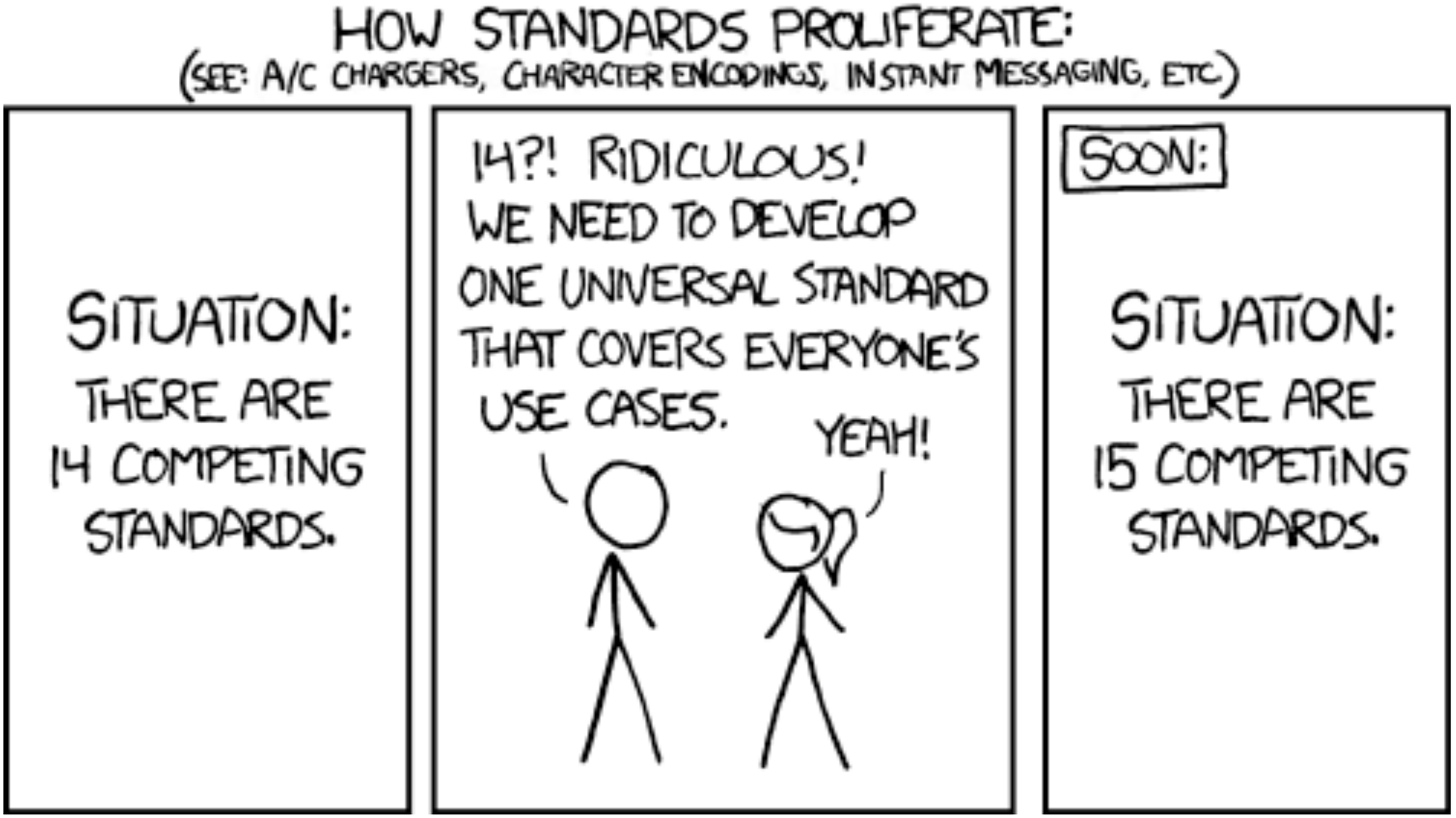
Bremen

CG VR

Computergraphik I
Die Graphik-Pipeline & Einführung in OpenGL 3+

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

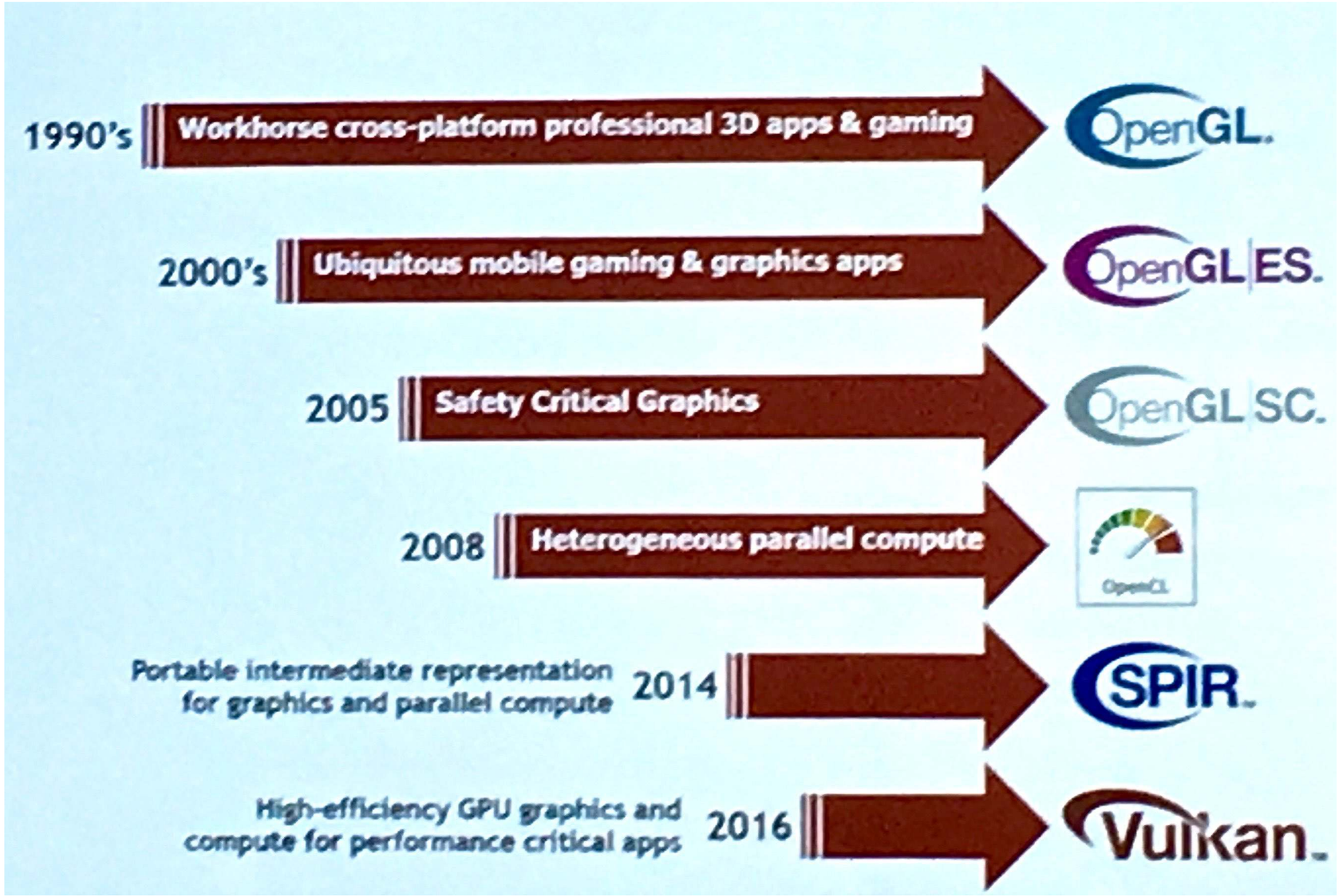

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

xkcd

OpenGL

- OpenGL ist ein Software-Interface für Graphik-Hardware mit ca. 250 verschiedenen Kommandos
  - Hardware-unabhängig, Hersteller-unabhängig
- Warum „Open“?
  - Weiterentwicklung durch die Khronos-Gruppe
    - NVIDIA, ATI, IBM, Intel, SGI, ...
  - Von jedem Lizenznehmer erweiterbar (mittels Extensions)
- Nicht dabei (ein Feature): Handhabung von Fenstern/Windows, Benutzereingabe
- Der Standard im embedded Bereich (OpenGL ES)
  - Smartphones, car infotainment, aircraft cockpits, ...

Khronos Open Standards for Graphics and Compute



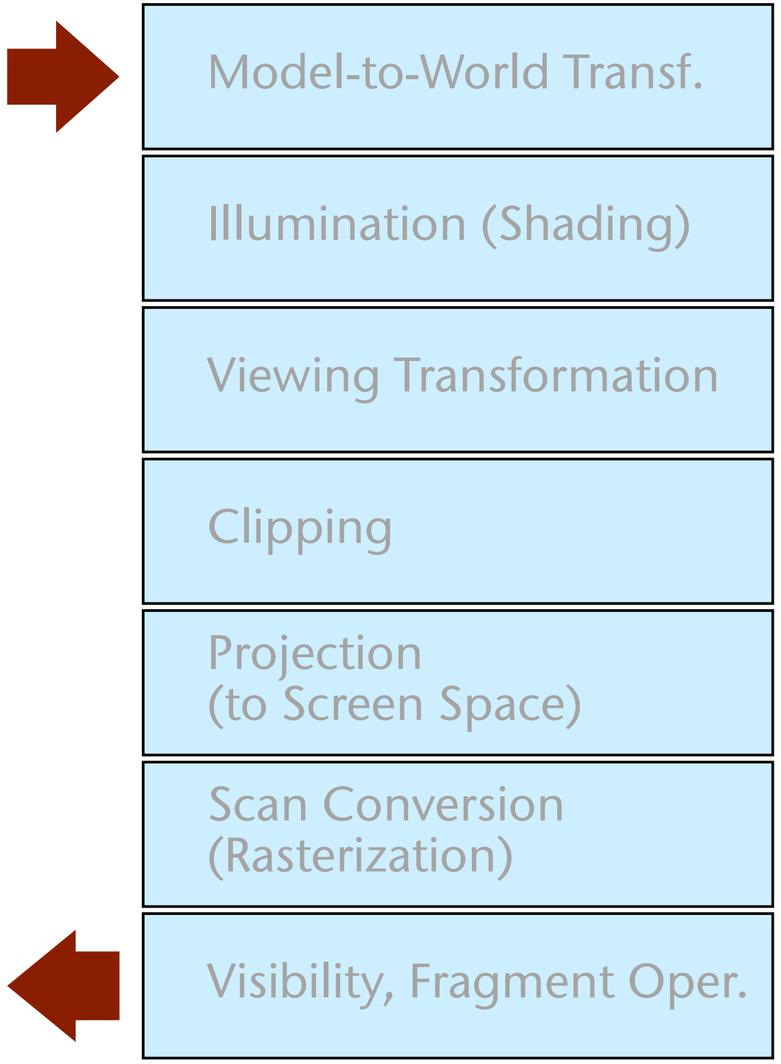
[Khronos Group 2016]

Die Grundstruktur von OpenGL-Programmen

- 3 Arten von Funktionen
  - Zustand ändern
  - Daten (z.B. Vertex-Koordinaten) übergeben
  - Geometrie darstellen
- Klarer Namensraum
  - Befehle fangen mit `gl`... an
  - Konstanten mit `GL_`...
- Darauf aufbauend gibt es diverse Tools:
  - Glm: Mathematische Funktionen (Vektoren, Matrizen, ...)
  - glew/glfw/...: Libraries zur Benutzerinteraktion, Fenstermanagement
  - Qt: beinhaltet Kombination und noch viel mehr

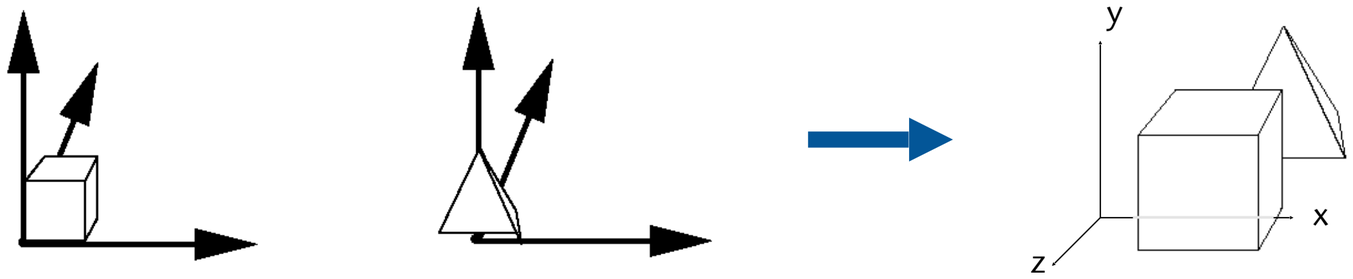
"A Trip Down the Graphics Pipeline" (GPU)

- Input:
  - Geometric model: description of all objects, surfaces,
  - Transformations
  - Lighting model: Computational description of object and light properties, interaction (e.g., reflection)
  - Synthetic Viewpoint (or Camera): Eye position and viewing frustum
  - Raster Viewport (Frame Buffer): Pixel grid onto which image plane is mapped
- Output:
  - Colors / Intensities suitable for framebuffer display (for example, 24 Bits RGB values at each pixel)



Model Transformation (Model-to-World)

- Jedes 3D Modell (Objekt) wird im eigenen Koordinatensystem definiert (object space)
- Model Transformation positioniert die Objekte in einem allg. Koordinatensystem (world space)

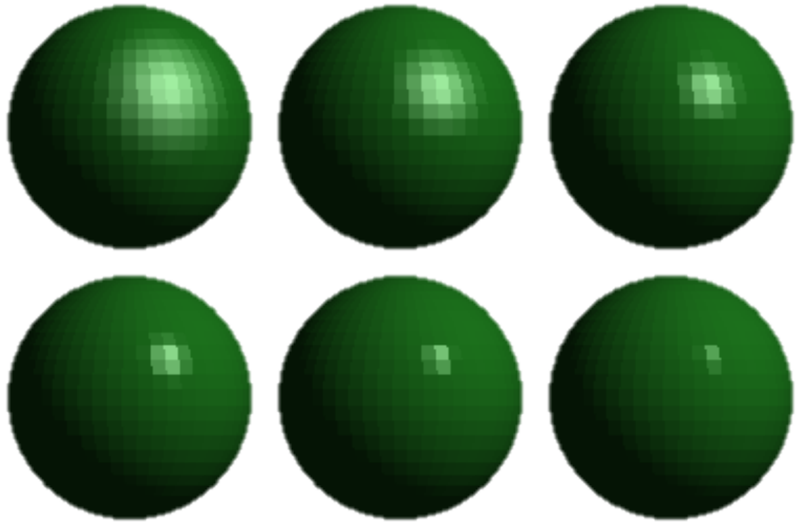


Object space(s) World space

- **Model-to-World Transf.**
- Illumination (Shading)
- Viewing Transformation
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility, Fragment Oper.

Illumination (Beleuchtung, Schattierung)

- Beleuchten von Polygonen (für Schattierung und Highlights) gemäß der Material-Eigenschaften, Oberflächen-eigenschaften und Lichtquellen
- Lokale Beleuchtungsmodelle (kein Lichttransport von einem Obj zum anderen)



Model-to-World Transf.

Illumination

Viewing Transformation

Clipping

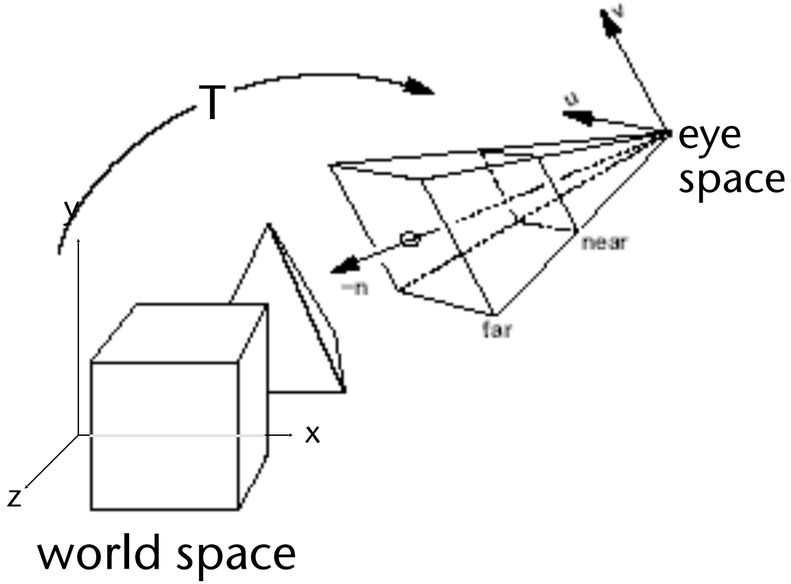
Projection
(to Screen Space)

Scan Conversion
(Rasterization)

Visibility, Fragment Oper.

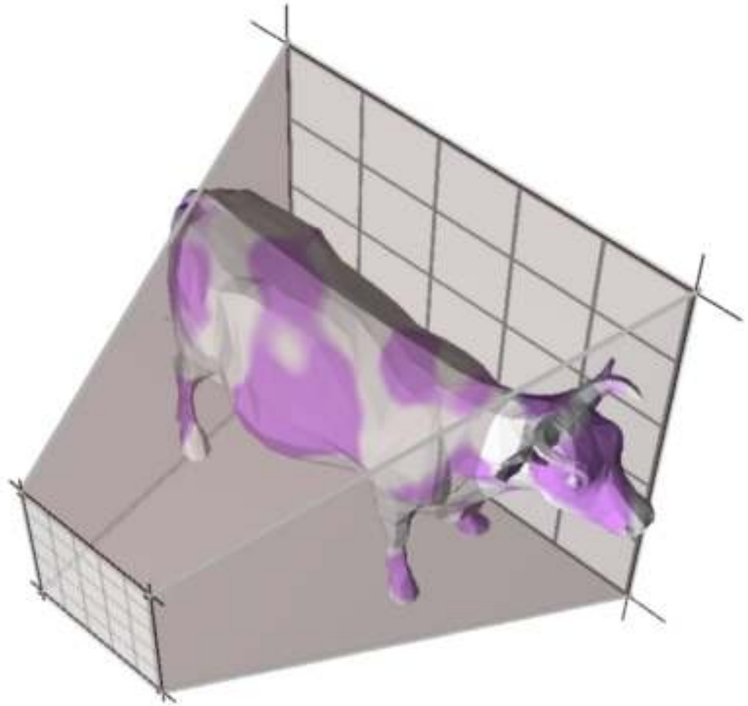
Viewing Transformation

- Umwandeln von Welt-Koord. nach Kamera-Koord.
- Bestimme eine Transformation T für die komplette Szene so, daß die Betrachter-Position in den Ursprung verschoben wird und Blickrichtung entlang der (-Z)-Achse ist

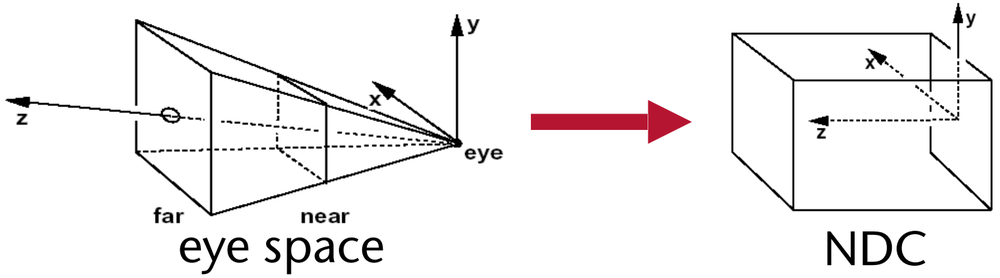


Clipping

- Abschneiden der Polygone, die außerhalb des sichtbaren Bereiches liegen (view frustum)

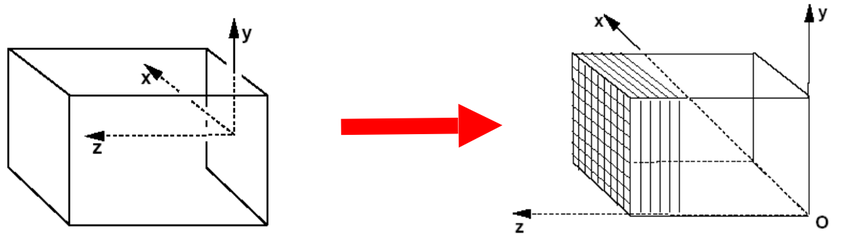


- Trick: transformiere Szene in normalisierte Koordinaten (NDC)

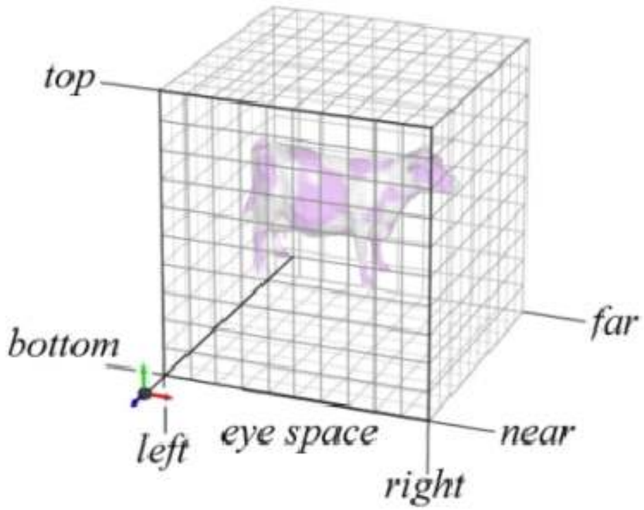


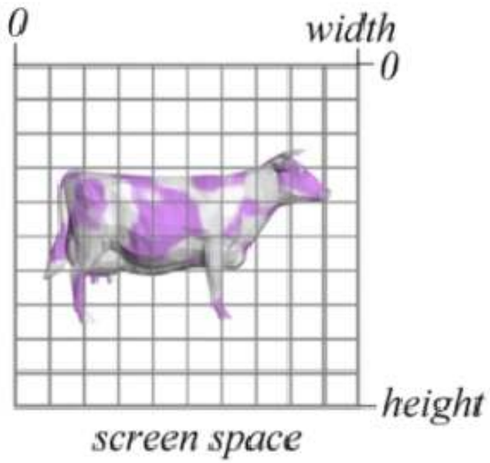
Projektion

- Das Objekt wird nach 2D projiziert und räumlich diskretisiert (*screen space*)



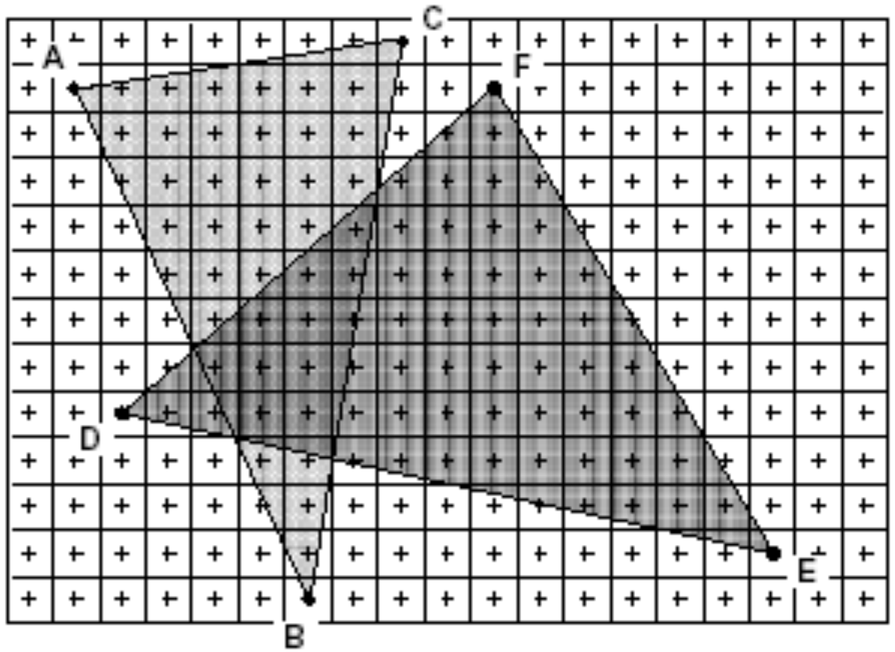
NDC Screen Space





Scan Conversion (Rasterisierung)

- Rasterisierung (= Diskretisierung) der Polygone in Pixel
- Gleichzeitig Ecken-Werte interpolieren (Farbe, Tiefenwert, ...)



Model-to-World Transf.

Illumination (Shading)

Viewing Transformation

Clipping

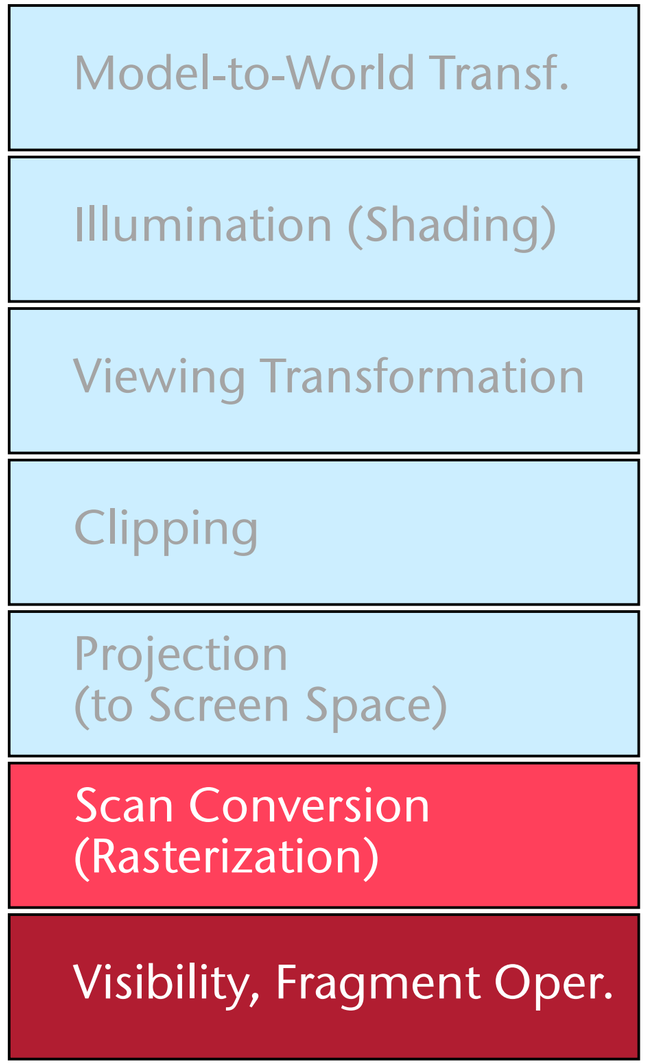
Projection
(to Screen Space)

Scan Conversion
(Rasterization)

Visibility, Fragment Oper.

Visibility (Sichtbarkeit) und Operationen

- Verdeckungen bestimmen
- Evtl. weitere Pixel-Operationen:
  - Blending mit vorhandenem Frame-Buffer-Inhalt
  - Maskierung (z.B. wegen Verdeckung durch andere Fenster)
  - Farb-Transfer



Definition Vertex

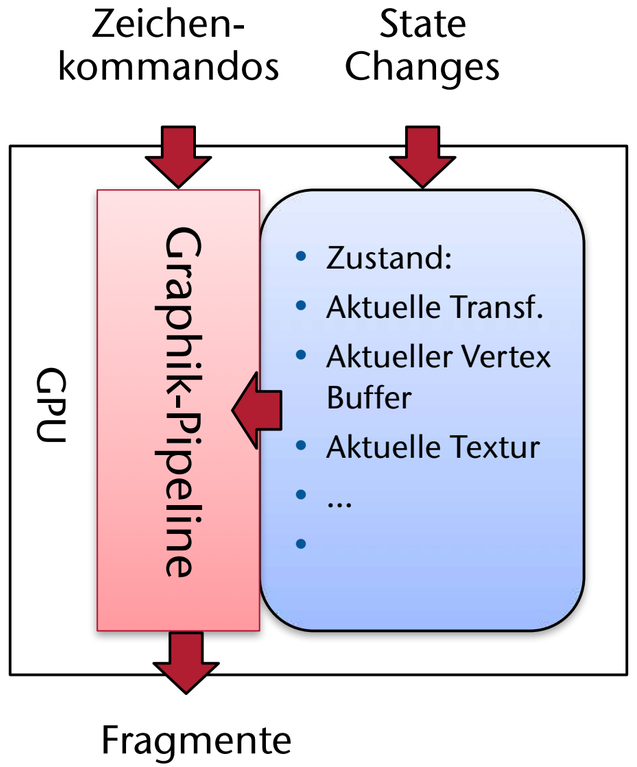
- Vertex = "Eckpunkt" = alle Attribute zur vollständigen Spezifikation eines Eckpunktes eines Primitives
  - 3D Koordinaten (zunächst in Modelling Coords!)
  - Farbe (evtl.)
  - Normale (meistens)
  - Textur-Koordinaten (u,v)
  - User-defined attributes (soviele man möchte)

Definition Fragment und Pixel

- Achtung: unterscheide zwischen Pixel und Fragment!
- Pixel :=
  eine Anzahl Bytes im Framebuffer bzw. ein Punkt auf dem Bildschirm
- Fragment :=
  eine Menge von Daten (Farbe, Koordinaten, Alpha, ...), die zum Einfärben eines Pixels benötigt werden
- M.a.W.:
  - Ein Pixel befindet sich am Ende der Pipeline (im Framebuffer)
  - Ein Fragment ist ein "Struct", das durch die Pipeline "wandert" und am Ende in ein Pixel gespeichert wird

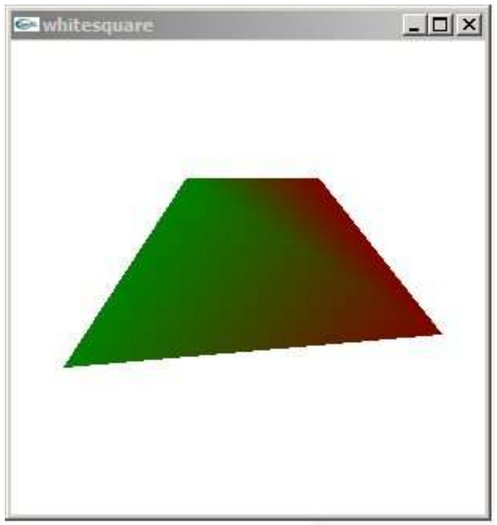
Die GPU als "State-Machine"

- Man versetzt die "Maschine" in einen Zustand, der so lange besteht, bis er wieder verändert wird
- Beispiel: kopiere Geometrie in eine Vertex-Liste (Vertex Buffer)
- Effizienter, als Daten jedes Mal neu zu übergeben



Beispiel zur Betrachtung des Datenflußes

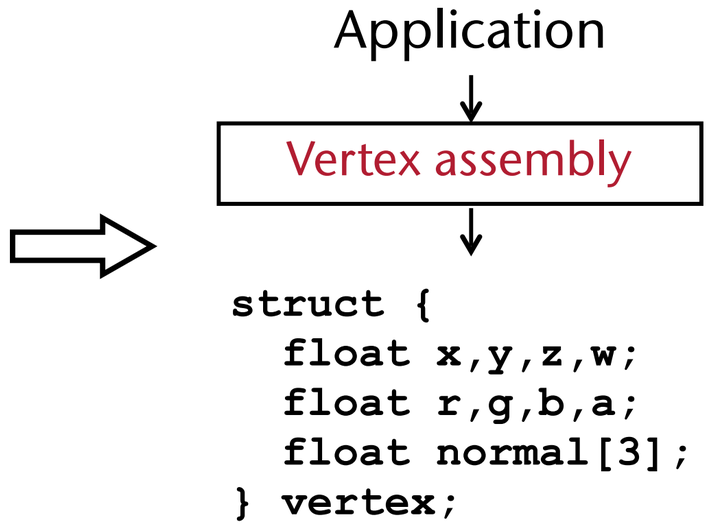
```
glClearColor(1, 1, 1, 1);              // white background
glClear( GL_COLOR_BUFFER_BIT );
glLoadIdentity();                      // ignore it for now
glOrtho(0, 100, 0, 100, -1, 1);        // set projection
glBegin( GL_POLYGON );                 // start primitive
   glColor3f(0, 0.5, 0);               // set color of vertex
   glVertex2f(11.0, 31.0);             // send coords of vertex
   glVertex2f(37.0, 71.0);             // next vertex, same color
   glColor3f(0.5, 0, 0);               // dark red
   glVertex2f(65.0, 71.0);
   glVertex2f(91.0, 38.0);
glEnd();                               // end primitive
glFlush();                             // force completion
```



Der Einfachheit halber hier noch in der alten, fixed-function, immediate mode Pipeline

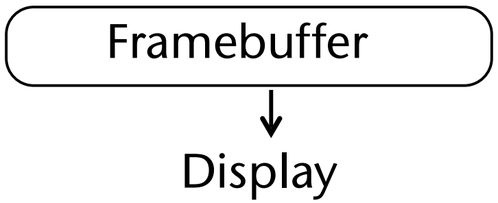
The Pipeline with Regards to Data Flow

- Force input to canonical format
- Convert to internal representation
  - E.g., integer to float
- Initialize unspecified values
  - E.g., z = 0, w=1
- Insert current modal state
  - E.g., color to (0, 0.5, 0, 1)
- In our example:

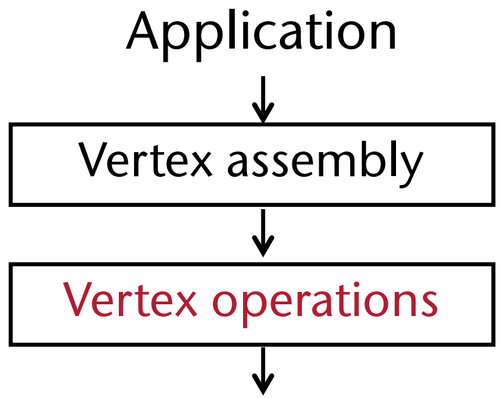


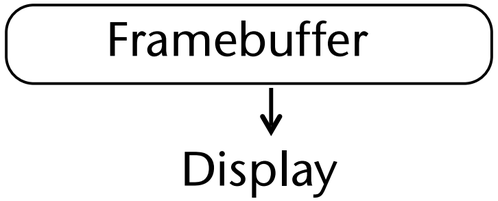

```
vertex v1 = {
  11, 31, 0, 1,   // x,y,z,w
  0, 0.5, 0, 1    // r,g,b,a
};
```

```
vertex v2 = {
  37, 71, 0, 1,   // x,y,z,w
  0, 0.5, 0, 1    // r,g,b,a
};
```

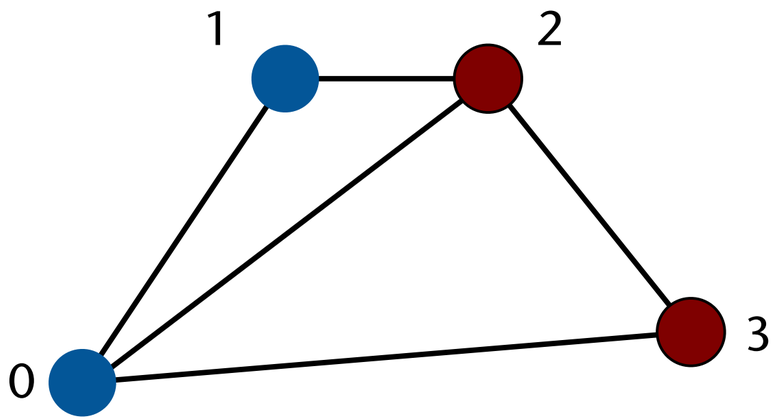


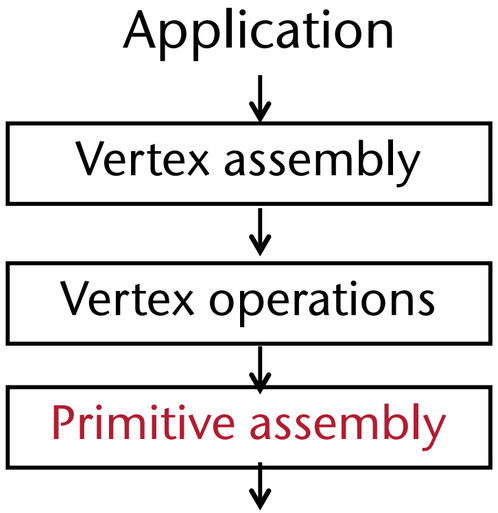
- Transform coordinates
  - Mostly 4x4 matrix arithmetic
- Compute (per-vertex) lighting
- Compute texture coordinates, if procedurally defined





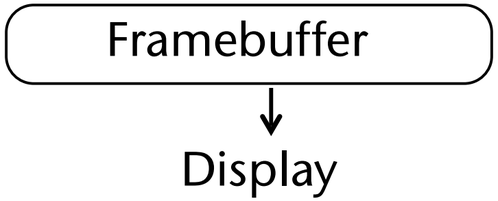
- Group vertices into primitives: points, lines, or triangles
- Old pipeline: decompose polygons to triangles
- In our example:
  - Create two triangles out of the polygon



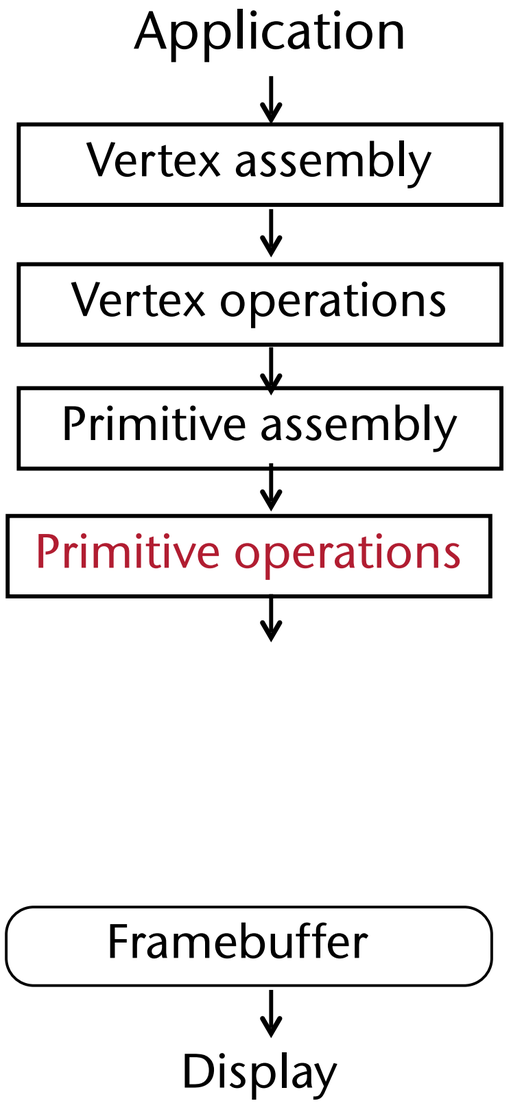



```
struct {
  vertex v0,v1;
} line;
```

```
struct {
  vertex v0,v1,v2;
} triangle;
```

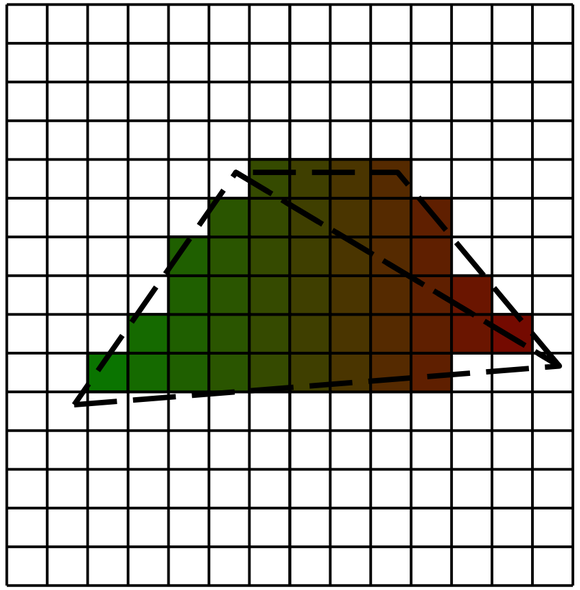


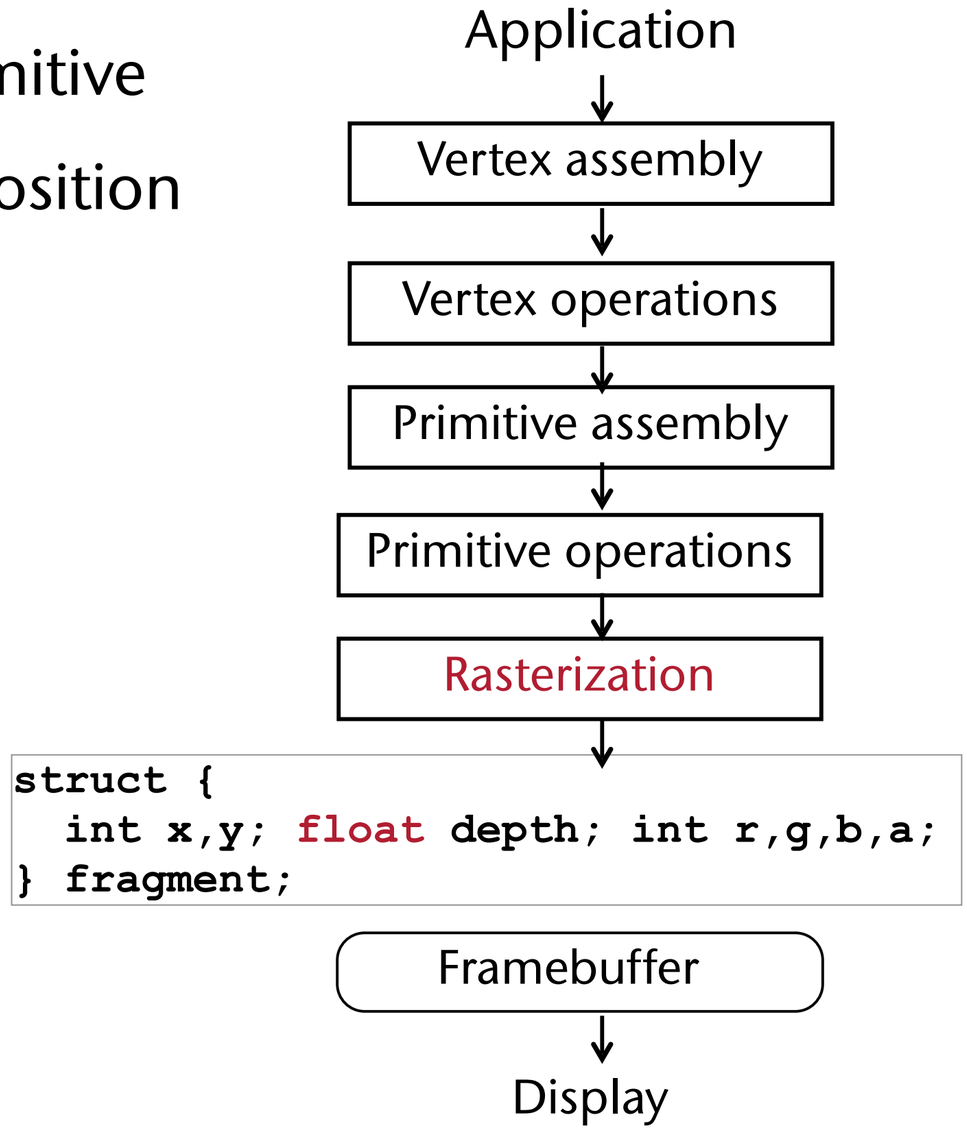
- Clip to the window boundaries
- Perform back-face / front-face operations
  - Backface culling ("Rückseiten" verwerfen)
  - Color assignment for 2-sided lighting
- In our example: nothing happens



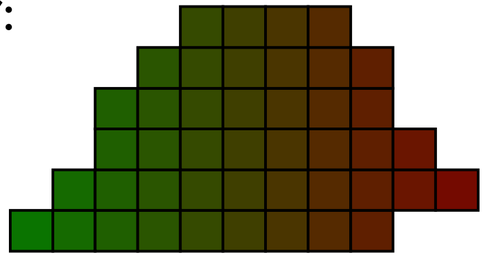
- Determine which pixels are included in the primitive
- Calculate attributes (e.g., color) at each pixel position
- Generate a fragment for each such pixel

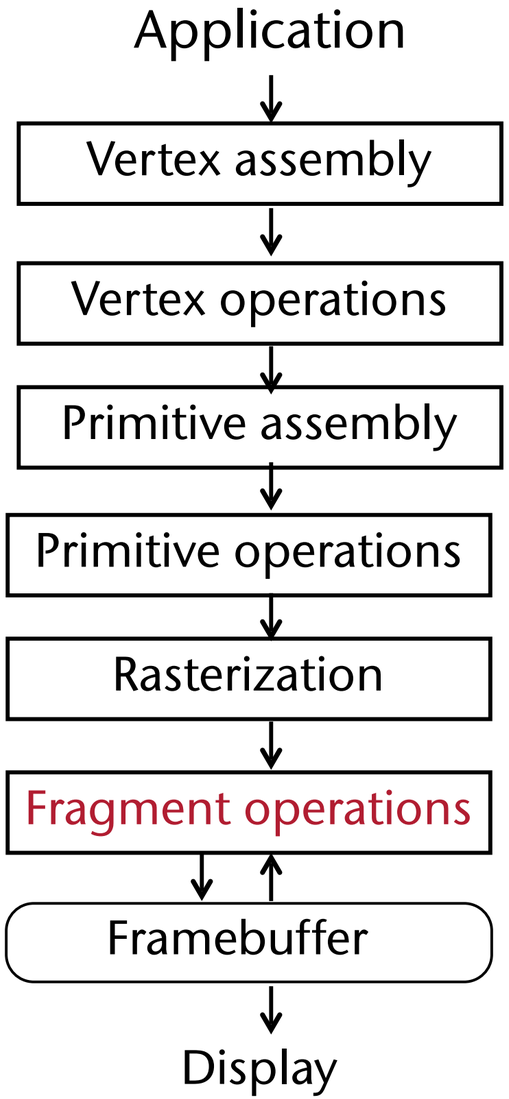
- In our example:



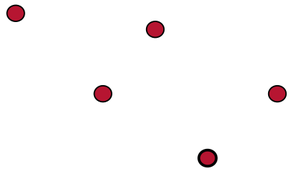


- Depth testing (aka z-buffering)
- Texture mapping
- Fog
- Scissor test
- Alpha test
- Color blending
- In our example: nothing to be done
  - Result in framebuffer:

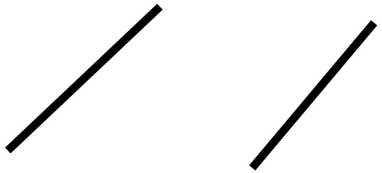




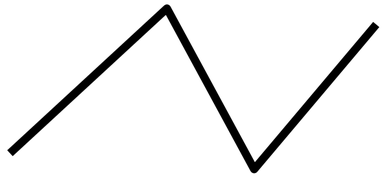
Primitive in OpenGL



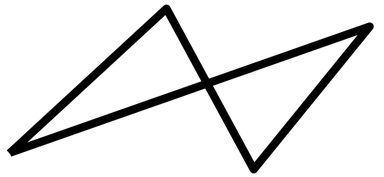
Points



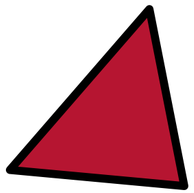
Lines



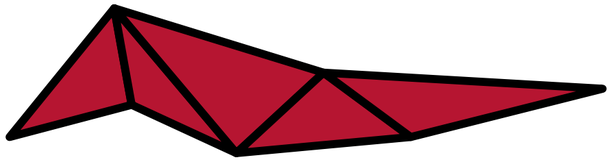
Line-Strip



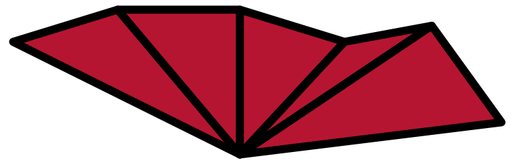
Line-Loop



Dreiecke

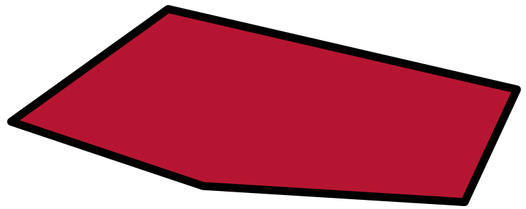


Band aus Dreiecken
(*Triangle Strip*)

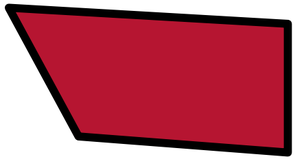


Fächer aus Dreiecken
(*Triangle Fan*)

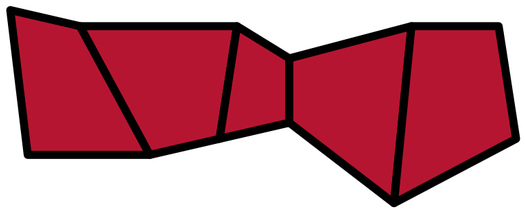
- Nicht mehr verfügbare Primitive (außer im compatibility profile):



Polygone



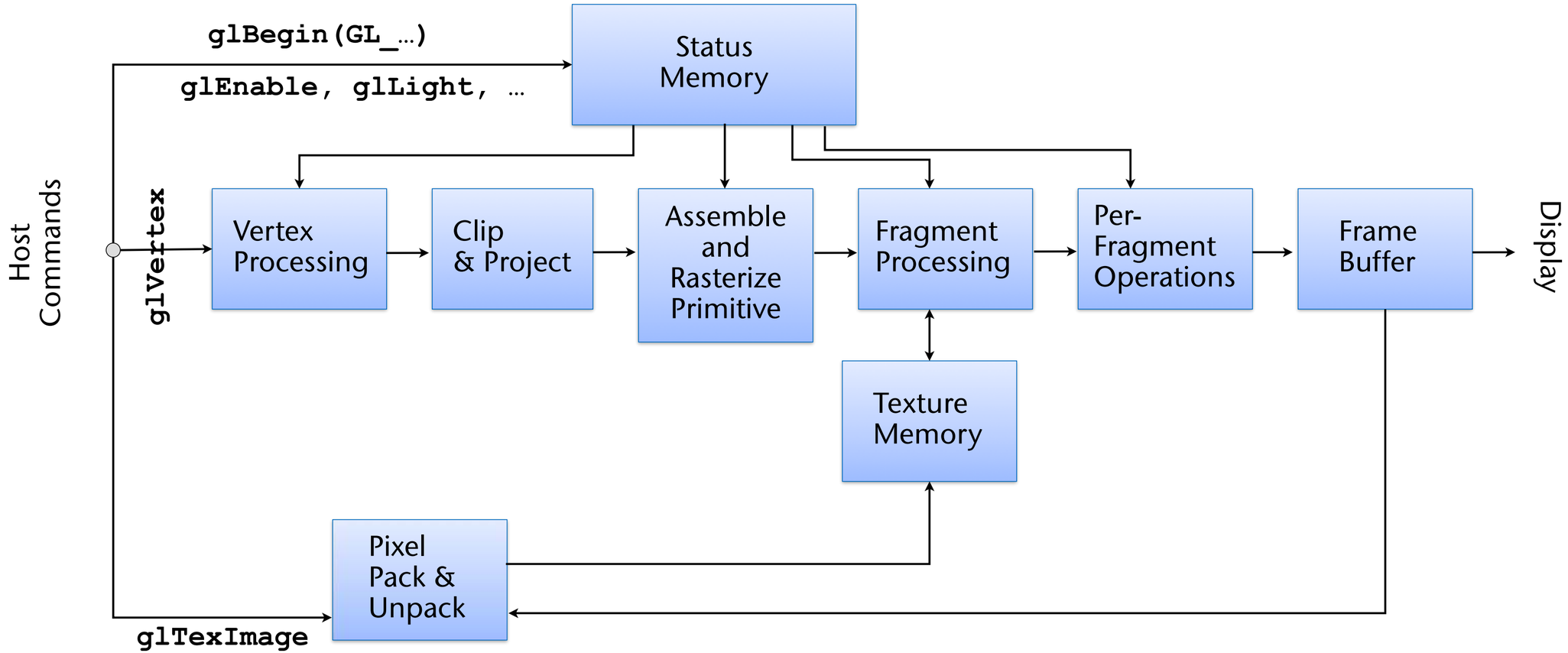
Vierecke
(*Quad* / *Quadrangle*)



Band aus Vierecken
(*Quad Strip*)

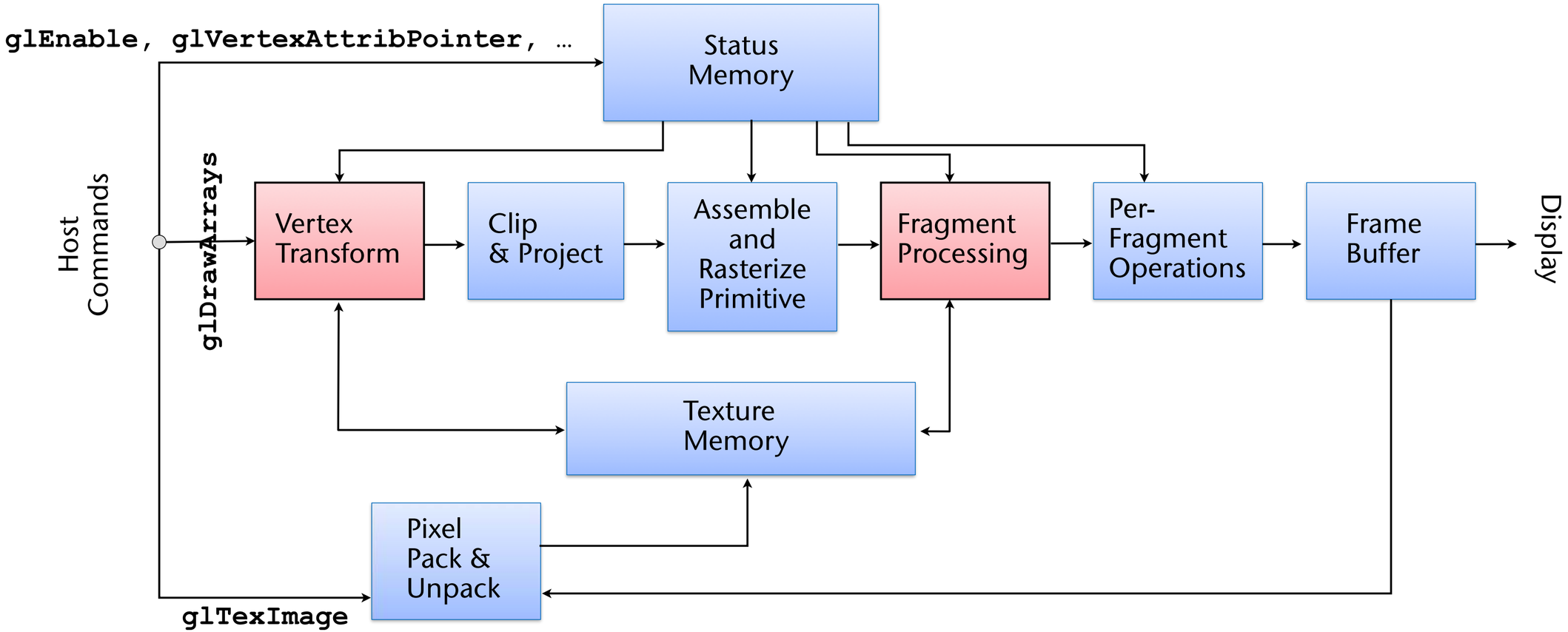
Früher: Fixed-Function Pipeline auf der GPU

- OpenGL 1: Fixed Function Pipeline
  - Sehr sorgfältig ausbalanciert (10 Jahre Erfahrung und Tuning)



Heute: Programmable Pipelines auf der GPU

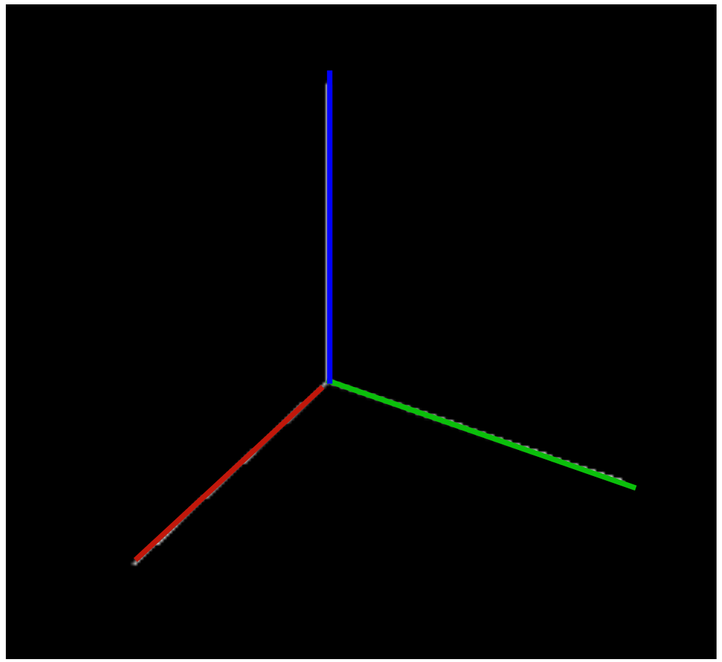
- OpenGL 3+: programmierbare Pipeline (sog. Shaders)
  - Vertex / Fragment Shader: per-vertex / per-fragment Operationen
  - Texturspeicher = allgemeiner Speicher für beliebige Daten



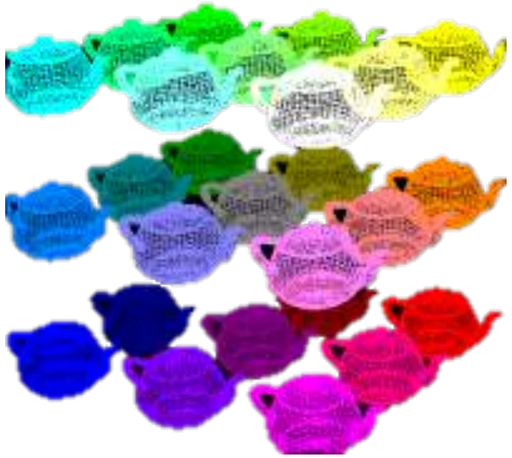
Der Immediate Mode

- Immediate Mode ("direkter Modus"):
  - Primitive werden sofort, wenn sie festgelegt sind, an das Display geschickt (Standard)
  - Die GPU kennt immer nur die Primitive, die aktuell durch die Pipeline laufen
- Weiteres Beispiel dafür:

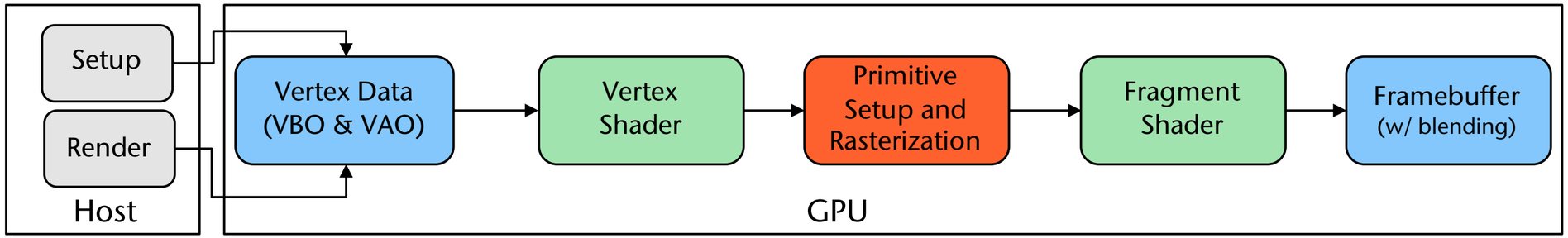
```
glBegin( GL_LINES );
   glColor3f ( 1.0, 0.0, 0.0 );
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 1.0, 0.0, 0.0 );

   glColor3f ( 0.0, 1.0, 0.0 );
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 0.0, 1.0, 0.0 );

   glColor3f ( 0.0, 0.0, 1.0 );
   glVertex3f( 0.0, 0.0, 0.0 );
   glVertex3f( 0.0, 0.0, 1.0 );
glEnd();
```



Der Retained Mode



- Retained Mode ("zurückhaltender Modus"):
  - Primitive werden in sog. Buffer Objects gespeichert
    - Vertex Buffer Objects = VBO, Vertex Array Objects = VAO
  - VBO's können auf der GPU (Graphikkarte) gehalten werden ⟶ Performance
  - Füttern der Pipeline:



  - Kann mehrmals mit unterschiedlichen Eigenschaften gerendert werden
- GPU erhält noch weitere Möglichkeiten zur Optimierung

FYI

- Weitere Features in OpenGL 3:
  - Kontext-Profile:
    - Core: keine Fixed-Function-Pipeline, nur Dreiecke, keine Quads / Polygone, ...
    - Compatibility: rückwärts-kompatibel bis OpenGL 1.0
  - Weitere Shader (Geometry Shader, erst in CG2)

- In OpenGL 4:
  - Noch mehr Shader (Tesselation Shader, Compute Shader)

Der Vertex Shader

- Berechnung der Position im World Space für jeden(!) Vertex (ggf. weitere Berechnungen)
- **`out`**-Variablen stellen die Verbindung zum Fragment Shader her
  - **`gl_Position`** ist eine vordefinierte out-Variable, *muss* gesetzt werden
- Beispiel:

```
#version 330            // OpenGL 3.3 oder höher

uniform mat4 matrix;    // Transformationsmatrix

in vec3 vPosition;      // Position des Vertex
in vec3 vColor;         // Farbe des Vertex

out vec3 fColor;        // ausgehende Farbe des V.

void main()
{
  fColor = vColor;
  gl_Position = matrix * vec4(vPosition, 1.0);
}
```

Schnittstelle zur Host-Seite
Uniforms:
matrix
Attributes:
vPosition
vColor

Der Fragment Shader

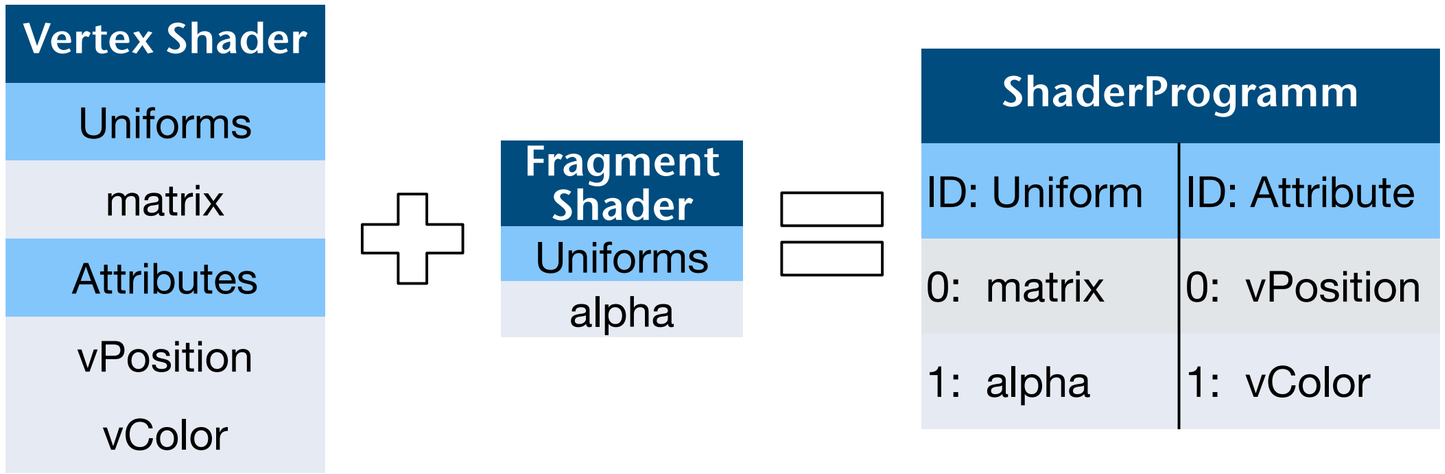
- Berechnet letzten Endes die Farbe für jedes ausgegebene Fragment
- Beispiel:

```
#version 330

uniform float alpha;

// vom Vertex Shader / Rasterizer
in vec3 fColor;

// einzige Ausgabe: die Farbe des Fragments
out vec4 color;

void main()
{
    color = vec4( fColor, alpha );
}
```

Schnittstelle zur Host-Seite
Uniforms:
alpha

Das Shader-Programm

- Shader-Programm = Vertex-Shader + Fragment-Shader (compiled & linked)



- Zugriff auf die Shader-Variablen:

```
// ID's / Handles zu den jeweiligen Variablen/Attributen
GLint matrixUni = glGetUniformLocation(shprog, "matrix");         // 0
GLint alphaUni  = glGetUniformLocation(shprog, "alpha");          // 1
GLint vPositionAttr = glGetAttribLocation(shprog, "vPosition"); // 0
GLint vColorAttr = glGetAttribLocation(shprog, "vColor");         // 1
GLint error = glGetAttribLocation(shprog, "unknown");             // -1
```

- Achtung:
  - Vertex-Attribute können vom Host nur dem Vertex Shader übergeben werden
  - Unbenutzte Variablen werden wegoptimiert

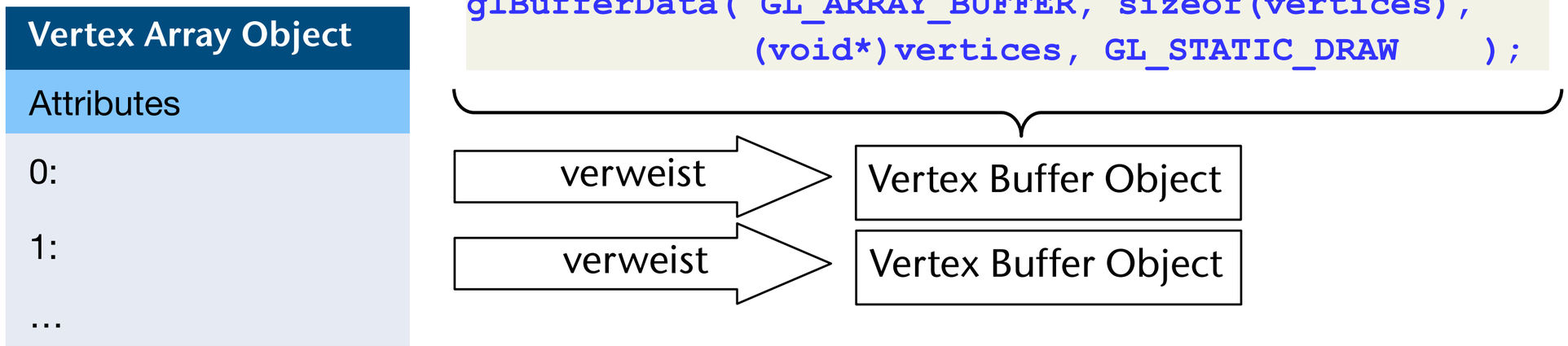
Prinzipielle Vorgehensweise des Renderns (CPU)

1. Shader kompilieren und zu Shader-Programm linken
2. IDs der Attribute und Uniforms aus dem Shaderprogramm auslesen
3. Buffer erzeugen
   - Vertex Buffer Object (VBO): für Vertex-Attribute (Position, Farbe, Normale, Textur, ...)
   - Layout der Buffer bekannt geben: welches Attribut steht an welcher Stelle im Array ⟶ "Verbindung" zwischen Attributen und Shader-Variablen (IDs)
   - Vertex Array Object (VAO) erzeugen: verweist auf die VBO's
4. Buffer mit Daten füllen
5. Konkrete Werte an die Uniforms zuweisen
6. Buffer Object rendern (mit aktivem Shader-Programm & VAO)

- Während eines Frames können Shader, VAOs, VBOs, Uniforms geändert werden

Generieren der Buffers

2.
```
// generate vertex buffer object on the GPU
glGenBuffers( 1, &m_vertices_buffer );
// set it as the active VBO (think "state")
glBindBuffer( GL_ARRAY_BUFFER, m_vertices_buffer );
// copy actual vertex coords into the VBO
glBufferData( GL_ARRAY_BUFFER, sizeof(vertices),
              (void*)vertices, GL_STATIC_DRAW    );
```

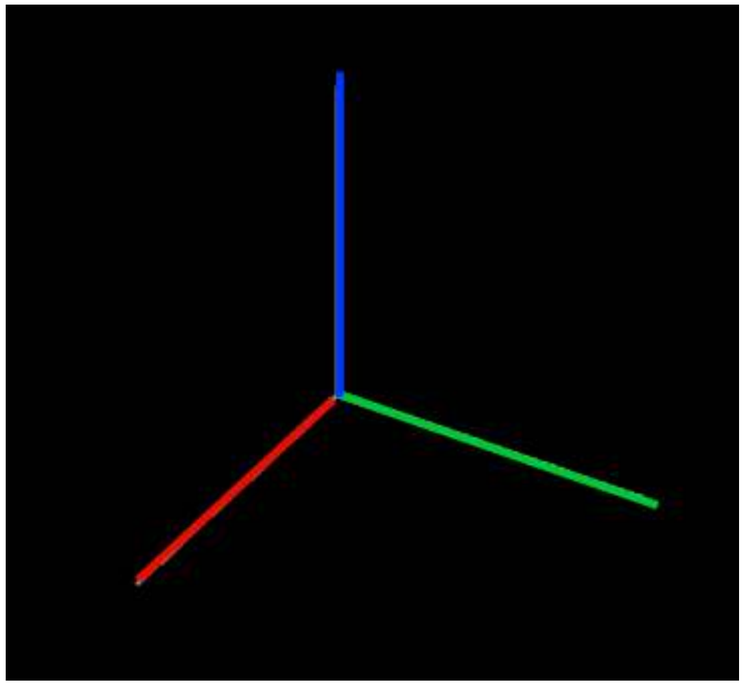


1.
```
GLuint my_obj;                      // Handle auf neues VAO
glGenVertexArrays( 1, & my_obj ); // erstelle 1 Vertex Array Object
glBindVertexArray( my_obj );        // setze my_obj als aktives VAO
```

Achtung: für alle Code-Beispiele im Folgenden nehmen wir an,
dass `glBindVertexArray(my_obj)` noch aktiv ist!

Beispiel (mit einem VBO pro Attribut)

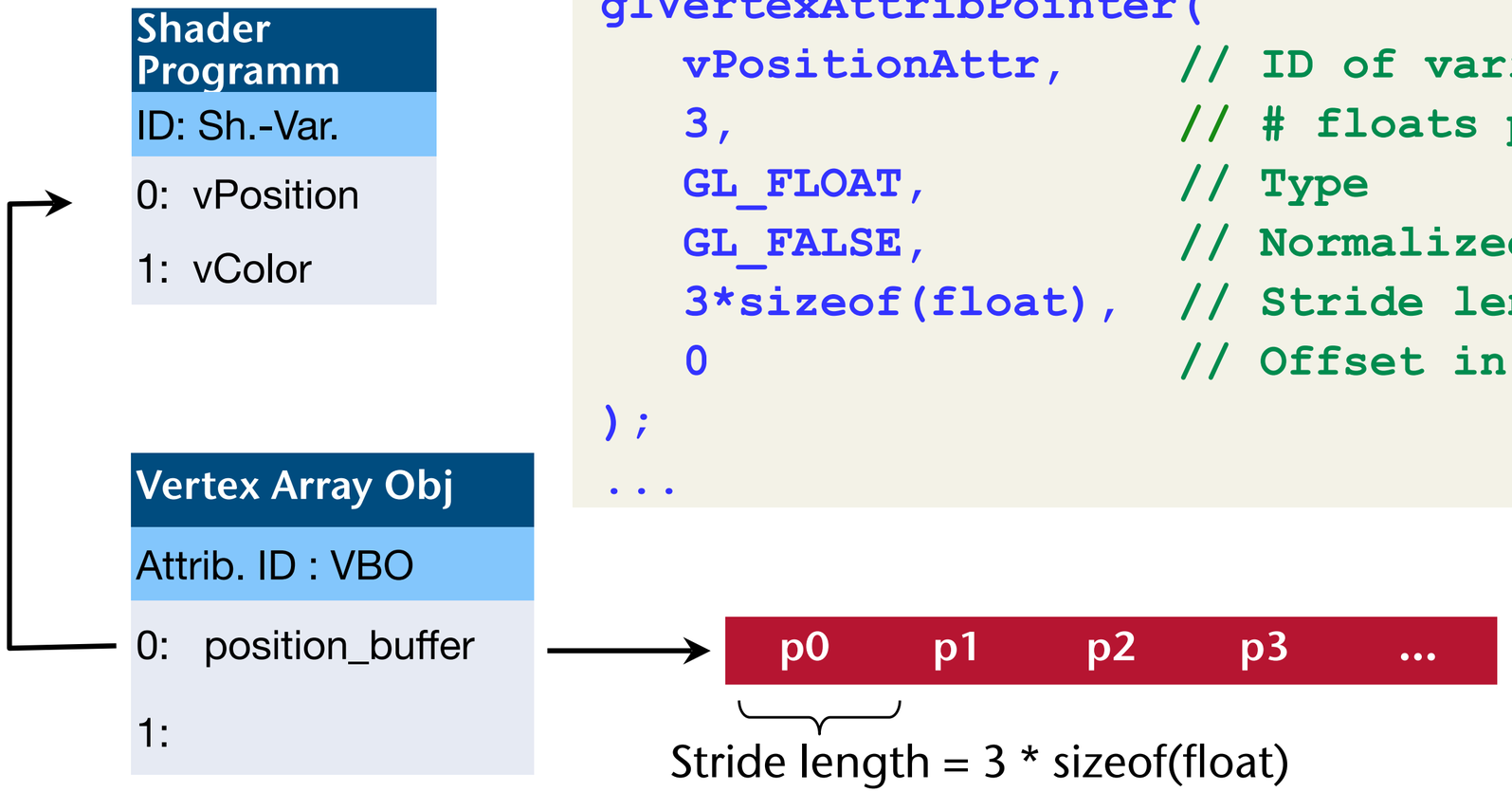
```
// Vertex positions (XYZ)
float * position = {
    0.0, 0.0, 0.0,  // Ursprung
    1.0, 0.0, 0.0,  // x-Achse
    0.0, 0.0, 0.0,  // Ursprung
    0.0, 1.0, 0.0,  // y-Achse
    0.0, 0.0, 0.0,  // Ursprung
    0.0, 0.0, 1.0}; // z-Achse
// Vertex colors (RGB)
float * color = {
    1.0, 0.0, 0.0,  // rot
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,  // grün
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,  // blau
    0.0, 0.0, 1.0};
```



```
glGenBuffer( 1, &position_buffer );                          // create(empty) VBO
glBindBuffer(GL_ARRAY_BUFFER, position_buffer);              // use it
glBufferData(GL_ARRAY_BUFFER, sizeof(position), position, GL_STATIC_DRAW);
.....
glGenBuffer( 1, &color_buffer );                             // again for colors
glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(color), color, GL_STATIC_DRAW);
.....
```

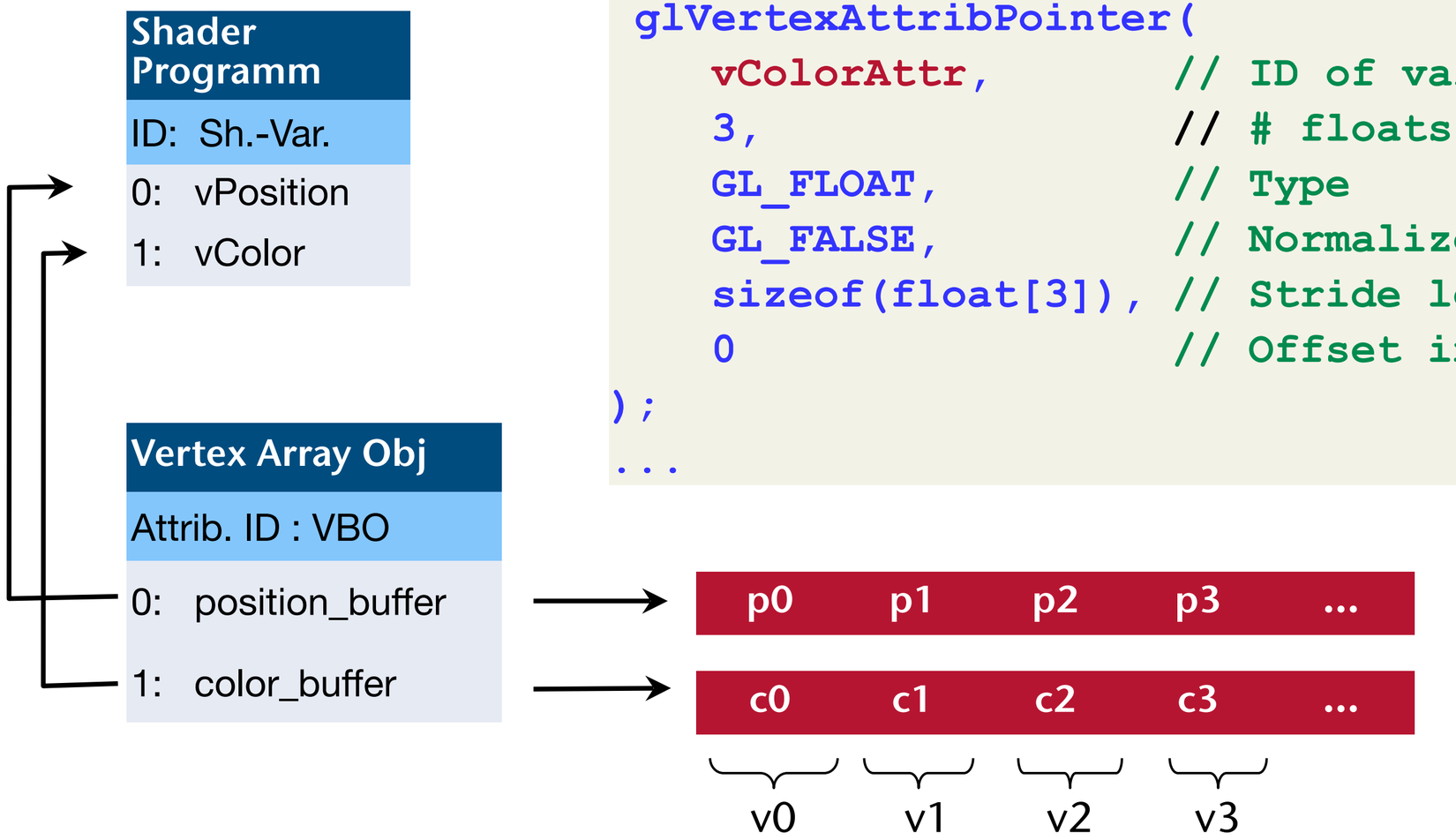
Verbindung zwischen VBO und Shader-Variablen herstellen

```
glBindBuffer( GL_ARRAY_BUFFER, position_buffer ); // prev.
glBufferData( ... );                              // slide
glEnableVertexAttribArray( vPositionAttr ); // s.S. 34 & 36
glVertexAttribPointer(
   vPositionAttr,     // ID of variable in shader prog
   3,                 // # floats per attrib.
   GL_FLOAT,          // Type
   GL_FALSE,          // Normalized?
   3*sizeof(float),   // Stride length
   0                  // Offset in VBO (in bytes)
);
...
```



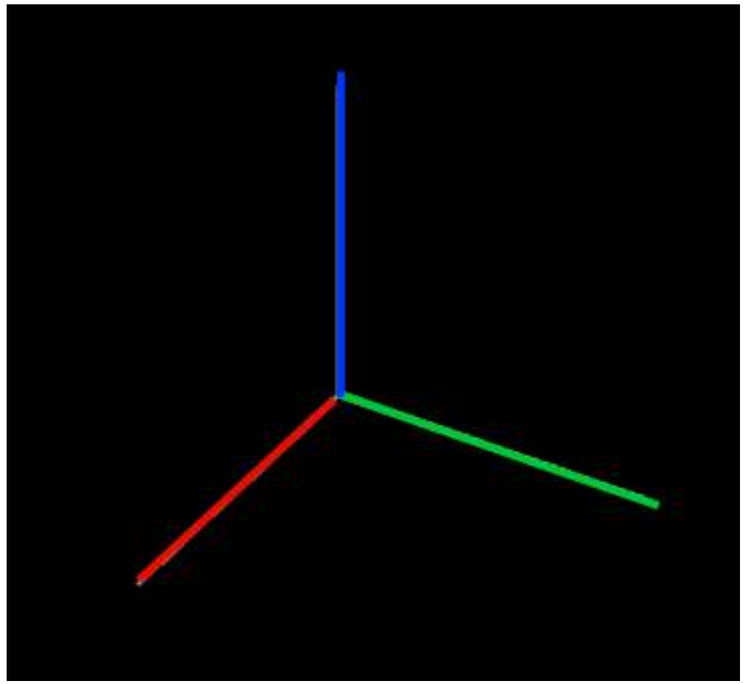
Verbindung zwischen VBO und Shader-Variablen herstellen

```
glBindBuffer( GL_ARRAY_BUFFER, color_buffer );
glBufferData( ... );
glEnableVertexAttribArray( vColorAttr );  // s.S. 34
glVertexAttribPointer(
    vColorAttr,          // ID of variable in shader prog
    3,                   // # floats per attrib.
    GL_FLOAT,            // Type
    GL_FALSE,            // Normalized?
    sizeof(float[3]),    // Stride length
    0                    // Offset in VBO (in bytes)
);
...
```



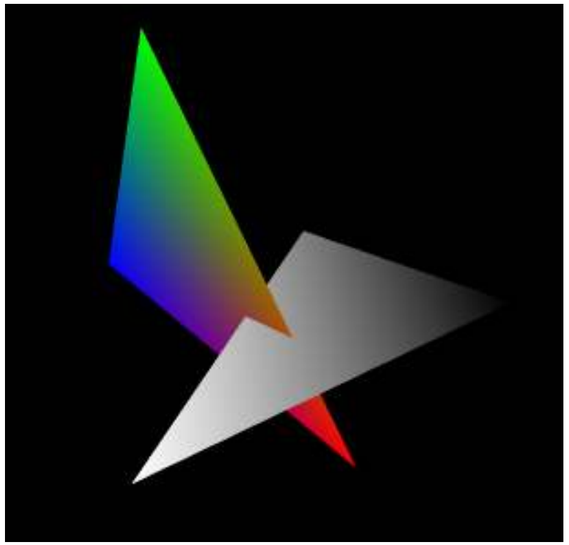
Rendering (at Runtime)

```
glBindVertexArray( my_obj );
glDrawArrays(
    GL_LINES, // render mode
    0,        // offset in bytes
    3 * 2     // # vertices
);
```



Beispiel mit interleaved Vertex Layout (nur *ein* VBO)

```
struct VertexData
{
    float position[3];
    float color[3];
};

// Vertex Positionen (XYZ, RGB)
VertexData * vertices =
{
  {{0.4, -0.2, 0.2}, {1.0, 0.0, 0.0}}, // v0, rot
  {{-0.1, 0.8, 0.3}, {0.0, 1.0, 0.0}}, // v1, grün
  {{-0.2, 0.3, 0.1}, {0.0, 0.0, 1.0}}, // v2, blau
  {{-0.1, -0.2, 0.5},{1.0, 1.0, 1.0}}, // v3, weiß
  {{0.8, 0.2, 0.1},  {0.0, 0.0, 0.0}}, // v4, schwarz
  {{0.3, 0.4, 0.0},  {0.5, 0.5, 0.5}}  // v5, grau
};
```

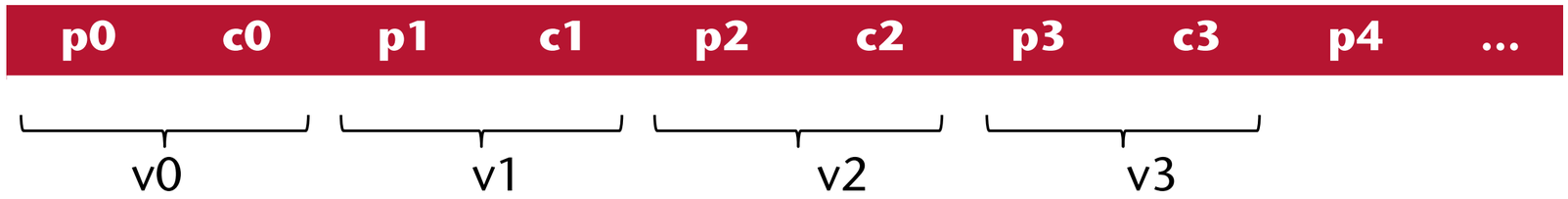


VBO erzeugen und füllen

```
VertexData * vertices =
{
    {{0.4, -0.2, 0.2}, {1.0, 0.0, 0.0}},
    {{-0.1, 0.8, 0.3}, {0.0, 1.0, 0.0}},
  [...]
};
```

```
struct VertexData
{
    float position[3];
    float color[3];
};
```

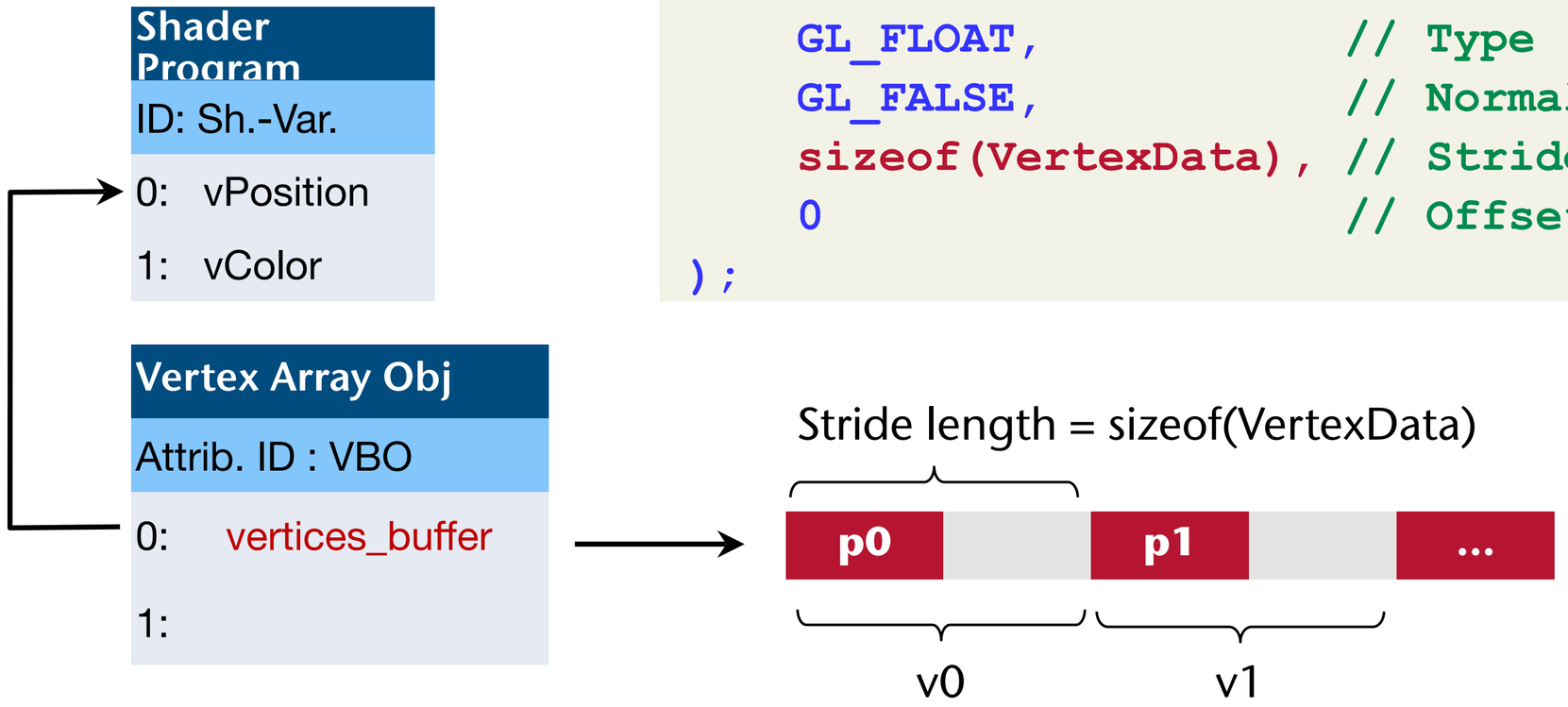
(Erinnerung)




```
glGenBuffer( 1, &vertices_buffer );
glBindBuffer(GL_ARRAY_BUFFER, vertices_buffer );
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

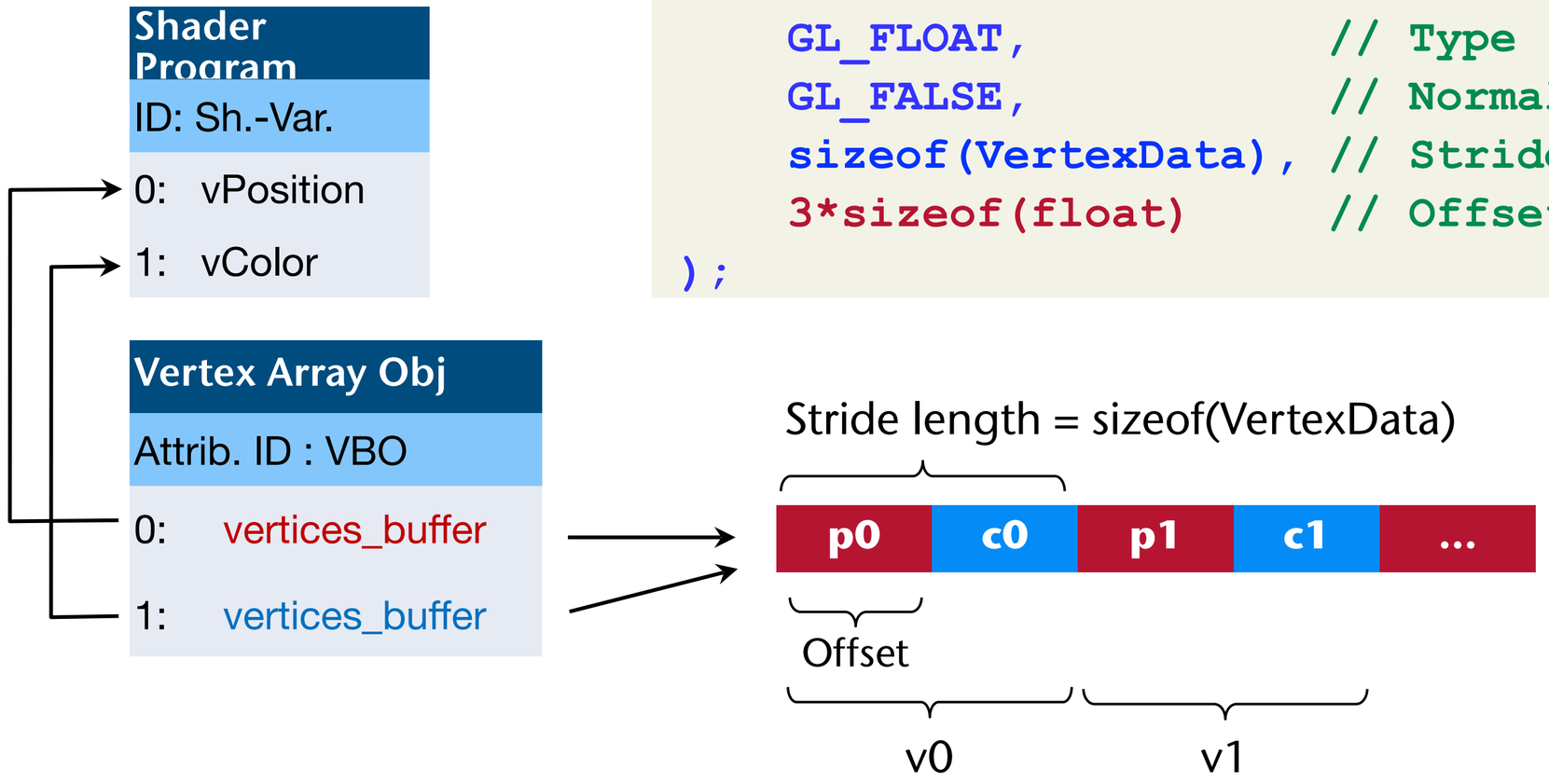
Verbindungen herstellen

```
glEnableVertexAttribArray( vPositionAttr );
glVertexAttribPointer(
    vPositionAttr,
    3,                  // # floats per attrib.
    GL_FLOAT,           // Type
    GL_FALSE,           // Normalized?
    sizeof(VertexData), // Stride length
    0                   // Offset in the VBO
);
```



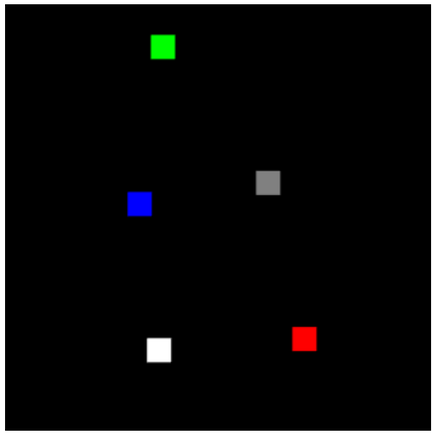
Verbindungen herstellen

```
glEnableVertexAttribArray( vColorAttr );
glVertexAttribPointer(
    vColorAttr,
    3,                  // # floats per attrib.
    GL_FLOAT,           // Type
    GL_FALSE,           // Normalized?
    sizeof(VertexData), // Stride length
    3*sizeof(float)     // Offset in the VBO
);
```

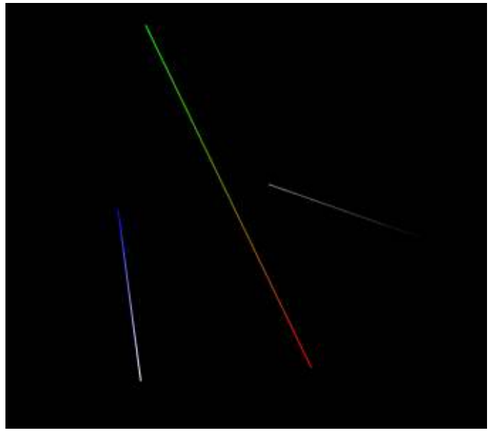


Rendering

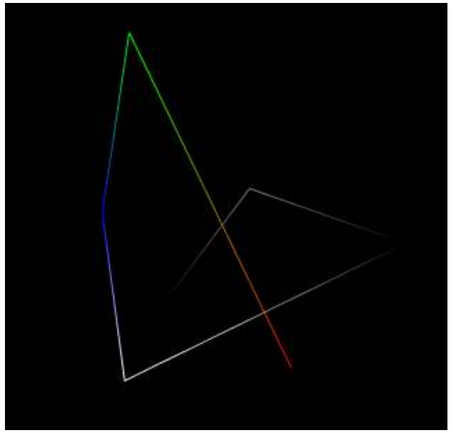
```
glDrawArrays( prim_type, 0, 3*2 );
```
(Type des Primitivs kann zur Laufzeit festgelegt werden)



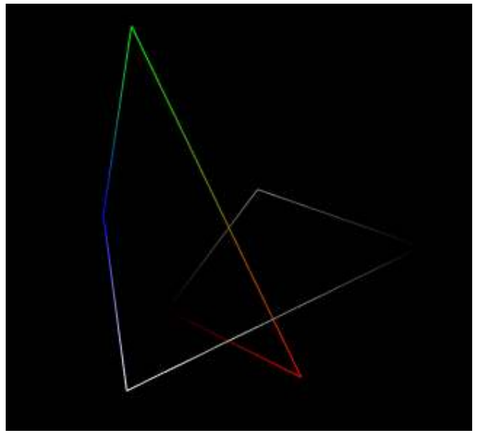
GL_POINTS



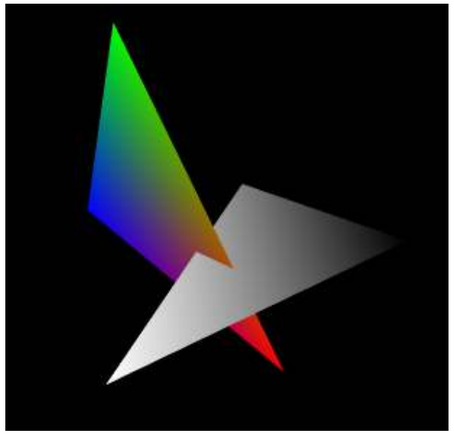
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



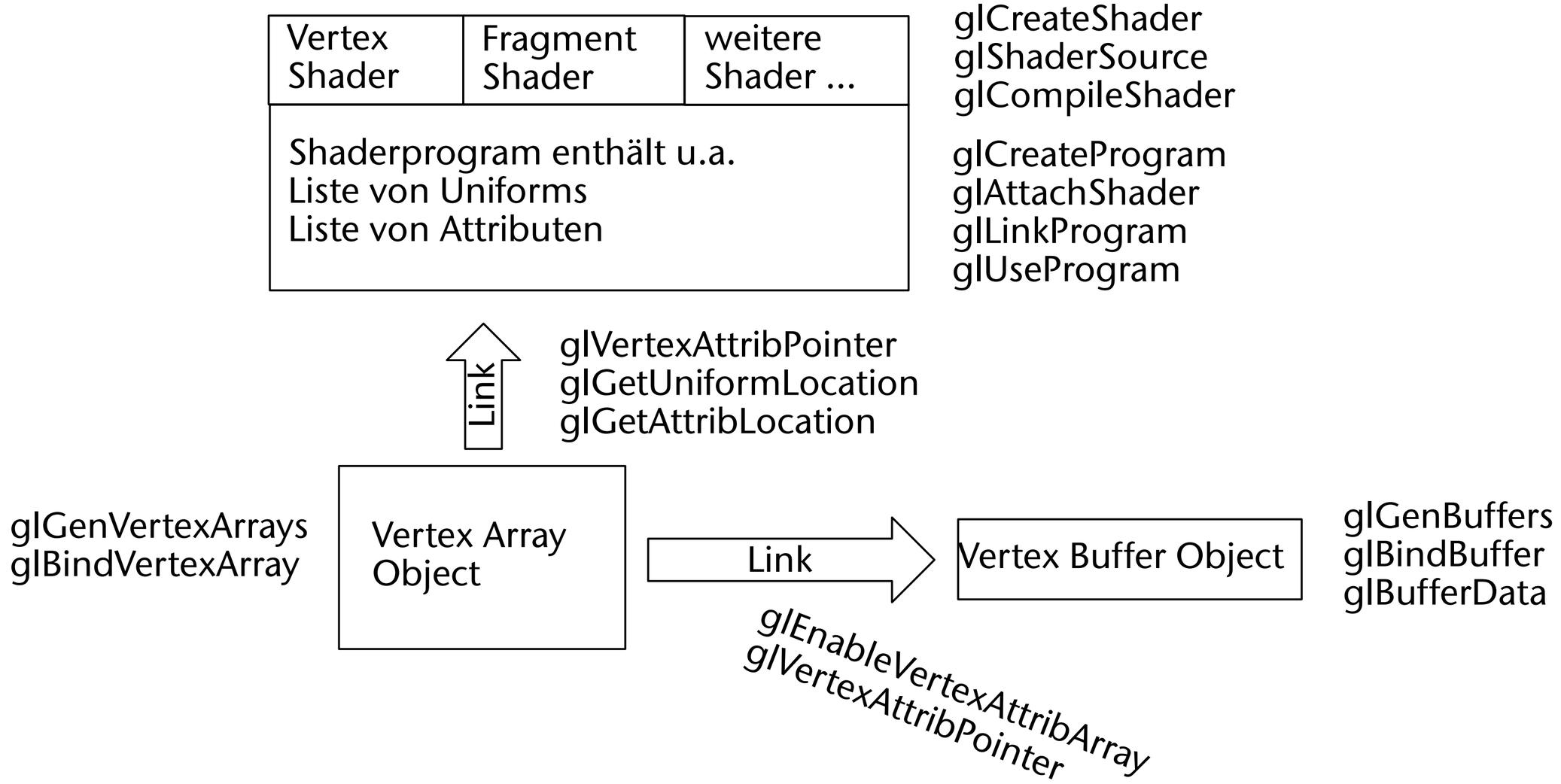
GL_TRIANGLES

Bemerkung

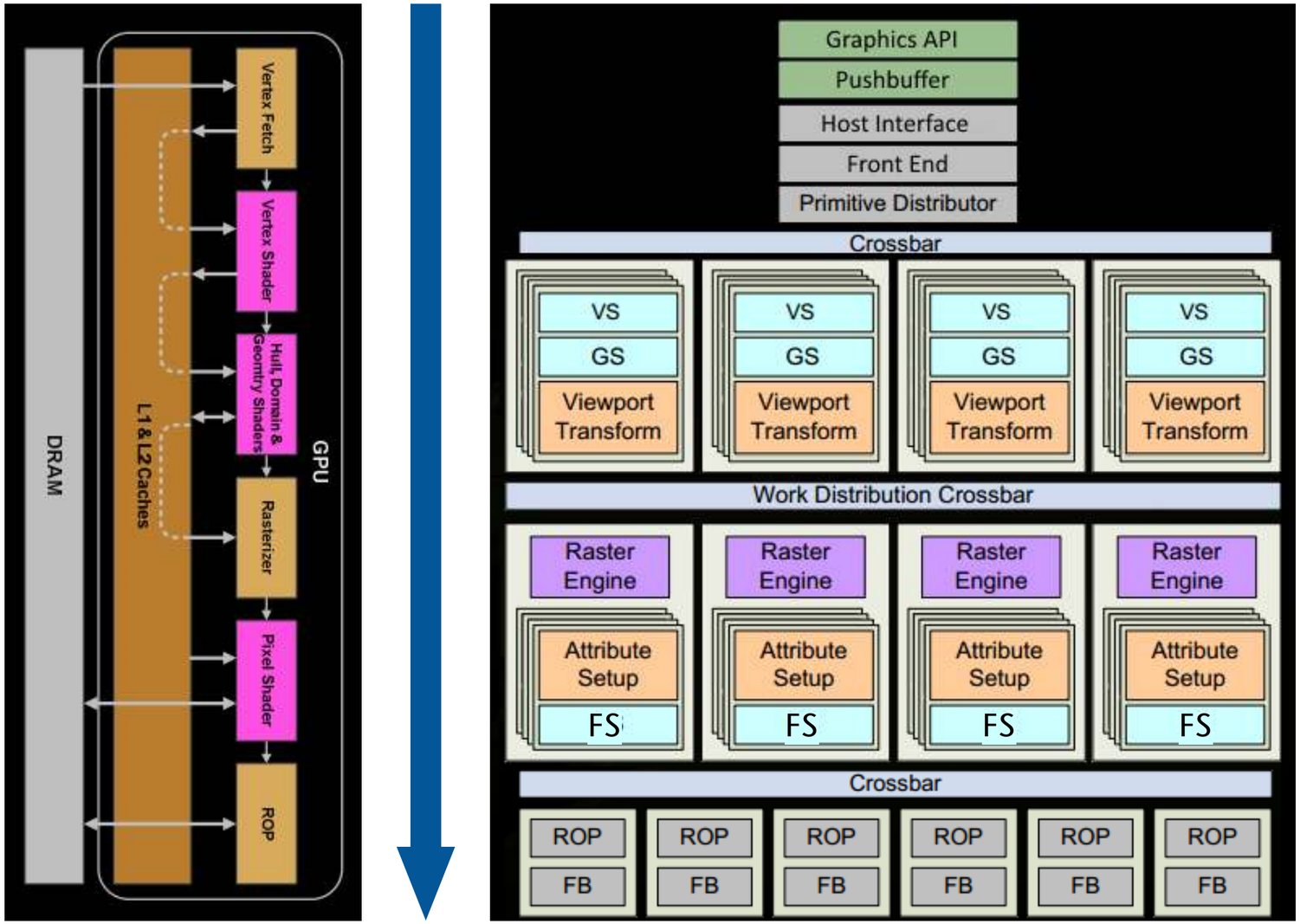
- Manchmal könnte es praktisch sein, eine eigene "VAO-Klasse" zu schreiben, die ein Interface ähnlich zum *immediate mode* hat:

```
// Initialisierung
VertexArrayObject vao = new VertexArrayObject( num_vertices );
 vao->begin( GL_TRIANGLES );
 vao->addVertex(  0.0,  0.0,  0.0 );
 vao->addVertex(  0.5,  0.75, 0.0 );
 vao->addVertex( -0.5, -0.25, 0.0 );
 ...
 vao->end();

 // display
 vao->draw();
```

OpenGL Cheat Sheet zu VAO's & VBO's



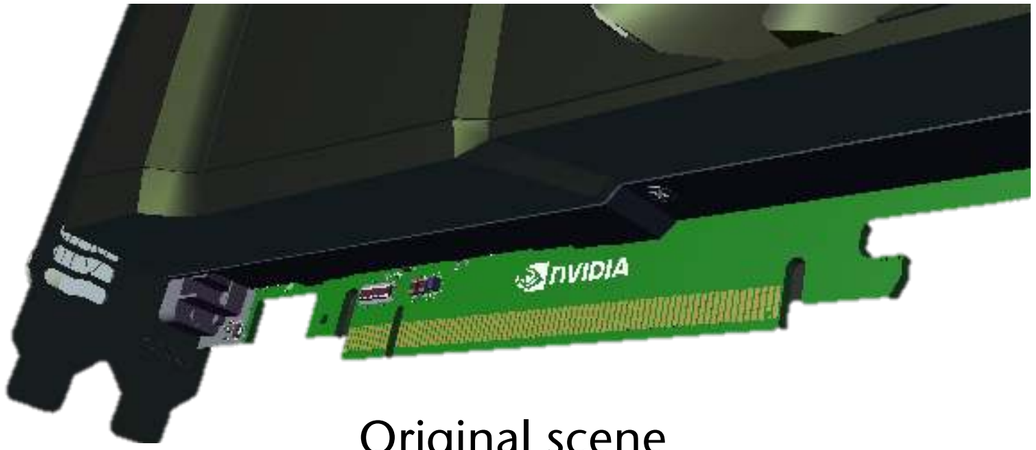
A Hardware View of the Pipeline



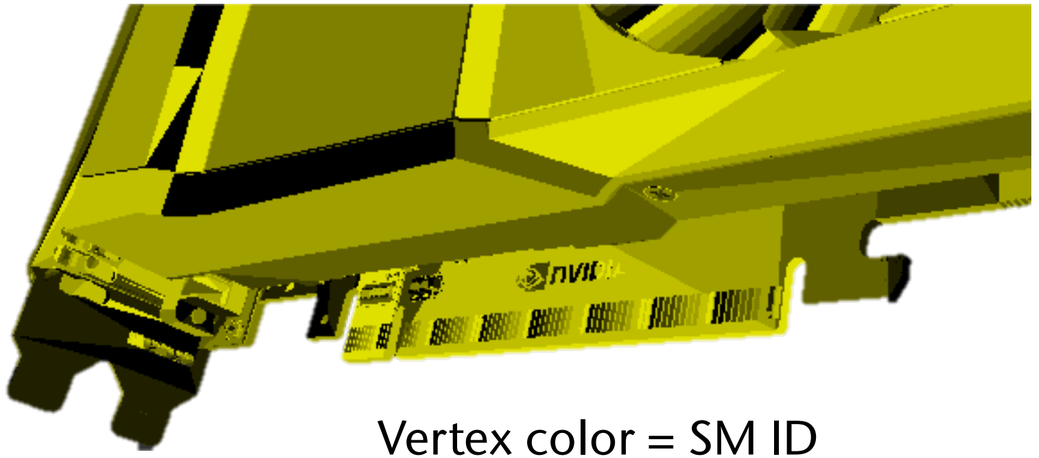
GPU Architecture

Streaming Multiprocessor (SM)

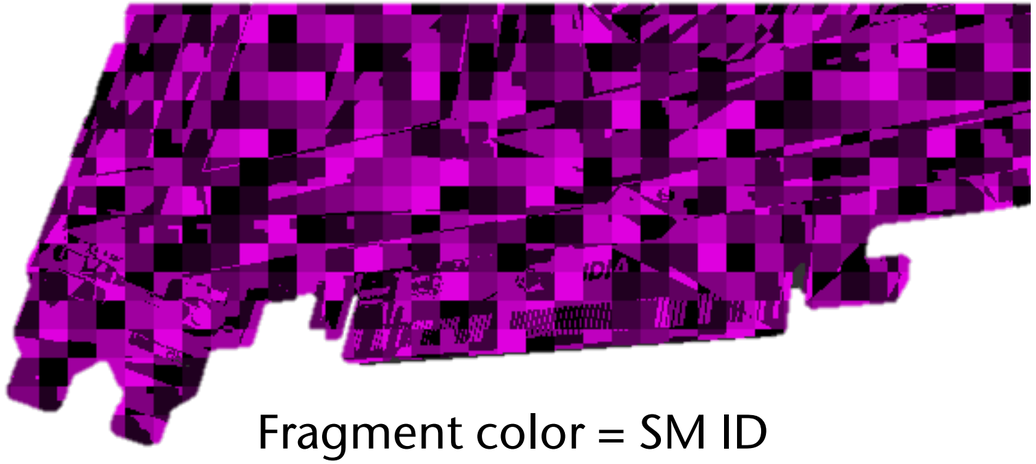
How the GPU Distributes Work



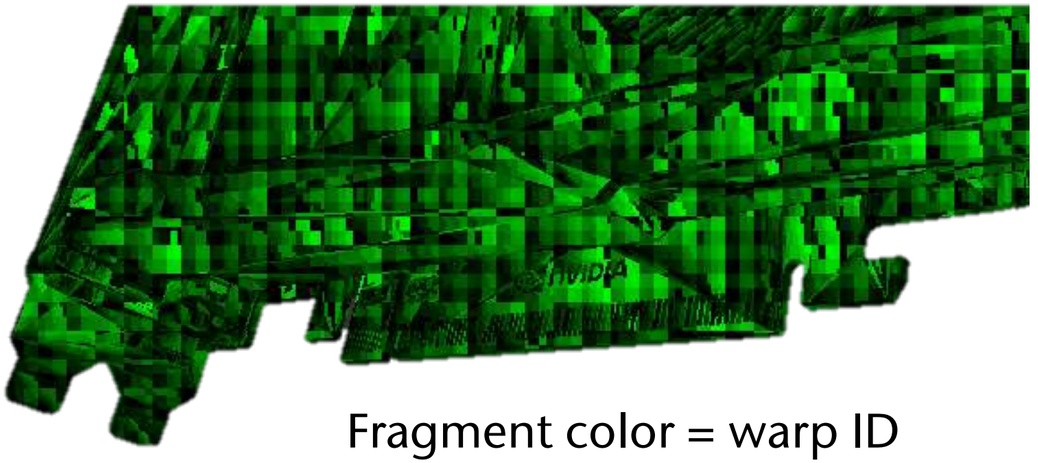
Original scene



Vertex color = SM ID



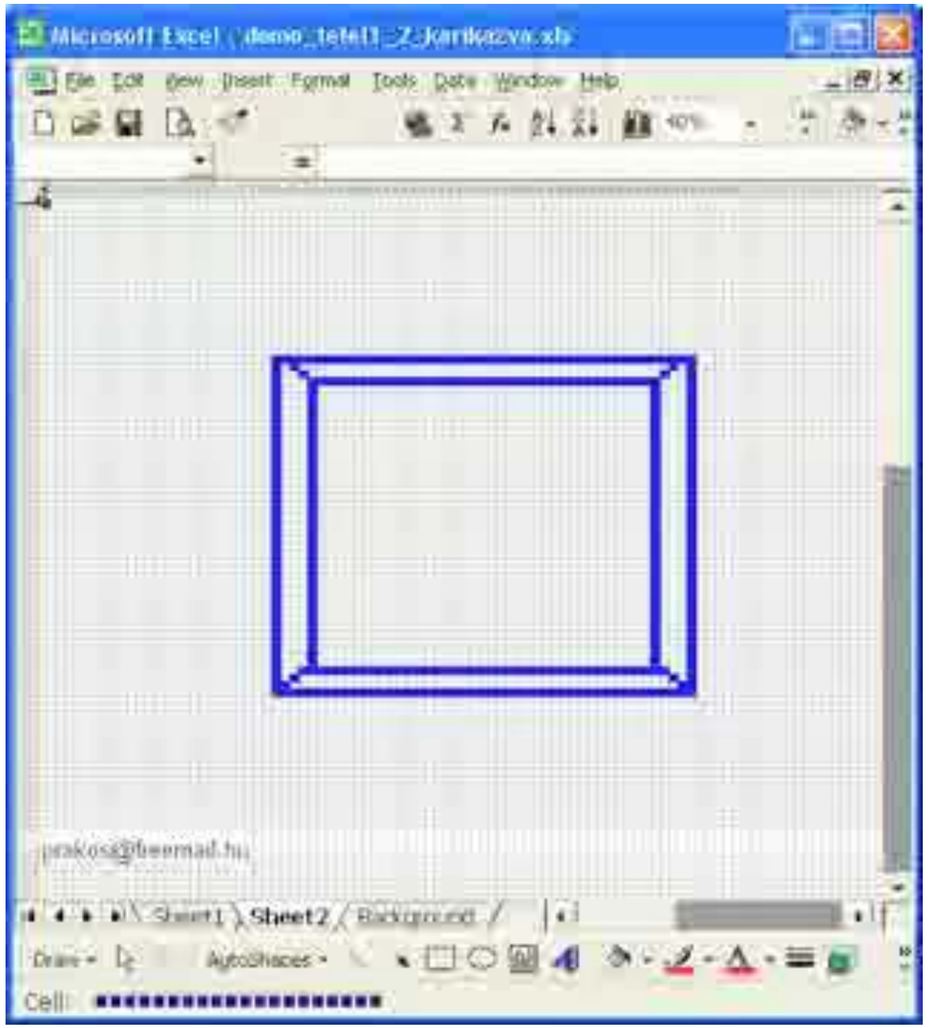
Fragment color = SM ID

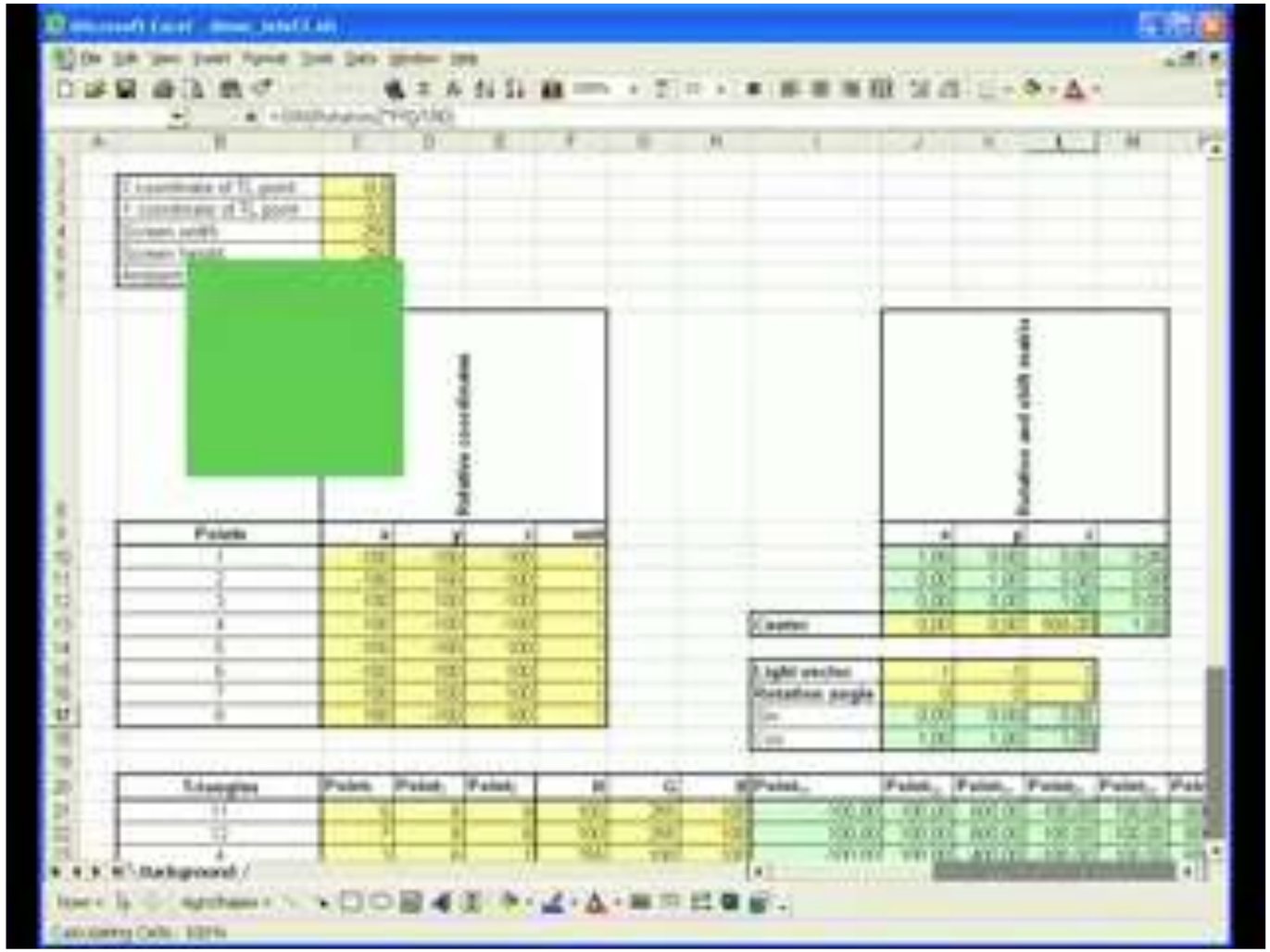


Fragment color = warp ID

Christoph Kubisch: Life of a triangle

Microsoft Excel: Revolutionary 3D Game Engine? ☺





http://cgvr.cs.uni-bremen.de/teaching/cg_literatur/excel_3d_engine/