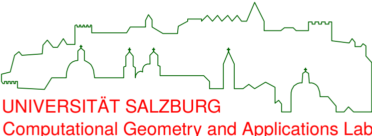
Introduction to OpenGL

Martin Held

FB Informatik
Universität Salzburg
A-5020 Salzburg, Austria
held@cs.sbg.ac.at

10. Februar 2023



Legal Fine Print and Disclaimer

To the best of our knowledge, these slides do not violate or infringe upon somebody else's copyrights. If copyrighted material appears in these slides then it was considered to be available in a non-profit manner and as an educational tool for teaching at an academic institution, within the limits of the "fair use" policy. For copyrighted material we strive to give references to the copyright holders (if known). Of course, any trademarks mentioned in these slides are properties of their respective owners.

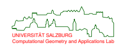
Please note that these slides are copyrighted. The copyright holder(s) grant you the right to download and print it for your personal use. Any other use, including non-profit instructional use and re-distribution in electronic or printed form of significant portions of it, beyond the limits of "fair use", requires the explicit permission of the copyright holder(s). All rights reserved.

These slides are made available without warrant of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. In no event shall the copyright holder(s) and/or their respective employers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, arising out of or in connection with the use of information provided in these slides.

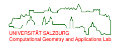
UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Table of Content

1 OpenGL



1. **OpenGL**
   - Introduction to OpenGL
   - Basic OpenGL
   - Coordinates and Transformations
   - Event-Handling and Callbacks
   - Textures
   - Loading 3D Models



1 OpenGL

- Introduction to OpenGL
  - What is OpenGL?
  - Compiling and Linking an OpenGL Program
- Basic OpenGL
- Coordinates and Transformations
- Event-Handling and Callbacks
- Textures
- Loading 3D Models

What is OpenGL?

- OpenGL stands for “Open Graphics Library”.
- Designed by Silicon Graphics Inc. (SGI) in 1991.
- Initial design in 1982 (“IRIS GL”).
- For many years, development of OpenGL had been coordinated by an Architectural Review Board (ARB).
- In 2006, the ARB and the Khronos Board of Directors voted to transfer control of the OpenGL API standard to the non-profit technology consortium Khronos Group.
- As of February 2021, the following companies were promoter members of the Khronos Group: AMD, Apple, ARM, Epic Games, Google, HUAWEI, IKEA, Imagination Technologies Group, Intel, Nvidia, Qualcomm, Samsung, Sony, Valve, VeriSilicon.
- The Khronos Group now controls
  the adaption/extension of OpenGL to reflect new hardware and software advances,

  > *“. . . to bring advanced 3D graphics to all hardware platforms and operating systems — from supercomputers to jet fighters to cell phones.”*

- Official website: https://www.opengl.org/.

What is OpenGL?

- OpenGL is a high-performance system interface to graphics hardware.
- It is the most widely used library for high-end platform-independent computer graphics; de-facto industry standard.
- It runs on different operating systems (including Unix/Linux, Windows, MacOS) without requiring changes to the source code.
- Platform-specific features can be implemented via extensions.
- OpenGL is a C Library of several hundreds of distinct functions.
- OpenGL is not object-oriented.
- Several (commercial) versions of an OpenGL library have been implemented.
- OpenGL functionality is also provided by Mesa, `http://www.mesa3d.org`, which is free. Mesa 20.x implements the OpenGL 4.6 API. (OpenGL 3.3 and Mesa 10.x would be perfectly fine for this course, though!)

What is OpenGL?

- OpenGL takes advantage of graphics hardware where it exists; whether or not hardware acceleration is used depends on the availability of suitable drivers.
- OpenGL does not come with any windowing functionality; it has to rely on additional libraries (such as GLFW).
- It ties into standard C/C++; various other language bindings exist, too. In particular, OpenGL can be used from within
  - C, C++,
  - Java,
  - Python,
  - Fortran,
  - Ada.
- OpenGL supports
  - polygon rendering,
  - texture mapping,
  - anti-aliasing,
  - shader-level operations.
- OpenGL does not provide or (directly) support high-level graphics like
  - ray tracing,
  - radiosity calculations,
  - volume rendering.

OpenGL 3.x/4.x versus OpenGL 2.x

- Note that OpenGL 1.x and 2.x differ substantially from OpenGL 3.x and OpenGL 4.x:
  - Modern OpenGL is entirely shader-based.
  - Modern OpenGL no longer relies on tons of state variables.

Be careful . . .

. . . when studying tutorials in the Web! A surprisingly large number of tutorials still teach old-style "legacy" OpenGL.

- Hint: It is old-style OpenGL if you see statements like `glBegin` or `glColor4f`.

No GLU anymore

- OpenGL 3.0 deprecated the entire Graphics Library Utilities (GLU) of "legacy" OpenGL 1.x/2.x. It was removed in OpenGL 3.1. This means that GLU will fail to work in OpenGL 3.x/4.x contexts.
- Similarly, GLUT commands like `glutSolidSphere()` do no longer work.

Sample Tutorials

- https://open.gl/:
  Requires a GPU compatible with OpenGL 3.2, CMake; uses GLFW for context and window creation and GLEW for access to newer OpenGL functions.
- http://www.opengl-tutorial.org:
  Requires a GPU compatible with OpenGL 3.3. Similar to https://open.gl/.
- LEARN OPENGL, https://learnopengl.com/:
  Similar to http://www.opengl-tutorial.org.
- http://ogldev.org/:
  Tutorials that require a GPU compatible with OpenGL 3.3.

OpenGL Libraries

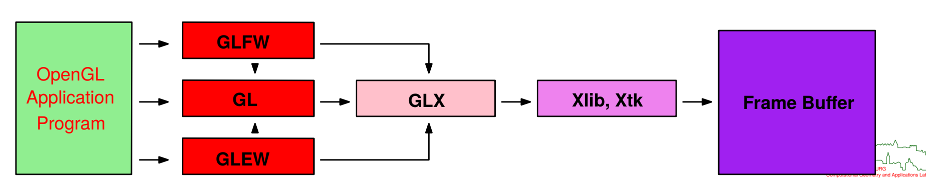
- OpenGL proper does not provide any windowing functionality! That is, it does not support opening a window or getting input from the mouse or a keyboard.
- Quote taken from the OpenGL 3.1 Specification (chapter 2, first paragraph):

  > *OpenGL is concerned only with rendering into a frame buffer (and reading values stored in that frame buffer). There is no support for other peripherals sometimes associated with graphics hardware, such as mice and keyboards. Programmers must rely on other mechanisms to obtain user input.*

- Thus, a link to the underlying windowing system (GLX for X windows, WGL for Windows, AGL for Macintosh) and an add-on library (e.g., GLFW) is required!

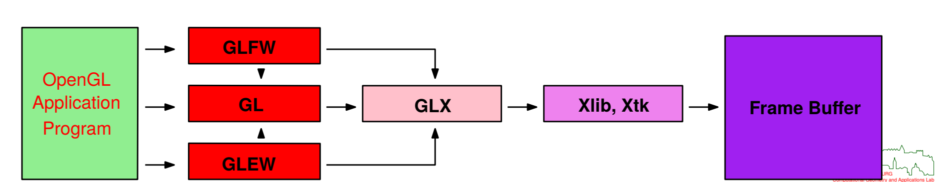
OpenGL Libraries: GLFW

- GLFW is a light-weight multi-platform library for OpenGL.
- It supports Windows (XP and later), OS X (10.7 Lion and later) and Unix-like operating systems that run the X Window System.
- Its commands start with the prefix `glfw`. E.g., `glfwInit()`.
- It can create and manage windows as well as handle standard input (via keyboard, mouse or joystick).
- It can control multiple monitors and enumerate video modes.
- In addition to portability, its single biggest advantage is its simplicity.
- Its biggest disadvantage is its lack of menus and buttons.



OpenGL Libraries: GLEW

- The OpenGL Extension Wrangler Library (GLEW) is a cross-platform library that provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.
- That is, it makes it easy to access OpenGL extensions that are available on a particular system.
- GLEW commands start with the prefix `glew`.
- Easy to use: Include `glew.h` and run `glewInit()`.



Compiling and Linking an OpenGL Program

- The source code for an OpenGL program has to contain the following directives for including OpenGL header files:
  - If GLEW is used:

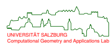
    ```
    #include <GL/glew.h>
    ```

  - If GLFW is used:

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

- Note: When compiling and linking an OpenGL program, the OpenGL header files and libraries have to be available for inclusion. This means, e.g., using the `-lgl` loader flags, and possibly, `-L` flags for the X libraries.
- Better alternative: Resort to cmake!
- See the sample files on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html.

1. **OpenGL**
   - Introduction to OpenGL
   - Basic OpenGL
     - Basic Program Structure
     - Creating an OpenGL Window and Context with GLFW and GLEW
     - Vertex Array Objects and Vertex Buffer Objects
     - Shaders
     - Index Buffer Objects
     - Data Types and Primitives
   - Coordinates and Transformations
   - Event-Handling and Callbacks
   - Textures
   - Loading 3D Models



Basic OpenGL Program Structure

```
#include <HeadersOpenGL>

int main()
{
    CreateWindow(title, width, height);
    CreateOpenGLContext(settings);

    while (windowIsOpen) {  /* event processing & drawing */
        while (event == GetNextEvent())
           HandleEvent(event);         /* e.g., handle mouse */

        UpdateScene();                 /* e.g., move objects */

        RenderScene();                /* generate next image */
        DisplayGraphics();             /* e.g., swap buffers */
    }
}
```

- Every real-time graphics application will have a program flow that boils down to this structure, no matter whether it uses OpenGL or some other library.

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Creating an OpenGL Window and Context

- We use GLFW to create an OpenGL display window.
- It comes as no surprise that you need to load the header file and initialize GLFW.

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

/* initialization of GLFW */
glfwSetErrorCallback(errorCallback);
if (glfwInit() != GLFW_TRUE) {
    fprintf(stderr, "Cannot initialize GLFW\n");
    exit(EXIT_FAILURE);
}
...
/* termination of GLFW */
glfwTerminate();
```

- GLFW error callback function:

```
static void errorCallback(int err, const char* logText)
{
    fprintf(stderr, "GLFW err %d: %s\n", err, logText);
}
```

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Creating an OpenGL Window and Context

- The `glfwWindowHint()` function is used to set some GLFW options.

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
```

- Window creation:

```
const GLuint WIDTH = 800, HEIGHT = 600;
GLFWwindow* myWindow = glfwCreateWindow(WIDTH, HEIGHT,
                                "OGL Demo", NULL, NULL);
if (myWindow == NULL) {
   fprintf(stderr, "Cannot open GLFW window\n");
   exit(EXIT_FAILURE);
}
```

- The first two parameters specify the width and height of the drawing area.
- The fourth parameter tells GLFW to use the monitor in windowed mode, and the last parameter would allow to share resources with an existing OpenGL context.
- Roughly, a context stores all of the state data associated with an instance of OpenGL. A process can create multiple OpenGL contexts, and each context can represent a separate drawing area, e.g., a window in a graphics application.

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Creating an OpenGL Window and Context

- For fullscreen mode:

```
GLFWwindow* myWindow = glfwCreateWindow(WIDTH, HEIGHT,
                          "OGL Demo",
                          glfwGetPrimaryMonitor(), NULL);
```

- Making an OpenGL context active:

```
glfwMakeContextCurrent(myWindow);
```

Creating an OpenGL Window and Context: Event-Handling Loop

- This should be enough to get an OpenGL window mapped to your screen:

```
/* event-handling and rendering loop */
while (!glfwWindowShouldClose(myWindow)) {
    /* poll events */
    glfwPollEvents();
    /* Swap buffers */
    glfwSwapBuffers(myWindow);
    /* close window upon hitting the escape key */
    if (glfwGetKey(myWindow, GLFW_KEY_ESCAPE) ==
        GLFW_PRESS)
        glfwSetWindowShouldClose(myWindow, GL_TRUE);
}
```

- The event-handling loop always needs to call `glfwSwapBuffers()` and `glfwPollEvents()`.
- You can ignore events that you do not want to handle. (We'll learn more on event handling later . . .)
- Do not forget to handle the escape key (or some other key) to return to the desktop if using the fullscreen mode.

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Creating an OpenGL Window and Context

- One technical issue remains: At runtime a graphics application needs to check which functionality is supported by a GPU, as specified in the driver provided by the vendor of the GPU, and needs to link to them.
- This is tedious and is best handled by resorting to GLEW:

```
#include <GL/glew.h>

/* initialization of GLEW */
glewExperimental = GL_TRUE;
GLenum glewStatus = glewInit();
if (glewStatus != GLEW_OK) {
   fprintf(stderr, "Error: %s\n",
            glewGetErrorString(glewStatus));
   exit(EXIT_FAILURE);
}
```

- Make sure to include `glew.h` prior to other OpenGL-related headers!
- Setting `glewExperimental` forces GLEW to use a "modern" OpenGL method for checking whether a function is available.
- See `window.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html.

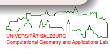
UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Vertex Buffer Object

- Classical bottleneck in pre-OpenGL 3.1: Whenever a vertex is specified in a pre-OpenGL 3.1 application, by means of `glVertex`, its coordinates need to be sent to the GPU.
- Goal: Increase performance by using the GPU rather than the CPU and by decreasing the amount of data that is exchanged between CPU and GPU.
- Basic idea:
  - We pack the vertex and attribute data into arrays.
  - A vertex array is transferred to the GPU and stored in the GPU memory.
  - Array data that is in the GPU memory can be rendered via a simple call to a callback function: `glDrawArrays()`
- This leads to vertex array objects and vertex buffer objects.

No object-oriented "object"

OpenGL is fairly liberal in its use of the word "object"! That is, an OpenGL "object" is not to be understood as an object in the object-oriented programming meaning. Rather, OpenGL objects tend to be simple arrays of data for which we get a handle (i.e., an identifier) to interact with.



Vertex Buffer Object

- OpenGL expects vertices to be stored in arrays.

```
float vtx[] = {
    0.0f,  0.0f, /* x- and y-coords of 1st vertex */
    0.5f,  0.5f, /* x- and y-coords of 2nd vertex */
    0.5f, -0.5f  /* x- and y-coords of 3rd vertex */
};
```

- A Vertex Buffer Object (VBO) is an array of data, typically floats.
- E.g., it may hold data like world coordinates, color, texture coordinates and, possibly, application-specific data.
- It will reside in the high-speed memory of the GPU.

```
GLuint myVBO;
glGenBuffers(1, &myVBO);
glBindBuffer(GL_ARRAY_BUFFER, myVBO);
```

- Since GPU memory is managed by OpenGL, you get a positive number as a reference to it.
- The `glBindBuffer()` function turns a VBO into the active buffer.

Vertex Buffer Object

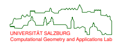
- Once an VBO is active, we can copy the vertex data to it.

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vtx),
             vtx, GL_STATIC_DRAW);
```

- Depending on the intended type of use, the last argument of `glBufferData()` determines the kind of GPU memory (relative to writing and drawing speed) in which the data is stored:

  `GL_STATIC_DRAW`: Generated once, no changes, drawn many times.
  `GL_DYNAMIC_DRAW`: Changed a few times, drawn many times.
  `GL_STREAM_DRAW`: Changed and drawn many times.

- We can store more than just the 2D or 3D coordinates of points in a VBO. E.g., store 2D texture coordinates or 3D normals.
- Hence, it is likely that most of the VBOs will consist of arrays of two and three floats.



Vertex Array Object

- A Vertex Array Object (VAO) is used to tell OpenGL how the VBO is arranged. E.g., it might be divided into variables of two floats each.
- That is, a VAO is not the actual object storing the data, but a descriptor of the data.

```
GLuint myVAO;
glGenVertexArrays(1, &myVAO);
glBindVertexArray(myVAO);
```

- Once a VAO has been bound, every call to `glVertexAttribPointer()` will cause the attributes associated with a VBO to be stored in that VAO.

Warning

Only attribute bindings performed after binding a VAO refer to it! Thus, make sure to bind an VAO at the start of your code!

Shader Programs

- GPU-based rendering is envoked through so-called *shader programs*.
- An application sends data to the GPU, and the GPU does all the rendering.
- Starting with OpenGL 3.1, OpenGL is entirely shader-based:
  - The state model is replaced by a data-flow model.
  - Several pre-OpenGL 3.1 functions are deprecated, and backwards compatibility is not required. (At least not in Core Mode.)
  - No default shaders — but each application has to provide at least a vertex and a fragment shader.
- Vertex Shader:
  - Processes input vertices (e.g., of triangles) individually.
  - Influences the attributes of a vertex, e.g., position, color, and texture coordinates.
  - Performs the perspective transformation.
- Fragment Shader:
  - Calculates individual fragment colors.
  - E.g., it might sample a texture or simply output a color.
  - It may also be used for lighting and for creating advanced effects like bump-mapping effects.
- More shaders (e.g., Tesselation and Geometry Shaders) added with OpenGL 4.1.

Shader Programs: Execution Pipeline

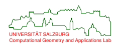
vertex data (arrays) → vertex shader → tesselation shaders → geometry shader → shape assembly → culling, clipping → rasterization → fragment shader → tests and blending

- The geometry shader is optional. It can discard, modify or pass through primitives, or even generate new ones. E.g., it could generate squares out of input vertex data.
- In the shape assembly, the GPU forms *primitives* (e.g., triangles, line segments) out of the vertices. Up to this stage, all operations are carried out on the vertices.
- Rasterization converts the primitives into pixel-sized *fragments*.

OpenGL Shading Language

- Shaders are written in the OpenGL Shading Language (GLSL).
- The GLSL is C/C++-like with overloaded operators.
- New data types (e.g., matrices, vectors) and C++-like constructors.
  E.g., `vec3 myVec=vec3(1.0,0.0,1.0)`.
- Similar in use to NVIDIA's Cg and Microsoft's HLSL.
- GLSL code is sent to the shaders as source code.
- New OpenGL functions added to compile and link that code and to exchange information with the shaders.
- Shaders can be written inside the C/C++ code, or can be stored in files and loaded.

- We will only discuss vertex shader and fragment shader very briefly.

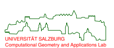


OpenGL Shading Language: Sample Vertex Shader

- Since the triangle in our sample is already given by 2D vertices, a vertex shader can be fairly simple.

```
/* define the vertex shader */
const char* vertexShaderSource = GLSL(
   in vec2 position;
   void main() {
      gl_Position = vec4(position, 0.0f, 1.0f);
   }
);
/* compile the vertex shader */
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource,
               NULL);
glCompileShader(vertexShader);
```

- Since we deal with homogeneous coordinates, the last argument of `gl_Position()` will, in general, be 1.0 ....



OpenGL Shading Language: Sample Vertex Shader

Warning

No error will be reported by `glGetError()` if a shader fails to compile!

- Hence, make sure to check explicitly!

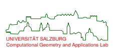
```
/* check whether the vertex shader has compiled */
GLint status;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);
if (status != GL_TRUE) {
   fprintf(stderr, "Vertex shader did not compile\n");
   char vertexCompilerLog[512];
   glGetShaderInfoLog(vertexShader, 512, NULL,
                      vertexCompilerLog);
   fprintf(stderr, "%s", vertexCompilerLog);
   exit(EXIT_FAILURE);
}
```

OpenGL Shading Language: Sample Fragment Shader

- For simplicity, we'll draw the triangle entirely red and get the following simple fragment shader.

```
/* define and compile the fragment shader: */
/* we'll get a red triangle                */
const char* fragmentShaderSource = GLSL(
   out vec4 outColor;
   void main() {
      outColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);
   }
);

GLuint fragmentShader = glCreateShader(
                                    GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource,
               NULL);
glCompileShader(fragmentShader);
```



Creating a Shader Program

- We form a shader program by linking the vertex and fragment shader into one unit.

```
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
```

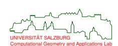
- To make the shader program active, we use the following statement:

```
glUseProgram(shaderProgram);
```

Creating a Shader Program

- Again, no error checking is done by OpenGL! Hence, make sure to check whether linking the shader program worked.

```
bool checkShaderProgramLinkStatus(GLuint programID)
{
    GLint status;
    glGetProgramiv(programID, GL_LINK_STATUS, &status);
    if(status == GL_FALSE) {
        GLint length;
        glGetProgramiv(programID, GL_INFO_LOG_LENGTH,
                       &length);
        GLchar* log = new char[length + 1];
        glGetProgramInfoLog(programID, length, &length,
                            &log[0]);
        fprintf(stderr, "%s", log);
        return false;
    }
    return true;
}
```



Specifying the Vertex Atributes

- We obtain a reference to the position input in the vertex shader — in our example this will be 0 — and then use `glVertexAttribPointer()` to specify how the input data is organized:

```
const char* attrName = "position";
GLint posAttrib = glGetAttribLocation(shaderProgram,
                                       attrName);
if (posAttrib == -1) {
    fprintf(stderr, "Error for attrib %s\n", attrName);
    exit(EXIT_FAILURE);
}
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE,
                      2*sizeof(float), 0);
```

- The arguments of `glVertexAttribPointer()` are as follows:
  1. Reference to the input.
  2. Number of values for that input, i.e., components ot `vec`.
  3. Type of each component.
  4. Normalization to [-1.0,1.0] requested?
  5. *Stride*: How many bytes are between every position attribute in the array?
  6. *Offset*: Byte offset for the first component of the first attribute.

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Drawing the Sample Triangle

- Finally, we set the background to black and draw the triangle.

```
/* set the window background to black */
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
   glClear(GL_COLOR_BUFFER_BIT);
/* draw the triangle */
   glDrawArrays(GL_TRIANGLES, 0, 3);
```

- The call `glClearColor(0.0,0.0,1.0,0.0)` would set the background color to "no red, no green, maximum blue". (The fourth parameter pertains to blending, and can be ignored for the moment.)
- Each argument is a floating-point value in the range $[0, 1]$, specifying the amount of red, green and blue.
- The arguments of `glDrawArrays()` are as follows:
  1. Type of primitives to be rendered.
  2. How many vertices shall be skipped at the beginning.
  3. Number of vertices. (Not the number of primitives!)

Cleaning Up in the End

- Do not forget to release all resources in the end:

```
glDeleteProgram(shaderProgram);
glDeleteShader(fragmentShader);
glDeleteShader(vertexShader);
glDeleteBuffers(1, &myVBO);
glDeleteVertexArrays(1, &myVAO);
```

- See `drawing.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html.

Drawing a Colorful Triangle

- We will now modify the sample code to draw a colorful triangle, by assigning the RGB values for red, green and blue to the vertices; see `colored_tri.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html:

```
float vtx[] = {
    0.0f,  0.0f, 1.0f, 0.0f, 0.0f, /* coords, red   */
    0.5f,  0.5f, 0.0f, 1.0f, 0.0f, /* coords, green */
    0.5f, -0.5f, 0.0f, 0.0f, 1.0f  /* coords, blue  */
};
```

- Modified vertex shader:

```
const char* vertexShaderSource = GLSL(
    in vec2 position;
    in vec3 colorVtxIn;
    out vec3 colorVtxOut;
    void main() {
        colorVtxOut = colorVtxIn;
        gl_Position = vec4(position, 0.0, 1.0);
    }
);
```

Drawing a Colorful Triangle

- Modified fragment shader:

```
const char* fragmentShaderSource = GLSL(
   in vec3 colorVtxOut;
   out vec4 outColor;
   void main() {
      outColor = vec4(colorVtxOut, 1.0f);
   }
);
```

- Modified bindings:

```
GLint posAttrib = glGetAttribLocation(shaderProgram,
                                        "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE,
         5*sizeof(float), 0);
GLint colAttrib = glGetAttribLocation(shaderProgram,
                                        "colorVtxIn");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
         5*sizeof(float), (void*)(2*sizeof(float)));
```

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Index Buffer Object

- Typically, geometric objects will reuse vertices. E.g., the two triangles of a rectangle share two vertices, and it is a waste of precious GPU memory to store them twice.
- Index Buffer Objects (IBOs) are applied for re-using vertex data:
- We model a rectangle formed by five vertices and four triangles:

```
float vtx[] = {
   -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, /* lower-left corner  */
    0.5f, -0.5f, 0.0f, 1.0f, 0.0f, /* lower-right corner */
    0.5f,  0.5f, 0.0f, 0.0f, 1.0f, /* upper-right corner */
   -0.5f,  0.5f, 1.0f, 1.0f, 1.0f, /* upper-left corner  */
    0.0f,  0.0f, 0.0f, 0.0f, 0.0f  /* center             */
};
GLuint idx[] = {
   0, 1, 4,
   1, 2, 4,
   2, 3, 4,
   3, 0, 4
};
```

Index Buffer Object

- Creation of an IBO:

```
/* generate one Index Buffer Object */
GLuint myIBO;
glGenBuffers(1, &myIBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, myIBO);
/* copy the element data to it */
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(idx), idx,
             GL_STATIC_DRAW);
```

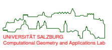
- For drawing we use `glDrawElements()` rather than `glDrawArrays()` in the rendering loop:

```
glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, 0);
```

- The arguments of `glDrawElements()` are as follows:
  1. Type of primitives to be rendered.
  2. Number of element indices. (Not the number of triangles!)
  3. Type of element indices.
  4. Offset.
- See `colored_quad.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html.
- Note that the square appears to be a quad in the graphics window: distortion!

OpenGL Naming Conventions

- All OpenGL functions have the prefix `gl`, followed by one or more capitalized words to denote the function. E.g., `glBindBuffer()`.
- GLEW and GLFW use the same scheme for naming their functions.
- Recall that OpenGL is C-based and, thus, does not have function overloading.
- As consequence, suffixes after the main part of a function name are used for prodiving information on the specific number and type of arguments that a function accepts. E.g.:
  - `glUniform2f()` indicates that this function takes two parameters (in addition to its standard arguments) which are of type `GLfloat`.
  - `glUniform2fv()` indicates that these two floats are passed as a one-dimensional array rather than two individual parameters.
- All OpenGL constants begin with `GL` and use underscores to separate words. E.g., `GL_STATIC_DRAW`.



OpenGL Data Types

Data Type	Min. Prec.	Description	Suffix
`GLbyte`	8 bits	signed integer	b
`GLubyte`	8 bits	unsigned integer	ub
`GLshort`	16 bits	signed integer	s
`GLushort`	16 bits	unsigned integer	us
`GLsizei`	32 bits	integer size	i
`GLint`	32 bits	signed integer	i
`GLuint`	32 bits	unsigned integer	ui
`GLenum`	32 bits	enumeration type	ui
`GLfloat`	32 bits	floating-point value	f
`GLclampf`	32 bits	floating-point value clamped to [0.0, 1.0]	f
`GLdouble`	64 bits	floating-point value	d
`GLclampd`	64 bits	floating-point value clamped to [0.0, 1.0]	d

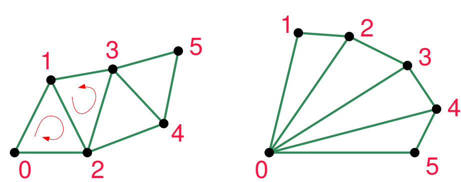
- An OpenGL implementation must use at least the minimum number of bits specified. It may use more bits than the minimum number required to represent a GL type.
- An OpenGL data type may but need not match the "corresponding" C data type in a specific implementation.
- Thus, use the OpenGL types to assure portability!

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

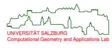
OpenGL Graphical Primitives

OpenGL Primitive	Description	Min. #(vertices)
GL_POINTS		1
GL_LINES		2
GL_LINE_STRIP		2
GL_LINE_LOOP		2
GL_TRIANGLES		3
GL_TRIANGLE_STRIP		3
GL_TRIANGLE_FAN		3

- Make sure to pay close attention to how vertices are grouped for the strips and fans.



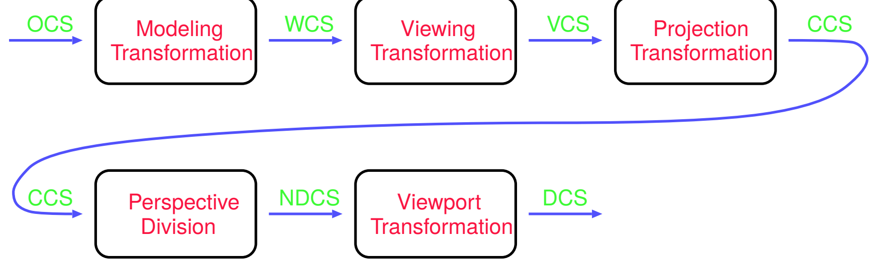
1. **OpenGL**
   - Introduction to OpenGL
   - Basic OpenGL
   - Coordinates and Transformations
     - OpenGL Coordinate Systems
     - Coordinate Transformation Pipeline
     - Handling Transformations within OpenGL
     - Camera and View Transformation
     - Projections
   - Event-Handling and Callbacks
   - Textures
   - Loading 3D Models



OpenGL Coordinate Systems

- The units of the coordinates of a vertex depend on the application; those coordinates are called *world coordinates*.
- A standard right-handed coordinate system is assumed for the world coordinates: the positive x-axis is to your right, the positive y-axis is up and the positive z-axis points out of the screen towards you.
- Internally, OpenGL will convert to *camera coordinates* and later to *window coordinates*.
- OpenGL's camera is placed at the origin pointing in the negative z-direction of the world coordinate system.
- The camera cannot be moved. Rather, one has to apply an inverse transformation to the scene to be rendered.
- OpenGL supports the definition of a *viewing volume*: Only (those portions of) objects that are inside this 3D region will be drawn ("*clipping*").

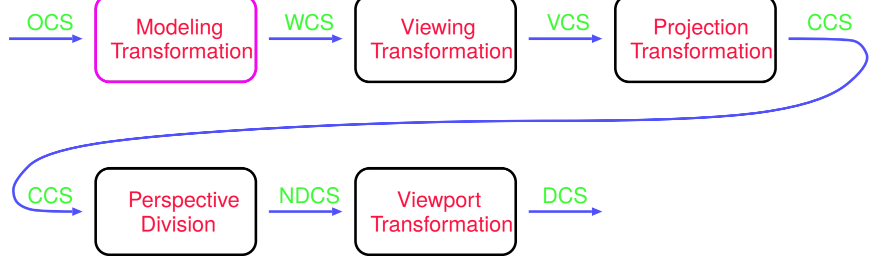
OpenGL Coordinate Transformation Pipeline



- The coordinates of a 3D point p undergo several transformations until eventually a pixel on the screen corresponding to its 2D equivalent p' is set.
- This sequence of transformations is encoded in the *OpenGL transformation pipeline*: from object coordinate system (OCS) to world coordinate system (WCS), viewing coordinate system (VCS), clipping coordinate system (CCS), normalized device coordinate system (NDCS), and finally to device coordinate system (DCS).

  $p' = \mathbf{M}_{\text{proj}} \cdot \mathbf{M}_{\text{view}} \cdot \mathbf{M}_{\text{model}} \cdot p$

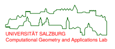
OpenGL Coordinate Transformations: Modeling Transformation



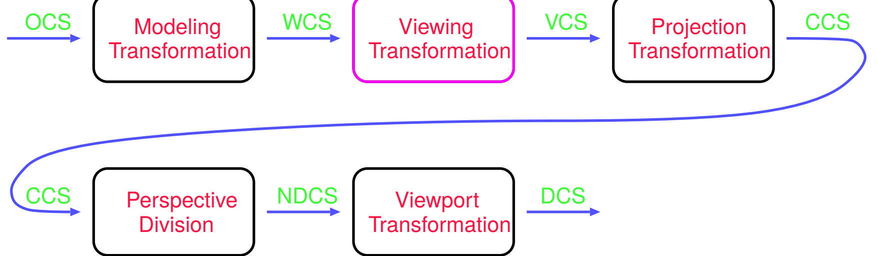
- The modeling transformation places an object somewhere in the world. Typically, this re-positioning of an object is carried out by
  1. scaling it,
  2. rotating it, and
  3. translating it.

$$p_{WCS} = \mathbf{M}_{\text{model}} \cdot p_{OCS}, \quad \text{with } \mathbf{M}_{\text{model}} := \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}.$$

- It can be as simple as an identity transformation, though.

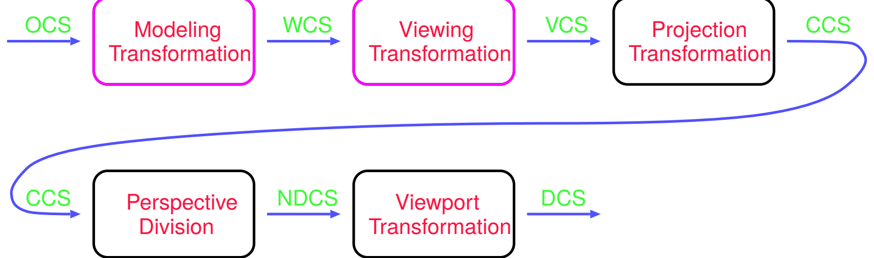


OpenGL Coordinate Transformations: Viewing Transformation

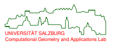


- Suppose that position and orientation of a camera are specified in world coordinates as a frame $[p, < a, b, v >]$, where p is the position and $< a, b, v >$ form a coordinate system such that a points right and b points up in a plane parallel to the image plane, and v is the direction of viewing.
- The viewing transformation $\mathbf{M}_{\text{view}}$ transforms the world coordinate system such that p becomes the origin, b points into the y-axis and v points into the negative z-axis. (Recall that the actual OpenGL camera is not moved!)

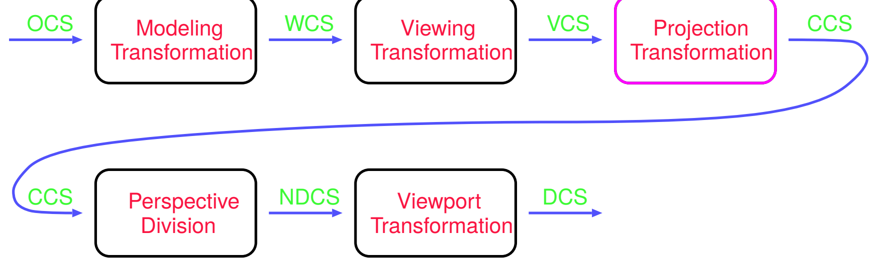
OpenGL Coordinate Transformations: Model View Transformation



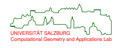
- Modeling transformation and viewing transformation combined are called *model view transformation*.
- Older versions of OpenGL forced the user to resort to model view transformations.



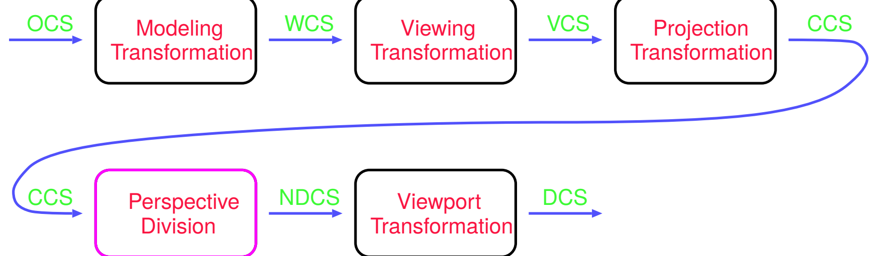
OpenGL Coordinate Transformations: Projection Transformation



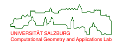
- The projection transformation transforms the world such that the viewing volume specified by the camera is mapped into an axis-aligned box as a canonical viewing volume.
- This is the earliest time that clipping can be implemented (in hardware) in a camera-independent manner.



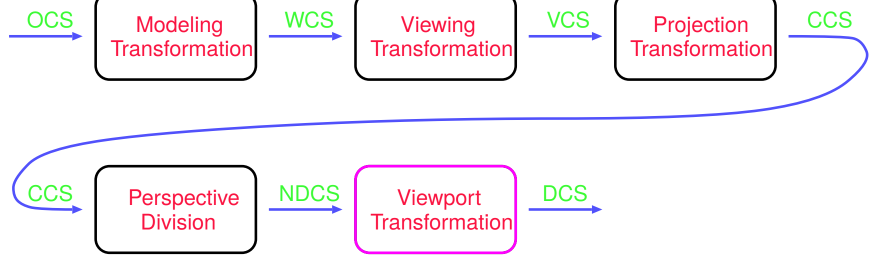
OpenGL Coordinate Transformations: Perspective Division



- The perspective division maps an axis-aligned viewing box to the axis-aligned cube $[-1, 1]^3$.
- The projection transformation and the perspective division are carried out as one functional unit by OpenGL, in dependence on the type of perspective.



OpenGL Coordinate Transformations: Viewport Transformation



- In the final viewport transformation, OpenGL uses information obtained from the graphics window or the parameters of

  ```
  glViewport(x, y, width, height);
  ```

  where
  - (x, y) is the location of the lower-left corner of the viewport, and
  - `width` and `height` are its dimensions,

  to map NDC to screen coordinates (DC).
- All arguments of `glViewport()` are specified in pixels.

Handling Transformations within OpenGL: GLM

- OpenGL (internally) uses homogeneous coordinates and 4×4 matrices to carry out transformations.
- There are several ways to handle the math related to transformations ...
- I find it easiest to employ GLM, the OpenGL Math library.
- It is a headers-only library and provides vector and matrix classes for handling the math of (likely) all the transformations that you will need, including support for quaternions.
- Since it is based on the GLSL specifications, it ties into GLSL neatly.
- Usage:

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

  - The second header file provides functions that make the computation of transformation matrices easy.
  - The third file is used for converting an GLM matrix into an array of floats (for usage by OpenGL proper).
- Make sure to get a recent version of GLM in order to avoid tons of warnings about deprecated functions.

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Handling Transformations within OpenGL: GLM Constructors

- If a single scalar parameter is given to a vector constructor, it is used to initialize all components of the vector to that value:

```
glm::vec4 Position = glm::vec4(glm::vec3(0.1), 1.0);
```

- If a single scalar parameter is given to a matrix constructor, then all diagonal elements will be set to the value, and all other elements will be set to `0.0f`:

```
glm::mat4 Model = glm::mat4(1.0);
```

Sample Model Transformation

- We rotate our colored square by 45° (around the z-axis):

```
/* define a model-view transformation */
glm::mat4 model = glm::mat4(1.0);
model = glm::rotate(model, glm::radians(45.0f),
                    glm::vec3(0.0f, 0.0f, 1.0f));
```

The first command gives us a 4×4 unit matrix, and the second command multiplies this matrix by a rotation around the z-axis.

Degree vs. radians

Some versions of GLM take the angle in degrees instead of radians. We force radians by means of `#define GLM_FORCE_RADIANS` and use `glm::radians(degrees)`.

- The next step is to instruct the shader program to apply this model transformation to every vertex:

```
const char* uniformName;
uniformName = "model";
/* pass the model matrix to the shader program */
GLint uniformModel = glGetUniformLocation(shaderProgram,
                                          uniformName);
if (uniformModel == -1) {
   fprintf(stderr, "Error: could not bind uniform %s\n",
                   uniformName);
   exit(EXIT_FAILURE);
}
glUniformMatrix4fv(uniformModel, 1, GL_FALSE,
                   glm::value_ptr(model));
```

The first parameter of `glUniformMatrix4fv()` is the handle of the matrix, the second parameter specifies the number of matrices, the third parameter concerns transposing of the matrix prior to its use, and the `glm::value_ptr()` function in the fourth parameter converts the matrix into 16 floats.

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Sample Model Transformation

- We do also have to modify the code for the vertex shader:

```
/* vertex shader with model-view matrix added */
const char* vertexShaderSource = GLSL(
    in vec2 position;
    in vec3 colorVtxIn;
    out vec3 colorVtxOut;
    uniform mat4 model;
    void main() {
        colorVtxOut = colorVtxIn;
        gl_Position = model * vec4(position.x, position.y,
                                    0.0, 1.0);
    }
);
```

- See `transformed_quad.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html.

Sample Model Transformation

- We now make the quad spin around the origin in a continuous fashion by adding an animation matrix `anim` in the vertex shader and in the event-handling loop:

```
const char* vertexShaderSource = GLSL(
   in vec3 position;
   in vec3 colorVtxIn;
   uniform mat4 anim;
   uniform mat4 model;
   out vec3 colorVtxOut;
   void main() {
      colorVtxOut = colorVtxIn;
      gl_Position = anim * model * vec4(position, 1.0);
   }
);
```

Sample Model Transformation

- Definition and binding of animation matrix:

```
/* define a transformation matrix for the animation */
glm::mat4 anim = glm::mat4(1.0f);
uniformName = "anim";
GLint uniformAnim = glGetUniformLocation(shaderProgram,
    uniformName);
glUniformMatrix4fv(uniformAnim, 1, GL_FALSE, glm::
    value_ptr(anim));
```

- Animation matrix in the event-handling loop, prior to the actual draw command:

```
/* make the quad spin around */
anim = glm::rotate(anim, glm::radians(0.1f),
                   glm::vec3(0.0f, 0.0f, 1.0f));
glUniformMatrix4fv(uniformAnim, 1, GL_FALSE,
                   glm::value_ptr(anim));
```

- Of course, a suitable value for the angular increment depends on the speed of your GPU.
- And, of course, this is a brute-force way to keep an object spinning . . .
- See `spinning_quad.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html.

Camera and View Transformation

- We now add matrices for the view and projection transformations.
- It is easiest to use GLM's `glm::lookAt()` function to position the camera:

```
/* define a view transformation */
glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 2.0f),
                             glm::vec3(0.0f, 0.0f, 0.0f),
                             glm::vec3(0.0f, 1.0f, 0.0f));
```

- Parameters:
  - the first parameter specifies the position of the camera,
  - the second parameter specifies a point towards the camera is aiming, and
  - the third parameter specifies a vector that is pointing up.

  In our case, this is a trivial view onto the *xy*-plane.

Caveat

Re-positioning the camera causes the appropriate reverse transformation to be applied to the model. (The OpenGL-internal camera always stays at the origin!) This transformation can cause parts or all of the objects to become invisible if the viewport is not changed appropriately.

Camera and View Transformation

- Of course, we do have to pass the view and projection matrices to the shader:

```
/* pass the view matrix to the shader program */
GLint uniformView = glGetUniformLocation(shaderProgram,
                                          "view");
glUniformMatrix4fv(uniformView, 1, GL_FALSE,
                   glm::value_ptr(view));
/* define a currently trivial projection transformation
    */
glm::mat4 proj = glm::mat4(1.0f);
/* pass the projection matrix to the shader program */
GLint uniformProj = glGetUniformLocation(shaderProgram,
                                          "proj");
glUniformMatrix4fv(uniformProj, 1, GL_FALSE,
                   glm::value_ptr(proj));
```

- We now use the following sample setting, see `mvp_quad.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html:

```
/* quad consisting of four triangles in the plane z=1 */
float vtx[] = {
    -0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 0.0f,
     0.5f, -0.5f, 1.0f, 0.0f, 1.0f, 0.0f,
     0.5f,  0.5f, 1.0f, 0.0f, 0.0f, 1.0f,
    -0.5f,  0.5f, 1.0f, 1.0f, 1.0f, 1.0f,
     0.0f,  0.0f, 1.0f, 0.0f, 0.0f, 0.0f
};
/* vertex shader */
const char* vertexShaderSource = GLSL(
  in vec3 position;  /* quad is in 3D! */
  uniform mat4 model; uniform mat4 view; uniform mat4 proj;
  in vec3 colorVtxIn; out vec3 colorVtxOut;
  void main() {
      colorVtxOut = colorVtxIn;
      gl_Position = proj*view*model * vec4(position,1.0);
  }
);
```

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

- Of course, with a modified definition of `vtx[]`, we also have to adapt `glVertexAttribPointer()` accordingly:

```
glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE,
                      6*sizeof(float), 0);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
                      6*sizeof(float),
                      (void*)(3*sizeof(float)));
```

- We put the camera at $(0, 0, 2)$:

```
glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 2.0f),
                             glm::vec3(0.0f, 0.0f, 0.0f),
                             glm::vec3(0.0f, 1.0f, 0.0f));
```

- We will continue with discussing different ways to project our 3D scene to 2D, again using functions provided by GLM.

Multiplication order

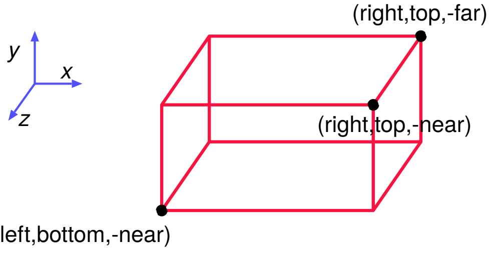
Matrix multiplication is not commutative. Watch the order of your matrices!

- The viewing volume of an orthographic projection is set up by the following GLM command:

```
glm::mat4 proj = glm::ortho(-2.0f, 2.0f,   /* left/right  */
                            -1.5f, 1.5f,   /* top/ bottom */
                             0.5f, 1.5f);  /* near/far    */
```

Camera's View!

Note that `near` and `far` are specified as seen from the camera!



- For a camera positioned at the origin, a point with world coordinates (x, y, z) is rendered if and only if

$$\begin{aligned} \texttt{left} &\le x \le \texttt{right}, \\ \texttt{bottom} &\le y \le \texttt{top}, \\ -\texttt{far} &\le z \le -\texttt{near}. \end{aligned}$$

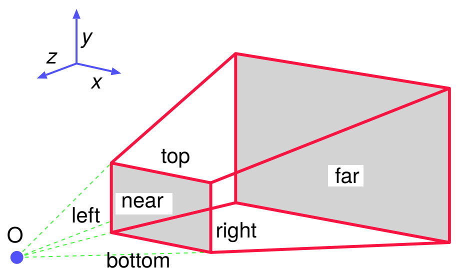
Perspective Projection: Frustum

- The viewing volume of a perspective projection can be set up by the following GLM command:

```
glm::mat4 proj = glm::frustum(-2.0f,2.0f, /* left/right   */
                              -1.5f,1.5f, /* top/ bottom */
                               0.9f,1.1f);/* near/far     */
```

Camera's View!

Note: `near` and `far` are positive and specified as seen from the camera!

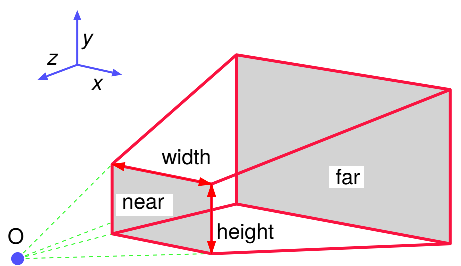


Perspective Projection: Field-of-View

- The viewing volume of a perspective projection can also be specified in a more intuitive manner (with 1.3 in radians being roughly 75°):

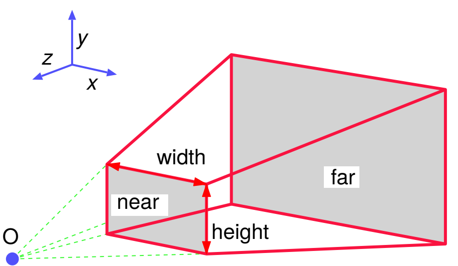
```
glm::mat4 proj = glm::perspective(1.3f,       /* angle  */
                                  4.0f/3.0f,  /* aspect */
                                  0.9f,       /* near   */
                                  1.1f);      /* far    */
```

- We have $\texttt{height} = 2 \cdot \texttt{near} \cdot \tan(\texttt{angle}/2)$, and $\texttt{aspect} = \texttt{width}/\texttt{height}$.
- Again, `near` and `far` both are positive terms that indicate the distance from the camera!



Perspective Projection: Field-of-View

- The `angle` specifies the field-of-view angle in the y-direction.
- The availability of the aspect ratio in `glm::perspective()` makes it easier to respond adequately to a reshaping of the graphics window.
- Decreasing `angle` without moving the objects or changing the camera position corresponds to switching from a wide-angle lens to to a telephoto lens, i.e., to zooming in.
- Increasing `angle` corresponds to zooming out.
- Re-positioning the camera causes the appropriate reverse transformation to be applied to the objects. This transformation can cause parts or all of the objects to become invisible if `near` and `far` are not also changed appropriately.
- Recall that distortion may occur if the aspect ratios do not match!



1 **OpenGL**
- Introduction to OpenGL
- Basic OpenGL
- Coordinates and Transformations
- Event-Handling and Callbacks
  - Keyboard
  - Mouse
- Textures
- Loading 3D Models

Event Handling: GLFW Callback Functions

- As virtually all other graphics APIs, OpenGL/GLFW also handle *events* (such as the pressing of a mouse button or a keystroke) by means of *callback functions*.
- Roughly, an OpenGL program runs in a loop, polling the hardware for new events, and calling callback functions for those events for which callback functions were declared. (All other events are ignored!)
- Each callback function has to be *registered*. The command

```
glfwSetXXXCallback(myWindow, YYY);
```

  tells OpenGL to use the function `YYY()` as callback function for events related to `XXX`.
- GLFW provides two ways to check for events:
  - `glfwPollEvents()` continually checks for events and processes events upon receipt.
  - `glfwWaitEvents()` puts the thread that runs the OpenGL program to sleep until at least one event has been received.
- A decent OpenGL program will always offer a way to terminate it gently, e.g., by reacting appropriately if “Q(uit)” or “ESC” was pressed by a user.
- See GLFW’s documentation of input handling for more details.

Event Handling: Keyboard Input

- GLFW recognizes two types of events related to keys — key events and character events (related to Unicode code) — but we will focus only on key events.
- The following callback function instructs OpenGL to close the window if the user has pressed “Q”, “q” or “ESC”:

```
static void keyCallback(GLFWwindow* myWindow, int key,
                        int scanCode, int action, int mod)
{
   if (((key == GLFW_KEY_ESCAPE) || (key == GLFW_KEY_Q)) &&
       (action == GLFW_PRESS))
      /* close window upon hitting the ESC key or Q/q */
      glfwSetWindowShouldClose(myWindow, GL_TRUE);
}
```

- Here, the `scancode` is system-specific stuff, and `mod` is a bit field describing which modified keys were held down. E.g., `GLFW_MOD_SHIFT`, `GLFW_MOD_CONTROL`.
- We register this callback function by using the following command:

```
glfwSetKeyCallback(myWindow, keyCallback);
```

- Key and button actions are `GLFW_RELEASE`, `GLFW_PRESS` and `GLFW_REPEAT`. (The last action means that a key was held pressed until it repeated.)

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Event Handling: Mouse Input

- Whenever the mouse cursor is moved, a callback is triggered and the current position is passed to a callback function (if registered):

```
static void cursorPosCallback(GLFWwindow* window,
                              double x_pos,
                              double y_pos)
{
   printf("Mouse is at (%6.1f,%6.1f)\n", x_pos, y_pos);
}
glfwSetCursorPosCallback(myWindow, cursorPosCallback);
```

- The coordinates can be converted to integers with the floor function.
- One can also query the cursor coordinates directly:

```
double x_pos, y_pos;
glfwGetCursorPos(window, &x_pos, &y_pos);
```

Mouse coordinates . . .

. . . have their origin at the upper-left corner of the window!

Event Handling: Mouse Input

- An enter/leave callback provides notification when the mouse cursor enters or leaves a window:

```
static void cursorEnterCallback(GLFWwindow* myWindow,
                                int entered)
{
   if (entered) printf("Cursor entered window!\n");
   else         printf("Cursor left window!\n");
}
glfwSetCursorEnterCallback(myWindow, cursorEnterCallback
    );
```

- A scroll callback notifies about scrolling:

```
static void scrollCallback(GLFWwindow* myWindow,
                           double x_off, double y_off)
{
   printf("Scrolled by (%6.1f,%6.1f)\n", x_off, y_off);
}
glfwSetScrollCallback(myWindow, scrollCallback);
```

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

- A mouse button callback provides information on button presses and releases:

```
static void mouseButtonCallback(GLFWwindow* myWindow,
                                int button, int action,
                                int mods)
{
    if ((button == GLFW_MOUSE_BUTTON_LEFT) &&
        (action == GLFW_PRESS)) {
        double x_pos, y_pos;
        glfwGetCursorPos(myWindow, &x_pos, &y_pos);
        printf("Lft mouse button pressed (%6.1f,%6.1f)\n",
               x_pos, y_pos);
    }
}
glfwSetMouseButtonCallback(myWindow, mouseButtonCallback
    );
```

- See `callbacks_quad.cc` on https://www.cosy.sbg.ac.at/~held/teaching/einfuehrung_graphik/cg.html.

1 **OpenGL**
- Introduction to OpenGL
- Basic OpenGL
- Coordinates and Transformations
- Event-Handling and Callbacks
- Textures
  - Loading a Texture
  - Using a Texture
  - Adapting the Shaders
- Loading 3D Models

Loading a Texture Image

- OpenGL does not directly support the loading of textures. Rather one has to resort to third-party libraries.
- Up to version 2.0, GLFW allowed to load some types of texture files. However, this feature has been removed from GLFW 3.0.
- We resort to SOIL, the Simple OpenGL Image Library, for loading images:

```
/* load texture image */
GLint texWidth, texHeight;
GLint channels;
unsigned char* texImage = SOIL_load_image("katze.png",
                                          &texWidth,
                                          &texHeight,
                                          &channels,
                                          SOIL_LOAD_RGB);
if (texImage == NULL) {
   fprintf(stderr, "Image file could not be loaded\n");
   exit(EXIT_FAILURE);
}
```

- SOIL also offers `SOIL_load_OGL_texture` but this function dates back to 2008 and uses features that are not supported by modern OpenGL.

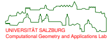
UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

Generating a Texture

- Once the image has been loaded, we can generate the texture:

```
GLuint textureID;
glActiveTexture(GL_TEXTURE0); /* texture unit 0 */
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D, textureID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight,
             0, GL_RGB, GL_UNSIGNED_BYTE, texImage);
SOIL_free_image_data(texImage);
```

- The function `glActiveTexture()` specifies which texture unit a texture object is bound to when `glBindTexture()` is called. (Unit 0 is default.)
- Parameters of `glTexImage2D`:
  1. Texture target.
  2. Level-of-detail, with 0 being the base image. Can be used for mipmaps.
  3. Internal pixel format to be used by GPU.
  4. Width of texture image.
  5. Height of texture image.
  6. According to the specification, it should always be 0 ...
  7. Format of the pixels in the image array.
  8. Format of the pixels in the image array.
  9. Image array.



Using a Texture

- Texture coordinates — denoted by *s* and *t* for a 2D texture — are, by default, normalized and range in the interval $[0.0, 1.0]$.
- By convention, $(0.0, 0.0)$ corresponds to the lower-left corner of the texture space, and $(1.0, 1.0)$ corresponds to the upper-right corner.
- The simplest approach to supplying texture coordinates is to specify them on a per-vertex basis, as we did in the case of color for our colored quad (`colored_quad.cc`):

```
float vtx[] = {
    /*   vertex coords     texture                   */
    -0.5f, -0.5f, 1.0f, 0.0f, 0.0f, /* lower-left   */
     0.5f, -0.5f, 1.0f, 1.0f, 0.0f, /* lower-right  */
     0.5f,  0.5f, 1.0f, 1.0f, 1.0f, /* upper-right  */
    -0.5f,  0.5f, 1.0f, 0.0f, 1.0f, /* upper-left   */
     0.0f,  0.0f, 1.0f, 0.5f, 0.5f  /* center       */
};
```

- Note that sharing vertices within different faces becomes problematic once different texture coordinates are supposed to be used.

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

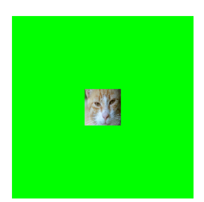
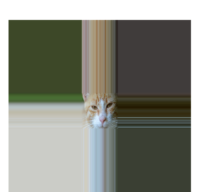
Using a Texture

- Wrapping is needed for texture coordinates that are outside of the unit square.

  GL_CLAMP_TO_BORDER: Specified color is used outside of border.
  GL_CLAMP_TO_EDGE: Texture values at the border are extended.
  GL_REPEAT: Texture image is repeated.
  GL_MIRRORED_REPEAT: Texture image is repeated in mirrored fashion.
  GL_MIRROR_CLAMP_TO_EDGE: One repetition, then clamp to edge.

- Texture options/parameters are set by means of `glTexParameter()`:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_BORDER);
GLfloat bdColor[] = { 0.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
                 bdColor);
```

Using a Texture

- It is unlikely that the resolution of the texture image will match the resolution required during rendering: Filtering:

  GL_NEAREST: Take color information from texel closest to query point.
  GL_LINEAR: Interpolate the colors of the four neighboring texels.

- Filters can be specified both for maximizing and minimizing if the pixel maps to an area smaller (greater, resp.) than one texel:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
```

- Alternatively, OpenGL can be instructed to use mipmaps. E.g.,

```
glGenerateMipmap(GL_TEXTURE_2D);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
```

Adapting the Shaders for a Texture

- We need to adapt our shaders to deal with vertices. Vertex shader:

```
const char* vertexShaderSource = GLSL(
    in vec3 position;
    in vec2 textureCoordIn;
    uniform mat4 mvp;
    out vec2 textureCoordOut;
    void main() {
        textureCoordOut = vec2(textureCoordIn.x,
                               1.0 - textureCoordIn.y);
        gl_Position = mvp * vec4(position, 1.0);
    }
);
```

- The inversion of the *y*-coordinates of the texture image is a technical twist (or hack) to deal with the problem that images tend to have their coordinate origin in the upper-left corner . . .

Adapting the Shaders for a Texture

- Modified fragment shader:

```
const char* fragmentShaderSource = GLSL(
   in vec2 textureCoordOut;
   out vec4 outColor;
   uniform sampler2D textureData;
   void main() {
      outColor = texture(textureData, textureCoordOut);
   }
);
```

- Texture uniform to be passed to shader:

```
uniformName = "textureData";
GLint uniformTex = glGetUniformLocation(shaderProgram,
                                        uniformName);
if (uniformTex == -1) {
   fprintf(stderr, "Error: could not bind uniform %s\n",
           uniformName);
   exit(EXIT_FAILURE);
}
glUniform1i(uniformTex, 0);
```

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab

1 OpenGL

- Introduction to OpenGL
- Basic OpenGL
- Coordinates and Transformations
- Event-Handling and Callbacks
- Textures
- Loading 3D Models

Loading 3D Models

- While loading 3D models is not exactly an OpenGL task, you are likely to run into it as soon as you try to build more complex scenes by resorting to models built by others.
- The *Open Asset Import Library* ("Assimp"), `http://www.assimp.org/` is a portable Open Source library to import various 3D model formats in a uniform manner.
- More recent versions of Assimp also can export 3D models, thus turning Assimp into a general-purpose 3D model converter.