OpenGL and shaders

A gentle introduction

"Teď už vím, že nemusím se bát
Tvé oči nejsou z tohoto světa"
- Visací zámek

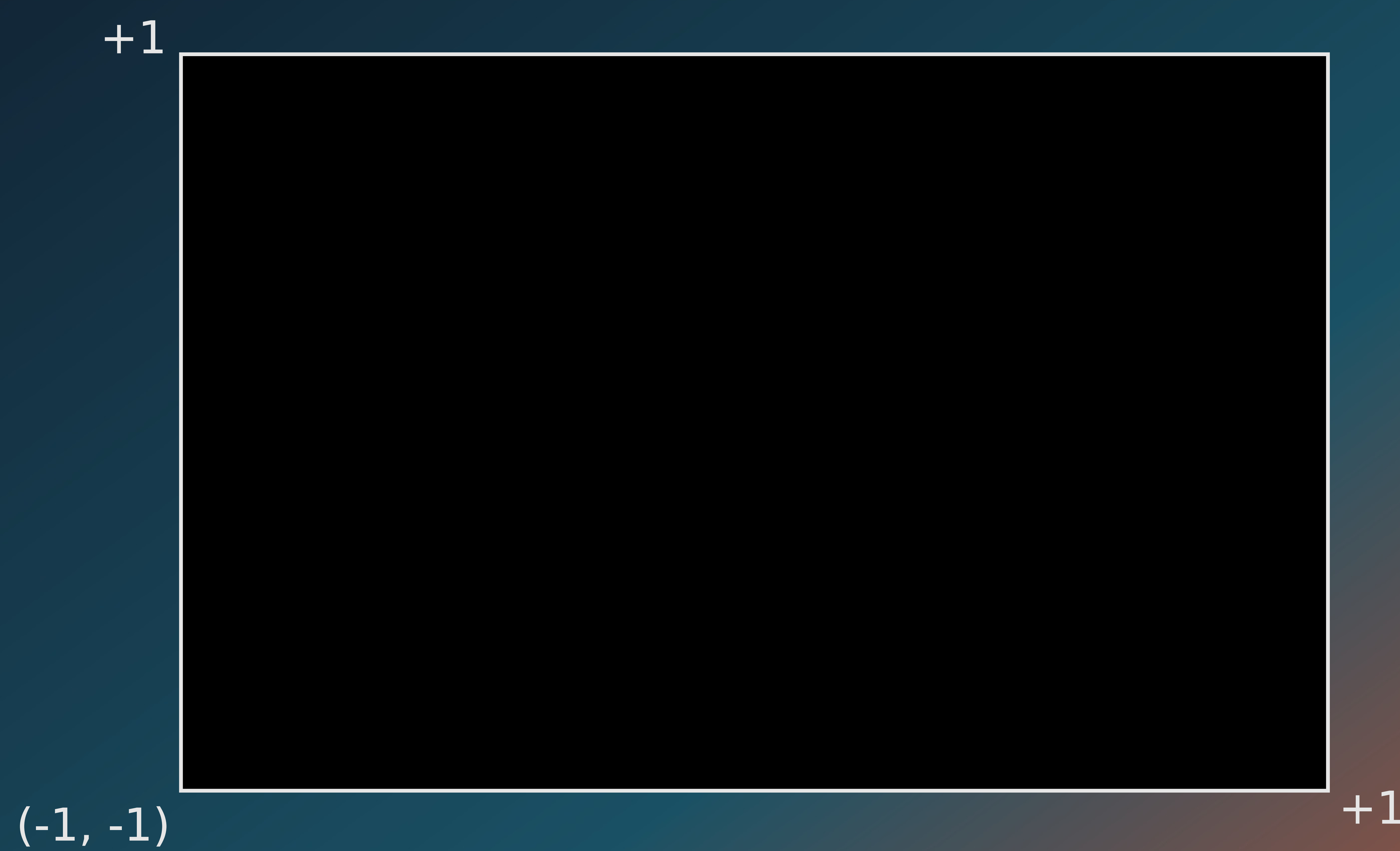
Jiri "BlueBear" Dluhos
(jiri.bluebear.dluhos@gmail.com)

Why the "new" OpenGL?

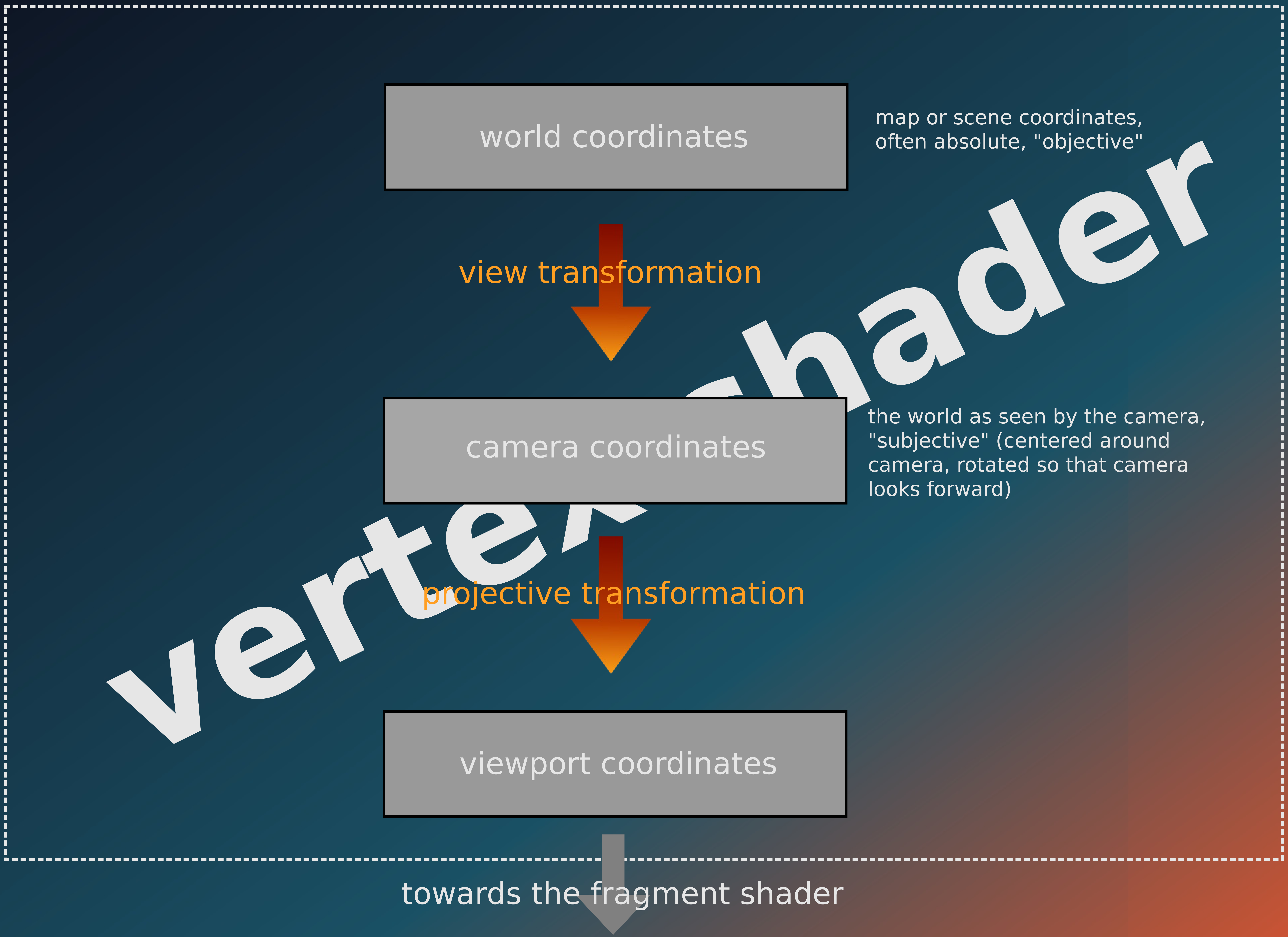
- idea in fact very old (since GL 2.0)
- what made the difference is the programmable pipeline
- all 3D transformations, vertex formats, and methods of rendering are programmer-settable, almost nothing is fixed
- so much freedom that it, for some, looks like chaos; you need to choose almost everything and keep it consistent
- lots of extensions add to the complexity
- in short, the freedom can make you mad

Viewport coordinates

- maps the whole screen/window to (-1, +1) in both coordinates (scales automatically; (0, 0) in centre makes perspective matrices nicer)



Coordinate transformations



Vertex shader

- called for each vertex
- input: vertex *attributes* (at a minimum, its position)
- output:
  - at a minimum, vertex coordinates in screen coordinates (viewport coordinates + Z coordinate (depth) + W coordinate (weight))
  - plus anything else the programmer wants to pass to the fragment shader (e.g. color, texture coordinate, etc.)
- output from the vertex shader is automatically *interpolated* and *perspective corrected* (unless the programmer specifies otherwise), then passed to the fragment shader
- basic usage:
  - coordinate transformations (MVP: model, view, projection matrix)
  - per-vertex lighting

Trap #1

with the vertex shader, you can choose your coordinate system, especially:

- left-handed or right-handed
- left- or right- multiplication
  - P*MV**vertex* (vertex is a column vector) or
  - *vertex**MV*P (vertex is a row vector)

... but that leads to differences between various documents and how-tos

Fragment shader

- called for each *fragment* (pixel candidate)
  - note: this is very often, easily >1 million calls per scene
- input:
  - vertex viewport coordinates + Z coordinate + weight
  - any extra values passed from vertex shader, interpolated and perspective-corrected by the hardware (typically texture coordinates, normal and tangent vector)
- output:
  - fragment color *or* discarding a fragment
- basic usage:
  - texturing, texture blending, fog
  - per-fragment lighting (using vectors passed from the vertex shader)
  - bump-mapping (per-fragment lighting alterations based on a texture)

A trivial shader combo

```
// VERTEX SHADER
#version 120                // this code is GLSL 1.2 compatible

// vertex attributes - what is sent to us for every vertex
in vec3 vCoords;            // vertex coordinates
in vec3 vColor;             // vertex color (RGB)

uniform mat4 vModelviewMatrix;
uniform mat4 vProjectionMatrix;

varying vec3 fColor;        // sent to fragment shader, varies across the rendered triangle

void main()
{
    gl_Position = vProjectionMatrix * vModelviewMatrix * vCoords;
    fColor = vColor;
}
```

```
// FRAGMENT SHADER
#version 120

in vec3 fColor;                 // passed from the vertex shader and interpolated across the triangle

void main()
{
    gl_FragColor = vec4(fColor, 1.0);
}
```

Exchanging data with shaders

- **shader variables**
  - *vertex attributes* (sent for each vertex when drawing)
  - *uniforms* (directly settable/gettable, stay constant during one drawing operation)
- **GPU-side buffers**
  - reside in GPU memory (fast access from shaders)
- **textures**
  - original use: pixmaps to draw on various objects
  - can be used to send (and receive) any data (vectors, height values), can be even drawn onto

Trap #2

Starting from GL3.2, all data buffers are in GPU memory
+ much faster (thus allowing for better effects)
- you have to create and properly initialize about 4 objects just to get a triangle rendered (cca 15 GL calls)

dreaded "black screen of doom"

- when you have your code almost right (no GL errors are produced) but nothing visible (some initialization is forgotten somewhere)

Trap #3

GL API is historically a state machine and this still mostly persists; it has lots of global state

GL object -> configure -> use

GL object -> bind -> configure -> unbind
GL object -> bind -> use -> unbind

(better with some extensions and GL>4.2)

A simple triangle (simplified)

```
// Describes a single vertex
class Vertex {
public:
    glm::vec3 coords;
    glm::vec3 color;

    Vertex(const glm::vec3 coords, const glm::vec3 &color)
        : coords(coords), color(color) {}
};

// A multicolored triangle
std::array<Vertex, 3> vertices = {
    Vertex(glm::vec3(-1.0f, -1.0f, 0.0f), glm::vec3(1.0f, 0.0f, 0.0f)),
    Vertex(glm::vec3(1.0f, -1.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f)),
    Vertex(glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f))
};

// preparation code: create a GPU-side buffer and upload the triangle data into it
glGenBuffers(1, &vertex_buffer);
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, vertices.size()*sizeof(Vertex), vertices.data(), GL_STATIC_DRAW);

// preparation code: locate the attributes (their offsets) in the shader's attribute table
attribute_coords = glGetAttribLocation(shader_program, "vCoords");
attribute_color = glGetAttribLocation(shader_program, "vColor");

// ... in rendering code:
glUseProgram(shader_program);
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glVertexAttribPointer(attribute_coords, 3, GL_FLOAT, GL_FALSE,
    sizeof(Vertex), reinterpret_cast<void*>(offsetof(Vertex, coords)));
glVertexAttribPointer(attribute_color, 3, GL_FLOAT, GL_FALSE,
    sizeof(Vertex), reinterpret_cast<void*>(offsetof(Vertex, color)));
glEnableVertexAttribArray(attribute_coords);
glEnableVertexAttribArray(attribute_color);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Resources

- Wikipedia has many good articles
  - start here: en.wikipedia.org/wiki/3D_projection
- www.lighthouse.org
- www.opengl.org/wiki - very practical reference