Computer Graphics
– Shaders

J.-Prof. Dr. habil. Kai Lawonn

Introduction

- Shaders are little programs that rest on the GPU
- These programs are run for each specific section of the graphics pipeline
- Shaders are nothing more than programs transforming inputs to outputs
- Shaders are isolated programs, they cannot communicate with each other; only via their inputs and outputs

GLSL

GLSL

- Shaders are written in the C-like language GLSL and tailored for graphics
- Begin with a version declaration, followed by a list of input and output variables, uniforms and its main function
- Start at its main function where input variables and output are processed

GLSL

- A shader typically has the following structure:

```
#version version_number

in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;
void main()
{
    // process input(s) and do some weird graphics stuff
    ...
    // output processed stuff to output variable
    out_variable_name = weird_stuff_we_processed;
}
```

GLSL

- Vertex shader each input variable also known as a vertex attribute
- Maximum number of vertex attributes allowed to declare limited by the hardware
- OpenGL guarantees there are always at least 16 4-component vertex attributes available
- Some hardware allow more, can be retrieved with GL_MAX_VERTEX_ATTRIBS:

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " << nrAttributes;
```

Types

Types

- GLSL has data types specifying what kind of variable to work with
- GLSL has most of the default basic types:
  - int
  - float
  - double
  - uint
  - bool
- GLSL also features two types, we will using a lot: vectors and matrices

Vectors

- Vector is a 1,2,3 or 4 component container
- Can take the following form (n represents the number of components):
  - vecn: the default vector of n floats
  - bvecn: a vector of n Booleans
  - ivecn: a vector of n integers
  - uvecn: a vector of n unsigned integers
  - dvecn: a vector of n double components
- Mostly: vecn since floats are sufficient for most purposes
- Components can be accessed via:
  - vec.x, _.y, _.z and _.w to access their first, second, third and fourth component respectively
- GLSL also allows use rgba for colors or stpq for texture coordinates (accessing the same components)

Swizzling

- The vector datatype allows for some interesting and flexible component selection called swizzling
- Swizzling allows for the following syntax:

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

Swizzling

- Any combination of up to 4 letters to create a new vector is possible
- But, It is not allowed to access the .z component of a vec2
- Can also pass vectors as arguments to different vector constructor calls, reducing the number of arguments required:

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```

Ins and Outs

Ins and Outs

- Shaders can have inputs and outputs and they can be passed
- GLSL defined the in and out keywords specifically for that purpose
- Each shader can specify inputs and outputs using those keywords
- If an output variable matches with an input variable of the next shader stage, they're passed along

Ins and Outs

- The vertex shader differs in its input: receives its input straight from the vertex data on the CPU
- The input variables is organized with location metadata:
  - E.g.: layout (location = 0)

Ins and Outs

- The other exception is that the fragment shader requires a vec4 color output variable (fragment shaders needs a final output color)
- If you'd fail to specify an output color in your fragment shader OpenGL will render your object black (or white)

Ins and Outs

- To send data from one shader to the other:
  - 1. Declare an output in the sending shader
  - 2. Declare an input in the receiving shader
- Types and the names need to be equal! (Otherwise OpenGL will not link them)

Ins and Outs

- Example from the triangle example:

```
#version 330 core
layout (location = 0) in vec3 aPos;
void main()
{
   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0); //works: vec4(aPos, 1.0);
}
```

```
#version 330 core
out vec4 FragColor;
void main()
{
   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Ins and Outs

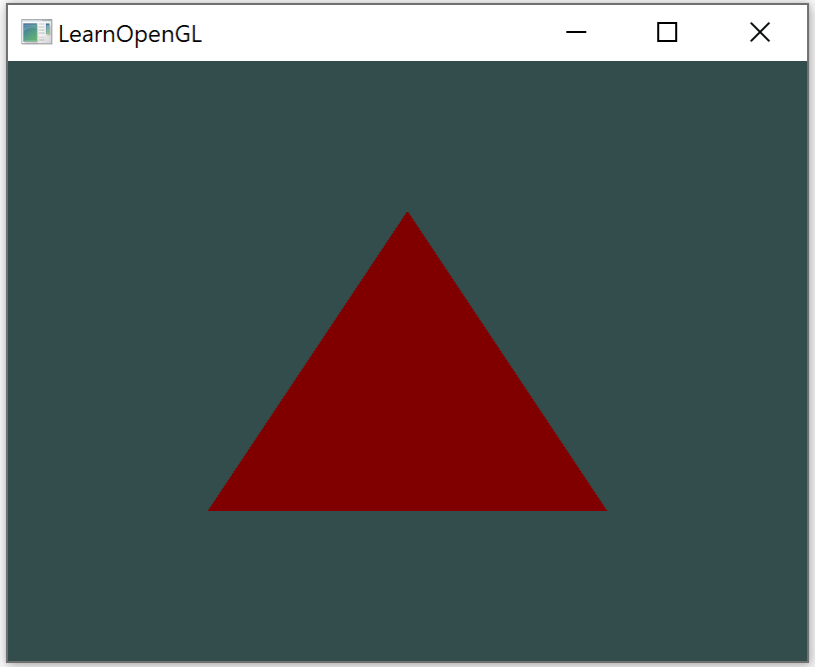
- Example from the triangle example:

```
#version 330 core
layout (location = 0) in vec3 aPos;
out vec4 vertexColor;
void main()
{
   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0); //works: vec4(aPos, 1.0);
   vertexColor = vec4(0.5, 0.0, 0.0, 1.0);
}
```

```
#version 330 core
out vec4 FragColor;
in vec4 vertexColor;
void main()
{
   FragColor = vertexColor;
}
```

F5

- We passed a variable along and get:



Uniforms

Uniforms

- Uniforms are another way to pass data from the CPU to the shaders on the GPU
- Uniforms are slightly different to vertex attributes:
  - Uniforms are global → uniform variable is unique per shader program object
  - Can be accessed from any shader at any stage in the shader program
  - Uniforms will keep their values until reseted or updated (on the CPU)

Uniforms

- To declare a uniform in GLSL, simply add the uniform keyword to a shader with a type and a name
- Then it can be used:

```
#version 330 core
out vec4 FragColor;
uniform vec4 ourColor;
void main()
{
   FragColor = ourColor;
}
```

Uniforms

- Uniforms are global variables, can be defined in any shader
- No need to go through the vertex shader again to get something to the fragment shader
- Not using this uniform in the vertex shader so there’s no need to define it there

Uniforms

If a declared uniform isn't used anywhere, the compiler will silently remove the variable from the compiled version → cause for several frustrating errors;

Keep this in mind!

Uniforms

- The uniform is currently empty
- First need to find the index/location of the uniform attribute in our shader
- Once we have the index/location of the uniform, we can update its values
- Let's change the color over time (render loop):

```
glUseProgram(shaderProgram);

float timeValue = glfwGetTime();
float greenValue = sin(timeValue) / 2.0f + 0.5f; //greenValue in [0,1]

int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

Uniforms

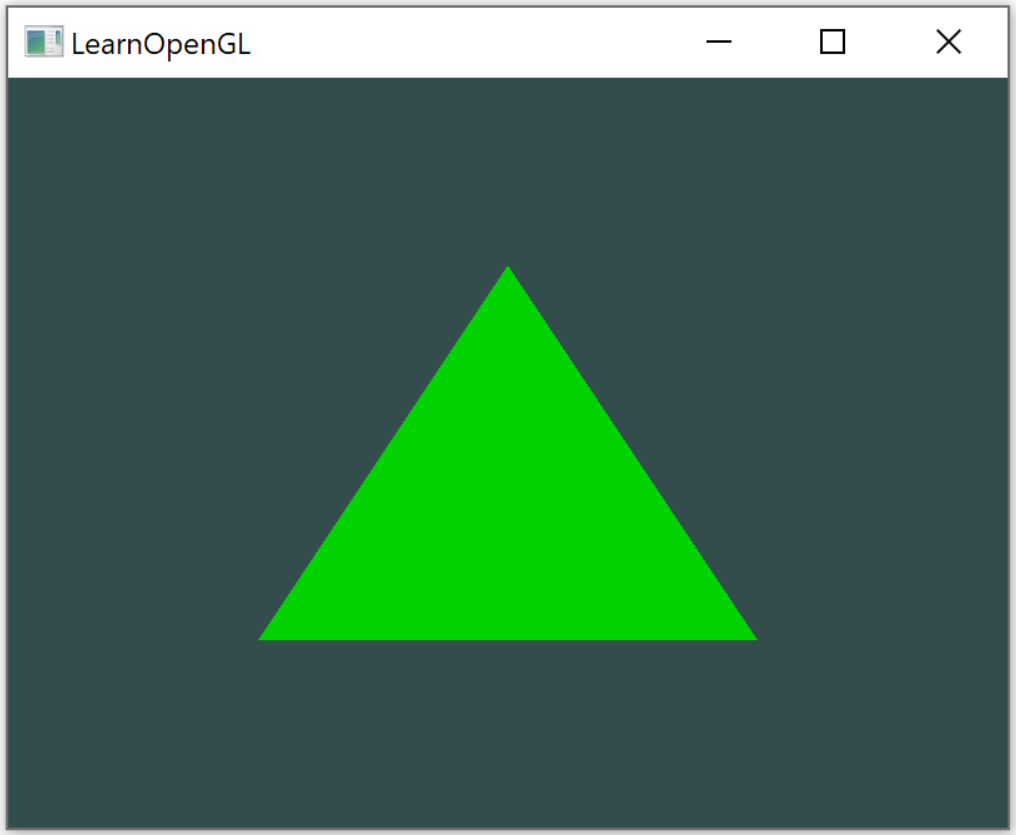
- First, retrieve the running time in seconds via glfwGetTime() and set the greenValue in [0,1]
- Then, query for the location of the ourColor uniform using glGetUniformLocation
- If glGetUniformLocation returns -1, it could not find the location
- Lastly set the uniform value using the glUniform4f function
- (Note: finding the uniform location does not require to use the shader program first, but updating a uniform does require you to first use the program because it sets the uniform on the currently active shader program)

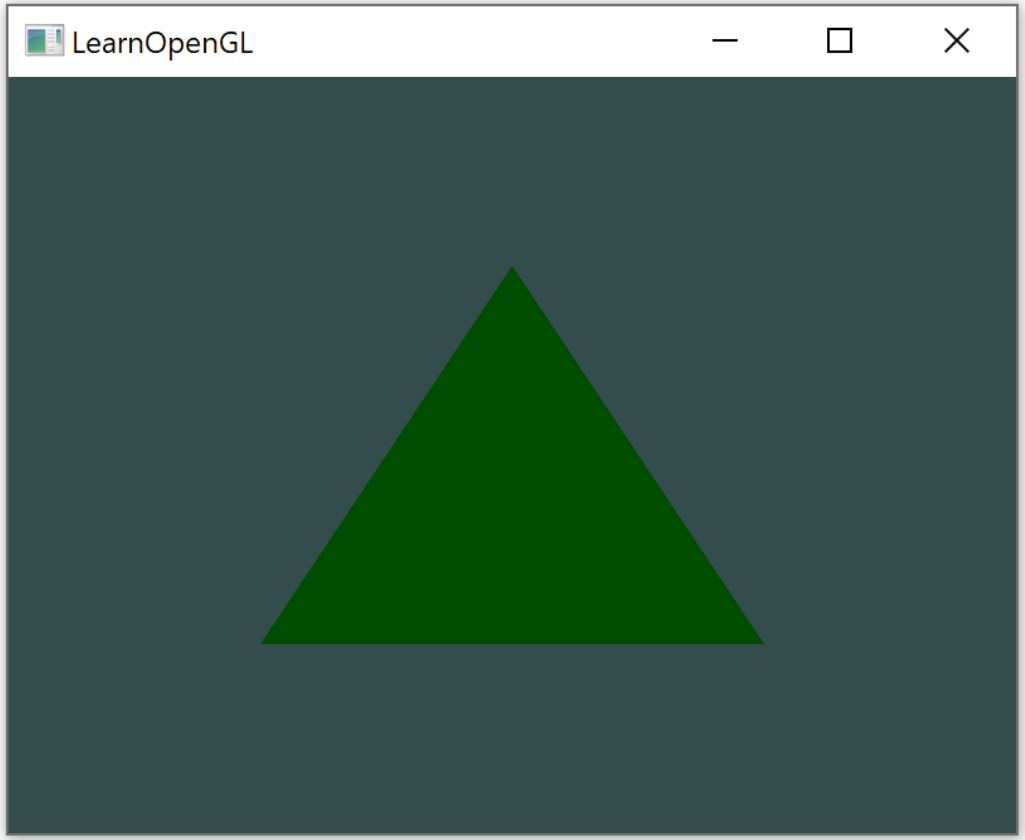
```
glUseProgram(shaderProgram);

float timeValue = glfwGetTime();
float greenValue = sin(timeValue) / 2.0f + 0.5f; //greenValue in [0,1]

int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

Uniforms

```
while (!glfwWindowShouldClose(window))
{
        processInput(window);
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
        glUseProgram(shaderProgram);

         // update shader uniform
         float timeValue = glfwGetTime();
         float greenValue = sin(timeValue) / 2.0f + 0.5f;
         int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
         glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);

        glDrawArrays(GL_TRIANGLES, 0, 3);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
```

F5





Uniforms

OpenGL is in its core a C library: does not have native support for type overloading.

A function can be called with different types and OpenGL defines new functions for each type required;

Example glUniform: function requires a specific postfix for the type of the uniform:

- **f: the function expects a float as its value**
- **i: the function expects an int as its value**
- **ui: the function expects an unsigned int as its value**
- **3f: the function expects 3 floats as its value**
- **fv: the function expects a float vector/array as its value**

Uniforms

- Uniforms are a useful for setting attributes that might change in render iterations, or for interchanging data
- What if we want to set a color for each vertex?
- In that case we'd have to declare as many uniforms as we have vertices
- A better solution: include more data in the vertex attribute

More attributes!

More attributes!

- Previously, we filled a VBO, configured vertex attribute pointers and stored it all in a VAO
- Now, add color data to the vertex data as 3 floats:

```
float vertices[] = {
    // positions         // colors
    0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  // bottom right
   -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  // bottom left
    0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f   // top

};
```

More attributes!

- More data require to send to it the vertex shader:
- Set the location of the aColor attribute to 1 with the layout specifier:

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has attribute position 1
out vec3 ourColor; // output a color to the fragment shader
void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // set ourColor to the input color we got from the vertex data
}
```

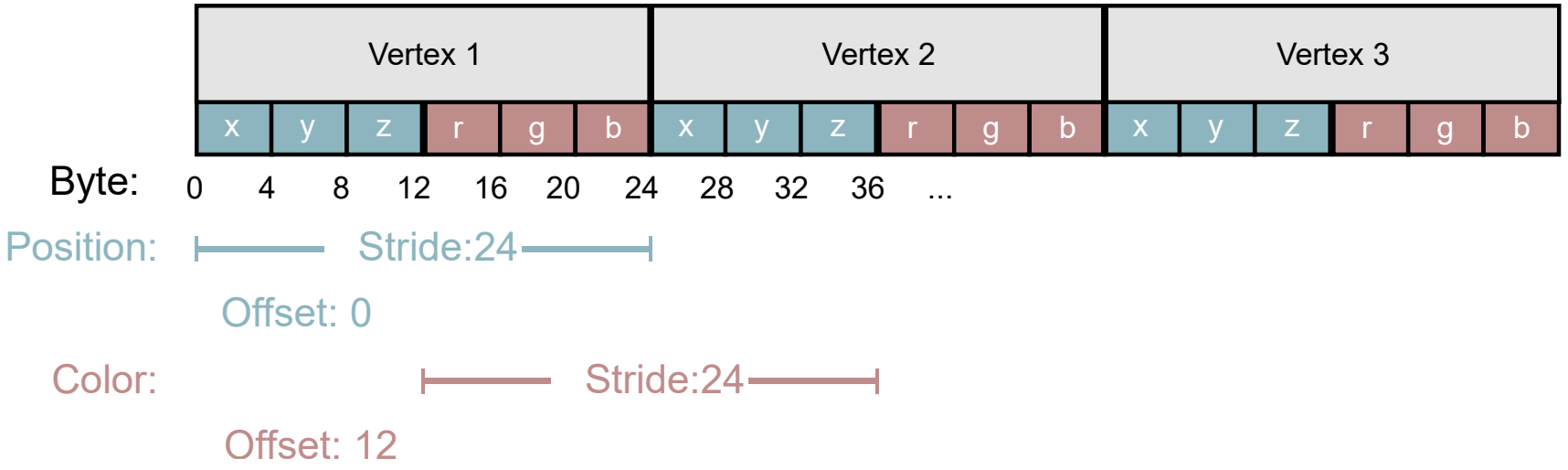
More attributes!

- No longer use a uniform for the fragment's color, but ourColor as output:

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
void main()
{
    FragColor = vec4(ourColor, 1.0);
}
```

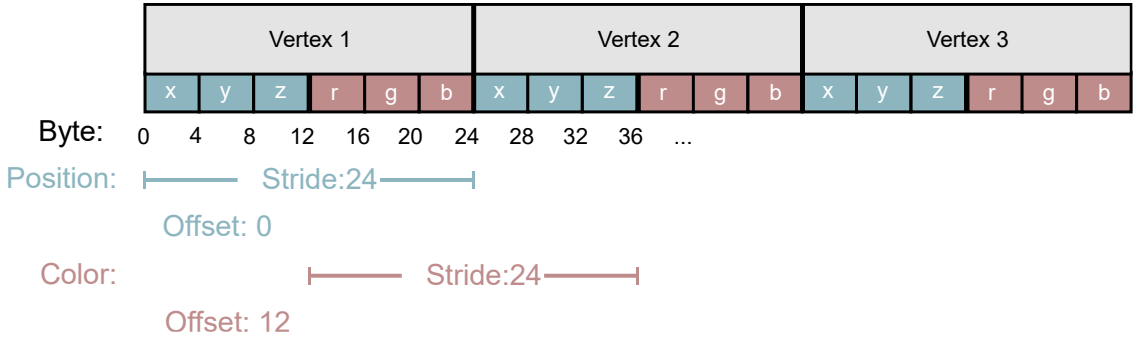
More attributes!

- Because of the added vertex attribute and updated VBO's memory → re-configure the vertex attribute pointers:




```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

More attributes!

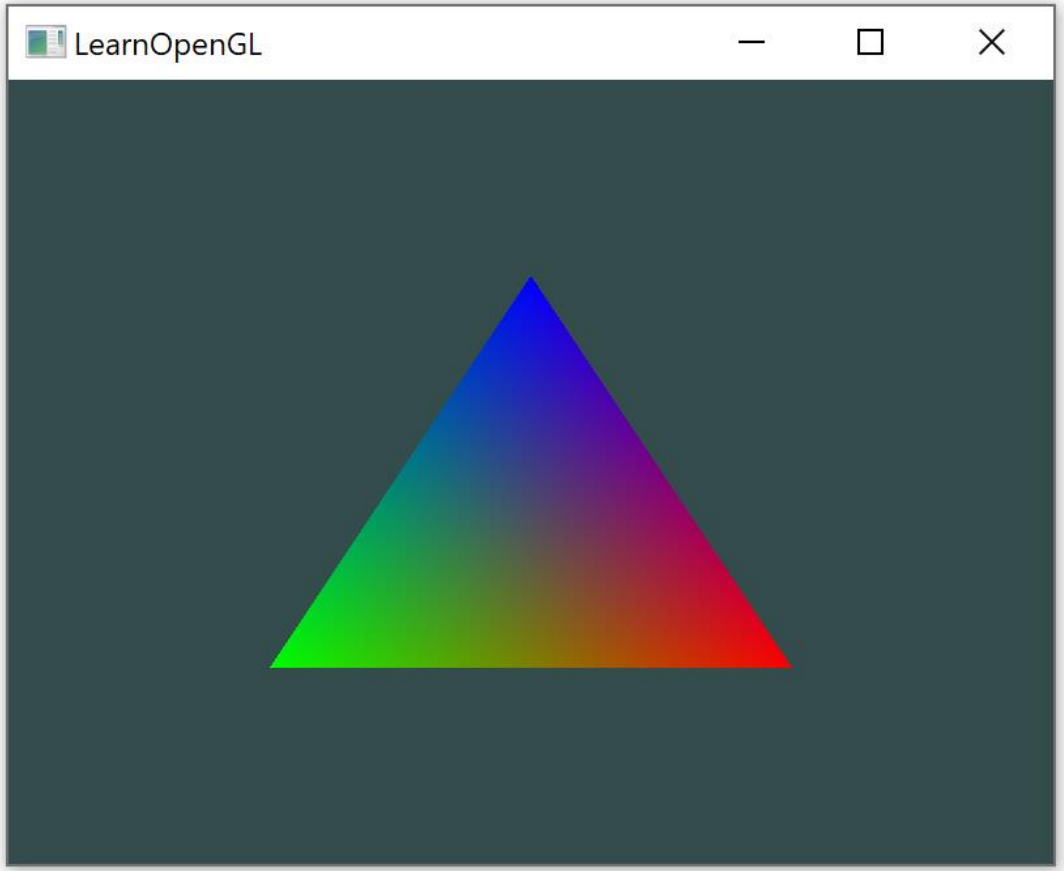


- Configuring the vertex attribute on attribute location 1
- Two vertex attributes, so re-calculate the stride value: next attribute move 6 floats to the right (3 for position, 3 for color) → stride of 6 x size of a float in bytes (= 24 bytes)
- Also, specify an offset:
  - Position vertex offset of 0
  - Color starts after the position data so offset is 3 * sizeof(float) in bytes (= 12 bytes)

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

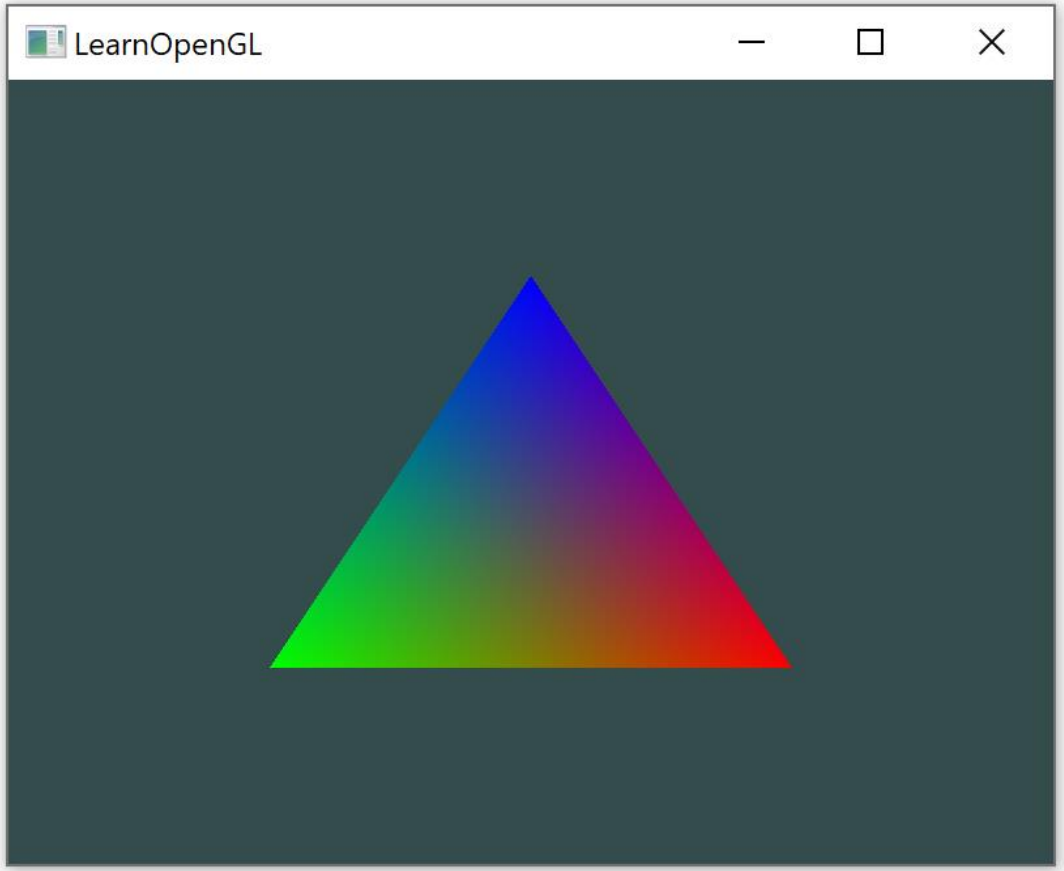
F5…

- …colorful



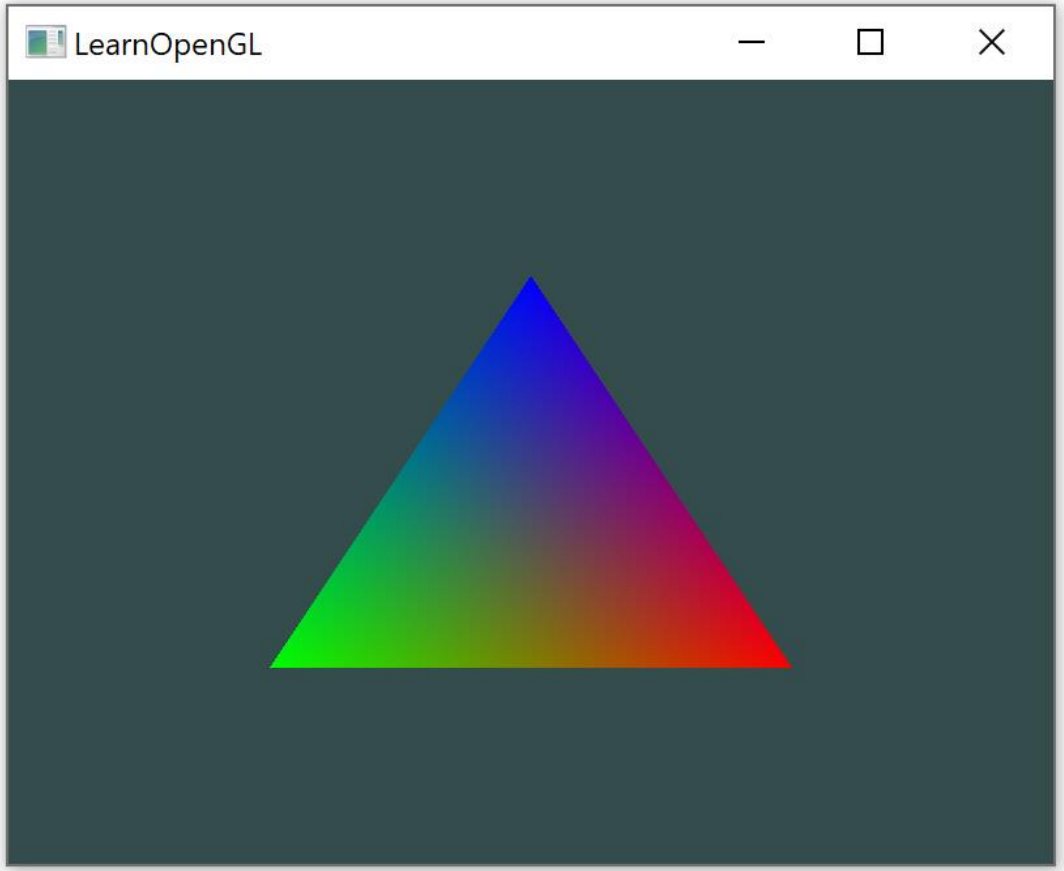
F5…

- Only supplied 3 colors, but the result looks different
- What happened?



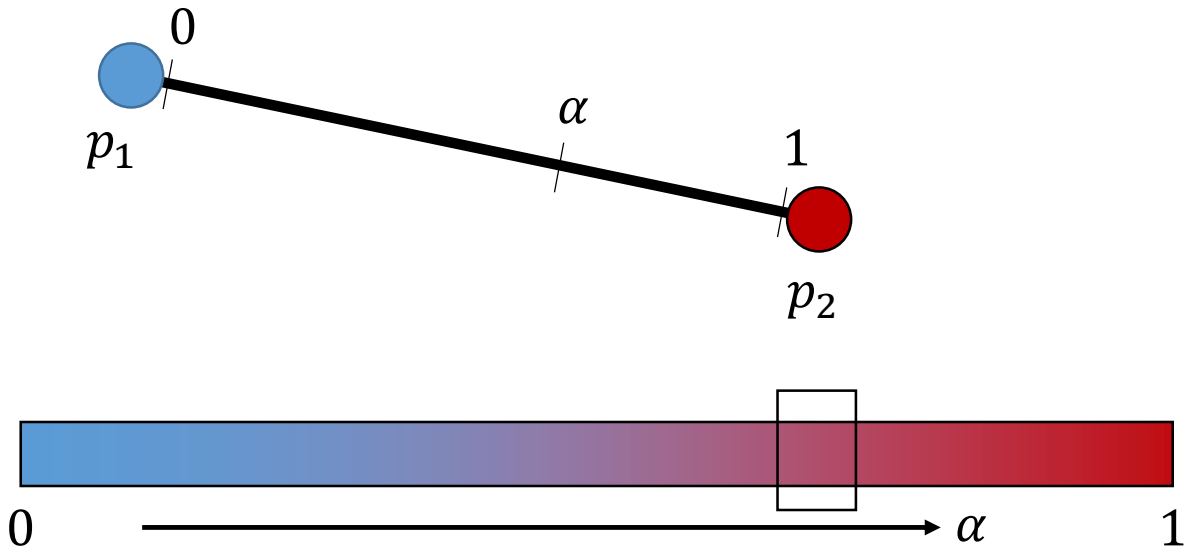
Interpolation

- This is all the result of fragment interpolation in the fragment shader
- When rendering a triangle the rasterization stage usually results in a lot more fragments than vertices originally specified
- The rasterizer determines the positions of each of those fragments based on where they reside on the triangle shape
- Based on these positions, it interpolates all the fragment shader's input variables



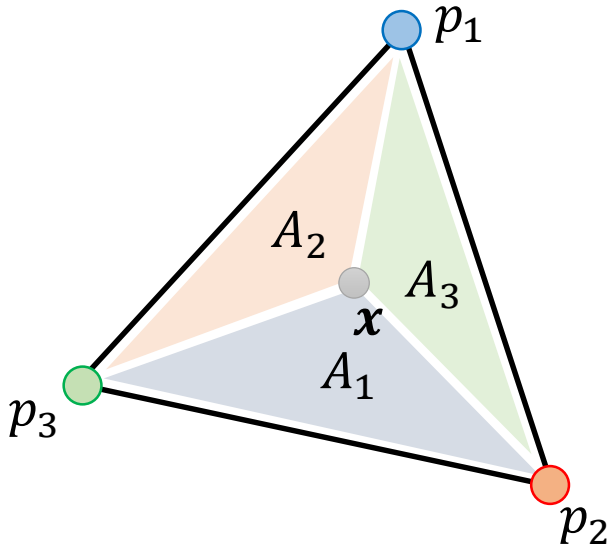
Interpolation

- But let's solve a simpler problem first
- We want to define a color for every $\alpha \in [0, 1]$



Data interpolation

- Barycentric interpolation:



$\alpha_1 = {}^{A_1}/_{A}$ percentage blue

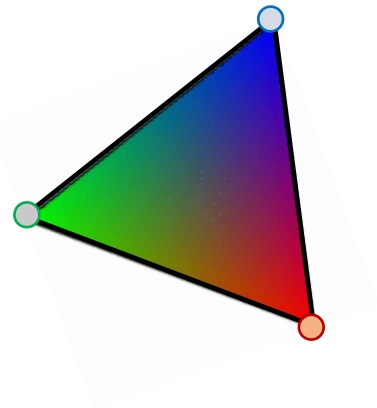
$\alpha_2 = {}^{A_2}/_{A}$ percentage red

$\alpha_3 = {}^{A_3}/_{A}$ percentage green

A … area of whole triangle

$$\boldsymbol{x} = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3$$

$$\alpha_1 + \alpha_2 + \alpha_3 = 1$$



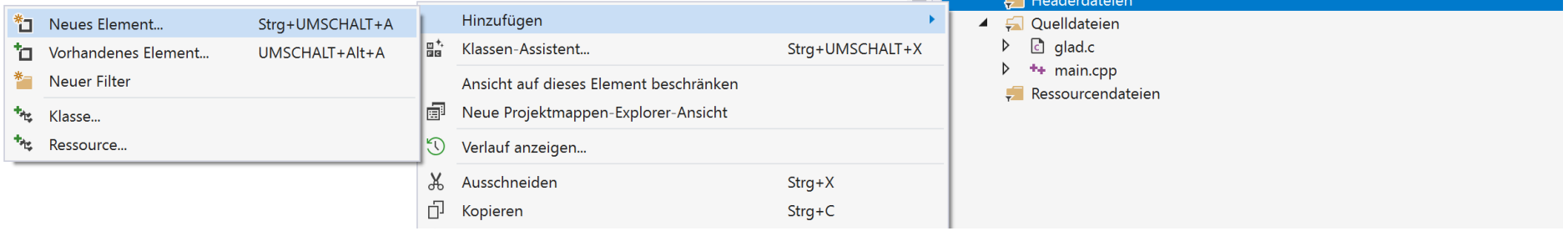
Own Shader Class

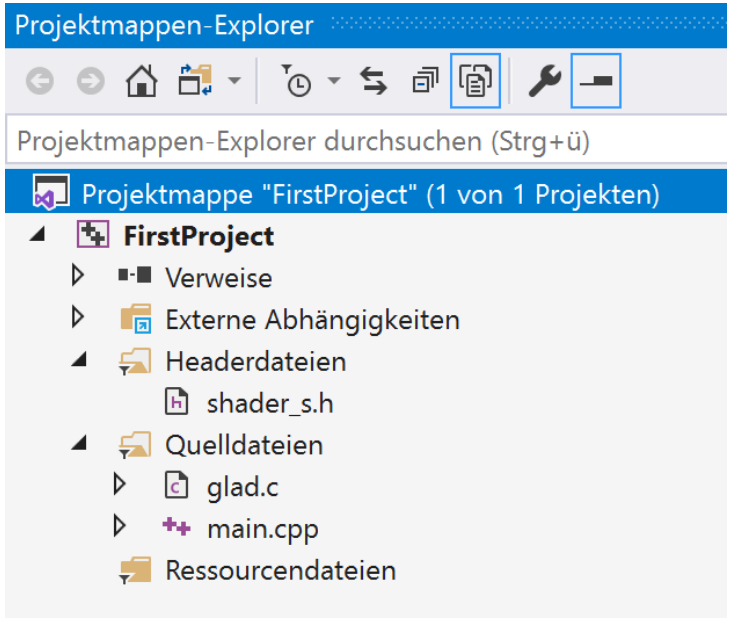
Own Shader Class

- Writing, compiling and managing shaders can be quite cumbersome
- Thus, building a shader class that reads shaders from disk, compiles and links them, checks for errors and is easy to use
- This also gives an idea how we can encapsulate some of the knowledge we learned so far into useful abstract objects

Own Shader Class

- Create a new header: shader_s.h





Own Shader Class

```
#ifndef SHADER_H
#define SHADER_H
#include <glad/glad.h>; // include glad to get all the required OpenGL headers
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
class Shader
{
public:
// the program ID
unsigned int ID;
// constructor reads and builds the shader
Shader(const GLchar* vertexPath, const GLchar* fragmentPath);
// use/activate the shader
void use();
// utility uniform functions
void setBool(const std::string& name, bool value) const;
void setInt(const std::string& name, int value) const;
void setFloat(const std::string& name, float value) const;
};
#endif
```

Own Shader Class

Several preprocessor directives at the top of the header file:

Using these lines of code informs compiler to only include and compile this header file if it hasn't been included yet, even if multiple files include the shader header. This prevents linking conflicts.

```
#ifndef SHADER_H
#define SHADER_H
…
#endif
```

Own Shader Class

- The shader class holds the ID of the shader program
- Its constructor requires the file paths of the source code of the vertex and fragment shader (store on disk as simple text files)
- To add a little extra: add several utility functions:
  - Activates the shader program
  - All set... functions query a uniform location and set its value

Reading from file

- C++ filestreams to read the content from the file into several string objects:

```
Shader(const char* vertexPath, const char* fragmentPath)
{
// 1. retrieve the vertex/fragment source code from filePath
std::string vertexCode;
std::string fragmentCode;
std::ifstream vShaderFile;
std::ifstream fShaderFile;
// ensure ifstream objects can throw exceptions:
vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
…
```

Reading from file

- C++ filestreams to read the content from the file into several string objects:

```
try
{
// open files
vShaderFile.open(vertexPath);
fShaderFile.open(fragmentPath);
std::stringstream vShaderStream, fShaderStream;
// read file’s buffer contents into streams
vShaderStream << vShaderFile.rdbuf();
fShaderStream << fShaderFile.rdbuf();
// close file handlers
vShaderFile.close();
fShaderFile.close();
// convert stream into string
vertexCode = vShaderStream.str();
fragmentCode = fShaderStream.str();
}
…
```

Reading from file

- C++ filestreams to read the content from the file into several string objects:

```
catch (std::ifstream::failure e)
{
std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
}
const char* vShaderCode = vertexCode.c_str();
const char* fShaderCode = fragmentCode.c_str();
```

Compile Shaders

- Next, compile and link shaders

```
// 2. compile shaders
unsigned int vertex, fragment;
int success;
char infoLog[512];
// vertex Shader
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
// print compile errors if any
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
if (!success)
{
glGetShaderInfoLog(vertex, 512, NULL, infoLog);
std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::
endl;
};
```

Compile Shaders

- Next, compile and link shaders

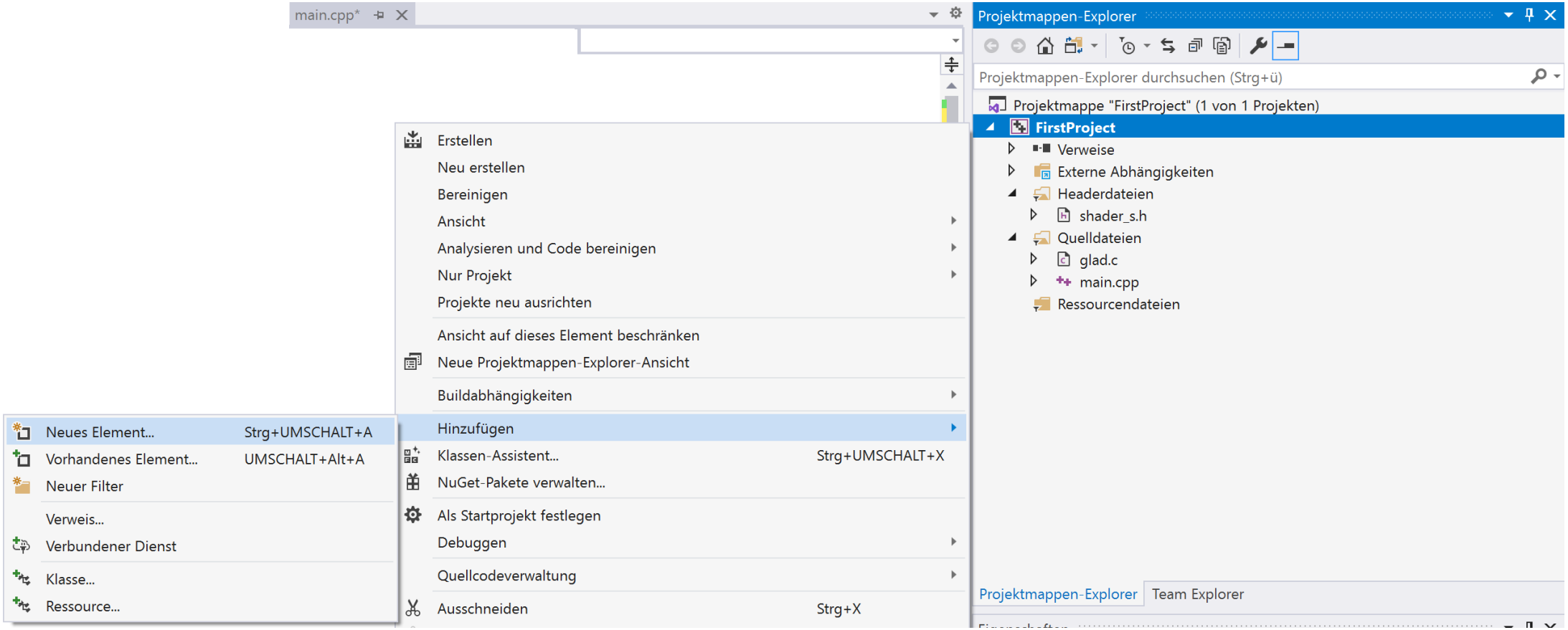
```
// similiar for Fragment Shader
[...]
// shader Program
this->Program = glCreateProgram();
glAttachShader(this->Program, vertex);
glAttachShader(this->Program, fragment);
glLinkProgram(this->Program);
// print linking errors if any
// print linking errors if any
glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
if (!success)
{
  glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
  std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}
// delete the shaders as they’re linked into our program now and no longer necessery
glDeleteShader(vertex);
glDeleteShader(fragment);
```

Use and Set

- Use and set functions:

```
void use()
{
    glUseProgram(ID);
}
// utility uniform functions
// ------------------------------------------------------------------------
void setBool(const std::string& name, bool value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
}
// ------------------------------------------------------------------------
void setInt(const std::string& name, int value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}
// ------------------------------------------------------------------------
void setFloat(const std::string& name, float value) const
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}
```

Add Shader to Project



Add Shader to Project

- Name them vertex_shader.vs and fragment_shader.fs

Ready to use

- Using the shader class is fairly easy; we create a shader object once and from that point on simply start using it:

```
#include "shader_s.h"
…
Shader ourShader(“vertex_shader.vs", “fragment_shader.fs"); // your own shader files
…
while (...)
{
    ourShader.use();
    ourShader.setFloat("someUniform", 1.0f);
    DrawStuff();
}
```

Fun with Shaders*

Before we start…

- Whenever we change the shader, we have to recompile the project
- This might be cumbersome especially, when we want make small changes in, e.g., color

Before we start…

- Thus, we want to change the shader class a bit

```
class Shader
{
public:
    unsigned int ID;
    // constructor generates the shader on the fly
    // ------------------------------------------------------------------------
    Shader(const char* vertexPath, const char* fragmentPath)
    {
```

```
class Shader
{
public:
    unsigned int ID;
    // constructor generates the shader on the fly
    // ------------------------------------------------------------------------
    init(const char* vertexPath, const char* fragmentPath)
    {
```

Before we start…

- Add a refresh function and set the path of the sources

```
public:
    unsigned int ID;
    void refresh()
    {
        glDeleteProgram(ID);
        init(vertexPathShader, fragmentPathShader);
    }
    // constructor generates the shader on the fly
    // ------------------------------------------------------------------------
    void init(const char* vertexPath, const char* fragmentPath)
    {
        setVertexPath(vertexPath);
        setFragmentPath(fragmentPath);
    …
    }
```

Before we start…

- Define the set functions and use the source path in private

```
void setVertexPath(const char* vertexPath)
    {
        int len = strlen(vertexPath)+1;
        vertexPathShader = new char[len];
        strcpy_s(vertexPathShader, len,vertexPath);
    }
    void setFragmentPath(const char* fragmentPath)
    {
        int len = strlen(fragmentPath)+1;
        fragmentPathShader = new char[len];
        strcpy_s(fragmentPathShader, len, fragmentPath);
    }
private:
    char* vertexPathShader;
    char* fragmentPathShader;
```

Before we start…

- Set ourShader as a global variable (outside and before main())
- Whenever we press ‘1’ refresh the shader

```
Shader ourShader;

int main()
{
…
 ourShader.init("vertex_shader.vs", "fragment_shader.fs");
…
```

```
void processInput(GLFWwindow* window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS)
        ourShader.refresh();
}
```

Done!

- If we run the program and we change the shader, we only have to press ‚1‘ to refresh the shader
- No recompile

Basics

- Let‘s do something with the shaders

Basics

- For this, we create again a rectangle as we did in the last lecture
- This time it will be on the whole screen:

```
float vertices[] = {
         1.0f,  1.0f, 0.0f,  // top right
         1.0f, -1.0f, 0.0f,  // bottom right
        -1.0f, -1.0f, 0.0f,  // bottom left
        -1.0f,  1.0f, 0.0f   // top left
    };
    unsigned int indices[] = {  // note that we start from 0!
        0, 1, 3,  // first Triangle
        1, 2, 3   // second Triangle
    };
```

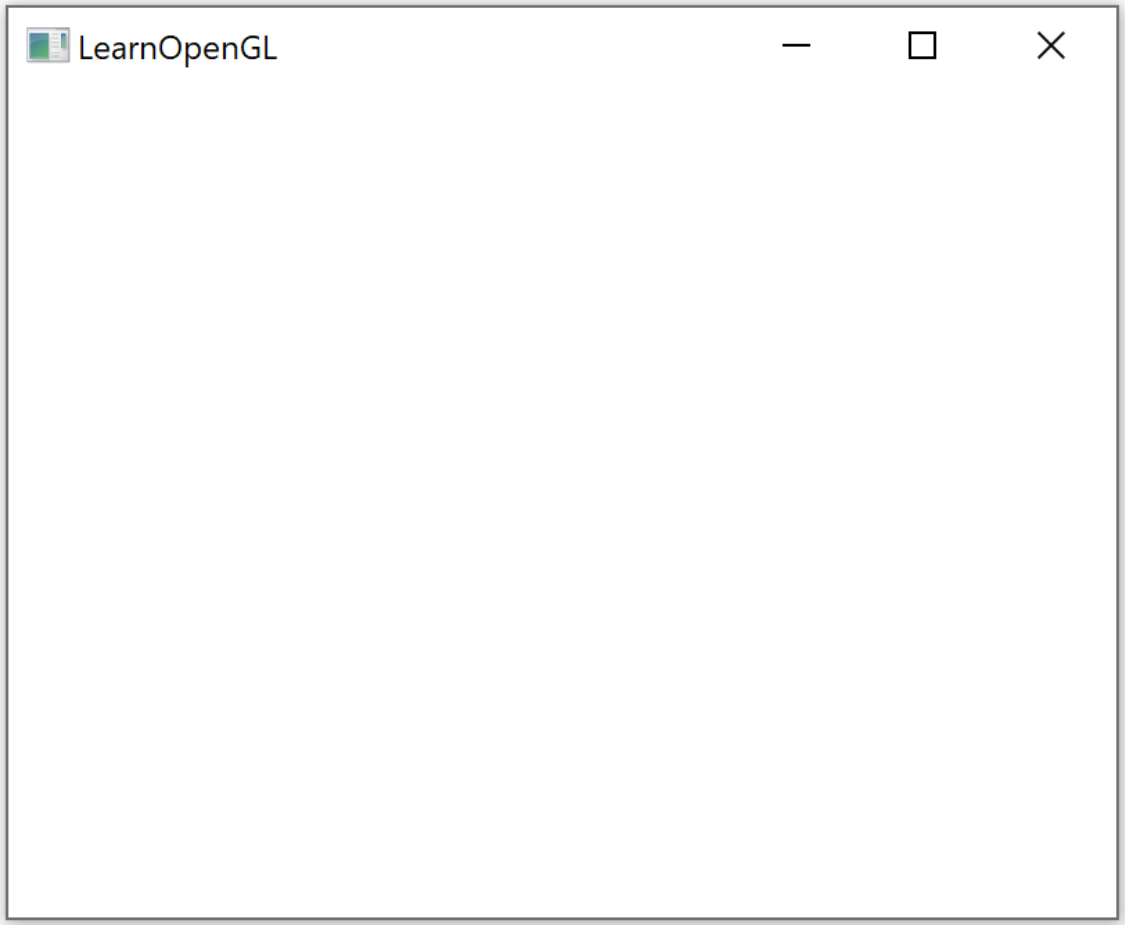
Basics

- Then, we create a simply fragment shader

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

F5…

- …boring

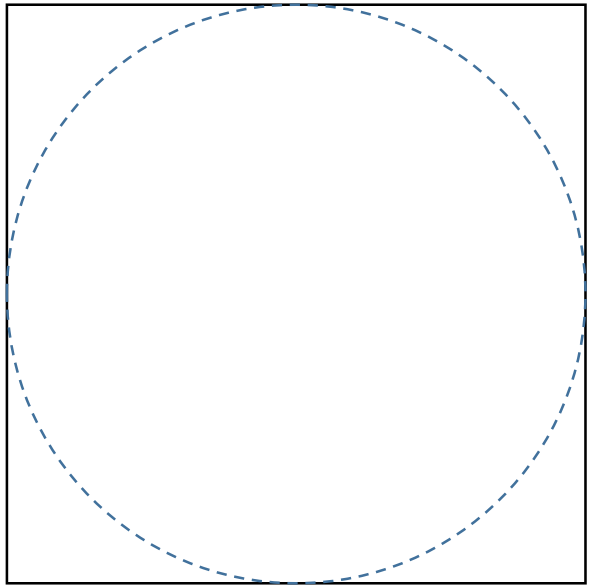


Concept

- Beginner tends to program in the shader as they would program in C:
  - Explicitly drawing every element
- Fragment Shaders are called for every fragment
- Drawing the full scene each time is very costly
- Only draw the current fragment

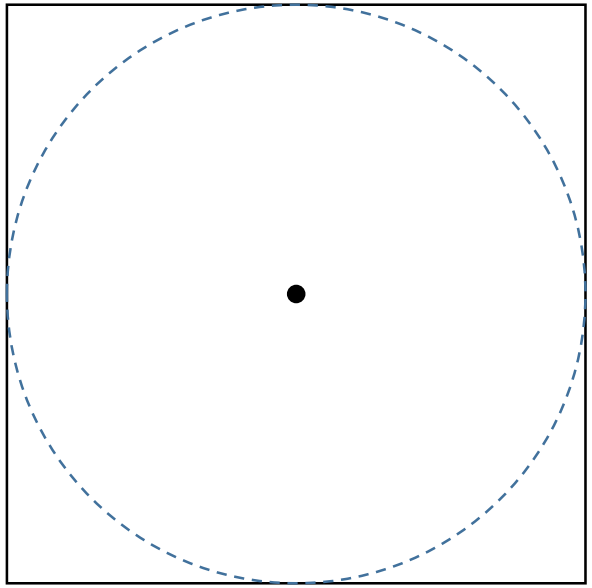
Circle

- For the beginning, we want to draw a single circle
- Assume, we have a window of size 100x100
- The circle to be drawn should have a radius of 50
- How can we decide for every pixel if it should have a color (inside the circle) or not (outside)



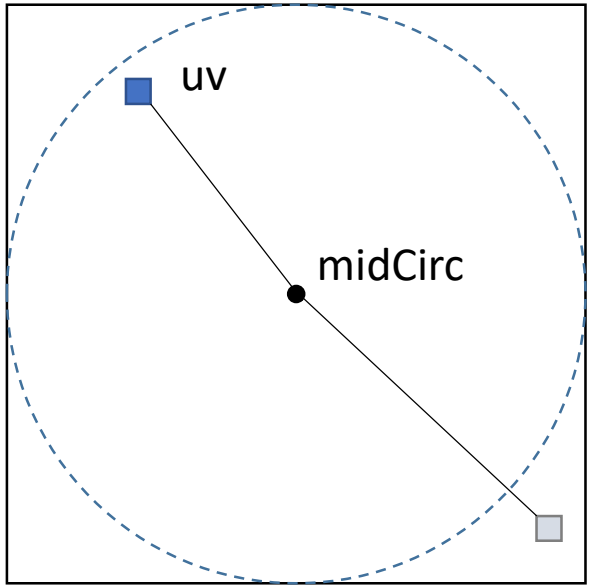
Circle

- Easiest way is to check for the distance from the midpoint



Circle

- Easiest way is to check for the distance from the midpoint
- If the distance is smaller than the radius, draw the fragment otherwise discard it:
  - If `length(uv-midCirc)<=radius` → `draw`
  - `Otherwise` → `discard`



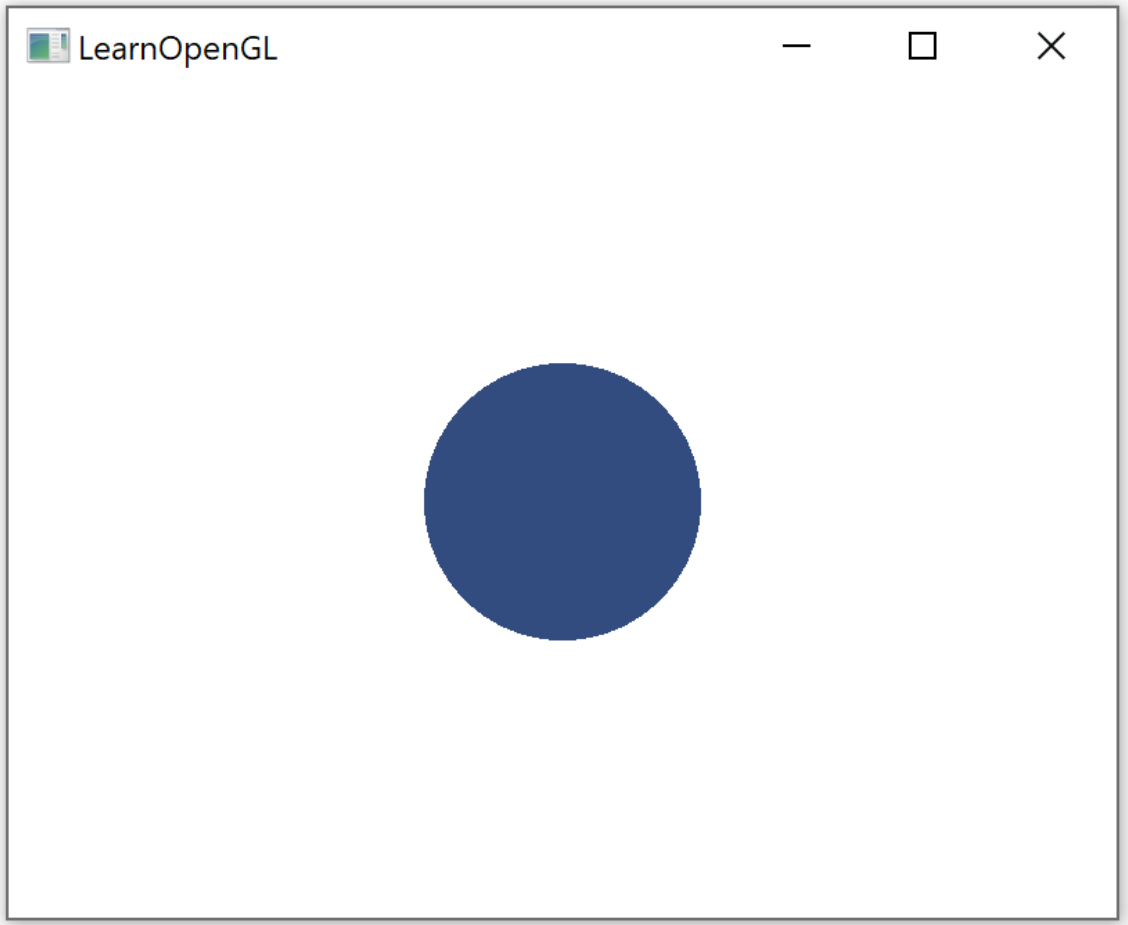
Circle

- A single circle:

```
#version 330 core
out vec4 FragColor;
void main()
{
    vec4 color=vec4(1);
    vec2 uv = gl_FragCoord.xy;
    vec2 midCirc=vec2(400,300);
            if(length(uv-midCirc)<=100)
            {
                color.rgb=vec3(0.2,0.3,0.5);
             }
    FragColor = color;
}
```

F5…

- … an expected blue circle



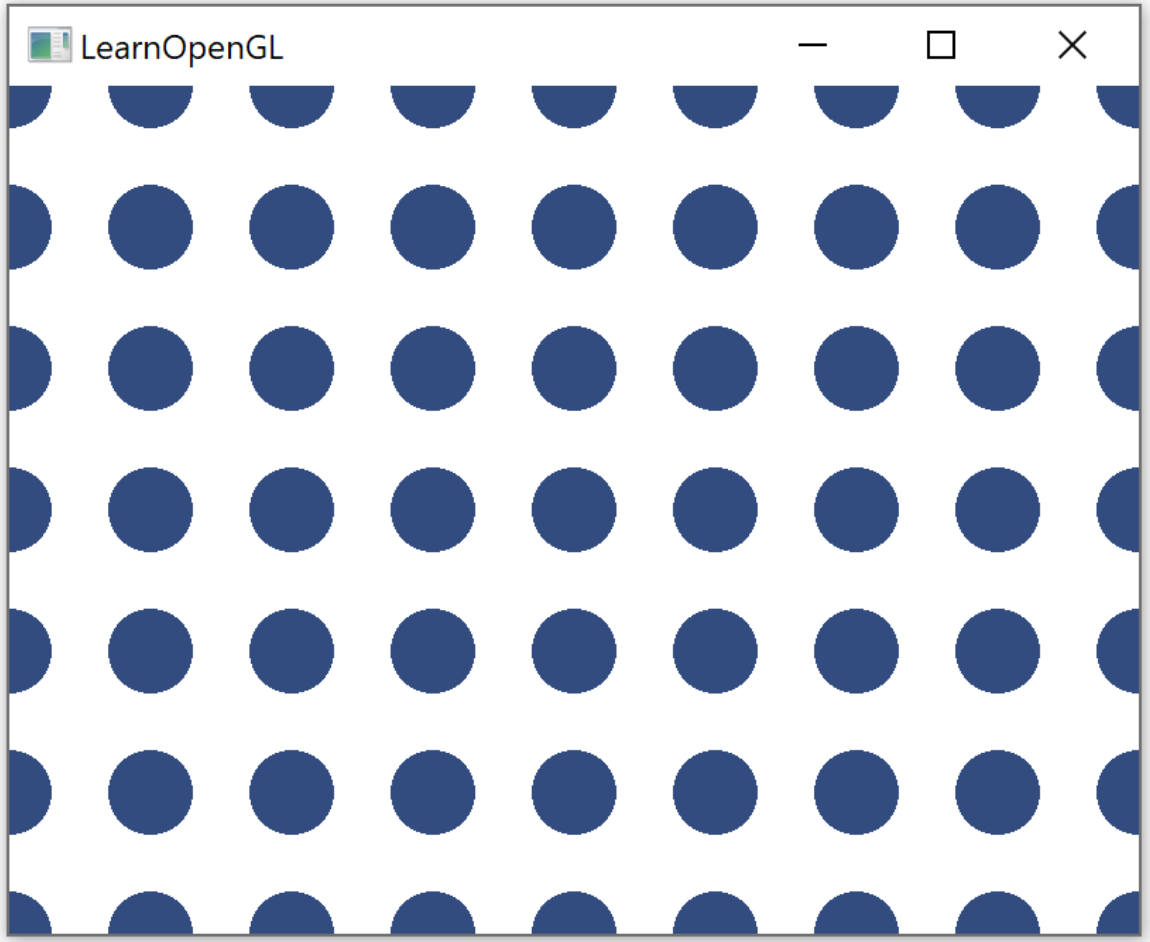
Circles

- Do not:

```
#version 330 core
out vec4 FragColor;
void main()
{
    vec4 color=vec4(1);
    vec2 uv = gl_FragCoord.xy;
    for (int i=0;i<900;i+=100)
    {
        for (int j=0;j<700;j+=100)
        {
            vec2 midCirc=vec2(i,j);
            if(length(uv-midCirc)<=30)
            {
                color.rgb=vec3(0.2,0.3,0.5);
            }
        }
    }
    FragColor = color;
}
```

F5…

- …blue circles



Circles

- Every fragment iterates over all objects (circles), which is not efficient!

```
for (int i=0;i<900;i+=100)
    {
        for (int j=0;j<700;j+=100)
        {
            vec2 midCirc=vec2(i,j);
            if(length(uv-midCirc)<=30)
            {
                color.rgb=vec3(0.2,0.3,0.5);
            }
        }
    }
FragColor = color;
}
```

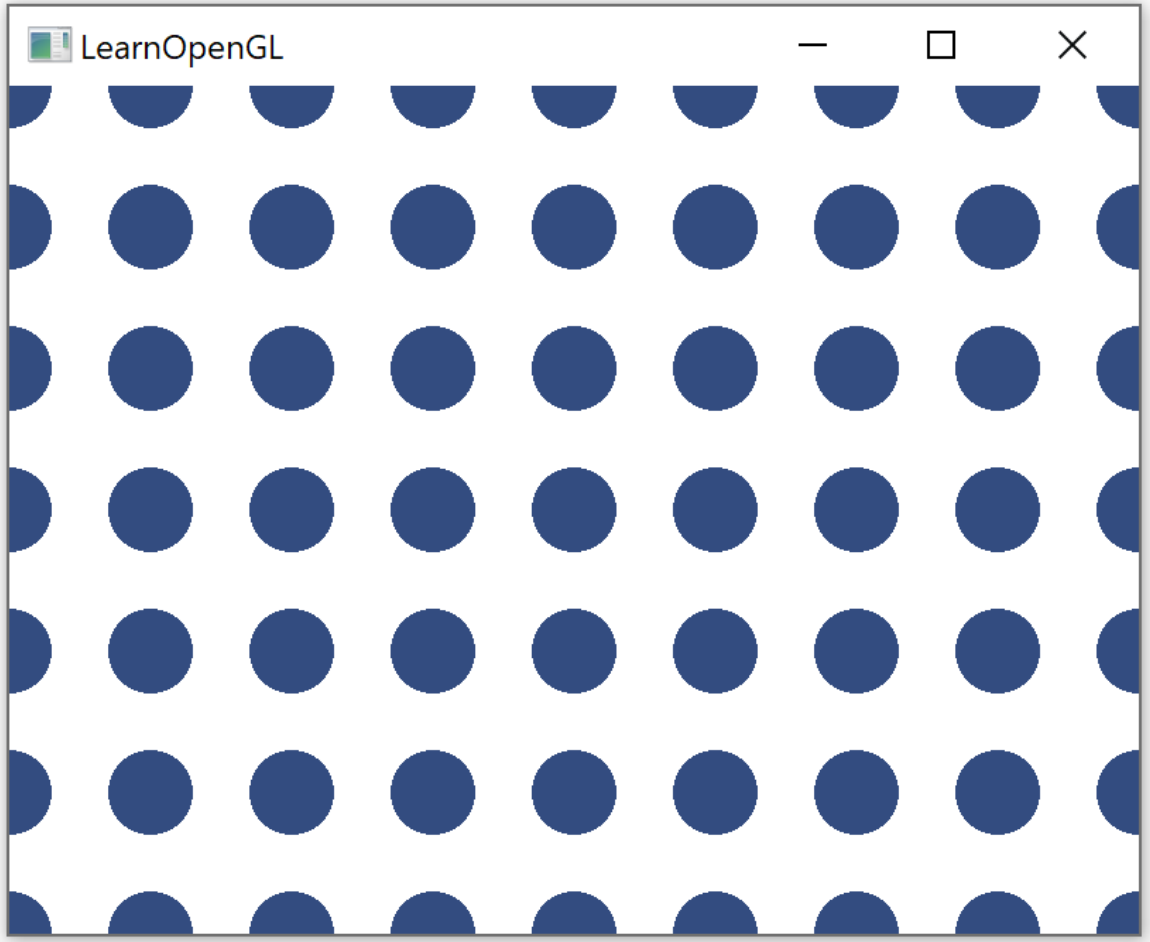
Circles

- Better:

```
#version 330 core
out vec4 FragColor;
void main()
{
    vec4 color=vec4(1);
    vec2 uv = gl_FragCoord.xy;

    vec2 uv_mod = mod(uv-50, 100);
    if ( length(uv_mod-50) < 30 )
        color.rgb=vec3(0.2,0.3,0.5);

    FragColor = color;
}
```

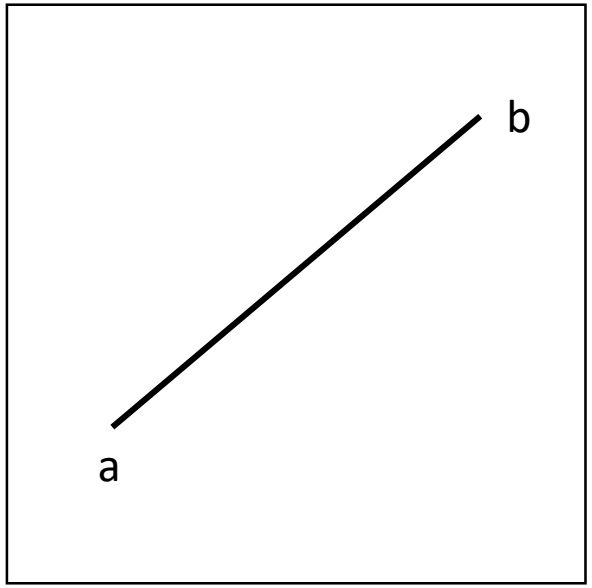
F5…

- …efficient blue circles



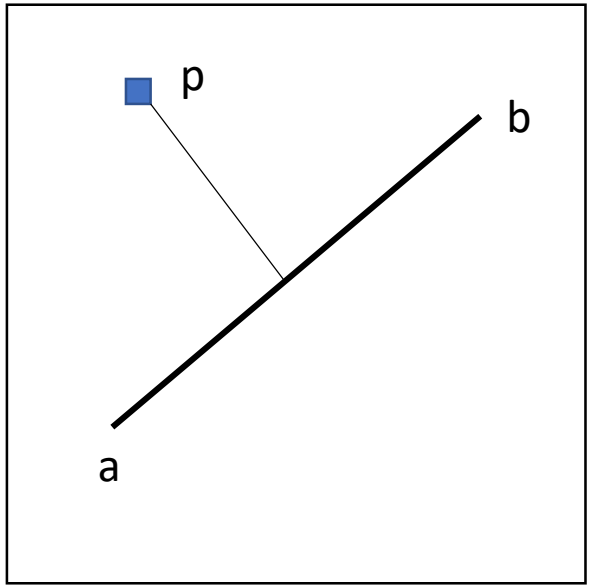
Line

- Now, we want to draw a single line connecting two points a,b
- How can we decide for every pixel if it lies on the line or not



Line

- Similar to the circle example, we want to calculate the distance of a fragment to the line
- If the (shortest) distance is below a threshold, the fragment gets a color otherwise it is discarded



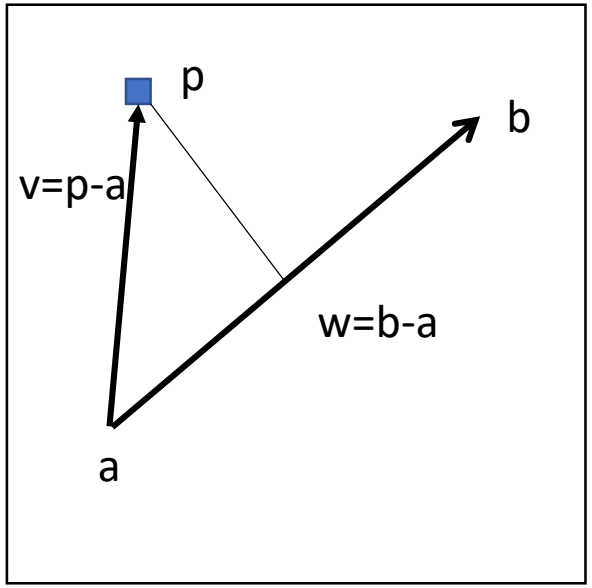
Line

- Every point x on the line can be expressed by:

$$x = a + \lambda w$$

- We have to find the orthogonal projection of p on the line
- Therefore:

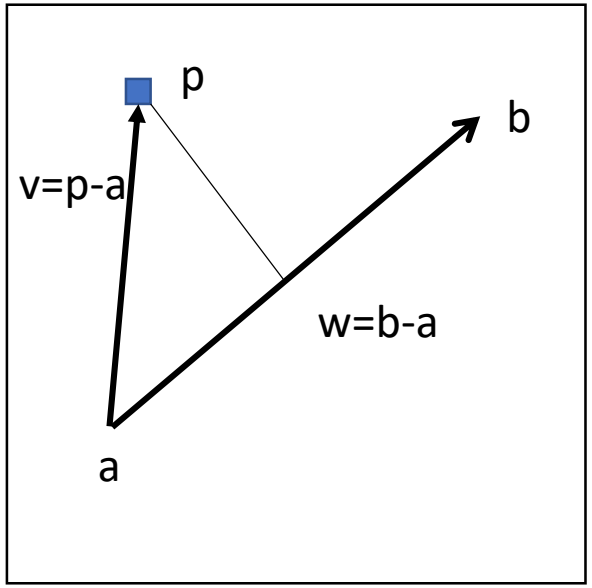
$$\langle p - x, w \rangle = 0$$



Line

- This yields:

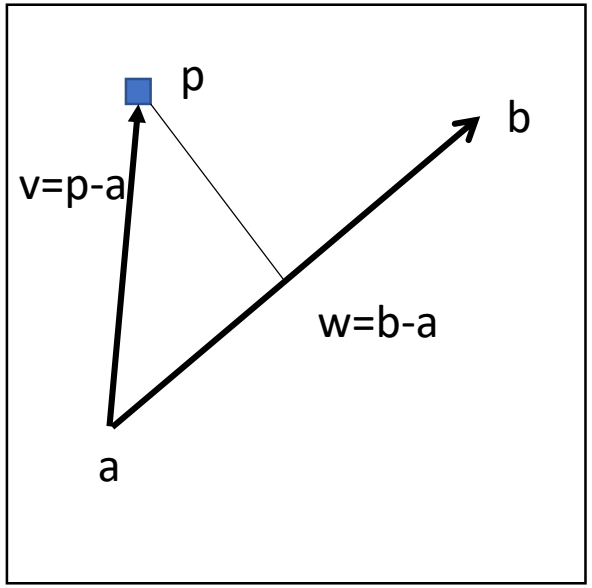
$$
\begin{aligned}
0 &= \langle p - x, w \rangle \\
&= \langle p - (a + \lambda w), w \rangle \\
&= \langle p - a, w \rangle - \langle \lambda w, w \rangle \\
\lambda \langle w, w \rangle &= \langle p - a, w \rangle \\
\lambda &= \frac{\langle v, w \rangle}{\langle w, w \rangle}
\end{aligned}
$$



Line

- We only want to consider points on the line:
  - If $\lambda > 1 \rightarrow \lambda = 1$
  - If $\lambda < 0 \rightarrow \lambda = 0$
- This can be achieved with the clamp(λ,0,1) command
- The distance is then (λ_c = clamp(λ,0,1)):

$$d = \|a + \lambda_c w - p\|$$
$$= \|\lambda_c w - v\|$$



$$\lambda = \frac{\langle v, w \rangle}{\langle w, w \rangle}$$

Line

$$\lambda = \frac{\langle v, w \rangle}{\langle w, w \rangle}$$

$$d = \|\lambda_c w - v\|$$

- Add a function:

```
#version 330 core
out vec4 FragColor;

float line(vec2 p, vec2 a,vec2 b)
{
    vec2 v=p-a;
    vec2 w=b-a;
    float lambda = dot(v, w) / dot(w, w);
    lambda=clamp(lambda,0,1);
    return length(lambda*w-v);
}

void main()
{…}
```

Line

$$\lambda = \frac{\langle v, w \rangle}{\langle w, w \rangle}$$

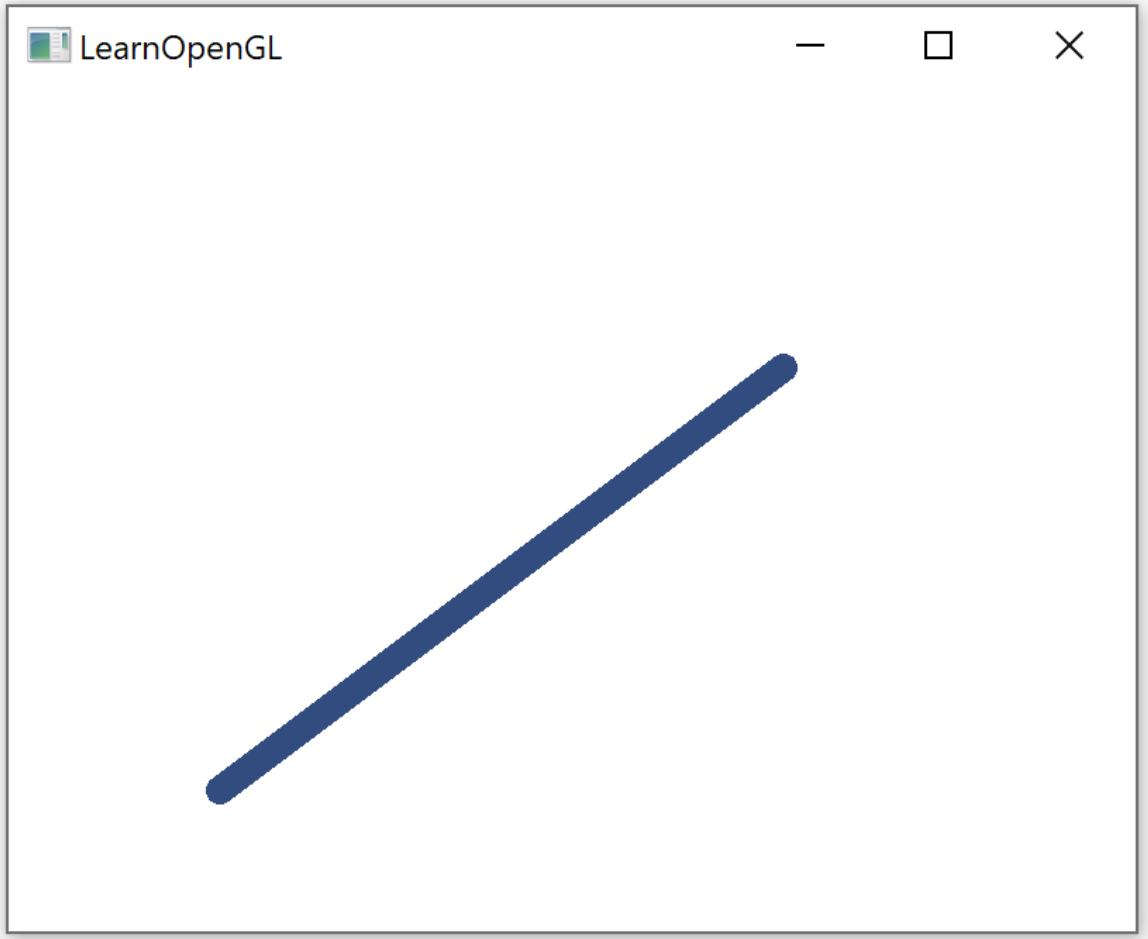
$$d = \|\lambda_c w - v\|$$

- Draw the line

```
void main()
{
    vec4 color=vec4(1);
    vec2 uv = gl_FragCoord.xy;

    if(line(uv,vec2(150,100),vec2(550,400))<10)
        color.rgb=vec3(0.2,0.3,0.5);

    FragColor = color;
}
```

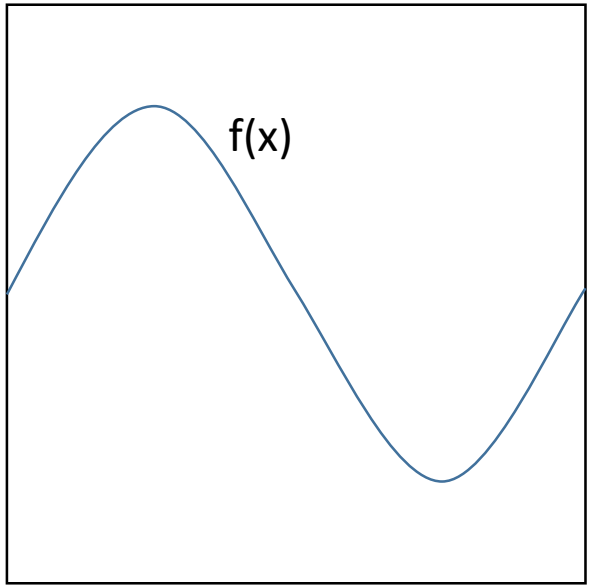
F5…

- … a line



Functions

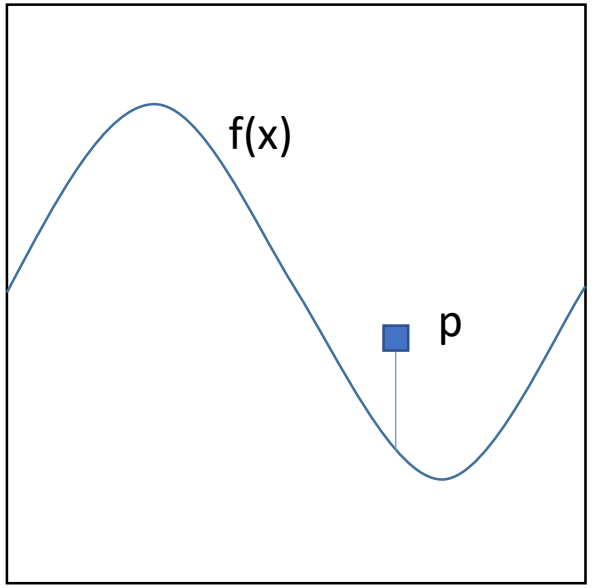
- Now, we want to draw a function
- How can we decide for every pixel if it lies on the function or not



Functions

- A straightforward approach would be to test if:

$$d = |p.y - f(p.x)| < \epsilon$$



Functions

$$d = |p.y - f(p.x)| < \epsilon$$

- Add a function:

```
#version 330 core
out vec4 FragColor;

float f(float x) //just a scaled sin function
{
    return sin(x*3.14159*2)/2.25+0.5;
}
void main()
{…}
```

Functions

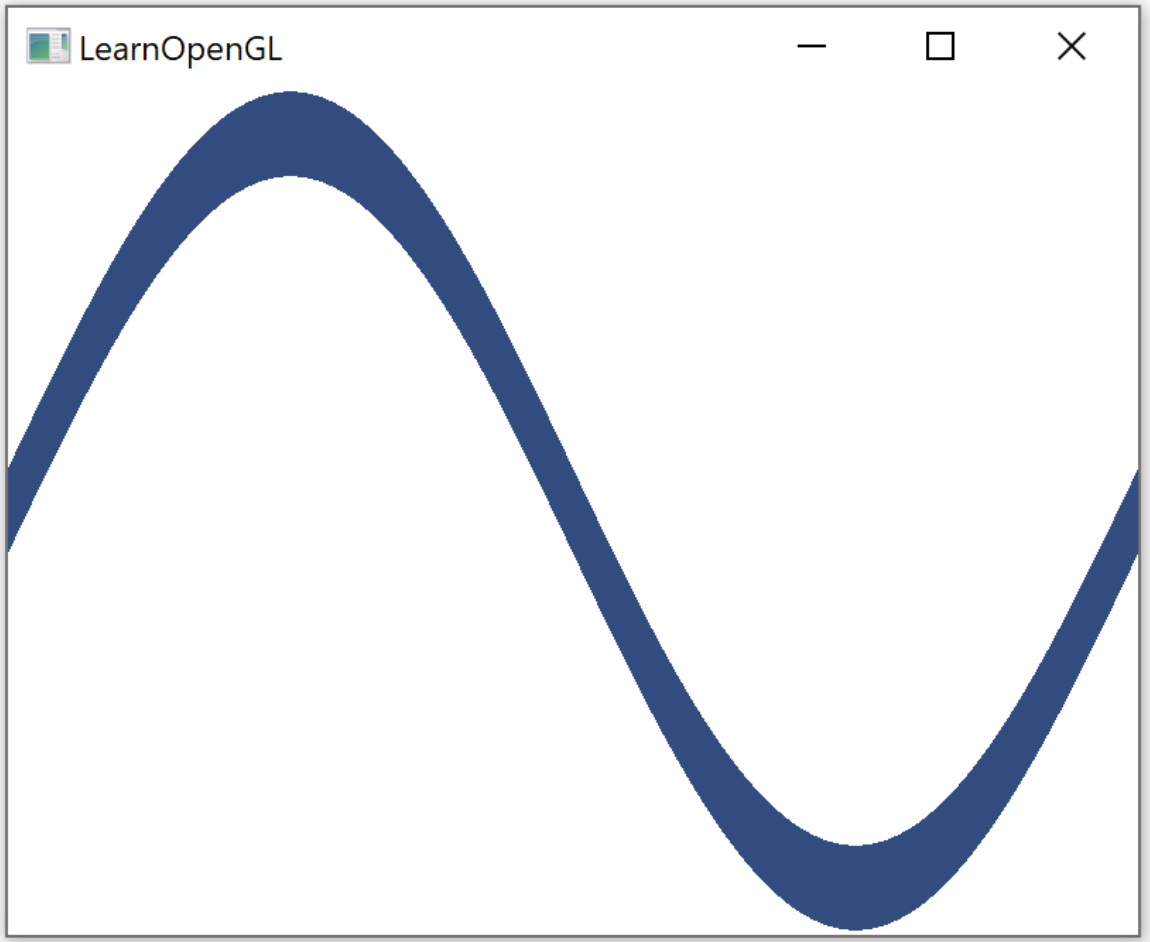
$$d = |p.y - f(p.x)| < \epsilon$$

- Draw the function:

```
void main()
{
    vec4 color=vec4(1);
    vec2 p = gl_FragCoord.xy / vec2(800,600);

    if(abs(p.y-f(p.x))<0.05)
        color.rgb=vec3(0.2,0.3,0.5);

    FragColor = color;
}
```

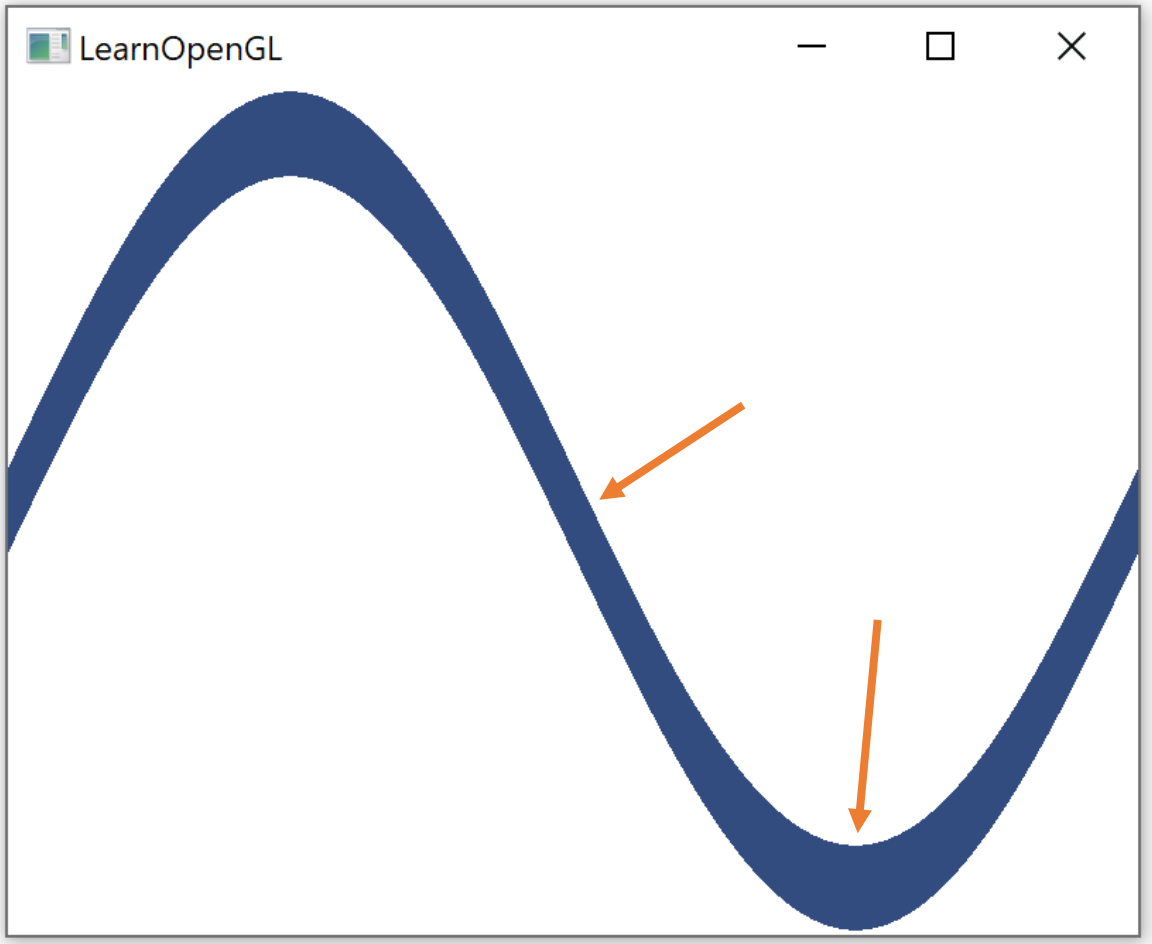
F5…

- … sin



F5…

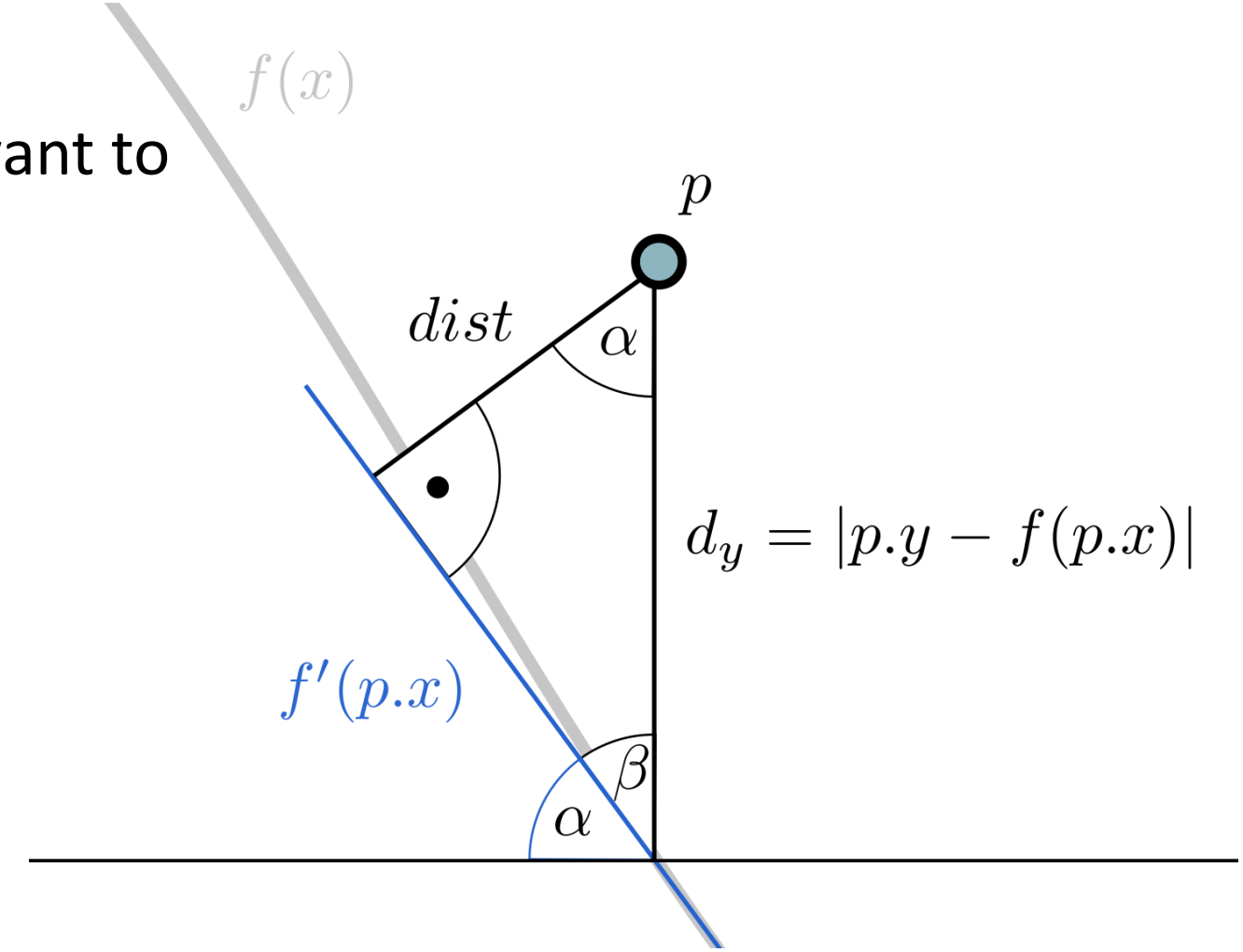
- … sin
- But it has different widths



Correction

- Instead of calculating d_y, we want to determine $dist$:

$$\cos(\alpha) = \frac{dist}{d_y}$$

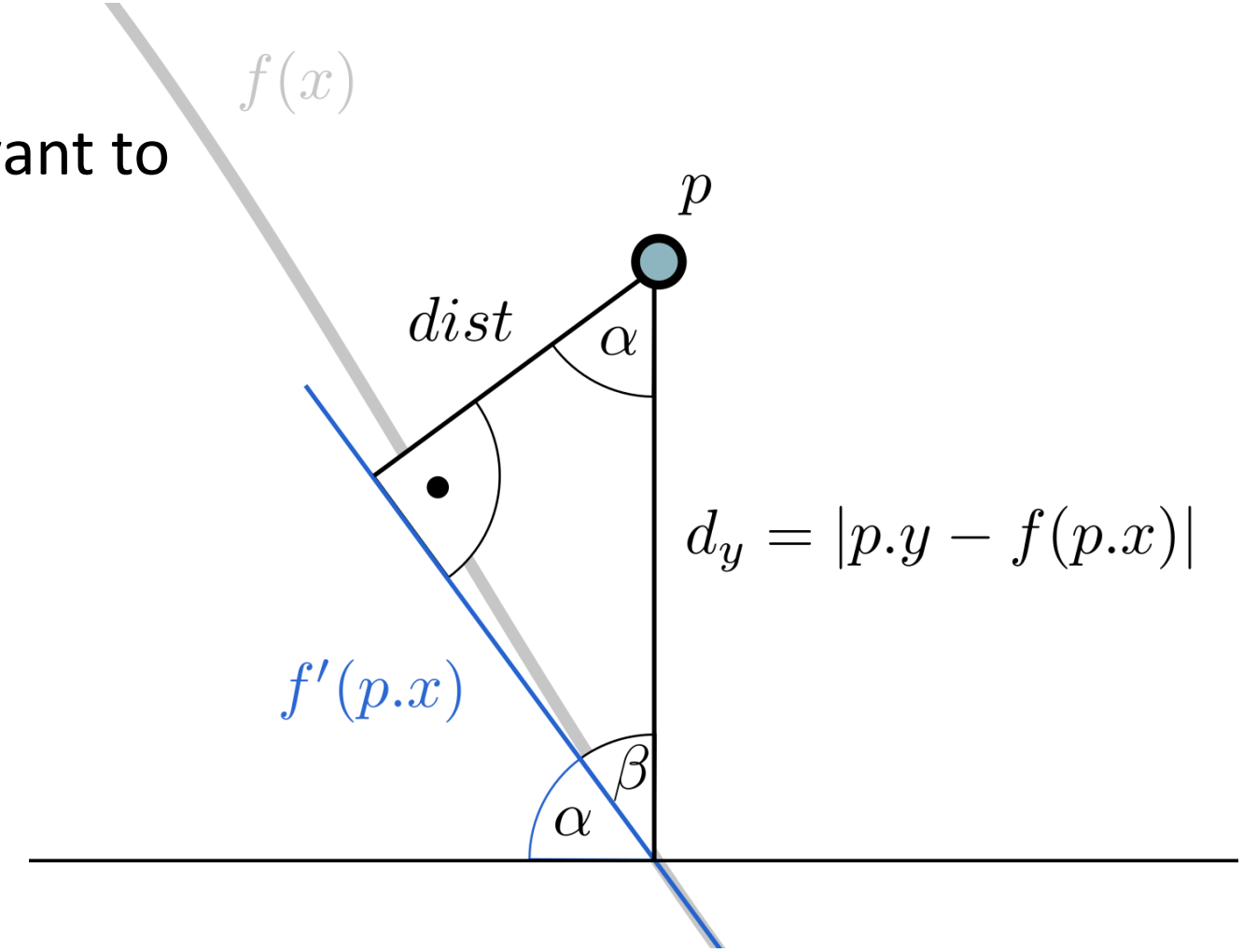


Correction

- Instead of calculating d_y, we want to determine $dist$:

$$\cos(\alpha) = \frac{dist}{d_y}$$

$$dist = d_y \cos(\alpha)$$



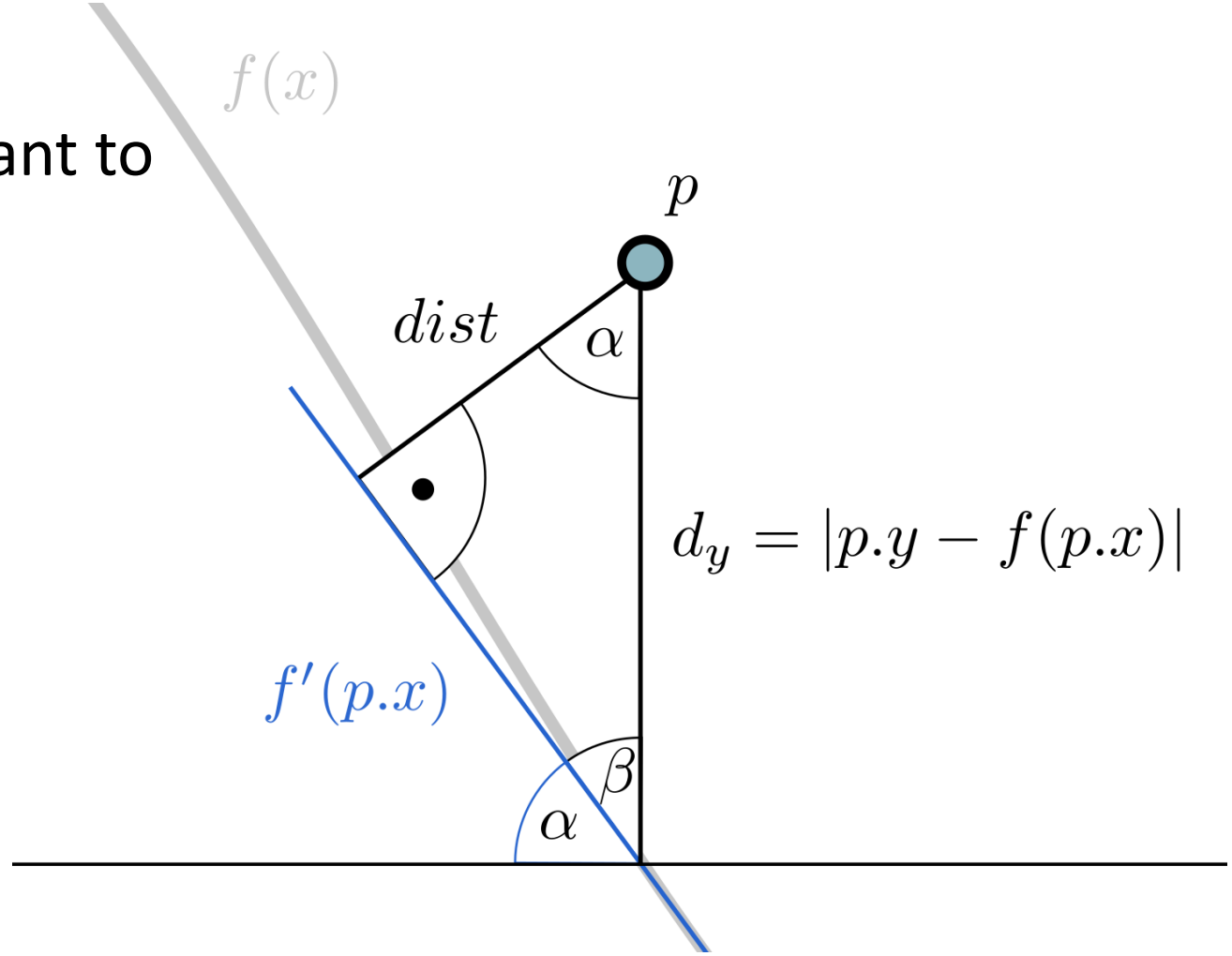
Correction

- Instead of calculating d_y, we want to determine $dist$:

$$\cos(\alpha) = \frac{dist}{d_y}$$

$$dist = d_y \cos(\alpha)$$

$$= d_y \cos(\tan^{-1}(f'(p.x)))$$



Correction

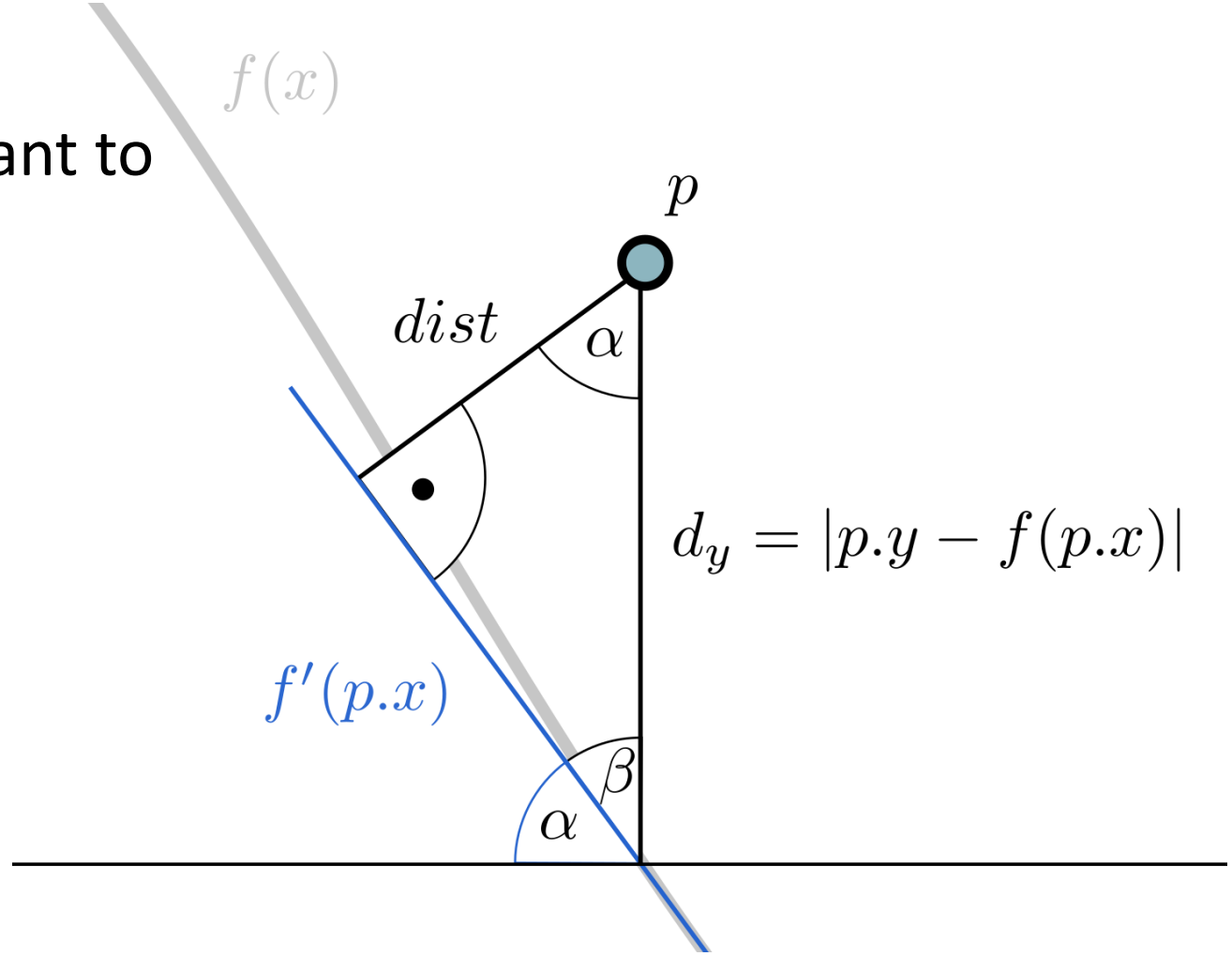
- Instead of calculating d_y, we want to determine $dist$:

$$\cos(\alpha) = \frac{dist}{d_y}$$

$$dist = d_y \cos(\alpha)$$

$$= d_y \cos(\tan^{-1}(f'(p.x)))$$

$$= d_y \frac{1}{\sqrt{1 + f'(p.x)^2}}$$



Correction

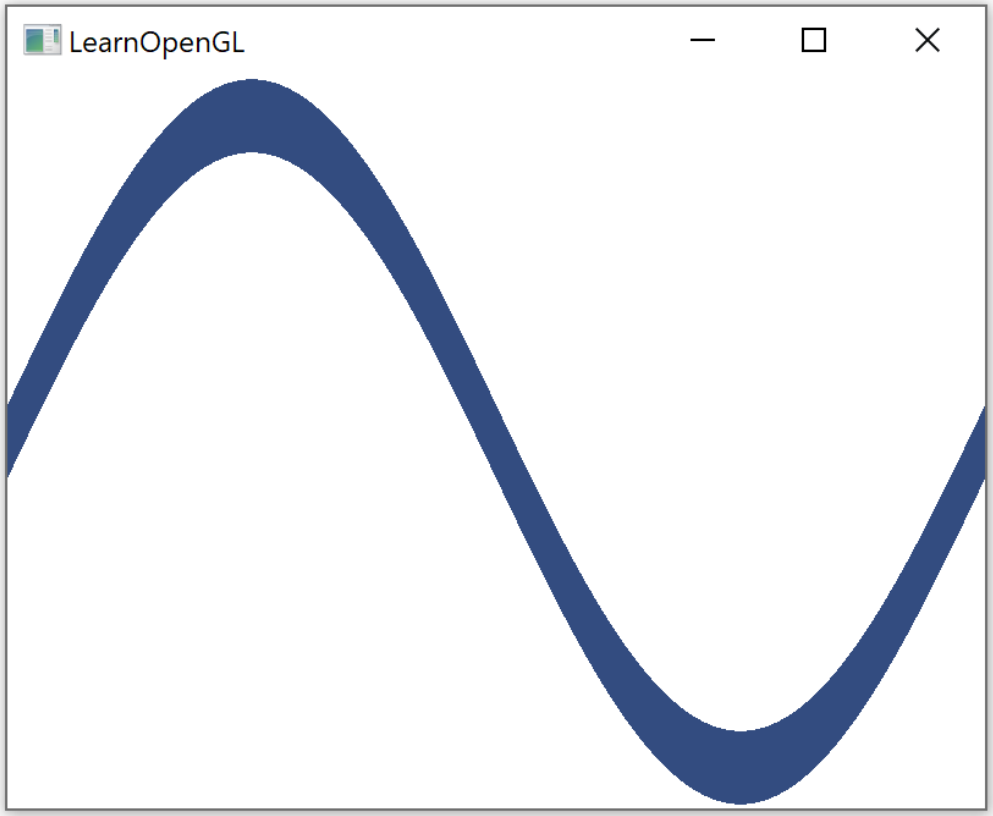
$$dist = d_y \frac{1}{\sqrt{1 + f'(p.x)^2}}$$

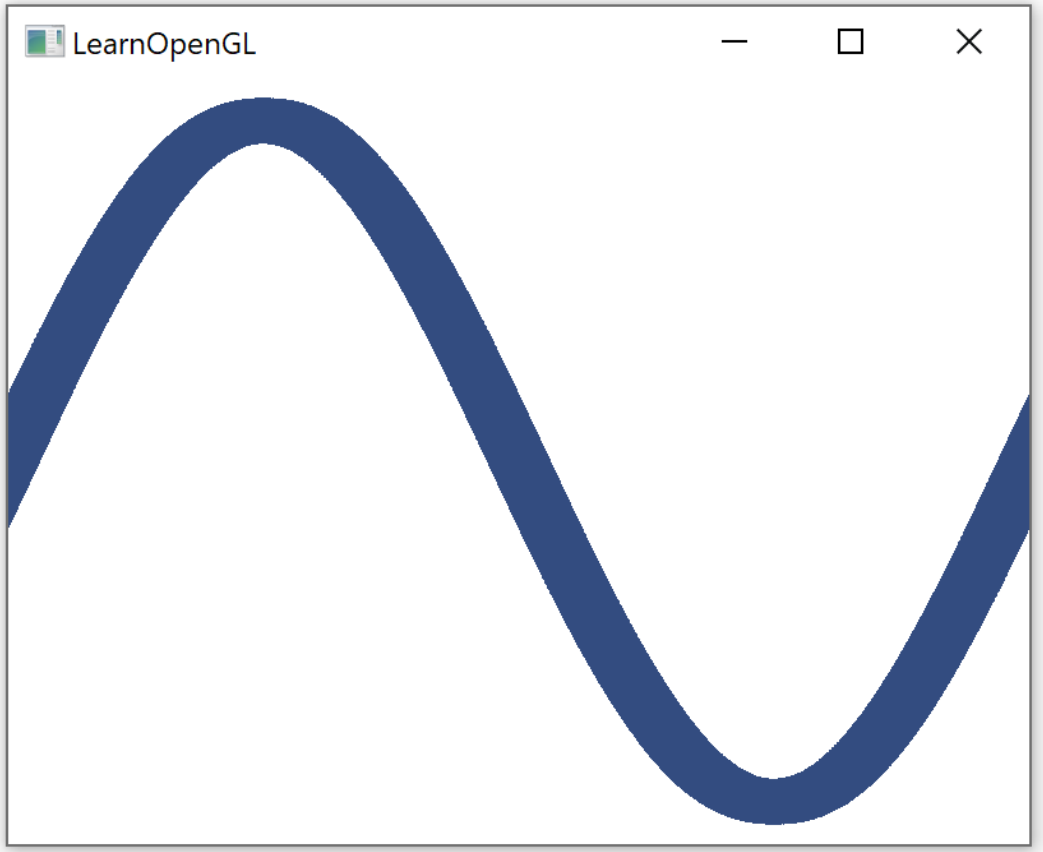
- Draw the function:

```
void main()
{
    vec4 color=vec4(1);
    vec2 p = gl_FragCoord.xy / vec2(800,600);

    float eps=0.0001;
    float df= (f(p.x) - f(p.x + eps)) / eps;
    float dist=abs(p.y-f(p.x))/sqrt(1+df*df);

    if(dist<0.03)
        color.rgb=vec3(0.2,0.3,0.5);

    FragColor = color;
}
```

F5…

- … left old, right new





Aliasing

- What can we do with aliasing effects?

Aliasing

- For this, we use the integrated function smoothstep(a,b,x)
- smoothstep is defined as:

```
float smoothstep(float a, float b, float x)
{
    t = clamp((x - a) / (b - a), 0.0, 1.0);
    return t * t * (3.0 - 2.0 * t);
}
```

- This function interpolates the value x between a,b smoothly (Hermite interpolation)

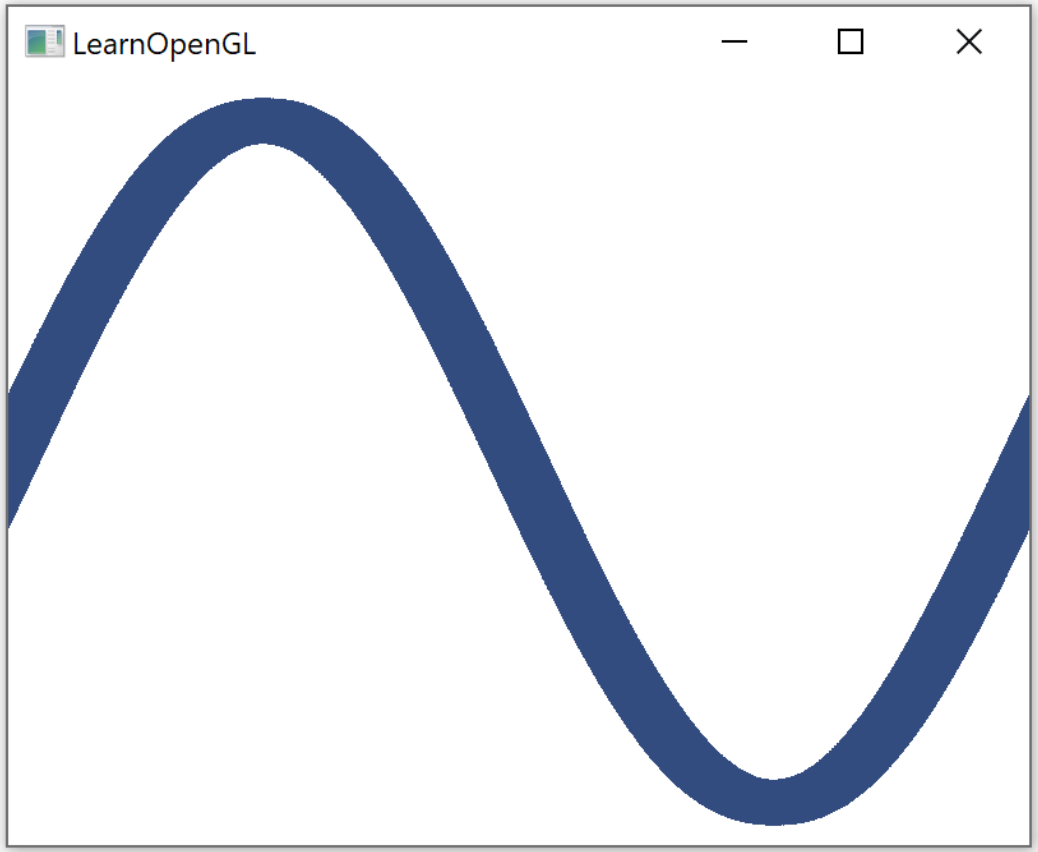
Aliasing

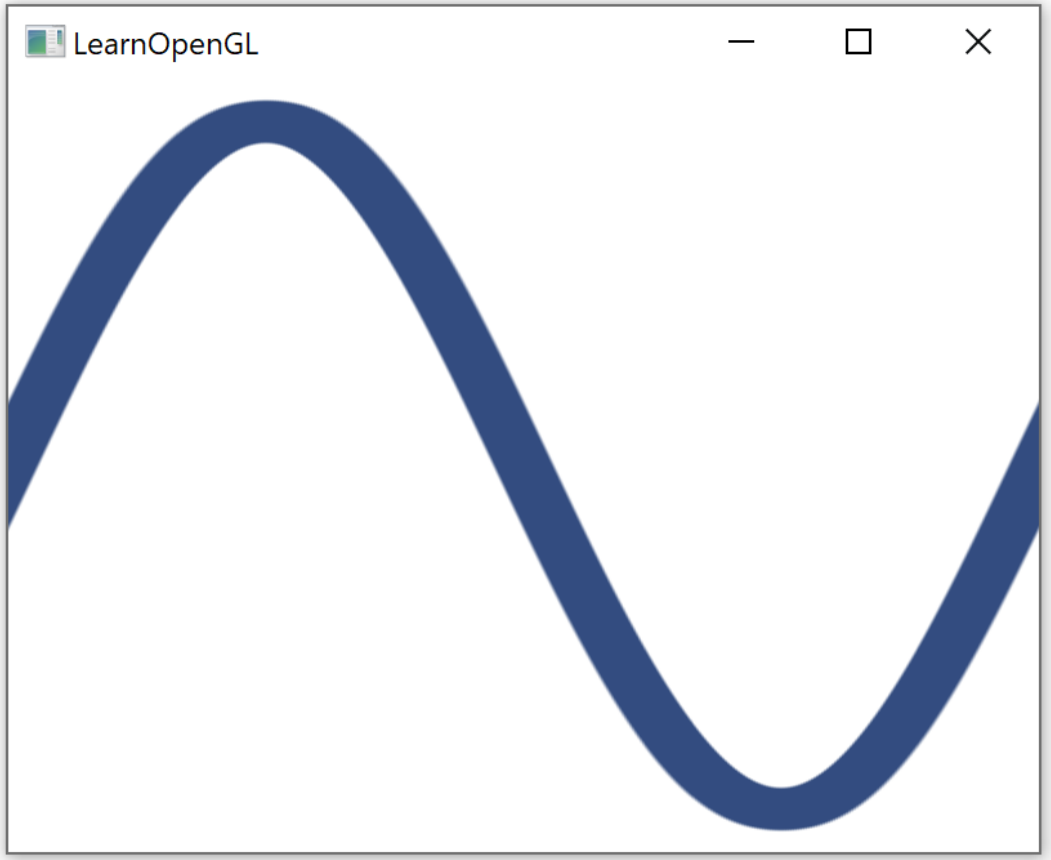
- We can use this function to transits between the white background color and the blue color:

```
float dist=abs(p.y-f(p.x))/sqrt(1+df*df);
vec4 blue=vec4(0.2,0.3,0.5,1.0);

dist= smoothstep(0.025, 0.03, dist); // smooth interpolation between 0.025 and 0.03

color = mix(blue, color, dist); // linear interpolation between color

FragColor = color;
```

F5…

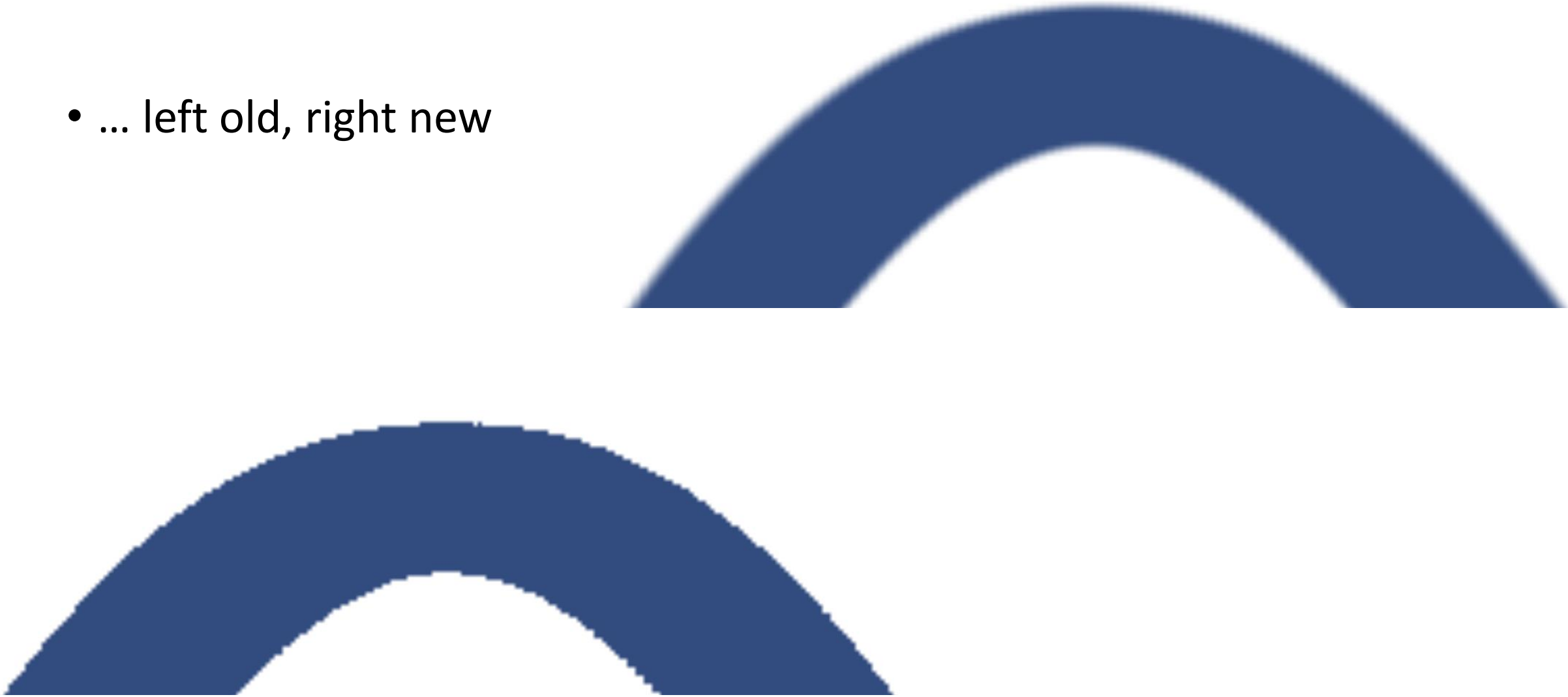
- … left old, right new





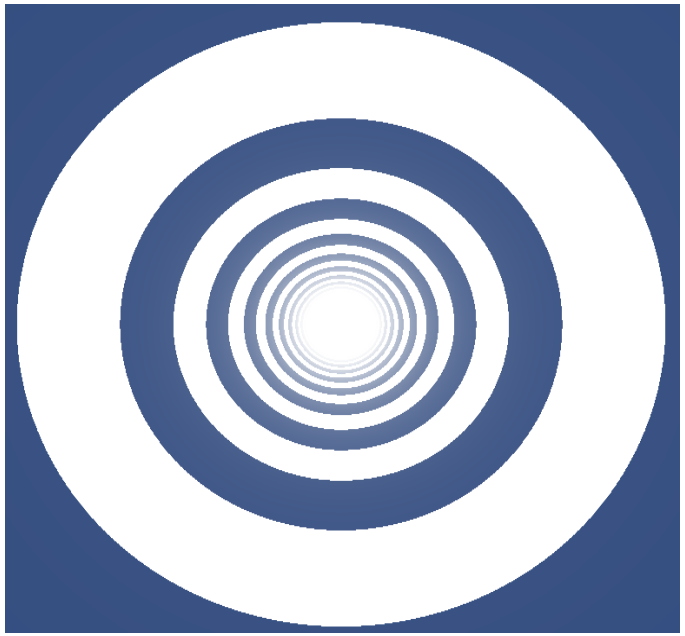
F5…

- … left old, right new



Tunnel

- As the last example, we want to fly through a tunnel



Tunnel

- Finally, we want to animate (use time as a uniform):

```
float time= glfwGetTime();
ourShader.use();
ourShader.setFloat("time", time);
```

```
uniform float time;

…
void main()
{
…
}
```

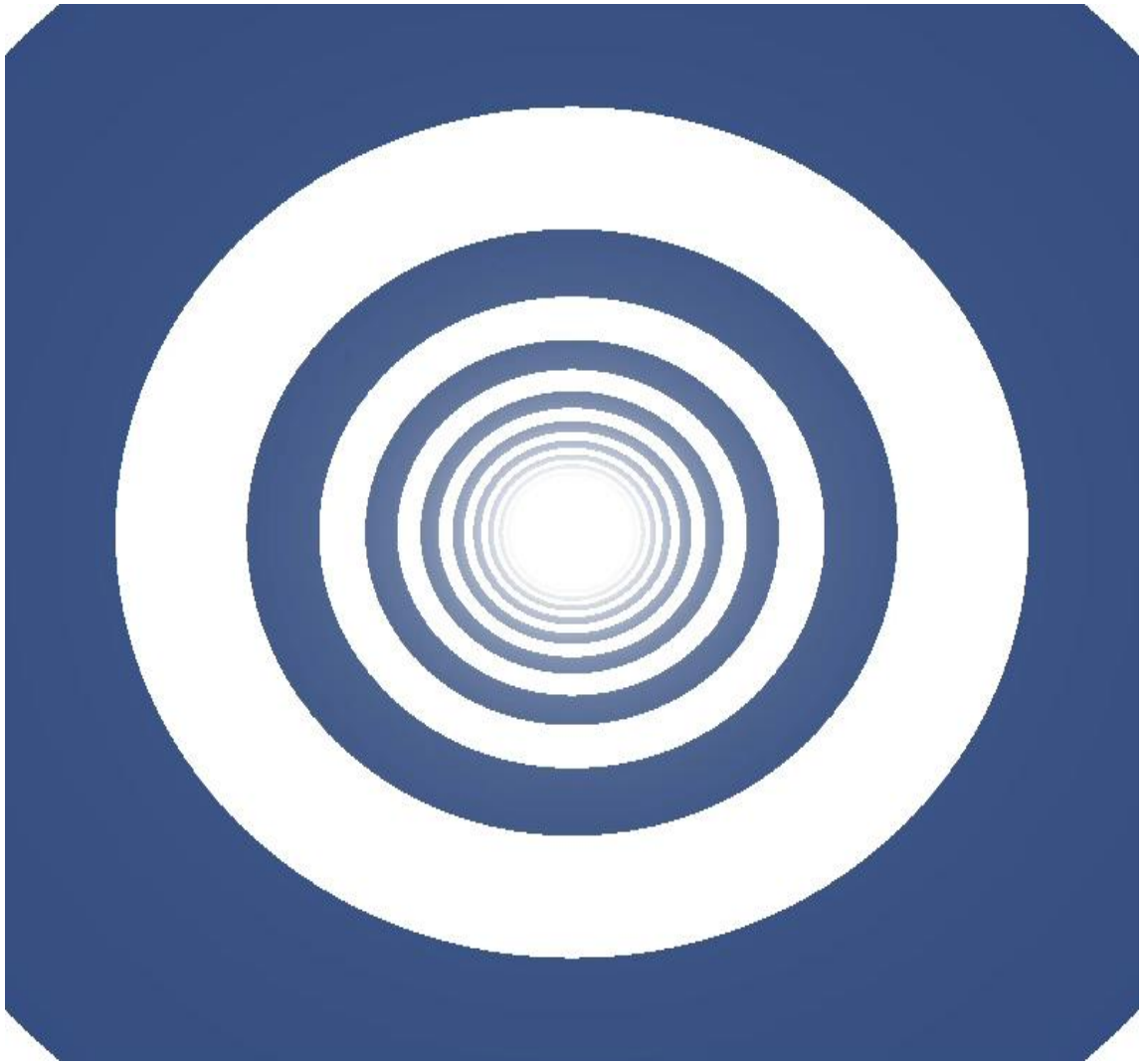
Tunnel

- Draw the tunnel

```
vec4 color=vec4(1);
vec4 blue=vec4(0.2,0.3,0.5,1.0);

vec2 p = gl_FragCoord.xy/ vec2(800,600)-vec2(0.5);

float t=3*time;
float lambda=1/length(p);

if(fract((lambda+t)/2)<0.5) // compute the fractional part: 13.6 → 0.6
    color = mix(blue,color, smoothstep(0, 20, lambda)); // part inside gets foggy

FragColor = color;
```

F5…

- … wow



Examples

- Visit https://www.shadertoy.com/ for more crazy examples

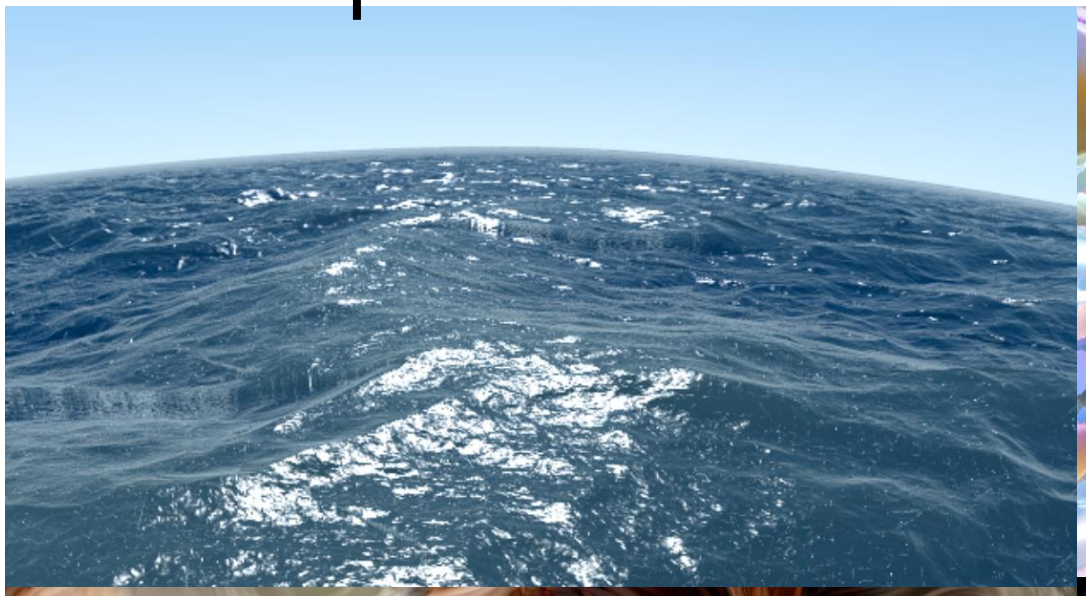
Image

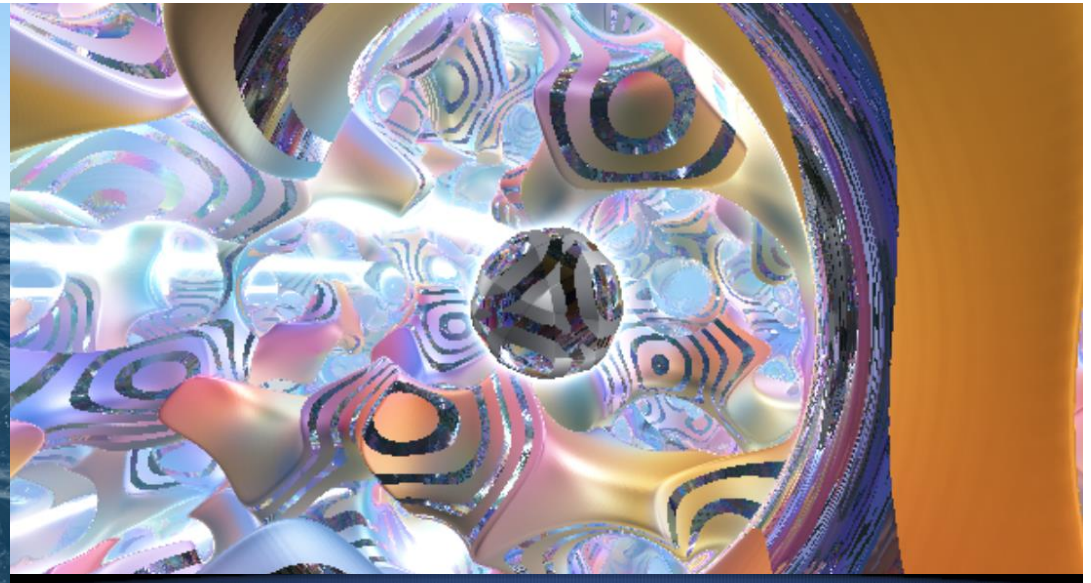
Shader Inputs

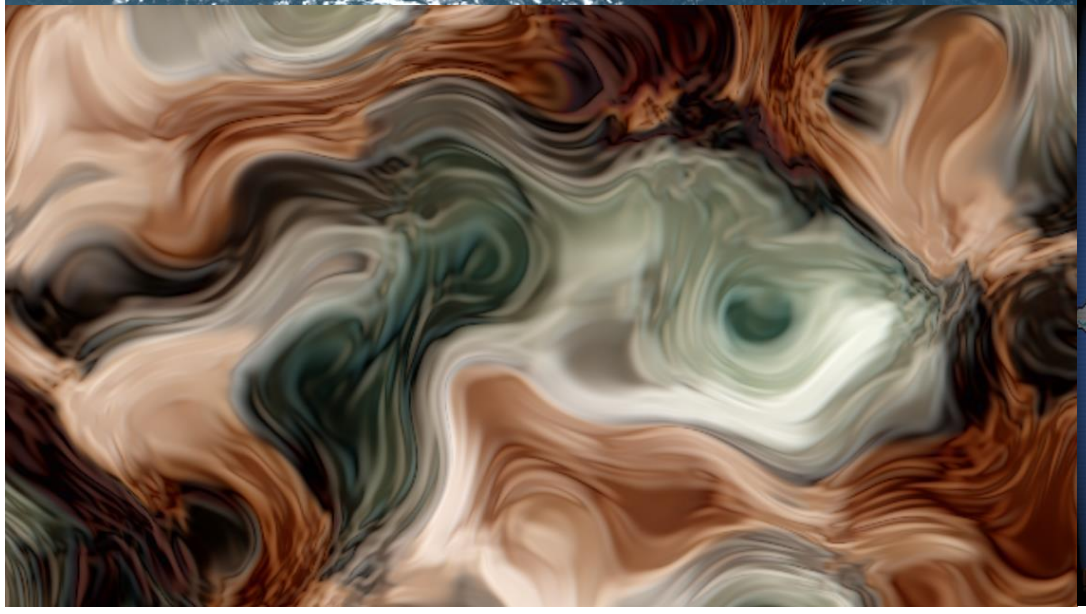
```
// Created by inigo quilez - iq/2014
// License Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

// The final product of some live coding improv. The process is live narrated in this
// video: https://www.youtube.com/watch?v=0ifChJ0nJfM

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 p = fragCoord.xy / iResolution.xy;
    vec2 q = p - vec2(0.33,0.7);

    vec3 col = mix( vec3(1.0,0.3,0.0), vec3(1.0,0.8,0.3), sqrt(p.y) );

    float r = 0.2 + 0.1*cos( atan(q.y,q.x)*10.0 + 20.0*q.x + 1.0);
    col *= smoothstep( r, r+0.01, length( q ) );

    r = 0.015;
    r += 0.002*sin(120.0*q.y);
    r += exp(-40.0*p.y);
    col *= 1.0 - (1.0-smoothstep(r,r+0.002, abs(q.x-0.25*sin(2.0*q.y))))*(1.0-smoothstep(0.0,0.1,q.y));

    fragColor = vec4(col,1.0);
}
```

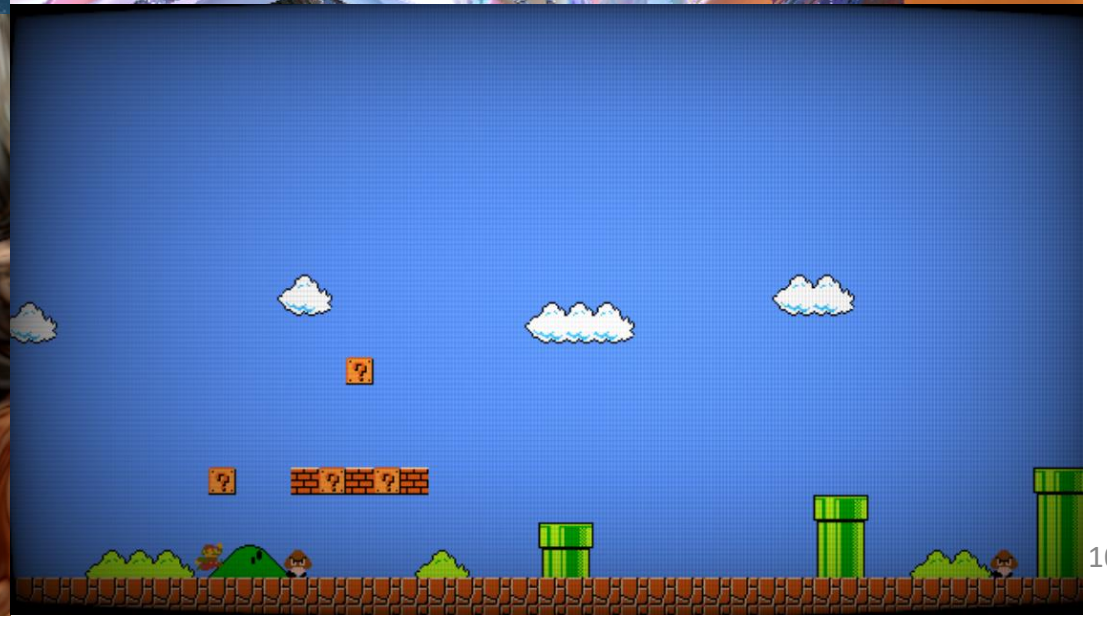
Compiled in 0.0 secs **(analyze)** 430 chars S ?

Examples









Questions???