Computer graphics

Linus Falk

February 27, 2023

Contents

1 Introduction to computer graphics **2**

2 Graphics programming **5**

3 Transformations **13**

4 3D viewing and projection **16**

Lecture 1: Introduction

monday 16 jan 10:15

1 Introduction to computer graphics

What can it be used for

- Entertainment
  - Games, pushing graphic card development forward..
- User interfaces
- Applications
- Data visualization
  - Information Visualization
    - Charts diagrams etc
    - Plots of N-d data after dimensionality
  - Data visualization
    - ...
- Image processing
  - Extracts information
  - produces outputs like: classifications and segmented objects
- Computer vision
  - feature tracking
  - Tracking
  - **3D reconstruction**
- Photogrammetry?
  - Reconstruct 3D models from photos taken from different viewpoints. How do the different points align. Use **CG** to view result.
- Cultural heritage?
  - preserve art works, 3D scanning and other techniques.
  - Display the result for the audience with **CG**
- CAD
  - 3D modeling
- much much more..

Digital images

Gray scale image a (raster graphics) is stored as a matrix with intensity representing each pixel.A modern computer monitor got around 2660x1600 pixels, or picture elements. Bit depth describes how many bits that are used to represent the intensity. "True color" is 24 bit, 8 bit per color channel. A colored image is represented by three color images Red Green and Blue, some times an additional channel **alpha** for storing opacity.

High dynamic range HDR images

8 bits per color channel often sufficient. In many computer graphics applications a higher dynamic range is needed. It is therefore common to use more bits per color channel, e.g. 16 or 32 bit floating point values:

- When images are manipulated, to avoid artifacts.
- When using image data in the purpose of measuring light information from a scene to realistically integrate synthetic objects

Image based lightning with high dynamic range images of real world environment.

useful to create a realistic illuminations.

To capturing a the great contrast between sky and the ground, 8 bit depth is not usually enough.

Computer graphics history

University of Utah prominent sections that did a lot of work in this are during 70s. Many methods invented by people from this sections, Sutherland, Blinn, Phong, Guoraud, Catmull (Turing awards winner), Newell and others. During the 80s we got Graphical user interfaces, Mac, Amiga and graphics in movies.

- Utah teapot
- Stanford bunny

In the 90s fully animated CG movies and special effects. 00s Rise of the GPU's, start to be able to program and do custom things on GPUs. A move toward physically based rendering, start to use more correct models of illumination thanks to better computational resources. 10s saw fusion with Image processing and computer vision.. Film rendering to **path tracing**. Real-time **ray tracing**. 20s machine learning?? this would mean new principles. Neuro .... read more of own interest

Quick look at what we will learn

- Transformation
  - Affine transformations in homogeneous coordinates
  - Orthographic and perspective projections for cameras.
- Shading

  - Going from geometry to what color to put on a surface. A more general term than illumination
  - Gouraud shading (computed per-vertex)
  - Phong cont...
- Illumination
  - The Phong reflection model
  - Blinn-Phong
- Texure mapping
  - Texture mapping
  - Bump mapping
  - Environment mapping
  - Geometry
  - Normal
- The programmable **Graphics pipeline**

Pipeline

- Input are the objects and their vertices
- The output are the pixels on the screen
- Performs:
  - transformations
  - clipping and assembly
  - shading, texturing and illumination

Conclusion for this lecture

Study the slides but also the other recourse. Prepare for the practicals (assignment and project) and start to play around with graphics programming.

tuesday 17 jan 10:15

2 Graphics programming

Typically deals with how to define 3D scene, with virtual camera and how to create a 2D projection of the 3D scene

Real time vs offline rendering

Real time used in games, scene must be updated 30-60 fps here is speed > image quality, but today hardware can do pretty much both. **offline** rendering is used for animated movies and visual effects and is not meant to be used interactively. In this field rendering of a single frame can take several hours. Image quality > speed in this case. Movie production uses clusters to calculate the renders in parallel.

How do we draw

Version 1, traditionally used mainly in offline rendering

Version 1

```
For every pixel on screen
        Query all objects to be drawn,
        to determine the correct color for that
           pixel "Ray tracing"
end
```

Version 2, most common method for real time rendering

Version 2

```
For every object
        Draw the object to the screen,
        by setting the correct colors for the
           affected pixels
end
```

Representation

Polygonal meshes are typically used for representing 3D models, using collections of vertices, edges and faces. Faces can be arbitrary polygons but we typically split them in triangles so the implementation gets simpler.

Vertex data

Each vertex in the polygonal mesh has one or several attributes such as:

- Position

- Color
- Normal vector
- texture coordinate

The vertex data is typically loaded on the CPU and uploaded memory that is accessible from the GPU via **buffer objects**.

Transforms

Transforms are used to manipulate positions orientation, size and shape of object in the 2D/3D scene. Transforms are also used to define a virtual camera and go from one coordinate system to another. Theses transformations are often represented as 3 by 3 or 4 by 4 matrices. Examples of some basic transforms:

- Translation
- Rotation
- Scaling

Coordinate systems

The OpenGL pipeline has 6 different coordinate systems

- Object
- World
- Eye (or camera)
- Clip
- Normalized devices
- Window (or screen)

Surface normal

A normal vector is a perpendicular vector to the surface that points outwards from the surface. It describes a local orientation of a surface at some vertex or face. Super important to lighting. Useful to know if the surface is pointing towards me or not

Can visualize it with RGB vector for example.

Shaders

In real time rendering is shader a small program that is compiled on the CPU and executed on the GPU. Common use is to apply transformations on vertices and compute and compute the level of light and colors of the fragments/pixel candidates. Using different shader or shader input can drastically change the visual appearance of a 3D object.

The programmable graphics pipeline

create illustrator image

OpenGL

OpenGL is a cross-platform low level API for rendering of 2D and 3D graphics. It is state-based and maintain a currents state of things (what shader is being used etc). The first version was launched in 1992. Used in many applications such as: games, simulations, scientific visualizations, CAD, mobile applications, VR etc. OpenGL only handles rendering and not any input or windowing. It is callable form many programming languages such as: C, C++, Python, Java, C#, JS, Rust and more. It comes in many variations:

- OpenGL for desktop
- OpenGL ES for embedded systems
- WebGl for web browsing 3D graphics.

Alternatives to OpenGL are: Metal (MacOS), Vulkan and DirectX (Microsoft).

OpenGL functions are executed in the host CPU application and responsible for:

- Creating and initializing buffers, shaders, textures etc.
- Upload data to GPU accessible memory.
- Configure the render state.
- Submit drew calls.
- Clear and swapping buffers.

The CPU and GPU

We here present a couple of very simplified "algorithms" for drawing a object:

"Immediate mode" rendering

```
initialize_system();
for (each object to be drawn){

        o = generate_object();
        draw_object(o);
}
cleanup();
```

"Retained mode" rendering

```
initialize_system();
for (each object to be drawn){

        o = generate_object();
        store_object(o);
}

draw_all_objects();
cleanup();
```

"Retained mode" rendering

```
initialize_system();
for (each object to be drawn){

        o = generate_object();
        store_object(o);
}
send_object_to_GPU();
//Can be a bottleneck, if we want to draw the
   same object multiple times
// it is beneficial to only send them once to the
    GPU

draw_all_objects();
cleanup();
```

Utility libraries

Since OpenGL is only used for rendering, we need to use a variety of so called utility libraries in our applications. Here are the ones we will be using in the labs:

- GLFW create and manage window.
- GLEW extension loader.
- GLM mathematics library.
- ImGui GUI.

GLFW gives us an interface between the windowing system and graphics system and allows us to create a window, executes the rendering loop and setup a mouse and keyboard interaction. **GLEW** is a cross-platform C/C++ library loading OpenGL extensions. GLEW will search for and pull in all the OpenGL extensions that we need/supported, **GLM** is mathematics library for graphical

programming and provides vector and matrix datatypes, transforms, quaternions, noise functions and much, much more. It is used in the host/CPU for C++ applications only. **ImGui** is a GUI library that is useful for tweaking rendering parameters.

Vertex buffer objects (VBOs)

VBO's are used for uploading arrays of vertex data/attributes to the GPU memory. Before we submit a draw call we must bind the VBOs. We can split our VBOs into groups of each vertex attribute or interleave several vertex attributes in one single VBO (this us usually more efficient).

Vertex array objects (VAOs)

A VAO stores references to one or several VBOs along with their states and configurations. At drawing we only have to bind the VAO, it simplifies the code so we don't have to bind and configure the VBOs separately at each draw call. Recent OpenGL core profiles require the use of VAOs

glDraqArrays and glDrawElements

These are draw commands for rendering graphics primitives: lines, points and triangles from array data stored in VBOs or VAOs.

Example:

```
void drawTriangle(GLuint program, GLuint vao)
{
  glUseProgram(program);

  glBindVertexArray(vao);
  glDrawArrays(GL_TRIANGLES, 0, 3);
  glBindVertexArray(0);
  glUseProgram(0);
}
```

OpenGL primitives

The following primitives are the basic building blocks in OpenGL applications:

- Point sprites: GL_POINTS
- Lines: GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
- Triangles: GL_TRIANGLE, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN

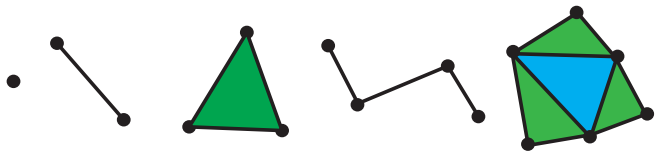

Figure 1: Example of caption

GLSL OpenGL shading language

GLSL is a high-level, cross-platform shading language for real-time rendering. It is based on the C-programming language and uses similar naming conventions, data-types and control structures. Following data-types and functions are built in:

- vector and matrix types
- various math and utility functions for graphics programming, dot, cross, max, normalize etc.
- texture lookup functions

Shader types

OpenGL support six different types of shaders. Vertex and fragment shaders are covered in this course.

- **Vertex**
- **Fragment**
- Geometry
- Tesselation control
- Tesselation evaluation
- Compute

The vertex and fragment shader

An example of the task the vertex shader have is to apply color and transformation on the input vertices and pass varying data the the fragment shader.

The fragment shader takes **uniforms** and interpolated data from the vertex shader and rasterize as input. It then computes the final color of fragments called pixel candidates by evaluating lightning equations.

The GLSL source code is typically stored in plain ASCII text files with the suffixes *.vert for vertex shaders and *.frag for fragment shaders. When the program starts the host application loads the GLSL source files into strings and then compile them into shaders that can be executed on the GPU.

Workflow - Creating, compiling and linking

The main steps in Creating, compiling and linking GLSL shaders are listed here:

1. Read vertex and framgent shaders source files into strings
2. Create vertex shader object from the vertex shader source string
3. Create fragment shader object from the fragment shader string
4. Compile the program
5. Link the compiled program and check for errors
6. De-attach and delete the shader object

Example: GLSL vertex shader

```
#version 330 // specifies the GLSL version
in vec4 a_position; // input vertex position
void main() {
  // just sets the output vertex position
  // to the input vertex position
  gl_Position = a_position;
}
50
GLSL fragment shader
(triangle.frag)
#version 330
out vec4 frag_color; // output fragment color
void main() {
  // sets the output fragment color to white
  frag_color = vec4(1.0, 1.0, 1.0, 1.0);
}
```

The variables types that are used for communicating with shaders are of three types:

- Attribute
- Varying
- Uniform

Attribute variables can be accessed via the **in** qualifier. **Varying variables** provides an interface for passing colors, normals, texture coordinates and other data between the vertex shader and the fragment shader. Varying data is be default linearly interpolated over the geometric primitive. The vertex shader uses the **out** qualifier to pass varying data the fragment shader. The fragment shader accesses the data via the **in** qualifier and writes an output via the **out** qualifier. **Uniform variables** are used for data that should be constant for all vertices and fragments. Examples of such data are: transforms, material

properties (color, opacity, etc), light sources, texture samplers, time and flags for enabling/disabling parts of the shading. **Uniform** variables can be accessed in both the vertex and fragment shader via the **uniform** qualifier.

Transforms in shader based OpenGL

Typically is the transform constructed in the host C++ program using GLM and passed to the vertex shader as uniform variables. On the GPU side, the vertex shader applies the transforms on the incoming vertex data.

3 Transformations

This section covers following:

- Obejct representation in computer graphics
- Some review of linear algebra
- Linear transformations such as: rotation and scaling with matrices
- Translation and homogeneous coordinates
- A lot of transforms can be expressed with matrix multiplications

Computer graphics pipeline image here!

Object representations for computer graphics

In computer graphics objects is described by its surface, this means that they are hollow. These objects are specified by a sets of 3D points, so called vertices. An object can be approximated by a multiple convex polygons. We can transform the object by transforming these vertices. We can approximate objects with small polygons, but if each polygon is smaller or in same size of a pixel there is no visual difference. Modern graphics hardware have the capability to render a lot of polygons fast.

Other object representations

Objects can also be expressed as a solid with Voxels = 3D pixels. An object can be described implicitly with coordinates and functions: $F(x, y, z) = C$.

Math: linear algebra review

Scalars are real or complex numbers, real number most often used. Points are locations in space and don't have any size or shape while vectors are directions in space with a magnitude but don't have any position. Review of vector operations:

- Addition and subtraction
- Multiplication by scalar
- Magnitude, Euclidian norm $\mathbf{a} = (a_x, a_y), ||\mathbf{a}|| = \sqrt{(a_x, a_y)} = \sqrt{\mathbf{a} \cdot \mathbf{a}}$
- Normalizing $\hat{\mathbf{a}} = \frac{\mathbf{a}}{||\mathbf{a}||}$ look out for division by zero!!!

Matrix multiplication:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax & by & cz \\ dx & ey & fz \\ gx & hy & iz \end{bmatrix} \tag{1}$$

Linear combinations of vectors

Convex combinations, that are coefficients are all positive and add up up to 1. For example linear interpolation: $\mathbf{p}(t) = (t)\mathbf{a} + (1-t)\mathbf{b}$, where **a** and **b** are points and t: $0 \leq t \leq 1$.
Dot product: $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$. Notice that vectors are orthogonal when the dot product is equal to zero and that vectors are less than 90 degrees apart if the dot product is positive. The dot product can be used to calculate the perpendicular **projection** of a vector onto another vector: $\mathbf{c} = (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$. It can also be used for the **reflection** of a vector over the normal vector $\hat{\mathbf{n}}$: $b = -a + 2(\mathbf{a} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$

The **cross product** with $u = (u_x, u_y, u_z), v = (v_x, v_y, v_z)$:

$$u \times v = \begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = (u_y v_z - u_z v_y)\mathbf{x} - (u_x v_z - u_z v_x)\mathbf{y} + (u_x v_y - u_y v_x)\mathbf{z} \quad (2)$$

The cross product $u \times v$ is perpendicular to both **u** and **v** and the orientation of it follows the right hand rule. The norm : $||\mathbf{u} \times \mathbf{v}|| = ||\mathbf{u}||||\mathbf{v}|| \sin\theta$

Transformations

In order to create and move objects we need to be able to transform the objects in different ways. Transformations are divided into many classes:

- Translation (move the object around)
- Rotation
- Scaling
- Shear
- Mirroring/Flipping
- ...

Translation is simply adding a constant to all points. $x' = x + \mathrm{d}x$ and $y' = y + \mathrm{d}y$. **Scaling** an object by making it either smaller or bigger by a constant scaling factor. Here we multiply each point of the object with the scaling factor: $x' = s_x x$ and $y' = s_y y$. **Rotation** (about the origin) is easist described with the rotation matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (3)$$

The process goes: translate the rotation centre to the origin, rotate and then translate back. **Translation** looks different in matrix form, here we need to introduce an extra dimension.

$$\begin{pmatrix} x' \\ y' \\ W \end{pmatrix} = \begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (4)$$

Note, if W = 0 then the point is not affected by translations. Using this extra dimension for the other translations also is called: Using homogenous coordninates:

- Translation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{5}$$

- Rotation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{6}$$

- Scaling

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{7}$$

The order of the matrices is important:

$$P' = T^{-1}(S(R(T(P)))) = (T^{-1}SRT)P \tag{8}$$

When moving in to 3D are translation and scaling basically the same as in 2D. Rotation becomes a bit more comlicated: Rx,Ry and Rz. Obersve also that RxRy $\neq$ RyRx.

4 3D viewing and projection

This lecture will cover rotation around arbritary axis, the "view" coordinate system, change of coordinate system, change of frame (change of coordinate system + translation), how to postion a camera and projections (orthogonal and perspective).

Rotation around an arbitary axis

For rotation around an arbritary axis $\mathbf{v}$, we need to find a matrix $\mathbf{M}$ that aligns (1,0,0) with $\mathbf{v}$, then apply $\mathbf{M}^{-1}$, apply rotation and $\mathbf{M}$ to the object. Here $\mathbf{M}$ is the change of coordinate system. But how do we find $\mathbf{M}$?

Change of coordinate system

The goal of change of coordinate system is to express a point with two different sets of basis vectors:

$$\mathbf{P} = x\mathbf{e}_1 + y\mathbf{e}_2 = x'\mathbf{f}_1 + y'\mathbf{f}_2 \tag{9}$$

If we assume to know the basis vectors of the new coordinate system $\mathbf{f}$, in terms of the old coordinate system $\mathbf{e}$.

$$\begin{aligned} \mathbf{f}_1 &= a\mathbf{e}_1 + b\mathbf{e}_2 \\ \mathbf{f}_2 &= c\mathbf{e}_1 + d\mathbf{e}_2 \end{aligned} \tag{10}$$

Applying this gives us:

$$\mathbf{P} = x\mathbf{e}_1 + y\mathbf{e}_2 = x'\mathbf{f}_1 + y'\mathbf{f}_2 = x'(a\mathbf{e}_1 + b\mathbf{e}_2) + y'(c\mathbf{e}_1 + d\mathbf{e}_2) \tag{11}$$

Coordinates in $\mathbf{f}$ transferred to $\mathbf{e}$:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{M} \begin{bmatrix} x' \\ y' \end{bmatrix} \tag{12}$$

If we want coordinates in $\mathbf{e}$ transferred to $\mathbf{f}$:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} x \\ y \end{bmatrix} \tag{13}$$

$\mathbf{M}$ is a pure rotation matrix if and only if the basis are Ortho-Normal (ON)-bases. In that case is $\mathbf{M}$ orthogonal and $\mathbf{M}^{-1} = \mathbf{M}^T$. An orthogonal matrix is a matrix where the rows and columns are mutually orthogonal unit-length vectors ?. Some properties of two orthogonal matrices are: the product is always orthogonal, always invertible and the inverse of an orthogonal matrix is equal to the **transpose** of the matrix.
How do we construct this ON basis if we only have one vector $\mathbf{v}$

$$\begin{aligned} v_1 &= \frac{\mathbf{v}}{|\mathbf{v}|} \\ v_2 &= \frac{v_1 \times v'}{|v_1 \times v'|} \\ v_3 &= v_1 \times v_2 \end{aligned} \tag{14}$$

How do we find a vector v' that is not parallel to v_1? We could just pick a random vector and check if the norm of the cross product is non zero. The problem with that is the risk of numerical error if the cross product is close to 0. A better way is to pick two orthogonal vectors $\mathbf{u}_1$ (1,0,0) and $\mathbf{u}_2$ (0,1,0). If $|\mathbf{u}_1 \times \mathbf{v}_1| > |\mathbf{u}_2 \times \mathbf{v}_1|$, set $\mathbf{v}' = \mathbf{u}_1$, otherwise $\mathbf{v}' = \mathbf{u}_2$.
We now have rotation about an arbitrary axis $\mathbf{v}$:

- Construct an **ON**-basis where $\mathbf{v}_1$=$\mathbf{v}$ is the first basis vector,
- Construct a corresponding "change of coordinates" matrix $\mathbf{M}$, which align (1,0,0) with $\mathbf{v}$
- Apply $\mathbf{M}^{-1}$ to object (transfering coordinates to a coordinate system where $\mathbf{v}$ is aligned with (1,0,0))
- Rotate around (1,0,0)
- Apply $\mathbf{M}$ (transfers coordinates back to the original coordinate system)

Some coordinate systems

Here follows some examples of coordinate systems in the pipeline:

$$p' = V_p * P * C * V_t * M * p \tag{15}$$

The **view transform** v_t puts the observer at the origin and align x and y axes with the ones of the screen. Uses a right hand coordinate system so the z-axis is pointing backwards. This simplifies the clipping, light perspective and HSR (?). The view transformation is a so called change of frame, a change of origin + a change of coordinate system. This can be split into a translation and a change of coordinate system :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{M}^{-1}\mathbf{T}(-P_0) \begin{bmatrix} x \\ y \end{bmatrix} \tag{16}$$

glm::lookAt creates a viewing matrix that is derived from an eye point indicating the center of the scene and an UP vector. This build the 4x4 matrix for you but what is described above is what going on behind the scenes.
The **projection** p. Two types of projections: **Perspective** and **Orthographic** projection. In the orthographic projection is the "center of projection" at infinity and the projected points are along the direction of projection DOP. This projection is easy to implement since we only need to set z = 0.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{17}$$

Perspective projection is a linear equation:

$$\begin{aligned} &(1) \quad \mathbf{p}' = t\mathbf{p} \\ &(2) \quad d = tp_z \end{aligned} \tag{18}$$

if we solve for (2) for t, we get:

$$\mathbf{p}' = \frac{d}{p_z}\mathbf{p} \tag{19}$$

Here we use W to fit the perspective transformation into a matrix multiplication:

$$\begin{aligned} x' &= x\frac{d}{z} \\ y' &= y\frac{d}{z} \\ z' &= z\frac{d}{z} \end{aligned} \tag{20}$$

$$\begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{21}$$

We divide by the homogeneous coordinate W to obtain the final projection, not that this is not a linear transformation.
glm::perspective(fovy, aspect, near, far) can create the projection matrix.