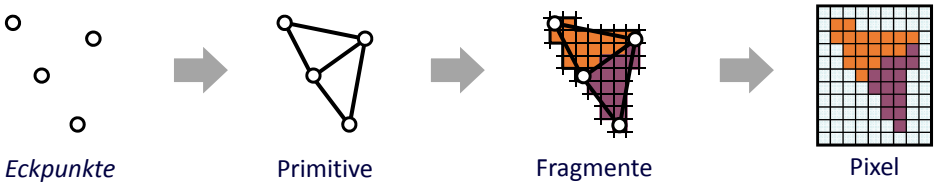
Computergrafik I
Kapitel 3: Die Rendering Pipeline

Wintersemester 2014/2015

Prof. Dr. Timo Ropinski
Forschungsgruppe Visual Computing

Bild-basiert vs. Geometrie-basiert

- Bisher: Bilderzeugung durch Ray Tracing
- Ab jetzt: Bilderzeugung durch Rasterisierung
  - Szenenobjekte (Primitive) gegeben als Menge von Eckpunkten (*Vertices*, singular Vertex)
  - Primitive durchlaufen verschiedene Koordinatensysteme
  - Primitive werden rasterisiert zu Fragmenten (=Pixelvorläufer)
  - Eine Untermenge der Fragmente wird zu Pixeln

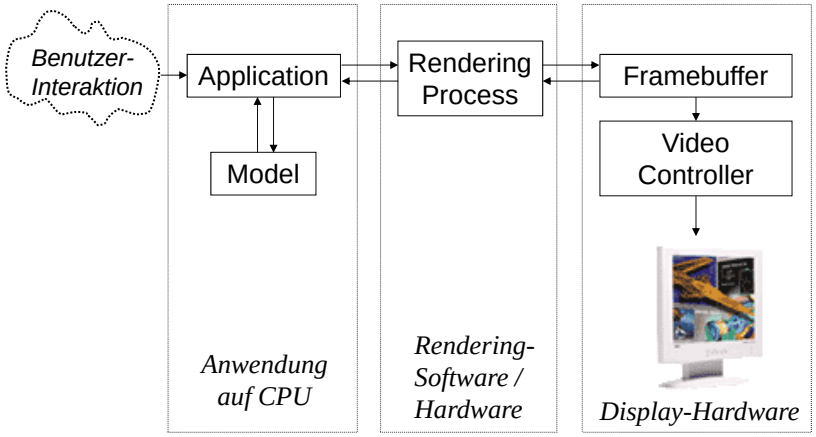


Kapitelstruktur

3.1 Der Rendering Prozess

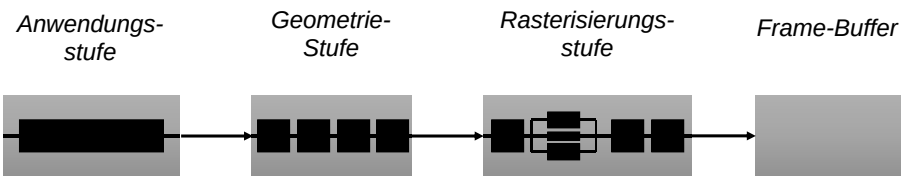
Vom geometrischen Primitiv über das Fragment hin zum Pixel

Geometrie-basiertes Rendering 1/2



Geometrie-basiertes Rendering 2/2

- Der Rendering Prozess wird durch die Rendering Pipeline realisiert
  - Verarbeitung von 3D Modellen zu 2D Bildern
- Konzeptionell besteht die Rendering Pipeline aus drei Stufen
  - Anwendungsstufe
  - Geometriestufe
  - Rasterisierungsstufe

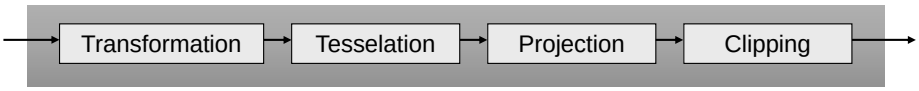


Anwendungsstufe

- Spezifikation der Modellelemente und der Szene
  - Szenenmodellierung: Anordnung und Attributierung der Szenenobjekte
- Interaktionsparadigmen
- Berechnung von Animationspfaden
- Kollisionserkennung (*collision detection*)
- Selektion der sichtbaren Objekte (*occlusion culling*)

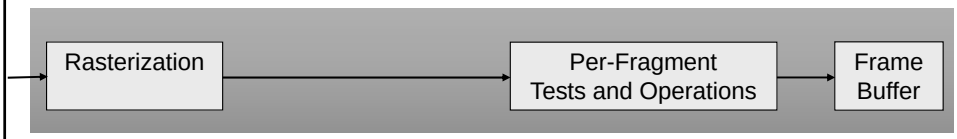
Geometriestufe

- Geometrische 3D-Transformation (*Transformation*)
  - Ausrichtung einzelner Szenenobjekte in der Gesamtszene
  - Grundtransformationen: Rotation, Skalierung, Verschiebung
- *Primitive Assembly* und Tesselierung (*Tesselation*)
  - Zusammensetzung der Eckpunkte zu Primitiven
  - Erzeugung zusätzlicher Dreiecke mittels Tesselierung
- Geometrische Projektion (*Projection*)
  - 3D nach 2D
  - z.B. perspektivische Projektion oder orthogonale Projektion
- Clipping
  - Entfernung nicht in der Zeichenfläche liegender Primitive



Rasterisierungsstufe

- Konvertierung der Primitive in Fragmente (*Rasterization*)
  - Diskretisierung erfolgt in Bezug auf ein Zielraster
  - Rasterisierungsalgorithmen für bestimmte Primitive optimiert (z.B. Linien, Dreiecke)
- Fragment Tests
  - Fragmente durchlaufen verschiedene Tests (z.B. Sichtbarkeit)
  - Fragmente werden - sofern entschieden wird sie zu nutzen - in den Framebuffer kopiert
    - Können dabei mit den aktuellen Pixelwerten geblendet werden



3.2 OpenGL Übersicht

Die Standard Bibliothek für Echtzeit-Grafik

OpenGL

- OpenGL (*Open Graphics Library*)
  - Plattform-unabhängiges 2D und 3D Grafik API
  - API Spezifikation erfolgt furch die Khronos Group
- Technische Umsetzung
  - Rendering Pipeline bestehend aus statischen und programmierbaren Pipeline Stufen
  - Typischerweise als ANSI C API implementiert (aber auch andere Bindings vorhanden, z.B. Java)
  - Programmierbare Pipeline Stufen werden in GLSL (*OpenGL Shading Language*) programmiert
  - Implementierung ist Hardware-abhängig und daher in den Grafiktreiber integriert

KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON

OpenGL - Entstehung

- Einheitliche Softwareschnittstelle fehlt
- Erkenntnisse im Bereich Grafik flossen in proprietäre Softwarepakete
  - HP's Starbase
  - SGI's Graphics Library (GL)
- Standardisierungsbemühungen
  - PHIGS
  - GKS
- Enstehung von OpenGL
  - SGI's GL gepaart mit Hardware
  - OpenGL Spezifikation vorangetrieben von Mark Segal & Kurt Akeley

OpenGL - Bibliotheken

- Ausprägungsformen
  - OpenGL
  - OpenGL ES – Eingeschränkte Version für eingebettete Systeme
  - WebGL – Javascript Anbindung für Web Development (Funktionsumfang entspricht dem vom OpenGL ES)
- GUI Integration
  - Kann über Plattform-spezifische Integration erfolgen (Windows: WGL, X11: GLX, Mac OS X: Cocoa)
  - Plattformunabhängige Lösungen existieren ebenfalls (GLUT, freeGLUT, GLFW)
- Populäre Hilfsbibliotheken
  - GLU (OpenGL Utility Library) – Sammlung von Hilfsfunktionen
  - GLEW (GL Extension Wrangler) – Management von Erweiterungen
  - GLM –Bibliothek für Vektor- und Matrix-Opertionen

OpenGL Funktionalität

- Funktionales API
  - Objekte, Szenen und Bilder werden durch eine Abfolge von Kommandos beschrieben (funktionaler Ansatz, low-level)
  - Gegensatz: Szenen- oder Objektbeschreibung (deklarativer Ansatz, high-level)
- OpenGL Kontext
  - Summe aller Renderingattribute mit aktueller Ausprägung
  - Agiert als Zustandsmaschine, wobei der aktuelle Zustand durch OpenGL Funktionsaufrufe verändert werden kann
  - Primitive werden abhängig vom aktuellen Zustand gerendert

OpenGL Kontext

- Kontext wird von der Anwendung erzeugt und modifiziert, und enthält alle Zustandsvariablen der OpenGL-Maschine
- Beliebig viele Kontexte pro Anwendung möglich, jedoch maximal ein aktiver Kontext pro Anwendung
  - Kontext ist i.A. an ein Ausgabefenster gebunden
- Management des aktiven OpenGL Kontexts
  - Spezifikation der grafischen Attribute erfolgt unabhängig von den Rendering Primitiven

Der OpenGL Rendering Prozess

- Ablauf
  1. Erstellung des Ausgabefensters
  2. Erstellung des Kontexts
  3. Spezifikation der Szenengeometrie
  4. Hochladen der Szenengeometrie auf die GPU
  5. Rendern der Szenengeometrie

Fenstererstellung mit GLFW 1/2

```
#include <GLFW/glfw3.h>
#include <stdlib.h>
#include <stdio.h>

static void error_callback(int error, const char* description) {
    fputs(description, stderr);
}
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int
mods) {
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```


Fenstererstellung mit GLFW 2/2

```
int main(void){
    GLFWwindow* window;
    glfwSetErrorCallback(error_callback);
    if (!glfwInit())
        exit(EXIT_FAILURE);
    window = glfwCreateWindow(640, 480, "Simple example", NULL, NULL);
    if (!window) {
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);
    while (!glfwWindowShouldClose(window)) {
        renderScene();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

OpenGL Namenskonventionen

- Kommando-Präfix "gl" :
  - glClear, ...
- Konstanten-Präfix "GL" :
  - GL_POLYGON, GL_LINES, ...
- Parameter-Typen sind explizit in Anweisungsnamen kodiert

Suffix	Data Type	C-Data Type	OpenGL typedef
b	8-bit Integer	signed char	GLbyte
s	16-bit Integer	short	GLshort
i	32-bit Integer	int / long	GLint, Glsizei
f	32-bit Floating-Point	float	GLfloat, GLclamf
d	64-bit Floating-Point	double	GLdouble, GLclampd
ub	8-bit unsigned Integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned Integer	unsigned short	GLushort
ui	32-bit unsigned Integer	unsigned int / long	GLuint, GLenum, GLbitfield
v	Pointer	*	-

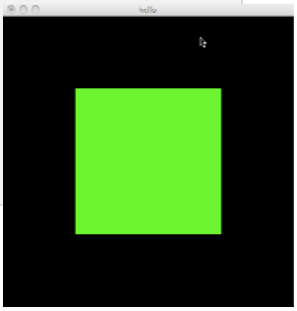
3.3 Rendern mit OpenGL

Übertragung von geometrischen Primitiven an die GPU

Rendern mit OpenGL (bis V2.1)

- Rendern erfolgt durch den *Immediate Mode*

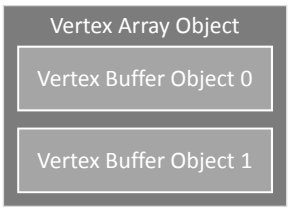
```
void renderScene() {
   glColor3f(0.0f, 1.0f, 0.0f); // Zeichenfarbe auf Grün setzen
   glBegin(GL_POLYGON);   // Beginn Primitiv
      glVertex2f(-0.5f,-0.5f);   // 1. Ecke
      glVertex2f(-0.5f,0.5f);    // 2. Ecke
      glVertex2f(0.5f,0.5f);     // 3. Ecke
      glVertex2f(0.5f,-0.5f);    // 4. Ecke
   glEnd();                      // Ende Primitiv
   glFlush(); // Ausgabepipeline synchronisieren
}
```



- Nachteile
  - Übertragung der Geometrie von der CPU an die GPU bei jedem Aufruf
  - Übertragung dauert deutlich länger als eigentlicher Zeichenvorgang

Rendern mit OpenGL (ab V3.0)

- Verwendung von *Vertex Array Objects (VAOs)*
  1. Spezifikation der Eckpunkte im CPU Speicher
  2. Übertragung der Eckpunktdaten von der CPU an die GPU
  3. Beschreibung des Speicherlayouts der Eckpunktdaten (Anzahl Eckpunkte, Anzahl Komponenten (x,y,z,…), Basistyp (float, double,…), …)
  4. Zeichnen des *VAOs*
  5. Freigabe des GPU Speichers
- Ein VAO fungiert dabei als Container für die Geometriedaten, die in ein oder mehreren *Vertex Buffer Objects* (*VBOs*) gespeichert sind



1. Spezifikation der Eckpunktdaten

- Eckpunktdaten werden als C Array spezifiziert

```
// Eckpunktdaten von zwei Dreiecken
GLfloat vertices[6][2] = { // 6 Eckpunkte mit 2 Komponenten (x,y)
   {0.0, 1.0}, {1.0, 1.0}, {1.0, 0.0}, // 1. Dreieck
   {0.5, 0.5}, {1.0, 0.3}, {0.7, 1.0} // 2. Dreieck
};
```

- In OpenGL muss ID für *VAO* und *VBO* generiert werden
- *VAO* und *VBO* wird mittels ID zur Bearbeitung ausgewählt

```
GLint vaoID;
glGenVertexArrays(1 , &vaoID);
glBindVertexArray(vaoID);
GLint vboID;
glGenBuffer(1 , &vboID);
glBindBuffer(vboID);
```


2. Übertragung der Eckpunktdaten

- *VBO* Allokation und Datenübertragung mittels

  glBufferData(GL_ARRAY_BUFFER , size, data, GL_STATIC_DRAW);
  - data ist die Adresse im Hauptspeicher
    (also *vertices* oder *&vertices[0]*)
  - size ist die Größe in Bytes
    (also *sizeof(vertices)* oder *numVertices * sizeof(float) * 2*)
- Für den Fall *data == NULL* wird nur der Speicher auf der GPU reserviert, aber es erfolgt keine Datenübertragung

3. Beschreibung des Speicherlayouts

- Dafür gibt es noch eine genauere Beschreibung:

  glVertexAttribPointer(attribIndex, size, type,normalized, stride, offset);

  - *size*: Anzahl Komponenten pro Eckpunkt (2 für xy, 3 für xyz)
  - *type*: Basistyp, typischerweise *GL_FLOAT* oder *GL_DOUBLE*
  - *stride*: Lücke zwischen Daten im Speicher (0: Daten direkt hintereinander)
  - *offset*: Wird auf die mit *glBufferData* angegebene Startadresse addiert. Kann zunächst einfach auf 0 gesetzt werden.
  - *attribIndex*: Attributnummer (0 für Position, später mehr…)
- Zunächst müssen Attribute aktiviert werden

  glEnableVertexAttribArray(attribIndex);
- Dabei sind die Koordinaten der Eckpunkte dem Attribut Index 0 zugewiesen
  - es gibt aber auch die Möglichkeit noch weitere Attribute zu vergeben (z.B. Farben, Normalen, …)

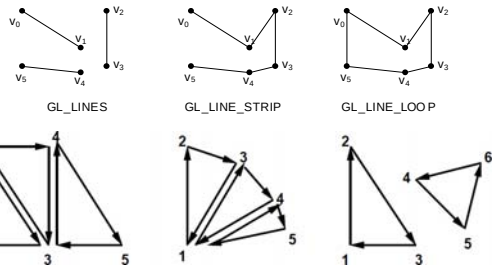
4. Zeichnen des VAOs 1/2

- Zeichnen erfolgt mit dem Befehl

  glDrawArrays(mode, first, count);
  - mode spezifiziert den Primitivtyp (z.B. GL_TRIANGLES oder GL_LINES)
- Von allen Eckpunkten werden ab Index first die nächsten count Eckpunkte gezeichnet. Die eigentlichen Eckpunktdaten müssen hier nicht mehr angegeben werden

4. Zeichnen des VAOs 2/2

- Es existiert eine geringe Anzahl an Primitivtypen
  - Points
  - Lines, Line Strips, Line Loops
  - Triangles, Triangle Strips, Triangle Fans, Patches



- Für Geometry Shader: mit Adjacency (wird später diskutiert)

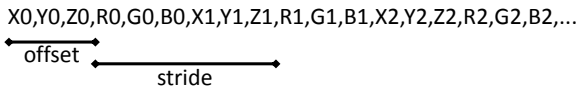
5. Freigabe des GPU Speichers

- VAO und VBO können wieder gelöscht werden mit

```
glDeleteBuffers(…);
glDeleteVertexArrays(…);
```

Zusätzliche Attribute

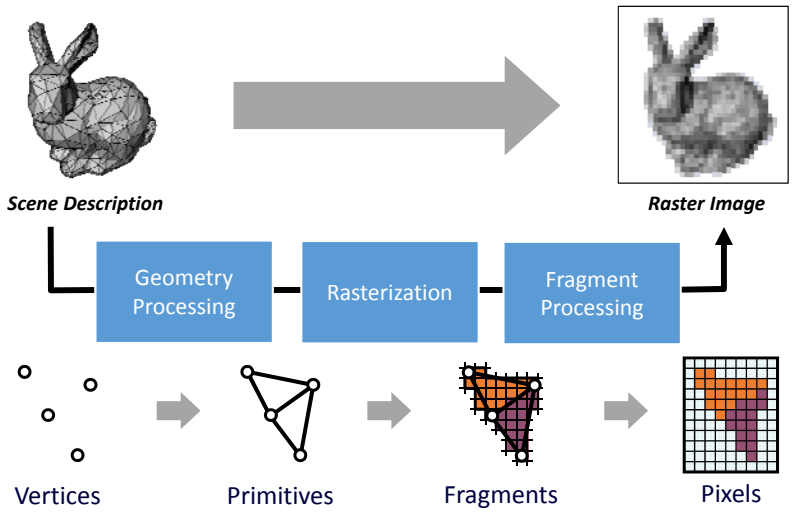
- Zusätziche Attribute sind zum Zeichnen notwendig
  - Frabe, Normalen, Textur-Koordinaten, ...
- Es existieren zwei Möglichkeiten diese Attribute den Eckpunktdaten zuzuweisen
  1. Attribute werden in separatem VBO an das gleich VAO gebunden
  2. Attribute werden *interleaved* mit Eckpunktdaten gespeichert, und *stride* und *offset* werden zum separieren benutzt



3.4 Die OpenGL Rendering Pipeline

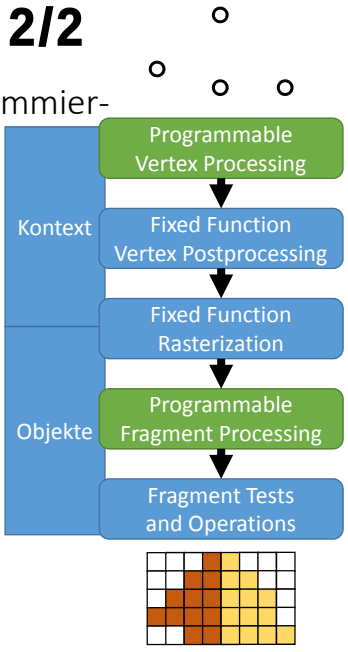
Programmierbare und statische Stufen

Die Renderingpipeline 1/2



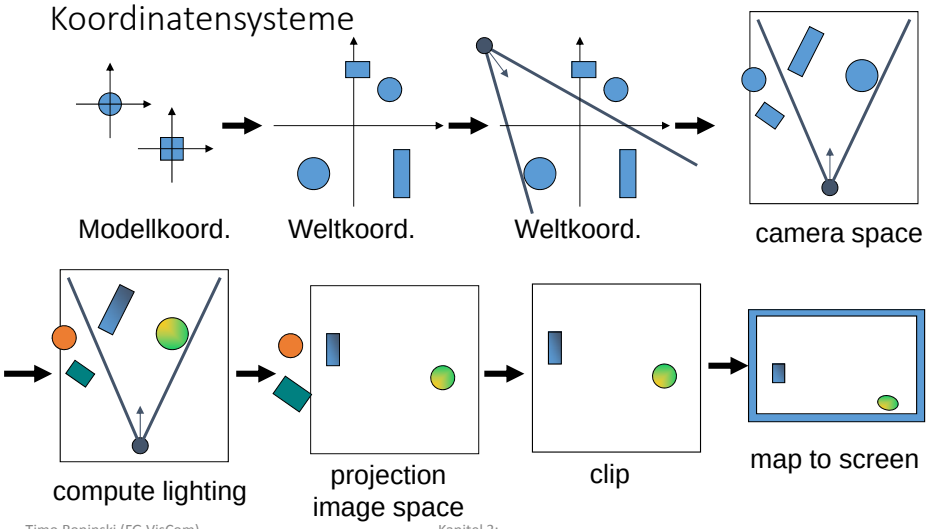
Die Renderingpipeline 2/2

- Abfolgung von fixen und programmierbaren Stufen
- Ablauf wird durch Kontext und Objekte beeinflusst
- Die programmierbaren Stufen werden durch *Shader* programmiert



Koordinatensysteme der Renderingpipeline

- Eckpunkte und später Fragmente durchlaufen mehrere Koordinatensysteme



Shader Programmierung – Historie 1/2

- Es existiert spezielle Grafkhardware (*dedicated*) und generelle Grafikhardware (*general purpose*)
- Spezielle Grafikhardware
  - Volumen-Rendering: VolumePro
  - Spielkonsolen: Angepasste NVidia-Chips in XBox
  - Grafikchips für mobile Endgeräte
- Generelle Grafikhardware
  - SGI's InfiniteReality
  - UNC's PixelFlow
  - NVidia's GeForce und ATI's Radeon Serie

Shader Programmierung – Historie 2/2

- Verschiedene Hersteller-abhängige Erweiterungen werden eingeführt, um Erweiterbarkeit für generelle Grafikhardware zu schaffen
  - Texture-Shader und Register-Combiner erlauben Modifikation der Anwendung von Texturen
- Anwendbarkeit ist aber noch sehr stark durch Möglichkeiten der Grafikhardware eingeschränkt
- Allgemeine Programmiersprache zur Entwicklung von Shadern wird benötigt

Assembler Shader

- Erste OpenGL Erweiterungen unterstützen Shaderprogrammierung in Assembler Code

```
!! ARBvp1.0
ATTRIB iPos = vertex.position ;
ATTRIB iNormal = vertex.normal ;
...
OUTPUT oPos = result.position ;
OUTPUT oColor = result.color ;
# Transform the vertex to clip coordinates
DP4 oPos.x, mvp [0], iPos ;
DP4 oPos.y, mvp [1], iPos ;
DP4 oPos.z, mvp [2], iPos ;
DP4 oPos.w, mvp [3], iPos ;
# Assign color contributions
...
MOV oColor.w, diffuseCol.w;
END
```

Shader Hochsprachen 1/2

- Offensichtlich ist Assemblercode schwer lesbar und schwer zu pflegen
- Hochsprachen zum Shading sind in C-ähnlicher Syntax spezifiziert
- Sourcecode wird vom Compiler in entsprechende ASM-Anweisungen übersetzt
- Benutzung der Shader nach initialer Bindung automatisch, sobald Geometrie gerendert wird
- Schnittstelle zu OpenGL indem entsprechende Variablen übergeben werden, oder der OpenGL-Zustand geändert wird (z.B. Binden von Texturen)

Shader Hochsprachen 2/2

- Existierende Hochsprachen für die Shader-Entwicklung
  - CG – C for Graphics von Nvidia
    - Cg wurde 2003 von NVidia als plattform-unabhängige Lösung angepriesen (in der Praxis eher semi-plattformunabhängig)
  - HLSL – High Level Shading Language von Microsoft
    - Microsoft stellte 2003 HLSL als Erweiterung zu DirectX vor (Fragment-Shader werden als Pixel-Shader bezeichnet)
  - GLSL – OpenGL Shading Language
    - GLSL wird seit 2005 zunächst als OpenGL Erweiterung unterstützt
- GLSL Syntax ähnlich zu C
  - Vekto- und Matrix-Basistypen existieren

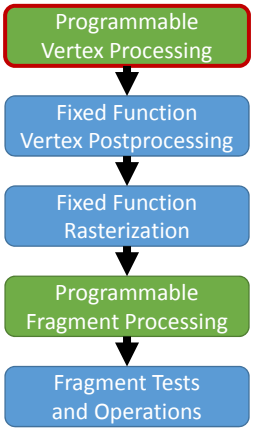
```
vec4  v1;   // same as 'float v1[4]' in C
ivec3 v2;   // same as 'int v2[3]' in C
vec4  v3 = v1.xzzy; // swizzling of components
```

Shader Typen

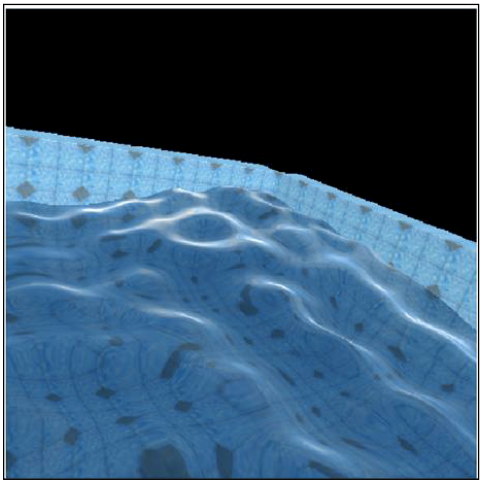
- Pipeline Konzept ist beibehalten worden und Shader sind für unterschiedliche Stufen verantwortlich
- Zuerst kam Programmierung der Geometriestufe durch Vertex-Shader auf
  - Vertex-Shader erlauben Modifizierbarkeit der Eckpunkt-Transformationen und Beleuchtung
- Danach gab es Erweiterung für bild-basierte Algorithmen durch Fragment-Shader
  - Fragment-Shader erlauben Programmierbarkeit der Texturanwendungen
- Anschließend folgten Geometry-Shader
  - Erzeugung neuer Geometrie (späteres Kapitel)
- Zuletzt wurden Tesselation-Shader eingeführt, die die Triangulierung von Primitiven beeinflussen

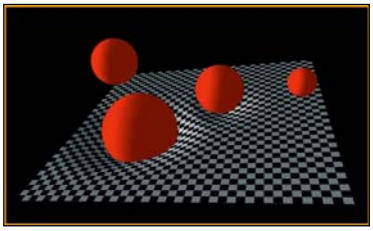
Vertex Shader Konzept

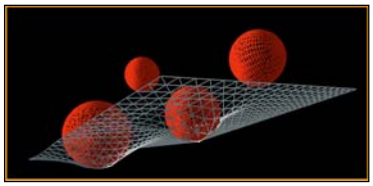
- Vertex-Shader erlauben die Veränderung der Geometrieverarbeitung
- Möglichkeiten eines Vertex-Shaders
  - geometrische Transformation von Vertices und Normalen
  - Beleuchtungsberechnung
  - Texturkoordinatengenerierung
  - Texturmatrix
  - Normalisierung von Normalen
  - Benutzerdefinierte Clipping-Planes
  - ...



Vertex Shader Beispiele 1/2

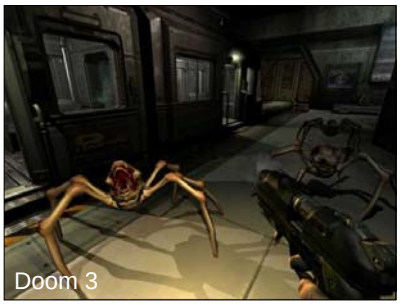


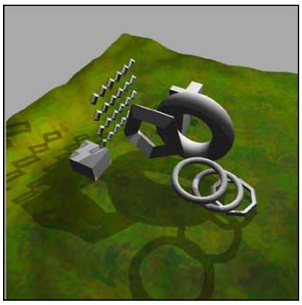




Vertex Shader Beispiele 2/2

- Berechnung von Schattenvolumen zum Rendern von Schatten
- Morphing
- Eigene Funktionen zur Texturkoordinaten-Generierung



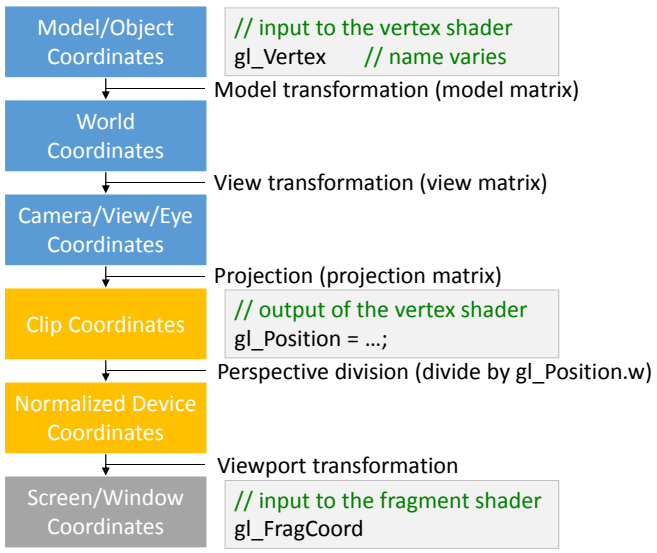


Vertex Shader – Koordinatensysteme 1/2

- Vertex Shader werden für jeden Eckpunkt ausgeführt
- Vertex Shader berechnet Koordinaten in Clipping Koordinaten
- Keine Information über benachbarte Eckpunkte vorhanden
- Die Anzahl der Eckpunkte kann nicht verändert werden
  - Vertex Shader können keine Eckpunkte generieren
  - Vertex Shader können keine Eckpunkte verwerfen
- Ausgabe ist immer die finale Koordinate des Eckpunktes

```
gl_Position = vec4(pos.x, pos.y, pos.z, pos.w);
```


Vertex Shader – Koordinatensysteme 2/2

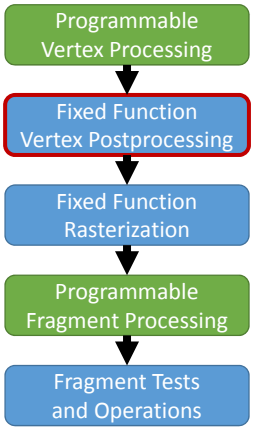


Vertex Shader Variablen

- Eingabe Typen
  - Uniforms : Nur Lesezugriff in VS (& FS)
    - Uniforms können nur vor glBegin() verändert werden
    - Z.B.: Lichtposition oder Lichtfarbe
  - Attribute : Nur im VS zugreifbar
    - Z.B.: Vertex Position oder Normale
  - Varyings: Datenübertragung vom VS in den FS
    - Lesezugriff im FS, Schreib- und Lesezugriff im VS

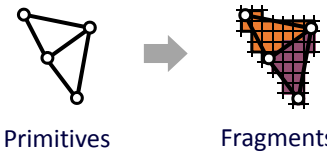
Fixed Function Vertex Postprocessing

- Clipping
  - Verwerfen der nicht-sichtbaren Geometrien
- Perspektivische Division
  - Transformation in Normalisierte Geräte-Koordinaten (*Normalized Device Coordinates*)
- *Viewport* Transformation
  - Abbildung in Bildschirmkoordinaten
  - Skalierung der Tiefenwerte (siehe später)

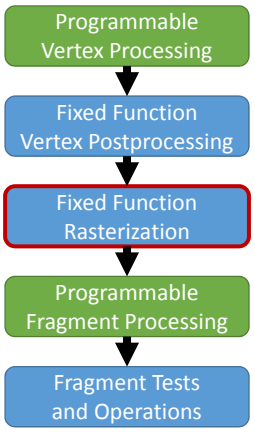


Fixed Function Rasterization

- Geometrie wird in Rasterkoordinaten umgewandelt

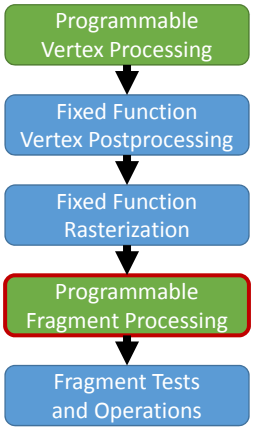


- Effiziente Algorithmen notwendig
- Algorithmen für verschiedene Primitivtypen spezialisiert
  - Linie, Dreieck, ...



Fragment Shader Konzept 1/2

- Fragment Shader beeinflussen die Farbe eines Pixels
- Möglichkeiten eines Fragment-Shaders
  - Blendung von Texturen
  - Nebelberechnung
  - Alpha-Test
  - zusätzliche Tiefen-Tests
  - Verwerfen von Fragmenten
  - Bump- und Environment-Mapping
  - ...



Fragment Shader Konzept 2/2

- Fragment Shader werden für jedes Fragment (=Pixelvorläufer) ausgeführt
- Fragment Shader berechnen die Farbe und den Tiefenwert des aktuellen Fragments
- Keine Information über benachbarte Fragmente vorhanden
- Die Anzahl der Fragmente kann verändert werden
  - Fragment Shader können keine Fragmente generieren
  - Fragment Shader können Fragmente verwerfen
- Ausgabe ist immer die finale Farbe des Fragments zusammen mit der Tiefe

```
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
gl_FragDepth = 0.5;
```


Shader Beispiel

```
/**
 * Transform vertex into clip coordinates.
 */
uniform mat4 modelViewMatrix_;
uniform mat4 projectionMatrix_;

void main() {
   vec4 eyePosition = modelViewMatrix_ * position;
   gl_Position = projectionMatrix_ * eyePosition_;
}
```

Vertex Shader

```
/**
 * Set fragment color to red.
 */
void main() {
   gl_FragColor = vec4(1.0,   // R
                       0.0,   // G
                       0.0,   // B
                       1.0); // A
}
```

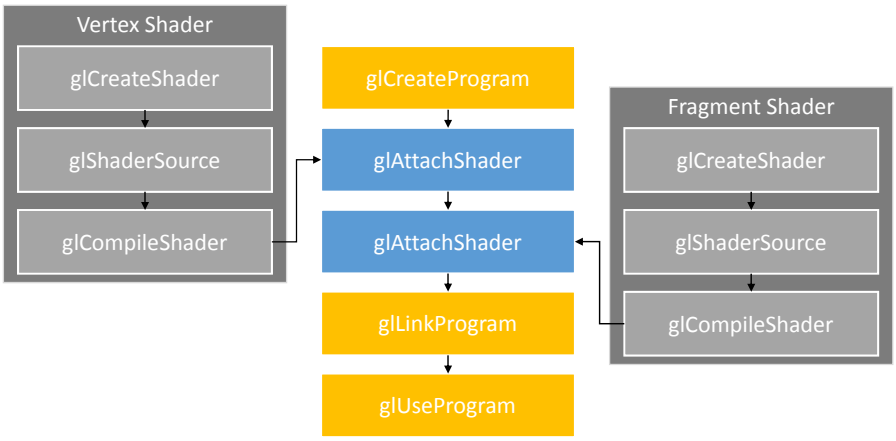
Fragment Shader

GLSL Shader Funktionen

- Trigonometrie-Funktionen
  - `radians, degrees, sin, cos, tan, asin, acos, atan`
- Potenz-Berechnungen
  - `pow, exp2, log2, sqrt, inversesqrt`
- Allgemeines
  - `abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep`
- Vektor- und Matrixoperationen
  - `length, distance, dot, cross, normalize, ftransform, faceforward, reflect, matrixCompMult`
- Und vieles mehr!

Benutzung von Shadern

- Shader werden zu einem Shader Program hinzugefügt



Binden von Shadern

```
v = glCreateShader(GL_VERTEX_SHADER); // vertex shader
char* vs = readTextFile(”vertex-shader.vert");
const char* vv = vs;
glShaderSource(v, 1, &vv, NULL);
free(vs);
glCompileShader(v);

f = glCreateShader(GL_FRAGMENT_SHADER); // fragment shader
char* fs = readTextFile(”fragment-shader.frag");
const char* ff = fs;
glShaderSource(f, 1, &ff, NULL);
free(fs);
glCompileShader(f);

p = glCreateProgram(); // create shader program
glAttachShader(p, v);
glAttachShader(p, f);
glLinkProgram(p);
glUseProgram(p);
```


Uniforms Setzen

- Uniforms werden verwendet um zwischen CPU und GPU zu kommunizieren
- Daten werden an *Uniform Locations* gebunden
  - Muss nach jedem Linkvorgang aktualisiert werden

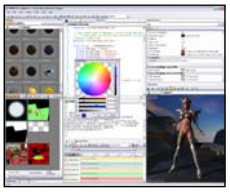
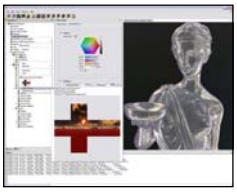
```
GLint loc0 = glGetUniformLocation(p, ”isoValue_");
glUniform1f(loc0, 0.45f);

float intensities[2] = {0.33f, 0.51f};
GLint loc1 = glGetUniformLocation(p, ”intensities_");
glUniform1fv(loc1, 2, intensities);

double vertices[12] = {0.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0, 1.0, 0.0,  0.0, 1.0, 0.0};
GLint loc2 = glGetUniformLocation(p, ”vertices_");
glUniform3dv(loc2, 4, vertices);
```

Laufzeit von Shadern

- Richtlinien für performante Shader
  - Da Fragment-Shader für eine große Anzahl an Fragmenten ausgeführt werden, sollten sie möglichst wenige Instruktionen enthalten
  - 3D Texturoperationen sind teurer als 2D Texturoperationen
  - Es sollten möglichst wenig Wechsel der aktiven Shader statt finden, da diese eigenen Zustand haben und dement-sprechend evtl. die Pipeline geleert werden muss
  - Bei häufigen Änderungen sollte das Verhalten eines Shaders über uniforms verändert werden, anstatt den Shader auszutauschen

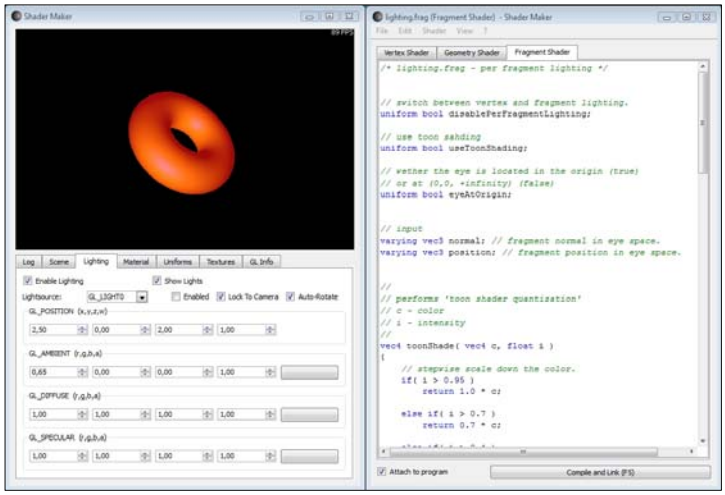
Shader Entwicklungsumgebungen 1/2

- FX Composer, NVPerfHUD (NVIDIA)
  - HLSL Shader IDE mit Performance Analyse und div. Statistiken
    www.developer.nvidia.com/page/tools.html
- RenderMonkey (ATI)
  - HLSL, GLSL Shader IDE mit Performance Analyse
    www.ati.com/developer/tools.html
- EffectEdit (Microsoft)
  - interaktiver HLSL Renderer
    http://msdn.microsoft.com/


Shader Entwicklungsumgebungen 2/2

- Shader Maker

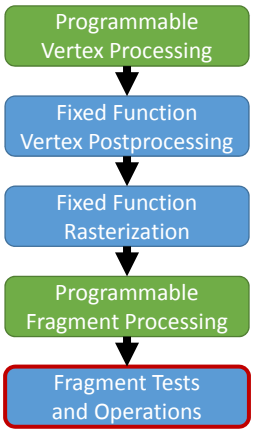


3.5 Fragment-Tests und Operationen

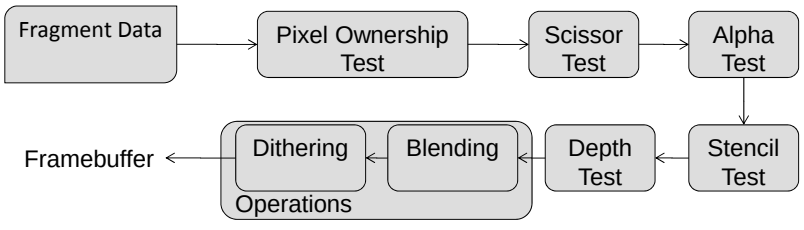
Der Endspurt zum Framebuffer

Fragment Test and Operations 1/2

- Nachdem Fragmente erzeugt wurden, müssen sie die Per-Fragment Operationen durchlaufen
  - Nur wenn alle diese Tests erfolgreich sind, landet das Fragment im Framebuffer und wird somit zum Pixel



Fragment Test and Operations 2/2

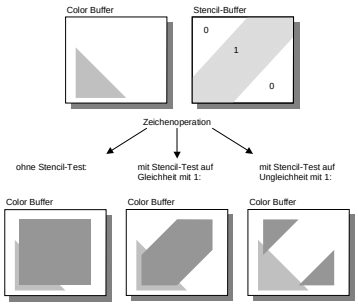


Alpha Test

- Wenn Framebuffer ein RGBA Buffer ist, kann der Alpha Test durchgeführt werden
- Der Test muss zunächst aktiviert werden
  - `glEnable(GL_ALPHA_TEST);`
- Die Vergleichsfunktion gibt an, welche Fragmente den Test überstehen
  - `glAlphaFunc(func, value);`
  - `func` kann folgende Werte annehmen `GL_NEVER, GL_LESS, GL_EQUAL, GL_GREATER, …`

Stencil Test 1/2

- Stenciling
  - Maskierung von Regionen im Bild
  - Zählung von Pixelüberdeckungen im Bild
  - Radierungen, pixelbasiertes Auflösen von Bildern, Verblenden
  - In 3D weiterführende Anwendungen:
    - Schatten, Constructive-Solid-Geometry, ...



Stencil Test 2/2

- Stencil Buffer
  - Ist eine Framebuffer-Komponente
  - Wird (indirekt) durch Zeichenoperationen befüllt
  - Keine speziellen Stencil Zeichenoperationen
- Stencil Operation
  - Modifikation des Stencil Buffers
- Stencil Funktion
  - Ausführung bei jeder Zeichenoperation

Ablauf Stencil Test

- Ablauf der Stencil-Operationen pro Fragment
  - Auswertung der Stencil Funktion
  - Ausführung der Stencil Operation
- Das Ergebnis der Stencil Funktion entscheidet, ob und wie das korrespondierende Stencil Pixel verändert wird
- Operationen: Wertzuweisung, Inkrementierung, Dekrementierung, Invertierung, ...
- In Abhängigkeit des rasterisierten Primitives wird indirekt in den Stencil Buffer geschrieben
  - Das rasterisierte Primitiv muss nicht im Farbbuffer erscheinen: das Zeichnen in den Farbbuffer lässt sich blockieren, so dass nur der Stencil Buffer modifiziert wird

Per-Fragment Operationen – Stencil Test

- Stencil Funktion und Stencil Operationen werden meist zusammen spezifiziert, weil sie voneinander abhängen.
- Beispiele:
  - Schreiben in den Stencil Buffer
    - Stencil Funktion = "immer positiv" (der Test liefert unabhängig vom Pixel immer "wahr")
    - Stencil Operation = "setze Stencil Pixel auf 1"
  - Schreiben in den Farbbuffer dort, wo im Stencil Buffer eine 1 steht
    - Stencil Funktion = "Stencil Pixel == 1"
    - Stencil Operation = "Stencil Pixel unverändert lassen"

Per-Fragment Operationen – Stencil Test

- Initialisierung des Stencil Buffers

```
glClearStencil(0); // init value
glClear(GL_STENCIL_BUFFER_BIT); // clear buffer
```

- Der Stencil Buffer sollte gemeinsam mit dem Farbbuffer initialisiert werden (Performance!)

```
glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Per-Fragment Operationen – Stencil Test

- Komponenten bzw. Bitplanes im Framebuffer können auf "read-only" gesetzt werden
- Anwendung: Zeichnen von Primitiven, ohne im Farbbuffer zu zeichnen
- Befehle in OpenGL

```
glColorMask(
 GLboolean r, GLboolean g,
 GLboolean b, GLboolean a
);
glStencilMask(GLuint bitplanemask);
```

(Die Bitmaske gibt an, in welche Bitplanes des Stencil Buffers tatsächlich geschrieben werden kann. Default: alle Bitplanes sind aktiviert.)

Per-Fragment Operationen – Stencil Test

- Stencil Funktion
  - Spezifikation durch
    - Stencil Vergleichsfunktion und
    - Stencil Vergleichswert
    - optional: Restriktion auf bestimmte Bitebenen im Stencil Buffer (Maske)

```
glStencilFunc(GLenum func, GLint refvalue, GLuint mask);
```

- `GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_NOTEQUAL, GL_GEQUAL, GL_GREATER`

Per-Fragment Operationen – Stencil Test

- Stencil Funktion Beispiele
  - Beispiel 1: Der Vergleich soll immer "wahr" sein (z.B. um einen Stencil Buffer zu beschreiben)

```
glStencilFunc(GL_ALWAYS, 0, 0xFFFF);
glEnable(GL_STENCIL_TEST);
...
glDisable(GL_STENCIL_TEST);
```

  - Beispiel 2: Der Vergleich soll dann wahr sein, falls der Referenzwert kleiner ist als der Wert im Stencil Buffer

```
glStencilFunc(GL_LESS, 2, 0xFFFF);
```


Per-Fragment Operationen – Stencil Test

- Stencil Operation
  - Spezifiziert Modifikation des Stencil Buffers in Abhängigkeit des Vergleichsergebnisses
  - Operationen:
    - `GL_KEEP`: momentaner Stencil Wert bleibt unverändert
    - `GL_ZERO`: Stencil Wert wird auf Null gesetzt
    - `GL_REPLACE`: Stencil Wert wird durch den Referenz-Wert ersetzt
    - `GL_INCR`: Stencil Wert wird um 1 erhöht
    - `GL_DECR`: Stencil Wert wird um 1 erniedrigt
    - `GL_INVERT`: Stencil Wert wird bitweise invertiert

Per-Fragment Operationen – Stencil Test

- Stencil Operation
  - Spezifikation der Operation für jedes Testergebnis:
    - Operation für den Fall, dass der Stencil Test negativ ausfällt (*fail*)
    - Operation für den Fall, dass der Tiefen Test negativ ausfällt (*zfail*, bzgl. Depth-Buffer in 3D)
    - Operation für den Fall, dass der Stencil Test positiv ausfällt (*zpass*)

```
glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

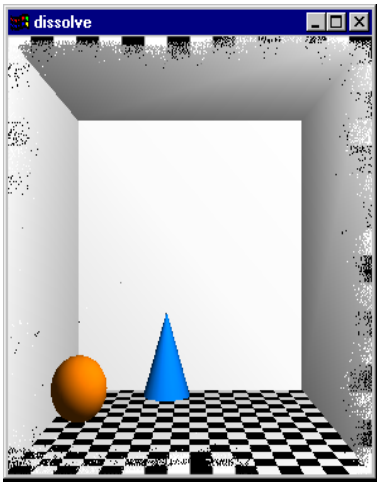

Stencil Buffer Beispiel 1/3

```
// setze Viewport und Ortho Projektion
// lösche Stencil Buffer mit 0
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);

// setze 1 dort, wo ein Pixel gezeichnet wird
glStencilFunc(GL_ALWAYS, 1, 0xFFFF);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
// setzt Farbbuffer auf read-only
glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);

// zeichne Primitiv und setze 1 im Stencil Buffer
…
// reaktiviere Farbuffer
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
```

Stencil Buffer Beispiel 2/3

```
void renderScene() {
   ...
   glClear(GL_COLOR_BUFFER_BIT);

   glDisable(GL_STENCIL_TEST);
   // zeichne unbeschränkt Primitive
   ...

   glEnable(GL_STENCIL_TEST);
   glStencilFunc(GL_EQUAL, 1, 0xFFFF);
   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
   // zeichne Primitive für Stencil-Wert==1
   ...
}
```


Per-FStencil Buffer Beispiel 3/3



Per-Fragment Operationen – Depth Test

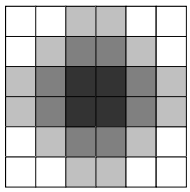
- Wenn der Framebuffer über einen Tiefenbuffer verfügt, kann der Depth Test durchgeführt werden
- Der Test muss zunächst aktiviert werden
  - `glEnable(GL_DEPTH_TEST);`
- Der Vergleich erfolgt mit dem im Buffer vorhandenen Tiefenwert
- Die Vergleichsfunktion gibt an, welche Fragmente den Test überstehen
  - `glDepthFunc(func);`
  - `func` kann folgende Werte annehmen `GL_NEVER, GL_LESS, GL_EQUAL, GL_GREATER, …`

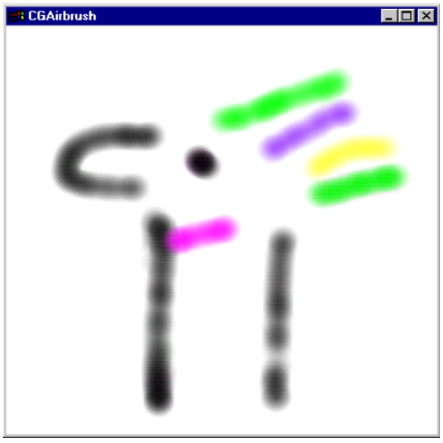
Per-Fragment Operationen – Blending

- Bildüberlagerung und -mischung
  - Alpha-Werte der Pixel
    - steuern die Kombination von Pixel-Werten
    - werden als Farbkomponente spezifiziert
      Farbpixel P = [R,G,B,A], R,G,B,A ∈ [0,1]
  - Anwendung: Modellierung von
    - transparenten Flächen
    - digitalen Bildkombinationen
    - Zeichen- und Maltechniken
  - Beispiel: Modellierung von Transparenz (alpha ≈ opacity)
    - A=0.0 vollständig transparent
    - A=0.2 80% transparent (d.h. 20% opak)
    - A=1.0 0% transparent (d.h. 100% opak)

Per-Fragment Operationen – Blending

- Beispiel-Anwendung für Transparenz: Zeichnen mit semi-transparentem Pinsel





Per-Fragment Operationen – Blending

- Arbeitsweise des Blendings
  - Blending ist eine Operation, die pro Fragment aufgerufen wird
  - Fragmente werden nach der Rasterisierung, aber vor dem Schreiben in den Framebuffer mit einer Blending Funktion modifiziert
  - Kombination
    - eines zu schreibenden **Source-Fragments**
    - mit dem im Framebuffer vorhandenen **Destination-Pixel**
    - unter Zuhilfenahme einer **Blending Funktion**.
- Zweistufiger Prozess
  - Bestimmung der Blending Faktoren
  - Berechnung des Pixelwertes aus Source-Fragment und Destination-Pixel

Per-Fragment Operationen – Blending

- Arbeitsweise des Blendings
  - Source Faktor: F_s
  - Destination Faktor: F_d
  - Source Fragment: $[P_{s,r},\ P_{s,g},\ P_{s,b},\ P_{s,a}]$
  - Destination Pixel: $[P_{d,r},\ P_{d,g},\ P_{d,b},\ P_{d,a}]$
  - Blended Pixel: $[F_s \cdot P_{s,r} + F_d \cdot P_{d,r},\ F_s \cdot P_{s,g} + F_d \cdot P_{d,g},\ F_s \cdot P_{s,b} + F_d \cdot P_{d,b},\ F_s \cdot P_{s,a} + F_d \cdot P_{d,a}]$

Per-Fragment Operationen – Blending

- Benutzung des Blendings
  - Einschalten (bzw. Ausschalten) des Blendings

```
glEnable(GL_BLEND);
glDisable(GL_BLEND);
```

  - Festlegen der Blending Faktoren

```
glBlendFunc(source factor, destination factor);
```

  - Blending Faktoren in OpenGL
    - `GL_ONE`
    - `GL_ZERO`
    - `GL_SRC_ALPHA`
    - `GL_ONE_MINUS_SRC_ALPHA`
    - `GL_DST_ALPHA`
    - `GL_ONE_MINUS_DST_ALPHA`
    - `...`

Per-Fragment Operationen – Blending

- Anwendungsbeispiele für das Blending
  - Beispiel 1: Kopieren von Source-Pixeln mit Überschreiben der Destination-Pixel

    ```
    glBlendFunc(GL_ONE, GL_ZERO);
    ```

    $P_{blend} = [P_{s,r}, P_{s,g}, P_{s,b}, P_{s,a}]$
  - Beispiel 2: Alpha-basiertes Mischen von Source Pixeln und Destination Pixeln

    ```
    glBlendFunc(GL_SRC_ALPHA,
      GL_ONE_MINUS_SRC_ALPHA);
    ```

    $Pblend = [P_{s,a} \cdot P_{s,r} + (1-P_{s,a}) \cdot P_{d,r},$
    $P_{s,a} \cdot P_{s,g} + (1-P_{s,a}) \cdot P_{d,g},$
    $P_{s,a} \cdot P_{s,b} + (1-P_{s,a}) \cdot P_{d,b},$
    $P_{s,a} \cdot P_{s,a} + (1-P_{s,a}) \cdot P_{d,a}]$

3.6 Der Framebuffer

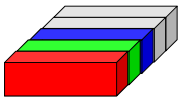
…enthält alles was auf dem Bildschirm (un)sichtbar ist

Framebuffer Verwaltung

- Viewport
  - Rechteckiger Bereich im Pixelraster, explizit festgelegt
  - Rendering Operationen sind auf diesen Bereich beschränkt
  - Alle Primitive, die ganz oder teilweise außerhalb liegen, werden am Viewport Bereich abgeschnitten (*Clipping*)
  - Alle Primitive werden im momentanen Modellkoordinatensystem (Default=Einheitsmatrix) gezeichnet

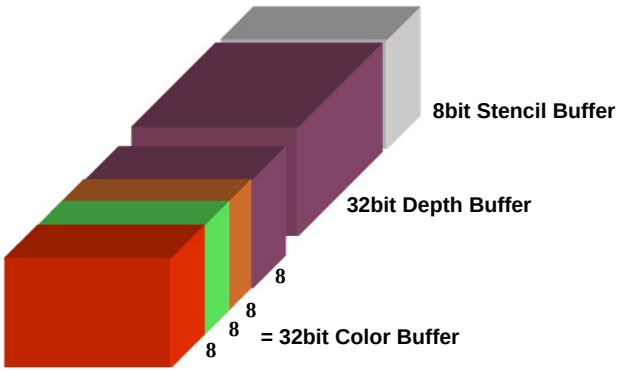
Framebuffer Verwaltung

- Framebuffer
  - An einen Kontext gebundenes Raster mit einer von der Anwendung vorgegebenen Auflösung und Struktur
  - Physische Realisierung auf der Computergrafik-Hardware
  - Logische Untergliederung:
    - Farbspeicher: *color buffer*
    - Bildmaskenspeicher: *stencil buffer*
    - Tiefenspeicher: *depth buffer*
    - …



Framebuffer Verwaltung

- Typische OpenGL Framebuffer Konfiguration



Framebuffer Verwaltung

- Alle Buffers eines Framebuffers haben die gleiche Rastergröße
- Zahl der Bitebenen pro Buffer kann individuell konfiguriert werden
- Einzelne Buffer werden heute direkt durch die Hardware unterstützt
- Buffer-Bezeichner
  - `GL_COLOR_BUFFER_BIT`
  - `GL_DEPTH_BUFFER_BIT`
  - `GL_STENCIL_BUFFER_BIT`
  - `GL_ACCUM_BUFFER_BIT`

Framebuffer Verwaltung

- Löschen eines Buffers

```
glClear(GLbitfield mask);
```

- Setzen der Initialisierungsfarbe für den Farbbuffer

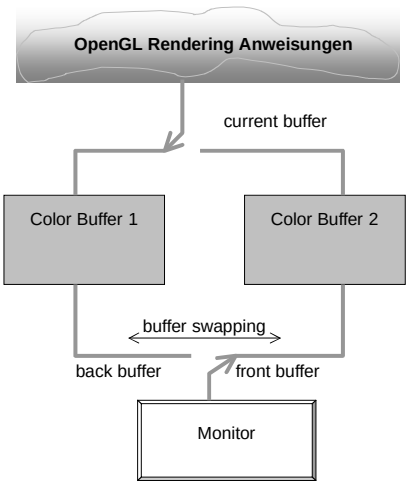
```
glClearColor(1.0,1.0,1.0,1.0);
glClear(GL_COLOR_BUFFER_BIT);
```


Framebuffer Verwaltung

- Double Buffering
  - Verwendung von zwei Farbbuffern
  - Bildaufbau findet im Hintergrund statt (**Back Buffer**)
  - Sichtbares Bild ist davon unabhängig (**Front Buffer**)
  - Nach Neudefinition des Bildes: Rollentausch (**Buffer Swapping**)
- Einsatz
  - Vermeiden von Flicker-Effekten
  - Animationen: glatte Bildübergänge

Framebuffer Verwaltung

- Double Buffering



3.7 Pixel-basiertes Rendering

Direkte Pixel-Verarbeitung ohne Geometrie

Pixelbasiertes Rendering

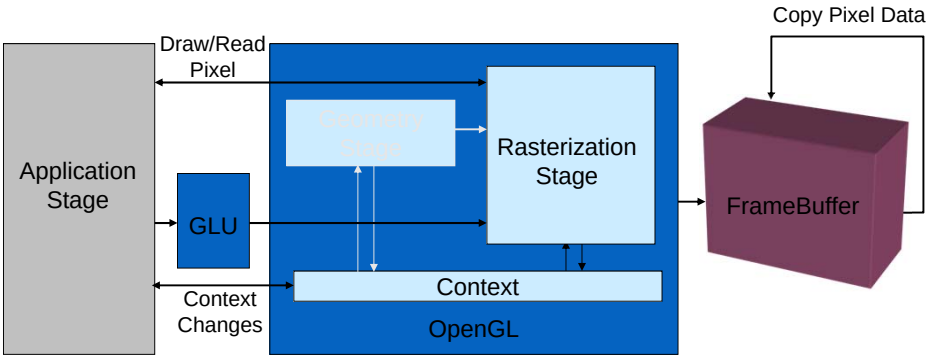
- Pixelbasierte Rendering Operationen
  - Allgemein
    - Lese- und Schreiboperationen für diverse Bilddatenformate, meist als externe Bibliothek (z.B. DevIl)
    - Konversionen zwischen Bilddatentypen
  - OpenGL mit Framebuffer Objekten
    - Kopieren von Framebuffer zu Framebuffer
    - Lesen von Framebuffer zum Hauptspeicher
    - Schreiben von Hauptspeicher zum Framebuffer

Bildoperationen

- Pixel-Array Operationen
  - `glDrawPixels()`: Schreibt Pixel-Array vom Hauptspeicher in den Framebuffer
  - `glReadPixels()`: Liest Pixel-Array vom Framebuffer in den Hauptspeicher (langsam!)
  - `glCopyPixels()`: Kopiert Pixel-Array innerhalb des Framebuffers
- Rasterposition
  - "Cursor" für das Einfügen/Lesen von Pixel-Arrays
  - Operationen nutzen momentane Rasterposition
  - `glRasterPos(x, y)`: Legt Rasterposition fest

Bildoperationen

- Rendering von Bilddaten mit OpenGL



Bildoperationen

- Raster-Koordinatensystem
  - Beispiel 1: Weltkoord. (für 2D, d.h. z = 0) = Bildschirmkoordinaten

```
glOrtho(0, winwidth, 0, winheight, -1, 1);
glRasterPos2i(10, 10); // Bildschirmkoordinaten
```

  - Beispiel 2: Normalisiertes Koordinatensystem

```
glOrtho(-1.0,1.0, -1.0,1.0, -1.0,1.0);
float p[2] = {0.0,0.0};
glRasterPos2fv(p); // Fenstermittelpunkt
```

- Fensterkoordinatentransformation
  - `glOrtho(xmin,xmax,ymin,ymax,zmin,zmax);`
  - Aufruf bei Initialisierung und bei Fenstergrößenänderung
  - `zmin` und `zmax` werden bei 2D-Grafiken auf –1 bzw. 1 gesetzt

Bildoperationen

- Schreiben von Pixel-Arrays in den Framebuffer

```
glDrawPixels(width, height, format, type, pixels);
glDrawBuffer(buffer);
```

  - Beispiel:

```
// 64x64 Array mit RGB-Komponenten
GLubyte img[64][64][3];
void draw() {
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2i(0,0);
    glDrawPixels(
      64, 64,           // Breite und Höhe
      GL_RGB,           // Pixel Format (Pixel-Verwendung)
      GL_UNSIGNED_BYTE, // Pixel Datentyp
      img               // Zu kopierendes Pixel-Array
    );
    glFlush();
}
```


Bildoperationen

- Lesen von Pixel-Arrays aus dem Framebuffer

```
glReadPixels(x,y, width, height, format, type,
  pixels);
glReadBuffer(buffer);
```

  - Beispiel:

```
GLubyte* img = 0;
int width, height;                      // Fenster Größe
void snapshot() {
    img = new GLubyte[width*height*3]; // Allokation
    glReadPixels(
      0, 0, width, height,              // Breite und
  Höhe
      GL_RGB,                           // Pixel Format
      GL_UNSIGNED_BYTE,                 // Pixel
  Datentyp
      img                               // Ziel Array
    );
}
```

Bildoperationen

- Formatangaben für Pixel-Array Operationen
  - `GL_RGB`: Rot-Grün-Blau-Werte des Framebuffers (FB)
  - `GL_RED`: Rot-Werte des FBs
  - `GL_ALPHA`: Transparenz-Werte des FBs
  - `GL_RGBA`: Rot-Grün-Blau-Alpha-Werte des FBs
  - `GL_LUMINANCE`: Helligkeitswerte des FBs
  - ...

Bildoperationen

- Datentypen für Pixel-Array-Operationen
  - `GL_UNSIGNED_BYTE`: eine Pixel-Komponente wird als 8-bit Wert dargestellt
  - `GL_UNSIGNED_INT`: eine Pixel-Komponente wird als 32-bit Integer-Wert dargestellt
  - Beispiel: `glReadPixels` mit
    - Format = `GL_RGBA` und
    - Datentyp = `GL_UNSIGNED_INT`

    ➔ pro Pixel werden 4 Komponenten (RGBA) in jeweils ein Integer gelesen (es werden also 4x4 =16 Bytes benötigt)

Bildoperationen

- Datentypen für Pixel-Array-Operationen
  - `GL_UNSIGNED_BYTE`: eine Pixel-Komponente wird als 8-bit Wert dargestellt
  - `GL_UNSIGNED_INT`: eine Pixel-Komponente wird als 32-bit Integer-Wert dargestellt
  - Beispiel: `glReadPixels` mit
    - Format = `GL_RGBA` und
    - Datentyp = `GL_UNSIGNED_INT`

    ➔ pro Pixel werden 4 Komponenten (RGBA) in jeweils ein Integer gelesen (es werden also 4x4 =16 Bytes benötigt)

3.8 Weiterführende Literatur

Zugrundeliegende und ergänzende Quellen

Literatur

- D. Shreiner, G. Sellers, J. Kessenich, B. Licea-Kane: OpenGL Programming Guide: The Official Guide to Learning OpenGL (8. Auflage), Addison-Wesley 2013.