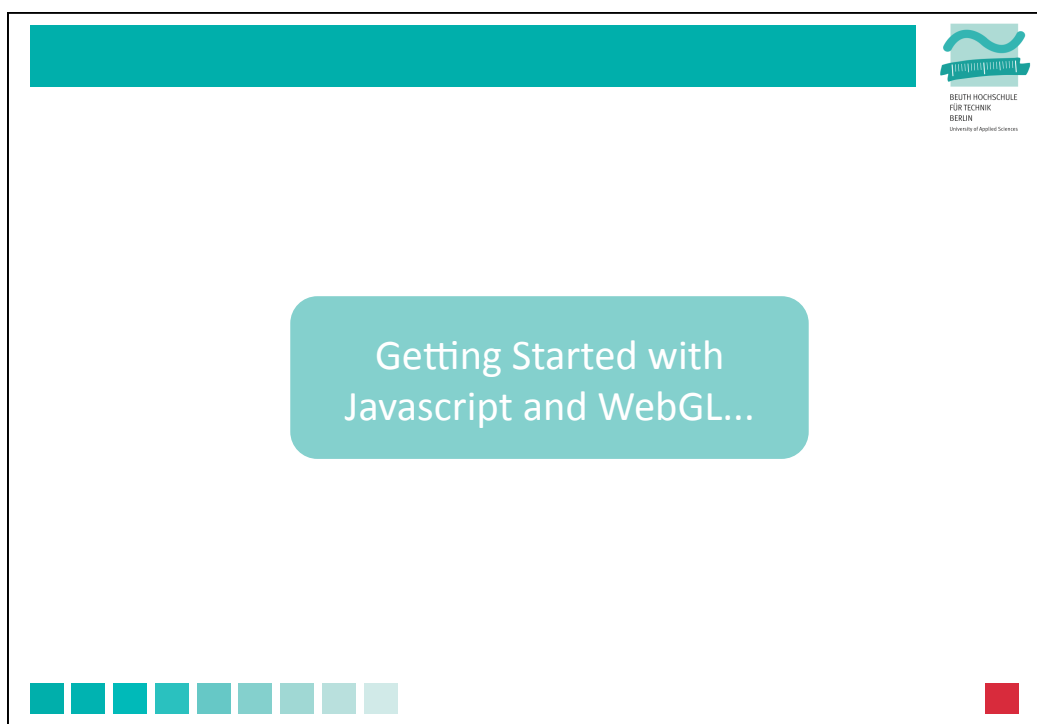
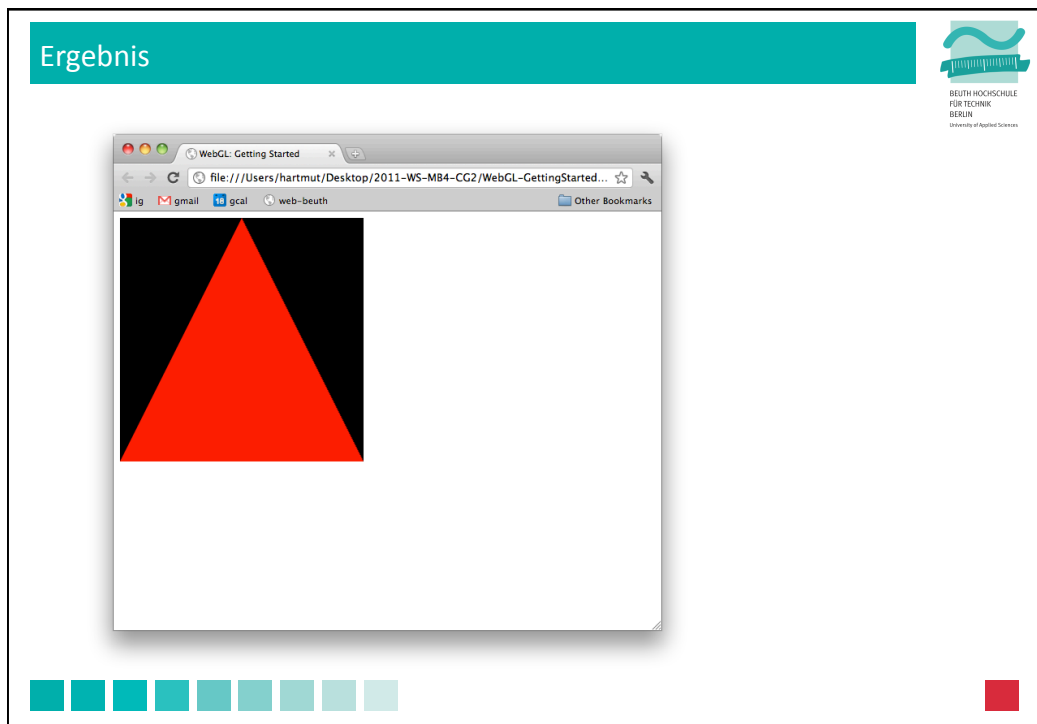
<Vertex Shader Sourcecode>

```
attribute vec4 vertexPosition;

void main() {
    gl_Position = vertexPosition;
}
```

- Hintergrundfarbe festlegen / Bild initialisieren
- Programm mit den kompilierten Shadern aktivieren
- Herausfinden, welchen „Slot“ das Attribut `vertexPosition` im Vertex-Shader belegt
 - Liefert eine ID des Attributs in bezug auf das aktuelle Programm zurück
- Das VBO „binden“ und auf das erste zu verwendende Datenelement zeigen
- Primitiven (hier: Dreiecke) erzeugen und in die Pipeline schicken!





Javascript vs. Java

Quelle: Core Javascript Guide on mozilla.org



- JavaScript and Java are similar in some ways but fundamentally different in others. The JavaScript language resembles Java but **does not have Java's static typing and strong type checking**. JavaScript **supports most Java expression syntax and basic control-flow constructs**.
- JavaScript is **a very free-form language compared to Java**. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to implement interfaces. Variables, parameters, and function return types are not explicitly typed.
- In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a **small number of data types representing numeric, Boolean, and string values**. JavaScript has a **prototype-based object model** instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports **functions without any special declarative requirements**. Functions can be **properties of objects, executing as loosely typed methods**.



Javascript vs. ECMAScript

Wir verwenden Javascript und ECMAScript synonym.

- 1995: Netscape veröffentlicht die *browserseitige Scriptsprache* LiveScript für den Netscape Navigator
- Netscape + Sun kooperieren zur Anbindung von Livescript an Java.
- Sun benennt Livescript in Javascript um
- 1998: ECMA-262 First Edition
- 2002: Javascript 1.5, entspricht ECMA-262 Version 3
- ...



Quelle: wikipedia.de

Von <http://www.ecma-international.org/>



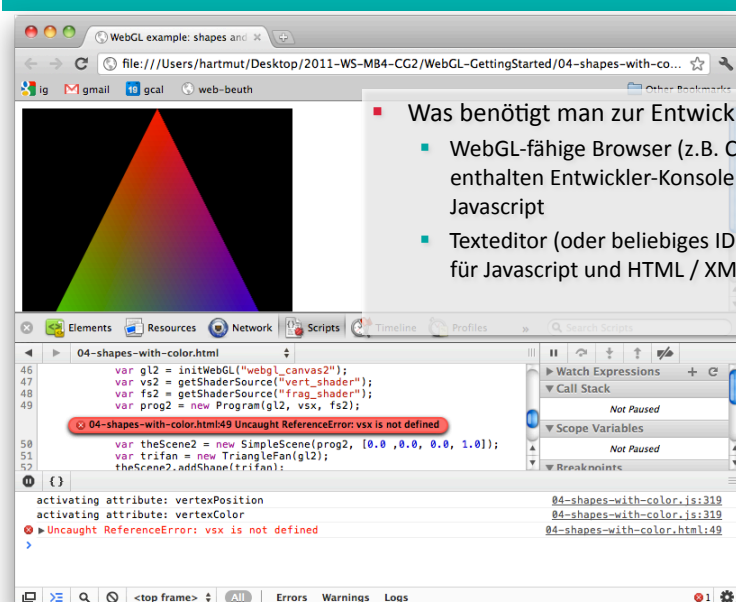
Spezifikationen / Dokumentation / Tutorial



- Spezifikationen und Referenzen im Web
 - HTML-Grundwissen und einfache Javascript-Referenz: <http://selfhtml.org>
 - ECMAScript: <http://www.ecma-international.org/>
 - WebGL: <http://www.khronos.org/webgl/>
Ist im wesentlichen eine Liste der spezifizierten ECMAScript-Funktionen mit Verweisen in die OpenGL-ES-Spezifikation
 - OpenGL ES 2.0: <http://www.khronos.org/opengles/>
für die ausführliche Erklärung der OpenGL-Befehle
 - OpenGL ES Shading Language Version 1.0 (im wesentlichen GLSL Version 1.20):
http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf
- Tutorials
 - Gute Einführung basierend auf Basis der NeHe-Tutorials:
<http://learningwebgl.com/>
 - Vorsicht bei vielen Web-Tutorials: oftmals extrem schlechter Programmierstil



Entwicklungsumgebung



- Was benötigt man zur Entwicklung mit WebGL?
 - WebGL-fähige Browser (z.B. Chrome, Firefox) enthalten Entwickler-Konsole und Debugger für Javascript
 - Texteditor (oder beliebiges IDE) mit Unterstützung für Javascript und HTML / XML



Javascript: Zuweisungen und Typen



```
// Globale Variablen
x1 = „Hello World“;
x2 = 34.4;

// Funktion mit lokalen Variablen
function myFunc(x,y) {
  var tmp = x*y;
  return tmp*tmp;
}

// Alternative Schreibweise
myFunc = function(x,y) { (...) }
```

■ Zuweisungen und Typen

- Nur wenige Grundtypen: Boolean, Zahl, String, Objekt.
- Dynamisches Typsystem: Typen müssen nicht deklariert werden, sondern werden zur Laufzeit ausgewertet / geprüft
- Eine neue Zuweisung kann jederzeit eine alte überschreiben

■ Funktionen

- Benannte Parameter, ebenfalls ohne Typdeklaration
- Lokale Variablen mit „var“
- Globale Variablen: ohne „var“



Javascript: Arrays und Kontrollstrukturen



```
// Ein Array erzeugen
var x = new Array();

// einiges hinzufügen
x.push("Halli");
x.push("Hallo");

// Schleife über alle Elemente
for(var i in x) {
  window.console.log(i + ": " + x[i]);
}

// Alternative Formulierung
for(var i=0; i<x.length; i++) { (...) }

// Array löschen
delete x;

// Bedingte Anweisungen
if(z != 3) {
  alert("Z ist nicht drei!");
} else { (...) }
```

■ Objekte

- Werden durch Aufruf einer Konstruktor-Funktion mittels new() erzeugt
- Können mit delete gelöscht werden
- Null kann als Platzhalter für „kein Objekt“ verwendet werden

■ Array-Objekt

- Mit new() erzeugen
- Zugriffsfunktionen siehe Sprachreferenz

■ Schleifen und Kontrollstrukturen

- for-Schleife in zwei alternativen Formen wie in Java
- while und do-while wie in Java
- break, continue
- if-else



Javascript: Helferlein

```
// Öffne ein Nachrichtenfenster
alert(Zahl + " ist keine Primzahl, weil
teilbar durch " + i);

// Schreibe eine Zeile in die Konsole
Window.console.log("Auf der Konsole")
```

- `alert()`
 - Browser öffnet ein Dialogfenster mit einer Nachricht
- `window.console.log()`
 - Schreibt eine Zeile in die Javascript-Konsole des Browsers
 - Normalerweise nur für Entwickler sichtbar
 - (vgl. `system.out.println()`)



Codestrukturierung

- Lagere längeren Javascript-Code in separate .js Dateien aus
 - Damit der Editor Syntax-Highlighting anwenden kann
- Lagere GLSL-Shader-Code aus
 - In separate HTML-Knoten oder
 - In separate Dateien
- Designe wiederverwendbare Javascript-Komponenten
 - Vermeide globale Funktionen und Variablen
 - Es kann mehrere WebGL-Kontexte geben (z.B. zwei Canvas-Objekte!)
 - Verwende (soweit für andere nachzuvollziehen) objektorientiertes Javascript
- Beispiele im Web – leider oft keine guten Beispiele
 - Große Versuchung, alles in eine große Javascript-Funktion zu schreiben
 - Seiteneffekte (globale Variablen) als Designprinzip



Einfaches Beispiel

Einfaches WebGL-Programm – Shader als JS-Strings

```

<!DOCTYPE html>
<html>
<head>
  <title>WebGL example: static scene</title>
  <script type="text/javascript"
    src="02-static-scene.js">
  </script>
  <script>
    window.onload = function () {
      gl = initWebGL("myCanvas");
      vs = "void main() { gl_Position = (...) } ";
      fs = "void main() { gl_FragColor = (...) } ";
      program = makeProgram(gl, vs, fs);
      buffer = makeBuffer(gl, program);
      drawScene(gl, program, buffer);
    };
  </script>
</head>
<body>
  <canvas id="myCanvas" width="500" height="500"></
  canvas>
</body>
</html>

```

- Shader als Javascript-String
 - Nur für sehr kurze Shader geeignet



Auslagern des Shader-Codes in <script>-Knoten

Auslagern der Shader in <script>-Knoten

```

(...)
<!-- vertex shader source code -->
<script id="vert_shader" type="x-shader/x-vertex">
  attribute vec4 vertexPosition;
  void main() {
    gl_Position = vertexPosition;
  }
</script>
<!-- fragment shader source code -->
<script id="frag_shader" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0);
  }
</script>
<script>
  window.onload = function () {
    gl = initWebGL("myCanvas");
    vs = getShaderSource("vert_shader");
    fs = getShaderSource("frag_shader");
    program = makeProgram(gl, vs, fs);
    buffer = makeBuffer(gl, program);
    drawScene(gl, program, buffer);
  };
</script>
(...)
```

- Shader in <script> ... </script>
 - Typ muß angegeben werden, damit der GLSL-Code nicht als Javascript interpretiert wird
 - Konvention für den Typ: „x-shader/x-vertex“ etc
 - Deutlich lesbarer für mittel-lange Shader
 - Übersichtlich in einer HTML-Datei
 - Zumeist kein korrektes Syntax-Highlighting
- Funktion zum Auslesen der Shader
 - Findet die <script>-Knoten anhand ihrer ID und liefert den Inhalt als String zurück



Funktion zum Auslesen des Shader-Codes aus dem DOM

Inhalt eines <script>-Knotens im DOM auslesen

```
getShaderSource = function(id) {
    var shaderScript = document.getElementById(id);
    if (!shaderScript) {
        return null;
    }

    var result = "";
    var k = shaderScript.firstChild;
    while (k) {
        if (k.nodeType == 3)
            result += k.textContent;
        k = k.nextSibling;
    }
    return result;
}
```

Auslesen des Shader-Codes

- Findet den entsprechenden Script-Knoten anhand der ID (getElementById()).
- Sammelt alle Textknoten unterhalb des <script>-Knotens ein
- Liefert die Gesamtheit des Textes als String zurück
- Javascript hat ein ausführliches Modell der Struktur eines HTML-Dokuments (DOM)

| Nummer | Knotentyp |
|--------|-----------------------------------|
| 1 | Elementknoten |
| 2 | Attributknoten |
| 3 | Textknoten |
| 4 | Knoten für CDATA-Bereich |
| 5 | Knoten für Entity-Referenz |
| 6 | Knoten für Entity |
| 7 | Knoten für Verarbeitungsanweisung |
| 8 | Knoten für Kommentar |
| 9 | Dokument-Knoten |
| 10 | Dokumenttyp-Knoten |
| 11 | Dokumentfragment-Knoten |
| 12 | Knoten für Notation |

Quelle: selfhtml.org



Verbesserung der Codestructur mittels OO

Hauptprogramm mit Programm-, Szene- und Shape-Objekten

```
window.onload = function () {
    gl = initWebGL("myCanvas1");
    var vs = getShaderSource("vert_shader");
    var fs = getShaderSource("frag_shader");
    prog = new Program(gl, vs, fs);
    theScene = new Scene(prog, [0.0, 0.0, 0.0, 1.0]);
    theScene.addShape(new Triangle(gl));
    theScene.draw();
};
```

Motivation

- Strukturierung des Codes ähnlich zu den aus dem Raytracer bekannten Komponenten
- Austausch / Wiederverwendung einzelner Komponenten je nach Anwendung
- Neu: der WebGL-Kontext und das Program

Objekt: Program

- Enthält compilierte und gelinkte Shader

Objekt: Scene

- Enthält eine Menge von geometrischen Objekten (Shapes)
- Kann sich selbst malen

Objekt: Shape / Triangle

- (→ später)



Objekt: Program

Objekt: Program

```
Program = function(gl, vShaderSource, fShaderSource) {

    // create a new WebGL program object
    this.gl = gl;
    this.glProgram = gl.createProgram();

    // compile and attach vertex shader
    vshader = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vshader, vShaderSource);
    gl.compileShader(vshader);
    gl.attachShader(this.glProgram, vshader);

    // compile and attach fragment shader
    fshader = this.gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fshader, fShaderSource);
    gl.compileShader(fshader);
    gl.attachShader(this.glProgram, fshader);

    // link the program so it can be used
    gl.linkProgram(this.glProgram);

    // method: enable this program for drawing
    this.use = function() {
        this.gl.useProgram(this.glProgram);
    }
}
```

Objekte in Javascript

- Objekte werden mittels eines Konstruktors definiert (nicht deklariert)
- Im Konstruktor können Instanzvariablen und Methoden mittels `this.x = ...` definiert werden

Objekt *Program*

- Konstruktor: faßt alle **OpenGL-Aufrufe** zusammen, die aus dem Shader-Sourcecode ein fertiges OpenGL-*Program* machen
- Speichert Kontext als `this.gl` und WebGL-Programm-Objekt als `this.glProgram`
- Method `use()` erlaubt es, das Programm zu „aktivieren“ und somit zwischen verschiedenen Programmen / Shadern zu wechseln

Anmerkung: auf Fehlerbehandlung wird zugunsten der Lesbarkeit auf der Folie verzichtet. Siehe Sourcecode für die Übung.

Objekt: Scene

Einfache Szene auf Basis eines einzelnen WebGL-Programms

```
Scene = function(program, backgroundColor) {

    this.program = program;
    this.bgColor = backgroundColor;
    this.shapes = new Array();

    this.addShape = function(shape) {
        this.shapes.push(shape);
    }

    this.draw = function() {

        var gl = this.program.gl; // shortcut

        gl.clearColor(this.bgColor[0], this.bgColor[1],
                     this.bgColor[2], this.bgColor[3]);
        gl.enable(gl.DEPTH_TEST);
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

        this.program.use();

        for(var i in shapes) {
            this.shapes[i].shape.draw(this.program);
        }
    }
}
```

Einfache Szene

- Basiert nur auf einem Program
- Hintergrundfarbe
- **Array** mit geometrischen Objekten

`addShape()`

- `Array.push()` fügt ein Element am Ende des Arrays hinzu

`draw()`

- Bild mit Hintergrundfarbe füllen (und Z-Buffer löschen)
- Programm aktivieren
- Schleife über alle Shapes `X`, `X.shape.draw()` aufrufen (→ später)

Anmerkung: Bei Javascript ist diese Definition möglich, ohne dass vorab etwas über den Typ der Objekte im Array `this.shapes` bekannt ist! (dynamisches Typsystem)

Wie spezifiziert und malt man mit WebGL Geometrie?



Non-OO Code für das Erzeugen und Malen von Geometrie

```
// create vertex buffer objects for our geometry
makeBuffer = function(gl, program) {
    var buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
    var vVertices = new Float32Array( [0,1,0, -1,-1,0, 1,-1,0 ] );
    gl.bufferData(gl.ARRAY_BUFFER, vVertices, gl.STATIC_DRAW);
    return buffer;
}

// draw based on buffered geometry
drawBuffer = function(gl, program, buffer) {
    (...)
    var location = gl.getAttribLocation(program, "vPosition");
    if(location < 0) {
        window.console.log("vPosition not used in shader!");
    } else {
        gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
        gl.vertexAttribPointer(location, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(location);
        gl.drawArrays(gl.TRIANGLES, 0, 3);
    }
}
```

- **makeBuffer()**
 - Erzeugt ein sog. Vertex Buffer Object (VBO), mit den Geometriedaten (bzw. später beliebige Vertex-Attribute) für ein oder mehrere Objekte
 - Lebt konzeptionell im Grafik-Speicher, also z.B. auf der GPU
- **drawBuffer()**
 - Findet heraus, über welchen „Slot“ die uniform-Variable *vPosition* im Shader angesprochen werden kann
 - Bindet das entsprechende VBO mit den Vertex-Daten und setzt einen Anfangs-Zeiger
 - Aktiviert das VBO und verbindet es mit dem uniform *vPosition*
 - Weist OpenGL an, Dreiecke zu malen, und zwar für insgesamt 3 Vertices



Objekt: VertexAttributeBuffer



Objektorientierte Version: VertexAttributeBuffer

```
VertexAttributeBuffer = function(gl, attrType, dataType,
                                numElements, dataArray)
{
    this.gl = gl;
    this.attributeType = attrType;
    this.dataType = dataType;
    this.numElements = numElements;
    this.numVertices = dataArray.length / this.numElements;

    // create the WebGL buffer object and copy the data
    this.glBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, this.glBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, dataArray,
                  gl.STATIC_DRAW);

    this.makeActive = function(program) {
        var gl = program.gl; // shortcut
        var loc = gl.getAttribLocation(program.glProgram,
                                       attribute);
        if(location < 0) { ... /* error handling */ }
        gl.bindBuffer(gl.ARRAY_BUFFER, this.glBuffer);
        gl.vertexAttribPointer(location, this.numElements,
                               this.dataType, false, 0, 0);
        gl.enableVertexAttribArray(location);
    }
}
```

- **Konstruktor**
 - Eigenschaften des Attribut-Buffers in Instanzvariablen speichern
 - Attribut-Typ bzw. Name im Shader (z.B. „vPosition“, „vColor“, „vNormal“, ...)
 - Primitiver Datentyp (gl.FLOAT, gl.INT, ...)
 - Anzahl der Elemente pro Attribut (z.B. 3 bei einem vec3, 4 bei einem vec4, etc.)
 - Das eigentliche Array mit den Daten. Hier liegen einfach alle Daten hintereinander im Speicher, z.B. 9 Floats bei drei Vertices mit je 3 Koordinaten.
 - Anzahl der Vertices berechnen
 - Aus Länge des Arrays / Anzahl der Elemente pro Attribut
 - Buffer instanzieren und die Daten hineinkopieren
 - Danach dürfen die Daten gelöscht werden, eine Kopie liegt im Grafikspeicher
- **Methode makeActive()**
 - Finde „Slot“ für die Uniform-Variable zu diesem Attribut (basierend auf dem Attribut-Typ bzw. Namen)
 - Binde und aktiviere VBO für diesen Slot



Nutzungsbeispiel für ein VertexAttributeBuffer

Nutzungsbeispiel für den VertexAttributeBuffer

```

Triangle = function(gl) {
    var vposData = new Float32Array( [ 0,1,0, -1,-1,0, 1,-1,0 ] );
    var vcolorData = new Float32Array( [ 1,0,0, 0,1,0, 0,0,1 ] );

    this.gl = gl;
    this.vpos = new VertexAttributeBuffer(gl, "vPosition", gl.FLOAT, 3, vposData);
    this.vcolor = new VertexAttributeBuffer(gl, "vColor", gl.FLOAT, 3, vcolorData);

    this.draw = function() {
        this.vpos.makeActive();
        this.vcolor.makeActive();
        gl.drawArrays(gl.TRIANGLE, 0, 3);
    }
}

```

Verallgemeinerung:
Beliebig viele Attribute...

Verallgemeinerung: Beliebige Primitiv-Typen und
beliebige Anzahl von Vertices...

- Definition eines einfachen geometrischen Objekts
 - Konstruktor: Vertex-Attribute für Position und Farbe anlegen: Position und Farbe
 - Draw():
 - Die beiden Attribute aktivieren
 - drawArrays() aufrufen, um das Objekt zu malen



Verallgemeinerung von Triangle auf Shape

Abstraktion eines Vertex-basierten geometrischen Objekts

```

VertexBasedShape = function(gl, primitiveType, numVertices) {

    this.vertexBuffers = new Array();
    this.primitiveType = primitiveType;
    this.numVertices = numVertices;

    // add a vertex attribute to the shape
    this.addAttribute = function(gl, attrType, dataType, numElements, dataArray) {
        this.vertexBuffers[attrType] = new VertexAttributeBuffer(gl, attrType, dataType,
                                                                    numElements, dataArray);

        var n = this.vertexBuffers[attrType].numVertices;
        if (this.numVertices != n) {
            /* error message - inconsistent numebr of vertices */
        }
    }

    this.draw = function(program) {

        for(attribute in this.vertexBuffers) {
            this.vertexBuffers[attribute].makeActive(program);
        }
        program.gl.drawArrays(primitiveType, 0, this.numVertices);
    }
}

```



Ableiten / Benutzen von VertexBasedShape



Ein Dreieck auf Basis eines VertexBasedShape

```
Triangle = function(gl) {

    // instantiate the shape as a member variable
    this.shape = new VertexBasedShape(gl, gl.TRIANGLES, 3);

    var vposition = new Float32Array( [ 0,1,0,   -1,-1,0, 1,-1,0 ] );
    var vcolor    = new Float32Array( [ 1,0,0,   0,1,0, 0,0,1 ] );
    this.shape.addAttribute(gl, "vertexPosition", gl.FLOAT, 3, vposition);
    this.shape.addAttribute(gl, "vertexColor",    gl.FLOAT, 3, vcolor);

}
```

- Vererbung in Javascript
 - Es wird nicht von Klasse zu Klasse vererbt, sondern dynamisch von Objekt zu Objekt
 - Jedes Objekt hat eine Eigenschaft names *prototype*, über welche Variablen und Methoden an alle Instanzen einer Klasse vererbt werden können
 - Leider gibt es ein paar unschöne Einschränkungen bei der Reihenfolge von Konstruktor- Aufrufen
- Alternative
 - Anstatt „Triangle erbt von Shape“ → verwende „Triangle hat ein Shape“
 - Ein Triangle kann beliebige Eigenschaften des zugehörigen Shape überschreiben
 - Der Nutzer eines so „abgeleiteten“ Objekts X ruft dann einfach X.shape.draw() auf



Noch ein „abgeleitetes“ Shape: TriangleFan



TriangleFan

```
TriangleFan = function(gl) {

    // instantiate the shape as a member variable
    this.shape = new VertexBasedShape(gl, gl.TRIANGLE_FAN, 10);

    var vposition = new Float32Array( [ 0,0,1, 0,1,0,   -0.7,0.7,0, -1,0,0, -0.7,-0.7,0, 0,-1,0,
                                         0.7,-0.7,0, 1,0,0,0, 0.7,0.7,0, 0,1,0 ] );
    var vcolor    = new Float32Array( [ 1,1,1, 1,0,0, 0,1,0,   0,0,1, 1,0,0,   0,1,0,
                                         0,0,1, 1,0,0, 0,1,0,   1,0,0, 1,0,0 ] );
    this.shape.addVertexAttribute(gl, "vertexPosition", gl.FLOAT, 3, vposition);
    this.shape.addVertexAttribute(gl, "vertexColor",    gl.FLOAT, 3, vcolor);

}
```

- Erzeugt einen „Fächer“ aus 10 Vertices
 - Primitiven-Typ: TRIANGLE_FAN
 - Acht Dreiecke (erster und letzter Vertex sind gleich)
- Zwei Vertex-Attribute: Position und Farbe



Etwas mehr Farbe

Vertexshader mit Farbe

```
attribute vec3 vertexPosition;
attribute vec3 vertexColor;
varying vec3 fragColor;

void main() {
    gl_Position.xyz = vertexPosition;
    gl_Position.w = 1.0;
    fragColor = vertexColor;
}
```

Fragmentsshader mit Farbe

```
precision mediump float;
varying vec3 fragColor;

void main() {
    gl_FragColor.rgb = fragColor;
    gl_FragColor.a = 1.0;
}
```

- Applikation übergibt zwei Attribute pro Vertex
 - vertexPosition
 - vertexColor
- Vertex-Shader deklariert entsprechend zwei *attribute*-Variablen
- Das Ergebnis des Vertex-Shaders sind zwei Vektoren:
 - gl_Position: fest eingebaute GLSL-Variable für die Vertex-Position
 - *varying fragColor*: benutzerdefinierte Variable für die Farbe
- Fragment-Shader übernimmt die Ergebnisse des Vertex-Shaders
 - *varying fragColor* als Input
- Ergebnis des Fragment-Shaders
 - GLSL-Variable *gl_FragColor*



Ergebnis: Interpolierte Vertex-Farben

