Computer Graphics 4 - OpenGL

Tom Thorne

thomas.thorne@ed.ac.uk www.inf.ed.ac.uk/teaching/courses/cg/

Overview

- OpenGL introduction
- Transformations
- GLSL
- Vertex shaders
- Fragment shaders
- Coursework

OpenGL

- C style API
- Internal state modified by function calls
- Versions 1.1 -> 4.5

Other APIs - DirectX, Vulkan, Metal

OpenGL API

Makes a lot of use of internal state e.g. functions like:

```
glBindBuffer(GL_ARRAY_BUFFER, objectVB);
```

Subsequent API calls acting on the array buffer will refer to the array buffer bound here.

Won't be teaching API - you don't need to know it for coursework, if you want to look functions up use:

- `http://docs.gl`

OpenGL API

Old style – immediate mode

```
glBegin(GL_TRIANGLES);
glColor3f(r1, g1, b1);
glVertex3f(v1.x,v1.y,v1.z);
glColor3f(r2, g2, b2);
glVertex3f(v2.x,v2.y,v2.z);
glColor3f(r3, g3, b3);
glVertex3f(v3.x,v3.y,v3.z);
glEnd();
```

Modern GPUs and APIs tend to seperate read only state (data) and drawing commands – programmable pipeline

OpenGL API

Pointers to functions for extensions must be retrieved and placed into variables (for every function you want to use!)

```
void(*)() f=glXGetProcAddress("glCreateShaderObjectARB");
f(...);
```

Solution: use GL extension wrangler (GLEW, http://glew.sourceforge.net)

OpenGL types

Primitive types:

```
GLfloat x;
GLint a;
GLuint b;
```

GLM library:

```
glm::vec4 x(1.0,1.0,1.0,1.0);
glm::mat4 y;
```

Buffers:

```
GL_RGB
GL_RGBA
GL_DEPTH24_STENCIL8
```

Initialisation

Creating windows/fullscreen, loading and compiling shader programs, reading object and texture files.

Window creation/UI:

- SDL
- GLFW
- Qt
- GTK+

Transformations

The old way:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1.0, 1.0, -1.0, 1.0, 3, 20.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

The new way:

- Generate matrices on CPU
- Transfer to GPU as a *uniform*
- Apply transformations in the vertex shader

Transformations

Using glm we can easily construct transformations and viewing matrices:

```
using namespace glm;
mat4 Projection =
    perspective(45.0, 4.0 / 3.0, 1.0, 20.0);
mat4 View =
    lookAt(eye, vec3(0, 0, 0), vec3(0, 1, 0));
mat4 Model =
    rotate(mat4(1.0), x_angle, vec3(1, 0, 0));
mat4 MVP = Projection * View * Model;
```

Build it yourself:

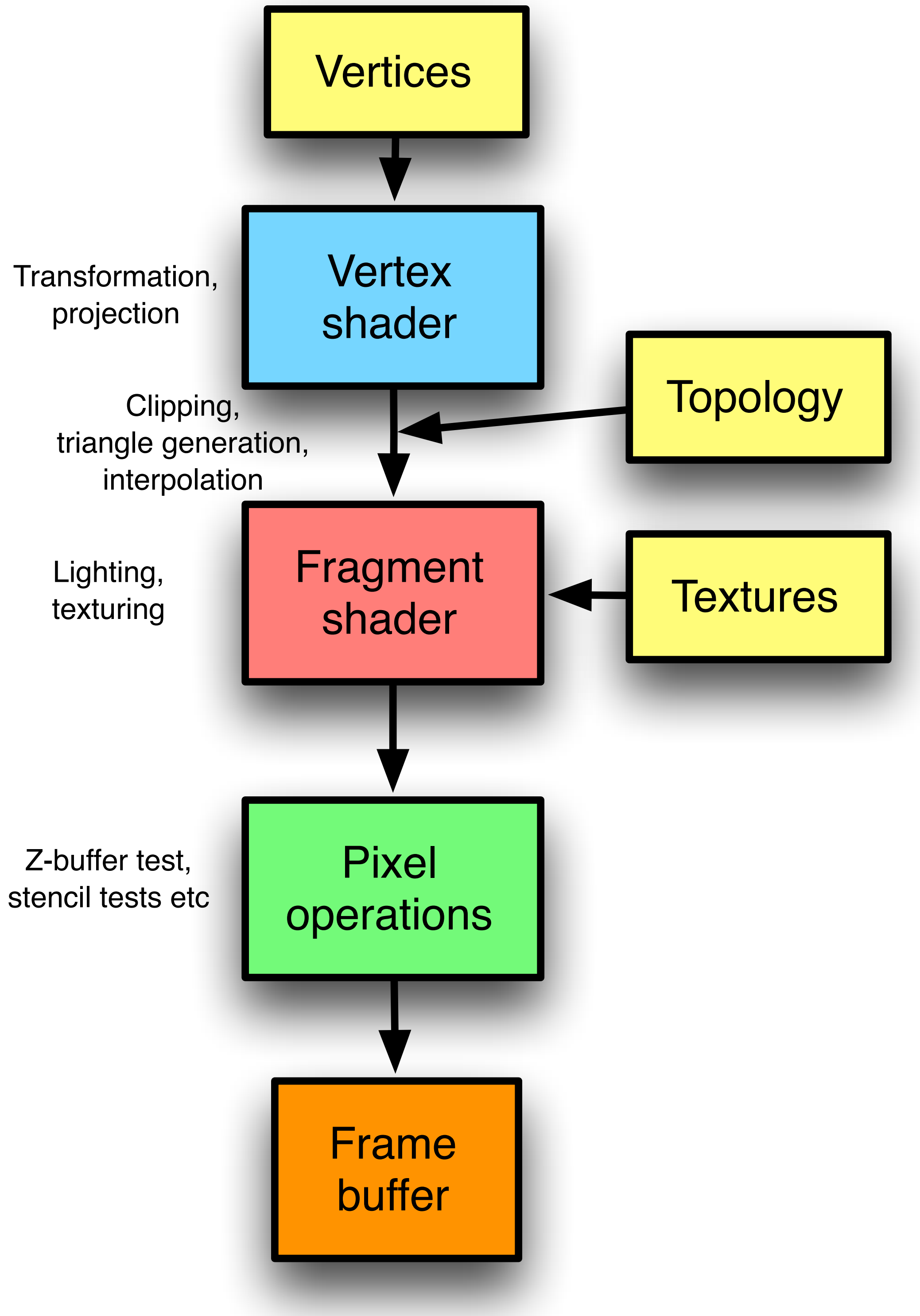
```
mat4 m;
m[0][0]=1.0;
//...
```

this is very similar syntax to GLSL

Background: GPU architecture

- Many graphics algorithms are easily parallelisable
- Not always using the fastest algorithm, but the most easily parallelisable
- Single instruction multiple data (SIMD) and Single Program Multiple Data (SPMD)
- Very good at performing the same floating point operation on lots of values at once

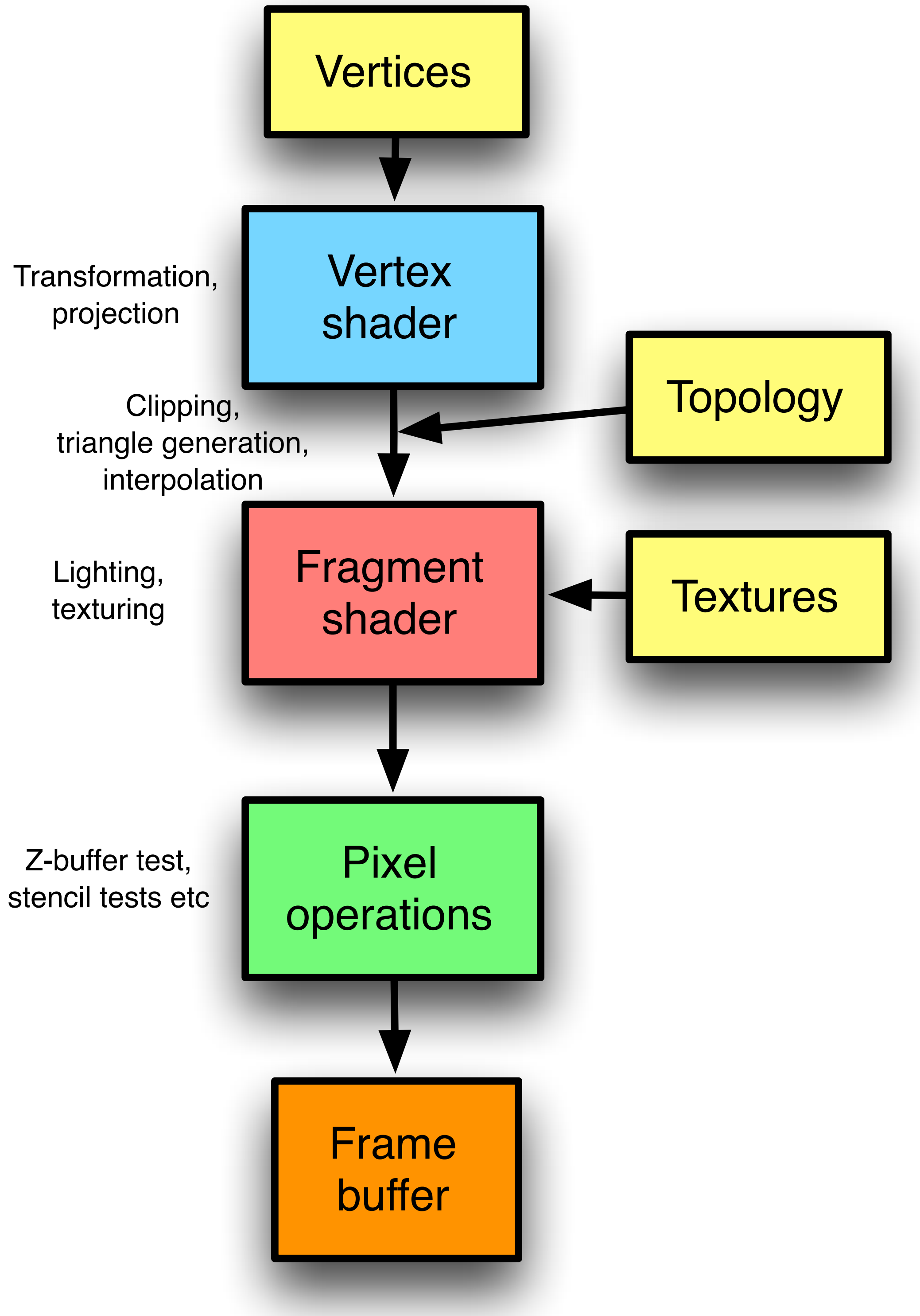
OpenGL programmable pipeline



Fixed state:

- Vertex data
  - Coordinates
  - Normals
  - Surface colour etc
  - Texture coordinates
- Polygon data
  - Indices within vertex list
- Textures
  - MIPMAPs automatically generated
  - Linear interpolation

OpenGL programmable pipeline



Programmer provides:

- Vertex shader
- Fragment shader

Output:

- Fixed function (but configurable) pixel operations
- Renders fragments into buffer. . .

GLSL

GL shader language:

- Distributed as source code
- Compiled to GPU (specific) machine code by graphics drivers
- Very similar syntax to C/glm
- Becoming more and more general
- Still some limitations - *e.g.* no recursion

```
void main() {
    vec4 v=vec4(position,1.0);
    norm=normalize(MN*vec4(normal,0.0));
    gl_Position=MVP*v;
}
```

`www.shadertoy.com`

Versions

Main changes are to specification of input/output from shaders:

```
#version 120
//OpenGL 2.1
attribute vec4 normal;
varying vec4 normalTrans;

#version 130
//OpenGL 3.0
in vec4 normal;
out vec4 normalTrans;

#version 330
//OpenGL 3.3
layout(location=0) in vec3 normal;
out vec4 normalTrans;
```

coursework uses 1.30 (130)

Vertex shaders

Run once on every *vertex*

- Transformation from local coordinates to projected screen coordinates
- Transformation of normal vector
- Extra processing of vertex attributes (*e.g.* colour, texture coordinates)

Vertex shaders - example

In GLSL:

```
#version 130

//uniform named "mvp"
uniform mat4 mvp;

//Vertex position input
in vec3 position;

void main() {
    vec4 v=vec4(position,1.0);
    //gl_Position is a special variable
    //for the vertex position
    gl_Position=mvp*v;
}
```

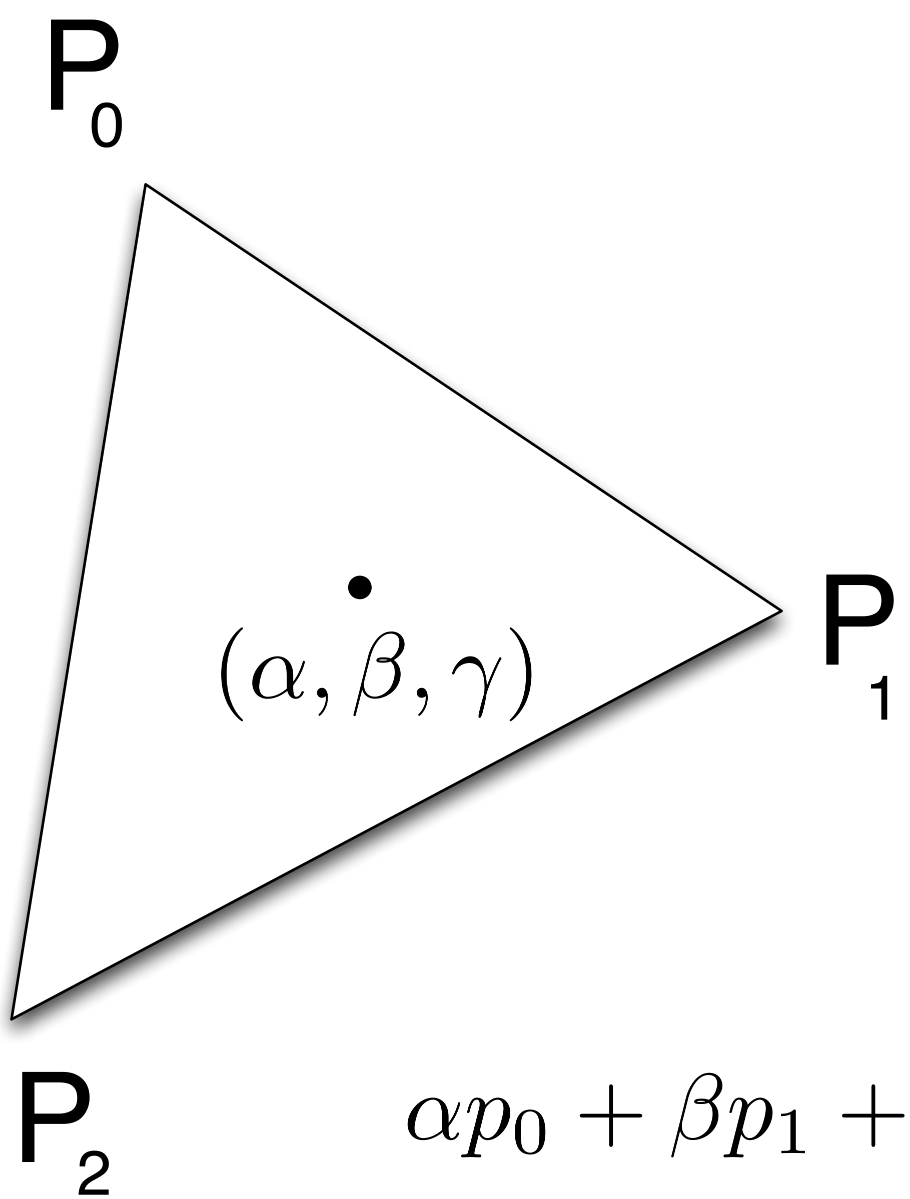
Fragment shaders

Run once for every fragment (pixel) inside a polygon

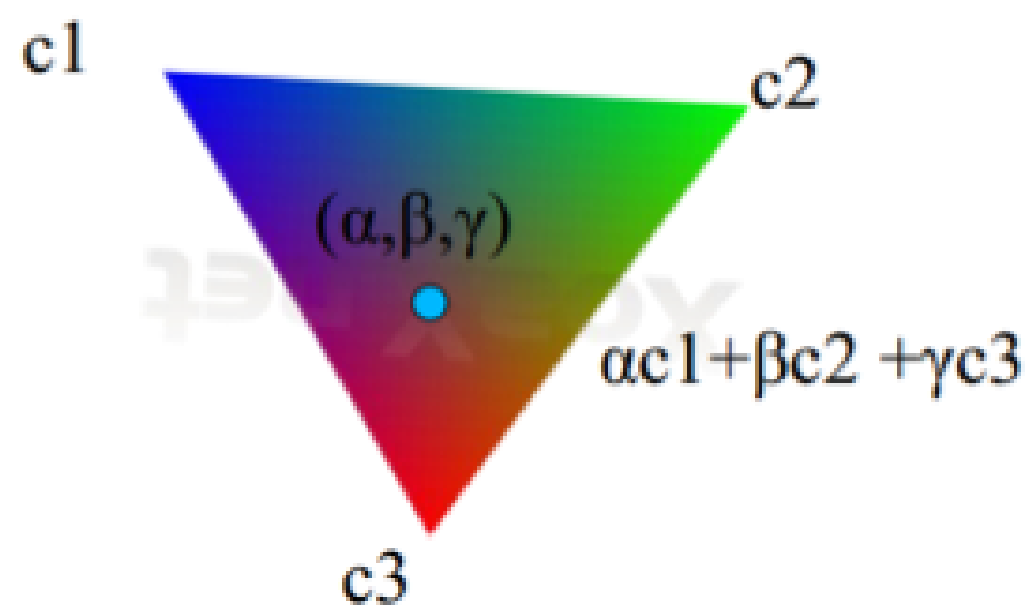
- Calculate lighting per pixel in a triangle
- GPU will automatically interpolate vertex attributes
- Access texture data (special functions)

Interpolation

Vertex attributes output from the vertex shader are interpolated by the GPU between the vertices of triangles, and given as input to the fragment shader for each pixel



$$\alpha p_0 + \beta p_1 + \gamma p_2$$



Fragment shaders

Output from a vertex shader:

```
out vec4 normal;
void main()
{
 normal = calculateTransformedNormal(inputNormal);
}
```

Process in fragment shader:

```
in vec4 normal;
out vec4 outColour;
void main()
{
    float r=0.5*normal.x+0.5; float g=0.5*normal.y+0.5;
    float b=0.5*normal.z+0.5; float a=1.0;
    outColour = vec4(r,g,b,a);
}
```

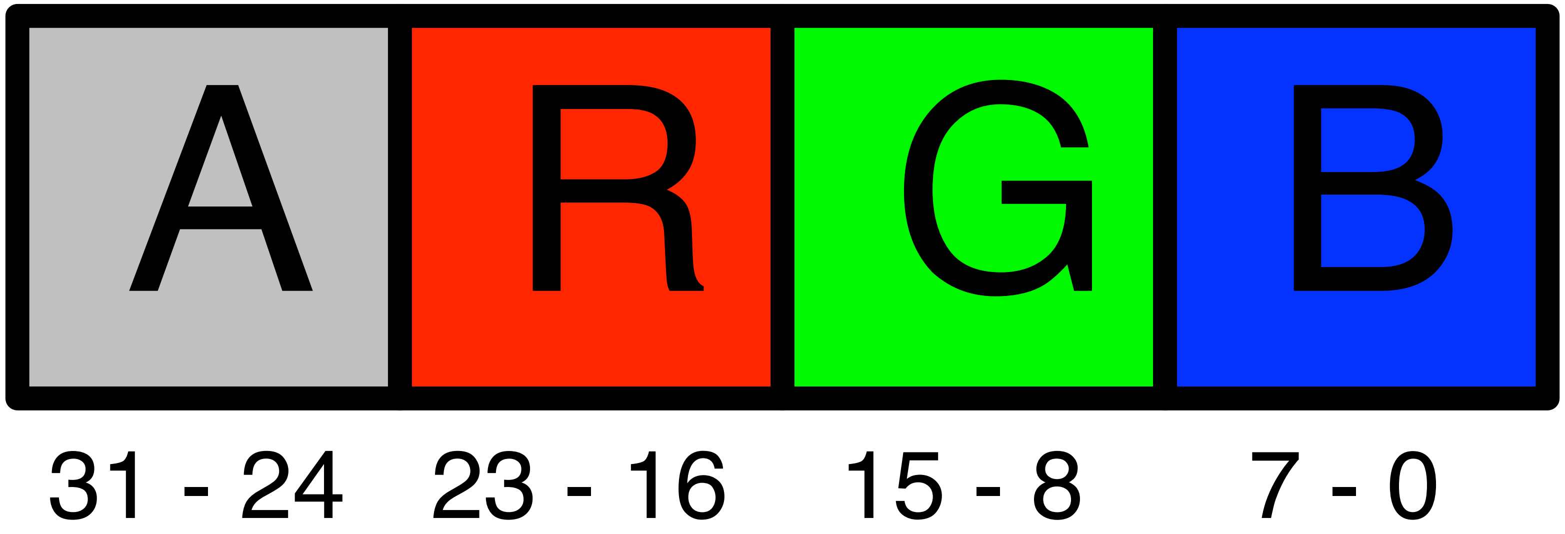
Textures

In the fragment shader:

```
uniform sampler2D texId;
void main()
{
    //x and y in range 0 to 1
    vec4 col = texture(texId,vec2(x,y));
    //col.r, col.g, col.b, col.a
    //all in range 0 to 1
    //...
}
```

Image representations

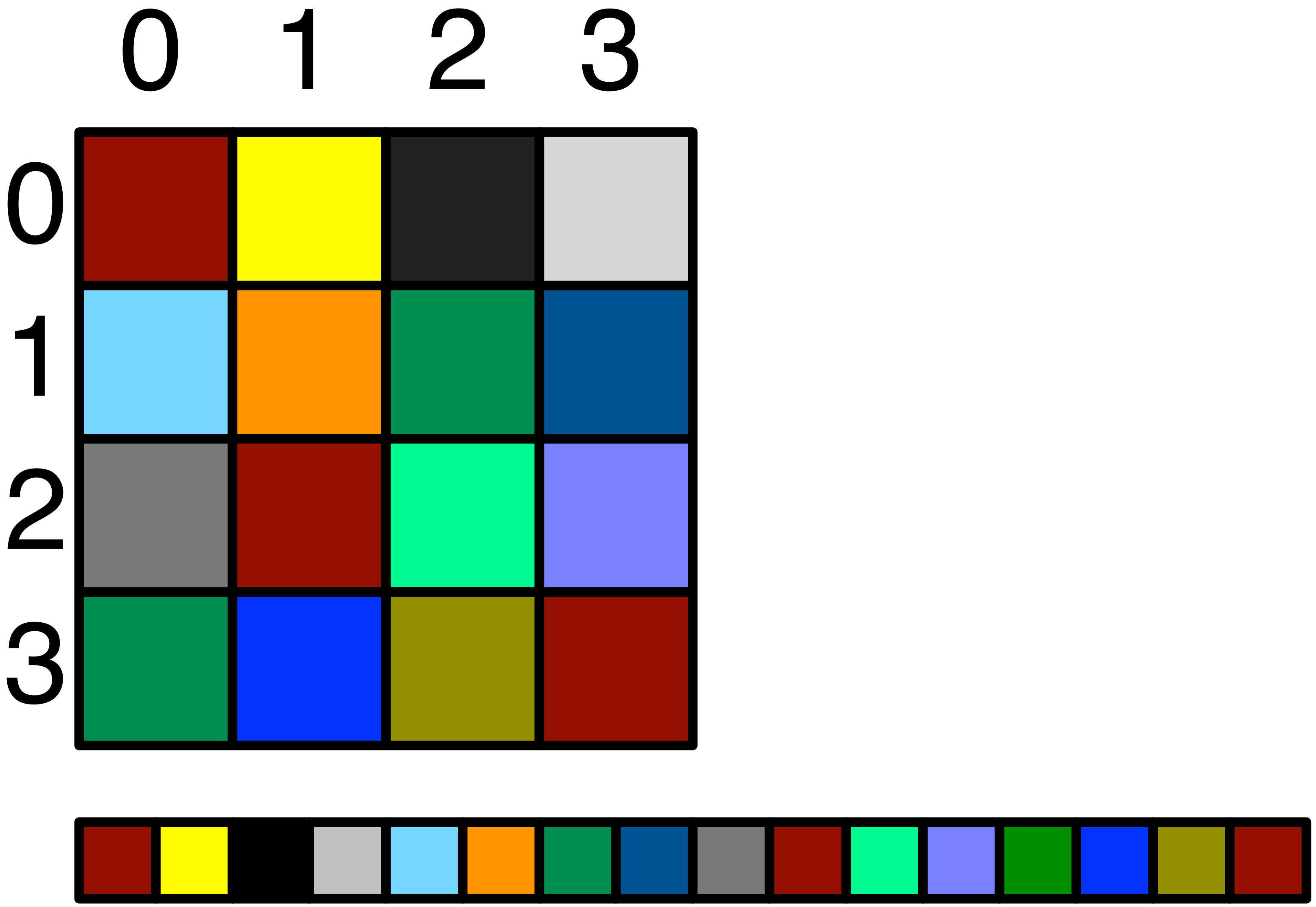
Standard 32 bit format for colours:



Other formats:

- 32bit Floats per channel (high dynamic range)
- 24bit RGB (usually padded)

Pixel buffers



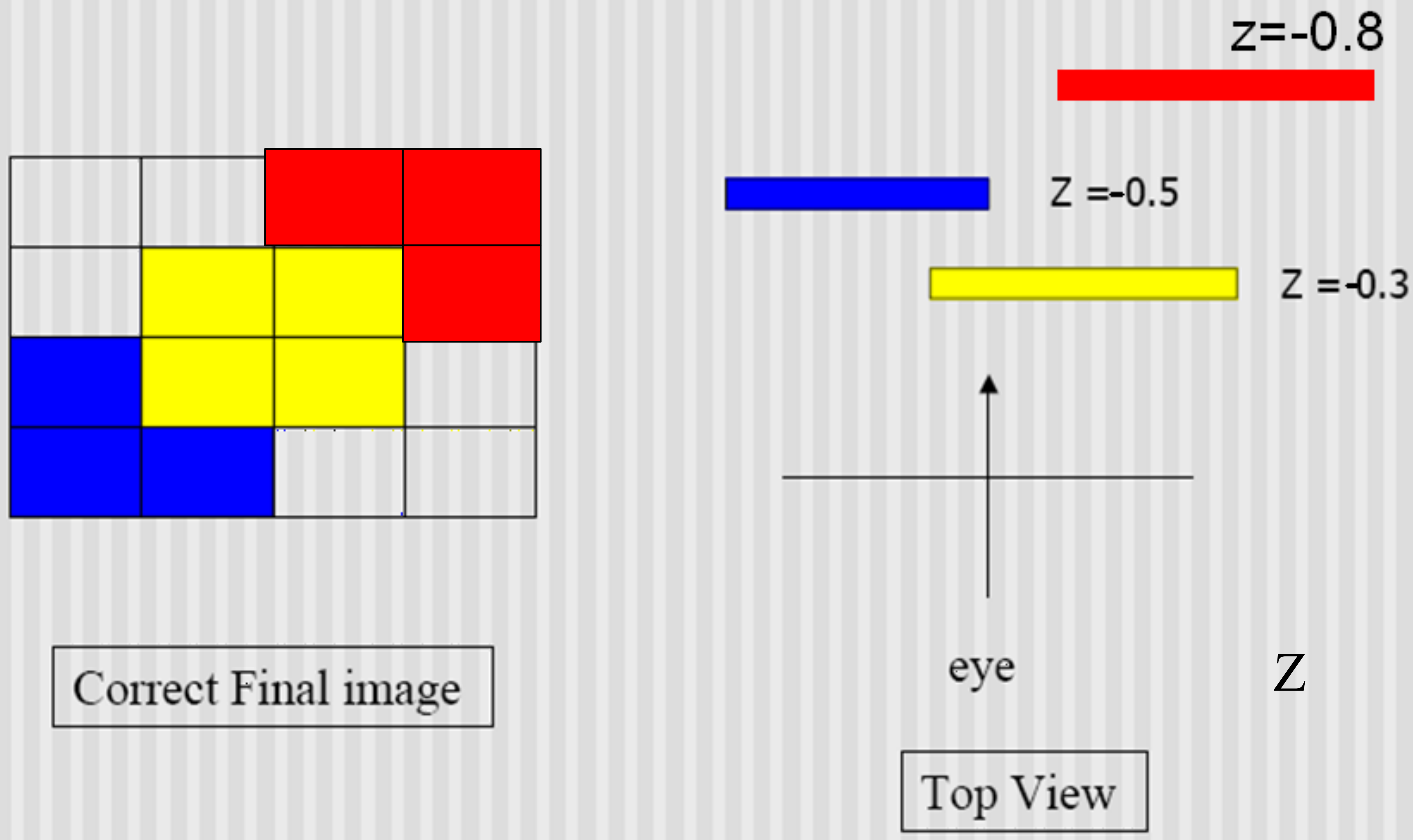

```
int array[4][4];
int arrayFlat[16];
array[y][x]=arrayFlat[y*width+x];
```

Z buffers

How do we make sure things closer to the camera are drawn on top of things behind?

- various methods covered later in the course
- current most popular method is the Z buffer
- implemented by GPU, very little work to use with OpenGL

Z buffer example



Z buffer example

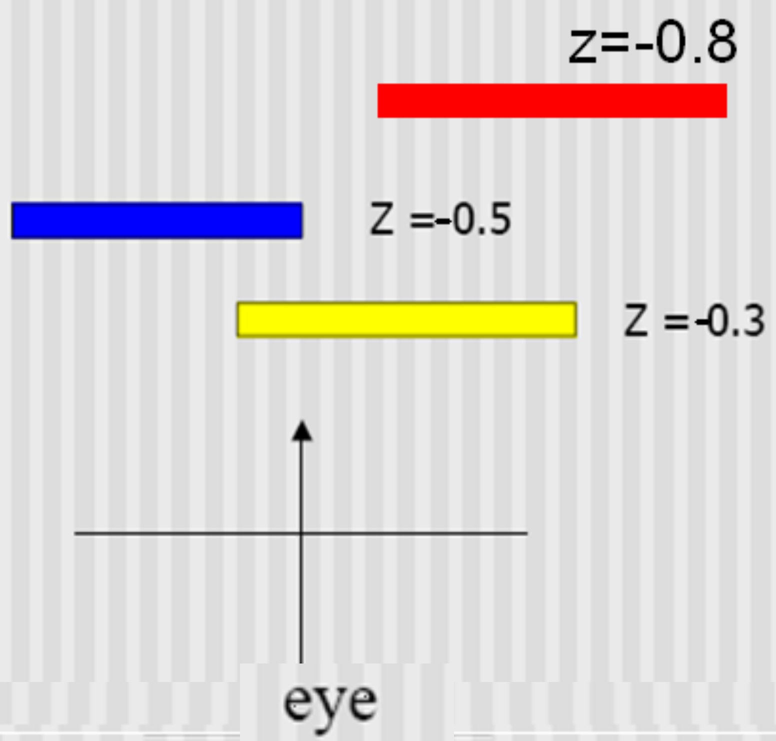
Step 1: Initialize the depth buffer

-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0

Z buffer example

Step 2: Draw the blue polygon (assuming the program draws blue polyon first – the order does not affect the final result any way).

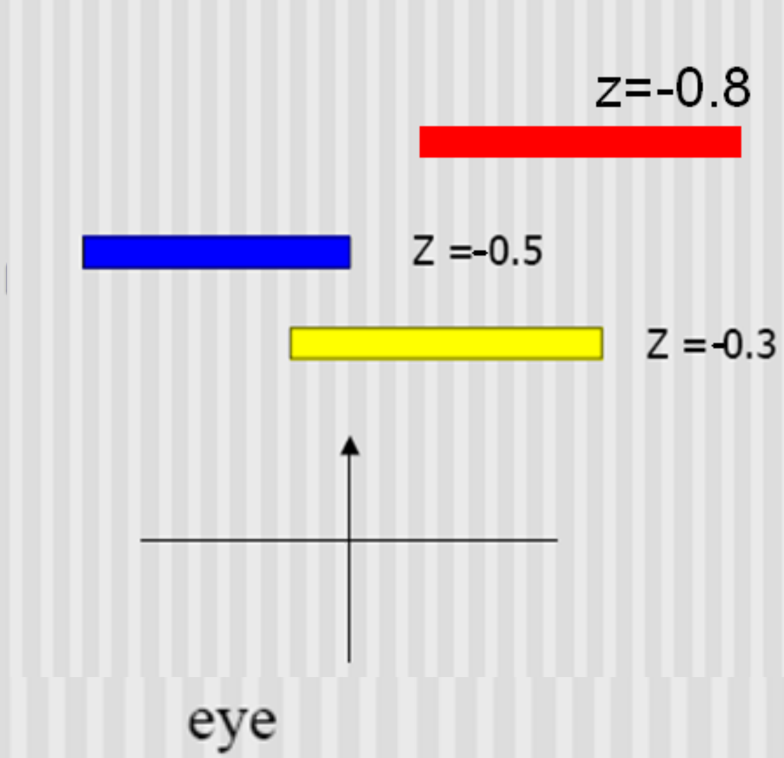
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-0.5	-0.5	-1.0	-1.0
-0.5	-0.5	-1.0	-1.0



Z buffer example

Step 3: Draw the yellow polygon

-1.0	-1.0	-1.0	-1.0
-1.0	-0.3	-0.3	- 1.0
-0.5	- 0.3	- 0.3	-1.0
-0.5	-0.5	-1.0	-1.0

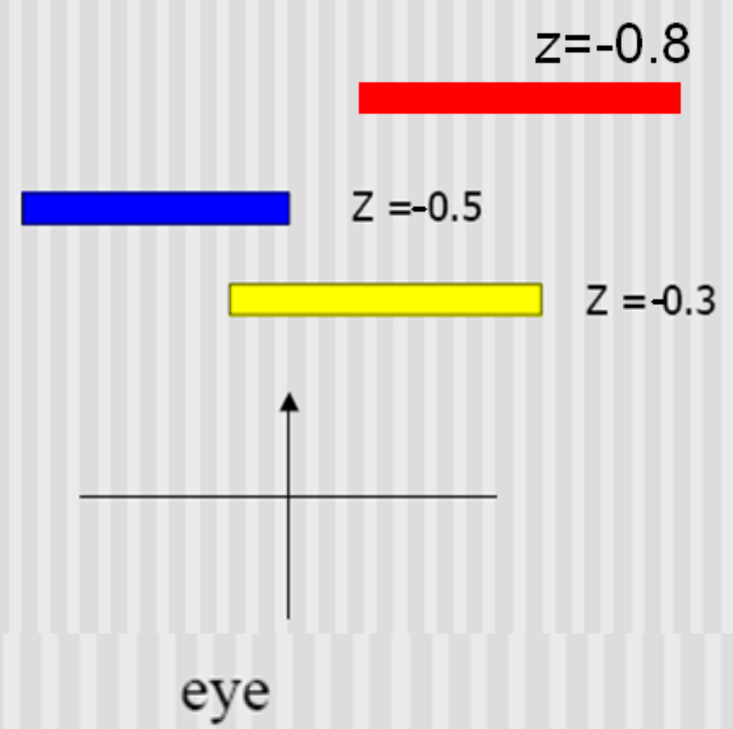


If the depth value is larger than that in the z-buffer, the pixel is coloured and value in the z-buffer is updated

Z buffer example

Step 4: Draw the red polygon

-1.0	-1.0	-0.8	-0.8
-1.0	-0.3	-0.3	-0.8
-0.5	-0.3	-0.3	-1.0
-0.5	-0.5	-1.0	-1.0



If the depth value is larger than that in the z-buffer, the pixel is coloured and value in the z-buffer is updated

Render to texture

Very useful technique for:

- Shadows
- Advanced lighting/environment mapping
- Postprocessing effects

```
glGenFramebuffers(1, &frameBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
\\... (generate a texture, store ID in renderTarget)
glFramebufferTexture2D(GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, renderTarget, 0);
```

- We can treat this texture just like any other, e.g. reading from it in a fragment shader.
- Render into multiple textures then combine to produce final image

Coursework

Coursework 1 starting point:

- OpenGL framework
- Uses glfw to handle basic window generation
- glm for maths
- Loads object file and creates vertex normals

To do:

- Vertex shader to transform
- Fragment shader for Phong illumination model
- Special effects...

Coursework

Vertex shader inputs/outputs

```
//Uniforms
//model transformation matrix
uniform mat4 model;
//view transformation matrix
uniform mat4 view;
//projection transformation matrix
uniform mat4 proj;
//normal transformation matrix
uniform mat4 normTrans;
//coordinate of eye/camera
uniform vec4 eyePos;
//coordinate of light source
uniform vec4 lightPos;
```

```
//Vertex attributes
//Vertex position
in vec3 position;
//Vertex normal
in vec3 normal;

//Outputs
//Vector to light source
out vec4 lightVec;
//Vector to eye/camera
out vec4 eyeVec;
//transformed surface normal
out vec4 normOut;
```

Coursework

Fragment shader

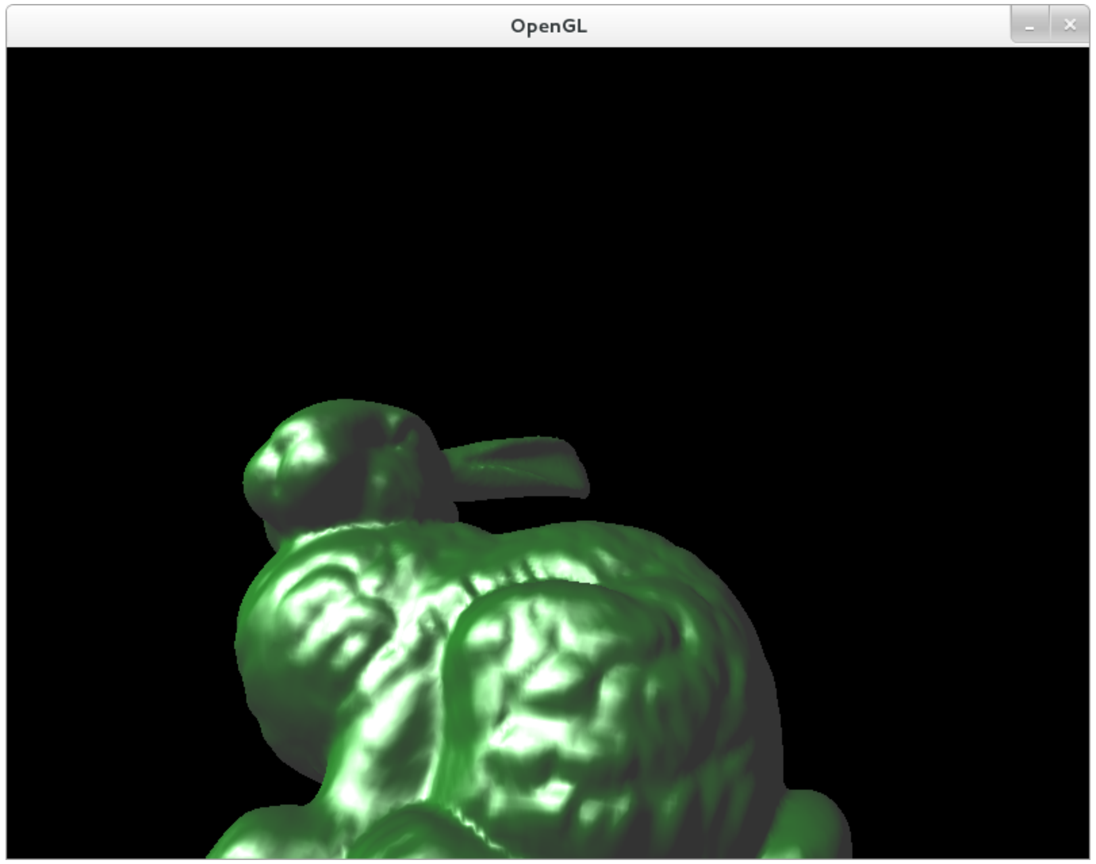
```
in vec4 lightVec;
in vec4 eyeVec;
in vec4 normOut;

uniform sampler2D tex;

out vec4 outColour;

void main() {
    float diff=0.4;
    float spec=0.2;
    float ambient=0.2;
    outColour=
    vec4(spec+ambient, spec+diff+ambient, spec+ambient, 1.0);
}
```

Coursework



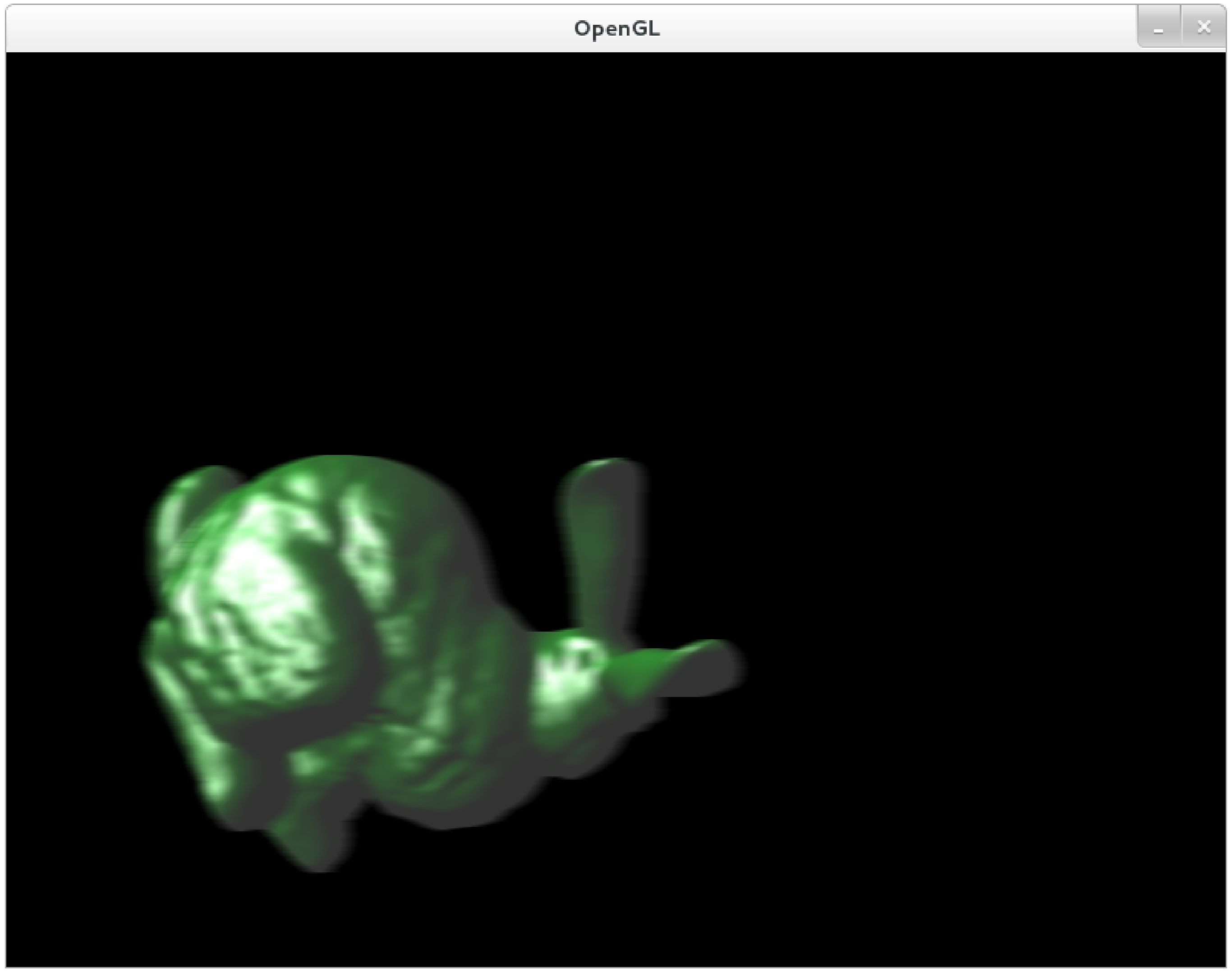
Fragment shader effects

A standard method for postprocessing (in 2D) using render to texture:

- Render scene into a texture and depth buffer
- Generate two large triangles covering the whole screen
- Fragment shader for triangles reads from scene texture and depth buffer
  - Ambient occlusion
  - Chromatic abberration
  - Bloom

Coursework ideas - blur

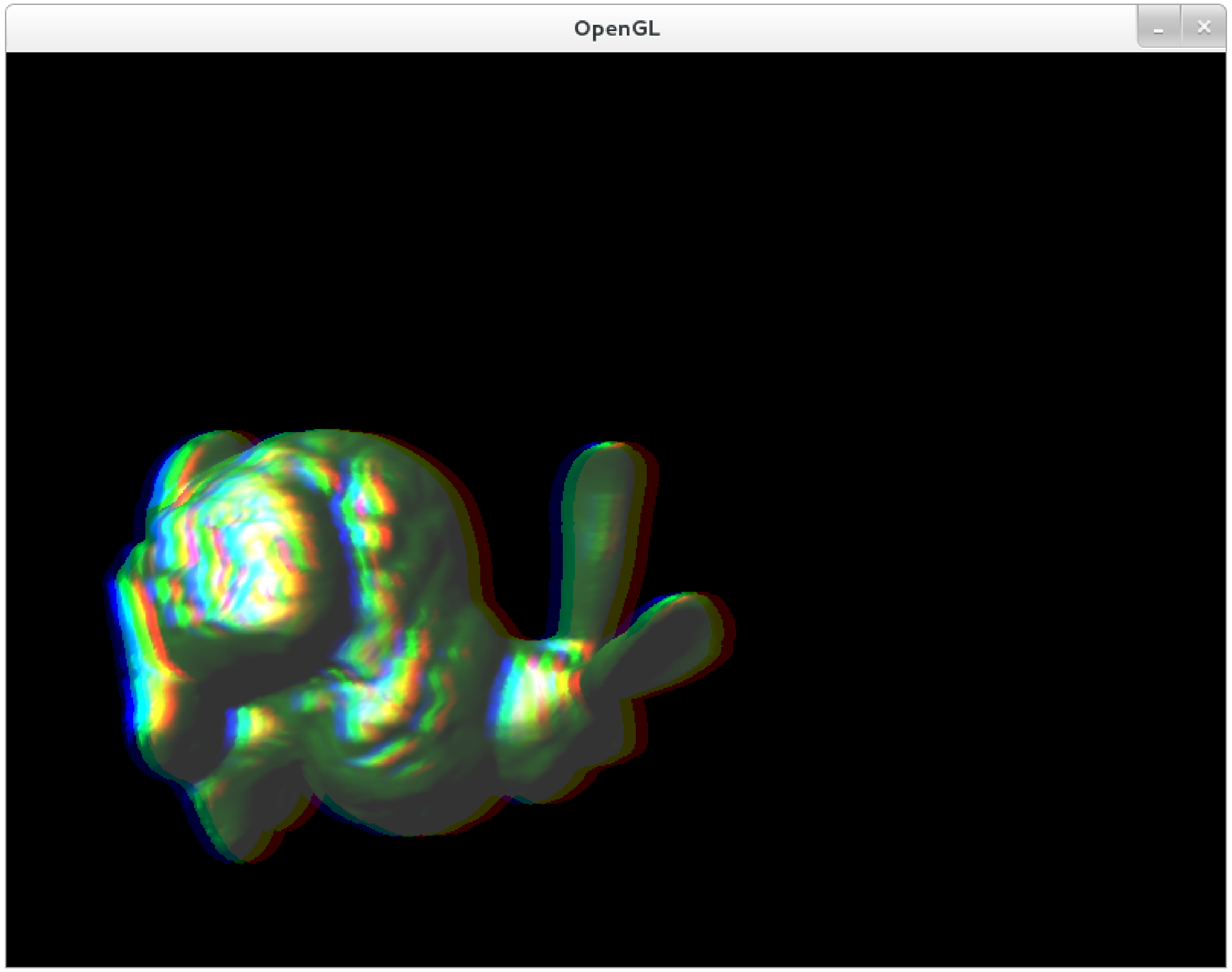
A simple blur effect - take the average of a small number (e.g. 4) of surrounding pixels.



Coursework ideas - chromatic aberration

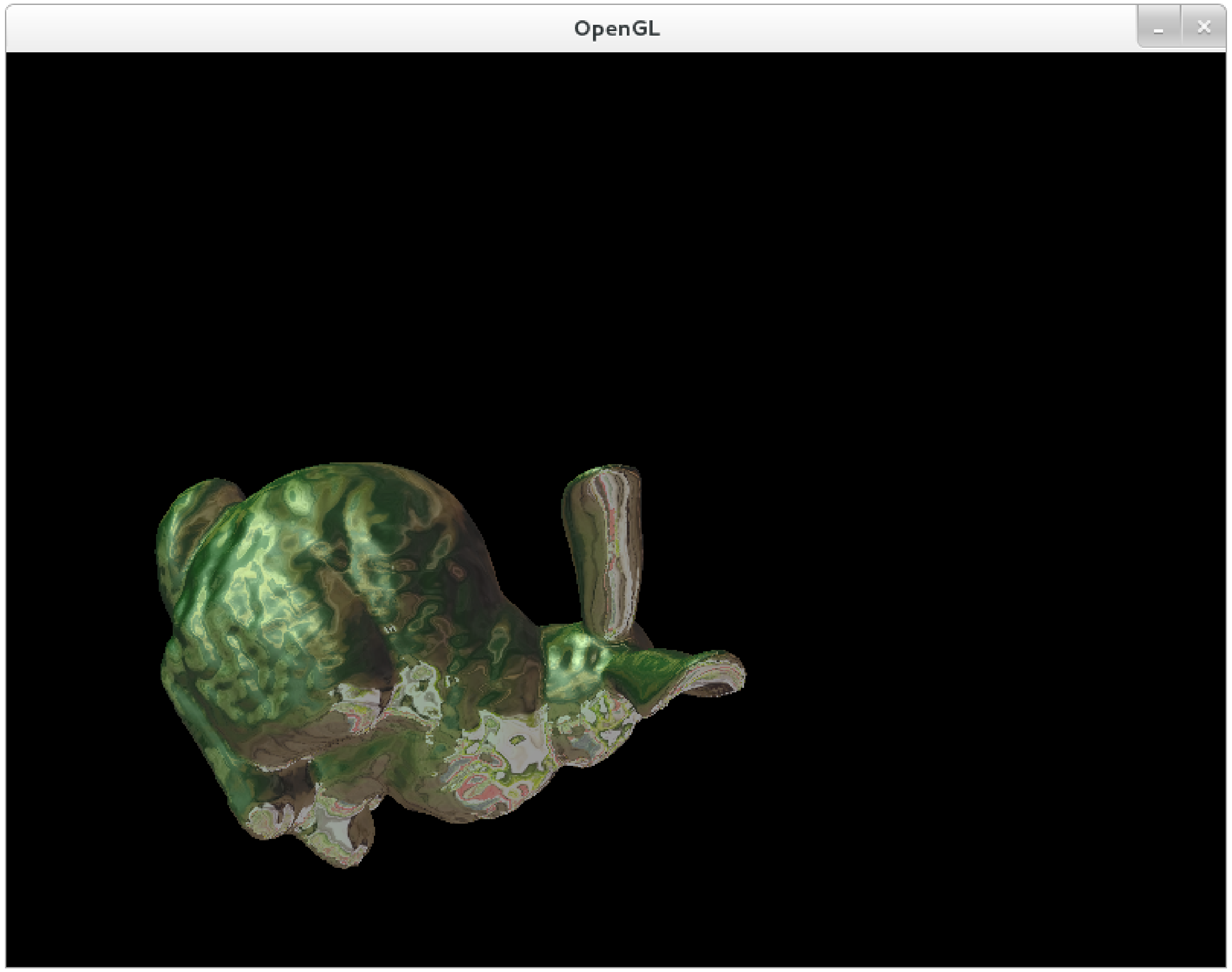
Caused by differing focal lengths of varying wavelengths of light, heavily (mis)used in video games.

Easy trick to produce this effect – offset red green and blue channels of image by small amounts.



Coursework ideas - environment mapping

Use x and y coordinates of reflection of eye vector around normal vector to index into the texture (in fragment shader for the 3D object, not in 2D postprocessing).



References

OpenGL tutorial:

- http://open.gl

OpenGL API reference:

- http://docs.gl

Examples of fragment shaders:

- http://shadertoy.com