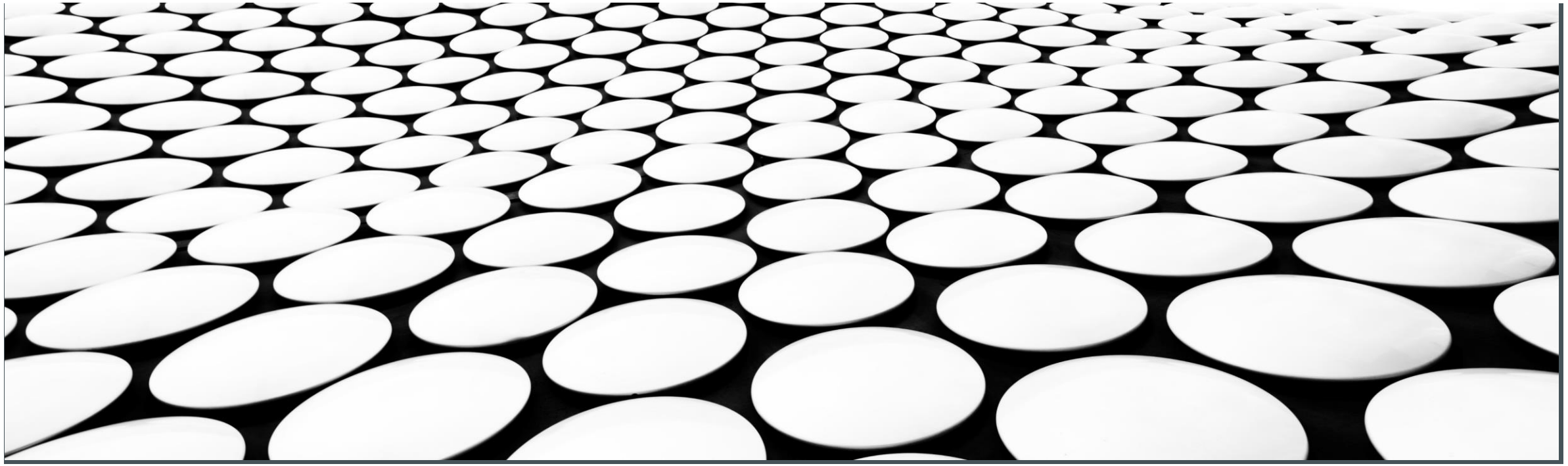
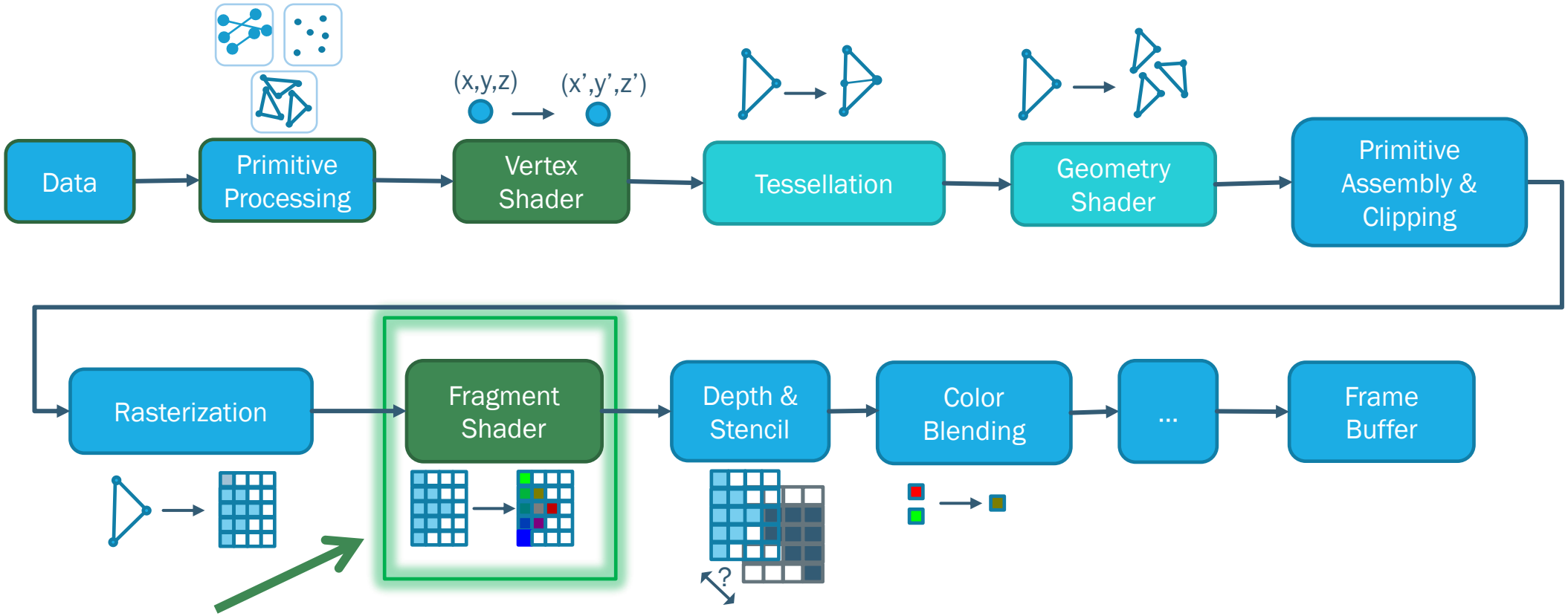
INFO-H-502 – VIRTUAL REALITY EXERCISES 02

ELINE SOETENS – LAURIE VAN BOGAERT

GAUTHIER LAFRUIT – DANIELE BONATTO



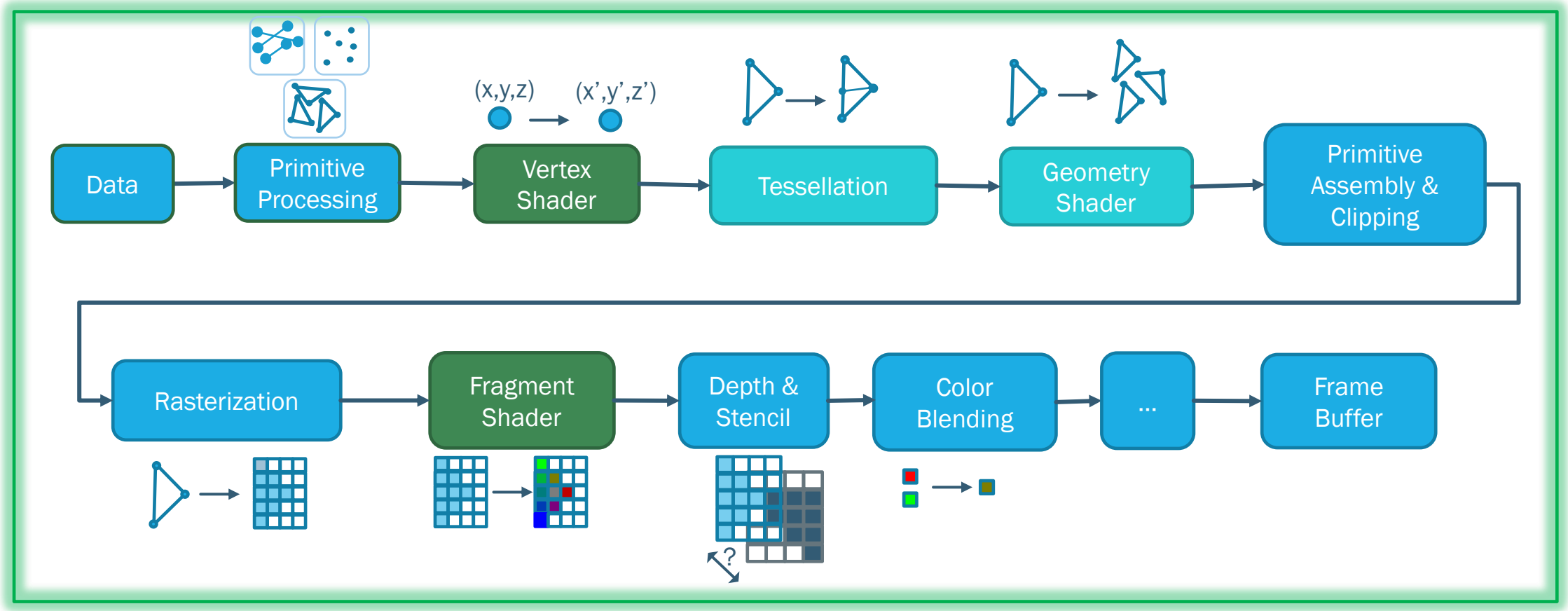
PREVIOUSLY



Overview + Fragment Shader

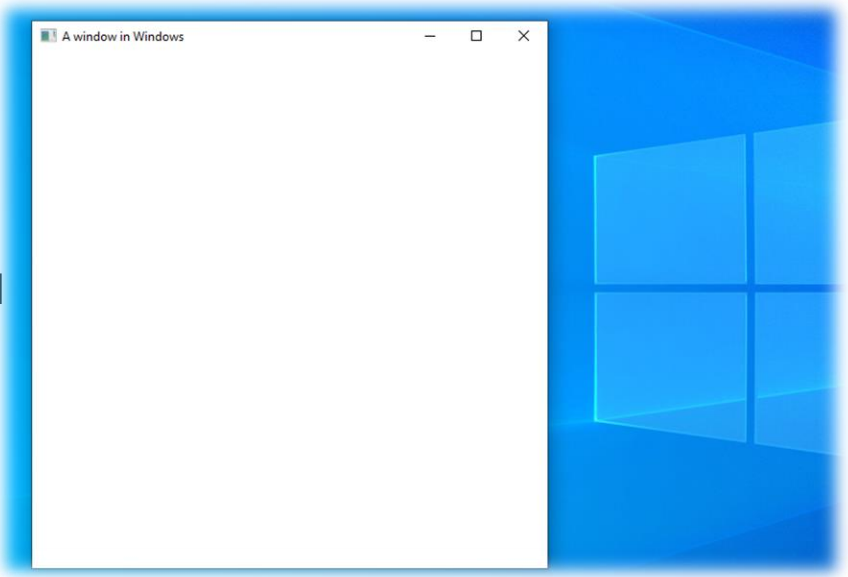
TODAY

We will use OpenGL, C++, Cmake + git
As well as 4 smaller libraries that will make our life easier



→ Program the whole pipeline!

GLFW AND WINDOW CREATION



- To use OpenGL you need an OpenGL context which need a window to be created
  - Problem: Window creation is platform-dependent … (and often need a bunch of ugly code)
  - Solution: Use a cross-platform library → GLFW that will take care of the “ugly” code for us

```
glfwInit() //Initialize GLFW
//specify the version and profile you want
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
//create a window
GLFWwindow * window = glfwCreateWindow(width, height, "Ex01", nullptr, nullptr);
//create the context
glfwMakeContextCurrent(window);
… //do some stuffs
//clean up ressources
glfwDestroyWindow(window);
glfwTerminate();
```

```
// test if a window should close
glfwWindowShouldClose(window);
// process the events
glfwPollEvents()
// swap the buffers to put the result of
the rendering operation on screen
glfwSwapBuffers(window);
```

```
//to be in sync with the screen
glfwSwapInterval(1);
```

→ More infos if interested at:
https://www.glfw.org/docs/latest/quick.html

LOADING OPENGL FUNCTION AND GLAD

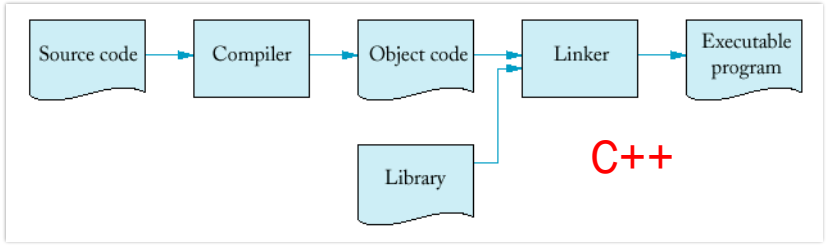
- OpenGL is a standard/specification, the functions are implemented by hardware manufacturer and their implementation are present in your graphics driver
  - → if you want to use them, you need to load them:
  - `PFNGLCOMPILESHADERPROC glCompileShader = (PFNGLCOMPILESHADERPROC) glfwGetProcAddress("glCompileShader");`
  - … For each function … which is an awful process ☹
  - We won't do that
- However, developers aren't (usually) masochist →They use an OpenGL loading library ☺
  - Like GLAD (but they are other ones: glew, gl3w, ect…)
  - Call one function once after the creation of the OpenGL context

```
//include glad before GLFW to avoid header conflict
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

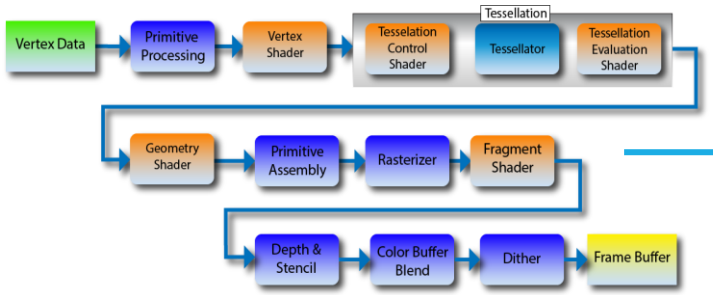
Don't forget to call this

```
//load openGL functions
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    throw std::runtime_error("Failed to initialize GLAD");
}
```

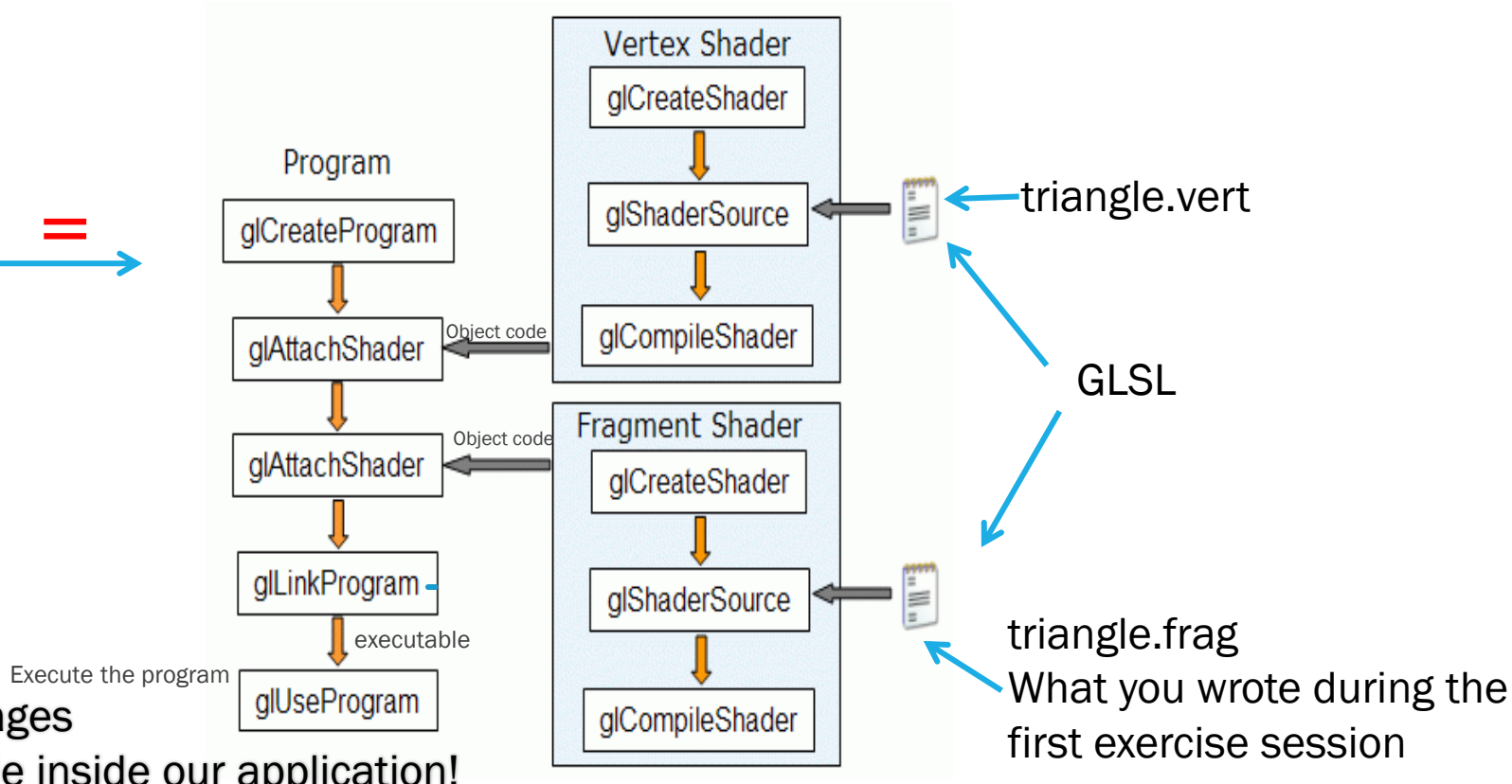
TECHNICALITIES - OPENGL PIPELINE



OpenGL



=



Above Right: Classical C++ compilation stages

The pipeline is like a “small” compiled code inside our application!

It uses all the classical compilation stages

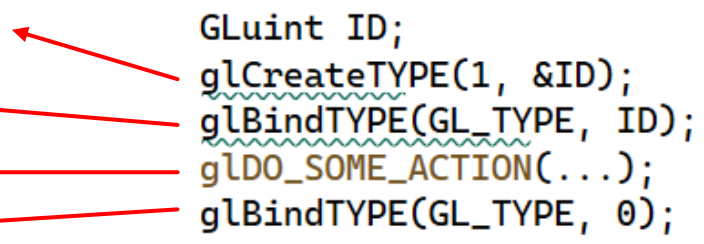
- Compile each shader independently from source to object codes (glCreateShader, glCompileShader)
- We link them into a “program” (glCreateProgram, glAttachShader, glLinkProgram)

Every program is associated with a unique GLuint (its ID)

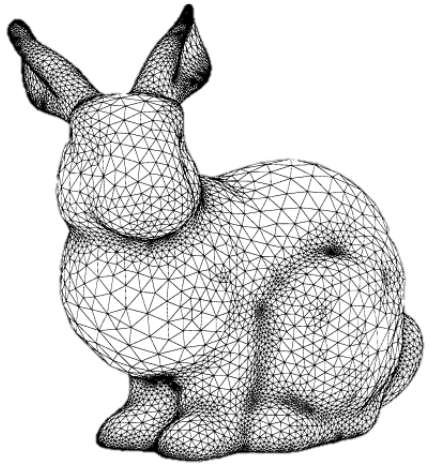
glUseProgram is used <u>every time</u> you want to use your program (and thus, apply your effects): glUseProgram(ID);

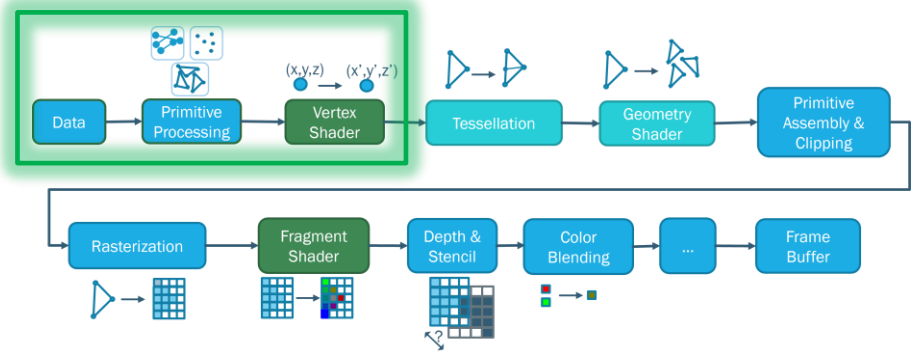
TECHNICALITIES - OPENGL AS A STATE MACHINE

- OpenGL is very old and use a global state machine:
  - You create an OpenGL object (buffer (VBO), VAO, EBO, Texture, …) and retrieve an ID for it
  - you need to call a function to say what you are going to work on (binding using ID)
  - You do your operations
  - You say that you stop working on this (unbinding using a binding to null)



DATA TRANSFERS





- We have 3 ways of transfering data in OpenGL:
  1. From the CPU to the GPU: this is used for objects (bunny), we use buffers
     - We create a vertex buffer object in the CPU
     - We write our vertex shader with a variable that will read one vertex at a time from the buffer
     - We write how to interpret our buffer **at each** rendering loop
       - glVertexAttribPointer(variable, how_many_elements_to_read, type, normalized, array_size, stride)
     - **Keyword: Attribute**

Buffer creation

```
const float data[9] = {
    // vertices
    -1.0, -1.0, 0.0,
    1.0, -1.0, 0.0,
    0.0, 1.0, 0.0,
};

GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Vertex by vertex

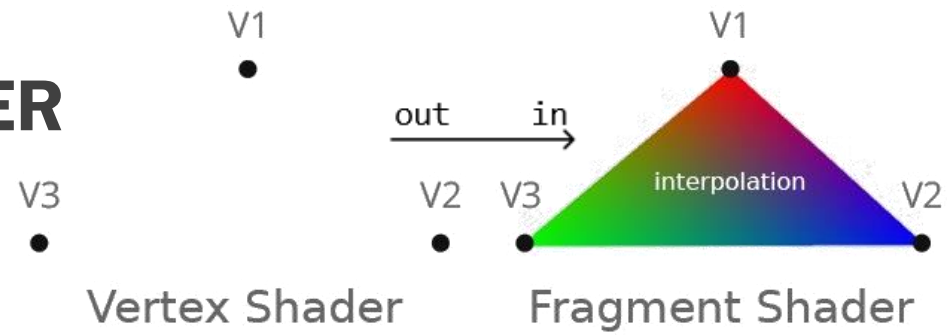
Vertex source

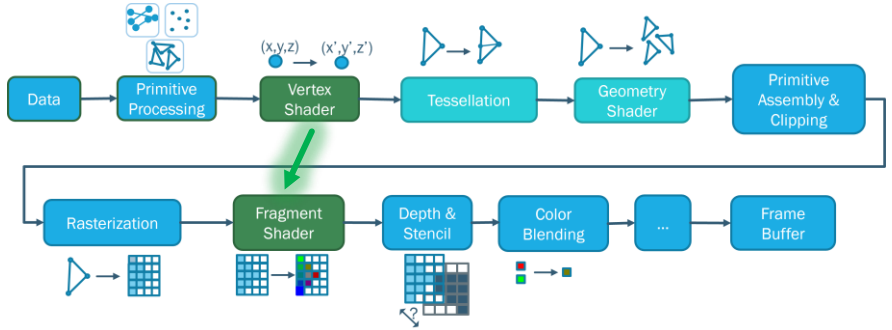
```
const std::string sourceV =
    "#version 330 core\n"
    "in vec3 position; \n"
    "void main(){ \n"
    "gl_Position = vec4(position, 1);\n"
    "}\n";
```

How to interpret our buffer (during rendering)

```
auto attribute = glGetAttribLocation(program, "position");
glEnableVertexAttribArray(attribute);
glVertexAttribPointer(attribute, 3, GL_FLOAT, false, 0, 0);
```

DATA TRANSFER





- We have 3 ways of transfering data in OpenGL:
  2. From the Vertex shader to the Fragment shader: Passing colors for example
     - We create an « output » variable in the vertex shader
     - We put values in this variable
     - We create the same variable name in the Fragment shader
     - We can now access the values contained in it
     - Keyword: Out/In

Vertex source

```
const std::string sourceV = "#version 330 core\n"
    "in vec3 position;\n"
    "out vec4 color;\n"
    " void main(){\n"
    "gl_Position = vec4(position, 1);\n"
    "color = gl_Position * 0.5 + 0.5;\n"
    "}\n";
```

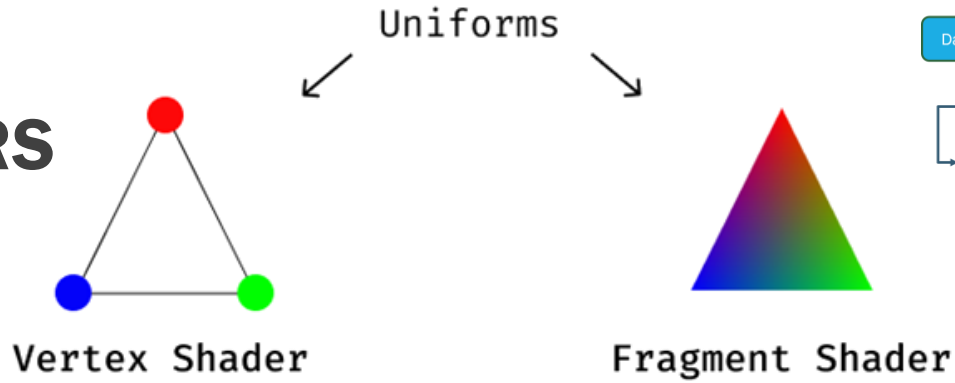
Fragment source

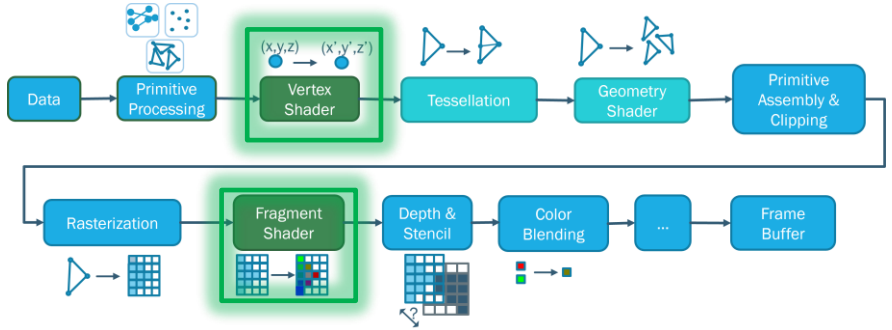
```
const std::string sourceF = "#version 330 core\n"
    "precision mediump float;\n"
    "out vec4 FragColor;\n"
    "in vec4 color;\n"
    "void main() {\n"
    "FragColor = color;\n"
    "}\n";
```

Needed for floats

Output is the color

DATA TRANSFERS





- We have 3 ways of transfering data in OpenGL:
  2. Between the Main Loop and the shaders
     - We create a uniform variable (Vertex OR Fragment)
     - We use it to do something
     - In C++, we need to retrieve the address in the GPU of the uniform in the shader
     - In the **rendering loop**, we can send data using it
     - Keyword: uniform

```
const std::string sourceF = "#version 330 core\n"
        "precision mediump float; \n"
        "out vec4 fragColor;\n"
        "in vec3 color;\n"
        "uniform float time;\n"
        "void main() { \n"
        "fragColor = vec4(color * time, 1.0); \n"
        "} \n";
```

Fragment/Vertex source

```
auto u_time = glGetUniformLocation(program, "time");
```

```
glUniform1f(u_time, (float)glfwGetTime());
```

DATA TRANSFERS

Name

glUniform — Specify the value of a uniform variable for the current program object

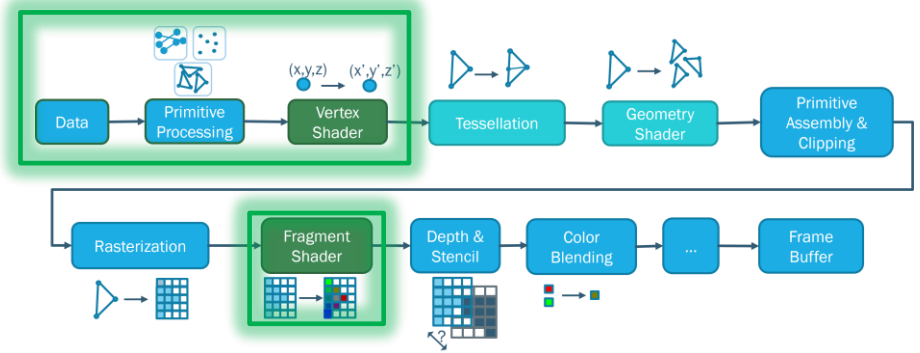
C Specification

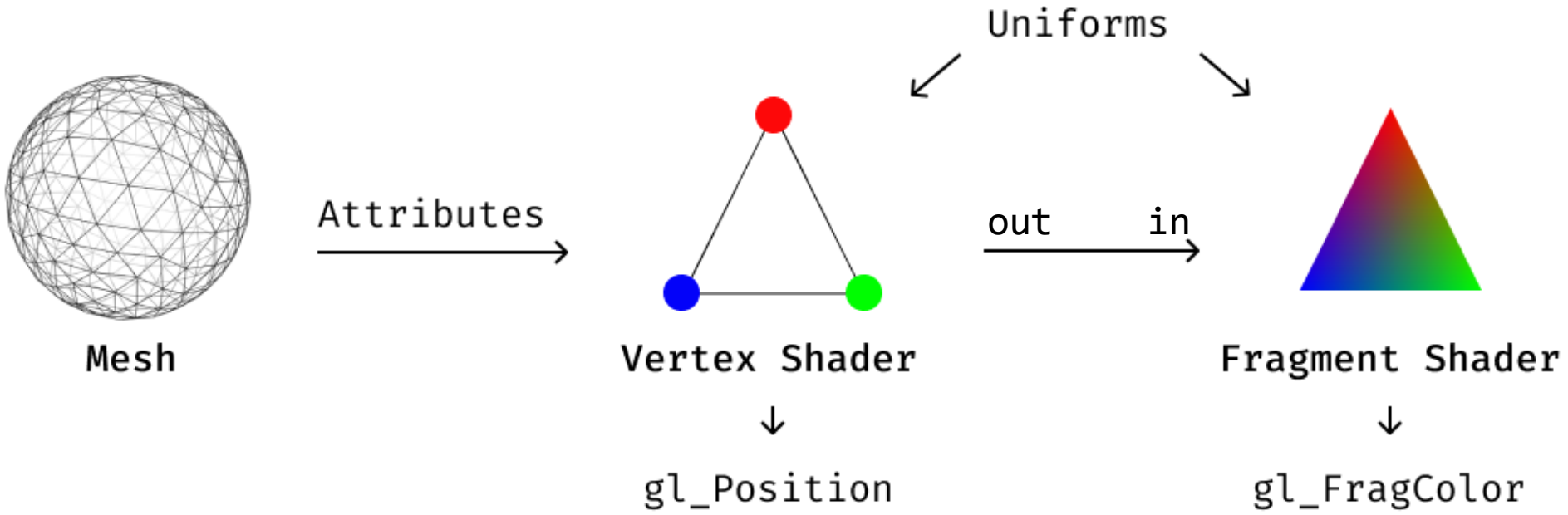
```
void glUniform1f( GLint location,
                  GLfloat v0);

void glUniform2f( GLint location,
                  GLfloat v0,
                  GLfloat v1);

void glUniform3f( GLint location,
                  GLfloat v0,
                  GLfloat v1,
                  GLfloat v2);

void glUniform4f( GLint location,
                  GLfloat v0,
                  GLfloat v1,
                  GLfloat v2,
                  GLfloat v3);

void glUniform1i( GLint location,
                  GLint v0);

void glUniform2i( GLint location,
                  GLint v0,
                  GLint v1);

void glUniform3i( GLint location,
                  GLint v0,
                  GLint v1,
                  GLint v2);
```

Many more types available

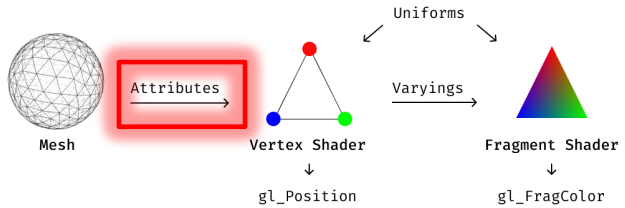
https://registry.khronos.org/OpenGL-Refpages/gl4/html/glUniform.xhtml

DATA TRANSFERS





INTERLUDE: VERTEXATTRIBPOINTER – ANATOMY – BUFFER TO GPU




```
const float objectData[24] = {
        // vertices          //color          //tex coord
        -1.0, -1.0, 0.0,     1.0, 0.0, 0.0,   0.0, 0.0,
        1.0, -1.0, 0.0,      0.0, 1.0, 0.0,   1.0, 0.0,
        0.0, 1.0, 0.0,       0.0, 0.0, 1.0,   0.5, 1.0,
    };
```

Type of data
GL_FLOAT usually

Is the data
normalized
false

```
void glVertexAttribPointer( GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void * pointer);
```

Index of the attribute
where the data is going

```
auto attribute = glGetAttribLocation(program, "position");
```

Number of
elements to read:
- vertices: 3
- Color: 3
- TexCoord: 2

Byte offset between
consecutive attributes

```
(3+3+2) * sizeof(float)
```

Byte offset of the first
component
- vertices: 0*sizeof(float)
- Color: 3*sizeof(float)
- TexCoord: 6*sizeof(float)

INTERLUDE: GETATTRIBLOCATION OR LAYOUT

```
const std::string sourceV =
    "#version 330 core\n"
    "in vec3 position; \n"
    "void main(){ \n"
    "gl_Position = vec4(position, 1);\n"
    "}\n";
```

```
auto attribute = glGetAttribLocation(program, "position");
glEnableVertexAttribArray(attribute);
glVertexAttribPointer(attribute, 3, GL_FLOAT, false, 0, 0);
```

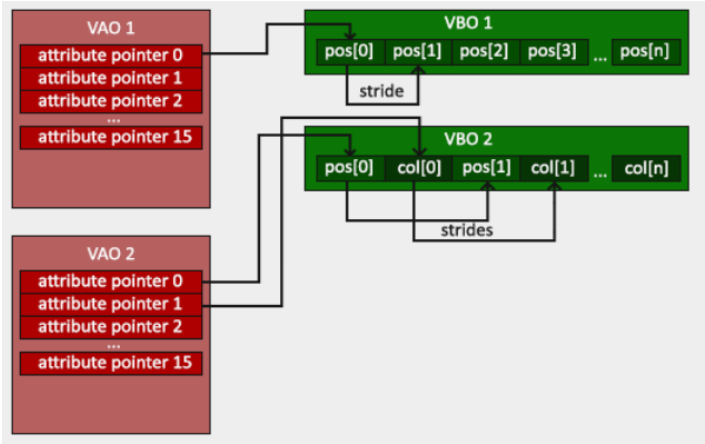
Get the location id with the function glGetAttribLocation

```
const std::string sourceV =
    "#version 330 core\n"
    "layout (location=0) in vec3 position; \n"
    "void main(){ \n"
    "gl_Position = vec4(position, 1);\n"
    "}\n";
```

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
```

Define the location id in the shader

INTERLUDE: VBO AND VAO



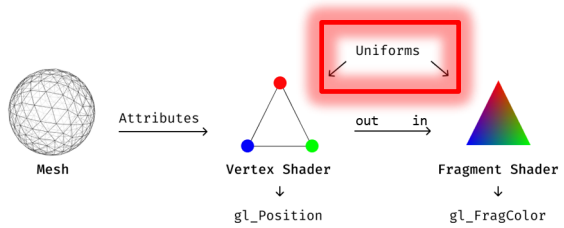
- Data stocked in the VBO
- At each rendering loop, we need to tell the vertex shader how to read the buffer
- We can use a vertex array object (VAO) to store « how to read the buffer » and which VBO to use
  - We create the VAO and VBO
  - We bind the VAO
  - We bind/configure the corresponding VBO
  - We just have to bind the VAO in the rendering loop

```
glBindVertexArray(VAO);
```

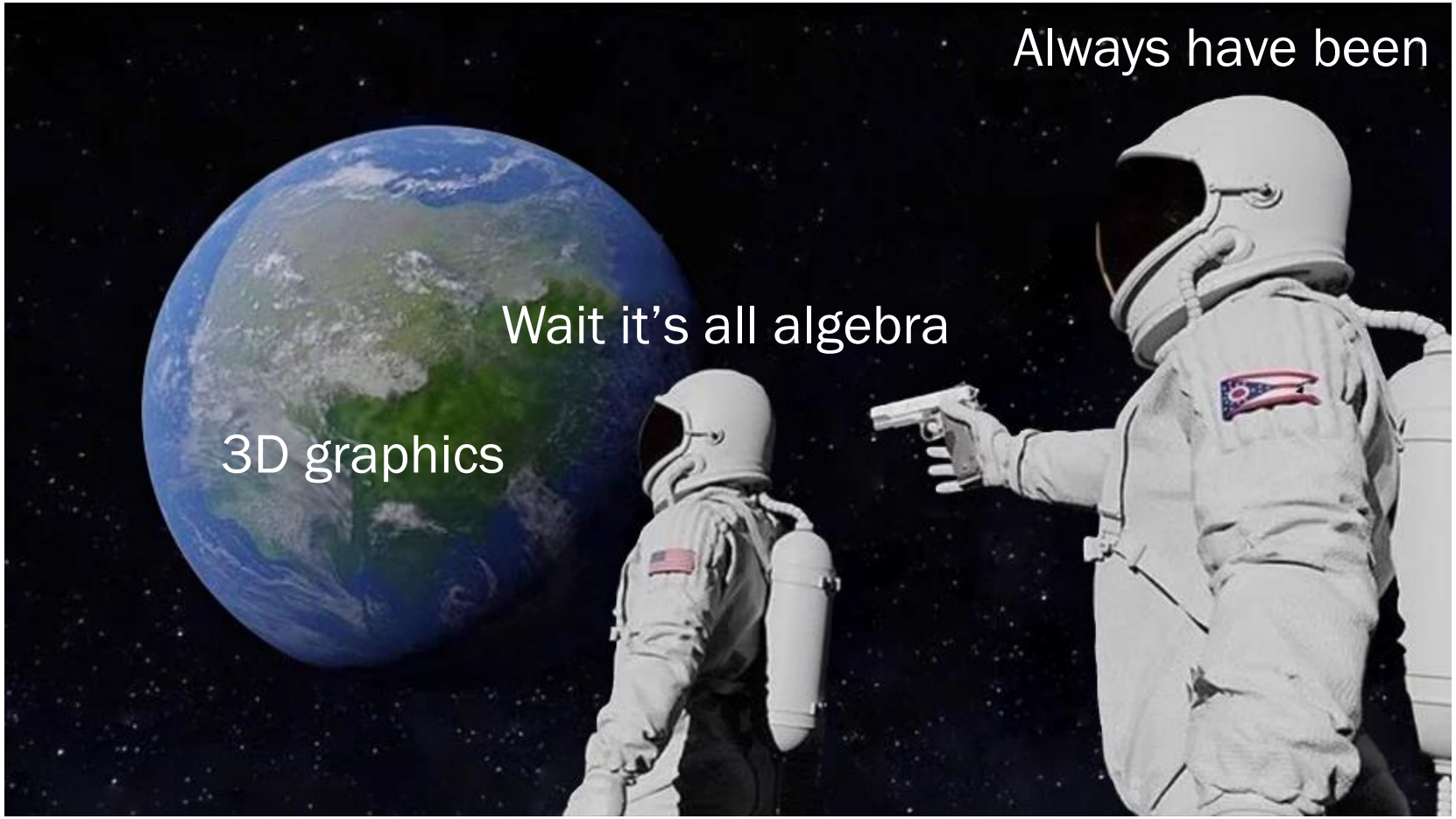
```
GLuint VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

MATHS IN OPENGL

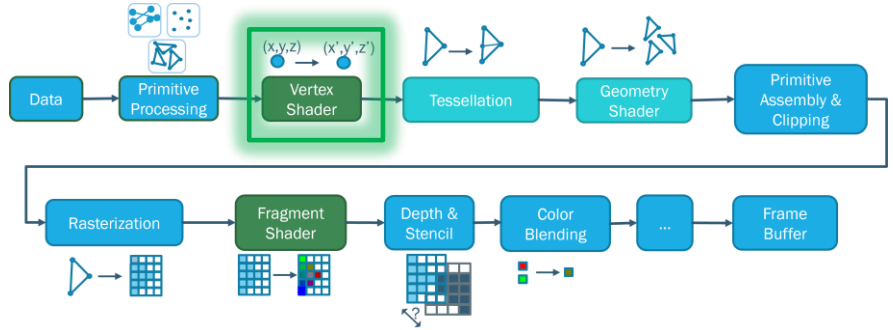
- No matrix or vector built-in function in OpenGL
- We will use GLM (Open**GL M**athematics), header library, already included in the exercise files
  - Basic utilities for **vec2, vec3, mat4**, …
  - Basic transformation like **scaling**, **translation**, …
  - Functions for **camera** and **perspective** matrices compatible with the OpenGL pipeline
  - Plays well with **uniforms**

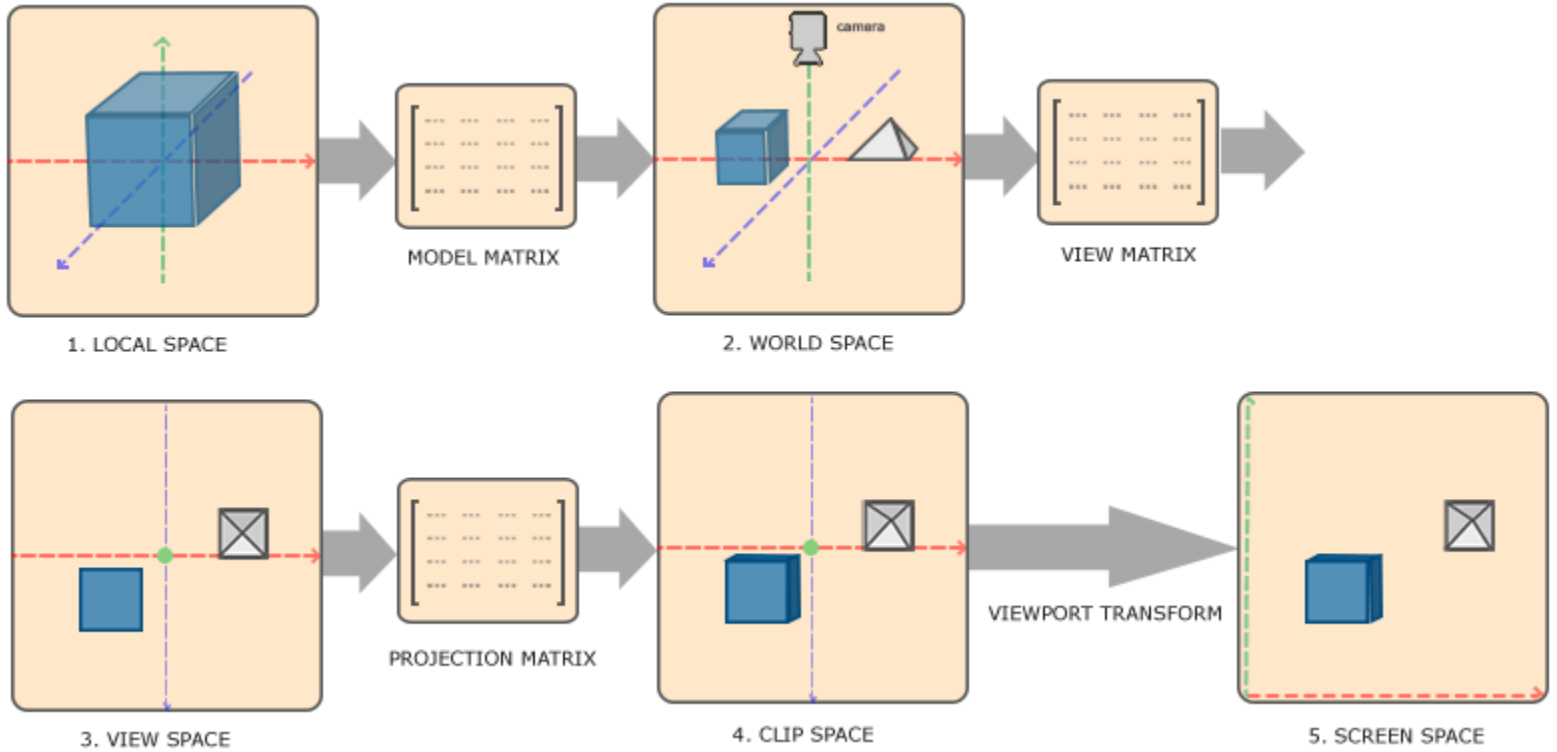


MODEL VIEW PROJECTION



MODEL VIEW PROJECTION



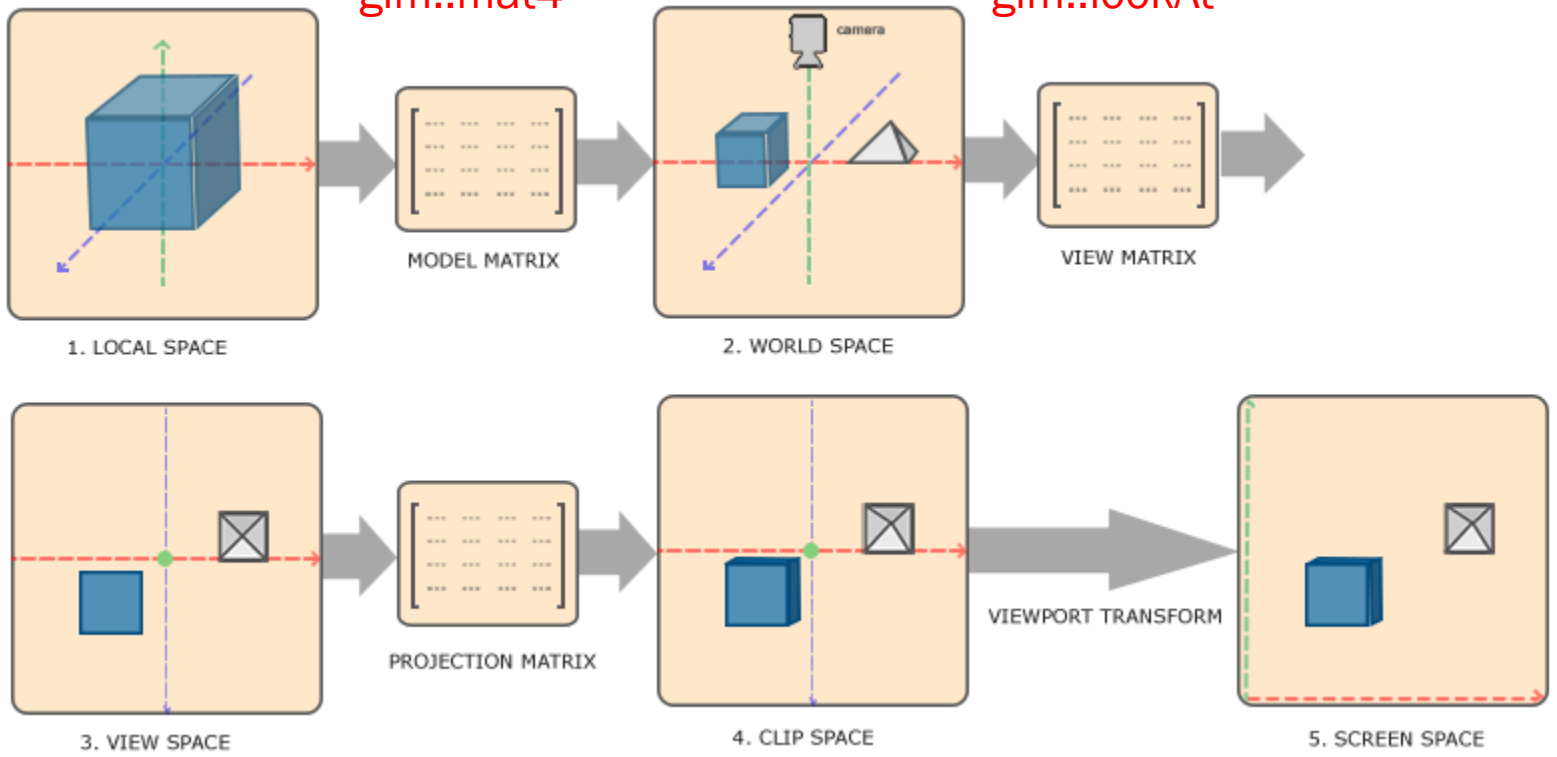


MODEL VIEW PROJECTION

Uniform matrices!
We can update the view and model at each frame

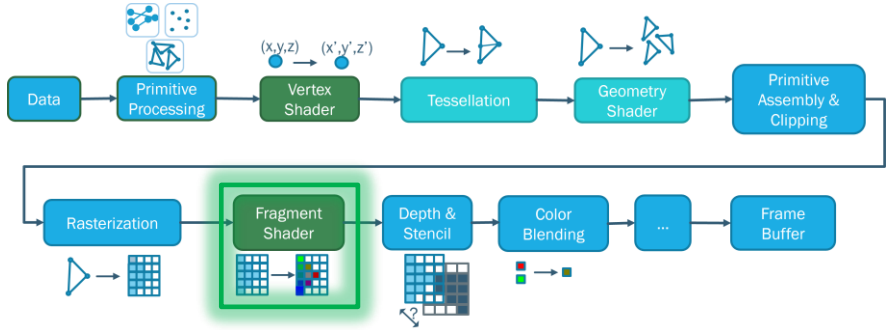
glm::mat4

glm::lookAt



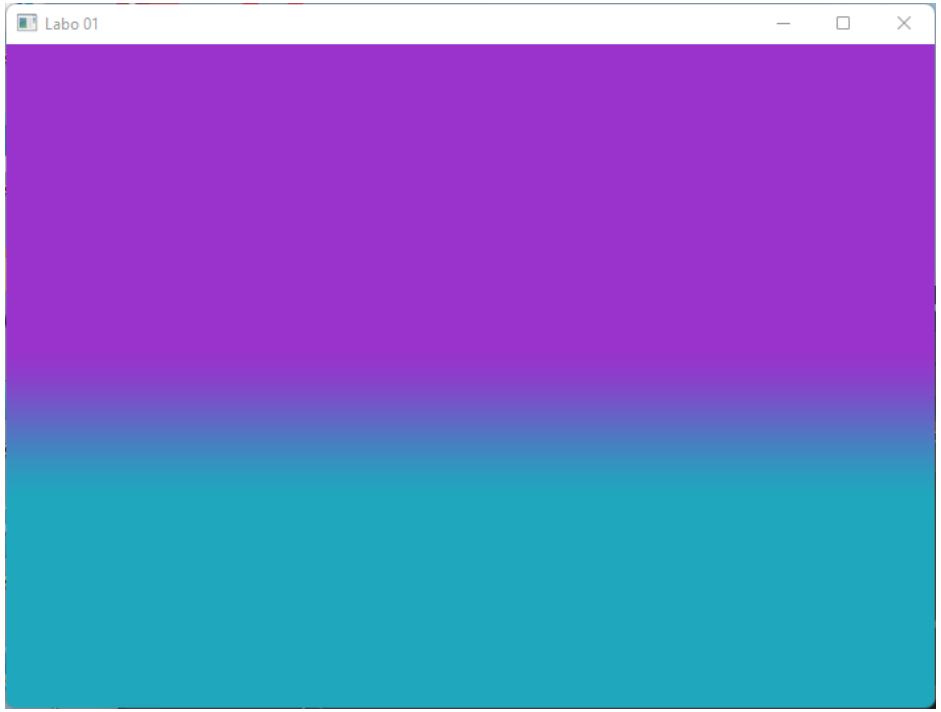
glm::perspective

FRAGMENT SHADER

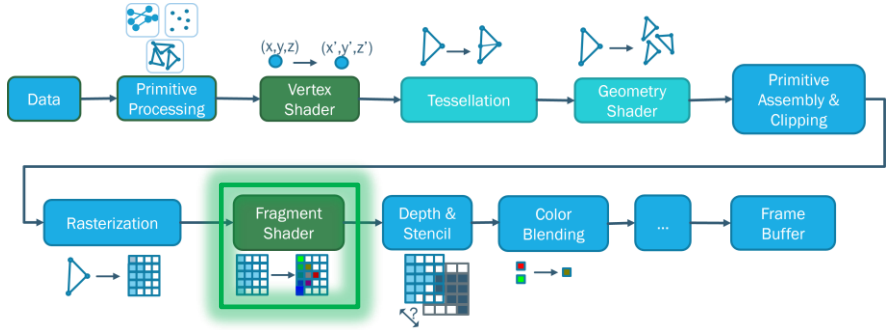


- We sent data from the CPU to the GPU (Buffers)
- We moved our mesh in the World coordinates (Vertex Shader)
- We moved again in camera coordinates (Vertex Shader)
- We performed a perspective deformation (Vertex Shader)

- Now, we are in the Fragment Shader!
  - But we have points in space!
  - Before we only had the screen coordinates
  - We can play with colors



TEXTURING



- Colors are sometime not enough

- To make a texture:
  - Create a texture object in the CPU memory
  - Configure the filtering and border warping
  - Read the image
  - Send the data to the GPU
  - Unbind the texture and free memory space

```
GLuint texture;
glGenTextures(1, &texture);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

stbi_set_flip_vertically_on_load(true);
int imWidth, imHeight, imNrChannels;
char file[128] = "../../../../LAB02/solutions/ex13/horse.jpg";
unsigned char* data = stbi_load(file, &imWidth, &imHeight, &imNrChannels, 0);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imWidth, imHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);

stbi_image_free(data);
glBindTexture(GL_TEXTURE_2D, 0);
```

- We use std_image, a header library, already included in the exercise files

TEXTURING

Texture Coordinates

```
const float positionsData[15] = {
    // vertices        // texture coords
    -1.0, -1.0, 0.0,    0.0, 0.0,
     1.0, -1.0, 0.0,    1.0, 0.0,
     0.0,  1.0, 0.0,    0.5, 1.0,
};
```

- To **render** a texture:
  - Change the buffer to take into account texture coordinates! (ADAPT THE VERTEX ATTRIB POINTERS)
  - Send the texture buffer ID as a uniform and activate it before drawing the object
  - Adapt the fragment shader to read the texture

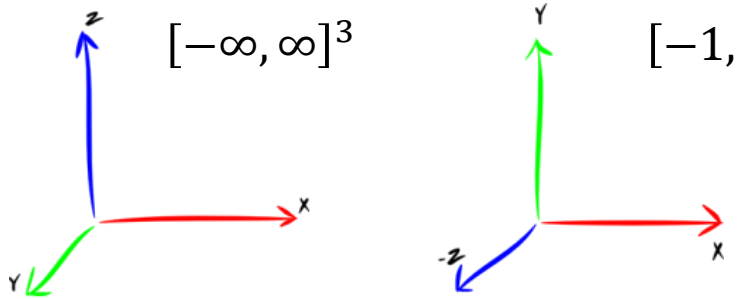
Rendering Loop

```
glUniform1i(u_texture, 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);

glDrawArrays(GL_TRIANGLES, 0, 3);
```

Fragment source

```
const std::string sourceF = "#version 330 core\n"
"out vec4 FragColor;"
"precision mediump float; \n"
"in vec2 v_texcoords; \n"
"uniform sampler2D u_texture; \n"
"void main() { \n"
"FragColor = texture(u_texture, v_texcoords); \n"
"} \n";
```

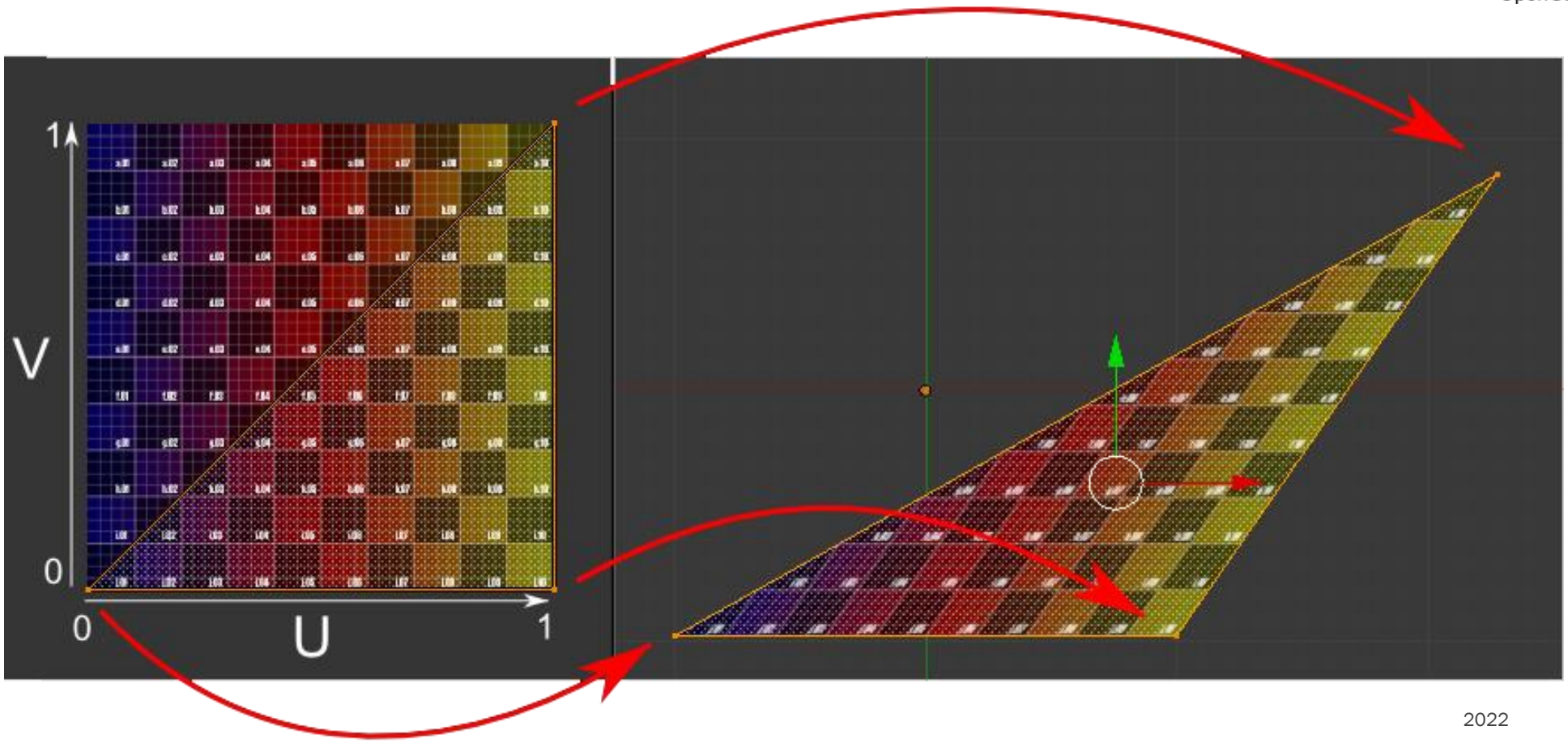
TEXTURE COORDINATES?



$[-\infty, \infty]^3$

$[-1,1]^3$

$[0,1]^2$



THE RENDERING LOOP

- We need an infinite loop to render our shaders!
  - Render our shader while the windows is open
  - During exercises: Implement a FPS counter

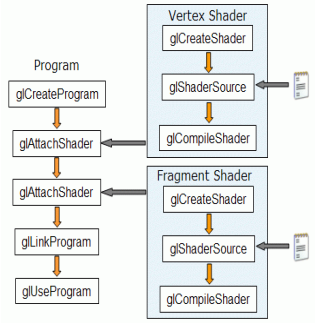
```
while (!glfwWindowShouldClose(window)) {
    glfwPollEvents();
    double now = glfwGetTime();

    //DRAW EVERYTHING

    fps(now);
    glfwSwapBuffers(window);
}
```

- Rendering functions:
  - Set the background color
  - Clear the screen from the previous frame
  - Activate some shader pipeline
  - Draw something

```
glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
glUseProgram(program);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

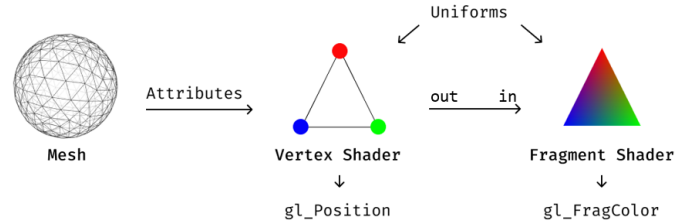
IMPORTANT PARTS OF THIS EXERCISE SESSION

- Boilerplate code
  - How to set up the window and context, load OpenGL
  - how to compile shaders
  - How to count frames (fps)

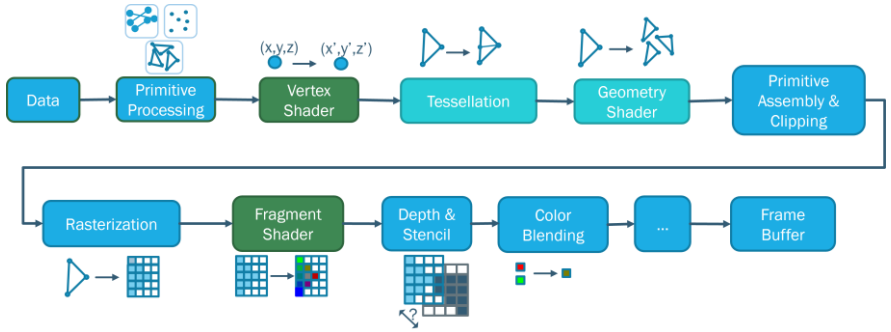



```
glfwInit() //Initialize GLFW
//specify the version and profile you want
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
//create a window
GLFWwindow * window = glfwCreateWindow(width, height, "Ex01", nullptr, nullptr);
//create the context
glfwMakeContextCurrent(window);
```

- How to transfer data
  - between C++ and the GPU (attributes)
  - Between vertex and fragment shaders (varying)
  - between C++ and the shaders, during the rendering (uniforms)



- How to write a vertex shader (with the matrix multiplications)
- How to write the fragment shader (very similar to ShaderToy!)



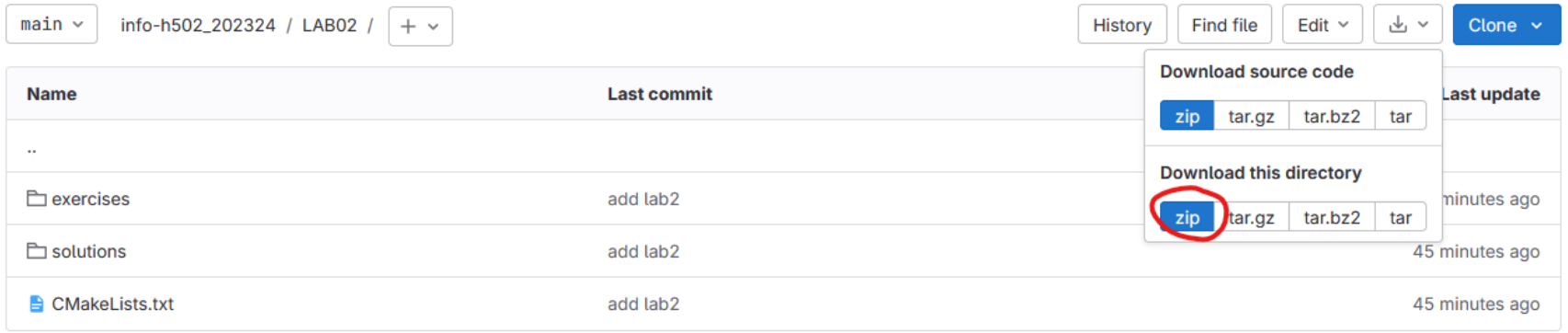
- If you master all of this, you can call it a day!
- the rest is loading some texture and playing with those concepts

GET THE EXERCISE FOR THIS SESSION

- Exercise have been updated on gitlab
- You can either :
  - Go to you cloned repository and pull the new exercices

```
\INFO-H502_202324>git pull
```

  - Download the folder we added from gitlab.



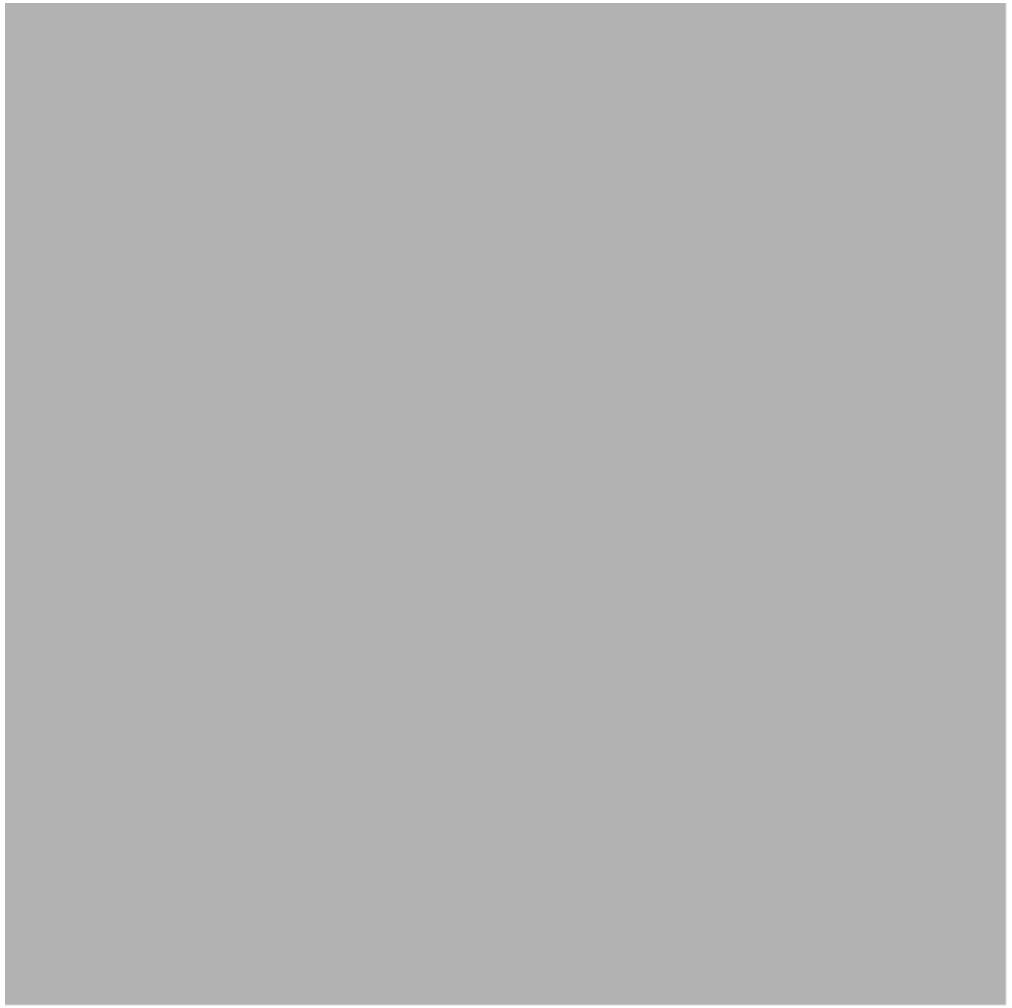
If you choose this solution, you will also need to update the CMakeList.txt

EXERCISES

- You can find in the folder the exercises and their solutions!
- One exercise per main.cpp file
- Follow the comments in the code and the text on the console to know what to do
- **IMPORTANT:** Use the debug mode
  - Catch error using glGetError() -> gives error ID that you have to look up
  - Or use the nice debugger with clear error message we made for you -> only in debug mode
- As usual, you can find useful Appendicies at the end!

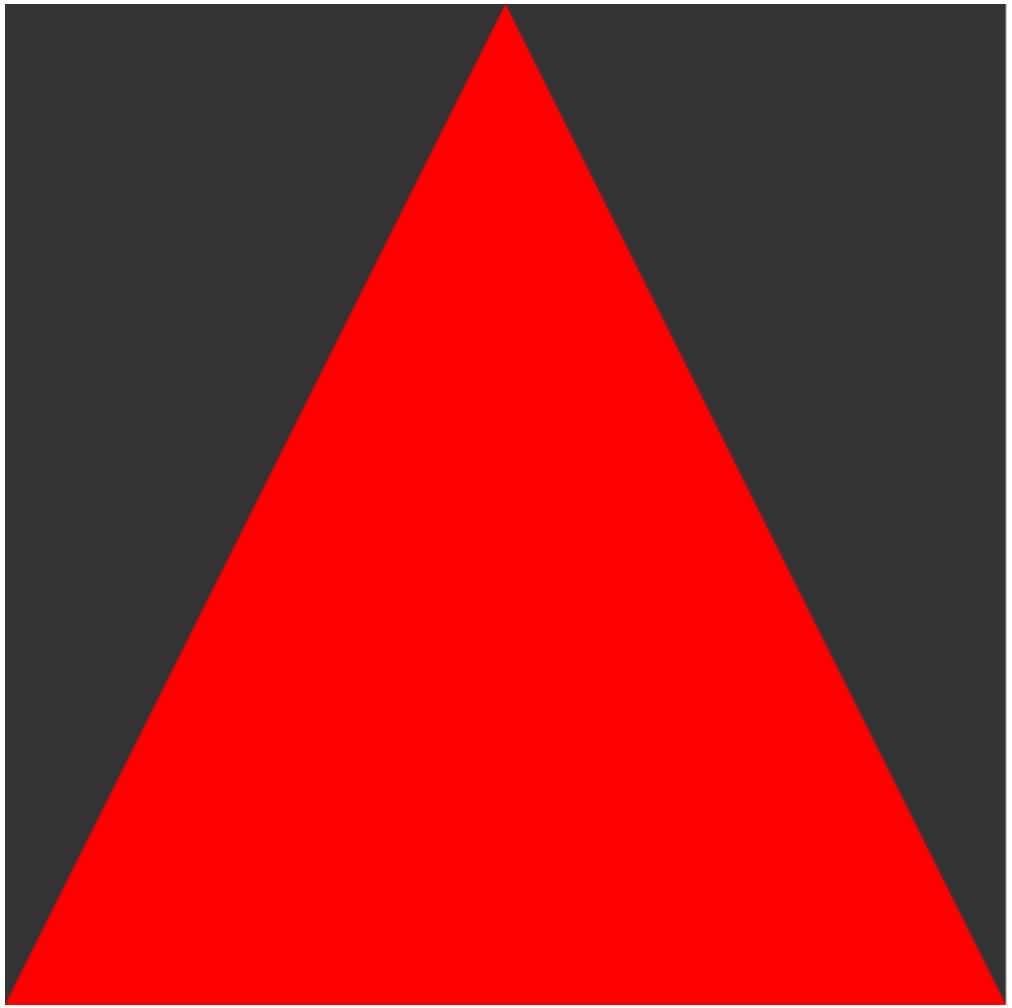
EXERCISE 01

- Make a window
  - This is just a gray square and
    you will not be able to draw something in it
    using only the fragment shader before the
    exercise 04
  - This exercise is about the boilerplate code of OpenGL,
    don't spend too much time on it.
  - You may fetch the code you need from:
    - The slides
    - The main.cpp of LAB01
    - The solution of this exercices



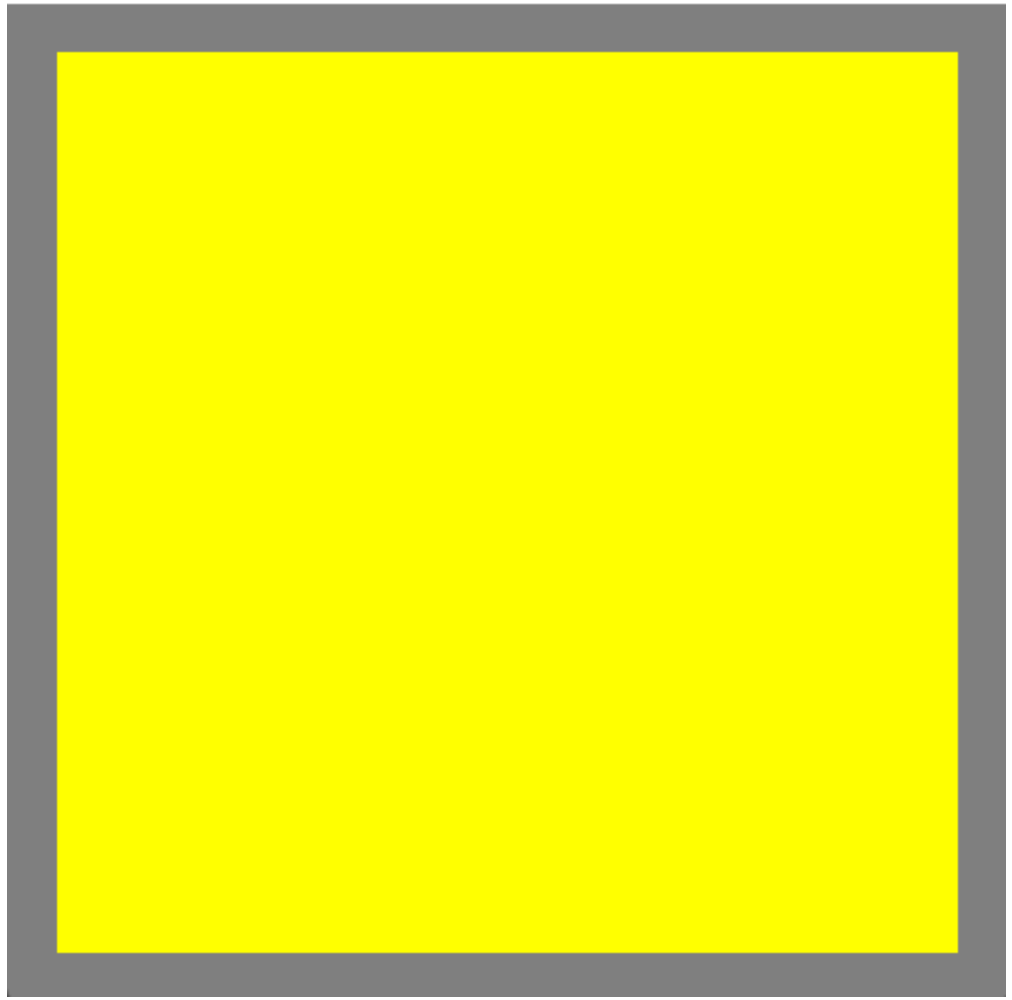
EXERCISE 02

- First triangle with the **Vertex** Shader



EXERCISE 03

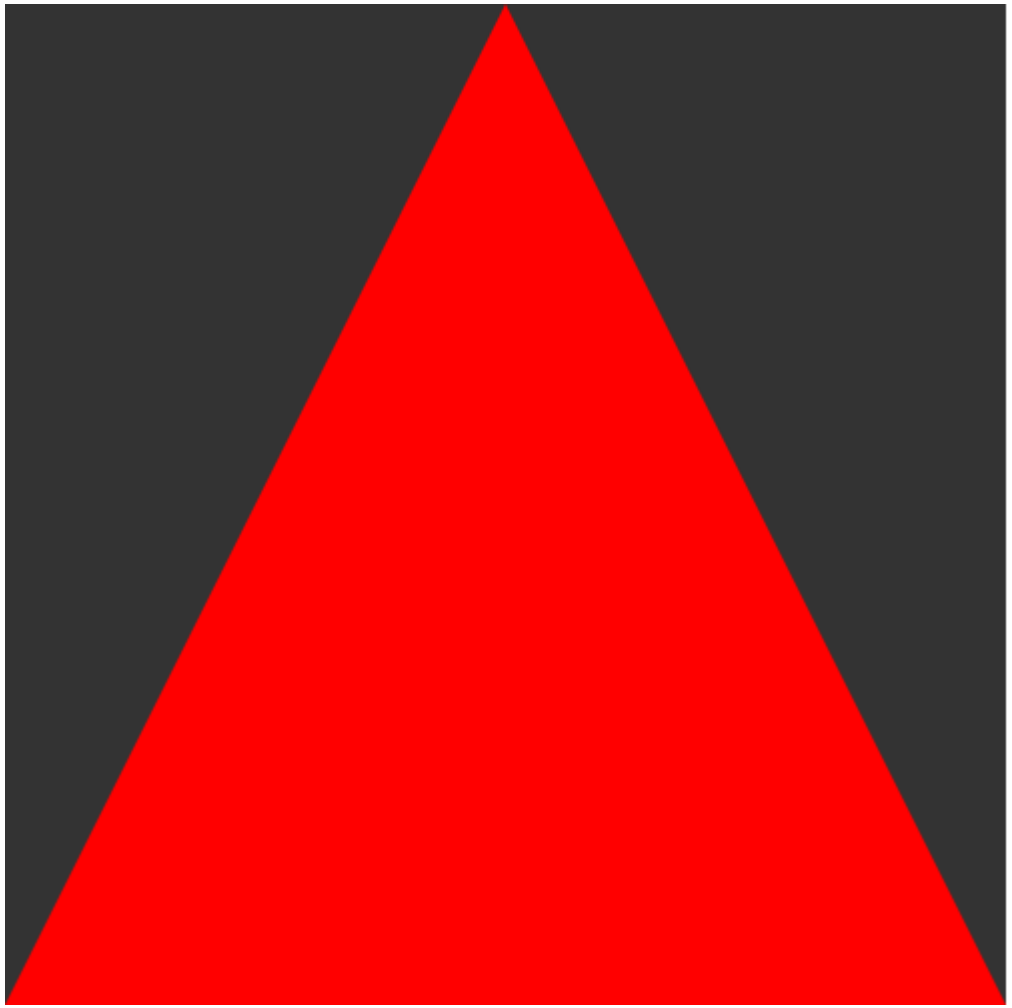
- A : Use the context debug (if you are on mac OS you can skip this)
- B : Draw a yellow rectangle rather than a red triangle



EXERCISE 04

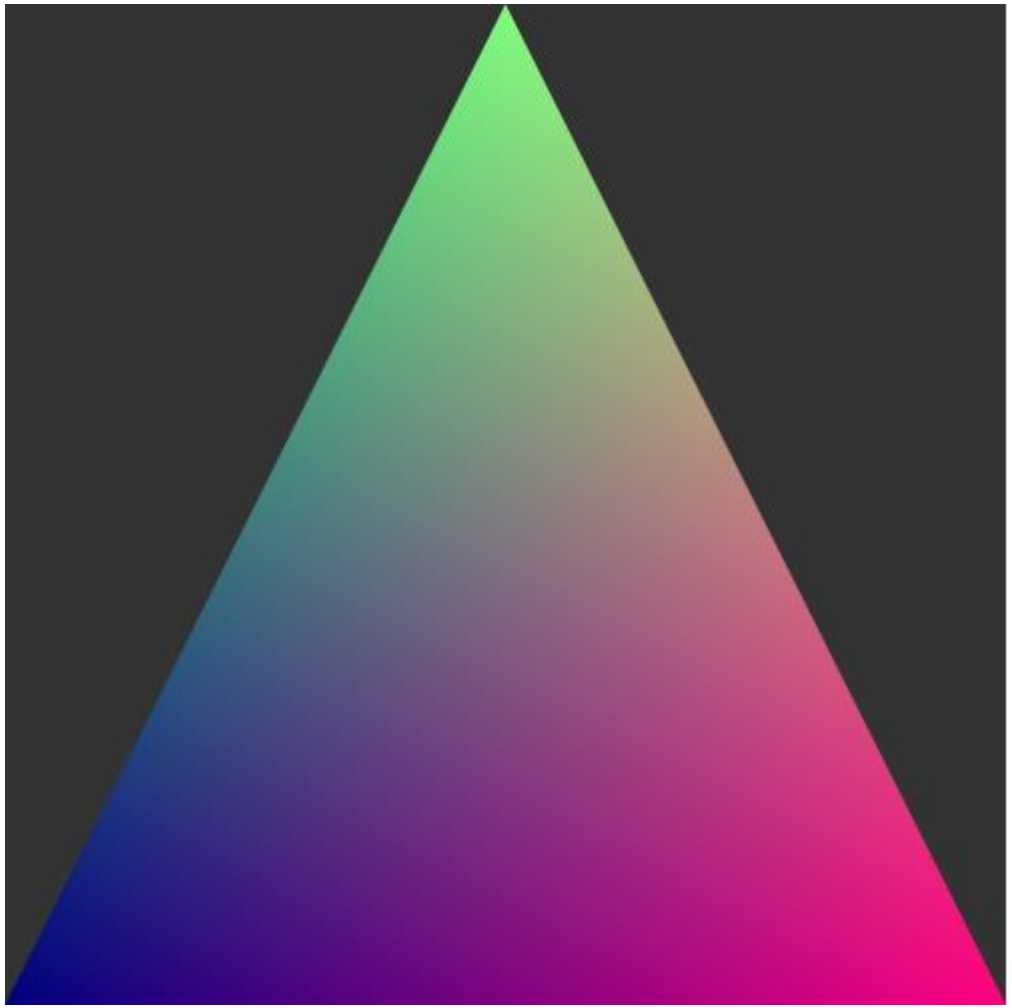
- FPS counter for the animation
  - FPS: Count the number of frames rendered by second
  - In practice: You compute how much time has passed from the previous frame to this one
    delta = now – previous
    and you estimate the FPS
    fps = 1/delta
  - Use glfwGetTime to know the current time

Note : you should show the fps count in the terminal, don’t try to put it in the window



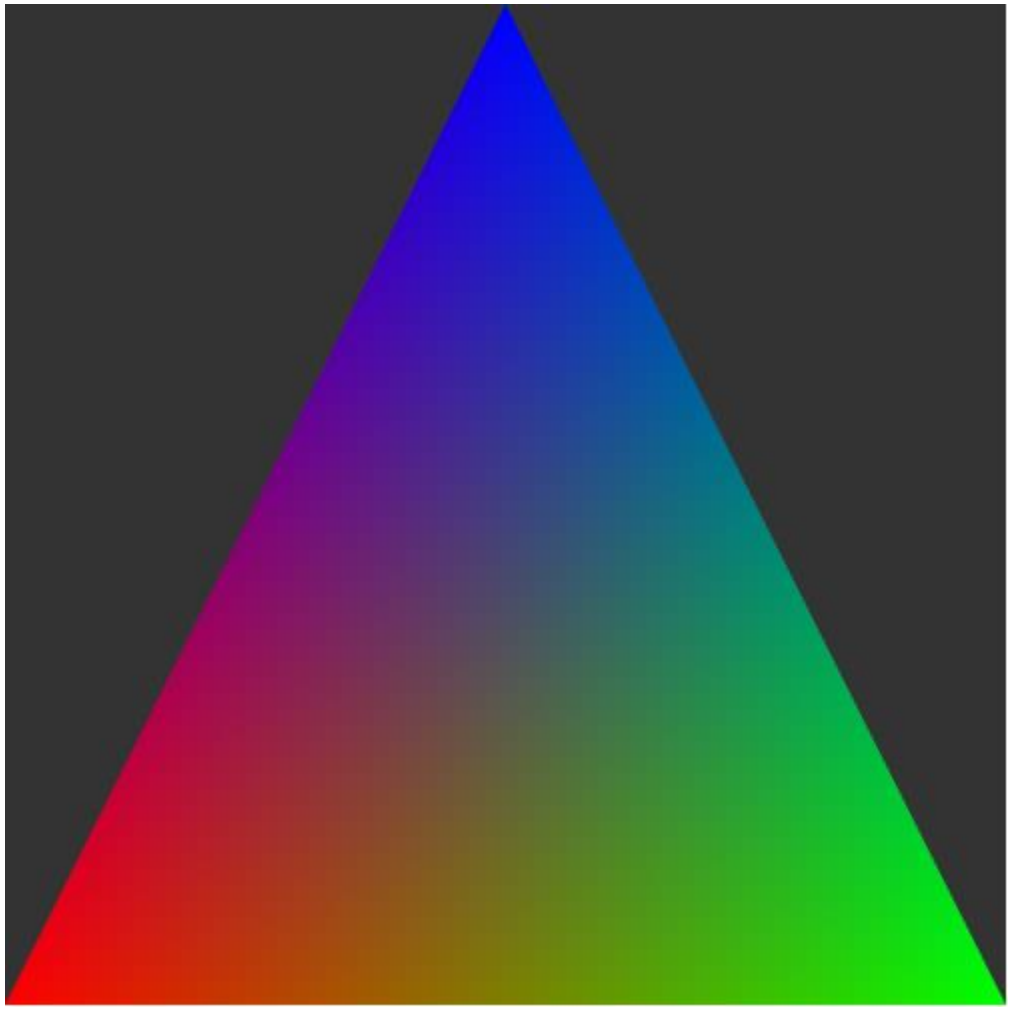
EXERCISE 05

- Passing data between
  the vertex and the fragment shaders



EXERCISE 06

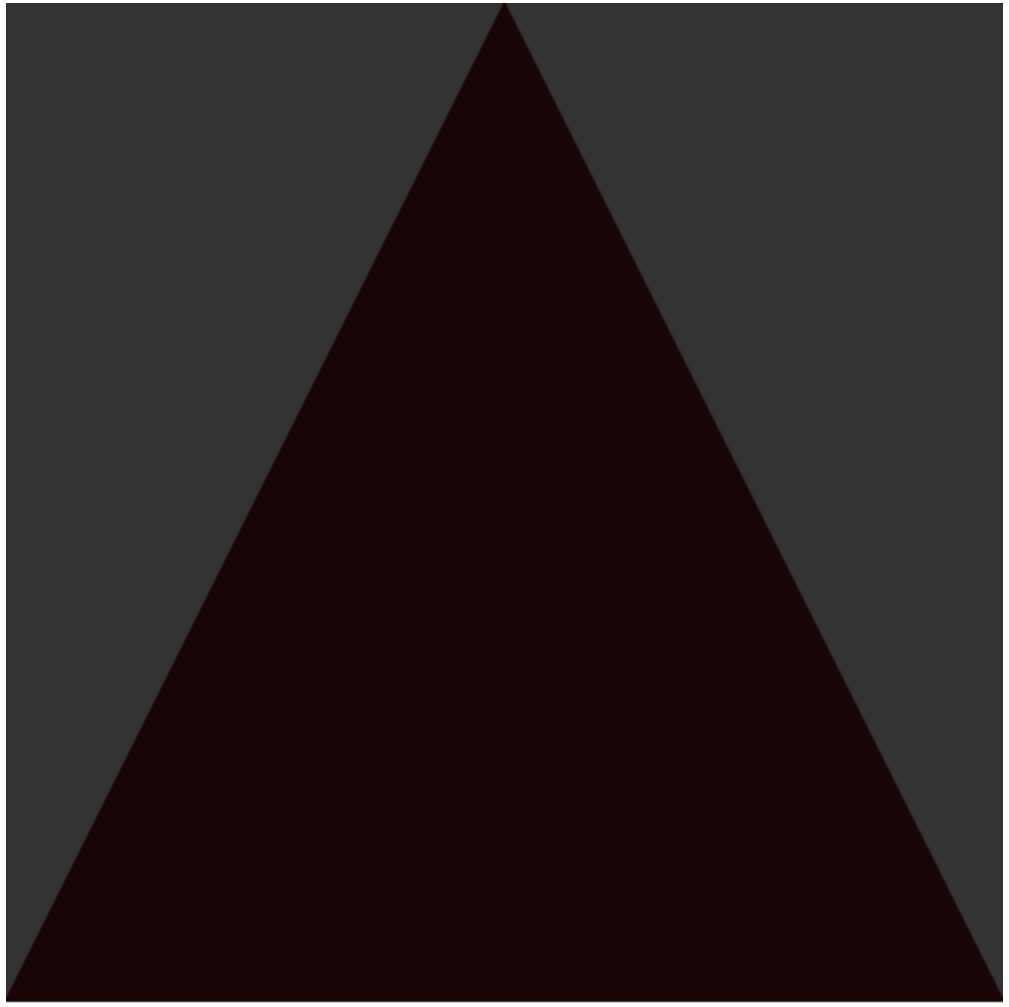
- Passing data from a buffer to the vertex shader to the fragment shader
- Same as Exercise 02 but with a buffer containing the positions AND the colors for each vertex



EXERCISE 07

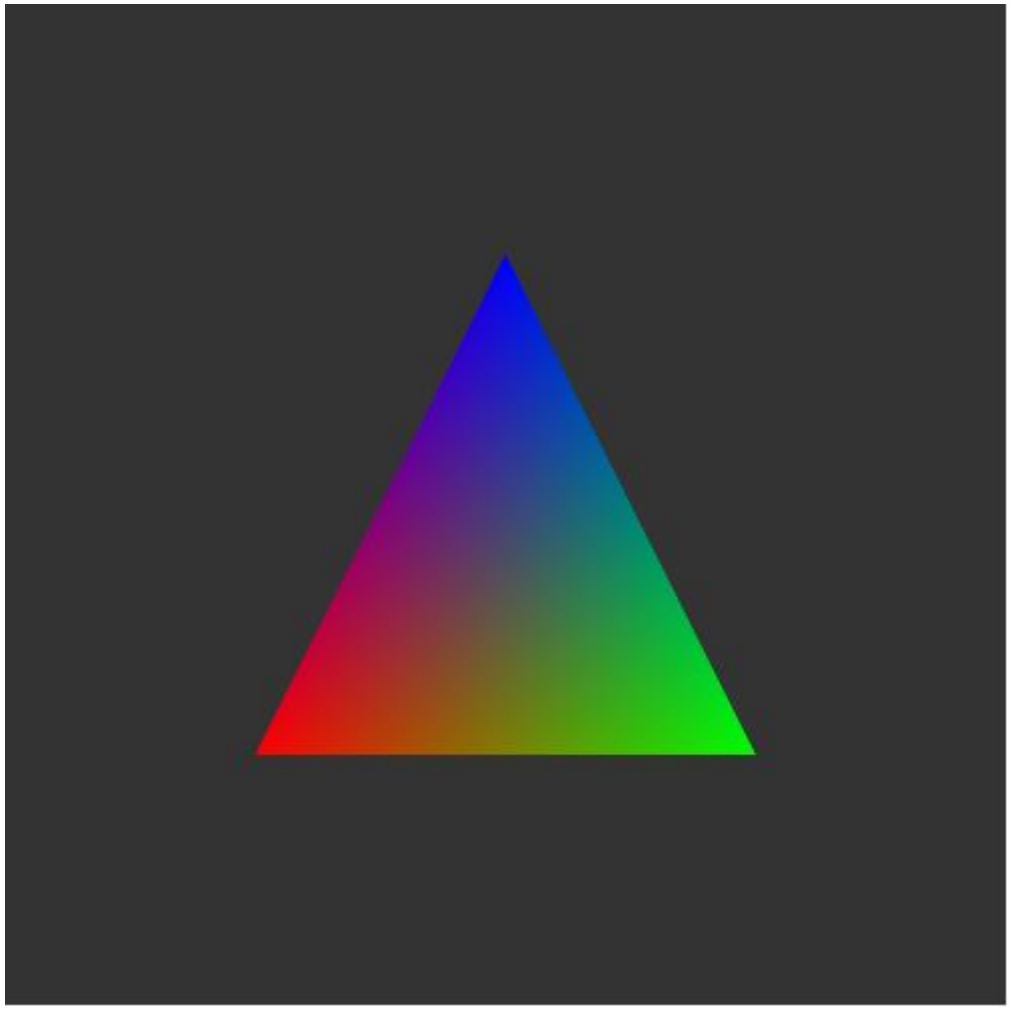
- Make a Shadertoy
  - This time you have everything you need
  - Pass data to the shader during the rendering loop
  - And draw whatever colors in the fragment shader

In the fragment shader you can use gl_FragCoord to access the screen position



EXERCISE 08

- Scale the triangle with linear algebra and the glm library



EXERCISE 09

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translate by - viewDistance in z direction

$$\begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translate origin to rotationCenter

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(rx) & -\sin(rx) & 0 \\ 0 & \sin(rx) & \cos(rx) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotate by rx radians about the x-axis

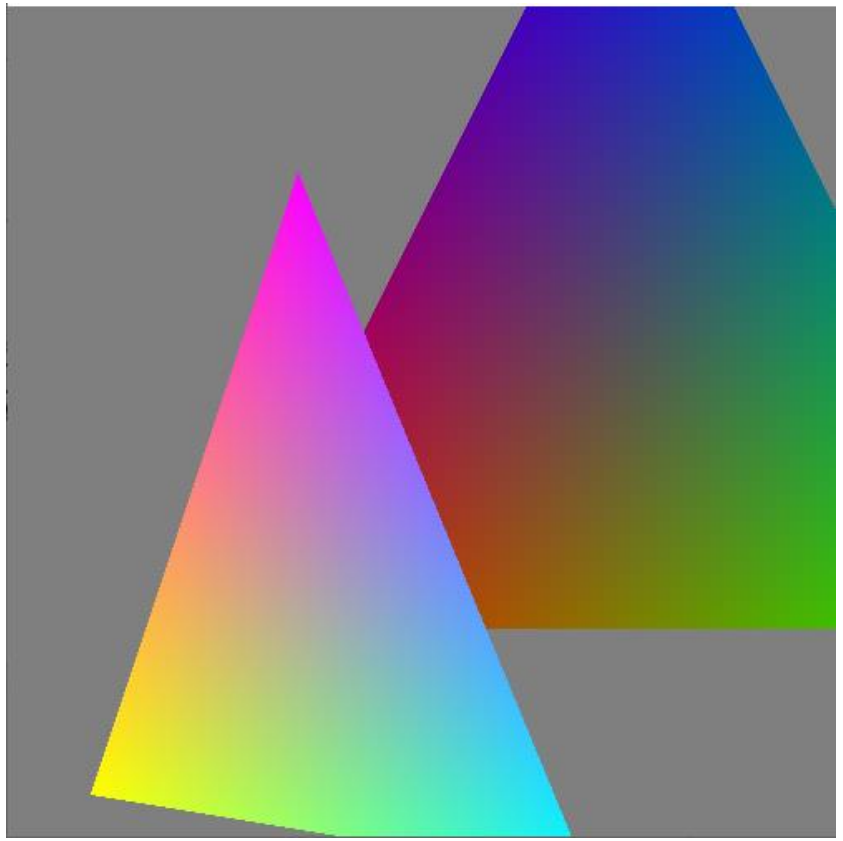
$$\begin{pmatrix} \cos(ry) & 0 & \sin(ry) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(ry) & 0 & \cos(ry) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotate by ry radians about the y-axis

$$\begin{pmatrix} 1 & 0 & 0 & -tx \\ 0 & 1 & 0 & -ty \\ 0 & 0 & 1 & -tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translate rotationCenter to origin

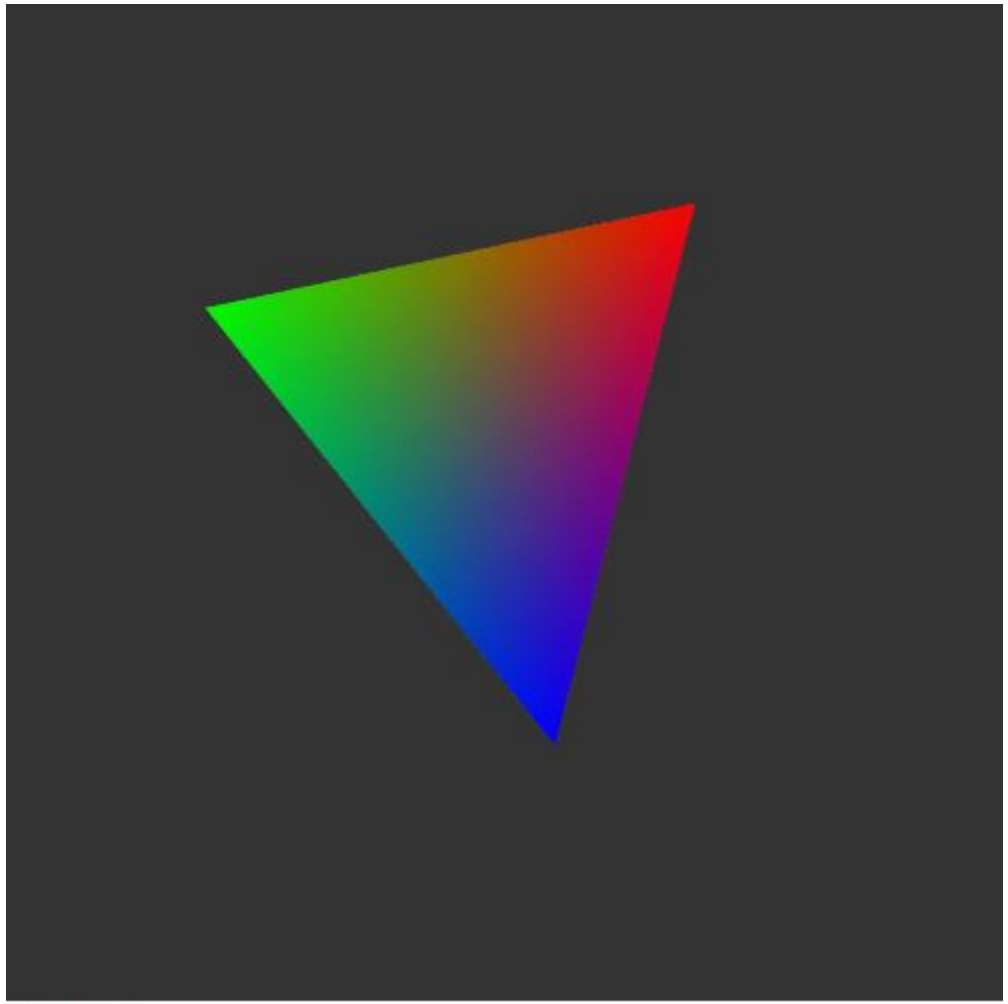
- Have 2 triangles (or other objects) and apply different transformations on them



EXERCISE 10

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Translate by - viewDistance in z direction}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Translate origin to rotationCenter}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(rx) & -\sin(rx) & 0 \\ 0 & \sin(rx) & \cos(rx) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Rotate by rx radians about the x-axis}} \underbrace{\begin{pmatrix} \cos(ry) & 0 & \sin(ry) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(ry) & 0 & \cos(ry) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Rotate by ry radians about the y-axis}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & -tx \\ 0 & 1 & 0 & -ty \\ 0 & 0 & 1 & -tz \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Translate rotationCenter to origin}}$$

- Scale it and make it rotate like hell

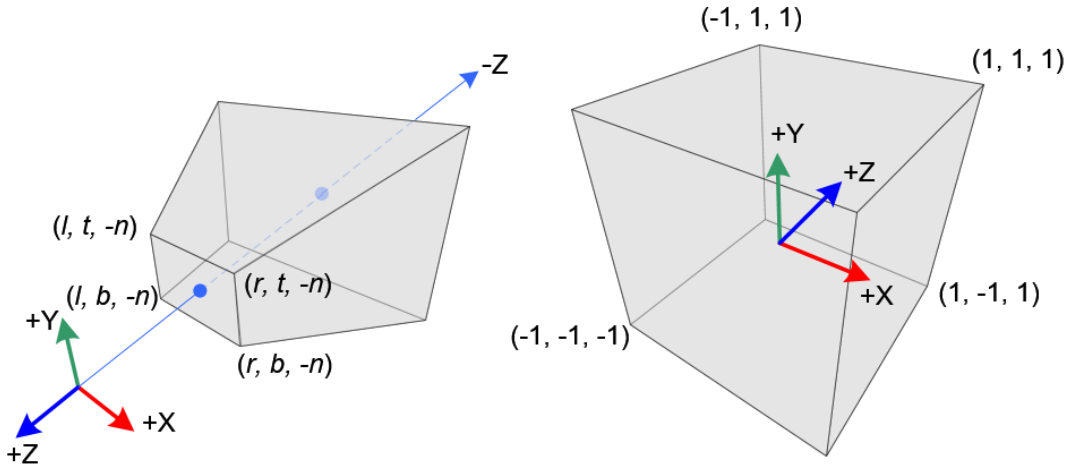


EXERCISE 11

- 3D and perspective
- Introduce Model, View, Perspective matrices
- Look carefully at glm::lookAt and glm::perspective to make them

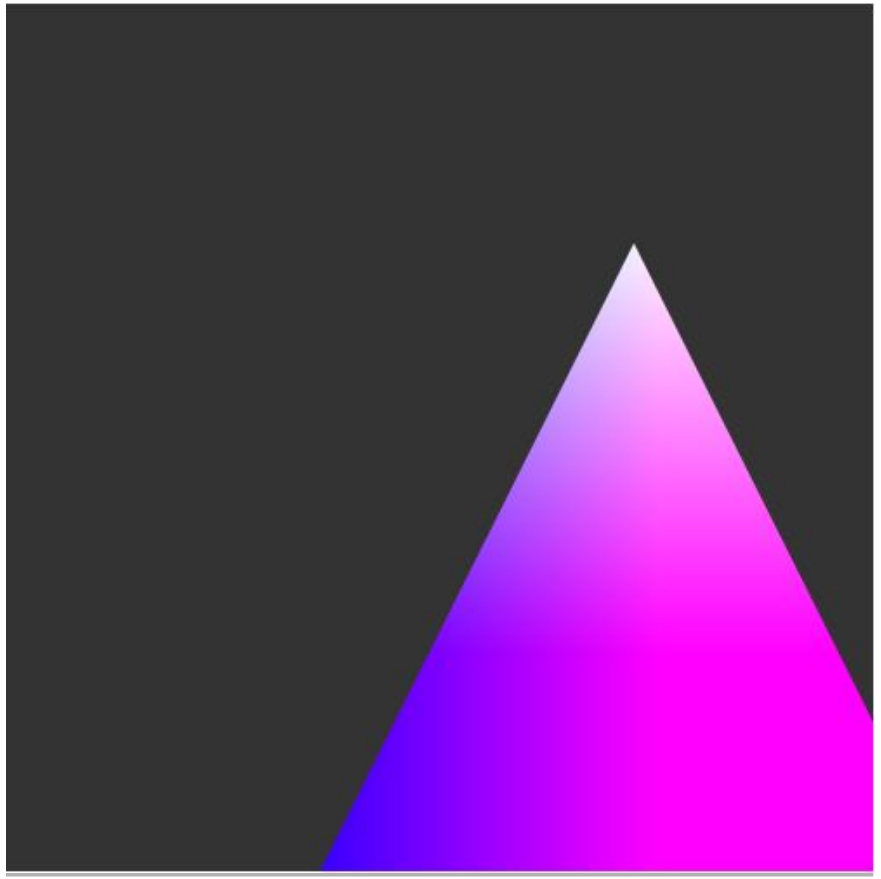
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(rx) & -\sin(rx) & 0 \\ 0 & \sin(rx) & \cos(rx) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(ry) & 0 & \sin(ry) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(ry) & 0 & \cos(ry) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -tx \\ 0 & 1 & 0 & -ty \\ 0 & 0 & 1 & -tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translate by - viewDistance in z direction | Translate origin to rotationCenter | Rotate by rx radians about the x-axis | Rotate by ry radians about the y-axis | Translate rotationCenter to origin



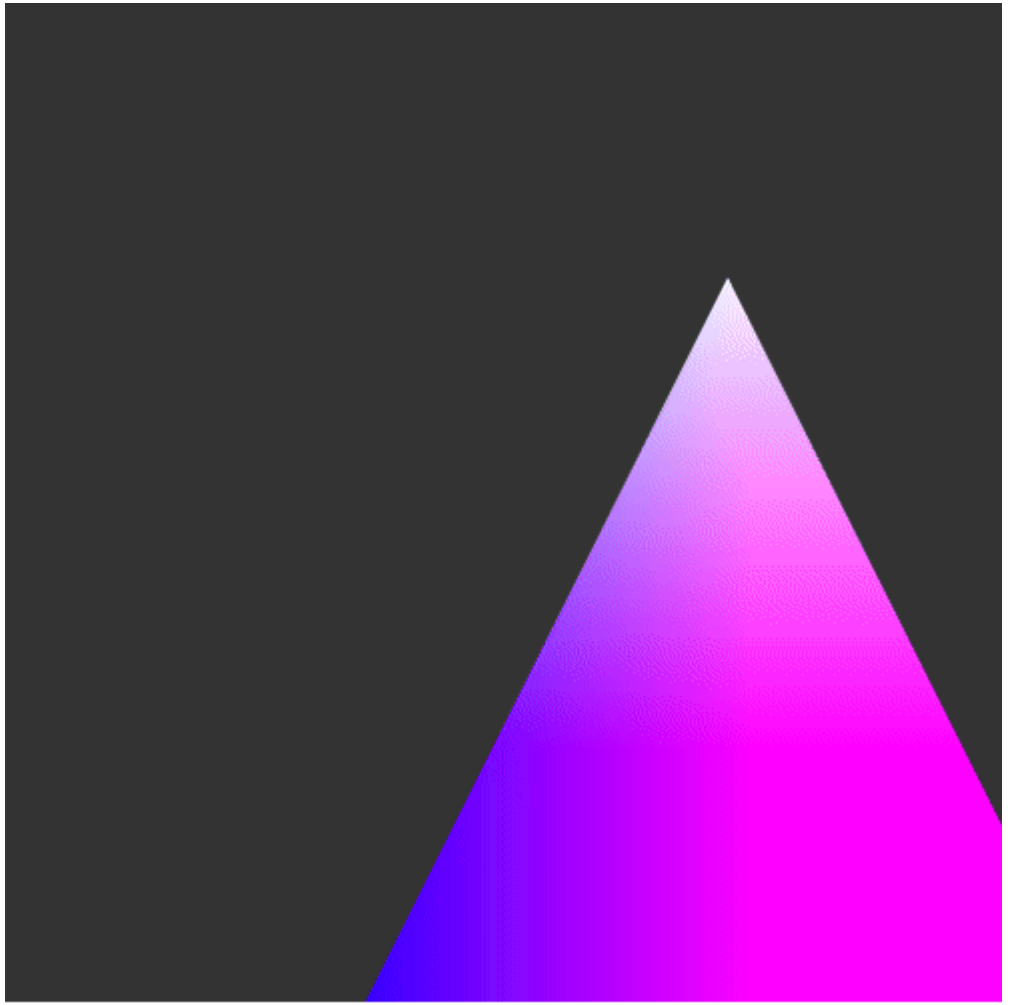
$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$P = \begin{bmatrix} \frac{\cot\frac{fovy}{2}}{aspect} & 0 & 0 & 0 \\ 0 & \cot\frac{fovy}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2*n*f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



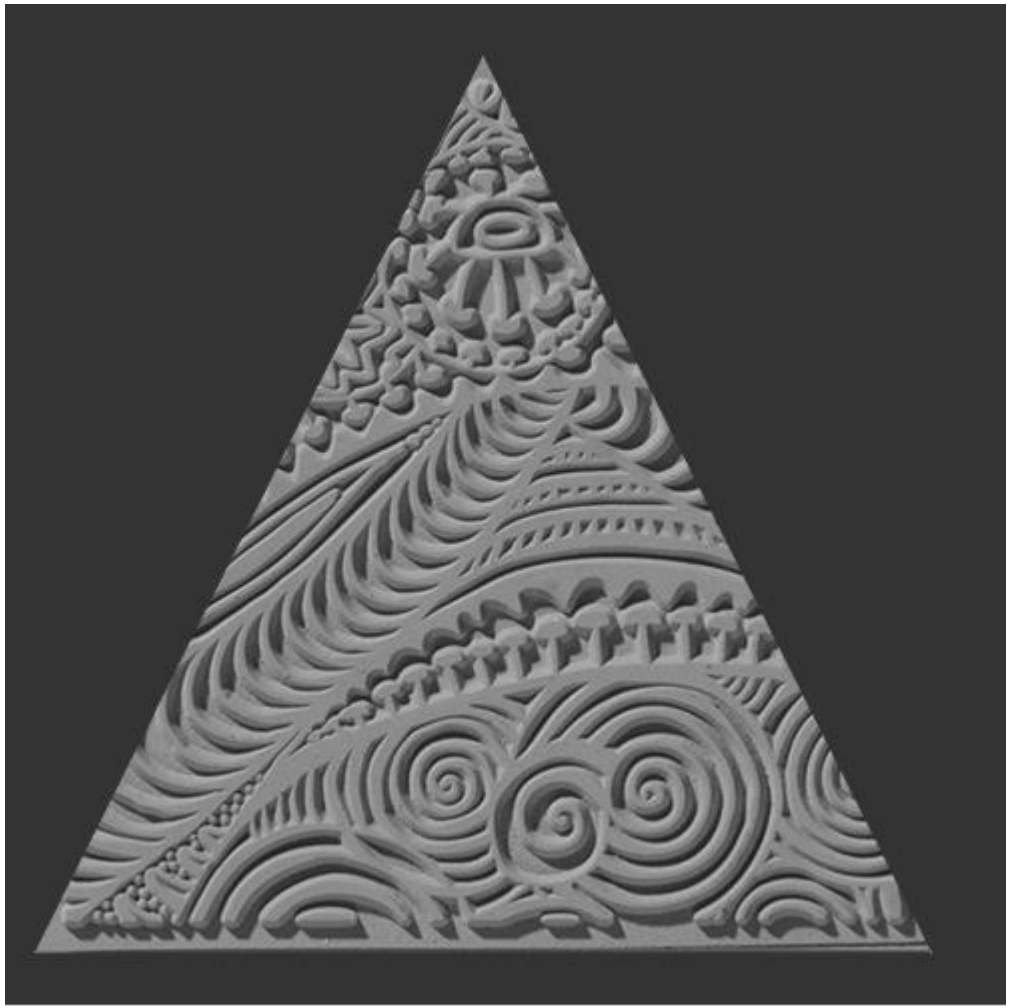
EXERCISE 12

- Use the keyboard to move the camera
  - Look at `glfwGetKey`



EXERCISE 13

- Load a texture
- You will need stb_image library



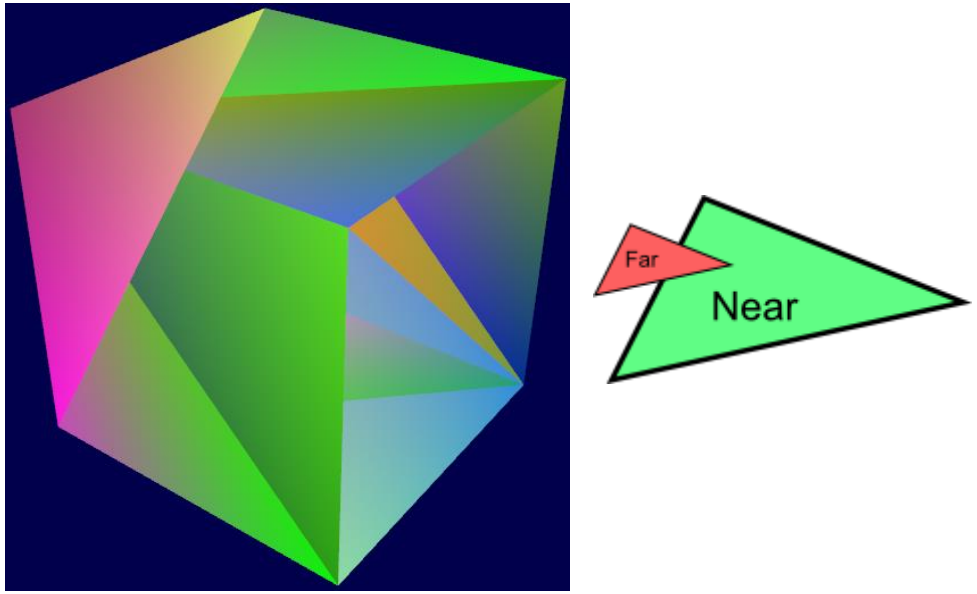
EXERCISE 14

- You source code should start to smell at this point
- It is time to do a good refactoring!
- Encapsulate:
  - The camera and shortcuts
  - The buffers
  - The Shaders
  - The Texture creation
- (PS: You can use object oriented, it’s not javascript)

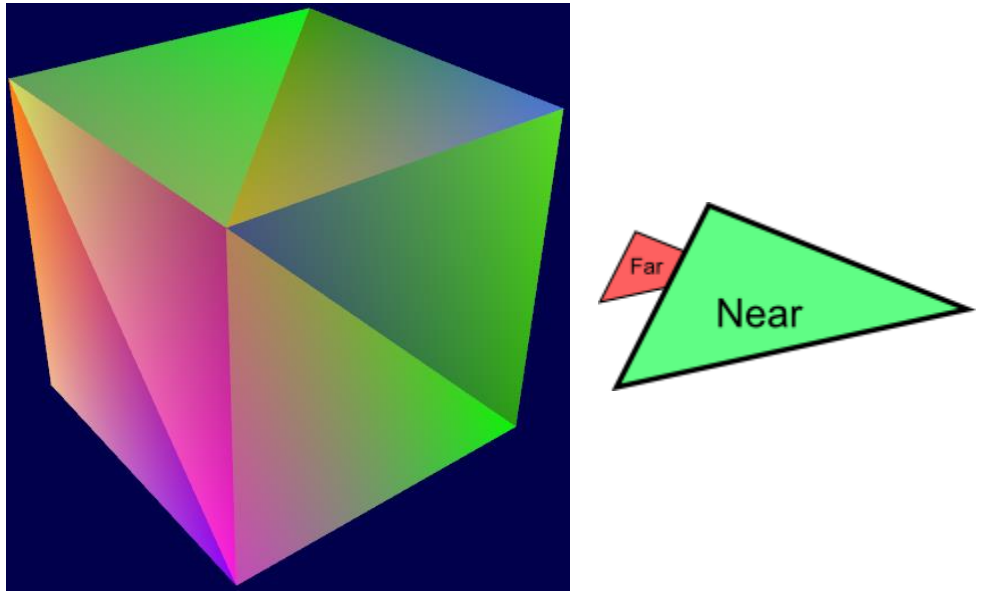
Probably your code if everything continues to be in one single main.cpp file ☺

APPENDIX 01: DEPTH BUFFER

```
gl.enable(gl.DEPTH_TEST);

gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```



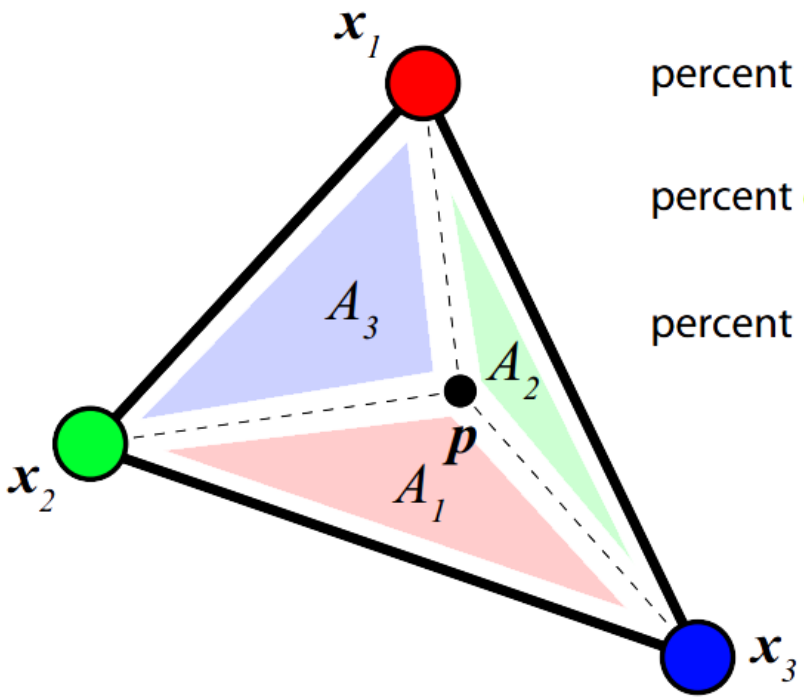
No depth buffer



With depth buffer

APPENDIX 02: COLOR INTERPOLATION (VERTEX TO FRAGMENT)

Barycentric Interpolation



$$\text{percent red} = \frac{A_1}{A} = \lambda_1 = \frac{area(x_2, p, x_3)}{area(x_1, x_2, x_3)}$$

$$\text{percent green} = \frac{A_2}{A} = \lambda_2 = \frac{area(x_1, p, x_3)}{area(x_1, x_2, x_3)}$$

$$\text{percent blue} = \frac{A_3}{A} = \lambda_3 = \frac{area(x_1, p, x_2)}{area(x_1, x_2, x_3)}$$

"barycentric coordinates"

Value at $\boldsymbol{p}$:

$$\sum_i \lambda_i = 1$$

$$(A_1\boldsymbol{x}_1 + A_2\boldsymbol{x}_2 + A_3\boldsymbol{x}_3)/A$$

$$=$$

$$\lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3$$

https://github.com/rarietta/CUDA-Interactive-Rasterizer

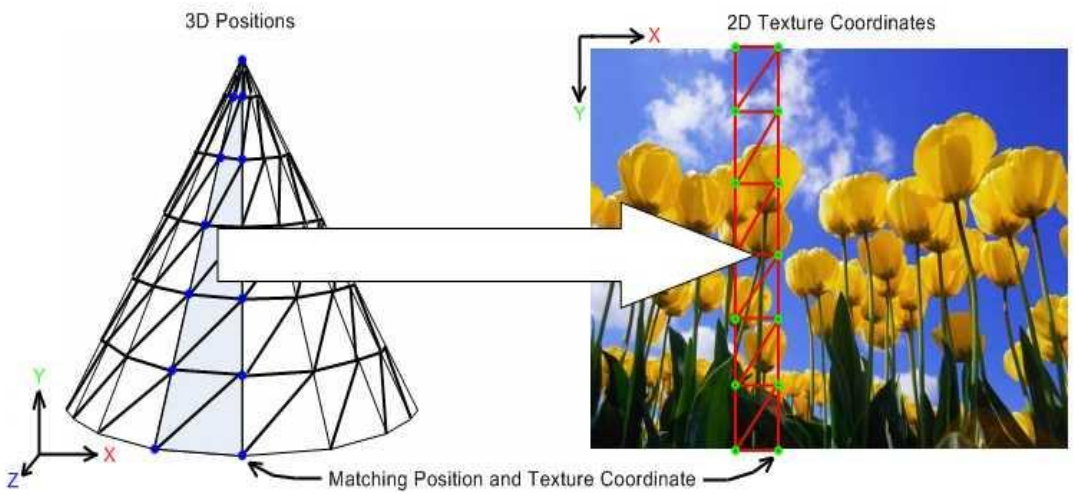
APPENDIX 03: STATE MACHINE

- OpenGL is a large state machine. Each state defines how OpenGL should operate.
- The state of OpenGL is called the **OpenGL context**
- Example (not real code):

```
GLuint objectID = 0;                                  // future pointer to the object in the GPU memory
glGenObject(1, &objectID);                            // make an object of type "Object" and return a pointer
glBindObject(TARGET, objectID);                       // say to OpenGL that we are going to modify the object
glSetObjectOption(TARGET, GL_OPTION1, value);         // set a property
glSetObjectOption(TARGET, GL_OPTION2, value);         // set another property
glBindObject(TARGET, 0);                              // unbind the current object
```

- Internally, this maps to a (hidden) structure in OpenGL:
  struct OpenGL_Context { ... object* object_Target; ... };
- Keep this workflow in mind, you are going to use it all the time
- Knowing that OpenGL is a state machine is not essential, but it can be helpful. A very good and short explanation from the implementation point of view can be found here:
  http://alfonse.bitbucket.org/oldtut/Basics/Intro%20What%20is%20OpenGL.html
  (file: What is OpenGL - Data Structure.docx)

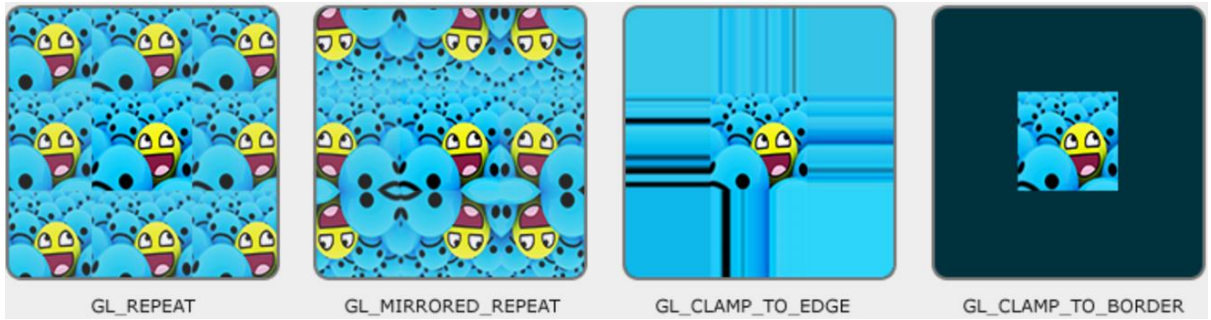
APPENDIX 04: HOW ARE TEXTURES MADE?



- Texturing can be hard!
- We rely on 3D softwares (eg: Blender) to make a texture mapping to a 3D object

- Maths: Heckbert, Survey of Texture Mapping, 1986, http://dx.doi.org/10.1109/MCG.1986.276672

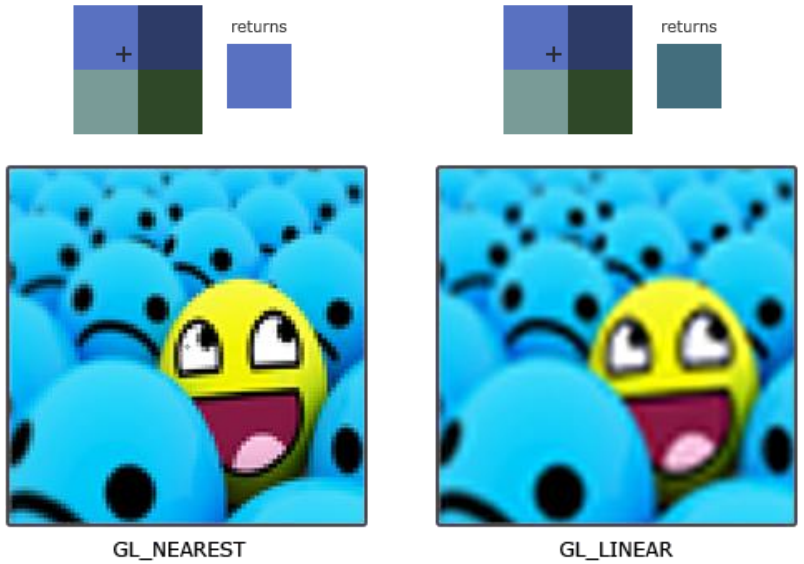
APPENDIX 05: TEXTURE WARPING



- Texture coordinates are in [0,1] range. What happens if you assign coordinates out of range?
- We use the texture parameters functions to choose the behaviour
  - GL_TEXTURE_WRAP_S or GL_TEXTURE_WRAP_T (textures are in (s,t) coordinates)

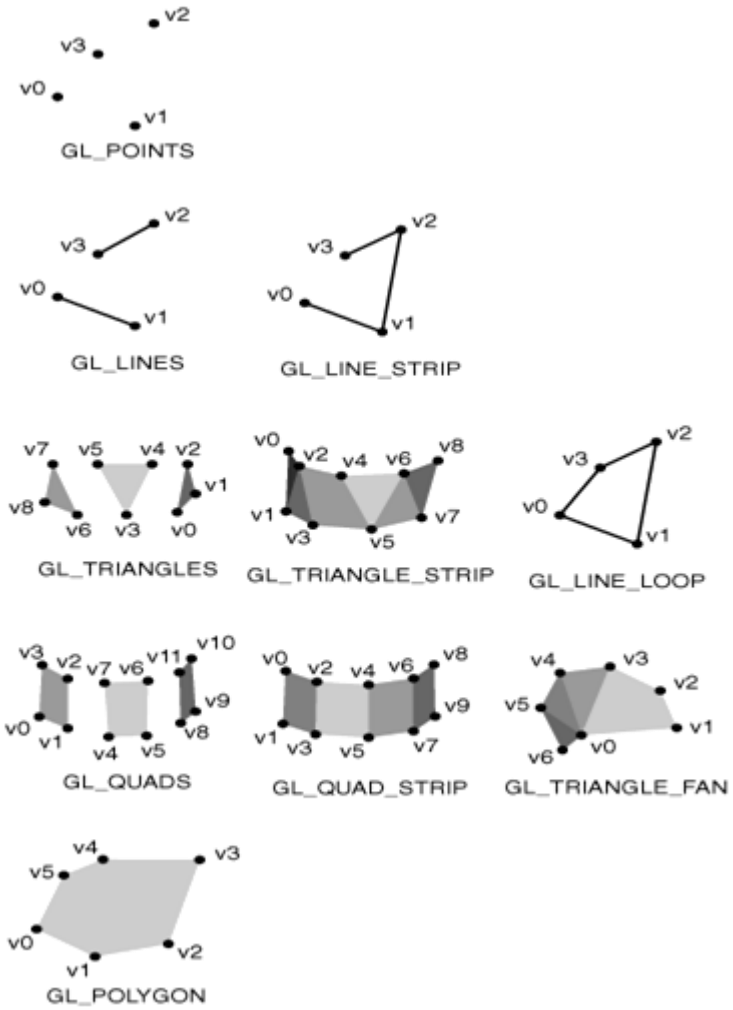
APPENDIX 06: TEXTURE FILTERING

- What happens if we (up/down)scale the image?
- We define it using the other texture parameters
  - GL_TEXTURE_MIN_FILTER or GL_TEXTURE_MAG_FILTER
- We can define several ways of scaling, eg:
  - Bilinear filtering
  - Nearest Neighbor
  - ...



ADVANCED APPENDIX 01: DRAWING FUNCTIONS

- Usually we draw triangles, however **gl.drawArrays** can take several different primitives
- Why?
  - Some softwares model meshes using quads or other primitives by default
- However, ultimately, everything becomes triangles



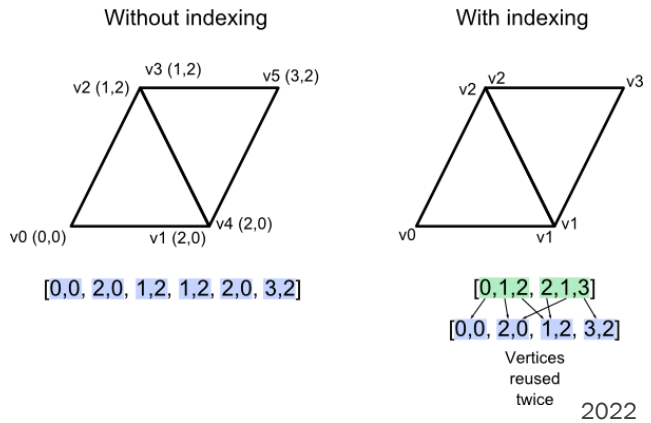
ADVANCED APPENDIX 02: ELEMENT BUFFER OBJECT (EBO)

- At the moment we use gl.drawArrays()
  - This function takes as input a buffer where each line is a vertex
  - However in a real mesh, the vertex will appears several times
- Gl.drawElements() (EBO Element Buffer Objects)
  - Let us draw using two buffers, one with unique vertices
  - The second with only the indices of which vertex is linked to which other

```
vert =Float32Array([
   // First triangle
    0.5f,  0.5f, 0.0f,  // TR
    0.5f, -0.5f, 0.0f,  // BR
   -0.5f,  0.5f, 0.0f,  // TL
   // Second triangle
    0.5f, -0.5f, 0.0f,  // BR
   -0.5f, -0.5f, 0.0f,  // BL
   -0.5f,  0.5f, 0.0f,  // TL
]);
```

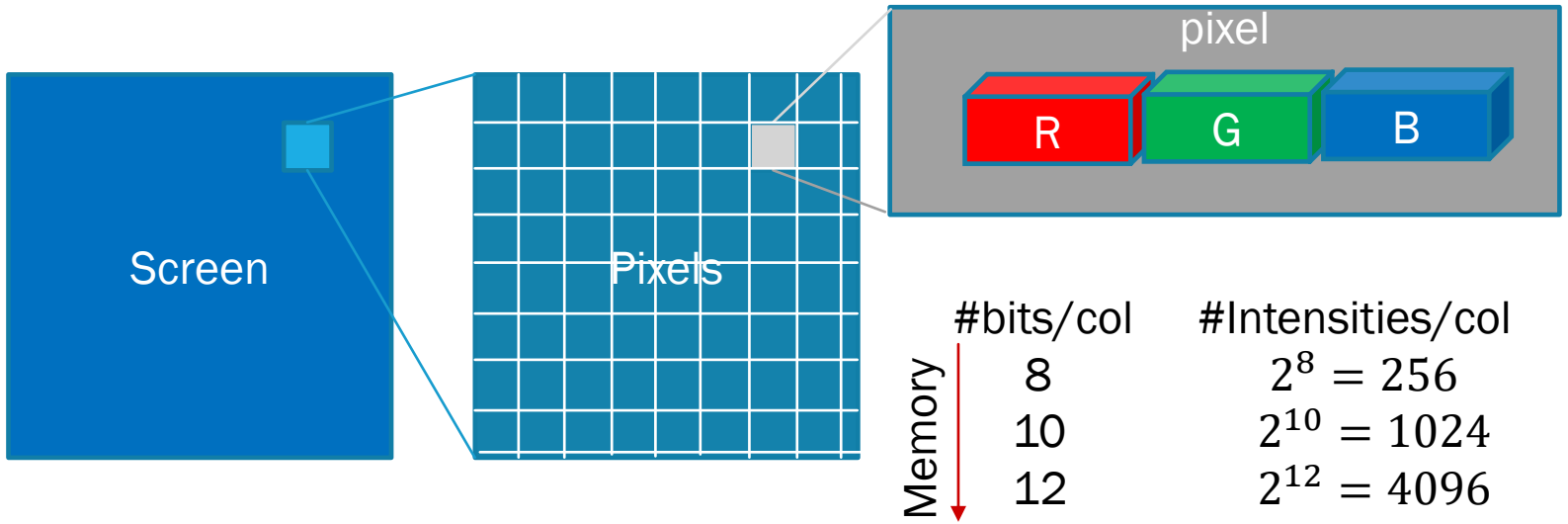
→

```
vert = Float32Array([
    0.5f, 0.5f, 0.0f, // TR
    0.5f, -0.5f, 0.0f, // BR
   -0.5f, -0.5f, 0.0f, // BL
   -0.5f, 0.5f, 0.0f]); // TL
indices = Float32Array([
    // Note that we start from 0!
    0, 1, 3, // First Triangle
    1, 2, 3, // Second Triangle
]);
```



ADVANCED APPENDIX 03: BUFFER PACKING

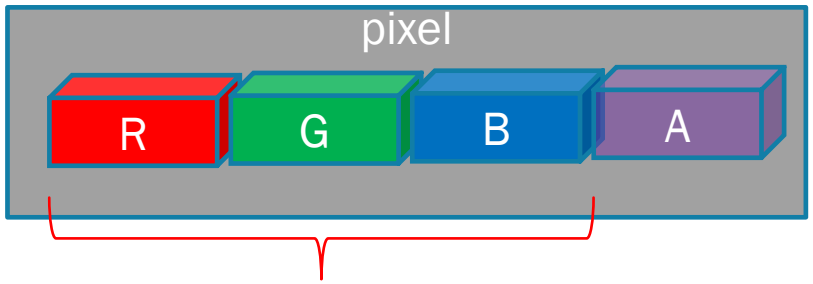
From the beginning we are talking about buffers, but, what do they contain?



Memory	#bits/col	#Intensities/col
	8	$2^8 = 256$
	10	$2^{10} = 1024$
	12	$2^{12} = 4096$

Adding alpha makes a vector more memory friendly!

Memory	#bits/Pixel	Total Colors
	$3 * 8 = 24$	$2^{24} = 16.7M$
	$3 * 10 = 30$	$2^{30} = 1B$
	$3 * 12 = 36$	$2^{36} = 69B$

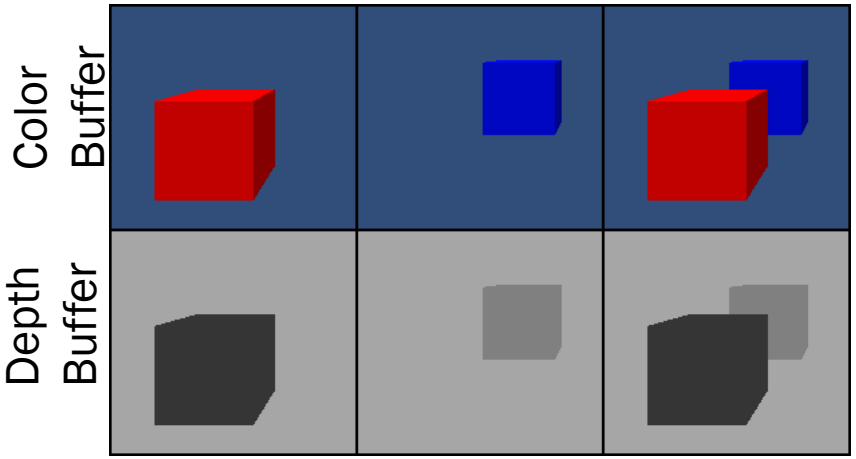


8 + 8 + 8 = 24 bits

Slides on buffers, colors, alpha component, double-buffers, lights and Rasterisation inspired by:
SIGGRAPH University 2014 – Fundamentals Seminar by Mike Bailey (Oregon State University)

ADVANCED APPENDIX 04: Z-BUFFER

- Beside the color buffer, we also have the Z-buffer or depth buffer
- Used for hidden surface removal
- Holds pixels depth
- Typically 32bits deep
- Integer or float point



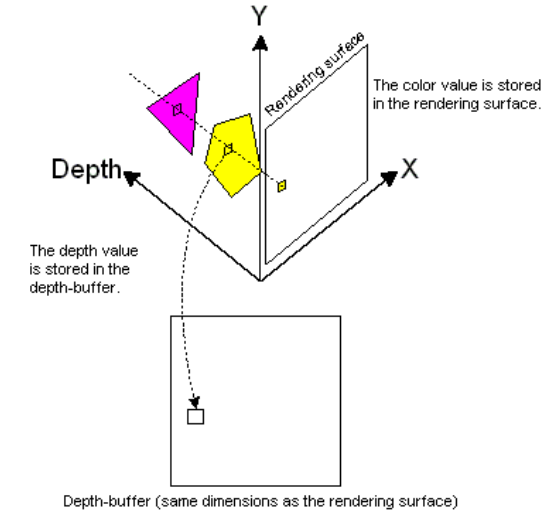
3-4 components

1 component!

#bits/Z	Total Z values
32	$2^{32} = 4B$

https://jakobknudsen.wordpress.com/2013/07/20/transparency-and-sorting/

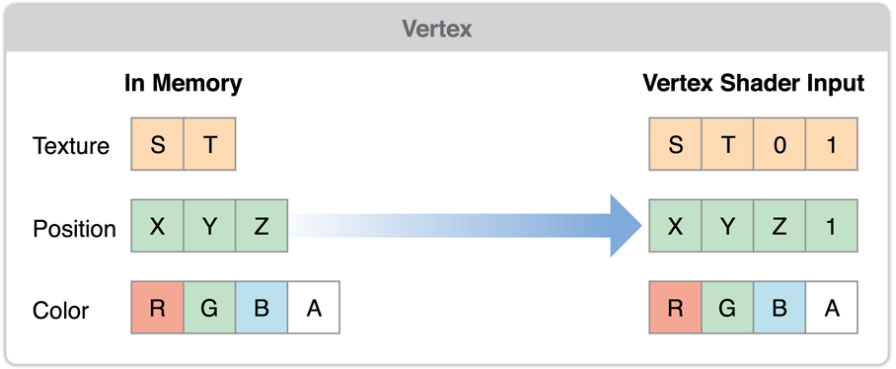
Whatever is closest to the camera should occupy the pixels.



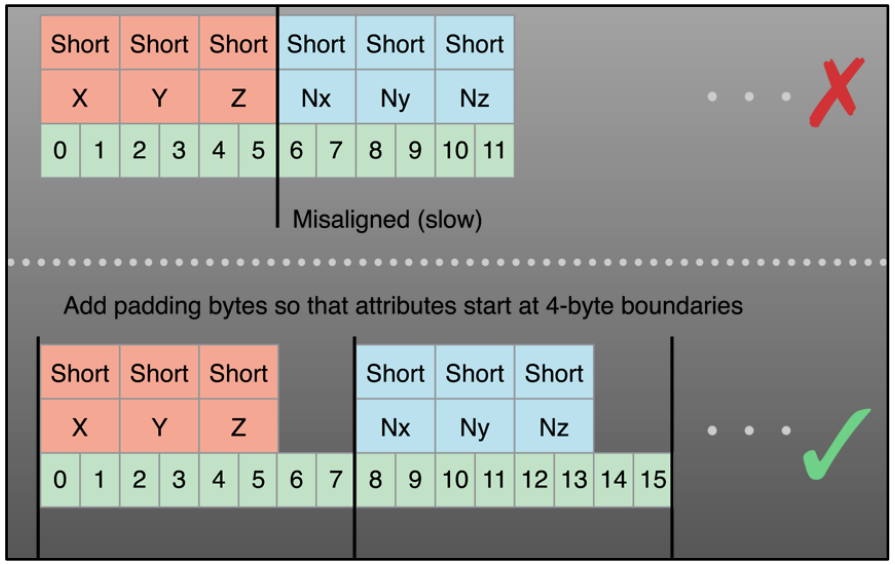
https://msdn.microsoft.com/en-us/library/windows/desktop/bb219616(v=vs.85).aspx

As you render, the pipeline put the depth value in the depth buffer. If there is already a depth here, it will compare if the new object has a lower depth. If it is the case, it replaces the color. Else it will disallow the color and depth writing

ADVANCED APPENDIX 05: DATA TRANSFER RAM TO VRAM



The data is transformed in 4 components. The fourth component is often useful to pass data between shaders without creating a new variable
eg: max value to normalize in the fragment pass

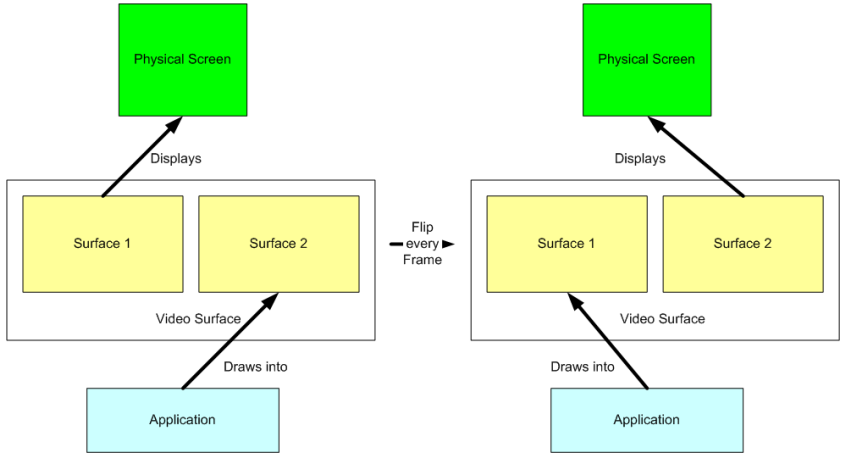


Try to avoid misaligned data!

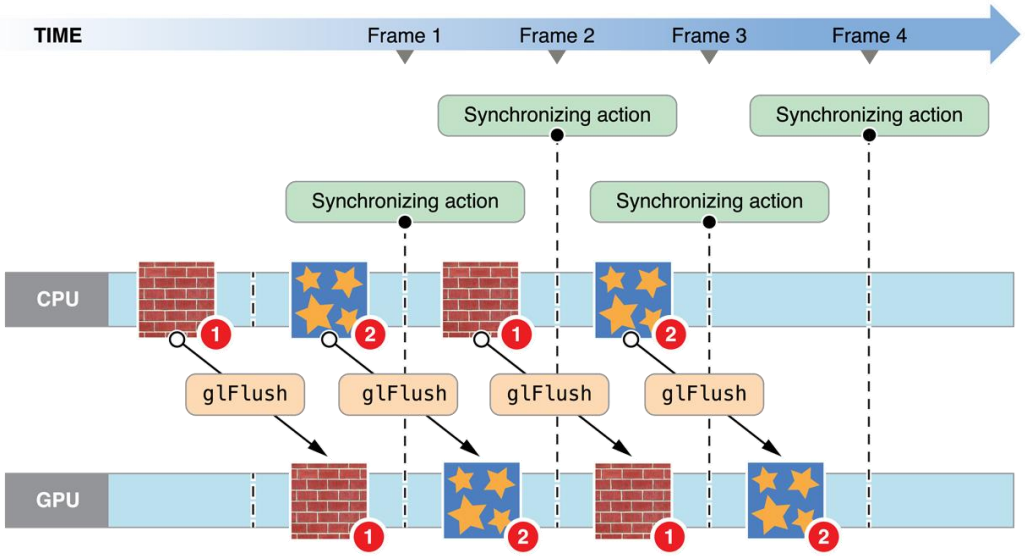
https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html

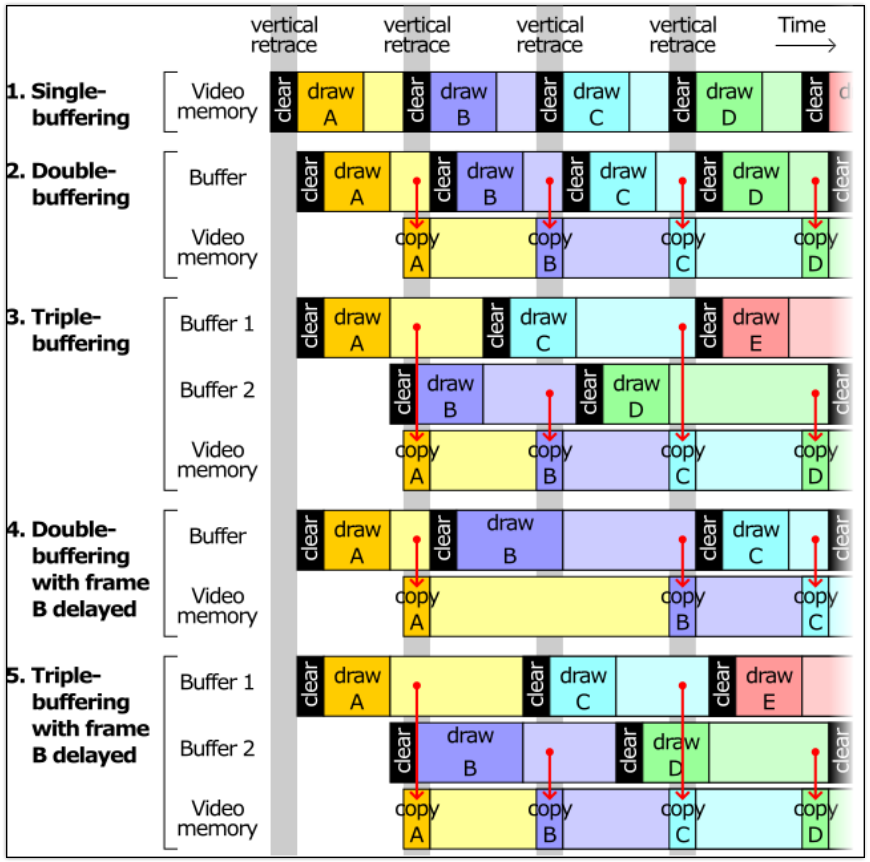
ADVANCED APPENDIX 08: N-BUFFERING

We never render directly to the screen, we render to a buffer and then we swap the screen buffer with it (Dual buffering)



http://ryanallanca.com/2014/12/





https://en.wikipedia.org/wiki/Multiple_buffering

https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESApplicationDesign/OpenGLESApplicationDesign.html

ADVANCED APPENDIX 09: USE THE SHADERS FOR COMPUTING

- Modern algorithms do arithmetics on the framebuffer color components instead of doing rendering!
- Compute shader
  - loop that can read from the fragment shader and send the data again to the vertex shader without going through the CPU

Interactive Cloth Simulation
https://www.youtube.com/watch?v=KBfxnayIIOY

Particle Systems
Real Time 3D Fluid and Particle Simulation and Rendering
https://www.youtube.com/watch?v=RuZQpWo9Qhs

Rigid Body & Fluid Dynamics
https://www.youtube.com/watch?v=aLwYWW2N16w

ANNEXE 10: HTML AND JAVASCRIPT

```
1   <!DOCTYPE html>
2   <html>
3   <head>
4   </head>
5   <body>
6       <canvas id="webgl_canvas" width="500" height="500"></canvas>
7       <script>
8       </script>
9   </body>
10  </html>
```

Basic canvas to show WebGL

Javascript code goes here

- HTML is a markup language (we specify the elements by tags): https://www.w3schools.com/html/
- Javascript (JS) let us interact with HTML and make animations: https://javascript.info/
- WebGL is the OpenGL for the web, it interacts nicely with JS and is shown in a canvas element in HTML: https://webglfundamentals.org/
- The links point to very nice tutorials, bookmark them, you'll need them!