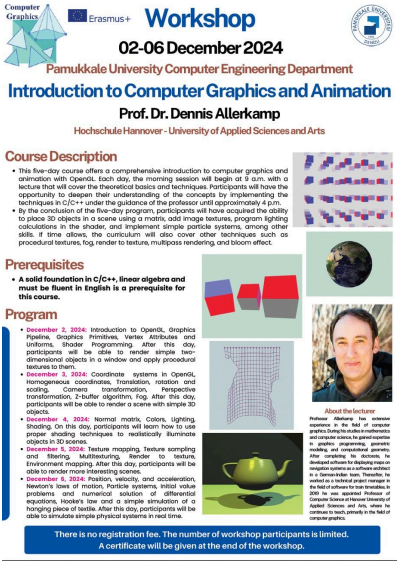
HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Introduction to Computer Graphics and Animation
Lecture 1 of 5

Prof. Dr. Dennis Allerkamp
December 2, 2024

Summary

Contents of the course


Computer Graphics

Erasmus+ Workshop

02-06 December 2024

Pamukkale University Computer Engineering Department

Introduction to Computer Graphics and Animation

Prof. Dr. Dennis Allerkamp

Hochschule Hannover - University of Applied Sciences and Arts

Course Description

- This five-day course offers a comprehensive introduction to computer graphics and animation with OpenGL. Each day, the morning session will begin at 9 a.m. with a lecture that will cover the theoretical basics and techniques. Participants will have the opportunity to deepen their understanding of the concepts by implementing the techniques in C/C++ under the guidance of the professor until approximately 4 p.m.
- By the conclusion of the five-day program, participants will have acquired the ability to place 3D objects in a scene using a matrix, add image textures, program lighting calculations in the shader, and implement simple particle systems, among other skills. If time allows, the curriculum will also cover other techniques such as procedural textures, fog, render to texture, multipass rendering, and bloom effect.

Prerequisites

- **A solid foundation in C/C++, linear algebra and must be fluent in English is a prerequisite for this course.**

Program

- **December 2, 2024:** Introduction to OpenGL, Graphics Pipeline, Graphics Primitives, Vertex Attributes and Uniforms, Shader Programming. After this day, participants will be able to render simple two-dimensional objects in a window and apply procedural textures to them.
- **December 3, 2024:** Coordinate systems in OpenGL, Homogeneous coordinates, Translation, rotation and scaling. Camera transformation, Perspective transformation, Z-buffer algorithm, Fog. After this day, participants will be able to render a scene with simple 3D objects.
- **December 4, 2024:** Normal matrix, Colors, Lighting, Shading. On this day, participants will learn how to use proper shading techniques to realistically illuminate objects in 3D scenes.
- **December 5, 2024:** Texture mapping, Texture sampling and filtering, Multitexturing, Render to texture, Environment mapping. After this day, participants will be able to render more interesting scenes.
- **December 6, 2024:** Position, velocity, and acceleration, Newton's laws of motion, Particle systems, Initial value problems and numerical solution of differential equations, Hooke's law and a simple simulation of a hanging piece of textile. After this day, participants will be able to simulate simple physical systems in real time.

About the lecturer

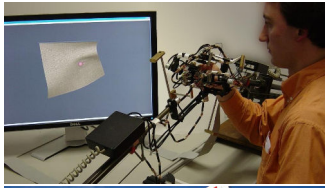
Professor Allerkamp has extensive experience in the field of computer graphics. During his studies in mathematics and computer science, he gained expertise in graphics programming, geometric modeling, and computational geometry. After completing his doctorate, he developed software for displaying maps on navigation systems as a software architect at TomTom in Amsterdam. Then, he worked as a technical project manager in the field of software for train timetables. In 2009 he was appointed Professor of Computer Science at Hanover University of Applied Sciences and Arts, where he continues to teach, primarily in the field of computer graphics.

There is no registration fee. The number of workshop participants is limited. A certificate will be given at the end of the workshop.

- Comprehensive introduction to computer graphics and animation with OpenGL
- Five-day course
- Lecture in the morning that will cover the theoretical basics and techniques
- Exercise in the afternoon (implementation of the techniques in C++)
- Participants will be able to
  - place 3D objects in a scene using a matrix,
  - add image textures,
  - program lighting calculations in the shader,
  - implement simple particle systems,
  - and other coputer-graphics techniques

About the lecturer

- Study and doctorate (1998 - 2010)
  - Mathematics with minor in computer science (Dipl.)
  - Doctorate at the Faculty of Electrical Engineering and Computer Science (Dr. rer. nat.)
  - Scientific employee (including EU project HAPTEX)
- Robert Bosch Car Multimedia GmbH (2010 - 2016)
  - Map display software development
  - Supervision of the Map-Team in Bangalore
  - Supervision of students
- HaCon Ingenieurgesellschaft mbH (2017 - 2019)
  - Timetable construction software development
  - Project management software development
- University of Applied Sciences Hannover (since 2019)
  - Professor for applied computer science and innovative teaching forms

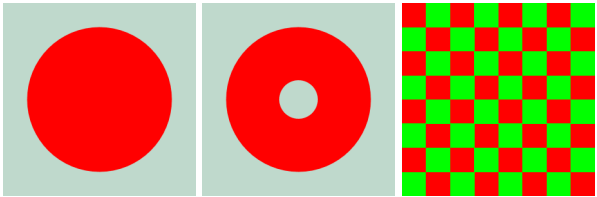


December 2, 2024

The following topics will be covered today:

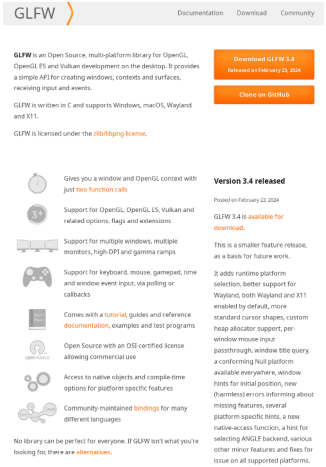
- Introduction to OpenGL
- Graphics Pipeline
- Graphics Primitives
- Vertex Attributes and Uniforms
- Shader Programming

After this day, participants will be able to render simple two-dimensional objects in a window and apply procedural textures to them.



Useful Libraries

GLFW



- GLFW is a library for creating windows and managing inputs
- Supports OpenGL, Vulkan, and more graphic APIs
- Simple integration and use
- Supports many operating systems and programming languages

Usage

```
#include <GLFW/glfw3.h>
…
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
GLFWwindow* window = glfwCreateWindow(600, 600, "Disc", nullptr, nullptr);
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
glfwMakeContextCurrent(window);
…
while (!glfwWindowShouldClose(window)) {
  …
  glfwSwapBuffers(window);
  glfwPollEvents();
}
glfwTerminate();
```

GLEW

- GLEW (OpenGL Extension Wrangler Library) is a cross-platform open-source C/C++ extension loading library
- OpenGL core and extension functionality is exposed in a single header file

Usage

```
#include <GL/glew.h> // include before GLFW header
…
glewInit();          // call after creating a window
…
```

GLM

- GLM is a C++ library for using vectors, matrices, and other mathematical functions with OpenGL
- Vulkan is also supported, but some peculiarities need to be observed
- Implements many functions required in computer graphics, especially the generation of transformation matrices
- Easy integration as a header-only library

Example

```
#include <glm/glm.hpp>
#include <glm/ext/matrix_transform.hpp>
#include <glm/ext/matrix_clip_space.hpp>
...
glm::mat4 projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 10.0f);
glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 8.0f), glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

stb_image

- stb_image is a C library for loading images
- Implements functions to load and decode images from files or memory
- Supports the image formats JPG, PNG, TGA, BMP, PSD, GIF, HDR, and PIC
- Easy integration as a header-only library

Example

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>

void loadTexture() {
  int width, height, channels;
  unsigned char* pixels = stbi_load("texture.jpg", &width, &height, &channels, STBI_rgb_alpha);
  VkDeviceSize imageSize = texWidth * texHeight * 4;

  if (!pixels) {
      throw std::runtime_error("failed to load texture image!");
  }
}
```

tinyobjloader

- tinyobjloader is a simple Wavefront OBJ loader
- Loads Wavefront OBJ files and stores the data in a simple data structure
- Supports materials, groups, and MTL files
- Easy integration as a header-only library

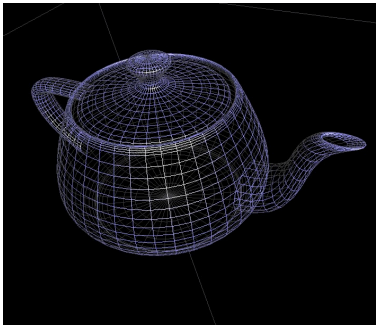
Example

```
#define TINYOBJLOADER_IMPLEMENTATION
#include "tiny_obj_loader.h"

int loadObject()
{
  tinyobj::ObjReader reader;

  if (!reader.ParseFromFile("teapot.obj")) {
    throw std::runtime_error(reader.Error());
  }
  auto& attrib = reader.GetAttrib();
  auto& shapes = reader.GetShapes();
  auto& materials = reader.GetMaterials();
}
```

Fundamentals of Computer Graphics

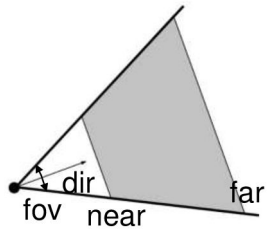
Rendering of Objects with Primitives

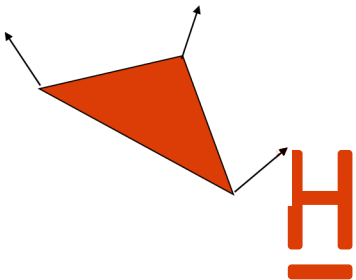
- Approximation of **objects** by **surfaces** and their illumination
- Approximation of **surfaces** via **primitives**
  - Points / Vertices, lines, triangles



What does “rendering” mean?

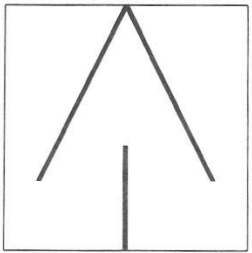
- View of a **3D scene** through a **virtual camera**
  - What makes it into the picture, what doesn't?
    - Everything seen from the **point** between **near** and **far** at an angle of **fov (field of view)** in the direction **dir (direction)**
  - Optical ratios, perspectives
- 3D scene **geometry**:
  - Primitives = **triangles**, lines, points
  - Light sources
  - Material properties of the geometry
  - Textures (images that are “stuck” onto the geometry)
- A **triangle** consists of 3 **vertices**
  - A vertex is a **3D position** and can contain **additional data** such as normals and color values

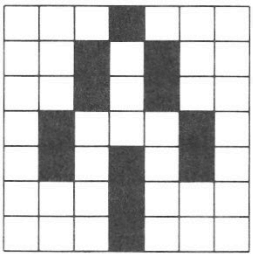


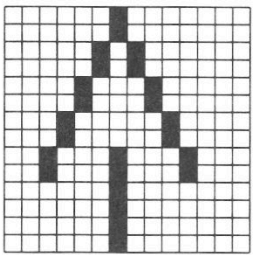


Representation via Rasterization

- Rasterization:
  - Through equations, all pixels can be determined and colored on a line/curve

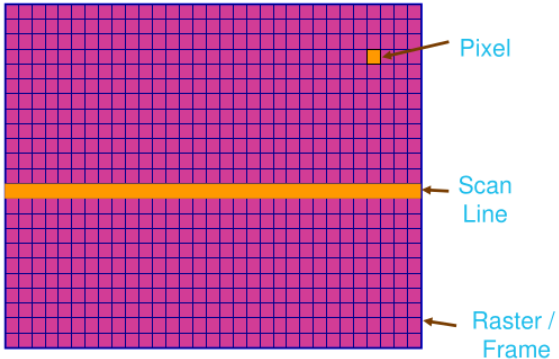




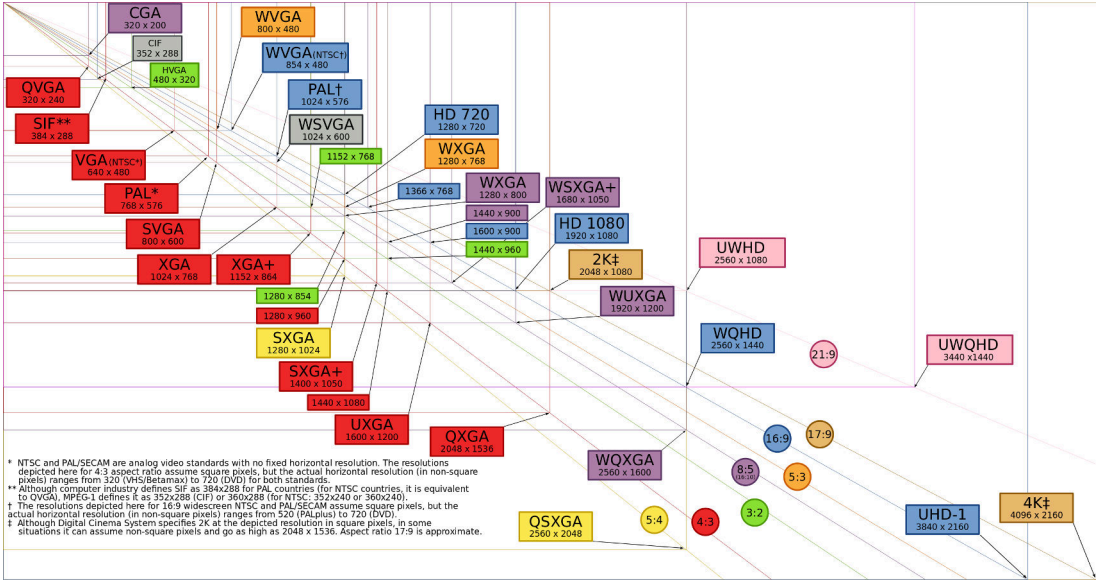


Technical Terms

- Raster:
  - Rectangular field of dots (raster points)
- Pixel:
  - A single image element or raster point
- Frame:
  - A single image displayed on the monitor
- Framerate:
  - Single images per second
- Scanline:
  - A pixel line
- Resolution:
  - Number of pixels per frame (320x240, 640x480, 1024x768, 1280x720, 1920x1080, 3840x2160)
- Pixel Density:
  - Pixels per inch, dpi
- Aspect ratio:
  - Width : Height (previously 4:3, now 16:9, 16:10, 21:9)



Pixel Resolutions



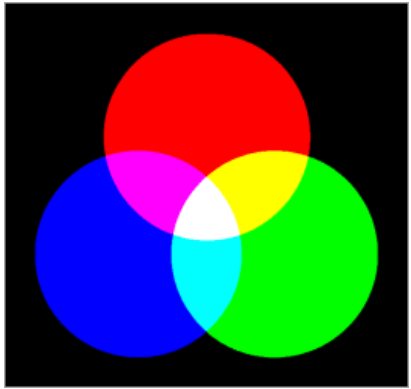
Source: https://en.wikipedia.org/wiki/Display_resolution

Color Depths

- Black and white: 1 bit/pixel
- Grayscale: 8 bit/pixel
- 8-bit color: Saves storage space
- 24-bit (RGB) color: 8 bit per color channel – red, green, blue
- How big must the frame buffer be for a 1600x1200 pixel image in true color (RGB)?
  - 8 bits for each RGB color channel
  - That's 24 Bit/Pixel
  - This amounts to 1600.1200.24 bit = 5.76 MBytes
  - Most graphics cards reserve 32 bit/Pixel for true color = 7.68 MBytes
- Data rate at 30 frames per second (FPS): 230 MBytes/sec

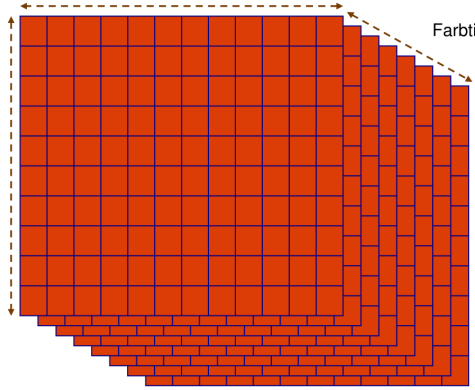
Additive Color Mixing

- RGB color model:
  - (0, 0, 0) black
  - (1, 0, 0) red
  - (0, 1, 0) green
  - (0, 0, 1) blue
  - (1, 1, 0) yellow
  - (1, 0, 1) magenta
  - (0, 1, 1) cyan
  - (1, 1, 1) white



The Frame Buffer: Double Buffering Algorithm

- Frame Storage
  - on the graphics card
- Double buffering
  - Display Frame Buffer 1
    Draw into Frame Buffer 2
  - Swap buffers
  - Display Frame Buffer 2
    Draw into Frame Buffer 1
  - Swap buffers
  - etc.
- Synchronization with refresh rate
  - Otherwise tearing



Introduction to OpenGL

What is OpenGL?

- An API for a real-time rendering system
  - Raster-based, State-based, Shader-based
- Definition Shader
  - *Shaders are special functions that are executed by the graphics hardware. These are small programs specifically compiled for the graphics card (GPU)*
- Other real-time rendering systems
  - DirectX
  - Vulkan
- Other rendering methods
  - Ray tracing
  - Volume rendering
  - ...

Why OpenGL?

- The 3D graphics API with the **largest market penetration**
  - API = Application Programming Interface
  - Has the **most powerful 3D language coverage**
- Unlike Direct3D, **platform-independent**
  - Independent of hardware
  - is delivered with graphics hardware
  - Has language binding to **all essential languages** (Perl, Python, C, C++, Java,...)
  - Runs with OpenGL ES on embedded systems and with WebGL on all common internet browsers.

OpenGL - Profiles

- Exists today in two different versions
  - “Compatibility Profile”
  - “Core Profile”
- Compatibility Profile
  - **All commands** (old and new) are available
  - Fixed-Pipeline functions
- Core Profile
  - **Only new commands** are available
  - Only usable with **Programmable Shaders**
  - **No helper function** for matrices, projections, illumination, etc.

Basic Structure of an OpenGL Program

- OpenGL is a **state machine**
  - is always in a certain state, called **OpenGL Context**
  - Never stateless due to **default states**
  - Example: color is defined => all subsequent objects are rendered in that color

```
// Set drawing mode
glEnable(…); // Turn on state
glDisable(…); // Turn off state
```

- Structure of **two essential parts**
  1. Initialization of a state: **How** is rendered
  2. Determination of 3D geometry: **What** is rendered

  → Rendering **using the state**

Platform Independence through Own Data Types

- OpenGL works across platforms
  - Windows since Windows 98/NT 4.0
  - MacOS since MacOS 9
  - X Window System (Linux, FreeBSD, etc.)
- Own data primitives
  - `GLfloat myFloat;`
  - `GLuint myUnsignedInt;`
  - `GLchar[] myChar;`
  - `GLbool myBool;`
  - …
- C Library = no overloading of functions!
  - Example: The function `glUniform*()` does not exist, it represents `glUniform2f()`, `glUniform3i()` and `glUniform3fv()`

OpenGL Objects

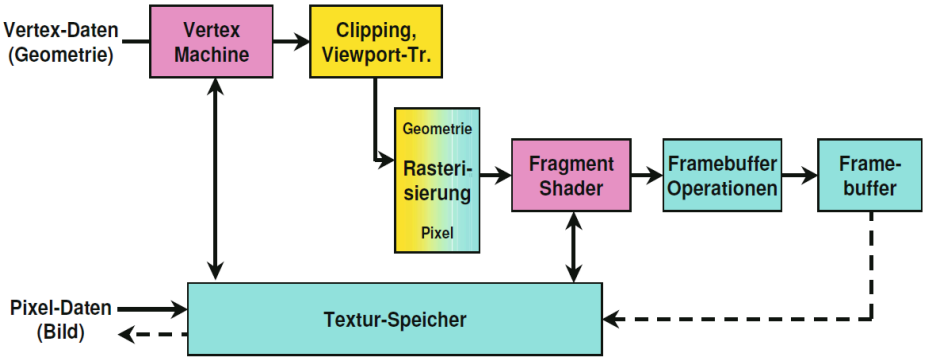
- C Library, no object orientation
- Procedure:
  - Creation of IDs, object allocation, assignment to the OpenGL context = current state, setting the parameters, returning to the default context
  - `unsigned int myObjectID = 0;`
    - IDs of objects are stored as ints
  - `glGen*(1, &myObjectID);`
    - Number, address of the identifier
    - Creates/allocates memory for a set of objects and returns the IDs to the address of myObjectID
  - `glBind*(OpenGL_target, &myObjectID);`
    - Activates the object for an OpenGL target
  - `glSet*(GL_OBJECT_TARGET,GL_OPTION_*,value)`
    - Everything set now is stored for this object as a state
  - `glBind*(OpenGL_target, 0);`
    - Until another object or the null object is binded to the target

The OpenGL Rendering Pipeline

- Pipeline follows the assembly line principle
  - Advantage: Individual computation steps can be **parallelized** (fast)
- Two different executions
  - Fixed Function Rendering Pipeline (FFP)
  - Programmable Rendering Pipeline (PRP) – **current standard**
- A rendering pipeline generally describes the **order of operations** in generating **images**

Programmable Rendering Pipeline

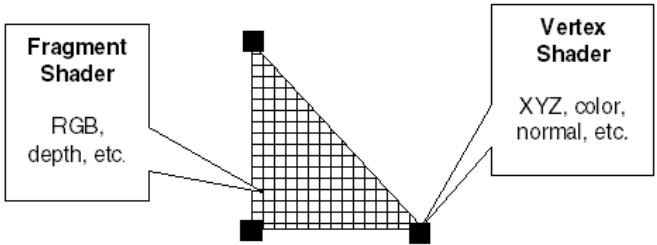
Overview



Computer graphics and image processing Nischwitz 2011 p.51

Programmable Rendering Pipeline

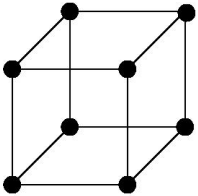
- Replaces part of the vertex operations with **Vertex Machine** and part of the fragment operations with **Fragment Shader**
- Shaders are **programmable**
- In Compatibility Profile **optionally usable**
- In Core Profile **absolutely necessary (Vertex Shader, Fragment Shader)**



http://climserv.ipsl.polytechnique.fr/documentation/idl_help/About_Shader_Programs.html

Vertex and Pixel Data

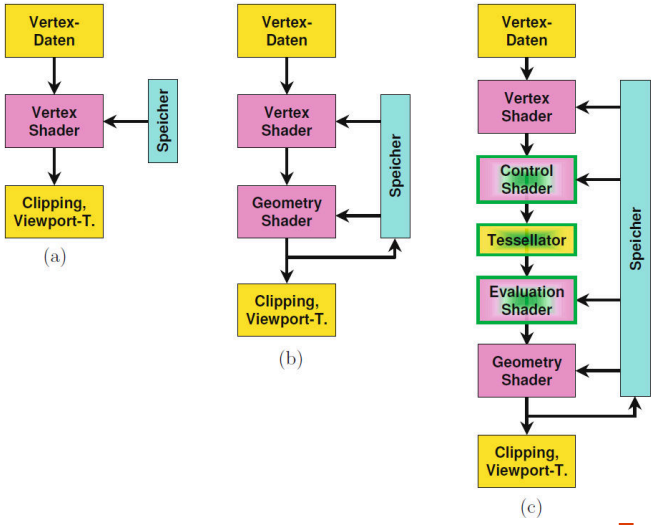
- Two different **data paths**
  - Vertex data
  - Pixel data
- Vertex Data
  - Are **vertices** / **corners** of a 3D geometry
  - May also contain further attributes, e.g., **texture coordinates (UV), normals and color values**
- Pixel data
  - Are **color values** of the individual texture points
  - Stored in **texture memory**
  - Can be **filtered, grouped and changed**
  - Also, **1D and 3D textures possible**



https://xxdegallaxx.wordpress.com/2014/02/04/vertices/
http://www.vdata.dk/wp_news/?p=5104

Vertex Machine

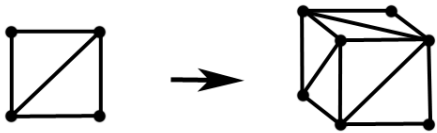
- Expands the vertex operations
  - a OpenGL 2.x:
    - Vertex Shader (**required**)
  - b OpenGL 3.x:
    - Geometry Shader (**optional**)
  - c OpenGL 4.x:
    - Control Shader (**optional**)
    - Tessellator
    - Evaluation Shader (**optional**)



Computer graphics and image processing Nischwitz 2011 p.52

Vertex Shader – Tasks

- Executed for **each vertex**
- **Transformations** of the vertices
  - **Projection transformations** including normalization
  - Forwarding of the transformed positions to the pipeline
  - Also **animations and lighting calculations** possible
- Cannot be evaluated:
  - Primitive assembly
  - Clipping
  - Viewport transformation



Vertex Shader - Calculates the vertices positions

https://code.tutsplus.com/articles/webgl-essentials-part-i–net-25856

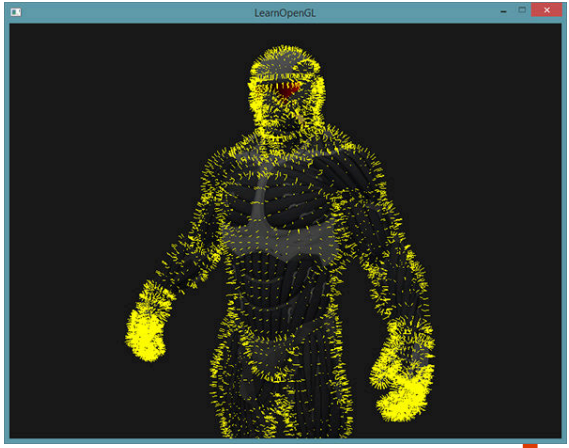
Vertex Shader

- **No helper matrices** (must be calculated and transferred themselves)
- Therefore, write your own matrices
- => Better, use existing **Library that calculates matrices**
  OpenGL Mathematics (GLM) http://glm.g-truc.net/0.9.8/index.html
- Example

```
#include <glm/glm.hpp>

#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);

glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

Geometry Shader

- Optional shader that **modifies primitives**
- These are polygons, triangles, lines or points
- Can **create or destroy new primitives**
- See code example in the Shader chapter

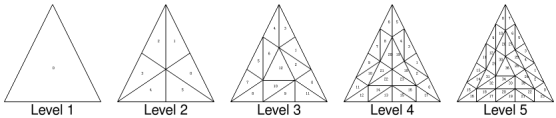


https://learnopengl.com

Tessellation Unit

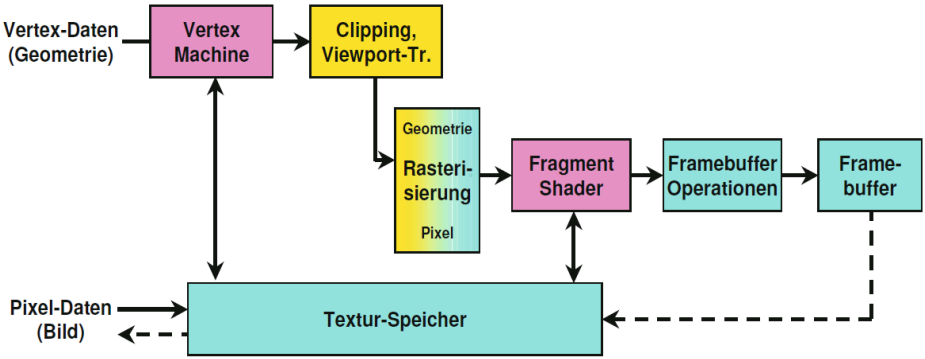
- **Tessellation** = subdivision into triangles
- **Control Shader** transforms control points
  - Determines a subdivision into triangle meshes
  - Hull shader in Direct3D
- **Tessellator** translates the control points into vertices
- **Evaluation Shader** deforms the vertices from the tessellator
  - Domain shader in Direct3D

http://xtreview.com/images/GeForce-8800-direct-x10/Geometry%20with%20DirectX%2010.jpgx



https://de.wikipedia.org/wiki/Tessellation-Shader#/media/File:Tessellation_Level_Table.png

Overview



Computer graphics and image processing Nischwitz 2011 p.51

Rasterization

- Determines the raster points that cover the primitives
- Interpolation of color values between vertices
- Enters depth values for each raster point
- Determines textures on polygons
- Maps vertex UV coordinates to fragment UV coordinates

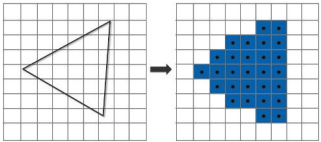
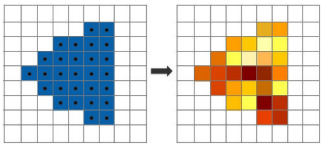


Image: http://gpudesign.bafree.net/wp-content/uploads/2010/07/p3.jpg

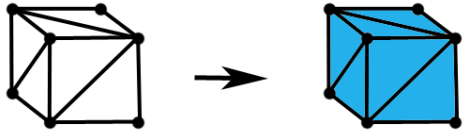
Fragment Shader

- Fragment = Pixel candidates
  - Contains data that will decide what a pixel will look like
  - Lighting, colors, visibility
- Tasks
  - Processes information on the color values of the texture or color at the fragment
  - Replaces or mixes color values at the fragment
  - Does not yet decide on visibility



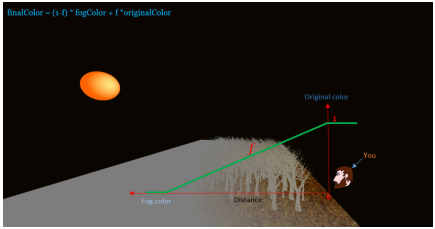
http://gpudesign.bafree.net/wp-content/uploads/2010/07/p4-300x132.jpg

- Evaluation **per fragment/pixel**
- Texture mapping
- Pixel-related lighting calculation
  - More complex methods like normal mapping
- Calculations with the depth value
  - Fog models
  - Flashlight
- **Forwarding of color values** to the pipeline



Fragment Shader - Colors in the Triangles

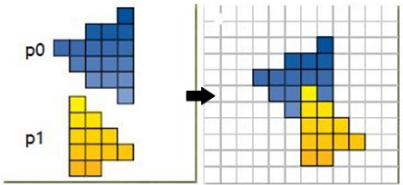
https://code.tutsplus.com/articles/webgl-essentials-part-i–net-25856



http://in2gpu.com/2014/07/22/create-fog-shader/

Framebuffer Operations

- Visibility test concerning the viewport
- Scissor Test
- Anti-Aliasing (Edge smoothing)
- Z-Buffer Test
- Stencil Test (Masking)
- Alpha Test
- Alpha Blending
- Writes result in the framebuffer



http://gpudesign.bafree.net/wp-content/uploads/2010/07/po.jpg

Framebuffer

- Contains the final color values
- Can be **displayed** on screen
- Usable as texture (**Multipass rendering**)
- Storable on a storage medium (**Offscreen rendering**)

http://i.i.cbsi.com/cnwk.1d/i/tim/2012/04/23/IB_TES.jpg

Basic Geometric Objects in OpenGL

Basic OpenGL Objects

- Basic objects consist of a *set of graphics primitives*
- Graphic primitives are *points*, *lines* or *planar polygons* (usually *triangles*)
- Triangles and line segments are defined by an *ordered set of vertices* (points in 3D space)
- Polygons also contain information on which vertices are *connected to each other and the directions of surfaces*

Abstraction Levels in Planar Polygons

- Polygons consist of two basic parts:
  - Ordered set/block of *vertices*
  - Specification of the *graphics primitive type* for a block of vertices
- *Faceted surfaces* can be determined using the graphics primitive type, which can be connected
- Accordingly compounded surfaces *can recreate closed surfaces*

Abstraction Levels in Planar Polygons

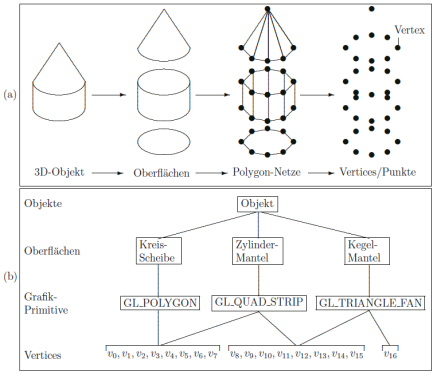


Bild 6.4: Darstellung eines 3-dimensionalen Objektes durch Oberflächen, Polygon-Netze bzw. Vertices. (a) Perspektivische Darstellung (b) hierarchische Baumstruktur

Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.77

Graphic Primitives in OpenGL

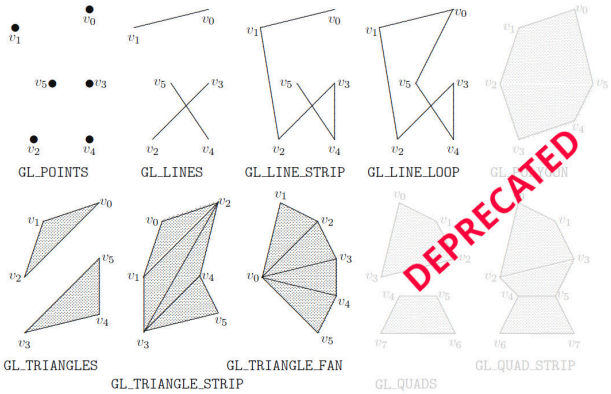
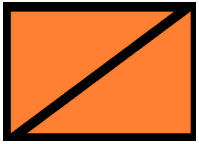


Bild 6.5: Beispiele für die Grafik-Primitive in OpenGL

Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.77

Triangles (GL_TRIANGLES)

- Connects three vertices with each other
  - forms a polygon
  - the first triangle is rendered from vertices V_0, V_1 and V_2,
  - the second from vertices V_3, V_4 and V_5 etc;
- if the number of vertices is not a whole multiple of three
  - the excess vertices are ignored



```
GLfloat vertices[][3] = {
    -0.5,  0.5, 0,
    -0.5, -0.5, 0,
     0.5,  0.5, 0,
    -0.4, -0.5, 0,
     0.6,  0.5, 0,
     0.6, -0.5, 0
};
…
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Triangles (GL_TRIANGLE_STRIP)

- Triangles (connected)
  1. Triangle rendered from vertices V_0, V_1 and V_2
  2. Triangle from vertices V_2, V_1 and V_3 (in this order),
  3. Triangle from vertices V_2, V_3 and V_4
  4. Triangle from vertices V_4, V_3 and V_5 etc;
- Order of vertices is important!
  - define front faces
- Simplest primitive for drawing complex surfaces
  - *most memory saving* and *fastest primitive*

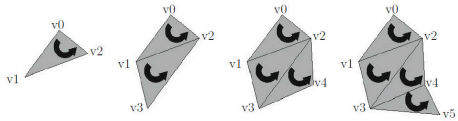


Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.94

Triangles (GL_TRIANGLE_FAN)

- Fans of triangles
- Like GL_TRIANGLE_STRIP but with a different vertex order:
  - First triangle from V_0, V_1 and V_2,
  - The second from V_0, V_2 and V_3,
  - The third from V_0, V_3 and V_4, etc.
- *Almost as effective* as GL_TRIANGLE_STRIP
- Suitable for circles / object peaks

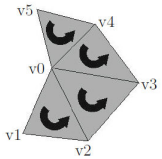


Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.96

Rendering Speed

Grafik-Primitiv-Typ	Rendering–Geschwindigkeit	durchschnittliche Anzahl an Vertices pro Viereck bzw. Linienstück
GL_POINTS	langsam	–
GL_LINES	mittel	2
GL_LINE_STRIP	schnell	1,1 (bei 10 Linienstücken)
GL_LINE_LOOP	sehr schnell	1,0 (bei 10 Linienstücken)
GL_POLYGON	langsam	4,0
GL_TRIANGLES	schnell	6,0
GL_TRIANGLE_STRIP	am schnellsten	2,2 (bei 10 Vierecken)
GL_TRIANGLE_FAN	schnell	2,2 (bei 10 Vierecken)
GL_QUADS	schnell	4,0
GL_QUAD_STRIP	sehr schnell	2,2 (bei 10 Vierecken)

Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.99

- Old commands from the FFP are *not* treated here (grey marked)

Modeling complex 3D scenarios

- Modeling complex 3D objects in OpenGL is very complex
- 3D objects only visible after compilation
- OpenGL does not offer functions for loading and saving of 3D scenarios
- Solution: Use modeling tools like 3dsmax, Maya, Blender, etc.

Figure: http://www.androidpolice.com/2013/08/28/autodesk-launches-maya-lt-targets-indie-and-mobile-game-developers-with-monthly-licensing-plans/#ap-lightbox

Vertex Data in OpenGL

Vertex Buffer Objects

- **(Vertex) Buffer Objects**
  - A memory area that the OpenGL Server (GPU) allocates and maintains
  - All data sent to the OpenGL Server is stored in Buffer Objects
  - In addition to Vertex Buffer Objects, there are 7 other types of Buffer Objects
- Located **in the graphics memory**
- Therefore **fast** because of direct access
- Data is usually in the form of **arrays**
- Can be changed **during runtime**
- Access is only possible via **Vertex Array Objects**

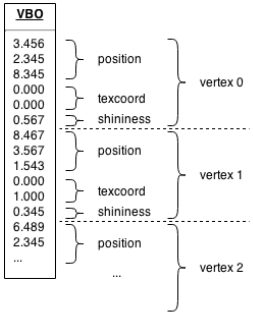


Figure: https://hackage.haskell.org/package/lowgl-0.4.0.1/docs/Graphics-GL-Low-BufferObject.html

Vertex Buffer Objects - Process

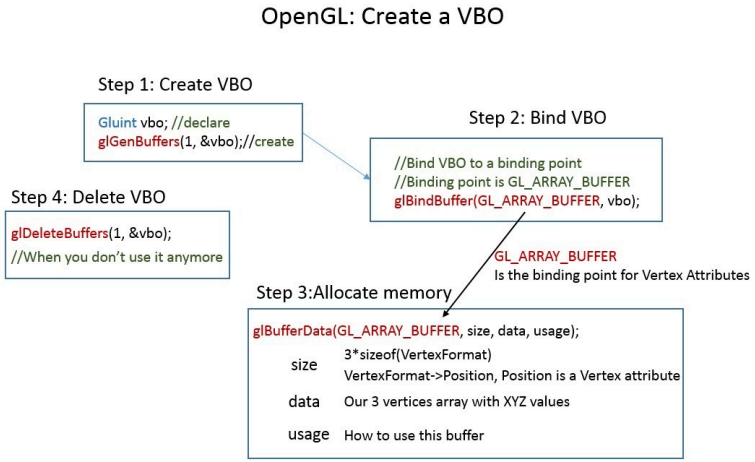


Figure: http://i1.wp.com/in2gpu.com/wp-content/uploads/2014/12/Create-vbo.jpg

Vertex Buffer Objects - Example

- `vertices[][3]` define 3 vertices (Triangle)
- `glBufferData`
  - Stores the *content* of Vertices in the graphics memory
- `GL_STATIC_DRAW`
  - tells the driver that the memory will *never* change
- Alternatively:
  - `GL_DYNAMIC_DRAW`
  - `GL_STREAM_DRAW`

```
GLuint vbo;
GLfloat vertices[][3] =
{ -0.5, 0.5, 0,
  -0.5, -0.5, 0,
   0.5, 0.5, 0 };

void init()
{
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
      GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Vertex Array Objects

- *References* and *describes* the contents of VBOs:
  - For example, it points to Vertices, Colors, Normals
  - NOT INCLUDED!
- Stores *states* of the VBOs
- Can reference *several* VBOs
- Responsible for sending to the rendering process (*Draw Call*) `glDrawArrays(…)`

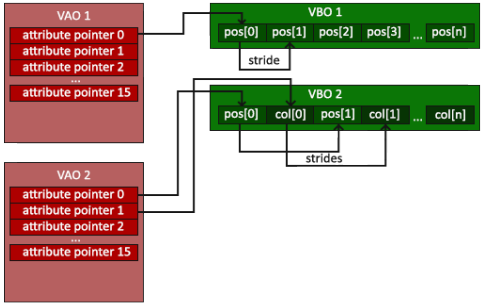


Figure: http://www.swiftgl.org/images/01/vertex-array-objects.png

Vertex Array Objects Example

1. Generate a VAO
2. Bind VAO
3. Bind VBO
4. Activate `glEnableVertexAttribArray(index)`
5. Reference with `glVertexAttribPointer(index,..)`
6. (Unbind VAO/VBO)

```
void init()
{
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
      GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glBindVertexArray(0);
}
```

glVertexAttribPointer

```
glVertexAttribPointer(  GLuint index, GLint size, GLenum type,
                        GLboolean normalize, GLsizei stride,
                        const GLvoid *VApointer );
```

Parameter:

index : *Index des Vertex-Attributs, z.B.* 0,
size : *Komponentenzahl pro Vertex-Attribut, z.B. 3 bei* x, y, z,
type : *Datentyp, z.B.* GL_FLOAT *für Gleitkommazahlen,*
normalize : *Normierung, z.B.* GL_FALSE *falls nicht normiert wird,*
stride : *Abstand in Bytes zwischen zwei Array-Elementen, z.B.* 0,
VApointer : *Zeiger auf das Vertex-Attribut-Array.*

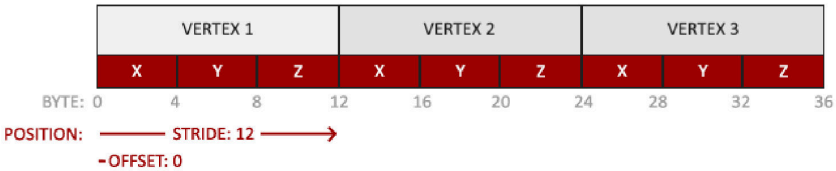


Figure: https://learnopengl.com/#!Getting-started/Hello-Triangle

- The predefined array structure can be referenced with the VAO
- Parameter: stride := Cycle size (in bytes)
- Parameter pointer := First byte that appears in the array for the respective component

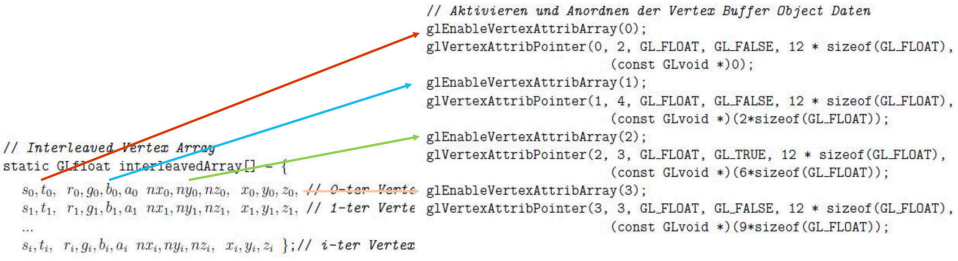


Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.114

OpenGL Drawing Techniques

Drawing with glDrawArrays

init() Funktion Auszug

```
glEnable(GL_DEPTH_TEST);

// Allocate memory for the buffer
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Create VAO for the VBO
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glEnableVertexAttribArray(0); // for vertex shader "in vec3 position"
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glBindVertexArray(0);
```

display() Funktion Auszug

```
// Clear Color and Depth Buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Use the shader for all next commands
glUseProgram(shaderProgram);
glBindVertexArray(vao);
// Draw Triangle - three vertices starting from 0
glDrawArrays(GL_TRIANGLES, 0, 3);
glBindVertexArray(0);
// Unbind Shader
glUseProgram(0);

// Double Buffering
glutSwapBuffers();
```

Index Buffer Objects

- Polygons that share vertices appear multiple times in a VBO
- With the help of an *Index Buffer Objects (IBO)*, the vertices of a VBO can be *indexed*
- In this example: *8 vertices, instead of 14*
- Computational effort reduced (GPU), *as fewer vertices need to be handled in the shaders*

```
// Vertex Array mit den acht Eckpunkten des Kubus
static GLfloat vertices[] = {   -1.0, +1.0, +1.0,  // 0 Vorderseite
                                +1.0, +1.0, +1.0,  // 1
                                +1.0, -1.0, +1.0,  // 2
                                -1.0, -1.0, +1.0,  // 3
                                -1.0, +1.0, -1.0,  // 4 Rückseite
                                +1.0, +1.0, -1.0,  // 5
                                +1.0, -1.0, -1.0,  // 6
                                -1.0, -1.0, -1.0,  // 7 };

// Index Array mit viersehn Referenzen zum Vertex Array
static GLubyte indices[] = {2, 3, 6, 7,          // Boden
                            4,                   // Rückseite links unten
                            3, 0,                // linke Seite
                            2, 1,                // Vorderseite
                            6, 5,                // rechte Seite
                            4,                   // Rückseite rechts oben
                            1, 0 };              // Deckel
```

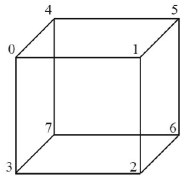


Bild 6.24: Ein Kubus mit acht eindeutig indizierten Eckpunkten

Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.107

Index Buffer Objects - Relation with VBO

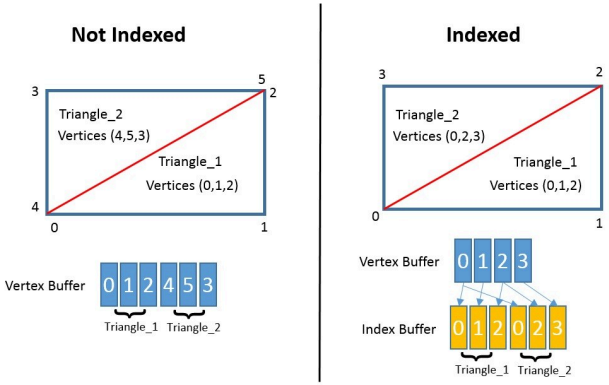


Figure: http://in2gpu.com/2015/07/09/drawing-cube-with-indices/

Index Buffer Objects - Example

- Same structure as VBO
  - Different: `GL_ELEMENT_ARRAY`, Array of `GLuint`, `GLshort`, `GLbyte`

```
GLuint vao;
GLuint vbo;
GLuint ibo;
GLuint shaderProgram;

GLuint indices[] =
  { 0,1,2,2,1,3 };

GLfloat vertices[][3]={
   -0.5,0.5,0,
   -0.5,-0.5,0,
    0.5,0.5,0,
    0.5,-0.5,0
      };
```

```
// Important to have type GL_ELEMENT_ARRAY_BUFFER
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

// Create VAO for the VBO
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); // Must be inside vao
glBindBuffer(GL_ARRAY_BUFFER, vbo); // Must be inside vao
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.106

Index Buffer Objects - Example

- `glDrawElements(…)` replaces `glDrawArrays(…)`

```
glDrawElements(GLenum mode, GLsizei numIdx, GLenum type,
                                const GLvoid *Idxpointer );
```

Parameter:

mode	: Grafikprimitivtyp, z.B. GL_TRIANGLE_STRIP,
numIdx	: Anzahl der Indices, z.B. 14 beim Kubus,
type	: Datentyp der Indices, z.B. GL_UNSIGNED_BYTE/SHORT/INT,
Idxpointer	: Zeiger auf das Index–Array.

Computergrafik und Bildverarbeitung Nischwitz 2011 S.106

```
// Rendering Function
void display()
{

    // Clear Color and Depth Buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Use the shader for all next commands
    glUseProgram(shaderProgram);
    glBindVertexArray(vao);
    // Draw Triangle with indices. VAO remembered IBO bind before
    glDrawElements(GL_TRIANGLES, sizeof(indices), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
    // Unbind Shader
    glUseProgram(0);




    // Double Buffering
    glutSwapBuffers();
}
```

Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.106

Primitive Restart

- Rendering via Arrays or Elements
  - `glDrawArrays` vs. `glDrawElements`
- Consolidation of DrawCalls
  - `glMultiDrawArrays` or `glMultiDrawElements`
- Indexed Rendering with Primitive Restart
  - Start new primitive with next index
  - Without Primitive Restart:
    - Triangle strip index array:
      `{ 0 1 2 3 65535 2 3 4 5 }`
    - 7 triangles
  - With Primitive Restart:
    - `glEnable(GL_PRIMITIVE_RESTART)` and `glPrimitiveRestartIndex(65535)`
    - 4 triangles: {0 1 2}, {2 1 3}, {2 3 4}, {4 3 5}

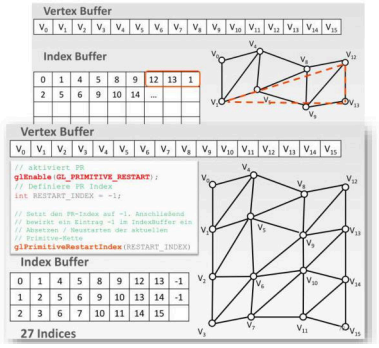


Figure: Draw Optimizations

Backface Culling

- OpenGL allows *cutting off of polygon back sides*
  - Makes rendering faster, as fewer fragments need to be processed
- Enable with `glEnable(GL_CULL_FACE)`
  - Default: disabled
- Culling can be specified regarding front/back with `glCullFace(Glenum mode)`
  - Modes { GL_FRONT, GL_BACK, GL_FRONT_AND_BACK }
  - Default: GL_BACK

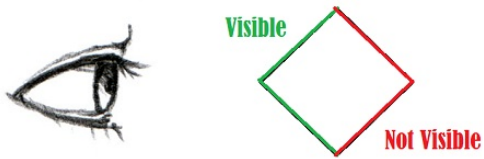


Figure: http://askagamedev.tumblr.com/post/92638684416/game-optimization-tricks-part-2-backface

Orientation / Winding of Polygons

- Triangles have a *front and back of the surface*
  - Front face and Back face
- Direction results from the arrangement of vertices in the image
  - GL_CW
    - clockwise, mathematically negative
    - Default: Back face direction
  - CL_CCW
    - counterclockwise, mathematically positive
    - Default: Front face direction
- Changing the behavior
  - void `glFrontFace(GLenum mode)`
  - mode = {GL_CW, GL_CCW}
  - Default: GL_CCW (mathematically positive)

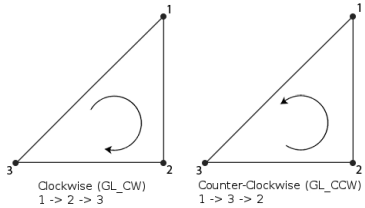


Figure:
https://www.khronos.org/opengl/wiki/Face_Culling

Consistent Polygon Orientation

- Construction of closed surfaces should have the same polygon orientation
- This makes it possible to use *Backface-Culling*

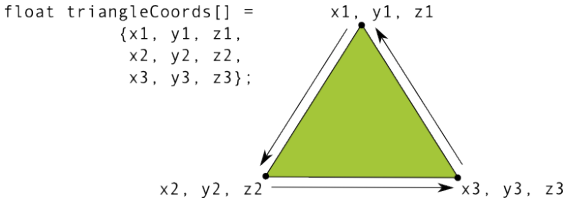


Figure: https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/graphics/opengl.html

Review: Triangles (GL_TRIANGLE_STRIP)

- Triangles (connected)
  1. Triangle rendered from vertices V_0, V_1 and V_2
  2. Triangle from vertices V_2, V_1 and V_3 (in this order),
  3. Triangle from vertices V_2, V_3 and V_4
  4. Triangle from vertices V_4, V_3 and V_5 etc;
- Order of vertices is important!
  - define front faces
- Simplest primitives for drawing complex surfaces
  - *most memory-saving* and *fastest primitives*

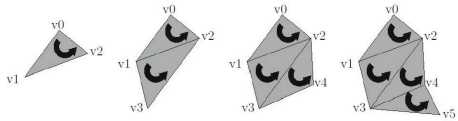


Figure: Computergrafik und Bildverarbeitung Nischwitz 2011 S.94

OpenGL Shading Language

Shading Programming Languages

- **No PRP possible without shaders**
- Three shading programming languages
  - Cg (C for Graphics) by NVIDIA - deprecated
  - HLSL (High Level Shading Language) from Microsoft
  - GLSL (OpenGL Shading Language) from the ARB Working Group
- They are **subroutines** on the graphics card
- Only GLSL relevant here (**Core Profile**, as it's the new standard)
  - Largest market share
  - Standard for OpenGL
  - **C syntax** similarity
  - In principle identical with HLSL

Data transfer in the GLSL Pipeline (Simplified)

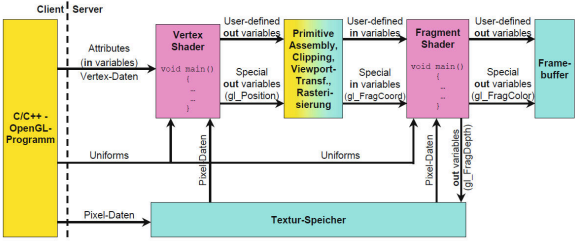
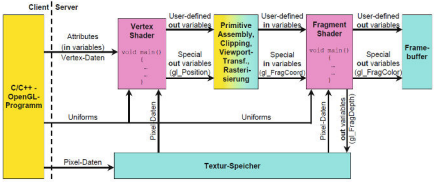


Figure: Computer Graphics and Image Processing Nischwitz 2011 p.55

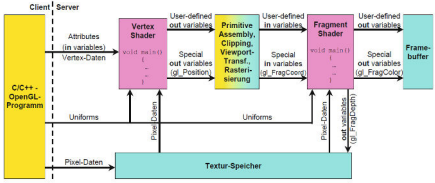
Data transfer in the GLSL Pipeline



- Works on the **Client-Server**-Principle
- Client side consists of the code in CPU and RAM
  - **Commands and data are sent to the server**
- Server = Graphics card
  - Vertex/Fragment-Shaders are **controlled on the server side**
  - Passed as `const GLchar*`

Inputs/Outputs in Shaders

- *in* Variables
  - Vary per Vertex/Fragment
- *Uniforms*
  - Vary per Draw Call
  - Passing parameters from the client side
- Textures (*Sampler* Variables)
  - Pixel data from the texture memory
- *out* Variables (for Output)
  - User-defined or built into OpenGL (built-in)



Data Types

- Basic data types = same convention as in C/C++

```
bool    bSwitch = 0;          // Boolsche Variable
int     iNum = -20;           // signed integer Variable
uint    uiCount = 10;         // unsigned integer Variable
float   fValue = 3.0f;        // floating point Variable
double  dPi = 3.14153265d;    // double-precision floating point
```

Data Type	Possibilities
Vector	vec2,vec3,vec4
Matrix	mat2,mat3,mat4
Matrix (non-square) mat	2x3,mat2x4, mat3x2, etc...
Texture	sampler1D, sampler2D, sampler3D
Texture Arrays	sampler1DArray, sampler2DArray, sampler3DArray

Initialization of Data Types

```
float fColor[3];                        // Array mit 3 float-Komponenten

struct Light{
    vec3 fColor;
    vec3 fPos; } light;                 // Struktur

in/out/uniform Matrices{
    uniform mat4 MV;
    uniform mat4 MVP;
    uniform mat3 NormalM; } mat;        // Block
vec4 fPos = vec4(0.3, 1.0, 0.0, 1.0);
mat4 fMat = mat4(1.0, 0.0, 0.0, 0.0,
                 0.0, 1.0, 0.0, 0.0,
                 0.0, 0.0, 1.0, 0.0,
                 0.0, 0.0, 0.0, 1.0);       // Einheitsmatrix
const mat4 Identity = mat4(1.0);            // konstante Einheitsmatrix
```

Initialization of Data Types

```
float fZ = fPos[2];                    // fZ erhält den Wert 0.0
vec4 fVec;
fVec = fMat[1];                        // fVec = (0.0, 1.0, 0.0, 0.0)
vec4 V1 = vec4(1.0, 2.0, 3.0, 4.0);
vec4 V2 = V1.zyxw;                   // V2 = (3.0, 2.0, 1.0, 4.0)
vec3 V3 = V1.rgb;                    // V3 = (1.0, 2.0, 3.0)
vec2 V4 = V1.aa;                     // V4 = (4.0, 4.0)
vec2 V4 = V1.ax;                     // Fehler, Mischung der Swizzling-Sets
vec4 V1 = vec4(1.0, 2.0, 3.0, 4.0);
V1.x = 5.0;                           // V1 = (5.0, 2.0, 3.0, 4.0)
V1.xw = vec2(5.0,6.0);                // V1 = (5.0, 2.0, 3.0, 6.0)
V1.wx = vec2(5.0,6.0);                // V1 = (6.0, 2.0, 3.0, 5.0)
V1.yy = vec2(5.0,6.0);                // Fehler, y wird doppelt verwendet
```

Built-in Functions/Operations in GLSL

- Most important mathematical functions built-in
- Accelerated by the graphics card
- *Same name convention* as in C/C++
- Many functions *accept matrices and vectors* as parameters

Operations

- Vector operations: `normalize`, `dot`, `cross`, `length`, `distance`, `reflect`, `refract`
- Overloading possible for 2-, 3- and 4-dimensional vectors
- Matrix Operations: `*`, `transpose`, `determinant`, `inverse`
- Trigonometry: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- Conversions: `radians`, `degrees`
- Exponential functions: `pow`, `exp`, `exp2`, `log`, `log2`, `sqrt`
- Miscellaneous:
  - linear interpolation (`mix`), Max and Min functions (`max`, `min`), absolute value (`abs`), signum function (`sign`), restriction to a value range (`clamp`), step function (`step`),
  - Partial derivatives (`dFdx`, `dFdy`), sum of absolute values of the partial derivatives of a function in the fragment shader (`fwidth`)

Custom Functions

- Similar to C functions, but recursion is not allowed
- Value passing through *in*, *out*, and *inout* variables

Example: What comes out? (in1, out1, out2?)

```
void MyFunction(in float inputValue, out int outputValue, inout float inAndOutValue)
{
    inputValue = 0.0;
    outputValue = int(inAndOutValue + inputValue);
    inAndOutValue = 3.0;
}
void main()
{
    float in1 = 10.5;
    int out1 = 5;
    float out2 = 10.0;
    MyFunction(in1, out1, out2);
}
```

GLSL in the Vertex Shader

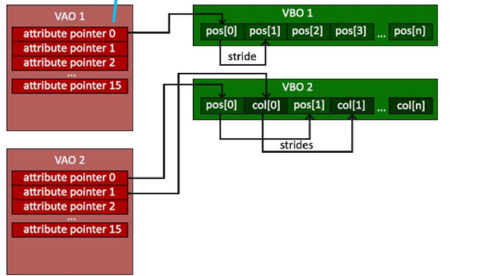
- *in* Variables are often Vertices, Color values, Normals
- Often contains *Uniform* Variables
  - Eg Transformation matrices and lighting parameters
- Performs calculations based on the *in/uniform* Variables
  - Eg. transforms the Vertices using the Uniforms
- Results are forwarded as *out* Variables
  - Eg transformed Vertices are forwarded to the Rasterizer

```
// Vertex Shader Inputs (built-in):
in int     gl_VertexID;
in int     gl_InstanceID;
// Vertex Shader Outputs (built-in):
out vec4 gl_Position;
out float gl_PointSize;
out float gl_ClipDistance[];
```

Vertex Shader – Example

- `layout (location = 0) in vec3 position`
  - Vertices designated with the AttribIndex 0 in the *VAO*
- `void main()`
  - as in C/C++, the main function
- `gl_Position`
  - a built-in out variable
  - gives positions of the Vertices to the pipeline
- *Later*: Transformation with matrices

```
#version 330 core

layout (location = 0) in vec3 position;

void main()
{
    gl_Position = vec4(position.x, position.y, position.z, 1.0);
}
```



GLSL in the Fragment Shader

- *in* Variables are interpolated color values, UV-coordinates, etc.
- Textures over Uniforms
  - E.g., from the UV coordinates and the texture, color values can be determined
- Results are forwarded as *out* Variable - Color value of the Pixel/Fragment.

```
// Fragment Shader Inputs (built-in):
in vec4   gl_FragCoord;
in bool   gl_FrontFacing;
in float  gl_ClipDistance[];
in vec2   gl_PointCoord;
in int    gl_PrimitiveID;
in int    gl_SampleID;
```

```
// Fragment Shader Outputs (built-in):
out vec4  gl_FragColor;                    // compatibility profile
out vec4  gl_FragData[];                   // compatibility profile
out float gl_FragDepth;
out int   gl_SampleMask[];
```

Fragment Shader - Example

- `out vec4 color`
  - Color value that is passed to the framebuffer
- `color = vec4(r,g,b,a)`
- In this example, it means that every fragment in the primitive has the color Orange `(1.0,0.5,0.2)`

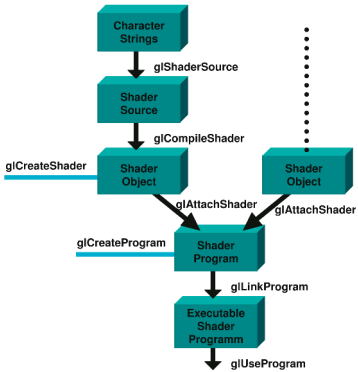
```
#version 330 core

out vec4 color;

void main()
{
    color = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

GLSL Language Construct

- Not all graphics cards know all commands
  - e.g., missing mathematical functions
- Therefore, versions are defined in the shader
  - `#version 410 // GLSL version 4.1, standard Core Profile`
  - `#version 410 GL_compatibility_profile`
  - `#version 410 GL_core_profile`

Compiling and Binding Shaders

- Compiler checks the program for errors
- Translates code into an *object code*
- Several object files are *linked*
- In OpenGL, compiler and linker are *integrated in the driver*
- Consists of several components (e.g., vertex and fragment shaders)



Compiling and Binding Shaders - Example

```
// Compile Vertexshader
GLuint vshader;
vshader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vshader, 1, &vertexShaderSource, NULL);
glCompileShader(vshader);
// Optional check for compile time errors

// Compile Fragmentshader
GLuint fshader;
fshader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fshader, 1, &fragmentShaderSource, NULL);
glCompileShader(fshader);

// Link shaders
shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vshader);
glAttachShader(shaderProgram, fshader);
glLinkProgram(shaderProgram);
// Optional check for linking errors
glDeleteShader(vshader);
glDeleteShader(fshader);
```