h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Visual Computing

Grafische Objekte und deren Programmierung

Hochschule Darmstadt

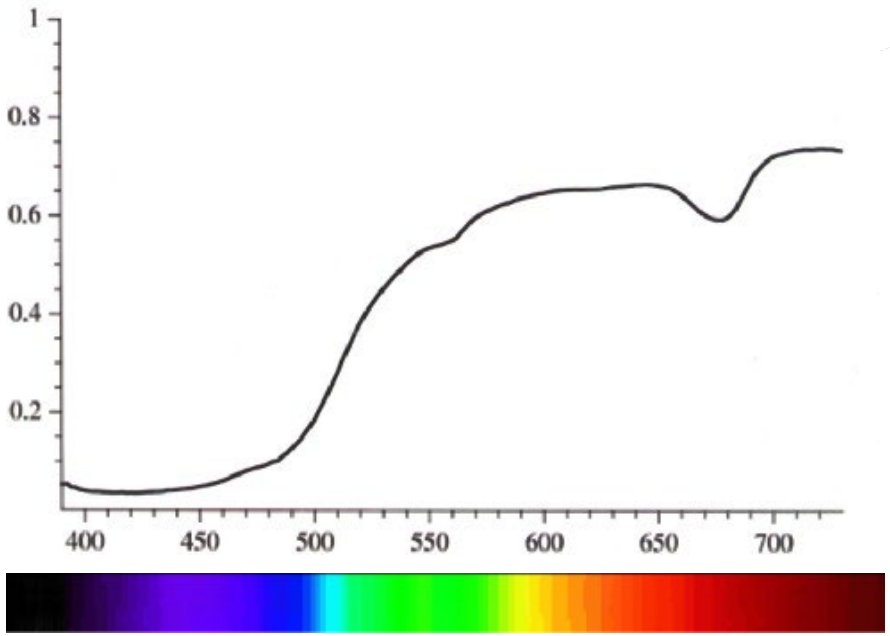
Prof. Dr. Elke Hergenröther
Björn Frömmer
Prof. Dr. Benjamin Meyer

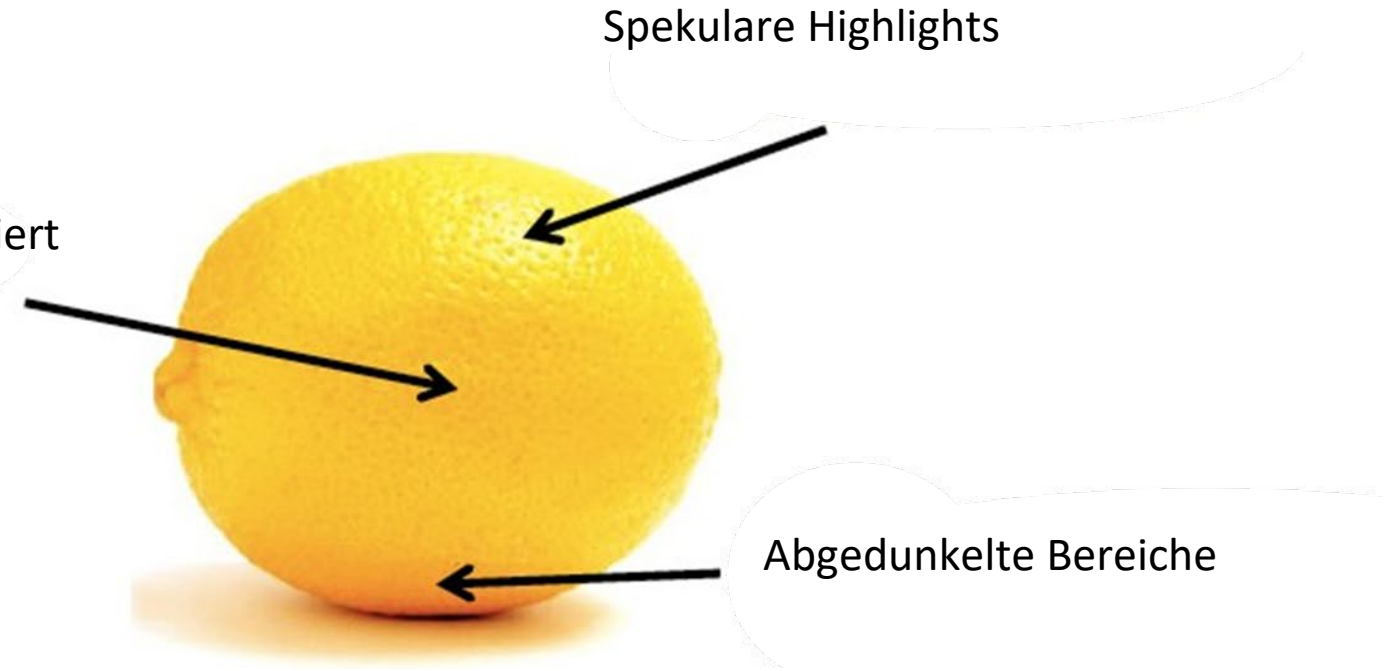
KAPITEL 3

Farben und Primitive

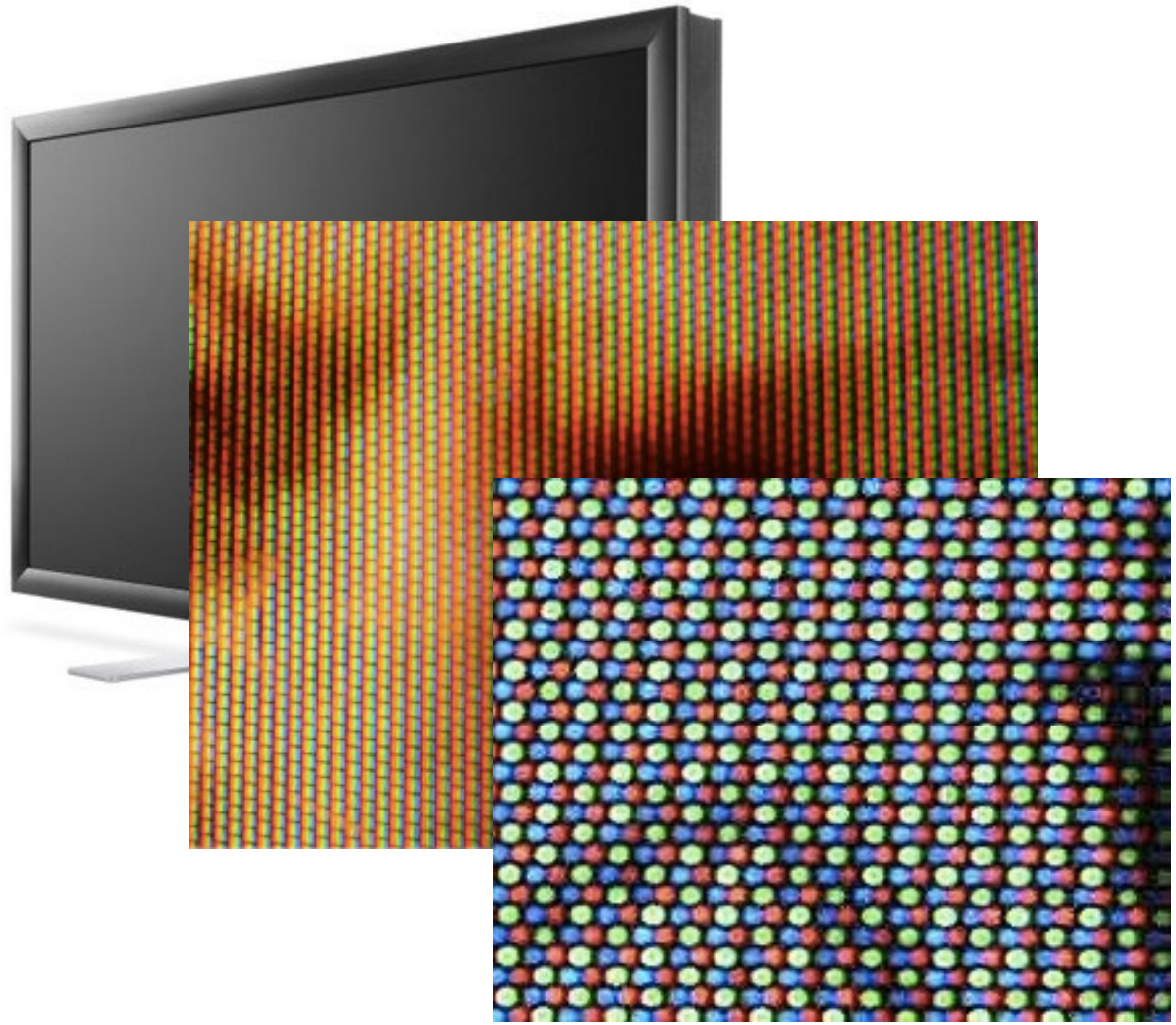
Farbräume

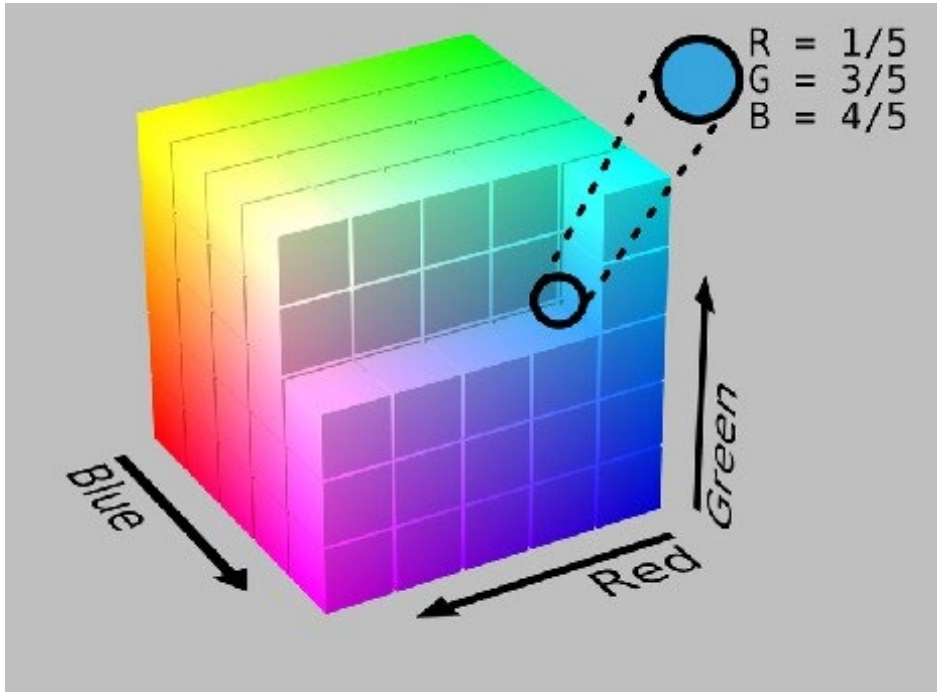
- Was ist Farbe, und wie würden Sie sie speichern?
- Physik: Reflektiertes Licht





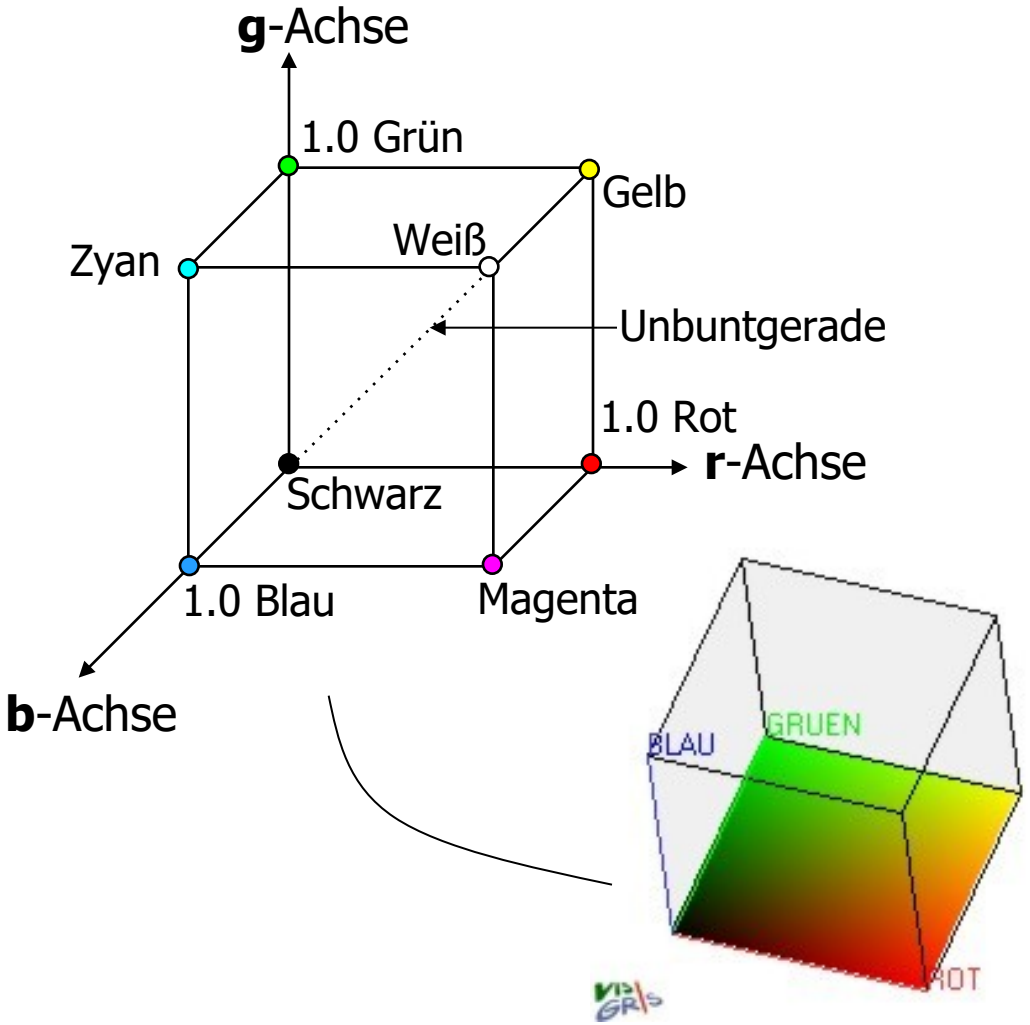
Farbräume





RGB Color Space

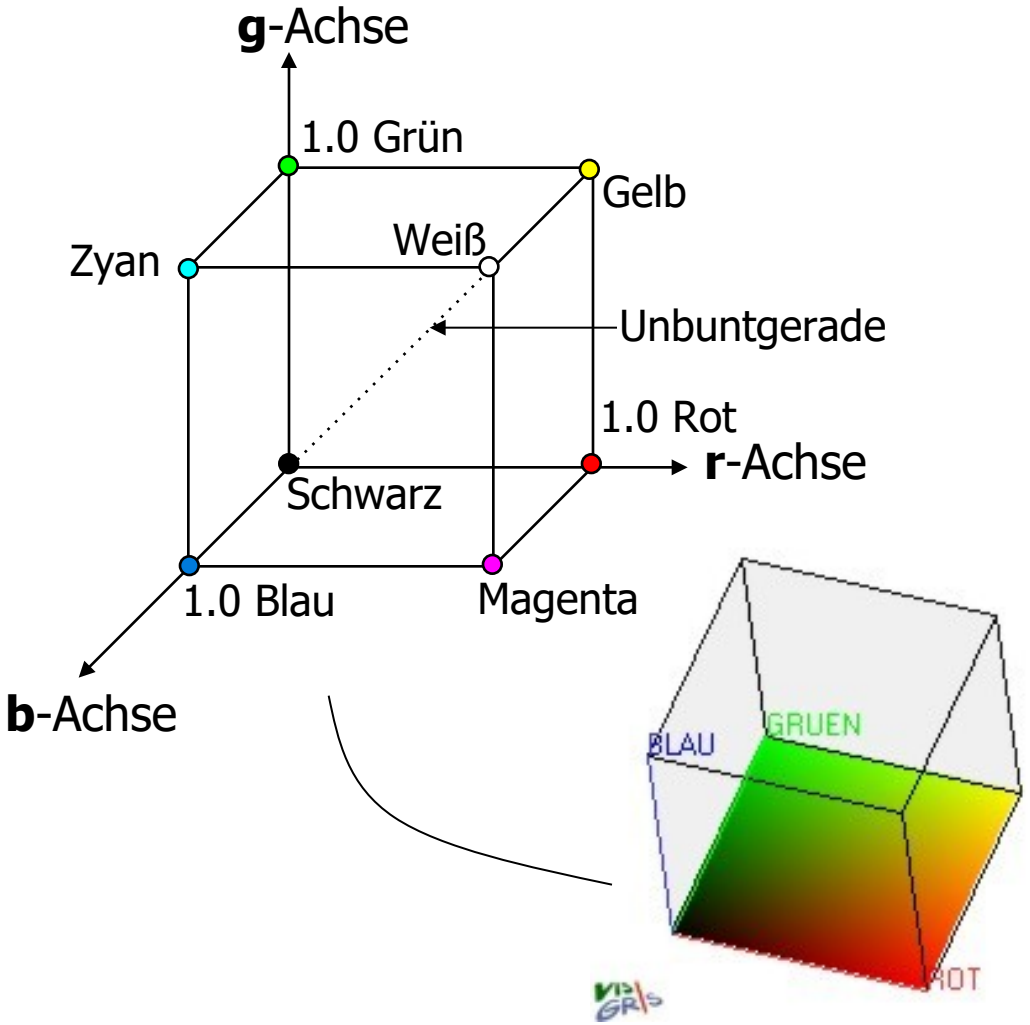
Der RGB-Farbraum



Allgemeines:

- 3D Farbraum
- Koordinatenwerte müssen zwischen 0 und 1 liegen.
- Farbe auf der Oberfläche des Würfels oder im Inneren
- Farbe wird durch einen 3D Vektor beschrieben:
  - Farbe = $[r, g, b]^t$
  - Rot = $[1, 0, 0]^t$
  - Ursprung: Schwarz $[0, 0, 0]^t$
- geringste Helligkeit: $[0, 0, 0]^t_{RGB}$
- **maximale Helligkeit?**

Der RGB-Farbraum



Zusammensetzung einer Farbe:

additive Farbmischung

Gelb = Rot + Grün
= $[1,0,0]^t + [0,1,0]^t$
= $[1,1,0]^t$

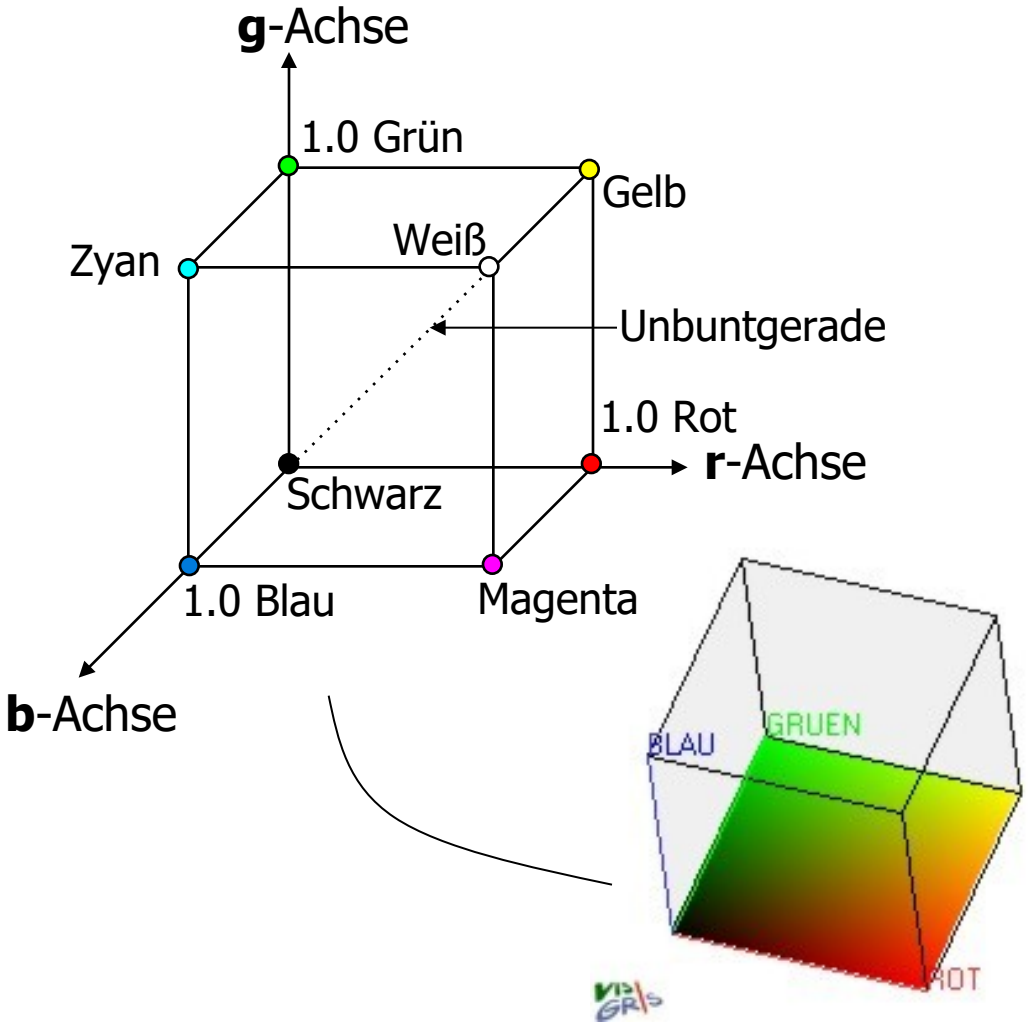
Weiß = Rot + Grün + Blau
= $[1,0,0]^t+[0,1,0]^t+[0,0,1]^t$
= $[1,1,1]^t$

Gelb (mittel hell) = $[0.5,0.5,0]^t$

Zyan = $[0,1,1]^t$

Magenta = $[1,0,1]^t$

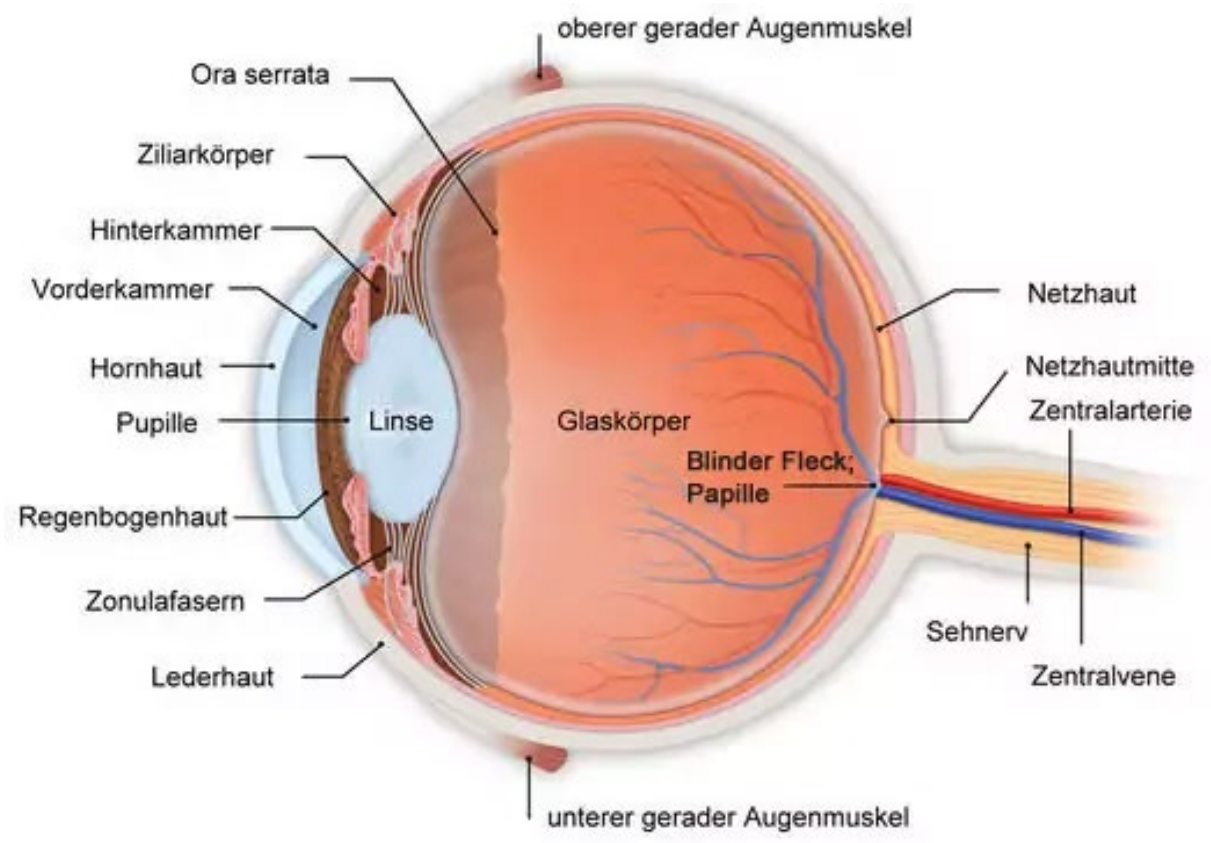
Der RGB-Farbraum



Eigenschaften des RGB-Modells

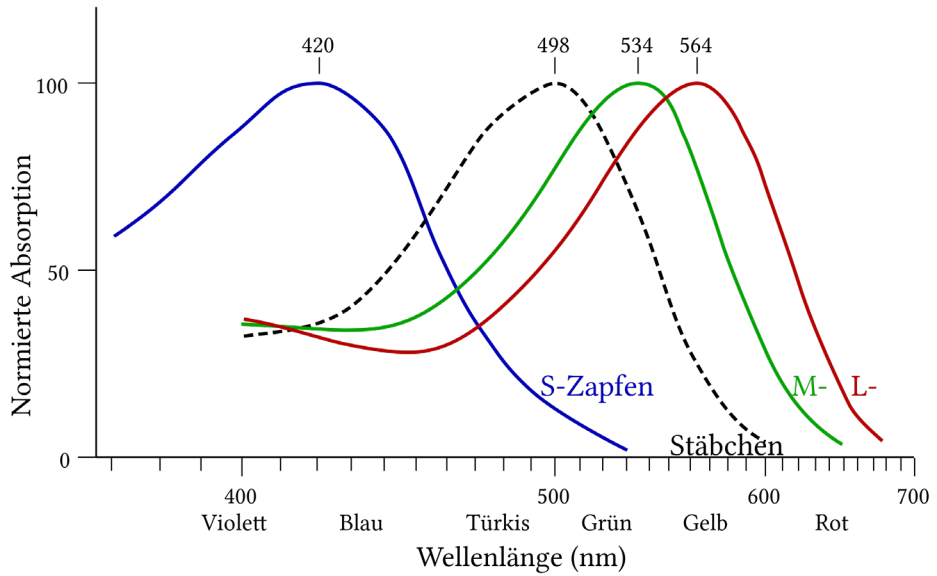
- Aussehen der Farbe wird in erster Linie durch die größte(n) Komponente(n) bestimmt.
- Wenn alle Farbkomponenten den gleichen Wert haben, handelt es sich um einen Grauton (**Unbuntgerade**)
- Wird **nicht** im Modell berücksichtigt:
  - Helligkeitsunterschiede bei blauen und grünen Farbtönen – siehe Farbwahrnehmung.

Farbwahrnehmung – das menschliche Auge



Farbwahrnehmung des Menschen

- Die *Stäbchen* sind noch bei geringer Lichtintensität unter 0,1 cd/cm² aktiv und für das *Nachtsehen* verantwortlich.
- Die drei verschiedenen Arten von *Zapfen* registrieren unterschiedliche Farbvalenzen. Jede Zapfenart hat einen spezifischen spektralen Empfindlichkeitsbereich.
  - S-Zapfen (S für Short) sind empfindlich für kürzere Wellenlängen (etwa 400–500 nm). S-Zapfen sind beim Menschen nur mit einem Anteil von zwölf Prozent aller Zapfen vertreten.
  - M-Zapfen (M für Medium) sind empfindlich für mittlere Wellenlängen (etwa 450–630 nm).
  - L-Zapfen (L für Long) sind empfindlich für längere Wellenlängen (etwa 500–700 nm).

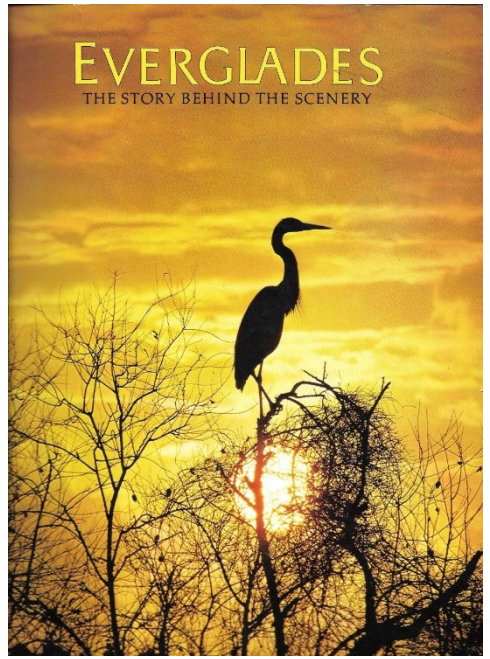


https://commons.wikimedia.org/wiki/File:Cone-response-de(2).svg

Übung: RGB-Farbmodell

Das Bild wurde in einen Rot-, einen Grün- und einen Blaukanal zerlegt.

- Welche Farbe hat der Schriftzug „EVERGLADES“?
- Welche Farben haben der Himmel, die Sonne und der Vogel?



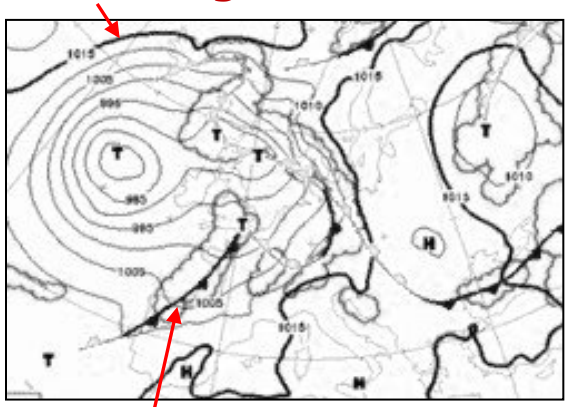
Übung: RGB-Farbmodell

Das Bild wurde in einen Rot-, einen Grün- und einen Blaukanal zerlegt.

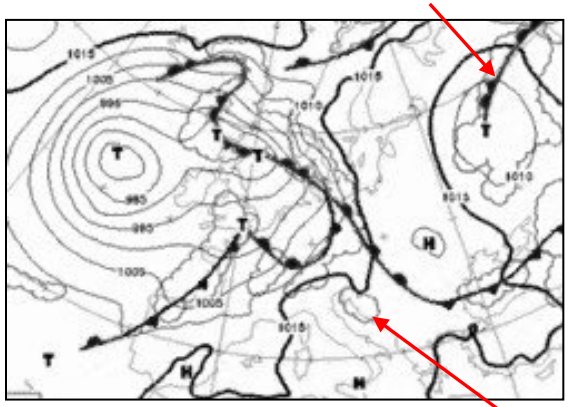
- Welche Farben haben die markierten Fronten und Tiefdruckgebiete?

Tiefdruckgebiet 1:

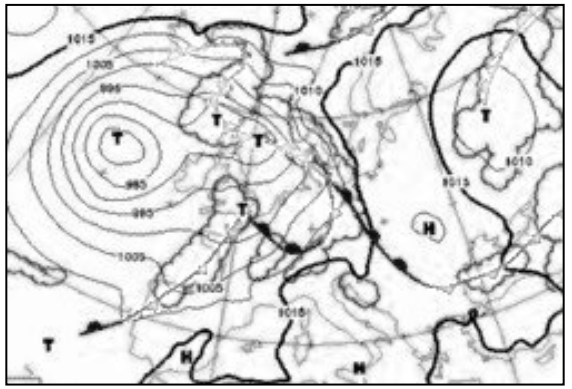
Front 2:



Rot-Kanal



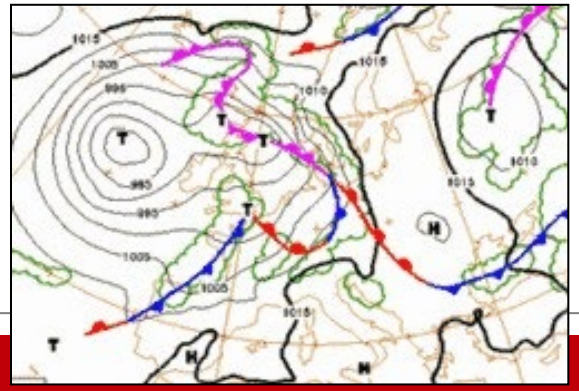
Grün-Kanal



Blau-Kanal

Front 1:

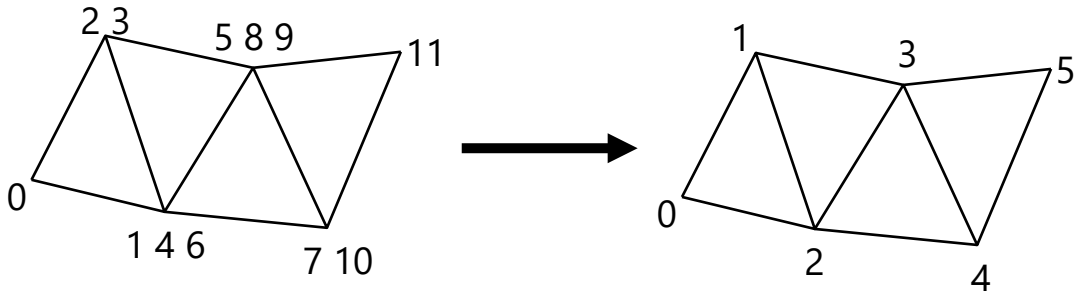
Tiefdruckgebiet 2:



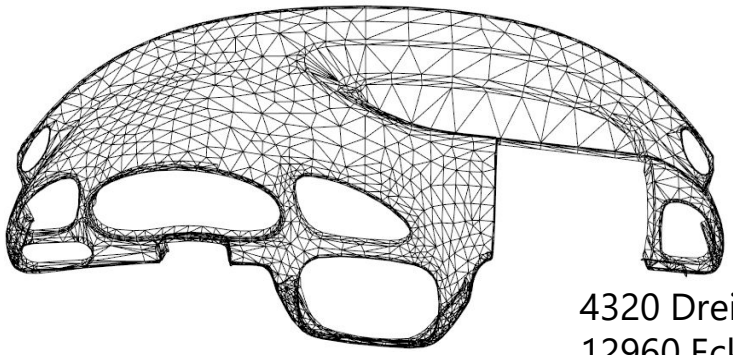
Original RGB-Bild

Triangle Strips / Dreiecksstreifen

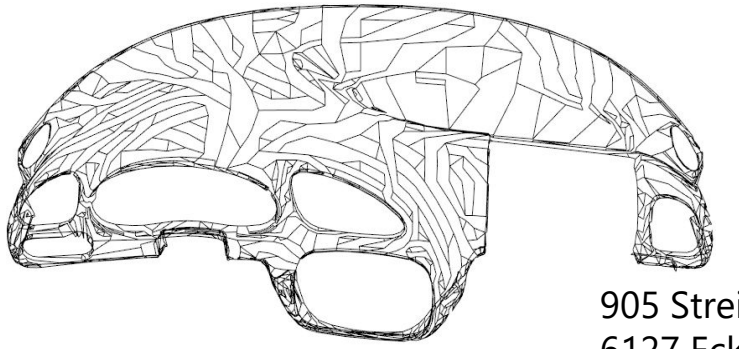
- Ziel ist es, möglichst wenig Elemente/Vertices anzulegen.
- Eckpunkte können durch zusammenhängende Strips recycelt werden.



Anzahl der Punkte ohne Streifen- und mit Streifenbildung



4320 Dreiecke
12960 Eckpunkte

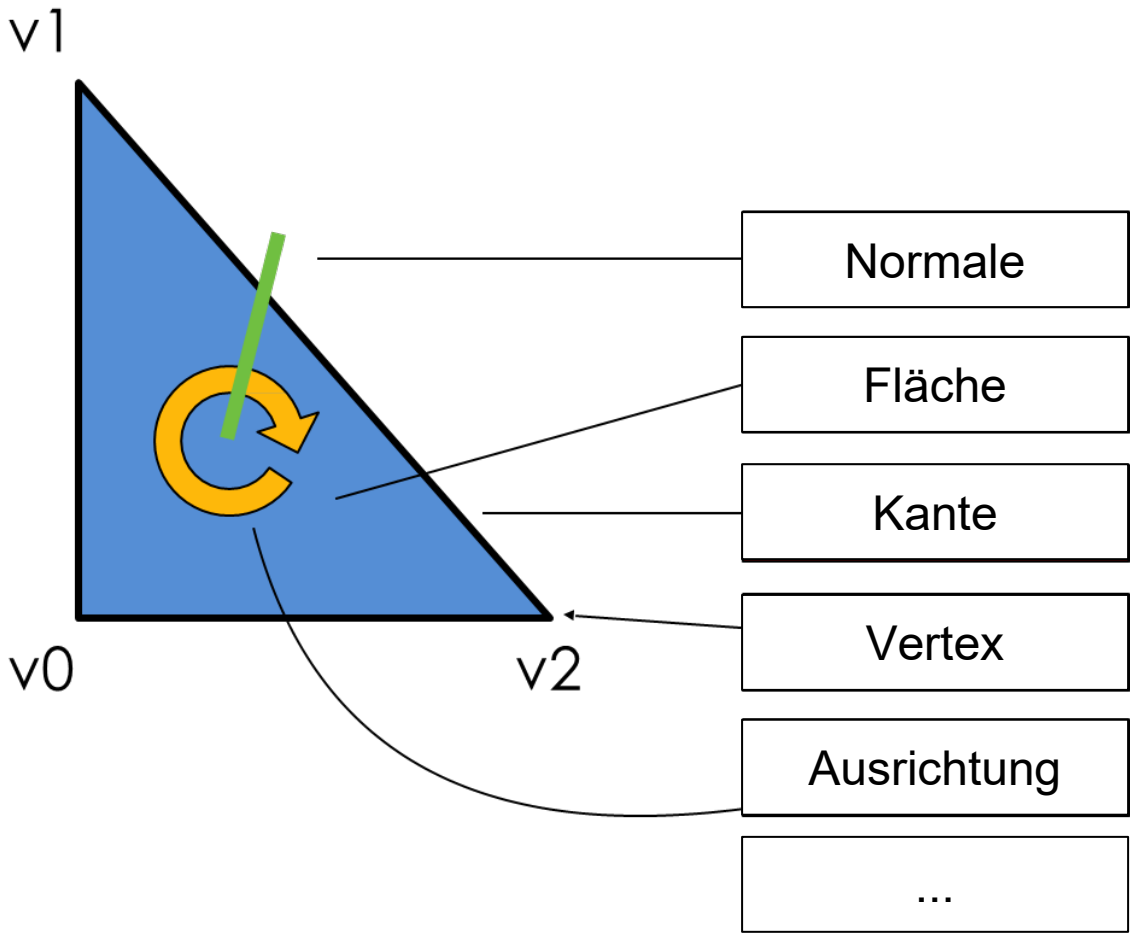


905 Streifen
6127 Eckpunkte

Szenenbeschreibung

- Was ist ein Primitiv? Es besteht aus
  - Eckpunkten/Vertices
  - Kanten
  - Flächen
  - Ausrichtung
  - Normalen

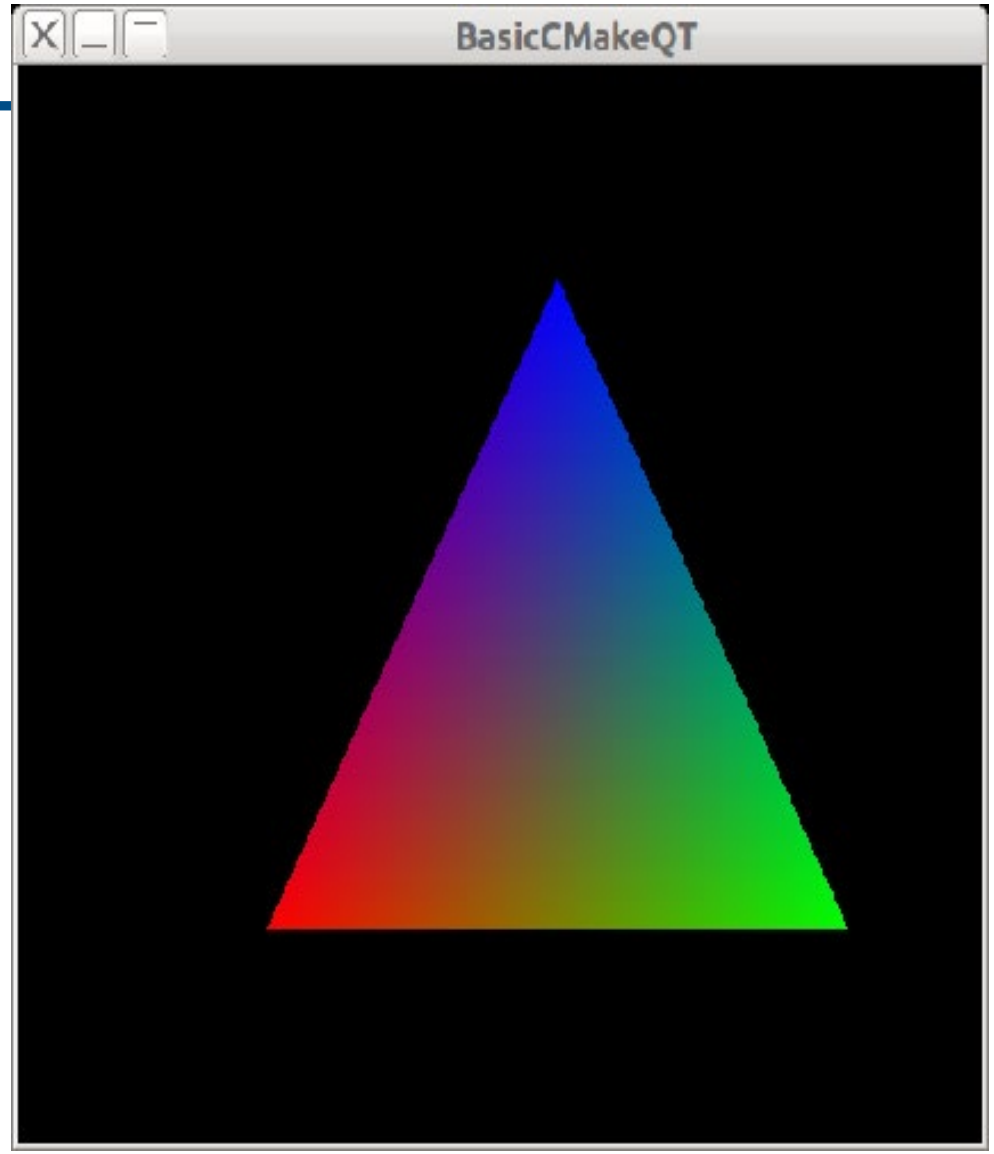
- **Aber all dies kann von den Vertices abgeleitet werden!**



Szenenbeschreibung II

Beispiel:

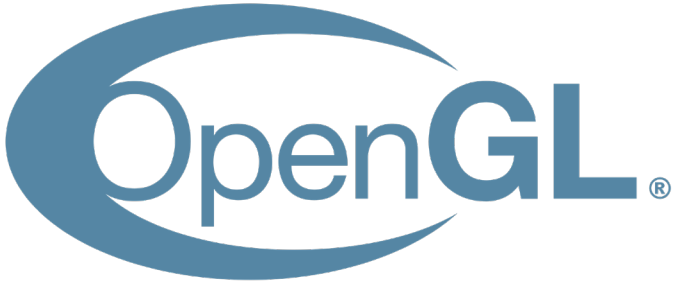
- Vertices speichern
  - 2D-Koordinaten und RGB-Farben
  - (5D insgesamt)



KAPITEL 4

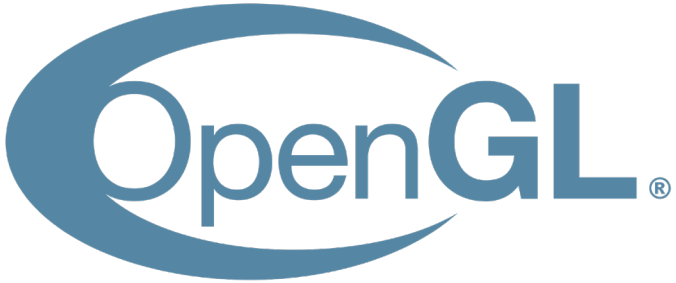
OpenGL

Allgemeines zu OpenGL



- OpenGL (Open Graphics Library) ist eine Spezifikation einer API für 3D-Graphik
- OpenGL spezifiziert (standardisiert) rund 250 Befehle
- Die Implementierung der Befehle findet man in den Grafikkartentreibern
  - Befehle werden dann entweder von der Grafikkarte ausgeführt
  - oder auf der CPU
- OpenGL ist ein Renderingsystem, keine Modellierungssoftware: komplexe Modelle müssen aus einfachen grafischen Primitiven aufgebaut werden.

Allgemeines zu OpenGL II

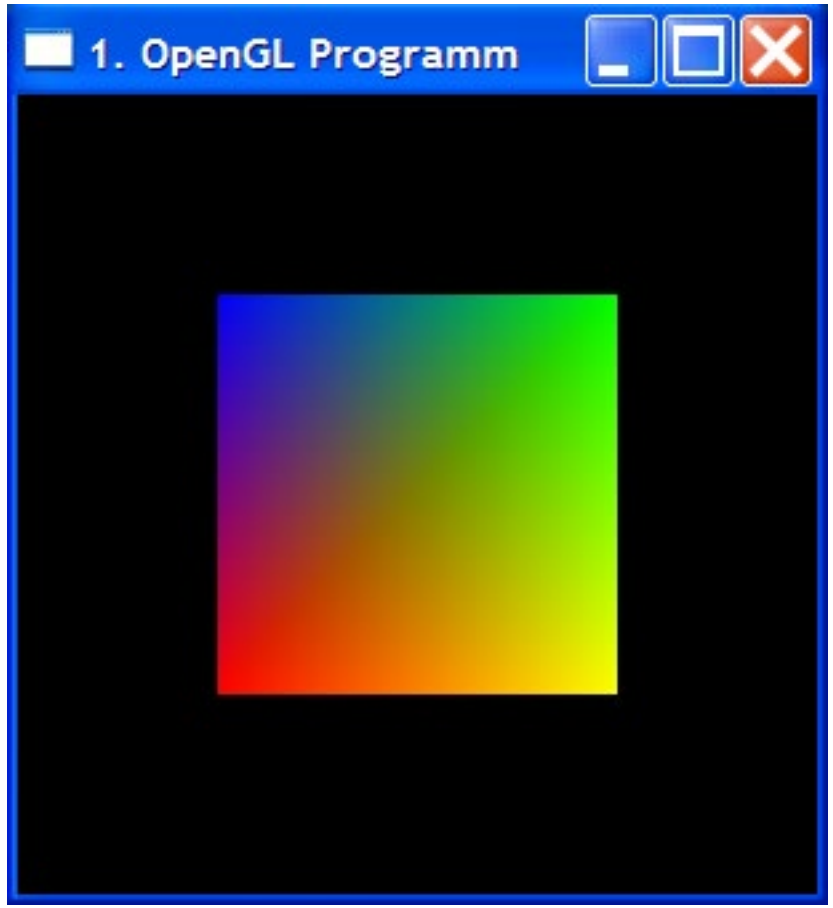


- OpenGL ist eine **State-Machine:**
  - Funktionen verändern den internen Zustand, bzw. verwenden ihn zur Darstellung.
  - Das heißt einmal angeschaltet, bleibt der betreffende Zustand aktiv bis er wieder ausgeschaltet oder umgeschaltet wird.
- OpenGL ist sehr „**explizit**“:
  - Was nicht explizit aktiviert wurde, bleibt aus.
  - Beispiel: Es nutzt nichts die Transparenz zu setzen, wenn man nicht explizit gesagt hat, dass Transparenzen berechnet werden sollen.
- Wir nutzen **GLFW** (Graphics Library Framework), eine Open Source, multiplattform-Bibliothek für OpenGL
  - Zzgl. GLEW und GLM als Erweiterungen für bestimmte Zusatzfunktionen
  - Einzige Alternative: GLUT (OpenGL Utility Toolkit), bzw. **FreeGlut** als Weiterentwicklung

Legacy-Code (OpenGL 1.x)

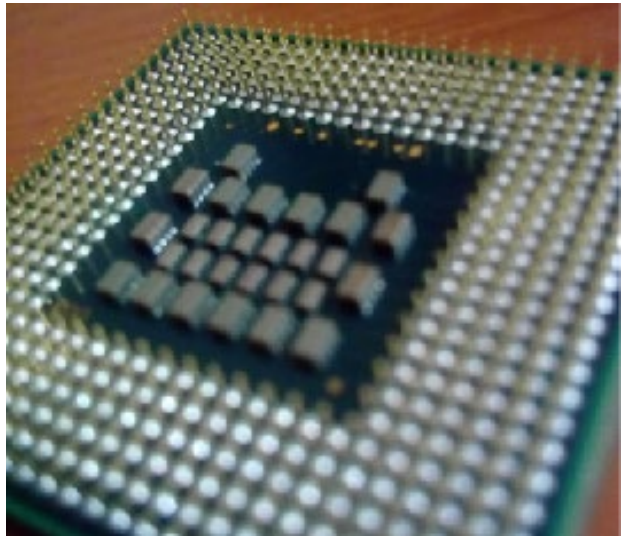
```
glBegin( GL_POLYGON );

        // State-Machine: Wenn nur der erste Farbwert
        // angegeben wird haben alle folgenden
        // Eckpunkte den gleichen Farbwert

        glColor4f( 1., 0., 0., 1.); // Rot
        glVertex3f( -0.5, -0.5, 0);

        glColor4f( 1., 1., 0., 1.); // Gelb
        glVertex3f(  0.5, -0.5, 0);

        glColor4f( 0., 1., 0., 1.); // Grün
        glVertex3f(  0.5,  0.5, 0);

        glColor4f( 0., 0., 1., 1.); // Blau
        glVertex3f( -0.5,  0.5, 0);

glEnd();
```

Ausgabe des OpenGL-Programms:



Zusammenspiel CPU und GPU

- **Heute**: OpenGL bietet Funktionen, die sowohl CPU als auch GPU ansprechen



© Lamproslefteris

Upload der Daten

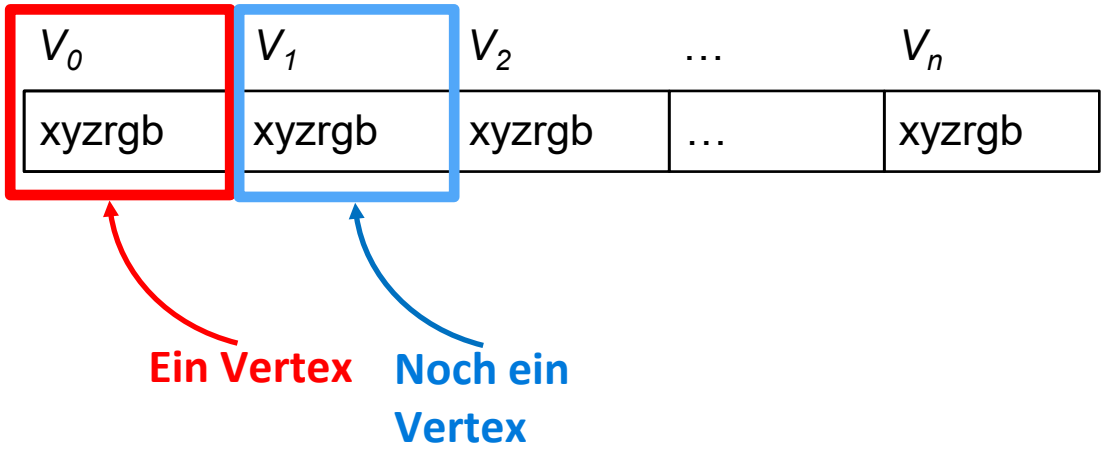
© GIGABYTE

- Datenerstellung/-manipulation
- Programmlogik

- Datenverarbeitung
- Bilderzeugung

Datenerstellung

- Die Geometrie wird als zusammenhängendes Datenarray gespeichert
  - **Interleaved**: Alle Daten eines Vertex V_i stehen hintereinander



- Im Quellcode:

```
float vertices[] = {-0.5, -0.5, 0.0, 0.0, 1.0, // Hier in 2D + Farbe (xyrgb)
                     0.5, -0.5, 0.0, 0.0, 1.0,
                     0.5,  0.5, 0.0, 1.0, 0.0,
                     0.0,  1.0, 1.0, 0.0, 0.0,
                    -0.5,  0.5, 0.0, 1.0, 0.0};
```

Das Vertex Buffer Object

- Die Geometriedaten müssen in den Speicher der GPU geladen werden
  - Hierfür stellt OpenGL spezielle Memory Buffer bereit, die sog. **Vertex Buffer Objects (VBO)**

- Objekte auf der GPU können über (eindeutige) IDs referenziert werden

- Der Datentransfer zur GPU erfolgt (fast) immer in drei Schritten:
  - Erzeugen einer ID
  - Aktivsetzen des entsprechenden Memory Buffers (siehe "State Machine")
  - Hochladen der Daten

VBOs in OpenGL (Übersicht)

ID generieren

- `void glGenBuffers(GLsizei n, GLuint * buffers);`
  - erzeugt und liefert eine (bzw. n) Buffer-ID, die in `buffers` gespeichert wird/werden
  - `n:` Gibt die Anzahl der zu generierenden Bufferobjekte an, meistens 1.

Korrekten Buffer aktivieren

- `void glBindBuffer(GLenum target, GLuint bufferID);`
  - `target:` Art des zu beschreibenden Buffers
    - `GL_ARRAY_BUFFER:` Buffer mit den eigentlichen Geometrie-Daten
    - `GL_ELEMENT_ARRAY_BUFFER:` Buffer mit Indizes auf ein anderes VBO
  - `bufferID:` Die zuvor mit glGenBuffers erzeugte ID des VBOs

Daten löschen

- `glDeleteBuffers(GLuint bufferID);`
  - Löschen eines Buffers mit der ID `bufferID`

VBOs in OpenGL (Übersicht)

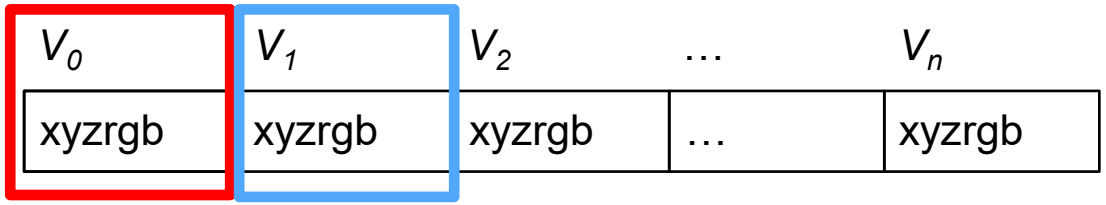
Hochladen der Daten auf die GPU

- `glBufferData(GLenum target, GLsizeiptr size, const void * data, GLenum usage);`
  - `target`: Art des zu beschreibenden Buffers (wie bei `glBindBuffer`)
    - `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, …
  - `size`: Größe der zu schreibenden Daten **in Bytes**
  - `data`: pointer auf das erste Element der hochzuladenden Daten
  - `usage`: Art des (erwarteten) späteren Zugriffs auf die Daten
    - `GL_STATIC_DRAW`: einmalig initialisiert, häufig gerendert
    - `GL_DYNAMIC_DRAW`: häufig verändert und gerendert
    - `GL_STREAM_DRAW`: selten verändert oder gerendert
    - …

VBOs in OpenGL

Frage: Wie sehen die Daten in den VBOs aus?

- OpenGL ist dumm! Es sieht nur ein Bündel von zugewiesenem Speicher



VBO mit Vertexdaten

- Es wird eine zusätzliche Erklärung benötigt, wie die Daten zu interpretieren sind!
  - für jedes zu rendernde Objekt wird ein **Vertex Attribute Object** erzeugt
  - Hier werden Zeiger auf die einzelnen Attribute gespeichert (Position, Farbe, etc.)

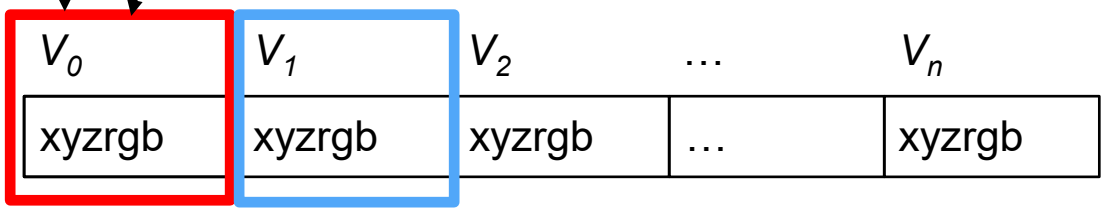
Das Vertex Array Object (VAO)

Jeder Attributszeiger beschreibt ein Vertex-Attribut

Attribut-Index	Beschreibung
0	"3 Koordinaten vom Typ GL_FLOAT pro Vertex"
1	"3 Koordinaten vom Typ GL_FLOAT pro Vertex"
...	...

1. Attribut: Vertex-Position

2. Attribut: rgb-Farbe



Aber Achtung:
Die Interpretation (Koordinaten, Farbe etc.) wird erst später festgelegt.

VAOs in OpenGL

ID generieren (analog zum VBO)

- `void glGenVertexArrays(GLsizei n, GLuint * arrays);`
  - erzeugt und liefert eine (bzw. n) Array-ID, die in `arrays` gespeichert wird/werden
  - `n:` Gibt die Anzahl der zu generierenden Bufferobjekte an, meistens 1.

Korrektes VAO aktivieren

- `void glBindVertexArray(GLuint vaoID);`
  - `vaoID:` Die zuvor mit glGenVertexArrays erzeugte ID des VAOs
  - **Nur ein VAO kann zeitgleich aktiv sein!**

VAO löschen

- `void glDeleteVertexArrays(GLuint vaoID);`
  - `vaoID:` Die zuvor mit glGenVertexArrays erzeugte ID des VAOs

VAOs in OpenGL II

Definieren der Vertexattribute (OpenGL mitteilen, wo welche Daten zu finden sind)

- `glVertexAttribPointer(GLuint index, GLuint size, GLenum type, GLboolean normalized, GLsizei stride, const void * offset);`
  - `index`: Nutzerdefinierte ID des Attributes (wird später auf der GPU benötigt)
  - `size`: Anzahl der "Koordinaten" pro Vertex, meistens 3
  - `type`: Datentyp, z.b. GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
  - `normalized`:
    - false: Fließkommawerte werden bereitgestellt, keine Normalisierung erforderlich
    - true: die bereitgestellten Werte werden auf den Bereich [0,1] für vorzeichenlose Daten und [-1,1] für vorzeichenbehaftete Daten abgebildet
  - `stride`: Größe eines Vertex **in Bytes**.
  - `offset`: Speicher-Offset **in Bytes**, an dem das Vertex-Attribut im aktiven Array beginnt (0 für das erste Attribut)
- glVertexAttribPointer bezieht sich immer auf das **derzeit aktive VBO (und VAO) !**

VAOs in OpenGL III

Ein- und Ausschalten einzelner Attribute

- `glEnableVertexAttribArray(GLuint index);`
- `glDisableVertexAttribArray(GLuint index);`
  - Der Index entspricht dem selbstgewählten Index in

    `glVertexAttribPointer(GLuint index, ...)`

- Per Default sind alle Attribute deaktiviert!

- Anwendungsbeispiel:
  - Zwei Rendermodi: Einer zur Visualisierung der Position (z.B. zur Fehlersuche) und einer zur reinen Farbdarstellung können in derselben VAO kombiniert werden

Codebeispiel

VBO erstellen für Position und Farbe

```
float[] vertices = … // stores the data of your vertices
// in this example 3 float for position followed by 3 floats for color
GLuint vaoID, vboID;

// generate and activate VBO and upload data //
glGenBuffers(1, &vboID);
glBindBuffer(GL_ARRAY_BUFFER, vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);

// generate and activate VAO //
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);

// describe VBO in the VAO //
glVertexAttribPointer(0, 3, GL_FLOAT, false, 24, 0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, false, 24, (void*)12);
glEnableVertexAttribArray(1);
```

Es werden zwei Attributzeiger definiert, die Vertexpositionen erhalten die ID 0 und die Farben die ID 1 im VAO

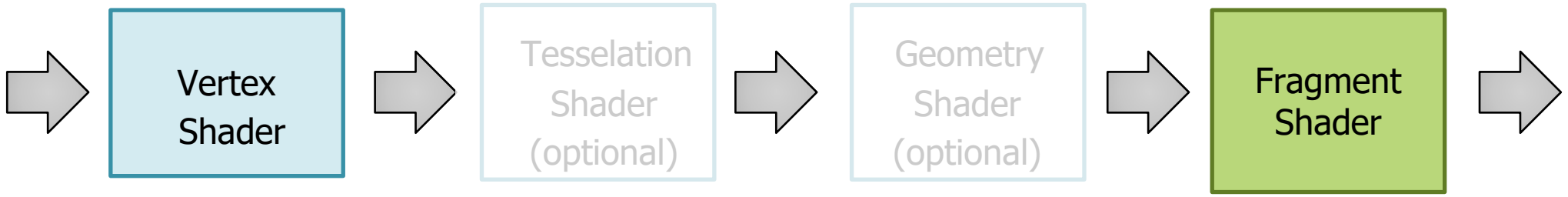
float = 4 byte

Beachte den Offset und den seltsamen Datentypen!

Datenverarbeitung

- Die hochgeladenen Daten können vor dem Rendern noch manipuliert werden.
- **Vorteil:** Die Berechnungen werden pro Vertex bzw. pro Pixel parallel ausgeführt

- Die gewünschten Berechnungen werden in sog. **Shadern** implementiert, kleinen Programmfragmenten, die direkt auf der GPU ausgeführt werden können.



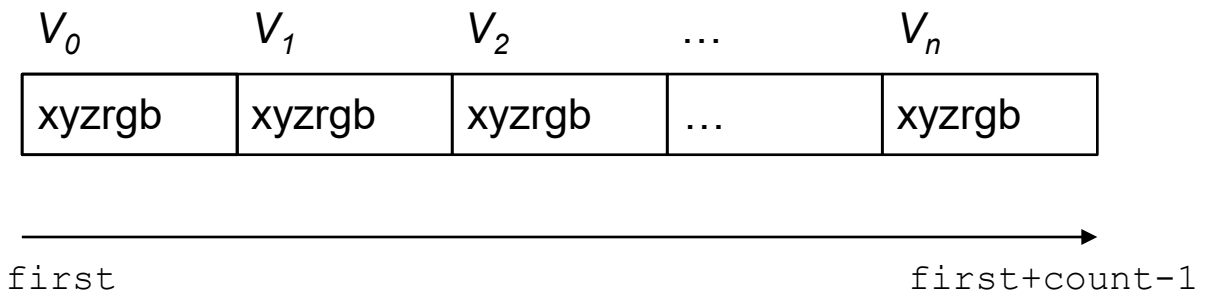
- Jeder Shader hat hierbei eine genau definierte Aufgabe:
  - **Vertex Shader:** Berechnet die **finale Position** jedes Vertex im Ausgabebild
  - **Fragment Shader:** Berechnet die **Farbe jedes Pixels** im Ausgabebild

Bilderzeugung

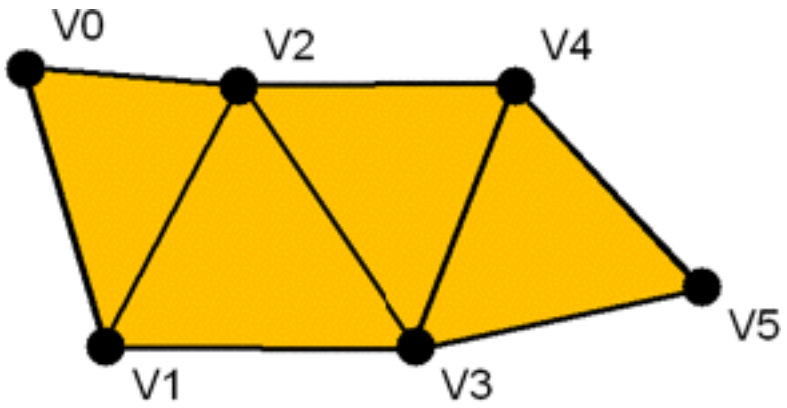
Rendering eines Vertex Array Objects (und allen verbundenen VBOs)

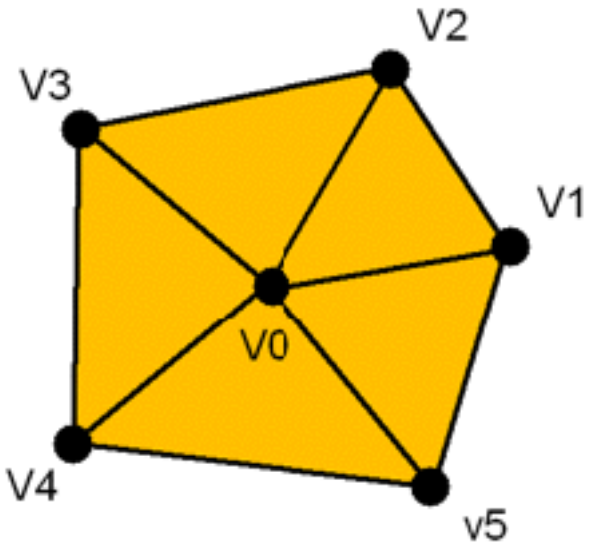
- `glDrawArrays(GLenum mode, GLint first, GLsizei count);`
  - Geht das gesamte VAO durch, startet beim Primitiv `first` und zeichnet die ersten `first + count - 1` Primitive
  - `mode`: Art der zu zeichnenden Primitive, z.B. `GL_TRIANGLES`

- Unflexiblel, aber schnell



Index Buffer Objects



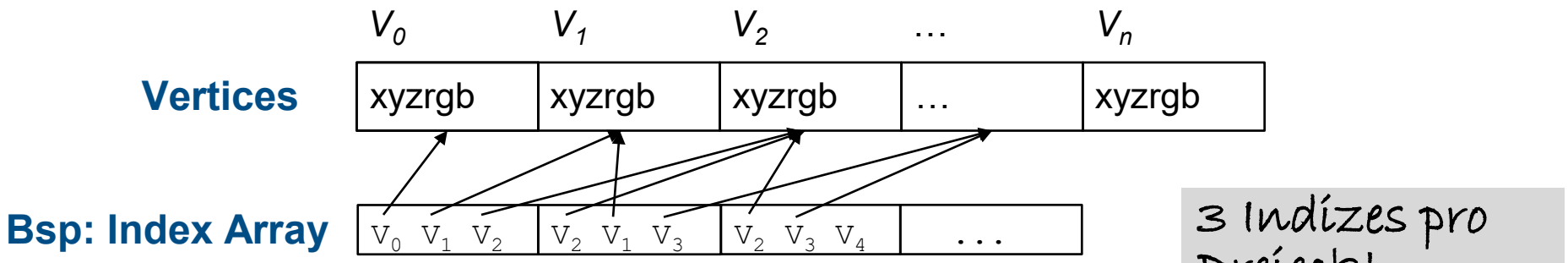


- Vertices werden häufig von Dreiecken gemeinsam genutzt (vgl. Folie *Triangle Strips*)
- Wiederverwendung von Vertices durch Indizierung
- Zusätzlicher Element-Array-Buffer erforderlich
  - `glGenBuffers(1, &iboID);`
  - `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboID);`
  - `glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(data), &data, GL_STATIC_DRAW);`

Index Buffer Objects II

Dereferenzierung über Indizes

- `glDrawElements(GLenum mode, GLsizei count, GLenum type, const void* indices);`
  - `mode`: Zeichnungsprimitiv, z.B. `GL_TRIANGLES`
  - `count`: Anzahl der Elemente (d.h. Indizes, nicht komplette Dreiecke! → Größe des Indizes-Arrays)
  - `type`: Typ der Daten in den Indizes, `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, `GL_UNSIGNED_INT`
  - `indices`: Zeiger auf den Anfang des aktiven Index-Arrays, normalerweise 0.



3 Indizes pro Dreieck!

Index Buffer Objects III

Unterschied zwischen glDrawArrays() und glDrawElements()

- `glDrawArrays()`: Brute-force Ansatz
- `glDrawElements()`: Flexibler (und manchmal schneller)
  - Wiederverwendung von Vertices für geringeren Datentransfer
  - Anpassung des IBOs ermöglicht es, nur Teile eines Objektes zu rendern

Codebeispiel mit IBO

VBO erstellen für Position, mit einem Index Buffer Object

```
// given: array of vertices and index array //
float[] vertices = …
int[] indices = …
GLuint vaoID, vboID;

// setup VBO //
glGenBuffers(1, &vboID);
glBindBuffer(GL_ARRAY_BUFFER, vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);

// setup VAO //
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 12, 0);
glEnableVertexAttribArray(0);

// setup IBO //
GLuint iboID;
glGenBuffers(1, &iboID);   //only works after glGenVertexArrays();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Codebeispiel mit IBO

Rendering with IBO

```
void setup(){
        // given: array of vertices and index array //
        // setup VAO //
        // setup VBO //
        // setup IBO //
}

void render(){
        // activate VAO //
        glBindVertexArray(vaoID);
        // render call //
        glDrawElements(GL_TRIANGLES, count, GL_UNSIGNED_INT, 0);

        // good programmers should reset //
        glBindVertexArray(0);
}
```

VBOs sind implizit aktiviert

Anzahl an Indizes, nicht Primitiven!

Primitive und Datentyp (des IBO) spezifizieren