Welcome to....

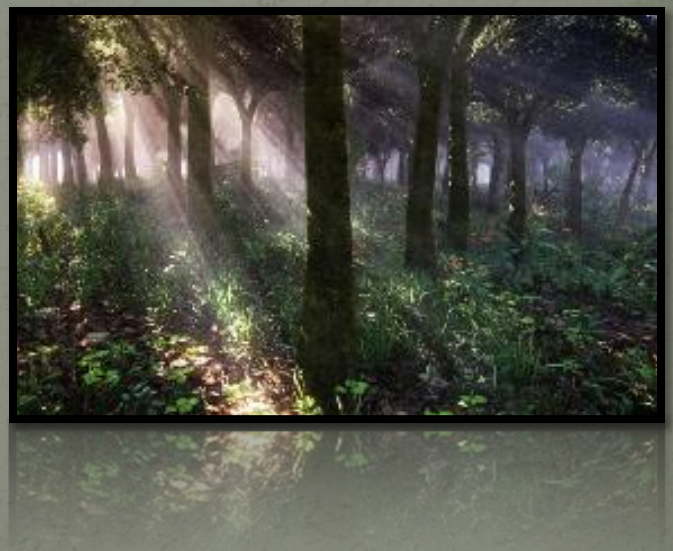
3D Rendering

An introduction to OpenGL & the Micro Game Engine

What is OpenGL?

- OpenGL == Open Graphics Library
- Standardized software interface to graphics hardware

- Allows us to:
  - Create (interactive) programs that produce (colored) images (of moving three-dimensional objects)
  - And more…



What is the Micro Game Engine?

- Small collection of C++ classes taking care of:
  - OpenGL initialization
  - Scene construction:
    - Loading meshes
    - Creating & adding behaviours
    - Constructing a nested scene graph
    - Creating/setting/configuring materials
    - Rendering the whole thing
  - Basically the GXPEngine…
    - but then 3d…
    - and in c++…

Course topics ?

- OpenGL basics
  - Background
  - Programmable rendering pipeline
  - Shader programming
- 3D basics
  - Vertices, UVs, normals & manipulating them with shaders
  - Matrix transforms (applied 3D math)
- Lighting basics:
  - Ambient, diffuse, specular lighting components
- Texturing basics:
  - Loading textures, height/splat/tri-planar mapping
- …using barebones examples & the Micro Game Engine

Buckle up for the ride ☺

- Starting with OpenGL can be pretty daunting:
  - Several 3rd party libraries required
  - Large C-style API (100's of functions, constants, etc)
    - Not object oriented (eg no new OpenGL() etc)
    - No overloaded functions (every data type, separate func)
  - Lot of different OpenGL versions available
  - 3d Math; dot, crosses, matrices, etc
  - Tons and tons of information on OpenGL available
  - Only 5 weeks to get up and running ☺
- Little steps, go by example…

What's in it for you?

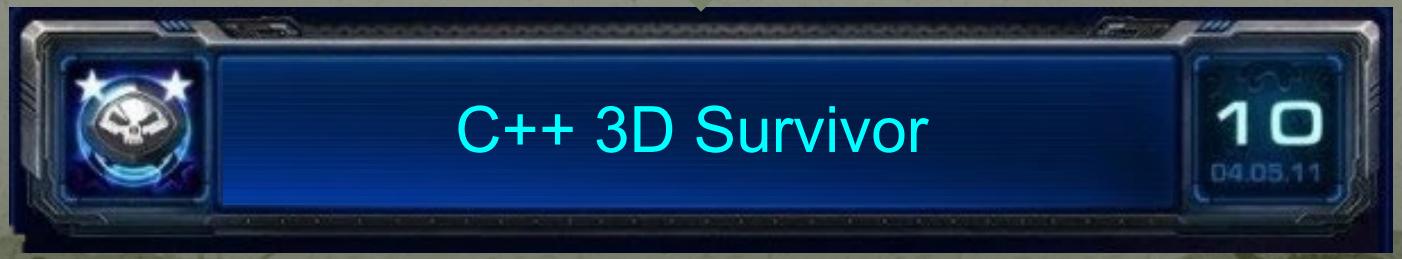
- 3 ECTS…
- Knowledge about modern GFX programming
- Knowledge about shader programming
- Cool good looking portfolio items
- It's fun and awesome just like physics was fun
- A head start with learning about C++ 3D engine dev.
- Prepares for next project → write/extend own engine

Course approach, how?

OpenGL lectures

Lab hours

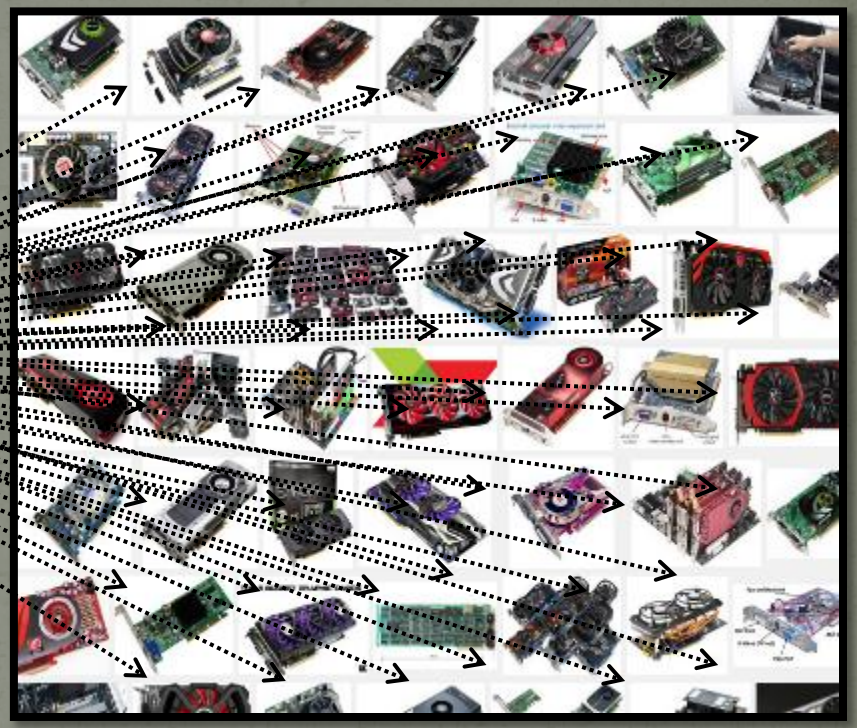
4 Graded lab assignments



OpenGL Background

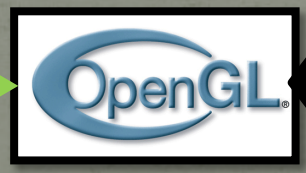
Why do we need it?
Where can we get it?
What can we do with it?
How can we run it?
What does it look like?

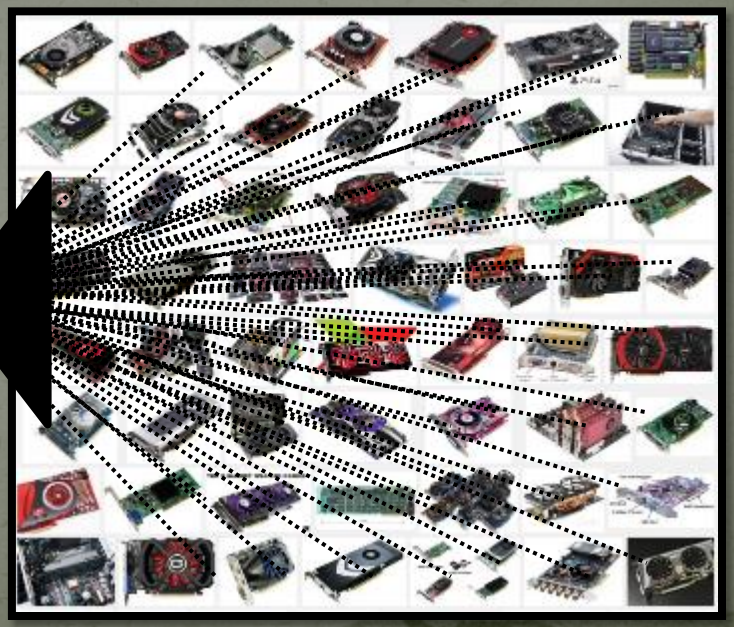
Why do we need an API like OpenGL?



How does OpenGL help?

- OpenGL provides us with a common interface, an adapter, to various graphics hardware:
  - Graphic card vendors write drivers whose API adheres to the OpenGL specification
  - Application developers access the driver through the API





Where can we "get" OpenGL?

- OpenGL is "just" a specification, no source code
- Developed by the Khronos Group
- Driver download provides an OGL **implementation**
- Driver can match whole or part of the (latest) OpenGL specification…
- How do we as programmers access all this goodness?
  - **That depends on your OS!**

OpenGL on Windows

- Windows provides OpenGL32.dll and some headers
- It's really old and has been around for a long time
- Does two things:
  - Exposes a really old OpenGL 1.1 API directly (vs 4.6)
  - Loads the real up to date driver by checking the registry (eg ATI, Nvidia, etc)
- To **access** all the **updated** functionality from those drivers we need to 1st link to OpenGL32.dll **and** then create function pointers into the loaded drivers
- Without this you are stuck with limited access to the underlying graphics driver/hardware (OpenGL 1.1)

OpenGL loading library

- includes up to date header files that match the OpenGL specification to link against (https://www.opengl.org/registry/)
- includes mechanism to check for function availability (in case the driver and specification are out of sync)
- links pointers to available OpenGL functions at runtime
- saves a ton of manual error prone labour

(We use GLEW, the OpenGL Extension Wrangler)

OpenGL

- So when we say “C++ OpenGL”, we actually mean:
  - “A graphics driver loaded at runtime which implements the OpenGL specification (a document) which can be accessed through the OpenGL API (bunch of C++ headers).”

What can OpenGL do (not)?

- Through OpenGL you can render primitives (triangles etc) on the GPU at an alarming rate (as in **millions** per frame).

- What OpenGL cannot do:
  - Create a window (rendering context)
  - Process keyboard, mouse events
  - Play audio
  - Load images from disk
  - Do 3D math on the CPU
  - Load different 3d formats from disk
  - Create/update/render a scene graph

SFML, (SDL2, GLFW)

GLM (VMATH)

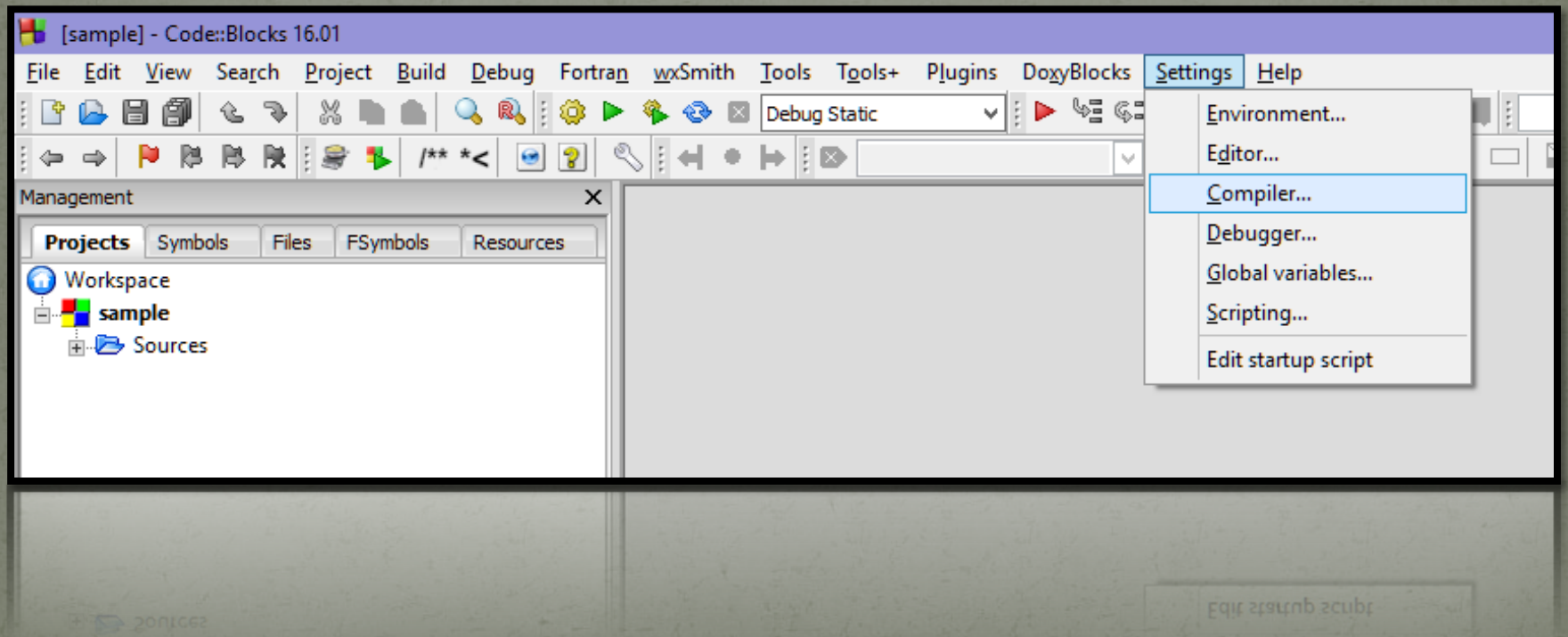
MGE

How do we run OpenGL?

- Download and configure required headers/libs (Either manually or using vcpkg or using bundle from bb)
- Two bundles available on BB: 1 for CodeBlocks, 1 for VS
  - Bundle contains SFML, GLEW, GLM: headers, libs, dlls
- After that setup your project & IDE:
  - Configure all libs/includes/targets manually **OR**
  - Reuse the project files of one of the given examples
- Press run & pray (Debug Static recommended…)
- Try this out after the lecture using the MGE !

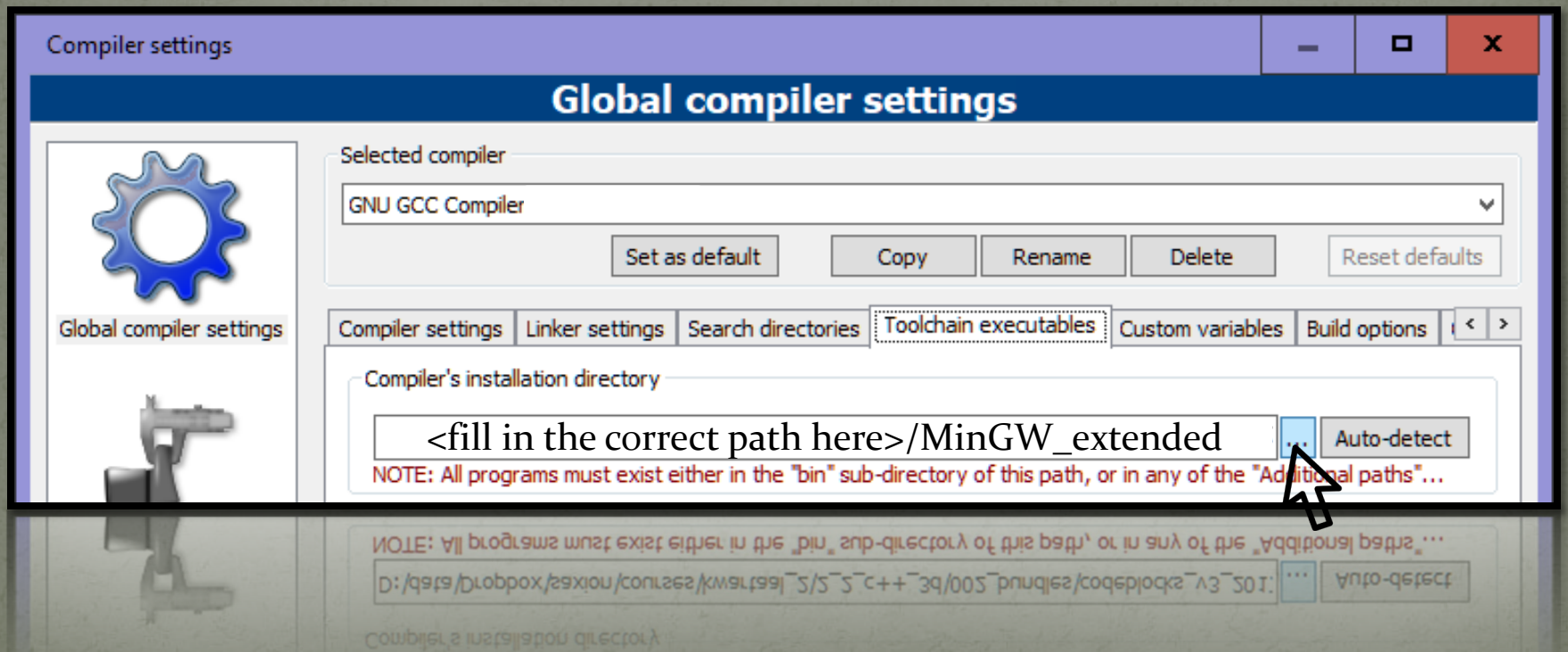
Configuring codeblocks

- Open the compiler settings:



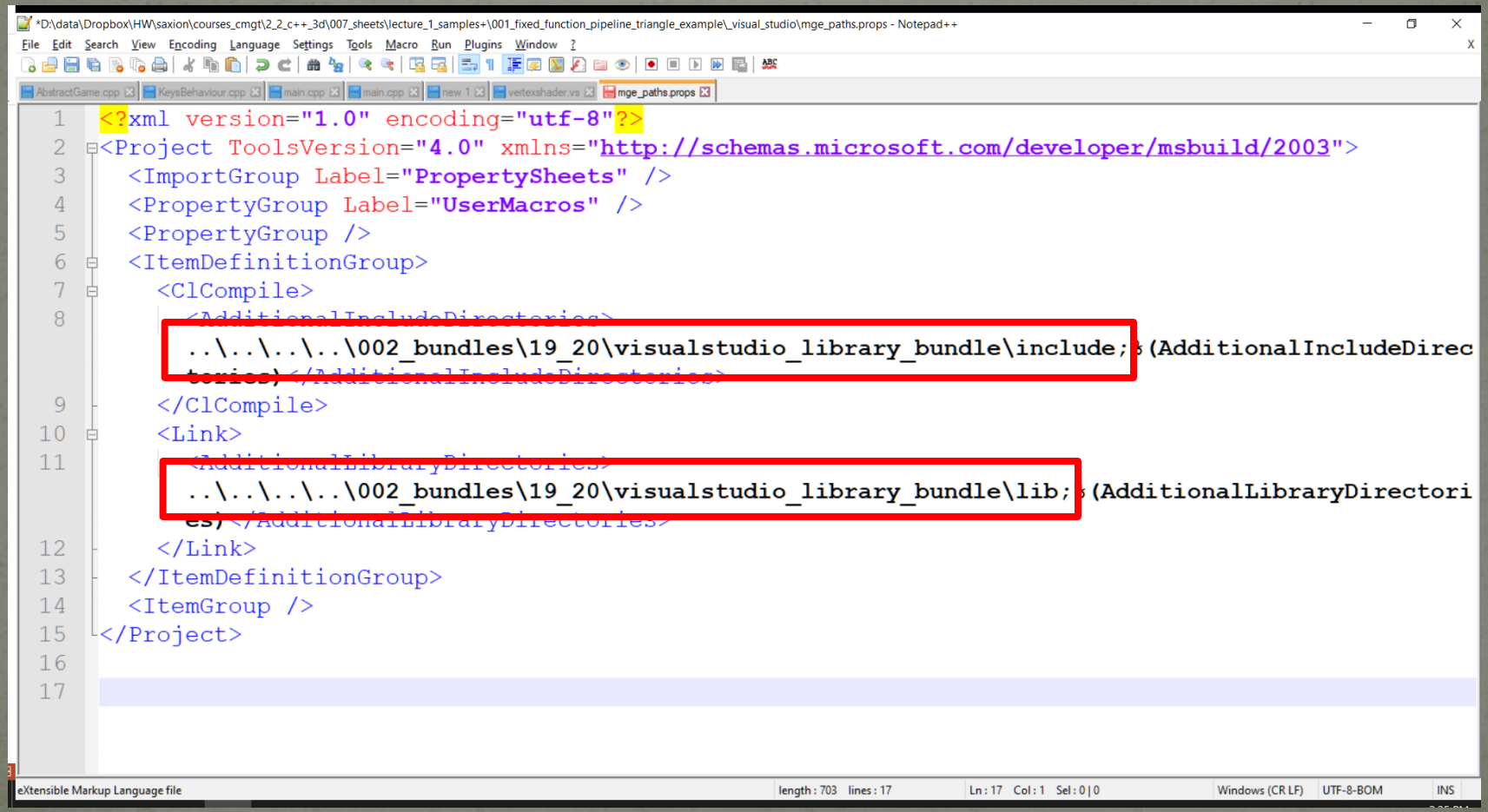
Configuring codeblocks

- Set the toolchain executable path to the MinGW_Extended path:



Configuring Visual Studio

- Update the paths in the .props file:



What you should see…

What does OpenGL code look like?

- Some google-as-I-go results:
  - **gl**Vertex
  - **gl**ClearColor
  - **gl**VertexAttribPointer
  - **GL**_ARRAY_BUFFER
  - **GL**float, **GL**int, **GL**uint
  - Etc…
  - Etc..
- Did you detect a pattern?
- BUT, it's not quite so simple as that, because…

OpenGL has changed a lot over the years

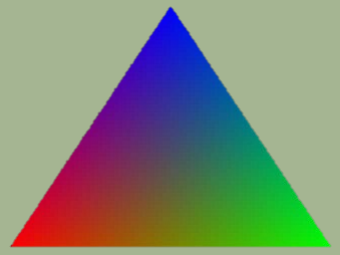
- OpenGL has been around a long time, since 1992
- Improved/updated continuously
- Major updates involved major API changes
- Big parts of the OpenGL API have been deprecated
- Makes learning about OpenGL ~~hell on earth~~ interesting:
  - Stuff is either deprecated and should no longer be used
  - Stuff is too new and not supported by your driver yet
- This course tries to guide you through this landscape:
  - How to recognize and avoid the old stuff
  - How to find out which version of OpenGL supports what
- What are the major changes you have to be aware of?

Basic OpenGL pipeline

IN

vertices
(uvs,
normal,
textures)

OUT



HOW?

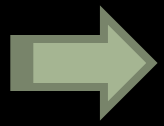
Biggest change in OpenGL history

OpenGL "Pipeline"

Fixed function pipeline

What you can do with OpenGL is fixed in the pipeline:

- set light parameters
- set blend modes
- set a matrix transform



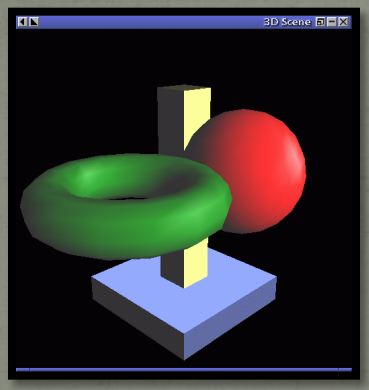
Programmable pipeline

The functions to perform 3d operations are stored in mini programs called Shaders. They are written by you, loaded and compiled on the GPU at runtime.

- Some quick examples to demonstrate the difference:
  - 001_fixed_function_pipeline_triangle_example
  - 004_programmable_pipeline_triangle_example_buffered

Biggest change in OpenGL history

Fixed function pipeline → Programmable pipeline



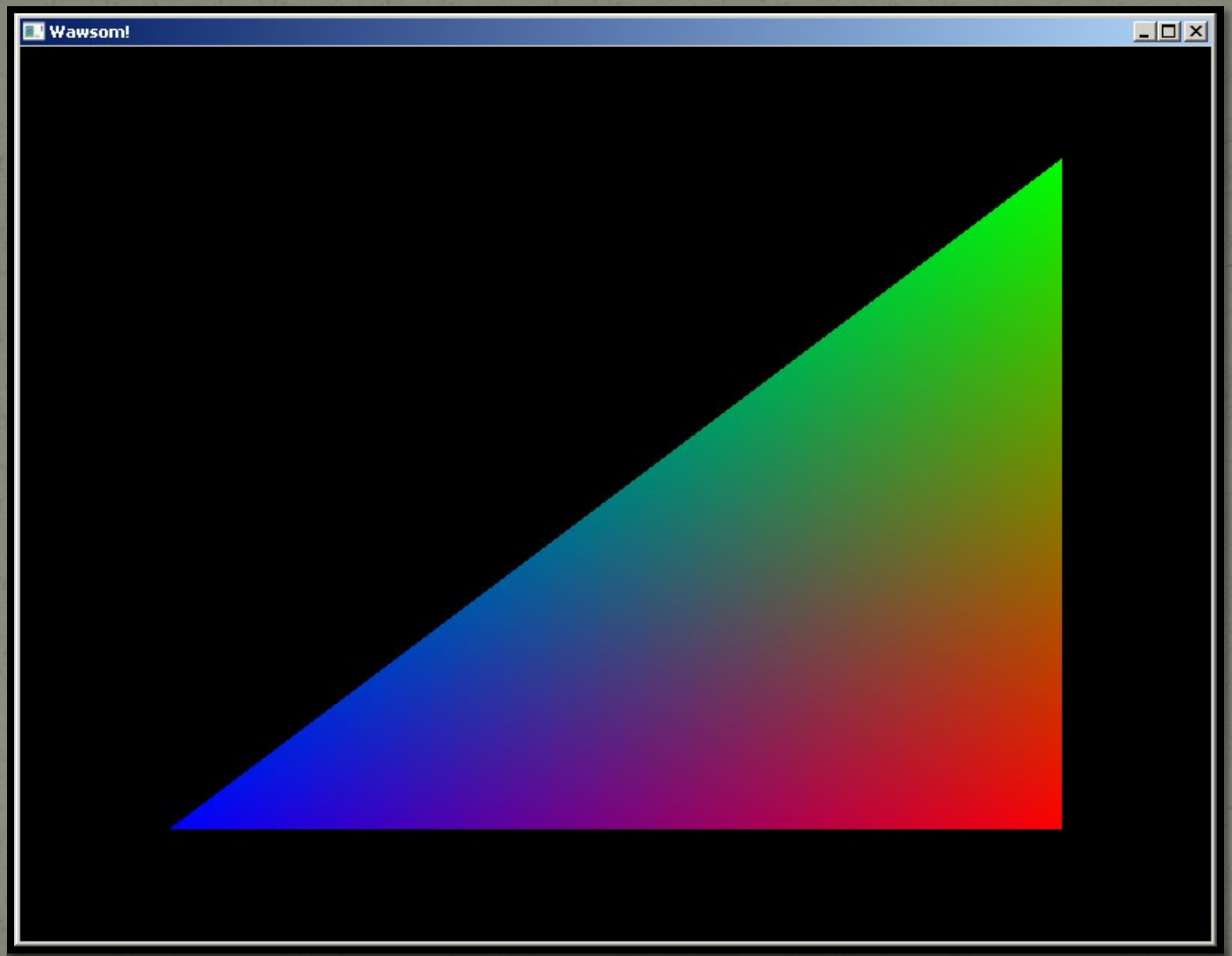
Moral of the story

- If you read things like:
  - glRotatef
  - glMatrixMode
  - glVertex3f
  - glBegin / glEnd
- you are reading the wrong tutorials!
- Instead look for tutorials/documentation with:
  - glCreateShader
  - glVertexAttribPointer
  - glGenBuffers
  - glBindBuffer

Main goal for today

Understanding the basics of modern OpenGL

Vehicle: Rendering a colored triangle



Understanding modern OpenGL

- To render something in OpenGL, we need to:

1. Create an OpenGL context/window
2. Program the OpenGL pipeline using shaders
3. Define data to render
4. Send data into pipeline
5. Render the data
6. Display the results
7. Rinse and repeat

Understanding modern OpenGL

- To render something in OpenGL, we need to:

1. **Create an OpenGL context/window**
2. Program the OpenGL pipeline using shaders
3. Define data to render
4. Send data into pipeline
5. Render the data
6. Display the results
7. Rinse and repeat

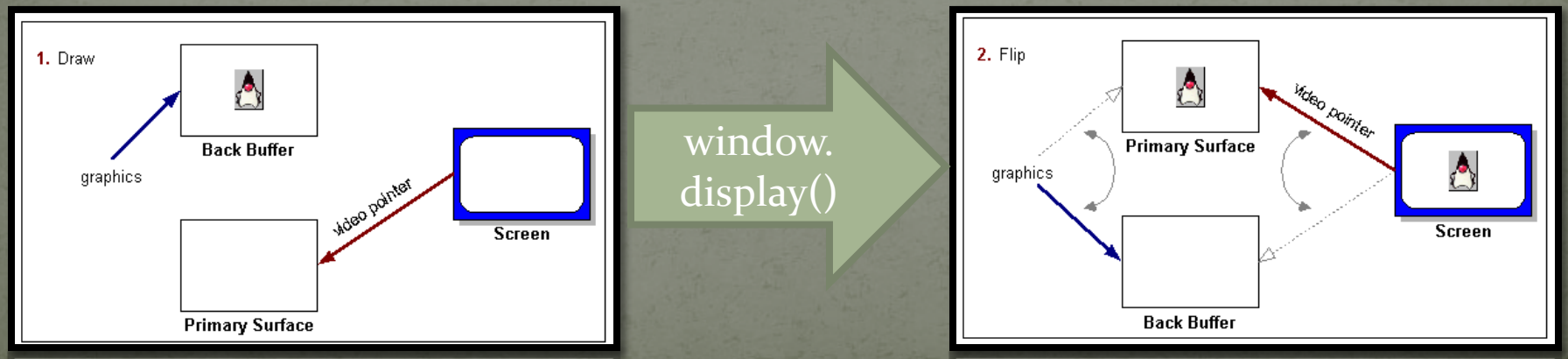
OpenGL context

- What is an OpenGL context?
  - Something we can render into (framebuffer)
  - Something that keeps track of all OpenGL **state info**
  - No context -> No OpenGL
- This means context has to be created outside of OGL:
  - **SFML**
  - GLFW
  - SDL2
- We use SFML for this for (no particular reason)

OpenGL FrameBuffer

- FrameBuffer holds end result of rendering:
  - image data to display on screen
  - **optionally** also a depth buffer to z sort "pixels" correctly (subject of another lecture)

- SFML creates two OpenGL FrameBuffers by default, so that it can use **double buffering**:



1. Setting up OpenGL 3.3 Context

SFML window == a valid FF OpenGL context

```
//Open SFML Window == Valid OpenGL Context
sf::Window window(
    sf::VideoMode (800, 600),        //800 x 600 x 32 bits pp
    "OpenGL Window",                 //title
    sf::Style::Default,              //Default, Fullscreen, Titlebar, etc
    sf::ContextSettings (
        24,                          //24 bits depth buffer, important for 3D!
        0,                           //no stencil buffer
        0,                           //no anti aliasing
        3,                           //requested major OpenGL version
        3                            //requested minor OpenGL version
    )
);

window.setVerticalSyncEnabled(true);
```

Initializing all extensions

glewInit == a valid PP OpenGL context

```
//initialize glew to load all available opengl functions/extensions
GLenum glewResult = glewInit();
if (glewResult != GLEW_OK) {
    std::cout << "Could not initialize GLEW, byeeee!" << std::endl;
    return -1;
}
```

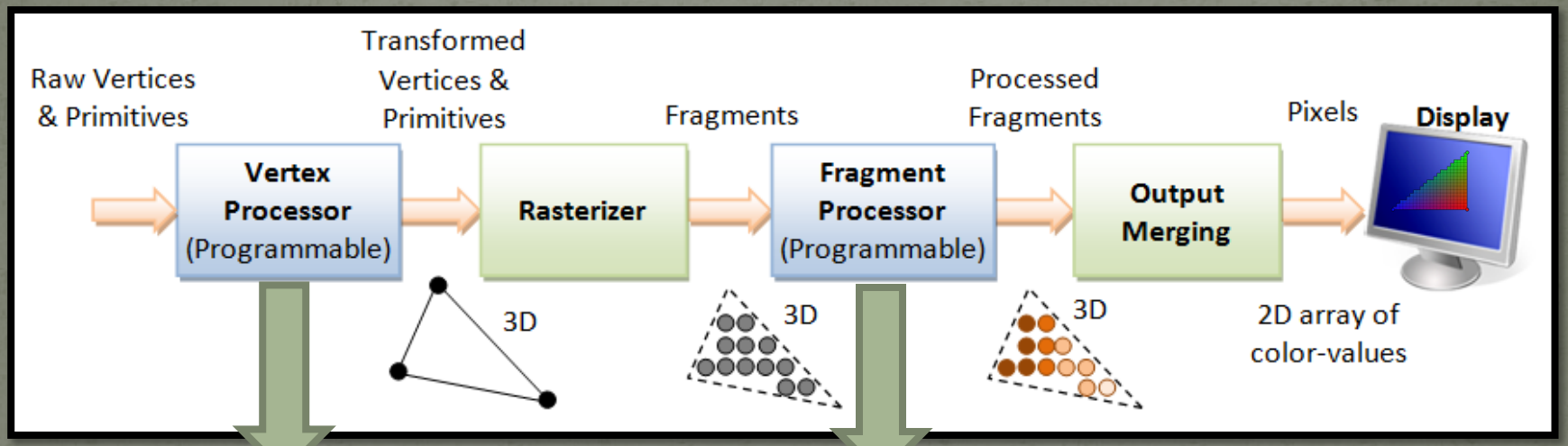
```
}
```

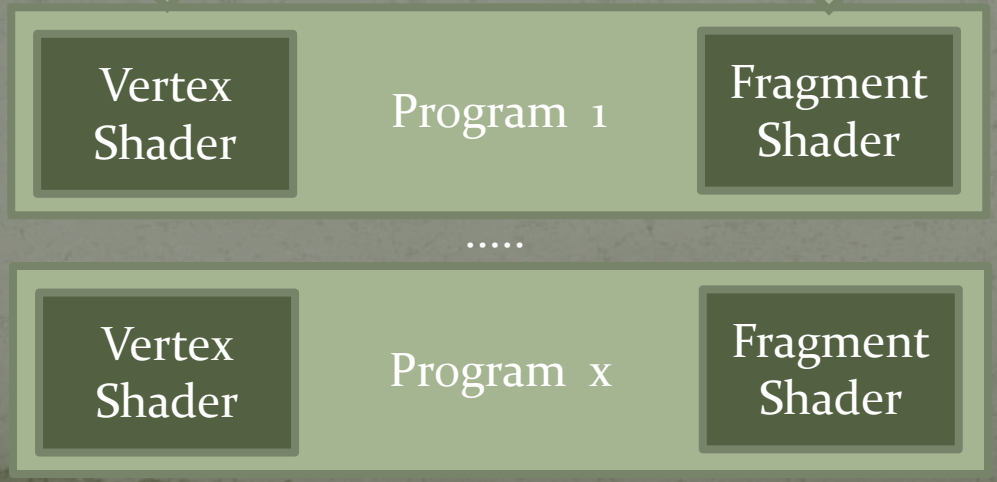
Understanding modern OpenGL

- To render something in OpenGL, we need to:

1. Create an OpenGL context/window
2. **Program the OpenGL pipeline using shaders**
3. Define data to render
4. Send data into pipeline
5. Render the data
6. Display the results
7. Rinse and repeat

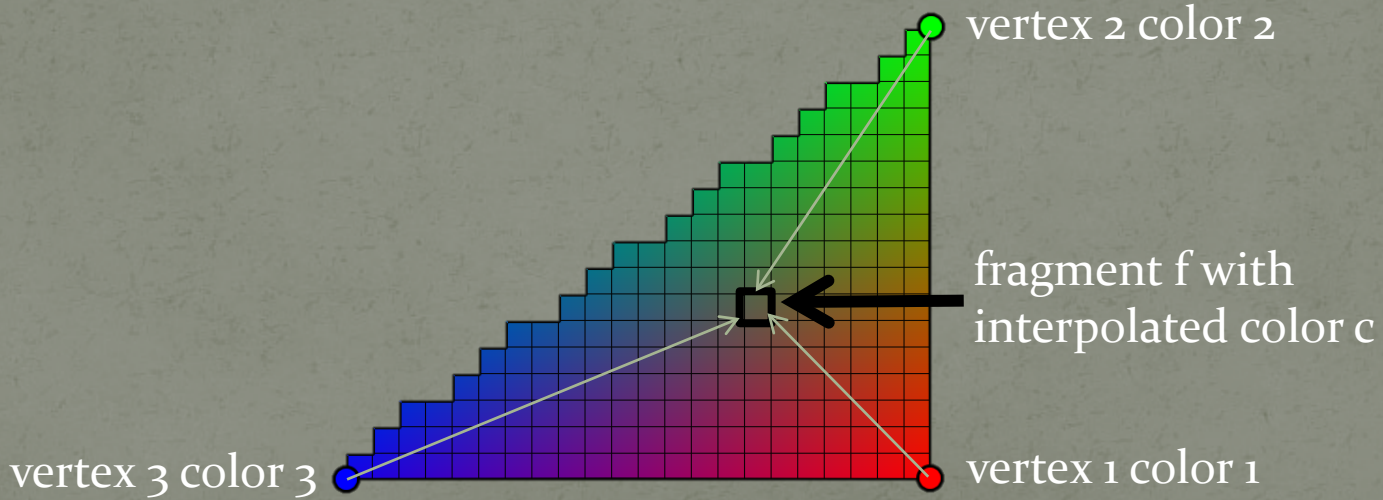
Understanding the OpenGL Pipeline





Vertex vs Fragment shader

- Vertex shader → runs 1x/vertex = 3 times
- Fragment shader → runs 1x/fragment = ? times
  - OpenGL automatically interpolates values to make up for missing data such as colors:



Defining the pipeline: Writing, loading & compiling shaders

Shaders are written in GLSL

GLSL = OpenGL Shading Language, **C-like** language

Vertex Shader in GLSL

```
#version 330

in vec3 vertex;
in vec3 color;

out vec3 fColor;

void main (void) {
    gl_Position = vec4(vertex,1);
    fColor = color;
}
```

Fragment Shader in GLSL

```
#version 330

in vec3 fColor;
out vec4 sColor;

void main (void) {
    sColor = vec4(fColor,1);
}
```

Set up Codeblocks for syntax highlighting:

Settings > Editor > Syntax Highlighting > OpenGL > Filemasks

GLSL Data Types

- Scalar types:
  - float, int, bool
- Vector types:
  - vec2, vec3, vec4, ivec2, ivec3, ivec4, bvec2, bvec3, bvec4
- Matrix types:
  - mat2, mat3, mat4
- Texture sampling:
  - sampler1D, sampler2D, sampler3D, samplerCube
- C++ Style Constructors:
  - `vec3 a = vec3(1.0, 2.0, 3.0);`

Casting and conversion

- GLSL is very flexible when it comes to conversions:
  - Vec4 v1 = vec4 (vec3(1,1,1),1);
  - Vec3 v2 = vec3 (vec4(1,1,1,1));
- GLM mimics this behavior as much as possible

Operators

- Standard C/C++ arithmetic and logic operators
- Operators overloaded for matrix and vector operations:

```
mat4 myMatrix;
vec4 myVecA, myVecB, myVecC;
myVecB = myMatrix*myVecA;
myVecC = myVecA*myVecB;
```

Components and Swizzling

- Vectors can use:
  - [ ], xyzw, rgba
- For example for vec3 v:
  - v[0], v.x, v.r all refer to the same element
- Vectors allow something called **Swizzling**:
  - vec3 a, b;
  - a.rgba = b.grba; //switches r and g channels

Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for

Functions

- Built in
  - Arithmetic: sqrt, power, abs
  - Trigonometric: sin, asin
  - Graphical: length, reflect
- User defined

Built-in Variables

- gl_Position: output position from vertex shader
- gl_FragCoord: input position in fragment shader
- ~~gl_FragColor: output color from fragment shader~~
  - Deprecated! Just declare a custom vec4 and assign it!
    - For example out vec4 sColor (for screen color)

Variable Qualifiers

- in, out
  - copy values to/between shaders
    - in vec3 position;
    - out vec4 color;
  - in / out values are interpolated where needed
  - For **per vertex/per fragment** attributes
- uniform
  - copy values to shaders:
    - uniform float time;
    - uniform vec4 rotation;
  - For **per renderloop** attributes

So what do these shaders do??

Vertex Shader in GLSL

```
#version 330

in vec3 vertex;
in vec3 color;

out vec3 fColor;

void main (void) {
    gl_Position = vec4(vertex,1);
    fColor = color;
}
```

Fragment Shader in GLSL

```
#version 330

in vec3 fColor;
out vec4 sColor;

void main (void) {
    sColor = vec4(fColor,1);
}
```

Getting your Shaders into OpenGL

- Separate shader files need to be **compiled and linked** to form an executable shader program
- The OpenGL **driver** provides the compiler and linker
- A shader program **must** contain vertex and fragment shaders, other shaders are optional

```
const GLchar* vertexShaderSource = //shader source either inline or read from file
GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
glShaderSource (vertexShaderID, 1, &vertexShaderSource, NULL);
glCompileShader (vertexShaderID);

const GLchar* fragmentShaderSource = //shader source either inline or read from file
GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource (fragmentShaderID, 1, &fragmentShaderSource, NULL);
glCompileShader (fragmentShaderID);
```

OpenGL as a State Machine

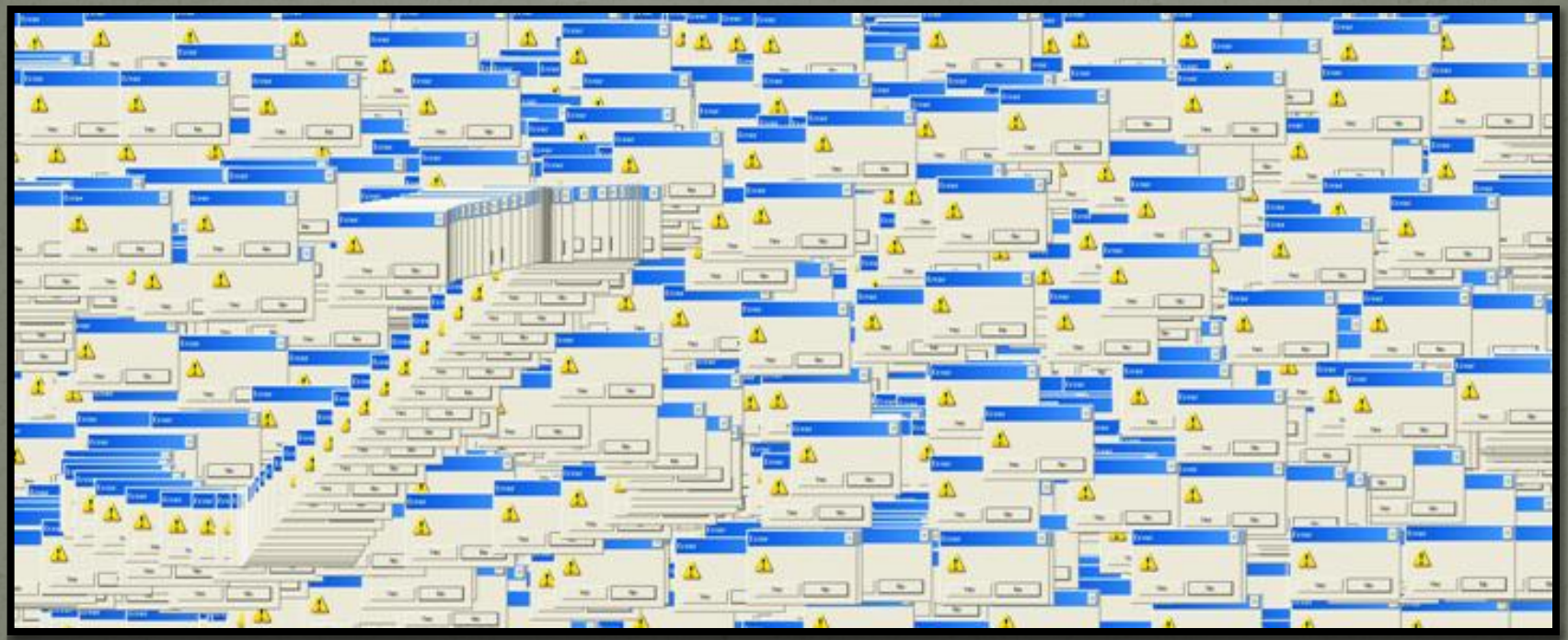
Everything in OpenGL works with handles / states:

Your Program		Your GPU
x = glCreateShader(GL_VERTEX_SHADER)	===>	VERTEX SHADER CREATED, ID = a
(x now has value a)	<===	RETURN ID;
glShaderSource (x, ..., source, ...)	===>	SELECT VERTEX SHADER WITH ID x LOAD source INTO IT
glCompileShader (x)	===>	SELECT VERTEX SHADER WITH ID x COMPILE IT

Linking the shaders into a program

```
GLuint programID = glCreateProgram ();
glAttachShader (programID, vertexShaderID);
glAttachShader (programID, fragmentShaderID);
glLinkProgram (programID);
```

Error checking ?



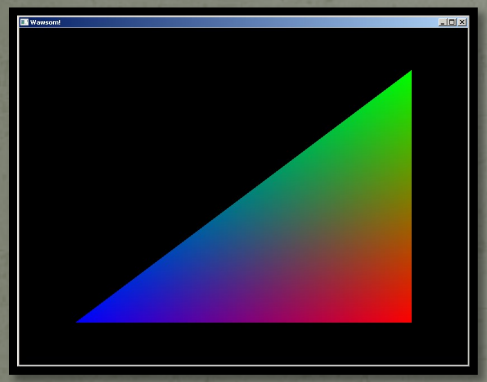
Understanding modern OpenGL

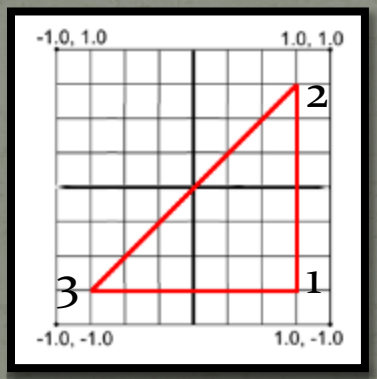
- To render something in OpenGL, we need to:

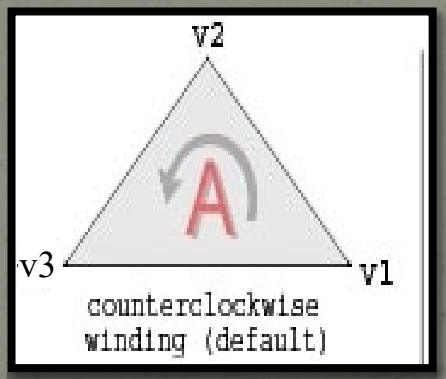
1. Create an OpenGL context/window
2. Program the OpenGL pipeline using shaders
3. **Define data to render**
4. Send data into pipeline
5. Render the data
6. Display the results
7. Rinse and repeat

3. Defining the data

- How do we define the data for our triangle?








```
GLfloat vertices[] = {
    0.75f,  -0.75f, 0,
    0.75f,  0.75f,  0,
    -0.75f, -0.75f, 0
};

GLfloat colors[] = {
    1,0,0,
    0,1,0,
    0,0,1
};
```

- We define our data as two separate arrays:
  - Vertex data:
    - 1 triangle, 3 points, each point an x,y,z value ➔ all in 1 array
    - Simple x,y,z without transformations this means coordinates should be in NDC (between -1 and 1)
  - Color data:
    - 1 triangle, 3 points, each point an r,g,b value ➔ all in 1 array

What we've got so far:

√ data

```
GLfloat vertices[] = {
    0.75f,  -0.75f, 0,
    0.75f,  0.75f,  0,
    -0.75f, -0.75f, 0
};

GLfloat colors[] = {
    1,0,0,
    0,1,0,
    0,0,1
};
```

?

√ vertex shader

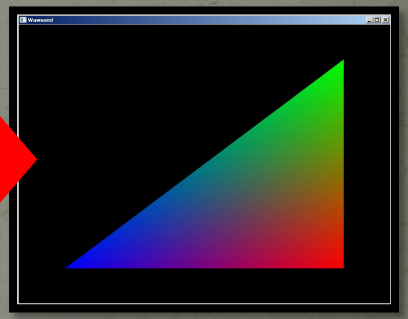
```
#version 330

in vec3 vertex;
in vec3 color;

out vec3 fColor;

void main (void) {
    gl_Position = vec4(vertex,1);
    fColor = color;
}
```

√ fragment shader

```
#version 330

in vec3 fColor;
out vec4 sColor;

void main (void) {
    sColor = vec4(fColor,1);
}
```

√ shader program

?



We've got our data, we've got our shaders, but we are not sending data into the shaders yet, rendering nor displaying things…

Understanding modern OpenGL

- To render something in OpenGL, we need to:

1. Create an OpenGL context/window
2. Program the OpenGL pipeline using shaders
3. Define data to render
4. **Send data into pipeline**
5. **Render the data**
6. **Display the results**
7. Rinse and repeat

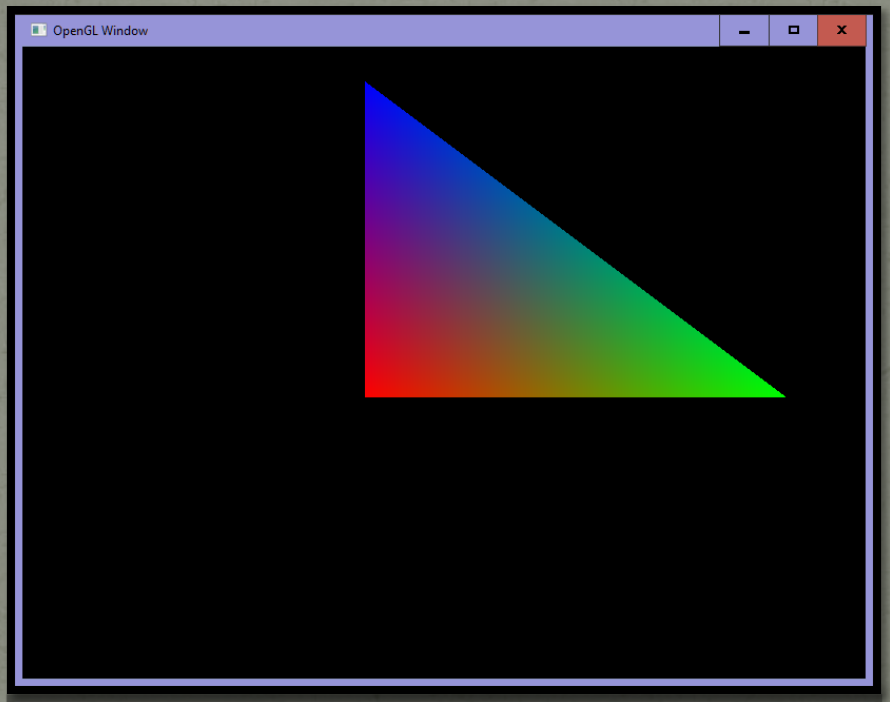
4-6. Rendering the data, easy way (002)

```
glClearColor(0, 0, 0, 1);
while (window.isOpen()) {
    glClear( GL_COLOR_BUFFER_BIT );

    glUseProgram (programID);

    GLint vertexIndex = glGetAttribLocation(programID, "vertex");
    GLint colorIndex = glGetAttribLocation(programID, "color");

    glEnableVertexAttribArray(vertexIndex);
    glEnableVertexAttribArray(colorIndex);

    glVertexAttribPointer(vertexIndex, 3, GL_FLOAT, GL_FALSE, 0, vertices);
    glVertexAttribPointer(colorIndex, 3, GL_FLOAT, GL_FALSE, 0, colors);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray (vertexIndex);
    glDisableVertexAttribArray (colorIndex);

    window.display();

    //... event handling goes here ...
}
```

THE WRONG WAY

Data is resend every frame…

…allows us to for example move the triangle… (003)



but it is also slow…

Buffering the data, the right approach (004)

- To generate a buffer to hold data on the GPU we use:

```
GLuint vertexBufferId;
glGenBuffers (1, &vertexBufferId);
glBindBuffer (GL_ARRAY_BUFFER, vertexBufferId);
glBufferData (GL_ARRAY_BUFFER, sizeof (vertices), vertices, GL_STATIC_DRAW);
glBindBuffer (GL_ARRAY_BUFFER, 0);
```

```
GLuint colorBufferId;
glGenBuffers (1, &colorBufferId);
glBindBuffer (GL_ARRAY_BUFFER, colorBufferId);
glBufferData (GL_ARRAY_BUFFER, sizeof (colors), colors, GL_STATIC_DRAW);
glBindBuffer (GL_ARRAY_BUFFER, 0);
```

- The result is called a VBO (vertex buffer object)
  - Strange name, there is no call with “vbo” in it
  - The buffer can contain other data besides vertices

Rendering the data using VBO's (004)

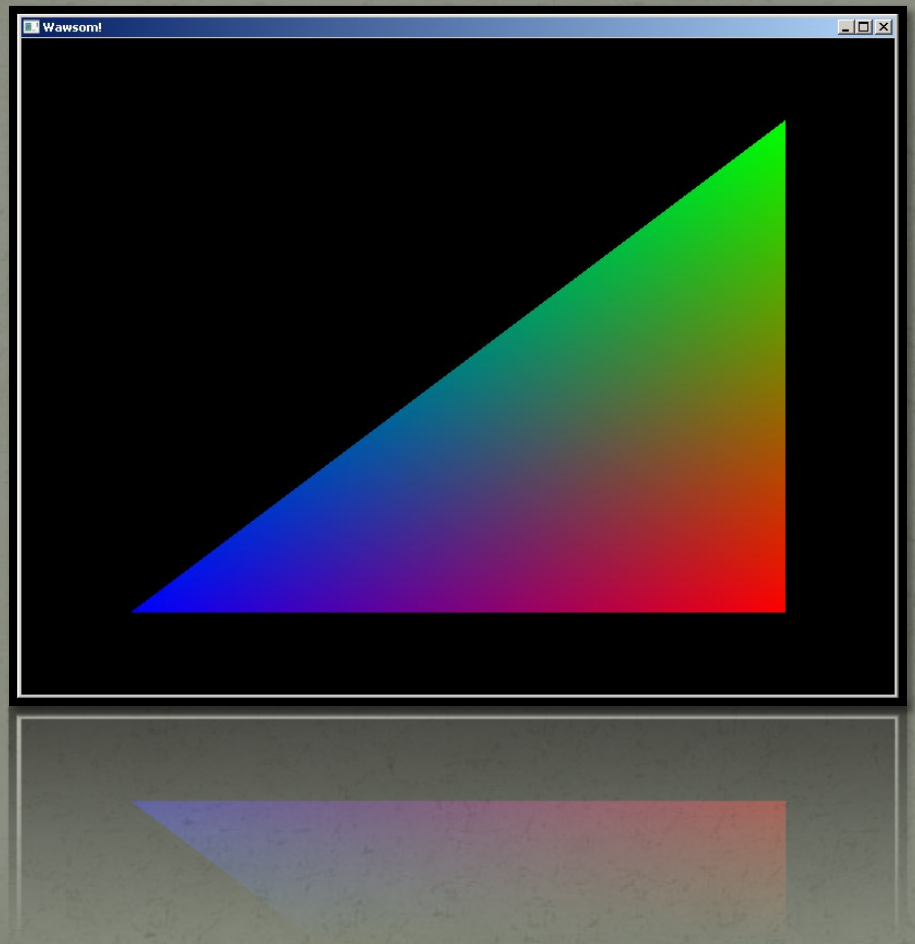
To use the buffers, this:

```
glVertexAttribPointer(vertexIndex, 3, GL_FLOAT, GL_FALSE, 0, vertices);
glVertexAttribPointer(colorIndex, 3, GL_FLOAT, GL_FALSE, 0, colors);
```

has to be changed to this:

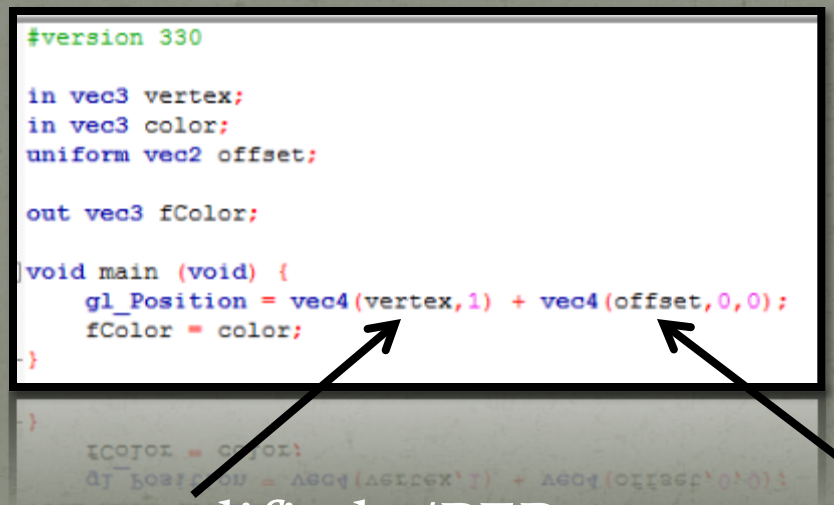
```
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferId);
glVertexAttribPointer(vertexIndex, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

glBindBuffer(GL_ARRAY_BUFFER, colorBufferId);
glVertexAttribPointer(colorIndex, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

Great! Now it's fast, but no longer moves!



Uniforms (005)

- To change the visuals without changing the data, we need to pass in data that can change **per renderloop**, not per vertex, using something called a uniform variable, or just "uniform":



- **in** variables are modified 1/PER vertex
- **uniform** variables are modified 1/PER renderloop

Passing data to Uniforms (005)

```
float elapsedTime = clock.getElapsedTime().asSeconds();
glUniform2f(
    glGetUniformLocation(programID, "offset"),
    0.5f*cos(elapsedTime),
    0.5f*sin(elapsedTime)
);
```

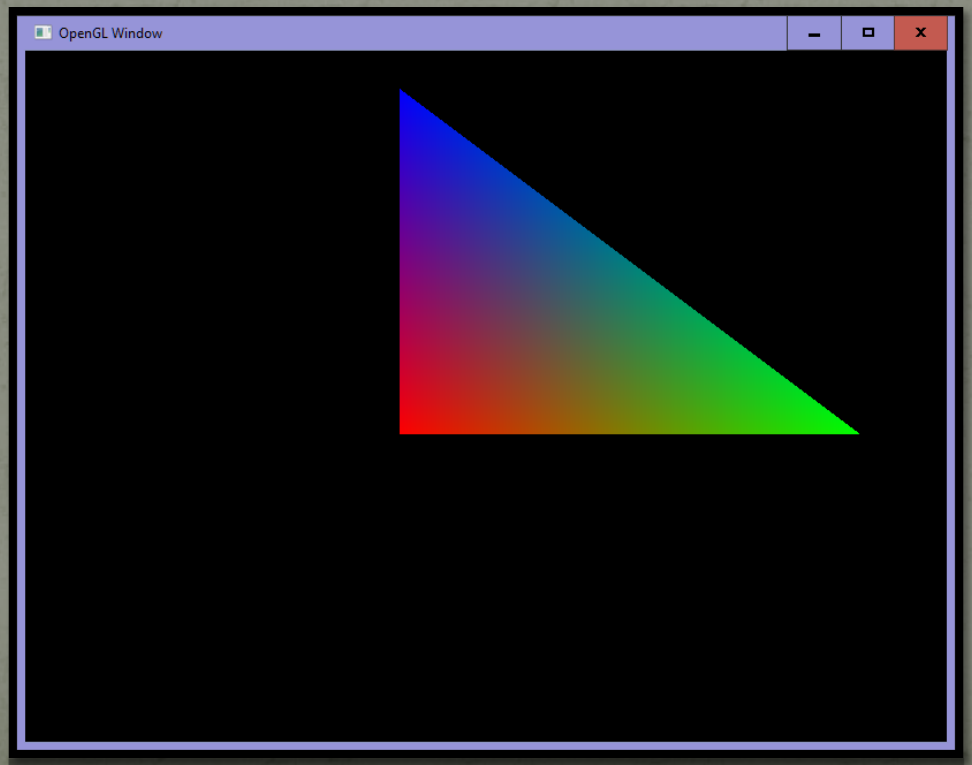
glUniform… has a lot variants

Because the OpenGL API doesn't allow overloading:

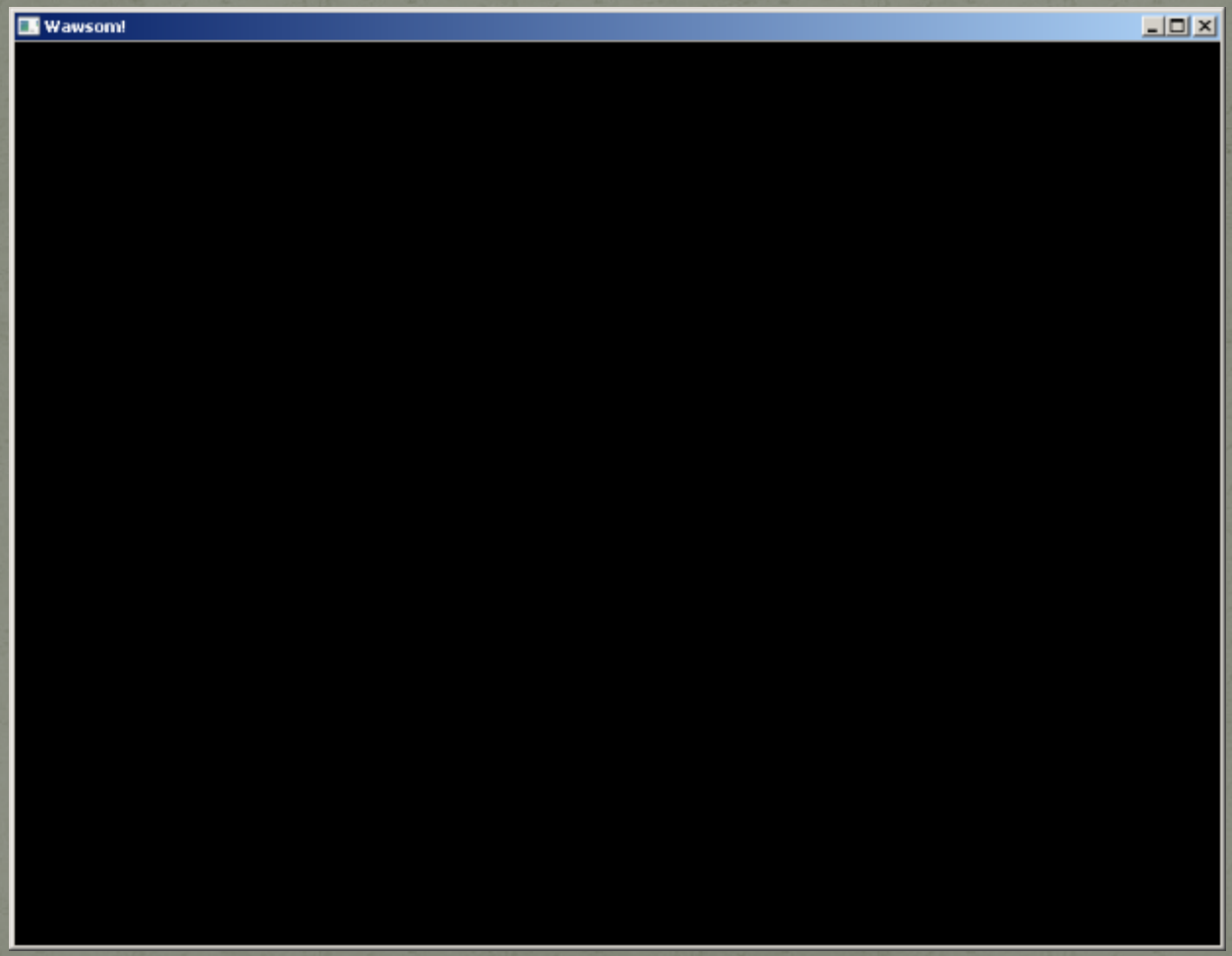
```
glUniform1f(GLint location, GLfloat v0);
glUniform2f(GLint location, GLfloat v0, GLfloat v1);
glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);
glUniform1i(GLint location, GLint v0);
glUniform2i(GLint location, GLint v0, GLint v1);
glUniform3i(GLint location, GLint v0, GLint v1, GLint v2);
glUniform4i(GLint location, GLint v0, GLint v1, GLint v2, GLint v3);
glUniform1ui(GLint location, GLuint v0);
glUniform2ui(GLint location, GLuint v0, GLuint v1);
glUniform3ui(GLint location, GLuint v0, GLuint v1, GLuint v2);
glUniform4ui(GLint location, GLuint v0, GLuint v1, GLuint v2, GLuint v3);
glUniform1fv(GLint location, GLsizei count, const GLfloat *value);
glUniform2fv(GLint location, GLsizei count, const GLfloat *value);
glUniform3fv(GLint location, GLsizei count, const GLfloat *value);
glUniform4fv(GLint location, GLsizei count, const GLfloat *value);
glUniform1iv(GLint location, GLsizei count, const GLint *value);
glUniform2iv(GLint location, GLsizei count, const GLint *value);
glUniform3iv(GLint location, GLsizei count, const GLint *value);
glUniform4iv(GLint location, GLsizei count, const GLint *value);
glUniform1uiv(GLint location, GLsizei count, const GLuint *value);
glUniform2uiv(GLint location, GLsizei count, const GLuint *value);
glUniform3uiv(GLint location, GLsizei count, const GLuint *value);
glUniform4uiv(GLint location, GLsizei count, const GLuint *value);
glUniformMatrix2fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix3fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix2x3fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix3x2fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix2x4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix4x2fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix3x4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
glUniformMatrix4x3fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
```

The End Result

A moving colored triangle whose shaders and data are compiled/buffered on the GPU, with a position that's being updated through a uniform:



How about those pesky errors?



Checking shader for errors

```
//compile and check compilation status
cout << "Compiling shader..." << endl;
glCompileShader (shaderHandle);
GLint compileStatus = 0;
glGetShaderiv (shaderHandle, GL_COMPILE_STATUS, &compileStatus);
if (compileStatus == GL_FALSE) {
    cout << "Compilation failed:" << endl;

    GLint logLength = 0;
    glGetShaderiv (shaderHandle, GL_INFO_LOG_LENGTH, &logLength);
    if (logLength > 0) {
        GLchar* errorLog = new GLchar[logLength];
        glGetShaderInfoLog (shaderHandle, logLength, NULL, errorLog);
        cout << errorLog << endl;
        delete [] errorLog;
    } else {
        cout << "No info available." << endl;
    }
    return 0;
}
```

Checking program for errors

```
cout << "Linking program..." << endl;
glLinkProgram (program);

int linkStatus = 0;
glGetProgramiv (program, GL_LINK_STATUS, &linkStatus);

if (linkStatus == GL_FALSE) {
    cout << "Linking failed... " << endl;

    int linkLogSize = 0;
    glGetProgramiv (program, GL_INFO_LOG_LENGTH, &linkLogSize);

    GLchar* linkLog = new GLchar[linkLogSize];
    glGetProgramInfoLog (program, linkLogSize, NULL, linkLog);
    cout << linkLog << endl;
    delete [] linkLog;
    return 0;

} else {
    cout << "Link successful." << endl;
}
```

Simplifying shader creation (006)

- Refactor shader code into separate ShaderUtil class
- Add error handling to ShaderUtil class
- Load shaders externally
- Refactor example, shader code becomes:

```
int main () {
    sf::Window window(sf::VideoMode (800, 600), "Wawsom!");
    window.setVerticalSyncEnabled( true );

    cout << "Initializing GlEW..." << endl;
    bool glewInitResult = (glewInit() == GLEW_OK);
    cout << "GlEW Initialized:" << glewInitResult  << endl;

    GLuint programID = ShaderUtil::createProgam("vertexshader.vs", "fragmentshader.fs");
```

Summing up

- We've seen:
  - what OpenGL is and why we would use it
  - the major changes in OpenGL history and what to avoid
  - how we setup an OpenGL application
  - how we define and load shaders
  - how shaders define in & output variables and interpolate this data automatically where needed
  - fill buffers with per vertex data to stream to the shaders
  - pass in per render loop variables through uniforms
  - how to improve our code by refactoring

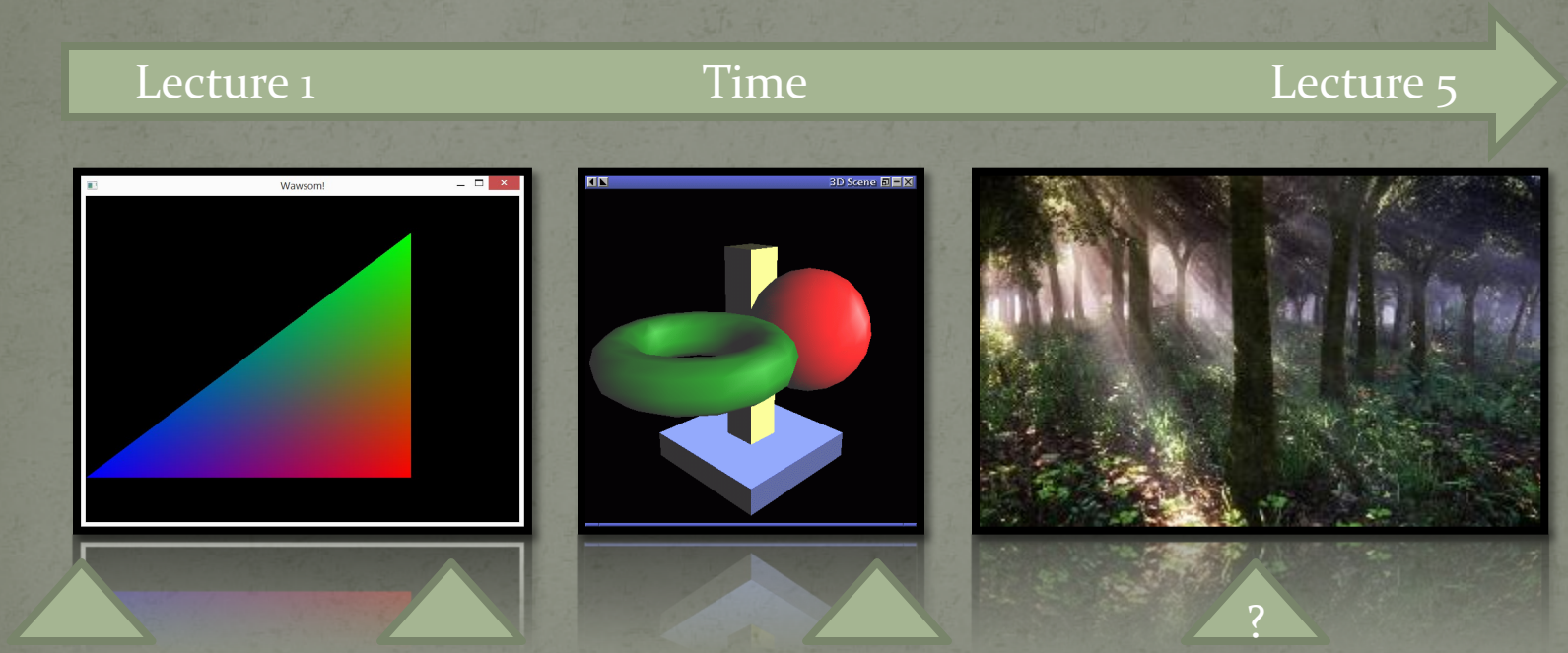
And there is more ...*

- You officially also need to use VAO's (skipped for now)
- Most renderers use indexed drawing (eg Unity)
- Check out learnopengl.com for nitty gritty details

(You can get through this course, without knowing these things)

Things yet to come

- moving into 3D, using textures, adding lighting, etc



Assignment 1

- Download assignment 1 from BlackBoard, in short:
  - Turn triangle into a quad
  - Implement a checkerboard pattern

  - Optionally add:
    - scale/rotate/light effect
    - radial checkerboards
    - multiple shaders
    - waving flags

References

- Short look at blackboard ….
- https://www.khronos.org/registry/OpenGL-Refpages/gl4/
- https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.3.30.pdf
- https://LearnOpenGL.com
- https://docs.gl


THANK YOU FOR LISTENING
ANY QUESTIONS?
imgflip.com