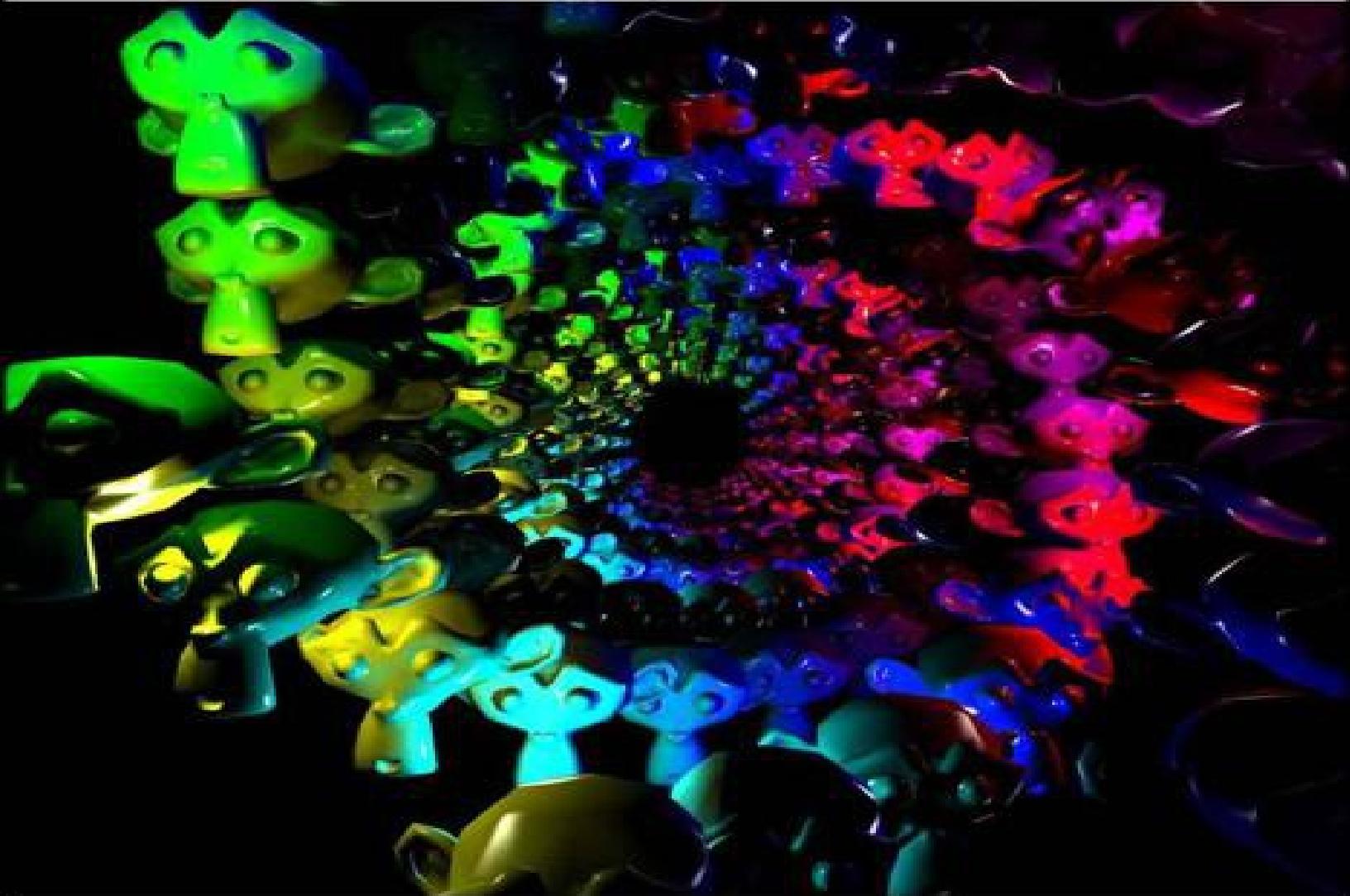


Anton's OpenGL 4 Tutorials



Anton Gerdelen

Table of Contents

Introduction

[Preface](#)

[Downloading and Compiling Demo Source Code](#)

Basics

["Hello Triangle" - OpenGL 4 Up and Running](#)

[Extended Initialisation](#)

[OpenGL 4 Shaders](#)

[Vertex Buffer Objects](#)

Transformation

[Vectors and Matrices](#)

[Virtual Camera](#)

[Quaternion Quick-Start](#)

[Ray-Based Picking](#)

Lighting and Texture Maps

[Phong Lighting](#)

[Texture Maps](#)

Tips and Tricks

[Screen Capture](#)

[Video Capture](#)

[Debugging Shaders](#)

[Gamma Correction](#)

[Extension Checks and the Debug Callback](#)

[Uniform Buffer Objects and Mapping Buffers](#)

Mesh Files

[Importing a Mesh File](#)

More Advanced Lighting and Texture Effects

[Multi-Texturing](#)

[Using Textures for Lighting Coefficients](#)

[Fragment Rejection](#)

[Alpha Blending for Transparency](#)

[Spotlights and Directional Lights](#)

[Distance Fog](#)

[Normal Mapping](#)

[Cube Maps: Sky Boxes and Environment Mapping](#)

New Shader Stages

[Geometry Shaders](#)

[Tessellation Shaders](#)

2d Rendering

[2d GUI Panels](#)

[Sprite Sheets and 2d Animation](#)

[Bitmap Fonts](#)

[Making a Font Atlas Generator Tool](#)

Animation

[Particle Systems](#)

[Hardware Skinning Part 1: Bones](#)

[Hardware Skinning Part 2: Skeleton Hierarchies](#)

[Hardware Skinning Part 3: Key-Frame Animation](#)

Multi-Pass Rendering

[Switching Framebuffer](#)

[Image Processing with a Kernel](#)

[Colour-Based Picking](#)

[Deferred Shading](#)

[Texture Projection Shadows](#)

Discussion

[Building Larger Programmes](#)

[Closing Remarks, Future Techniques, and Further Reading](#)

Anton's OpenGL 4 Tutorials

Preface

The Point of This Book

This book is a practical guide to starting 3d programming with OpenGL, using the most recent version. I started building this material as lab worksheets for the practical part of Stefan Petersson's most excellent 3d Programming course at Blekinge Tekniska Högskola in Sweden, and continued it as an on-line series afterwards. It's now extended into a book format, for off-line viewing, and with additional material. It would suit anyone learning 3d programming that needs a practical guide with some help for common problems. The original material is often used in this way by university courses and hobbyists. This book is a collection of worked-through examples of common real-time rendering techniques as used in video games or student projects. There are also some chapters or short articles for Tips and Tricks - not-so-obvious techniques that can add a lot of value to projects or make it easier to find problems.

This is not a guide for graphics experts, nor is it intended to describe in detail the latest features of OpenGL. It is also not intended to be an instruction to 3d programming theory. A light introduction is given to the topics covered, but there are much more comprehensive and capable sources for that. The idea is to be something like a lab manual - to get you going and over the trickier and more confusing hurdles presented by the API.

In computer lab support for graphics courses I had a couple of key jobs that I found I was repeating over and over for students, and these are my main motivations in the style of writing for this material:

1. To explain convoluted problems in a clear, practical, rational way, that appeals to common sense - not to programming or mathematical experience, and without using a lot of jargon.
2. "***I only get a black screen!***" - To provide a rational, step-by-step process of elimination to debug and diagnose 3d programming problems

so that they don't look so utterly hopeless.

3. And to make a list of the most common problems. I originally started my lab worksheets because I found that I was answering the same questions hundreds of times!

After the first hundred or so hours of that I was able to spot a problem from the other side of the lab, and the students would think I had some sort of super-power. In reality it's just that the same mistakes pop up all the time for people learning. I feel like this is the kind of help that's missing from other OpenGL references - and it is so easy to make mistakes with the OpenGL interface - so I've put all the advice that I can here for your benefit too.

An additional motivation for writing this content was the state of existing reference texts. When I first started teaching there were no OpenGL 4 books published. The larger tomes were not yet updated. At the time of writing some of the major texts have updated to OpenGL 4. You can assemble a collection of texts to educate yourself, which I will list, but these are very large, and very expensive volumes. None of them are satisfactory as a sole reference for a university course, which means you'll need to refer to several of them. This is an absolutely ridiculous expectation for a student budget, and I think very disappointing for education in this area, which has so much potential to grab attention and to motivate. I try to fill this gap here with something easy to get into, with human-digestible instruction steps, and not expensive or too heavy to travel with. It's still worth getting access to a collection of the larger volumes, which will give you precise technical detail and more advanced technical information. Perhaps you can get your library to buy them, or pool together with other enthusiasts.

What Should I Know Before Starting?

Programming Knowledge

Graphics programming is not easy. You will need to have strong C programming skills. You don't need to know C++ for OpenGL. The articles in this book assume that you are familiar with the following concepts:

- Built-in data-types; `int`, `float`, `unsigned int`, etc. and what sort of numbers that they can represent
- Boolean logic
- Functions and parameters
- Arrays, loops, and strings
- Compiling and linking C programmes
- File reading and writing
- Linking external libraries
- Pointers, addresses
- Casting between data types
- Dynamic memory allocation and deallocation
- Identifying different types of errors, and debugging

In other words, about the equivalent of the first 2 courses in a university Computer Science programme. If you're not a C programmer, but have some other programming background then I can suggest Herbert Schildt's "C/C++ Programmer's Reference". It has an excellent break-down of the basic functionality of C and C++. The pocket edition is great. The official "The C Programming Language" by Brian Kerninghan and Dennis Ritchie is in a 101 tutorial format, but it's short, boring, and expensive, although is the definitive guide. One to borrow, perhaps!

It's possible to learn the entirety of C in a short time and keep it all in your working knowledge. This is absolutely not the case with C++. I suggest learning C first. I write my code in C, but occasionally use a feature from C++, where I feel it will save me some time. This means that I use a C++ compiler. You can quite easily replace my C++ bits with the equivalent C

code and use a C compiler.

Mathematical Knowledge

I feel that people over-emphasise mathematics as a pre-requisite for programming. You barely need any maths to be a good programmer - what you need is logic. We do use some very specific mathematical concepts for 3d graphics, but actually learning mathematical concepts when you need to use them is quite possible, and in fact, probably the most effective way to learn. Some concepts that you might want to brush up on before starting:

- Basic trigonometric functions; sine, cosine, tangent.
- 3d vectors. Dot and cross product functions for vectors
- Matrices for affine transformations of vectors

We will be using a lot of the above techniques, and you will probably find it useful to put together a little summary sheet of the techniques that you need a reference for. You can get my cheat sheet at
http://antongerdelan.net/teaching/3dprog1/maths_cheat_sheet.pdf.

Hardware

You're going to need a fairly modern graphics adapter to code in modern OpenGL. Find out what graphics device your computer has, and look up on Wikipedia if it will support OpenGL 3 or 4. Most integrated graphics chips will not be good enough. Some laptops have a dedicated graphics adapter that will work, but most won't. Fairly modern AMD Radeon, Nvidia, and some new Intel adapters will work.

Although this book is aimed at using OpenGL 4, almost all of it will also run on OpenGL 3.2 too - I'll point out any small changes that need to be made. Earlier versions of OpenGL use a different interface, and will not be covered.

You'll also need a calculator and a pencil and paper. Most of the problems that you need to solve will be geometric, and drawing your problem is always a good first step. Solving the first few equations by hand on a calculator, and

comparing it to your drawing is far better than doing what most new students do - jumping in to an IDE right away, and getting frustrated when the debugger won't tell them what to do!

Working Efficiently

Graphics programming is very time consuming. If you're working by yourself, or in a small team, then you really need to avoid getting bogged-down, or it's going to be very frustrating. If I could go back in time and give some advice to myself when I was just starting graphics, this is what I'd say. Some of this is personal preference, and some is universal. More than likely though, you'll have to find out the techniques that work best for you by learning the hard way:

- Solve your problem on paper first. Don't start coding until you feel that you've got a complete concept of the algorithm that you'll be implementing. I can't emphasise this enough. The majority of students don't heed this advice and get very stuck in tangles of trial-and-error code.
- Print a copy of the Quick Reference Card. For every OpenGL function that you use, read through the parameters, and also look it up in the OpenGL reference pages to get a description of how it should be used. You can find these at <http://www.opengl.org/sdk/docs/>.
- For deeper insight you can look up the official OpenGL specification documents. These are found at <http://www.opengl.org/registry/>. The writing style of these documents is very unusual and, in my opinion, a bit confused, as they start with an introduction as if they were an instruction manual for non-experts, but quickly fall away into jargon that will only make sense to real experts. This makes them extraordinarily difficult to read, so I would only suggest using them as a secondary or tertiary reference to double-check small pieces of functionality or syntax. A further issue is that you might read the document for one particular version of OpenGL or GLSL, but find that features had been revised, and are commented on in specification documents from later versions. I believe the best mode of use is that you familiarise with the latest version, and keep your knowledge base up-to-date by skimming

over new versions as they are released. I tend to use them in reverse, that is, to find out the syntax for older versions of OpenGL or GLSL when I port my code backwards to support older machines.

- Avoid software-engineering techniques and design patterns. An uncomfortable secret is that most of the software design ideas that we're forced to learn these days are enormously time-consuming to implement, and don't actually provide any functional benefit at all. If the advocates are telling you that "it will save more time in the long run" - avoid. Be as minimalist as you can, using just enough structure to keep your code tidy, because all these other things are going to cost you time that you really want to be spending on writing graphics logic.
- If you're new to OpenGL do not try to separate everything into little generic functions or classes. Don't bother making wrappers. It's too easy to lose the thread of execution, and hard to see which OpenGL states are enabled. Use long functions, or write everything into `main()` until you're very comfortable with the flow of OpenGL.
- Only add a library to your project if it makes a very large improvement to your code. You will invest time learning and maintaining code for each library, not to mention the amount of fuss that they cause when compiling on new systems.
- I work much more quickly writing code in a plain text editor and compiling on the command line with a Makefile. Navigating the excessive menus of IDEs can be very time-consuming (especially if you have to wait for things to load). I have a little collection of "favourite" building and debugging tools that I run from the command line, and I'm always trying new alternatives. GNU/Linux and Apple have implementations of the Unix command-line tools, which are enormously powerful, and allow you to work much more quickly than navigating menus to do the same tasks, but are a bit esoteric and take a while to master.
- When you start writing shaders, make sure that you print the shader and linking logs to a file. Shaders are very hard to debug, and this is going to make it less of a "black box". We'll get to this when we discuss shaders in detail.
- Only use pointers and dynamic memory when you really need to. Use boring arrays whenever possible, within reason. You'll get far less fatal errors, and possibly have better memory locality, so things may run

more efficiently. People from a Java background are used to allocating instances of objects, and trying to use the same style in C, which has no garbage collection, can get a bit overwhelming.

- Clear your background to grey, not black. This way you get to see something if your triangles are black due to a missing texture or unlit surface.
- Get a decent keyboard and screen. Hunching over a laptop for hours is a really bad idea for your health. You'll type faster on a bigger keyboard, and a bigger, higher-resolution display will show up more of your mistakes and glitches.
- Occasionally try working on different desktops and laptops. Some OpenGL techniques aren't supported consistently across implementations. Some GLSL compilers will quietly ignore your bugs, which really isn't helpful if you intend to distribute your code to other machines. You'll learn which techniques need fall-backs or work-arounds for broader support. Different displays have different colour profiles, so your range of colour used might appear differently.

A combination of common sense and experience is going to make you more and more efficient. Some tools, libraries, and the vast majority of software engineering techniques and design patterns waste more time than they save - you'll eventually find what works best for you. **Keep it simple!**

Downloading and Compiling Demo Source Code

I have done my best to make sure that you do not need to refer to any complete demonstrations of code, nor to use any custom code framework that hides details of the API. You should be able to build all the code that you need, just from referring to explained snippets within the articles. I also think that it's a bad idea for learning when you can flick through a series of prepared demos and come away with the impression that you've mastered each topic, as this is not the same battle as having to start from scratch. Nevertheless, my demonstration videos just don't work in book format, and when you get really stuck, it can be nice to compare to a completed demo. I also wanted to provide something extra as a way of saying thanks for buying my book!

You can browse or download the entire packing of source code from:
https://github.com/capnramses/antons_opengl_tutorials_book.

Compiling with GCC or Clang

I use GCC to compile. Clang is even better, and has the same interface, if you prefer. To install GCC on GNU/Linux machines there is usually a package called something like `build-essentials` in your package manager. On Apple machines install XCode. On Windows download and install the MinGW collection.

All of the demos have Makefiles for the GCC compiler. You will see several; `Makefile.linux32`, `Makefile.linux64`, `Makefile.win32`, `Makefile.osx`. If you have the GNU Compiler Collection installed you can use `make -f Makefile.osx` for Apple, or `make -f Makefile.win32` for Windows. I tried to keep these as simple as possible, so you can open up the Makefile in a text editor and very clearly see what it is compiling and linking.

Compiling with Visual Studio

If you absolutely must use Visual Studio you will find a solution file in the main folder that should work with most versions of the IDE. I have compiled 32-bit versions of all of the libraries that I use. The linker files are in `common\msvc110\`, the include files are in `common\include\`, and the dynamic files are already in the `visual_studio\Debug\` folder.

If You Spot a Mistake

I'll try to update this collection and fix any issues. Please send me an email; [anton at antongerdelan.net](mailto:anton@antongerdelan.net) if you spot a mistake or something is missing. I won't be able to help with the usual problems of linking libraries using the menus in IDEs because I don't use them.

Hello Triangle - OpenGL 4 Up and Running

The idea of this article is to give a brief **overview** of all of the keys parts of an OpenGL 4 programme, without looking at each part in any detail. If you have used another graphics API (or an older version of OpenGL) before, and you want an at-a-glance look at the differences, then this article is for you. If you have never done graphics programming before then this is also a nice way of getting started with something that "works", and can be modified. The following articles will step back and explain each part in more detail.

Install Libraries

Get Main OpenGL Libraries

The main library that you need is **libGL**. There is also a set of utility functions in a library called **libGLU**, but you more than likely won't need it. [OpenGL](#) is a bit weird in that you don't download a library from the website. The [Kronos](#) group that control the specification only provides the interface. The actual implementations are done by video hardware manufacturers or other developers. You need to have a very modern graphics processor to support OpenGL version 4.x, and even so you might be limited to an earlier implementation if the drivers don't exist yet.

To get the latest `libgl` and `libglu`:

- **Windows** - Just upgrade the video drivers (Nvidia, AMD, or Intel).
- **Linux** - By default [MESA](#) is used, but typically it is a bit slow, and is currently only supporting OpenGL 3.0. To get OpenGL 4 support, switch to proprietary hardware drivers (Nvidia, AMD, or Intel) or download from manufacturer website.
- **Mac** - Apple has its own GL implementations. Check if your video card can support OpenGL 3 or 4 first. Wikipedia articles on AMD and Nvidia cards have a good list. If it should, then you can try to upgrade to the newest operating system - this will grab Apple's latest GL libraries. If you can get OpenGL 3.2 or newer (check by running this article's demo code) then you can do most things from OpenGL 4. Unfortunately, older Macs are still stuck with OpenGL 2.1, which is not compatible. In the next sections we will look at using 3.2 instead - this should work; otherwise dual boot or live disc boot to Linux!

Starting the GL Context With GLFW 3 and GLEW

[GLFW](#) is a helper library that will start the OpenGL "context" for us so that it talks to (almost) any operating system in the same way. The context is a

running copy of OpenGL, tied to a window on the operating system.

The compiled libraries that I downloaded from the website didn't work for me, so I built them myself from the source code. You might need to do this.

The documentation tells us that we need to add a `#define` before including the header if we are going to use the dynamic version of the library:

```
#define GLFW_DLL
#include <GLFW/glfw3.h>
```

There's a library called [GLEW](#) that makes sure that we include the latest version of GL, and not a default [ancient] version on the system path. It also handles GL extensions. On Windows if you try to compile without GLEW you will see a list of unrecognised GL functions and constants - that means you're using the '90s Microsoft GL. If you are using MinGW then you'll need to compile the library yourself. Much the same as with GLFW, the binaries on the webpage aren't entirely reliable - rebuild locally for best results.

Include [GLEW](#) before GLFW to use the latest GL libraries:

```
#include <GL/glew.h>
#define GLFW_DLL
#include <GLFW/glfw3.h>
```

Initialisation Code

Okay, we can code up a minimal shell that will start the GL context, print the version, and quit. You might like to compile and run this in a terminal to make sure that everything is working so far, and that your video drivers can support OpenGL 4.

```
#include <GL/glew.h> // include GLEW and new version of GL on Windows
#include <GLFW/glfw3.h> // GLFW helper library
#include <stdio.h>

int main () {
    // start GL context and O/S window using the GLFW helper library
    if (!glfwInit ()) {
        fprintf (stderr, "ERROR: could not start GLFW3\n");
        return 1;
    }
    // uncomment these lines if on Apple OS X
    /*glfwWindowHint (GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint (GLFW_CONTEXT_VERSION_MINOR, 2);
    glfwWindowHint (GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint (GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);*/

    GLFWwindow* window = glfwCreateWindow (640, 480, "Hello Triangle", NULL, NULL);
    if (!window) {
        fprintf (stderr, "ERROR: could not open window with GLFW3\n");
        glfwTerminate();
        return 1;
    }
    glfwMakeContextCurrent (window);

    // start GLEW extension handler
    glewExperimental = GL_TRUE;
    glewInit ();

    // get version info
    const GLubyte* renderer = glGetString (GL_RENDERER);
    const GLubyte* version = glGetString (GL_VERSION);
    printf ("Renderer: %s\n", renderer);
    printf ("OpenGL version supported %s\n", version);

    // tell GL to only draw onto a pixel if the shape is closer to the viewer
    glEnable (GL_DEPTH_TEST); // enable depth-testing
    glDepthFunc (GL_LESS); // depth-testing interprets a smaller value as "closer"

    /* OTHER STUFF GOES HERE NEXT */

    // close GL context and any other GLFW resources
    glfwTerminate();
    return 0;
}
```

There are 4 lines in the above code that you should uncomment if you're on Apple OS X. We will talk about this in the next tutorial. The short explanation is that it will get the newest available version of OpenGL on Apple, which will be 4.1 or 3.3 on Mavericks, and 3.2 on pre-Mavericks systems. On other systems it will tend to pick 3.2 instead of the newest version, which is unhelpful. To improve support for newer OpenGL releases, we can put the flag `glewExperimental = GL_TRUE;` before starting GLEW.

Make sure that your library paths to GLFW and GLEW are correct. I unzipped GLEW and GLFW into the same folder as my project. You may need to adjust linking and include paths, depending on where you put the files.

Compiling on Windows with MinGW

On MinGW I compile with this line:

```
g++ -o demo.exe main.cpp libglew32.dll.a glfw3dll.a -I include -lOpenGL32 -L ./ -lglew32 -lglfw3 -lm
```

Here I compile with the dynamic (.dll) versions of GLFW and GLEW. These projects have "dll.a" stub libraries that need to be compiled statically too. Note that `libGL` is called `OpenGL32` on all versions of windows (even 64-bit). I linked the C maths library at the end - you may or may not need to do this.

Compiling on Windows with Visual Studio

Make sure that you download the correct version of each library for your build target; 32 or 64 bit, and debug or release. Follow the instructions that come with each library to add the required files to the linking path and include path. Make sure that the .dll files are in the correct run-time folder - you may need to navigate the project settings to find or change this.

Compiling on Linux with G++

I prefer to static compile on Linux. In this case you need to link to additional

dynamic system libraries. These are listed in the documentation for GLEW and GLFW:

```
g++ -o demo main.cpp libGLEW.a libglfw3.a -I include -lGL -lX11 -lXxf86vm -lXrandr -lpthread -lXi -lm
```

Compiling on OS X with G++ or Clang

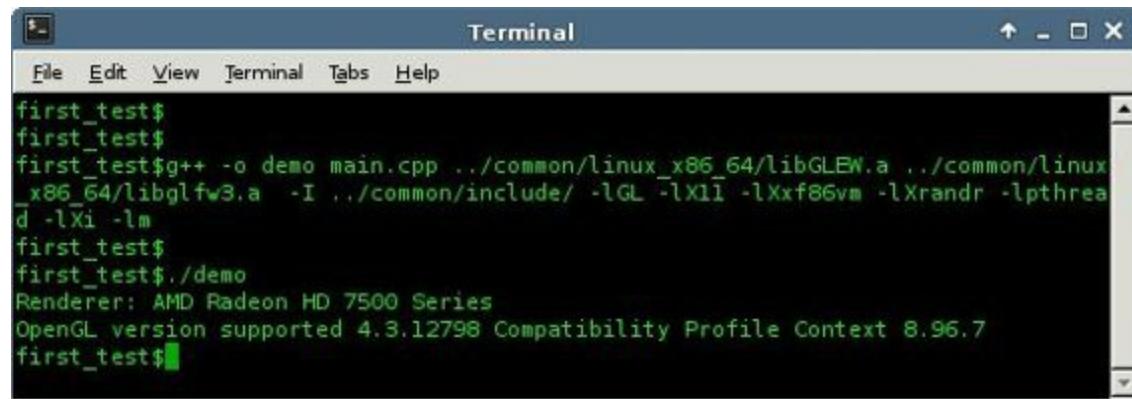
On Apple you would use the bundled development frameworks for system libraries:

```
g++ -framework Cocoa -framework OpenGL -framework IOKit -o demo main.cpp -I include -I/sw/include -I/usr/local/include libGLEW.a libglfw3.a
```

Of course, you can compile this as pure C with GCC or another C compiler as well. I tend to sneak in a few C++ short-cuts as a personal preference; I use what they call "C++--". On Linux you'll most likely install via repositories, and then you won't need the `-I` or `-L` path bits as they'll be on the system paths.

If you're having trouble linking the libraries, then I suggest building the samples that come with the libraries first, and having a read through their install instructions for your operating system.

If I compile and run I get this output:



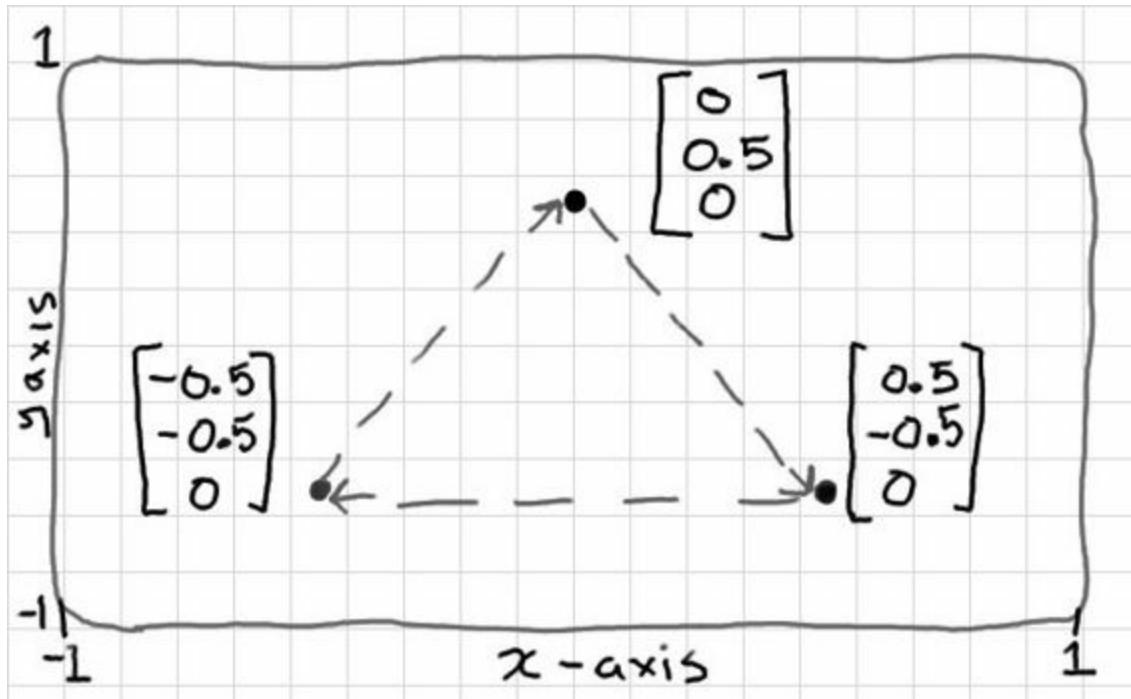
The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal itself contains the following text:

```
first_test$  
first_test$  
first_test$ g++ -o demo main.cpp ./common/linux_x86_64/libGLEW.a ./common/linux_x86_64/libglfw3.a -I ./common/include/ -lGL -lX11 -lXxf86vm -lXrandr -lpthread -lXi -lm  
first_test$  
first_test$ ./demo  
Renderer: AMD Radeon HD 7500 Series  
OpenGL version supported 4.3.12798 Compatibility Profile Context 8.96.7  
first_test$
```

This tells me that my Linux AMD driver can run up to OpenGL version 4.3

Define a Triangle in a Vertex Buffer

Okay, let's define a triangle from 3 points. Later, we can look at doing transformations and perspective, but for now let's draw it flat onto the final screen area; x between -1 and 1, y between -1 and 1, and z = 0.



It always helps to draw your problem on paper first. Here I want to define a triangle, with the points given in clock-wise order, that fits into the screen area of -1:1 on x and y axes.

We will pack all of these points into a big array of floating-point numbers; 9 in total. We will start with the top point, and proceed clock-wise in order: xyzxyzxyz. The order should always be in the same winding direction, so that we can later determine which side is the front, and which side is the back. We can start writing this under the `/* OTHER STUFF GOES HERE NEXT */` comment, from above.

```
GLfloat points[] = {
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f
};
```

We will copy this chunk of memory onto the graphics card in a unit called a

vertex buffer object (VBO). To do this we "generate" an empty buffer, set it as the current buffer in OpenGL's state machine by "binding", then copy the points into the currently bound buffer:

```
GLuint vbo = 0;
glGenBuffers (1, &vbo);
 glBindBuffer (GL_ARRAY_BUFFER, vbo);
 glBufferData (GL_ARRAY_BUFFER, sizeof (points), points, GL_STATIC_DRAW);
```

The last line tells GL that the buffer is the size of 9 floating point numbers, and gives it the address of the first value.

Note that OpenGL has its own custom versions of the built-in data types, hence the `GLuint`. These are usually just `typedefs` of the built-in types, but are guaranteed to be a consistent size on all different implementations. In other words, it's probably a good idea to use them if you want to port your code to lots of devices, but in all likelihood that won't make any difference to you and you can just use your compiler's defaults if you prefer. I try to use them to remind myself when a variable is "belonging to OpenGL", rather than being one of my own identifiers. I wouldn't use them for non-GL data, which I prefer to keep simple and pure if possible. Additionally, it's a bit quicker and cleaner to type `GLuint` than `unsigned integer`, and if you're using pure C then it also provides a boolean type.

Now an unusual step. Most meshes will use a collection of one or more vertex buffer objects to hold vertex points, texture-coordinates, vertex normals, etc. In older GL implementations we would have to bind each one, and define their memory layout, **every time that we draw the mesh**. To simplify that, we have new thing called the **vertex attribute object** (VAO), which remembers all of the vertex buffers that you want to use, and the memory layout of each one. We set up the vertex array object once per mesh. When we want to draw, all we do then is bind the VAO and draw.

```
GLuint vao = 0;
 glGenVertexArrays (1, &vao);
 glBindVertexArray (vao);
 glEnableVertexAttribArray (0);
 glBindBuffer (GL_ARRAY_BUFFER, vbo);
 glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

Here we tell GL to generate a new VAO for us. It returns an unsigned integer to identify it with later. We bind it, to bring it in to focus in the state machine.

This lets us enable the first attribute; 0. We are only using a single vertex buffer, so we know that it will be attribute location 0. The `glVertexAttribPointer` function defines the layout of our first vertex buffer; "0" means define the layout for attribute number 0. "3" means that the variables are `vec3` made from every 3 floats (`GL_FLOAT`) in the buffer.

You might try compiling at this point to make sure that there were no mistakes.

Shaders

We need to use a shader programme, written in OpenGL Shader Language (GLSL), to define **how to draw** our shape from the vertex attribute object. You will see that the attribute pointer from the VAO will match up to our input variables in the shader.

This shader programme is made from the minimum 2 parts; a **vertex shader**, which describes where the 3d points should end up on the display, and a **fragment shader** which colours the surfaces. Both are written in plain text, and look a lot like C programmes. Loading these from plain-text files would be nicer; I just wanted to save a bit of page real-estate by hard-coding them here.

```
const char* vertex_shader =
"#version 400\n"
"in vec3 vp;"
```

```
"void main () {"
```

```
    gl_Position = vec4 (vp, 1.0);"
```

```
"}";
```

The first line says which version of the shading language to use; in this case 4.0.0. **If you're limited to OpenGL 3**, change the first line from "400" to "150"; the version of the shading language compatible with OpenGL 3.2, or "330", for OpenGL 3.3.

My vertex shader has 1 input variable; a `vec3` (vector made from 3 floats), which matches up to our VAO's attribute pointer. This means that each vertex shader gets 3 of the 9 floats from our buffer - therefore 3 vertex shaders will run concurrently; each one positioning 1 of the vertices. The output has a reserved name `gl_Position` and expects a 4d float. You can see that I haven't modified this at all, just added a 1 to the 4th component. The 1 at the end just means "don't calculate any perspective".

```
const char* fragment_shader =
"#version 400\n"
"out vec4 frag_colour;"
```

```
"void main () {"
```

```
    frag_colour = vec4 (0.5, 0.0, 0.5, 1.0);"
```

```
"}";
```

You may need to change the first line of the fragment shader if you are on OpenGL 3.2 or 3.3. My fragment shader will run once per pixel-sized **fragment** that the surface of the shape covers. We still haven't told GL that it will be a triangle (it could be 2 lines). You can guess that for a triangle, we will have lots more fragment shaders running than vertex shaders for this shape. The fragment shader has one job - setting the colour of each fragment. It therefore has 1 output - a 4d vector representing a colour made from red, blue, green, and alpha components - each component has a value between 0 and 1. We aren't using the alpha component. Can you guess what colour this is?

Before using the shaders we have to load the strings into a GL shader, and compile them.

```
GLuint vs = glCreateShader (GL_VERTEX_SHADER);
glShaderSource (vs, 1, &vertex_shader, NULL);
glCompileShader (vs);
GLuint fs = glCreateShader (GL_FRAGMENT_SHADER);
glShaderSource (fs, 1, &fragment_shader, NULL);
glCompileShader (fs);
```

Now, these compiled shaders must be combined into a single, executable GPU shader programme. We create an empty "program", attach the shaders, then link them together.

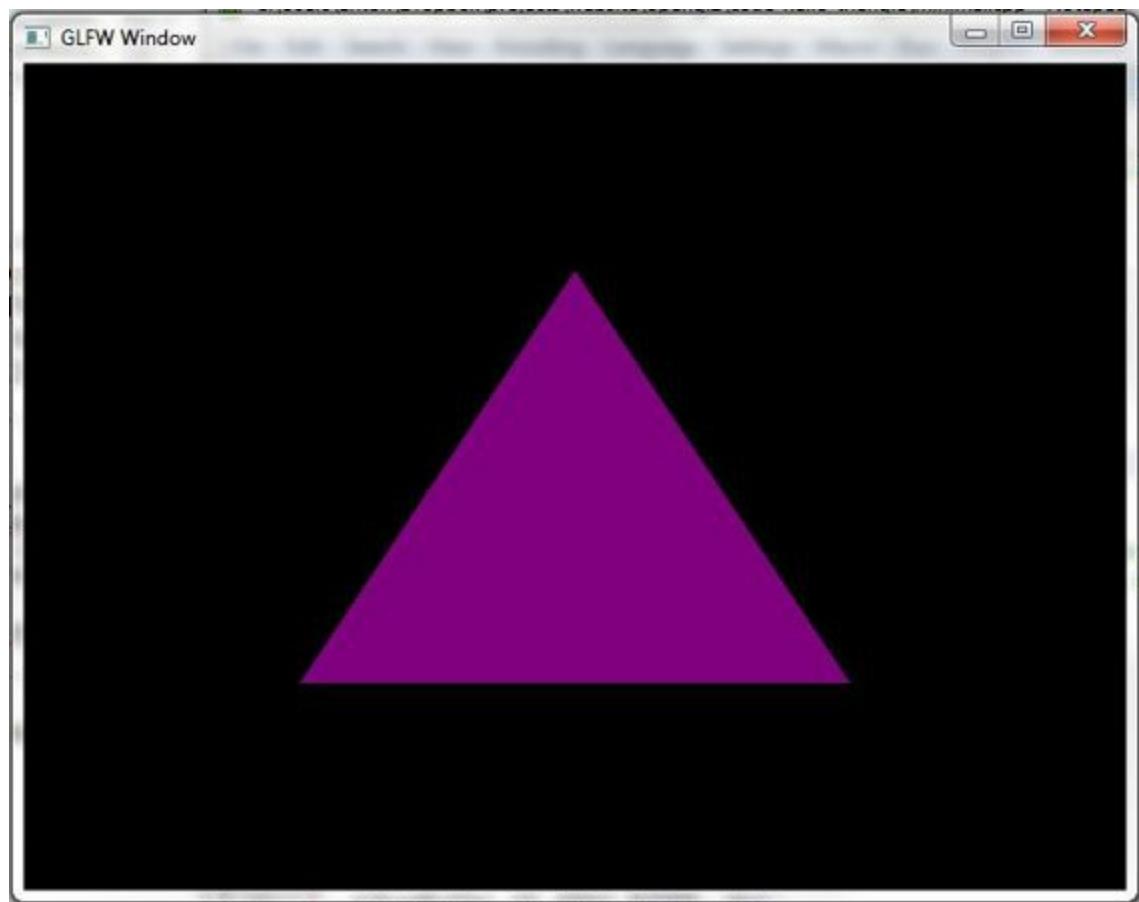
```
GLuint shader_programme = glCreateProgram ();
glAttachShader (shader_programme, fs);
glAttachShader (shader_programme, vs);
glLinkProgram (shader_programme);
```

Drawing

We draw in a loop. Each iteration draws the screen once; a "frame" of rendering. The loop finishes if the window is closed. Later we can also ask GLFW if the escape key has been pressed.

```
while (!glfwWindowShouldClose (window)) {  
    // wipe the drawing surface clear  
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glUseProgram (shader_programme);  
    glBindVertexArray (vao);  
    // draw points 0-3 from the currently bound VAO with current in-use shader  
    glDrawArrays (GL_TRIANGLES, 0, 3);  
    // update other events like input handling  
    glfwPollEvents ();  
    // put the stuff we've been drawing onto the display  
    glfwSwapBuffers (window);  
}
```

First we clear the drawing surface, then set the shader programme that should be "in use" for all further drawing. We set our VAO (not the VBO) as the input variables that should be used for all further drawing (in our case just some vertex points). Then we can draw, and we want to draw in triangles mode (1 triangle for every 3 points), and draw from point number 0, for 3 points. GLFW3 requires that we manually call `glfwPollEvents()` to update things non-graphical events like key-presses. Finally, we flip the swap surface onto the screen, and the screen onto the next drawing surface (we have 2 drawing buffers). Done!



Experimenting

- Load the shader strings from text files called `test.vert` and `test.frag` (a naming convention is handy).
- Change the colour of the triangle in the fragment shader.
- Try to move the shape in the vertex shader e.g. `vec4 (vp.x, vp.y + 1.0, vp.z, 1.0);`
- Try to add another triangle to the list of points and make a square shape. You will have to change several variables when setting up the buffer and drawing the shape. Which variables do you need to keep track of for each triangle? (hint: not much...).
- Try drawing with `GL_LINE_STRIP` or `GL_LINES` or `GL_POINTS` instead of triangles. Does it put the lines where you expect? How big are the points by default?
- Try changing the background colour by using `glClearColor ()` before the rendering loop. Something grey-ish is usually fairly neutral; 0.6f, 0.6f, 0.8f, 1.0f.
- Try creating a second VAO, and drawing 2 shapes (remember to bind the second VAO before drawing again).
- Try creating a second shader programme, and draw the second shape a different colour (remember to "use" the second shader programme before drawing again).

Common Mistakes

GLSL Mistakes

In OpenGL, your mistakes are mostly from mis-using the interface (it's not the most intuitive API ever... to put it kindly). These mistakes often happen in the shaders. In the next article we will look at printing out any mistakes that are found when the shaders compile, and print any problems with matching the vertex shader to the fragment shader found during the linking process. This is going to catch almost all of your errors, so this should be your first port-of-call when diagnosing a problem.

GL Function Parameter Mistakes

You can also easily make small mistakes in the C interface. GL uses a lot of [unsigned] **integers** (aka "`GLuint`") to identify handles to variables i.e. the vertex buffer, the vertex array, the shaders, the shader programme, and so on. GL also uses a lot of enumerated types like `GL_TRIANGLES` which also resolve to **integers**. This means that if you mix these up (by putting function parameters in the wrong order, for example), GL will think that you have given it valid inputs, and won't complain. In other words, **the GL interface is very poor at using strong typing** for picking up errors. These mistakes often result in a black screen, or "no effect", and can be very frustrating. The only way to find them is often to pick through, and check every GL function against its prototype to make sure that you have given it the correct parameters. My most common error of this type is mixing up location numbers with other indices (happens often when setting up textures, for example) - which can be very hard to spot.

GL State Machine Mistakes

Another, very tricky to spot, source of error is knowing which states to set in

the state machine before calling certain functions. I will try to point all of these out as they appear. An example from our code, above, is the `glDrawArrays` function. It will use the **most recently bound** vertex array, and the **most recently used** shader programme to draw. If we want to draw a different set of buffers, then we need to bind that before drawing again. If no valid buffer or shader has been set in the state machine then it will crash.

Next Steps

We will look at initialisation in more detail - particularly logging helpful debugging information. We will discuss the functionality of shaders and the hardware architecture. We will create more complex objects with more than 1 vertex buffer, and look at loading geometry from a file.

Extended Initialisation

So, following on from the minimal start-up last time, it's useful to get some statistics on what the computer's video driver can handle, and specify some additional parameters for start-up. Our GL programme will dump a lot of information, which can be a little infuriating to pick through when there's a problem, so we'll start by setting up a log file for capturing GL-related output.

Starting a Log File

Debugging graphical programmes is a huge pain. They don't have functions to print text to the screen any more, and having a console open on the side printing things out can quickly get overwhelming. I strongly suggest starting a "GL log" straight away, so you can load it up to check out what specifications a user's system has, and also debug any problems after the programme has finished.

The actual structure of your log functions will depend on your preferences. I prefer C `fprintf()` to C++ output streams so I'm going to make something that takes a variable number of arguments like `printf()` does. You might prefer to stream variables out, `cout` style. To make functions that take a variable number of arguments; `#include <stdarg.h>`.

```
#include <time.h>
#include <stdarg.h>
#define GL_LOG_FILE "gl.log"

bool restart_gl_log () {
    FILE* file = fopen (GL_LOG_FILE, "w");
    if (!file) {
        fprintf (
            stderr,
            "ERROR: could not open GL_LOG_FILE log file %s for writing\n",
            GL_LOG_FILE
        );
        return false;
    }
    time_t now = time (NULL);
    char* date = ctime (&now);
    fprintf (file, "GL_LOG_FILE log. local time %s\n", date);
    fclose (file);
    return true;
}
```

This first function just opens the log file and prints the date and time at the top - always handy. It might make sense to print the version number of your code here too. In GCC this would be the built-in strings `_DATE_` and `_TIME_`. Note that after printing to the log file we close it again rather than keep it open.

```
bool gl_log (const char* message, ...) {
    va_list argptr;
```

```

FILE* file = fopen (GL_LOG_FILE, "a");
if (!file) {
    fprintf (
        stderr,
        "ERROR: could not open GL_LOG_FILE %s file for appending\n",
        GL_LOG_FILE
    );
    return false;
}
va_start (argptr, message);
vfprintf (file, message, argptr);
va_end (argptr);
fclose (file);
return true;
}

```

This function is the main log print-out. The "..." parameter is part of C's variable arguments format, and lets us give it any number of parameters, which will be mapped to corresponding string formatting in the `message` string, just like `printf()`. We open the file in "a[ppend]" mode, which means adding a line to the existing end of the file, which is what we want, because we just closed it again since we wrote the time at the top. C has a rather funny-looking start and end function for processing the variable arguments. After writing to the file, we close it again. Why? Because if the programme crashes we don't lose our log - the last appended message can be very enlightening.

```

bool gl_log_err (const char* message, ...) {
    va_list argptr;
    FILE* file = fopen (GL_LOG_FILE, "a");
    if (!file) {
        fprintf (
            stderr,
            "ERROR: could not open GL_LOG_FILE %s file for appending\n",
            GL_LOG_FILE
        );
        return false;
    }
    va_start (argptr, message);
    vfprintf (file, message, argptr);
    va_end (argptr);
    va_start (argptr, message);
    vfprintf (stderr, message, argptr);
    va_end (argptr);
    fclose (file);
    return true;
}

```

I wrote a slight variation of the log function, specifically for error messages. It's the same, but also prints to the `stderr` terminal. I usually run OpenGL with the terminal open. If I print my error message to `stderr` it should pop up as soon as it occurs, which can make it obvious when something has gone

wrong.

Start GLFW Again

Error Checks

We can start GLFW in the same way as before, but add some extra checks. This will tell us if we've made a mistake such as calling a GLFW function with the wrong parameters. Before initialising GLFW, we can set up an error callback, which we can use to spit out some error information, then exit the programme. We create a little function for the callback:

```
void glfw_error_callback (int error, const char* description) {
    gl_log_err ("GLFW ERROR: code %i msg: %s\n", error, description);
}
```

This will tell us if there was a special problem initialising GLFW. I also put in an `assert ()` to make sure that the log file could be opened. If you want to use this then you'll need to include `assert.h`.

```
int main () {
    assert (restart_gl_log ());
    // start GL context and O/S window using the GLFW helper library
    gl_log ("starting GLFW\n%s\n", glfwGetVersionString ());
    // register the error call-back function that we wrote, above
    glfwSetErrorCallback (glfw_error_callback);
    if (!glfwInit ()) {
        fprintf (stderr, "ERROR: could not start GLFW3\n");
        return 1;
    }
    ...
}
```

Setting a Minimum OpenGL Version to Use

Before creating a window with GLFW, we can give it a number of "hints" to set specific window and GL settings. Our primary reason for doing this is to force OpenGL to use at least the minimum version of OpenGL that we are writing our code to support. For example; if we're using tessellation shaders, then we should probably stop the programme from running if the drivers can't support OpenGL 4. If you're using a Mac then this step is necessary - only a limited set of OpenGL implementations are available; 4.1 and 3.3 on

Mavericks, and 3.2 on pre-Mavericks. These are also limited to a "forward-compatible, core profile" context - the most conservative set of features with no backwards-compatibility support for features that have been made obsolete. To request the newest of these three versions, we "hint" to the window-creation process that we want **OpenGL 3.2 forward-compatible core profile**. Yes we can put "3.2", even if we are on Mavericks:

```
glfwWindowHint (GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint (GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint (GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint (GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

This should make us Mac-compatible. In the previous version of GLFW I had to manually hint the number of bits to use for depth and colour channels on Mac, but it looks like GLFW3 has sensible defaults instead. The "forward compatible" profile disables all of the functionality from previous versions of OpenGL that has been flagged for removal in the future. If you get the OpenGL 3.2 or 4.x Quick Reference Card, then this means that all of the functions in blue font are not available. This future-proofs our code, and there's no other option on Mac. The "compatibility" profile doesn't flag any functions as deprecated, so for this to work we also must enable the "core" profile, which does mark deprecation. Check your GL version printout. Mine now says:

```
OpenGL version supported 3.2.11903 Core Profile Forward-Compatible Context
```

You can also use the hinting system to enable things like stereoscopic rendering, if supported by your hardware.

Anti-Aliasing

Whilst we're adding window hints, it's good to know that we can put anti-aliasing hints here too. Even if our textures and colours are nicely filtered, the edges our meshes and triangles are going to look harsh when drawn diagonally on the screen (we'll see pixels along the edges). OpenGL has a built-in "smoothing" ability that blurs over these parts called multi-sample anti-aliasing. The more "samples" or passes it does, the more smoothed it will look, but it gets more expensive. Set it to "16" before taking screen shots!

```
glfwWindowHint (GLFW_SAMPLES, 4);
```

Window Resolution and Full-Screen

To change the resolution, or start in a full-screen window, we can set the parameters of the **glfwCreateWindow** function. To use full-screen display mode, we need to tell it which monitor to use, which is a new feature of GLFW 3.0. You can get quite precise control over what renders on the different monitors, which you can read about at http://www.glfw.org/docs/latest/group__monitor.html. We can just assume that we will use the primary monitor for full-screen mode.

You can ask GLFW to give you a list of supported resolutions and video modes with **glfwGetVideoModes()** which will be useful for supporting a range of machines. For full-screen we can just use the current resolution, and change our `glfwCreateWindow` call:

```
GLFWmonitor* mon = glfwGetPrimaryMonitor ();
const GLFWvidmode* vmode = glfwGetVideoMode (mon);
GLFWwindow* window = glfwCreateWindow (vmode->width, vmode->height, "Extended GL
Init", mon, NULL);
```

Now we can run in full-screen mode! It's a little bit tricky to close the window though - you might want to look at implementing GLFW's keyboard handling to allow an escape key to close the window. Put this at the end of the rendering loop:

```
if (GLFW_PRESS == glfwGetKey (window, GLFW_KEY_ESCAPE)) {
    glfwSetWindowShouldClose (window, 1);
}
```

Remember that our loop ends when the window is told to close. You'll find a list of all the key codes and other input handling commands at http://www.glfw.org/docs/latest/group__input.html.

You'll notice GLFW 3.0 has functions for getting and setting the gamma ramp of the monitor itself, which gives you much more control over the range of colours that are output, which was kind of a pain to do before. This is more of an advanced rendering topic so don't worry about this if you're just starting.

If you're running in a window then you'll want to know when the user resizes the window, or if the system does (for example if the window is too big and needs to be squished to fit the menu bars). You can then adjust all your variables to suit the new size.

```
// keep track of window size for things like the viewport and the mouse cursor
int g_gl_width = 640;
int g_gl_height = 480;

// a call-back function
void glfw_window_size_callback(GLFWwindow* window, int width, int height) {
    g_gl_width = width;
    g_gl_height = height;

    /* update any perspective matrices used here */
}
```

Then we can call: `glfwSetWindowSizeCallback (window, glfw_window_size_callback);`

You'll notice that if you resize your window that the OpenGL part doesn't scale to fit. We need to update the viewport size. Put this in the rendering loop, just after the `glClear ()` function:

```
glViewport (0, 0, g_gl_width, g_gl_height);
```

Printing Parameters from the GL Context

After initialising GLEW we can start to use the GL interface - **glGet()** to print out some more parameters. Most of this information is from previous incarnations of OpenGL, and is no longer useful. Some information is going to be really useful to determine the capabilities of the graphics hardware - how big textures can be, how many textures each shader can use, etc. We can log that here. I called this function right after where I log the GL version being used.

```
void log_gl_params () {
    GLenum params[] = {
        GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS,
        GL_MAX_CUBE_MAP_TEXTURE_SIZE,
        GL_MAX_DRAW_BUFFERS,
        GL_MAX_FRAGMENT_UNIFORM_COMPONENTS,
        GL_MAX_TEXTURE_IMAGE_UNITS,
        GL_MAX_TEXTURE_SIZE,
        GL_MAX_VARYING_FLOATS,
        GL_MAX_VERTEX_ATTRIBS,
        GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS,
        GL_MAX_VERTEX_UNIFORM_COMPONENTS,
        GL_MAX_VIEWPORT_DIMS,
        GL_STEREO,
    };
    const char* names[] = {
        "GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS",
        "GL_MAX_CUBE_MAP_TEXTURE_SIZE",
        "GL_MAX_DRAW_BUFFERS",
        "GL_MAX_FRAGMENT_UNIFORM_COMPONENTS",
        "GL_MAX_TEXTURE_IMAGE_UNITS",
        "GL_MAX_TEXTURE_SIZE",
        "GL_MAX_VARYING_FLOATS",
        "GL_MAX_VERTEX_ATTRIBS",
        "GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS",
        "GL_MAX_VERTEX_UNIFORM_COMPONENTS",
        "GL_MAX_VIEWPORT_DIMS",
        "GL_STEREO",
    };
    gl_log ("GL Context Params:\n");
    char msg[256];
    // integers - only works if the order is 0-10 integer return types
    for (int i = 0; i < 10; i++) {
        int v = 0;
        glGetIntegerv (params[i], &v);
        gl_log ("%s %i", names[i], v);
    }
    // others
    int v[2];
    v[0] = v[1] = 0;
    glGetIntegerv (params[10], v);
    gl_log ("%s %i %i\n", names[10], v[0], v[1]);
}
```

```

    unsigned char s = 0;
    glGetBooleanv (params[11], &s);
    gl_log ("%s %u\n", names[11], (unsigned int)s);
    gl_log ("-----\n");
}

```

Now, if we have a look at the log file after running this (call it after the window creation). My log says:

```

GL_Context Params:
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS 32
GL_MAX_CUBE_MAP_TEXTURE_SIZE 16384
GL_MAX_DRAW_BUFFERS 8
GL_MAX_FRAGMENT_UNIFORM_COMPONENTS 16384
GL_MAX_TEXTURE_IMAGE_UNITS 16
GL_MAX_TEXTURE_SIZE 16384
GL_MAX_VARYING_FLOATS 128
GL_MAX_VERTEX_ATTRIBS 29
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS 16
GL_MAX_VERTEX_UNIFORM_COMPONENTS 16384
GL_MAX_VIEWPORT_DIMS 16384 16384
GL_STEREO 0

```

This tells me that my shader programmes can use 32 different textures each - lots of multi-texturing options with my graphics card here. I can access 16 different textures in the vertex shader, and 16 more in the fragment shader. My laptop can support only 8 textures, so if I want to write programmes that run nicely on both you can see that I would make sure that they don't use more than 8 textures. In theory my texture resolution can be up to 16384x16384, but the actual memory available doesn't fit quite this much at once. I might get away with 8224x8224x8 bits (more than 5GB). The max uniform components means that I can send tonnes and tonnes of floats to each shader. Each matrix is going to use 16 floats, and if we're doing hardware skinning we might want to send a complex skeleton of 256 joints - 4096 floats to the vertex shader. So we can say that we have plenty of space there. Varying floats are those sent from the vertex shader to the fragment shaders. Usually these are 3d vectors, so we can say that we can send around 40 vectors between shaders. Vertex attributes are variables loaded from a mesh e.g. vertex points, texture coordinates, normals, per-vertex colours, etc. GL means 4d vectors here. I would struggle to come up with more than about 6 useful per-vertex attributes, so no problem here. Draw buffers is useful for more advanced effects where we want to split the output from our rendering into different images - we can split this into 8 parts. And, sadly, my video card doesn't support stereo rendering.

Monitoring the GL State Machine

If you look at the list for `glGet` you will see plenty of state queries; "the currently enabled buffer", the "currently active texture slot" etc. OpenGL works on the principle of a **state machine**. This means that once we set a state (like transparency, for example), it is then globally enabled for all future drawing operations, until we change it again. In GL parlance, setting a state is referred to as "**binding**" (for buffers of data), "**enabling**" (for rendering modes), or "**using**" for shader programmes.

The state machine can be very confusing. Lots of errors in OpenGL programmes come from setting a state by accident, forgetting to unset a state, or mixing up the numbering of different OpenGL indices. Some of the most useful state machine variables can be fetched during run-time. You probably don't need to write a function to log all of these states, but keep in mind that, if it all gets a bit confusing, you can check individual states.

Frame Rate Counter

We can add our familiar drawing (clearing) loop, but at the top call a `_update_fps_counter()` function which will update the title bar of our window with the number of times this loop draws per second.

```
while (!glfwWindowShouldClose (window)) {
    _update_fps_counter (window);
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glViewport (0, 0, g_gl_width, g_gl_height);

    /* DRAW STUFF HERE */

    glfwSwapBuffers (window);
    glfwPollEvents ();
    if (GLFW_PRESS == glfwGetKey (window, GLFW_KEY_ESCAPE)) {
        glfwSetWindowShouldClose (window, 1);
    }
}
```

So, GLFW has a function `glfwGetTime` which gives us a double-precision floating point number, containing the number of seconds since GLFW started. A lot of OpenGL tutorials use the (notoriously inaccurate) GLUT microseconds timer, but this one seems to do the trick. I didn't want to have floating global variables keeping track of the previous time, so I made it a static double. Working out the frame rate every frame is too accurate, and will incur a small cost as well. So I average the rate accrued over 0.25 seconds (or thereabouts - I assume it never gets slower than 4 frames per second). This gives me a readable result that is still responsive to changes in the scene. I use the `glfwSetTitle` function to put the rate in the title bar. You may prefer to render this as text on the screen...but we don't have the functionality to do that just yet.

```
void _update_fps_counter (GLFWwindow* window) {
    static double previous_seconds = glfwGetTime ();
    static int frame_count;
    double current_seconds = glfwGetTime ();
    double elapsed_seconds = current_seconds - previous_seconds;
    if (elapsed_seconds > 0.25) {
        previous_seconds = current_seconds;
        double fps = (double)frame_count / elapsed_seconds;
        char tmp[128];
        sprintf (tmp, "opengl @ fps: %.2f", fps);
        glfwSetTitle (window, tmp);
        frame_count = 0;
    }
}
```

```
    frame_count++;
}
```

Keep in mind that the frame rate is not a linear, reliable, measure of how fast your code is. You can't draw faster than the refresh rate of the monitor (around 70Hz or 70fps). Rendering at 100Hz is therefore not beneficial in the same way as it would be to game logic, which can then compute more time steps per second (more detailed movement paths and stuff). Fast GPU clocks will give you huge numbers when drawing nothing. This doesn't really mean anything that you can compare until you start drawing a more involved scene. Measuring frame rate is useful when optimising more complex scenes. If you are drawing at 30fps in a game with fast-moving animations it will be noticeably bad, but it might be okay in a slightly slower-paced game. You can use the fps counter to improve rendering techniques to get it back to whatever your programme's reasonable level is. Remember that frame rate is dependent on your particular hardware configuration - you want to look at frame rate on your "minimum spec" machine. That said, on any machine, it can give you a good idea of which techniques are relatively more GPU hungry than others.

Remember the maxim: **never optimise early**. You can waste all of your programming time trying to get your fps counter to go as high as possible. After this discussion we should now appreciate that this is dumb, because we won't even notice the difference - only put work into improving your frame-rate when it's going way too slowly; otherwise don't bother.

Extending Further

When we look at shaders next, we will log a lot more information. Lots of bugs will come from mixing up uniforms and attributes sent to shaders, so we will dump all of those identifiers to a log as well.

Problem?

Remember to link GL, glfw, and GLEW. My link path looks like this:

```
g++ -o demo main.cpp -lglfw -lGLEW -lGL
```

Remember to initialise GLFW first, then do any parameter setting, then create the window, then start GLEW, then start the drawing loop. Parameter fetching code can go just about anywhere.

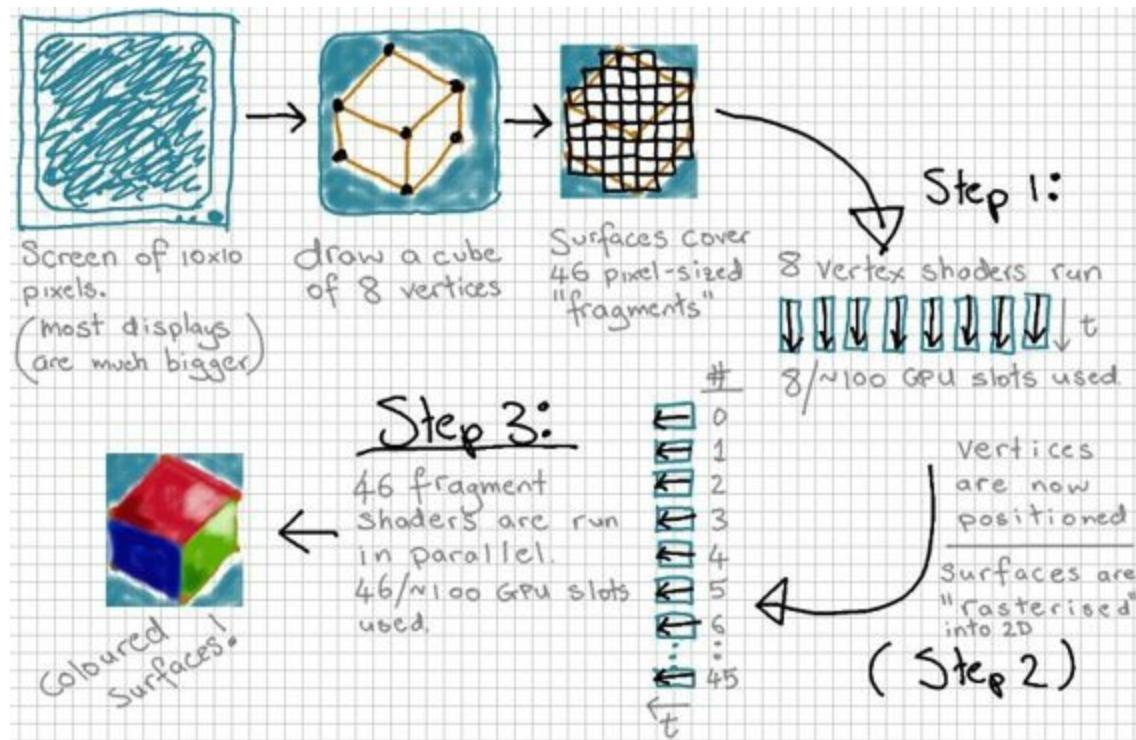
Include the GLEW header file **before** GLFW.

OpenGL 4 Shaders

Shaders tell OpenGL how to draw, but we don't have anything to draw yet - we will cover that in the vertex buffers article. If you would rather skip the introduction, you can jump straight to the **Example Shaders** and get them loaded into GL with the **Minimal C Code**.

Overview

Shaders are mini-programmes that define a **style** of rendering. They are compiled to run on the specialised GPU (graphics processing unit). The GPU has lots of processors. Each rendering stage can be split into many separate calculations - with one calculation done on each GPU processor; transform each vertex, colour each tiny square separately, etc. This means that we can compute a lot of the rendering in parallel - which makes it much faster than doing it with a CPU-based **software renderer** where we only have 1-8 processors (in the graphics world, "hardware" implies the graphics adapter). Example:



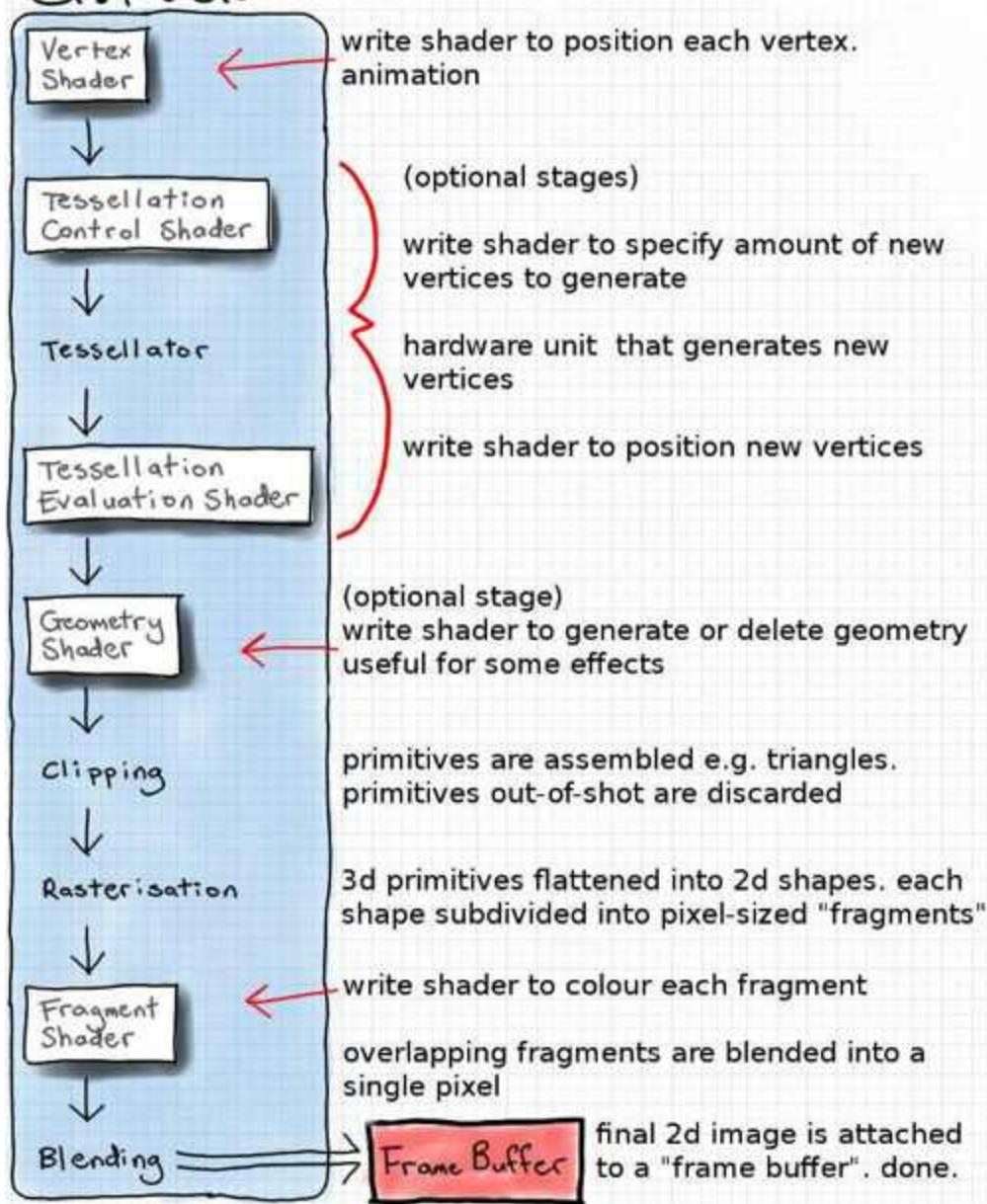
Shaders are a way of re-programming the graphics pipeline. If we wanted to use a different colouring method for the cube, or have an animated, spinning cube, we could tell OpenGL to switch to using a different shader programme. The rendering process has several distinct stages of transforming a 3d object in a final 2d image. We call this staged process the **graphics pipeline**. All of the stages of the graphics pipeline that happen on the GPU are called the

[programmable] hardware pipeline. Older OpenGL APIs had pre-canned functions like `glLight()` for driving the rendering model. We call this the **fixed-function pipeline** ("fixed" because it's not reprogrammable). These functions no longer exist, and we have to write the lighting equations ourselves in shaders. In OpenGL 4 we can write a shader to control many different stages of the graphics pipeline:

C.P.U.
glDrawArrays()
↓ go!

OpenGL 4 Hardware Pipeline

G.P.U.



A complete shader programme controls comprises a set of separate shader (mini-programmes) - one to control each stage. Each mini-programme - called a **shader** by OpenGL - is compiled, and the whole set are linked together to form the executable shader programme - called a **program** by OpenGL. Yes,

that's the worst naming convention ever. If you look at the Quick Reference Card (or further down the page) you can see that the API differentiates functions into `glShader` and `glProgram` (note US spelling of "programme").

Each individual shader has a different job. At minimum, we usually have 1 vertex shader and 1 fragment shader per shader programme, but OpenGL 4 allows us to use some optional shaders too.

Shader Parallelism

Shader programmes run on the GPU, and are highly parallelised. Each vertex shader only transforms 1 vertex. If we have a mesh of 2000 vertices, then 2000 vertex shaders will be launched when we draw it. Because we can compute each one separately, we can also run them all in parallel. Depending on the number of shader processing slots on the GPU, you might be able to compute all of your mesh's vertex shaders simultaneously.

Comparison of Selected OpenGL 4-Capable GPUs

- GeForce 605 - 48 shader cores
- Radeon HD 7350 - 80 shader cores
- GeForce GTX 580 - 512 shader cores
- Radeon HD 8750 - 768 shader cores
- GeForce GTX 690 - 1536 shader cores
- Radeon HD 8990 - 2304 shader cores

Because there is a lot of variation in user GPU hardware, we can only make very general assumptions about the ideal number of vertices or facets each mesh should have for best performance. Because we only draw one mesh at a time, keeping the number of separate meshes drawn per-scene to a low-ish level is more beneficial (reducing the **batch count** *per* rendered frame) - the idea is to keep as many of the processors in use at once as possible.

For example; my Sony Vaio Y-Series laptop says that it has an AMD 6310M graphics adapter. The AMD official specs don't actually tell me anything useful except that it runs at 500MHz, and the review magazines only tell people if they will play or not play a selection of the latest games with some very un-scientific "benchmarks" - useful for consumers perhaps, but doesn't give us any design hints. If I look this up on Wikipedia, however, it tells me that that core configuration has 80 shader slots, as well as 8 texture slots and 4 render target slots. This gives me some hint about how I might design my scenes to suit this. If I have several different objects in a scene that have less

than 80 vertices, or take up less than 80 pixels each on the screen, then I might be able to make better use of the hardware parallelism by combining them all into one vertex buffer - have them all drawn at the same time.

Difference Between Fragments and Pixels

A **pixel** is a "picture element". In OpenGL lingo, pixels are the elements that make up the final 2d image that it draws inside a window on your display. A **fragment** is a pixel-sized area of a surface. A fragment shader determines the colour of each one. Sometimes surfaces overlap - we then have more than 1 fragment for 1 pixel. **All of the fragments are drawn, even the hidden ones.**

Each fragment is written into the framebuffer image that will be displayed as the final pixels. If **depth testing** is enabled it will paint the front-most fragments on top of the further-away fragments. In this case, when a farther-away fragment is drawn after a closer fragment, then the GPU is clever enough to skip drawing it, but it's actually quite tricky to organise the scene to take advantage of this, so we'll often end up executing huge numbers of redundant fragment shaders.

Shader Language

OpenGL 4 shaders are written in OpenGL Shader Language version 4.00.9. The GLSL language from OpenGL versions 3 to 4 is almost identical, so we can port between versions without changing the code. OpenGL version 3.2 added a new type of shader: geometry shaders, and version 4.0 added tessellation control and tessellation evaluation shaders. These, of course, can not be rolled back to earlier versions. The first line in a GLSL shader should start with the simplified version tag:

```
#version 400
```

The different version tags are:

- OpenGL 1.2 - no GLSL - no tag
- OpenGL 2.0 - GLSL 1.10.59 - #version 110
- OpenGL 2.1 - GLSL 1.20.8 - #version 120
- OpenGL 3.0 - GLSL 1.30.10 - #version 130
- OpenGL 3.1 - GLSL 1.40.08 - #version 140
- OpenGL 3.2 - GLSL 1.50.11 - #version 150
- OpenGL 3.3 - GLSL 3.30.6 - #version 330
- OpenGL 4.0 - GLSL 4.00.9 - #version 400
- OpenGL 4.1 - GLSL 4.10.6 - #version 410
- OpenGL 4.2 - GLSL 4.20.6 - #version 420
- OpenGL 4.3 - GLSL 4.30.6 - #version 430

GLSL Operators

GLSL has the same operators as C, with the exception of pointers. Bit-wise operators were added in version 1.30.

GLSL Data Types

The most commonly used data types in GLSL are in the table below. For a

complete list see any of the official reference documents (<http://www.opengl.org/documentation/glsl/>).

- `void` - nothing - Functions that do not return a value
- `bool` - Boolean value as in C++
- `int` - Signed integer as in C
- `float` - Floating-point scalar value as in C
- `vec3` - 3d floating-point value - Points and direction vectors
- `vec4` - 4d floating-point value - Points and direction vectors
- `mat3` - 3x3 floating-point matrix - Transforming surface normals
- `mat4` - 4x4 floating-point matrix - Transforming vertex positions
- `sampler2D` - 2d texture loaded from an image file
- `samplerCube` - 6-sided sky-box texture
- `sampler2DShadow` - shadow projected onto a texture

If you leave out the version tag, OpenGL fall back to an earlier default - it's always better to specify the version.

File Naming Convention

Each shader is written in plain text and stored as a character array (C string). It is usually convenient to read each shader from a separate plain text file. I use a file naming convention like this;

- `texturemap.vert` - the vertex shader for my texture-mapping shader programme
- `texturemap.frag` - the fragment shader for my texture-mapping shader programme
- `particle.vert` - the vertex shader for my particle system shader programme
- `particle.geom` - the geometry shader for my particle system shader programme
- `particle.frag` - the fragment shader for my particle system shader programme

Some text editors (notepad++, gedit, ...) will do syntax highlighting for GLSL if you end with a ".glsl" extension. The GLSL reference compiler; [Glslang](#)

will check your shaders for bugs if they end in ".vert" and ".frag".

Example Shaders

GLSL is designed to resemble the C programming language. Each shader resembles a small C programme. Let us examine a very minimal shader programme that has only a vertex shader and a fragment shader. Each of these shaders can be stored in a C string, or in a plain text file.

In this case we just want to be able to accept a buffer of points and place them directly onto the screen. The hardware will draw triangles, lines, or points using these, depending on the draw mode that we set. Every pixel-sized piece (fragment) of triangle, line, or point goes to a fragment shader. Just for the sake of example, we want to be able to control the colour of each fragment by updating a `uniform` variable in our C programme.

Vertex Shader

The vertex shader is responsible for transforming vertex positions into clip space, the final coordinate space that we must transform points to before OpenGL rasterises (flattens) our geometry into a 2d image. Vertex shaders can also be used to send data from the vertex buffer to fragment shaders. This vertex shader does nothing, except take in vertex positions that are already in clip space, and output them as final clip-space positions. We can write this into a plain text file called: `test.vert`.

```
#version 400  
  
in vec3 vertex_position;  
  
void main() {  
    gl_Position = vec4(vertex_position, 1.0);  
}
```

GLSL has some built-in data types that we can see here:

- `vec3` is a 3d vector that can be used to store positions, directions, or colours.
- `vec4` is the same but has a fourth component which, in this variable, is

used to determine perspective. We will examine this in the **virtual camera** article, but for now we can leave it at 1.0, which means "don't calculate any perspective".

We can also see the `in` key-word for input to the programme from the previous stage. In this case the `vertex_position_local` is one of the vertex points from the object that we are drawing. GLSL also has an `out` key-word for sending a variable to the next stage.

The entry point to every shader is a `void main()` function.

The `gl_Position` variable is a built-in GLSL variable used to set the final **clip-space** position of each vertex.

The input to a vertex buffer (the `in` variables) are called per-vertex **attributes**, and come from blocks of memory on the graphics hardware memory called **vertex buffers**. We usually copy our vertex positions into vertex buffers before running our main loop. We will look at vertex buffers in the next tutorial. This vertex shader will run one instance for every vertex in the vertex buffer.

Fragment Shader

Once all of the vertex shaders have computed the position of every vertex in clip space, then the fragment shader is run once for every pixel-sized space (fragment) between vertices. The fragment shader is responsible for setting the colour of each fragment. Write a new plain-text file: `test.frag`.

```
#version 400

uniform vec4 inputColour;
out vec4 fragColour;

void main() {
    fragColour = inputColour;
}
```

The `uniform` key-word says that we are sending in a variable to the shader programme from the CPU. This variable is global to all shaders within the programme, so we could also access it in the vertex shader if we wanted to.

The hardware pipeline knows that the first `vec4` it gets as output from the fragment shader should be the colour of the fragment. The colours are **rgba**, or red, green, blue, alpha. The values of each component are floats between 0.0 and 1.0, (not between 0 and 255). The alpha channel output can be used for a variety of effects, which you define by setting a **blend mode** in OpenGL. It is commonly used to indicate opacity (for transparent effects), but by default it does nothing.

Minimal C Code

Note that there is a distinction between a shader, which is a mini-programme for just one stage in the hardware pipeline, and a shader programme which is a GPU programme that comprises several shaders that have been linked together.

To get shaders up and running quickly you can bang this into a minimal GL programme:

1. load a vertex shader file and fragment shader file and store each in a separate C string
2. call `glCreateShader` twice; for 1 vertex and 1 fragment shader index
3. call `glShaderSource` to copy code from a string for each of the above
4. call `glCompileShader` for both shader indices
5. call `glCreateProgram` to create an index to a new program
6. call `glAttachShader` twice, to attach both shader indices to the program
7. call `glLinkProgram`
8. call `glGetUniformLocation` to get the unique location of the variable called "`inputColour`"
9. call `glUseProgram` to switch to your shader before calling...
10. `glUniform4f` (`location, r,g,b,a`) to assign an initial colour to your fragment shader (e.g. `glUniform4f` (`colour_loc, 1.0f, 0.0f, 0.0f, 1.0f`) for red)

The only variables that you need to keep track of are the index created by `glCreateProgram`, and any uniform locations. Now we are ready to draw - we will look at drawing geometry with `glDrawArrays` in the next tutorial. To set or change uniform variables you can use the various `glUniform` functions, but they only affect the shader programme that has been switched to with `glUseProgram`.

OpenGL Shader Functions

For a complete list of OpenGL shader functions see the Quick Reference Card <http://www.opengl.org/documentation/gsl/>. The most useful functions are tabulated below. We will implement all of these:

- `glCreateShader()` - create a variable for storing a shader's code in OpenGL. returns `GLuint` index to it.
- `glShaderSource()` - copy shader code from C string into an OpenGL shader variable
- `glCompileShader()` - compile an OpenGL shader variable that has code in it
- `glGetShaderiv()` - can be used to check if compile found errors
- `glGetShaderInfoLog()` - creates a string with any error information
- `glDeleteShader()` - free memory used by an OpenGL shader variable
- `glCreateProgram()` - create a variable for storing a combined shader programme in OpenGL. returns `GLuint` index to it.
- `glAttachShader()` - attach a compiled OpenGL shader variable to a shader programme variable
- `glLinkProgram()` - after all shaders are attached, link the parts into a complete shader programme
- `glValidateProgram()` - check if a program is ready to execute. information stored in a log
- `glGetProgramiv()` - can be used to check for link and validate errors
- `glGetProgramInfoLog()` - writes any information from link and validate to a C string
- `glUseProgram()` - switch to drawing with a specified shader programme
- `glGetActiveAttrib()` - get details of a numbered per-vertex attribute used in the shader
- `glGetAttribLocation()` - get the unique "location" identifier of a named per-vertex attribute
- `glGetUniformLocation()` - get the unique "location" identifier of a named uniform variable
- `glGetActiveUniform()` - get details of a named uniform variable used in the shader

- `glUniform{1234}{ifd}()` - set the value of a uniform variable of a given shader (function name varies by dimensionality and data type)
- `glUniform{1234}{ifd}v()` - same as above, but with a whole array of values
- `glUniformMatrix{234}{fd}v()` - same as above, but for matrices of dimensions 2x2, 3x3, or 4x4

Adding Error-Checking Functionality

The first thing to do is extend the minimal code with some error-checking.

Check for Compilation Errors

Right after calling `glCompileShader`:

```
glCompileShader (shader_index);
// check for compile errors
int params = -1;
glGetShaderiv (shader_index, GL_COMPILE_STATUS, &params);
if (GL_TRUE != params) {
    fprintf (stderr, "ERROR: GL shader index %i did not compile\n", shader_index);
    _print_shader_info_log (shader_index);
    return false; // or exit or something
}
```

I call a user-defined function here to print even more information from the shader. See next section.

Print the Shader Info Log

```
void _print_shader_info_log (GLuint shader_index) {
    int max_length = 2048;
    int actual_length = 0;
    char log[2048];
    glGetShaderInfoLog (shader_index, max_length, &actual_length, log);
    printf ("shader info log for GL index %u:\n%s\n", shader_index, log);
}
```

This is the most useful shader debugging function; it will tell you which line in which shader is causing the error.

Check for Linking Errors

Right after calling `glLinkProgram`:

```
// check if link was successful
int params = -1;
```

```

glGetProgramiv (programme, GL_LINK_STATUS, &params);
if (GL_TRUE != params) {
    fprintf (stderr, "ERROR: could not link shader programme GL index %u\n", programme);
    _print_programme_info_log (programme);
    return false;
}

```

I call a user-defined function here to print even more information from the shader. See next section.

Print the Program Info Log

```

void _print_programme_info_log (GLuint programme) {
    int max_length = 2048;
    int actual_length = 0;
    char log[2048];
    glGetProgramInfoLog (programme, max_length, &actual_length, log);
    printf ("program info log for GL index %u:\n%s", programme, log);
}

```

Where `programme` is printing the index of my programme.

Print All Information

One of the more common errors is mixing up the "location" of uniform variables. Another is where an attribute or uniform variables is not "active"; not actually used in the code of the shader. We can check this by printing it, and we can print all sorts of other information as well. Here, `programme` is the index of my shader programme.

```

void print_all (GLuint programme) {
    printf ("-----\nshader programme %i info:\n", programme);
    int params = -1;
    glGetProgramiv (programme, GL_LINK_STATUS, &params);
    printf ("GL_LINK_STATUS = %i\n", params);

    glGetProgramiv (programme, GL_ATTACHED_SHADERS, &params);
    printf ("GL_ATTACHED_SHADERS = %i\n", params);

    glGetProgramiv (programme, GL_ACTIVE_ATTRIBUTES, &params);
    printf ("GL_ACTIVE_ATTRIBUTES = %i\n", params);
    for (GLuint i = 0; i < (GLuint)params; i++) {
        char name[64];
        int max_length = 64;
        int actual_length = 0;
        int size = 0;
        GLenum type;
        glGetActiveAttrib (programme, i, max_length, &actual_length, &size, &type, name);
    }
}

```

```

if (size > 1) {
    for (int j = 0; j < size; j++) {
        char long_name[64];
        sprintf (long_name, "%s[%i]", name, j);
        int location = glGetAttribLocation (programme, long_name);
        printf (" %i type:%s name:%s location:%i\n", i, GL_type_to_string (type),
long_name, location);
    }
} else {
    int location = glGetAttribLocation (programme, name);
    printf (" %i type:%s name:%s location:%i\n", i, GL_type_to_string (type),
name, location);
}
}

glGetProgramiv (programme, GL_ACTIVE_UNIFORMS, &params);
printf ("GL_ACTIVE_UNIFORMS = %i\n", params);
for (GLuint i = 0; i < (GLuint)params; i++) {
    char name[64];
    int max_length = 64;
    int actual_length = 0;
    int size = 0;
    GLenum type;
    glGetActiveUniform (programme, i, max_length, &actual_length, &size, &type, name);
    if (size > 1) {
        for (int j = 0; j < size; j++) {
            char long_name[64];
            sprintf (long_name, "%s[%i]", name, j);
            int location = glGetUniformLocation (programme, long_name);
            printf (" %i type:%s name:%s location:%i\n", i, GL_type_to_string (type),
long_name, location);
        }
    } else {
        int location = glGetUniformLocation (programme, name);
        printf (" %i type:%s name:%s location:%i\n", i, GL_type_to_string (type),
name, location);
    }
}
}

_print_programme_info_log (programme);
}

```

The interesting thing here are the printing of the attribute and uniform "locations". Sometimes uniforms or attributes are themselves arrays of variables - when this happens `size` is > 1 , and I loop through and print each index' location separately. I also have a home-made function for printing the GL data type as a string (normally it is an enum which doesn't look very meaningful when printed as an integer) - see next section.

GLenum Data Type to C String

This function converts a GL enumerated data type to a C string (for readable

printing).

```
const char* GL_type_to_string (GLenum type) {
    switch (type) {
        case GL_BOOL: return "bool";
        case GL_INT: return "int";
        case GL_FLOAT: return "float";
        case GL_FLOAT_VEC2: return "vec2";
        case GL_FLOAT_VEC3: return "vec3";
        case GL_FLOAT_VEC4: return "vec4";
        case GL_FLOAT_MAT2: return "mat2";
        case GL_FLOAT_MAT3: return "mat3";
        case GL_FLOAT_MAT4: return "mat4";
        case GL_SAMPLER_2D: return "sampler2D";
        case GL_SAMPLER_3D: return "sampler3D";
        case GL_SAMPLER_CUBE: return "samplerCube";
        case GL_SAMPLER_2D_SHADOW: return "sampler2DShadow";
        default: break;
    }
    return "other";
}
```

I only included the most-commonly used data types - I don't use the others so I just called them "other". Look up `glGetActiveUniform` to get the rest of the list.

Validate Programme

You can also "validate" a shader programme before using it. Only do this during development, because it is quite computationally expensive. When a programme is not valid, the details will be written to the program info log. `programme` is my shader programme index.

```
bool is_valid (GLuint programme) {
    glValidateProgram (programme);
    int params = -1;
    glGetProgramiv (programme, GL_VALIDATE_STATUS, &params);
    printf ("program %i GL_VALIDATE_STATUS = %i\n", programme, params);
    if (GL_TRUE != params) {
        _print_programme_info_log (programme);
        return false;
    }
    return true;
}
```

Best Practice

- All uniform variables are initialised to 0 when a programme links, so you only need to initialise them if the initial value should be something else. Example: you might want to set matrices to the identity matrix, rather than a zeroed matrix.
- Calling `gluniform` is quite expensive during run-time. Structure your programme so that `gluniform` is only called when the value needs to change. This might be the case every time that you draw a new object (e.g. its position might be different), but some uniforms may not change often (e.g. projection matrix).
- Calling `glGetUniformLocation` during run-time can be expensive. It is best to do this during initialisation of the component that updates the uniform e.g. the virtual camera for the projection and view matrices, or a renderable object in the scene for the model matrix. Store the uniform locations once, then call `glUniform` as needed, rather than updating everything every frame.
- When calling `glGetUniformLocation`, it returns -1 if the uniform variable wasn't found to be active. You can check for this. Usually it means that either you've made a typo in the name, or the variable isn't actually used anywhere in the shader, and has been "optimised out" by the compiler/linker.
- Modifying attributes (vertex buffers) during run-time is extremely expensive. Avoid.
- Get your shaders to do as much work as is possible; because of their parallel nature they are much faster than looping on the CPU for most tasks.
- Drawing lots of separate, small objects at once does not make efficient use of the GPU, as most parallel shader slots will be empty, and separate objects must be drawn in series. Where possible, merge many, smaller objects into fewer, larger objects.

Possible Extensions

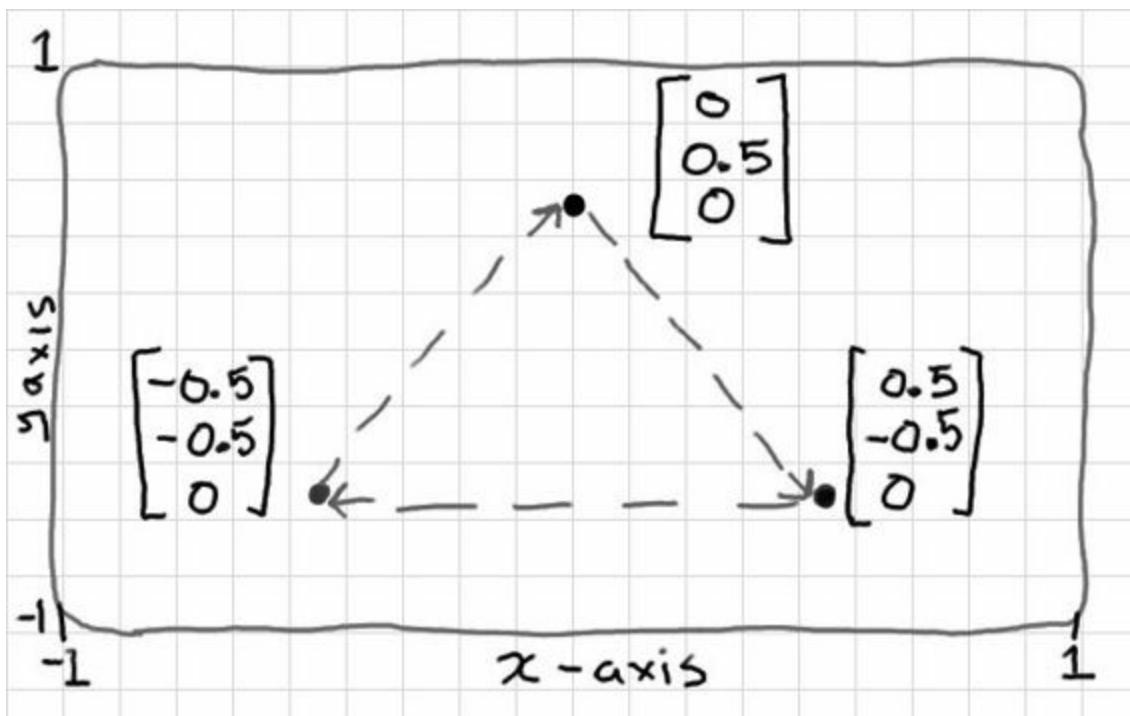
- Create a log file system, rather than a print-out system, so that you can scan through large volumes of information from each loaded shader.
- Larger projects will use many different shader programmes. It would make sense to have a *Shader Manager* interface or class to load shaders, and to make sure that shaders are re-used, rather than loaded multiple times.
- In the future we will sometimes use geometry and tessellation shaders, so we can consider upgrading our shader functions to handle more than just vertex and fragment shaders. We can just set some boolean flags to true or false to indicate which types of shader have been loaded before attaching and linking.
- If you have a shader manager, and have written a function along the lines of `setUniform (shader_index, value)` from the manager, it could then check to make sure that shader is in use first (a common cause of error).

Vertex Buffer Objects

A vertex buffer object (VBO) is nothing fancy - it's just an array of data (usually `floats`). We already had a look at the most basic use of vertex buffer objects in the Hello Triangle tutorial. We know that we can describe a mesh in an array of floats; `{x,y,z,x,y,z..x,y,z}`. And we also know that we need to use a **Vertex Array Object** to tell OpenGL that the array is divided into variables of 3 floats each.

The key idea of VBOs is this: in the old, "immediate-mode" days of OpenGL, before VBOs, we would define the vertex data in system memory (RAM), and copy them one-by-one each time that we draw. With VBOs, we copy the whole lot into a buffer before drawing starts, and this sits **on the graphics hardware memory** instead. This is much more efficient for drawing because the bottle-neck for drawing performance tends to be the bus between the CPU and the GPU. We try to minimise use of the bus, and keep as much data and processing on the graphics hardware as we can.

A VBO is not an "object" in the object-oriented programming sense, it is a simple array of data. OpenGL has several other types of data that it refers to as singular "buffer objects". I gather that the term "object" here implies that the GL gives us a handle or identifier number to the whole buffer to interact with, rather than a traditional address to the first element in the buffer. I should probably refer to these as "buffer objects" everywhere, but I'm not one for pedantry, nor am I beholden to any group, so I may well use "vertex buffer", and "vertex buffer object" interchangeably in the text.

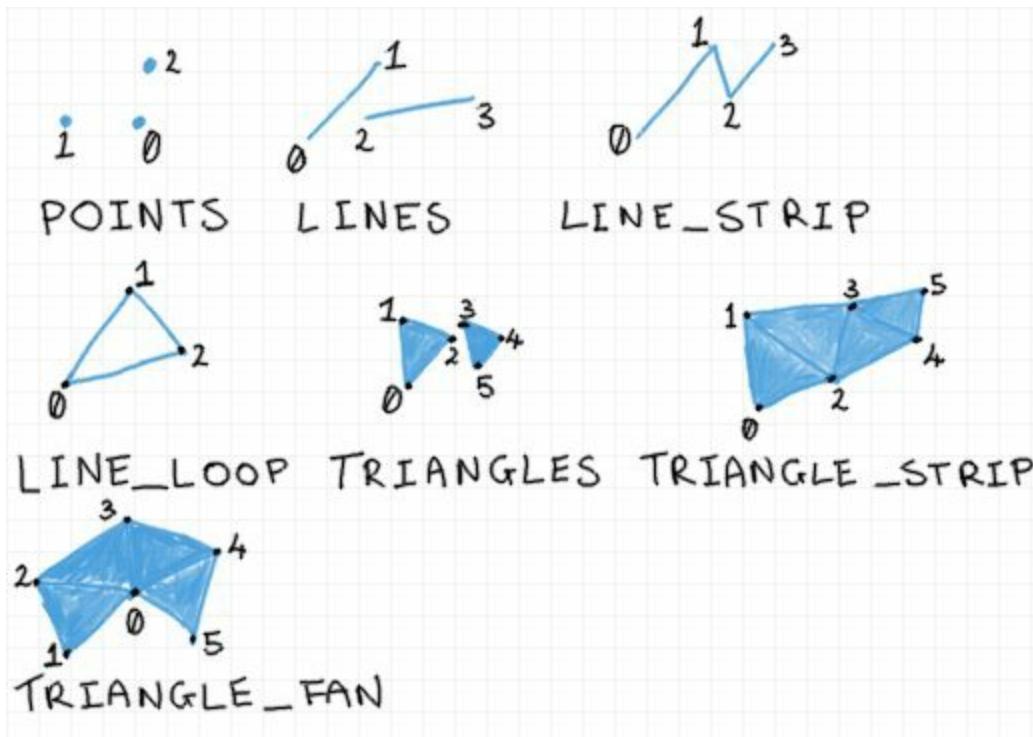


This shape might have a vertex buffer: {0, 0.5, 0, 0.5, -0.5, 0, -0.5, -0.5, 0}

We will look at managing vertex buffers in a little bit more depth. We will need this for enabling lighting and texturing effects later. We will break down the interface, and visualise how interpolation between the vertex shader and the fragment shader works.

Rendering Different Primitive Types

Write up a working demo that renders a triangle. You might use **Hello Triangle** if you haven't got one already. Then we can play with the `glDrawArrays()` function. So far, we have just looked at rendering triangles from sets of 3 points, but you can actually render in several different modes:



Try changing your `GL_TRIANGLES` parameter. You can see that points and lines are going to be useful for drawing things like charts or outlines. Triangle strip is a slightly more efficient method for drawing ribbon-like shapes. I've never found a legitimate use for triangle fans. You can actually change the size of the points in the vertex shader, so they are quite versatile. Lines no longer have very good supporting functions, so they're not as useful.

But How Do I Render a Wire-Frame Mode?

All of my students tried to make a wire-frame rendering mode so I thought I'd

point this out here - it's much easier to use the **glPolygonMode** built-in function than to attempt to draw everything with `GL_LINES`. Call `glPolygonMode(GL_FRONT, GL_LINE);` before rendering.

Using Multiple Vertex Buffers for One Object

We can store more than just 3d points in a vertex buffer. Common uses also include; 2d texture coordinates that describe how to fit an image to a surface, and 3d normals that describe which way a surface is facing so that we can calculate lighting. So it's quite likely that most of your objects will have 2 or 3 vertex buffers each.

You can use vertex buffers to hold any data that you like; the key thing is that the data is retrieved **once per vertex**. If you tell OpenGL to draw an array of 3 points, using a given vertex array object, then it is going to launch 3 vertex shaders in parallel, and each vertex shader will get a one variable from each of the attached arrays; the first vertex shader will get the first 3d point, 2d texture coordinate, and 3d normal, the second vertex shader will get the second 3d point, 2d texture coordinate, and 3d normal, and so on, where the number of variables, and size of each variable is laid out in the vertex array object.

Define Vertex Colours

Vertex colours are very seldom used in practise, but most modern GL tutorials will get you to create a buffer of colours as a second vertex buffer. Why? Because it's easy to **visualise how interpolation works** with colours. So let's do that - our colours will be similar to 3d points - except they will have r,g,b values of 0.0 to 1.0 instead of positions. Let's use our triangle of points again (as pictured above), and make a buffer of colours with one vertex completely blue, one completely red, and one completely green.

```
GLfloat points[] = {
    0.0f,  0.5f,  0.0f,
    0.5f, -0.5f,  0.0f,
   -0.5f, -0.5f,  0.0f
};
```

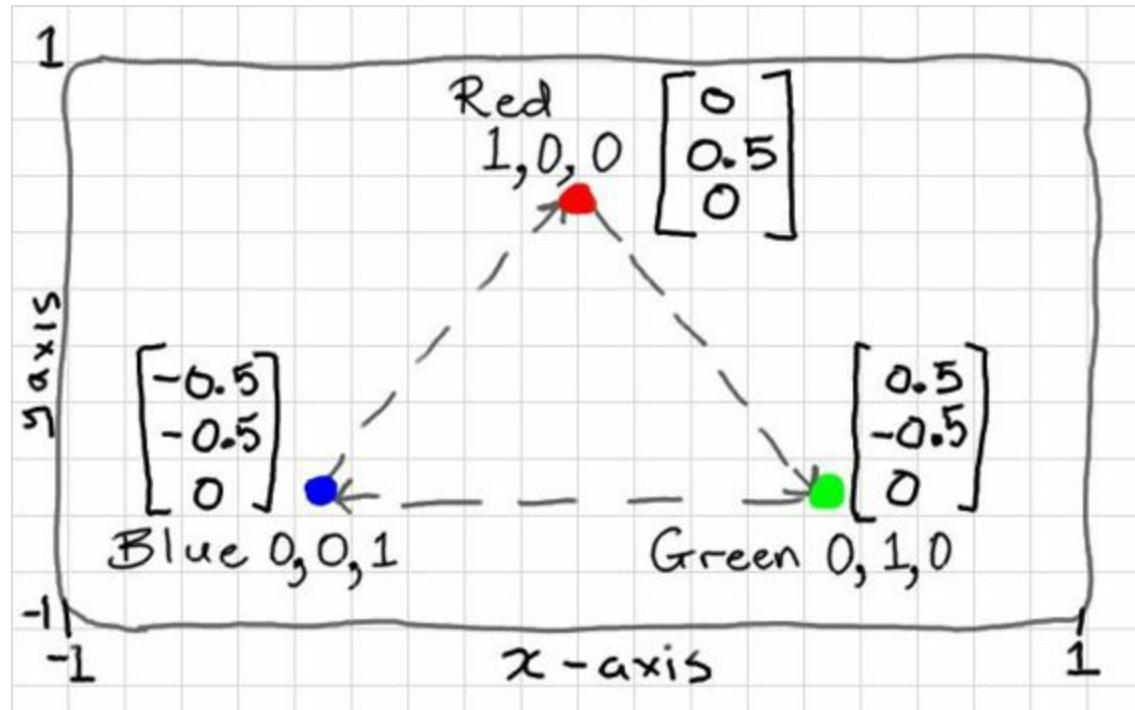
```
GLfloat colours[] = {
    1.0f, 0.0f, 0.0f,
```

```

    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f
};

```

You can see that the colour values are going to correspond to the position values. Example:



Create VBOs

Okay, now for each one we can create a GL vertex buffer object, bind it in the state machine, and copy the array of values into it. Both colours, and points have 9 components each (3 vertices with 3 components per vertex). We set up one buffer after the other, because the state machine can only have 1 buffer bound at a time.

```

GLuint points_vbo = 0;
 glGenBuffers (1, &points_vbo);
 glBindBuffer (GL_ARRAY_BUFFER, points_vbo);
 glBufferData (GL_ARRAY_BUFFER, sizeof (points), points, GL_STATIC_DRAW);

GLuint colours_vbo = 0;
 glGenBuffers (1, &colours_vbo);
 glBindBuffer (GL_ARRAY_BUFFER, colours_vbo);
 glBufferData (GL_ARRAY_BUFFER, sizeof (colours), colours, GL_STATIC_DRAW);

```

Define VAO

Our object now consists of 2 vertex buffers, which will be input "attribute" variables to our vertex shader. We set up the layout of both of these with a single vertex attribute object - the VAO represents our complete object, so we no longer need to keep track of the individual VBOs.

Note that we have to bind each VBO before calling `glVertexAttribPointer()`, which describes the layout of each buffer to the VAO.

The first parameter of `glVertexAttribPointer()` asks for an index. This is going to map to the indices in our vertex shader, so we need to give each attribute here a unique index. I will give my points index 0, and the colours index 1. We will make these match up to the variables in our vertex shader later. If you accidentally leave both indices at 0 (easy enough to do when copy-pasting code), then your colours will be read from the position values so $x \rightarrow$ red $y \rightarrow$ green and $z \rightarrow$ blue.

Both buffers contain arrays of floating point values, hence `GL_FLOAT`, and each variable has 3 components each, hence the 3 in the second parameter. If you accidentally get this parameter wrong (quite a common mistake), then the vertex shaders will be given variables made from the wrong components (e.g. position x,y,z gets values read from a y,z,x).

```
GLuint vao = 0;
 glGenVertexArrays (1, &vao);
 glBindVertexArray (vao);
 glBindBuffer (GL_ARRAY_BUFFER, points_vbo);
 glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glBindBuffer (GL_ARRAY_BUFFER, colours_vbo);
 glVertexAttribPointer (1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

That is the mesh side taken care of - you only need to do this part once, when creating the object. There is no need to repeat this code inside the rendering loop. **Just keep track of the VAO index for each type of mesh that you create.**

Enable Vertex Arrays 0 and 1 in a VAO

Okay, so we created a vertex array object, and described 2 attribute "pointers" within it. Unfortunately, **attributes are disabled by default** in OpenGL 4. We need to explicitly enable them too. This is easy to get wrong or overlook, and is not well explained in the documentation. We use a function called `glEnableVertexAttribArray()` to enable each one. This function only affects the currently bound vertex array object. This means that when we do this now, it will only affect our attributes, above. We will need to bind every new vertex array and repeat this procedure for those too.

```
glEnableVertexAttribArray (0);  
glEnableVertexAttribArray (1);
```

We are using 2 attributes (points and colours), and we know that these are numbered 0 (points), and 1 (colours); matching the numbers we gave when setting up the vertex attribute pointers, earlier.

A little aside about this...

Other GL incarnations don't have vertex attribute objects, and need to explicitly enable and disable the attributes every time a new type of object is drawn, which changes the enabled attributes globally in the state machine. We don't need to worry about that in OpenGL 4; **the vertex attribute object will remember its enabled attributes**. i.e. they are no longer global states. This isn't clear in the official documentation, nor is it explained properly in other tutorials - it's very easy to get a false-positive misunderstanding of how these things work and run into hair-pulling problems later when you're trying to draw 2 different shapes. I actually figured this out by writing a programme that used the scientific method to try and falsify (break in every conceivable way) my theory for how the logic of attribute enabling worked; if I'd just tried it until it worked (deduction) then I'd have been wrong! Well...I just don't know what to say...the bind/enable design is not a good one.

If you forget to enable an attribute, the shader won't know where to read the data from and you'll get weird results e.g. black colours. I believe attribute 0 is enabled by default, but the second attribute that we added will not be. This process is a bit excessive, and I feel should really be handled internally by OpenGL, but in this version we still need to do it.

Modify the Vertex Shader

Now, we want our shader to render using the second vertex buffer, so we need to add a second attribute to the top of the vertex shader. We are going to add the OpenGL 4 `layout` prefix to each attribute. This lets us manually specify a `location` for each attribute - and we are going to match this up to the index that we gave each one in `glVertexAttribPointer`. If you don't specify a location, the shader programme linker will automatically assign one. You can query this, which we did in **Shaders**, but I prefer to specify it manually so I can see the values as I write.

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_colour;

out vec3 colour;

void main () {
    colour = vertex_colour;
    gl_Position = vec4 (vertex_position, 1.0);
}
```

Now, the other interesting change here is that I took the `vertex_colour` input attribute and gave it to an **output** variable, which I called `colour`. This is where it gets interesting. It has no effect on our vertex shader, but outputs it to the next stage in the programmable hardware pipeline. In our case - the fragment shader.

If you want your GLSL code to be compatible with OpenGL 3.2, for example, so that it runs on a Mac, then you can't use the `layout` keyword. You can achieve the exact same effect by telling the shader's linker which variable should have which location. Right after compiling but before linking, you can call `glBindAttribLocation()` to do this:

```
...
glCompileShader (my_vertex_shader);
glCompileShader (my_fragment_shader);
glAttachShader (shader_programme, my_fragment_shader);
glAttachShader (shader_programme, my_vertex_shader);

// insert location binding code here
glBindAttribLocation (shader_programme, 0, "vertex_position");
glBindAttribLocation (shader_programme, 1, "vertex_colour");

glLinkProgram (shader_programme);
...
```

You don't need to use both methods - one or the other is fine. The third option is to let the linker decide which location to give each input variable, and then you use a value returned by `glGetAttribLocation()` for the first parameter of calls to `glVertexAttribPointer()`.

Modify the Fragment Shader

We can rewrite the fragment shader to use our new `colour` variable as an input, which will colour each fragment directly. Note that we add an `in` prefix to retrieve a variable from a previous stage. This in/out convention is a little different to how other OpenGL versions work. Our output fragment colour needs to be r,g,b,a (4 components) so we can just add a 1 to the end of our 3-component colour by casting it as a `vec4`.

```
in vec3 colour;
out vec4 frag_colour;

void main () {
    frag_colour = vec4 (colour, 1.0);
}
```

Test It

That's it! Our drawing loop should look the same as with the Hello Triangle demo. Note that it's just the single VAO that needs to be bound for drawing; other GL implementations will have you bind both arrays, but not OpenGL 4:

```
while (!glfwWindowShouldClose (window)) {
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glUseProgram (shader_programme);
    glBindVertexArray (vao);
    glDrawArrays (GL_TRIANGLES, 0, 3);
    glfwPollEvents ();
    glfwSwapBuffers (window);
    if (GLFW_PRESS == glfwGetKey (window, GLFW_KEY_ESCAPE)) {
        glfwSetWindowShouldClose (window, 1);
    }
}
```

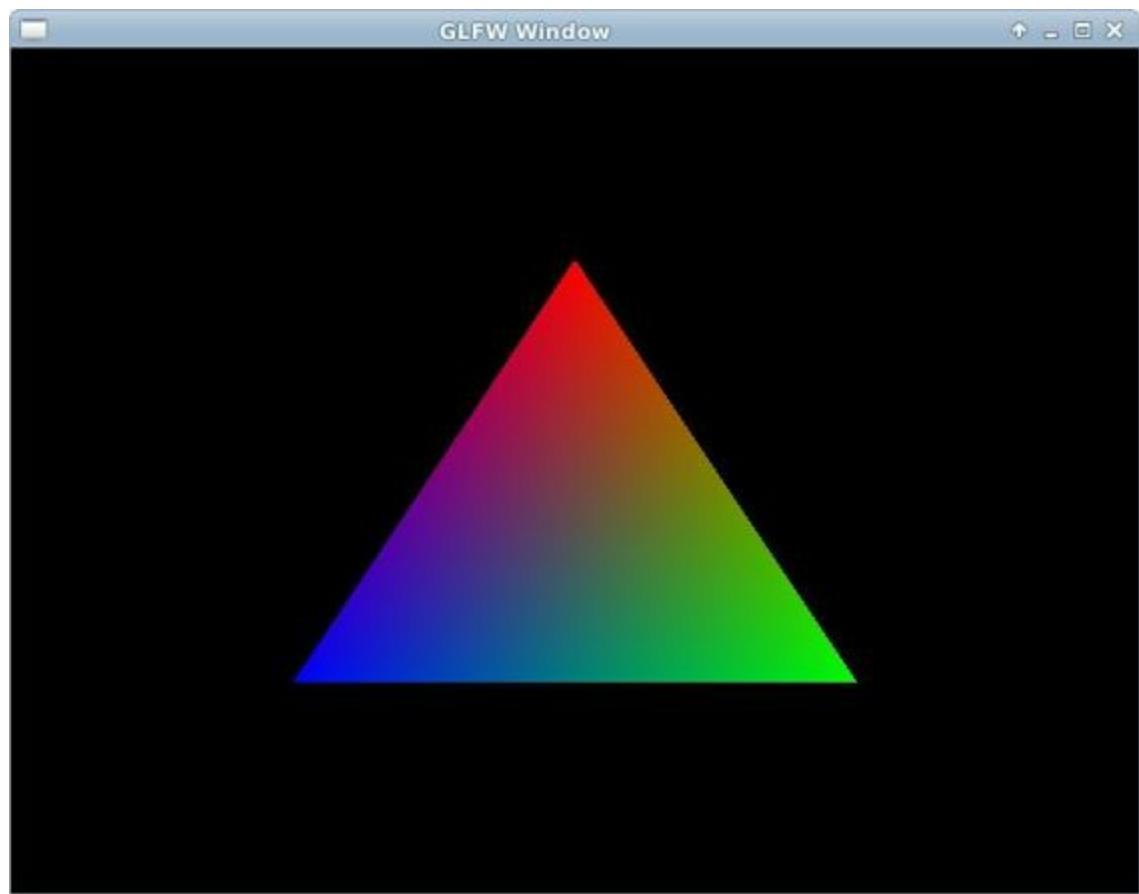
You can test that it works now!

Vertex Shader to Fragment Shader Interpolation

Now, remember, our triangle has **only 3 vertices**, but **1 fragment for every pixel-sized area of the surface**. This means that we have 3 colour outputs from vertex shaders, and perhaps 100 colour inputs to each fragment shader. How does this work?

The answer is that each fragment shader gets an **interpolated** colour based on its position on the surface. The fragment exactly on the red corner of the triangle will be completely red (1.0, 0.0, 0.0). A fragment exactly half-way between the blue and red vertices, along the edge of the triangle, will be purple; half red, half blue, and no green: (0.5, 0.0, 0.5). A fragment exactly in the middle of the triangle will be an equal mixture of all 3 colours; (0.3333, 0.3333, 0.333).

Keep in mind that this will happen with any other vertex buffer attributes that you send to the fragment shader; normals will be interpolated, and texture coordinates will be interpolated to each fragment. This is really handy, and we will exploit it for lots of interesting per-pixel effects. But it's quite common to misunderstand this when getting started with shaders - **vertex shader outputs are not sending a constant variable from a vertex shader to all fragment shaders**. We use uniform variables for that.



"Winding" and Back-Face Culling

The last thing that you should know about is a built-in rendering optimisation called back-face culling. This gives a hint to GL so that it can throw away the hidden "back" or inside faces of a mesh. This should remove half of the vertex shaders, and half of the fragment shader instances from the GPU - allowing you to render things twice as large in the same time. It's not appropriate all of the time - you might want our 2d triangle to spin and show both sides.

The only things that you need specify are if clock-wise vertex "winding" order means the front, or the back of each face, and set the GL state to enable culling of that side. Our triangle, you can see, is given in clock-wise order. Most mesh formats actually go the other way, so it pays to test this before wondering why a mesh isn't showing up at all!

```
glEnable (GL_CULL_FACE); // cull face  
glCullFace (GL_BACK); // cull back face  
glFrontFace (GL_CW); // GL_CCW for counter clock-wise
```

Try switching to counter clock-wise to make sure that the triangle disappears. If you were to rotate it around now you'd see the other side was visible. As with other GL states, this culling is enabled globally, in the state machine, you can enable and disable it between calls to `glDrawArrays` so that some objects are double-sided, and some are single-sided, etc. **Keep in mind which winding order you are making new shapes in..**

Common Mistakes

- ***I get positions but no colours!*** - Check the `glVertexAttribPointer` parameters. Make sure that each buffer has an unique **index** (first parameter). Make sure that these indices match up to the attribute **locations** in the vertex shader.
- ***My positions/colours/normals/texture coordinates are there, but wrong!*** - First, check the index given to `glVertexAttribPointer` against the actual location value in the vertex shader - have you mixed up 2

attributes so your colours are being read from positions?

- ***That didn't fix it!*** - Check the size and type parameters in `glVertexAttribPointer`. Most variables are 3-component floats, but some (like texture coordinates) should not be. Also check your vertex shader - did you specify the correct `vec3`, `vec2`, etc. type to match?
- ***Something is still wrong!*** - Check for any **shader errors** - most commonly a typo, or a mix-up between `vec4` and `vec3` types.

Vectors and Matrices

A *vector* means "a carrier" in Latin - some sort of storage. In our case, we represent 3d points, distances, and directions, which have 3 components each, as a single vector variable. We can also have 2d coordinates, and 4d points and directions - all represented as vectors. This is particularly useful when doing transformations. There are well-established mathematical functions for rotating, scaling, and translating (moving) vectors by multiplying them with a **transformation matrix**. Note that, in vector terminology, a variable that only has a single value; a length or a magnitude, for example, is referred to as a *scalar*.

Multiplying a vector with a matrix is done in a fixed order, and this produces a new, transformed, vector. Because of this fixed order, we can lay out a matrix to produce a particular affine operation; scale, rotate, or translate. You probably did this at school with 3X3 matrices and 3-component vectors.

Homogeneous Matrices

Now, you know that you can use Euler transformations and a bit of Pythagoras' theorem by hand to define rotations, translations are just a bit of addition, and scaling is just simple multiplication, so we don't actually need matrix transformations. The real advantage comes when using 4X4 "homogeneous" matrices because the extra space allows us to combine several transformations together into one matrix. We create a rotation, scaling, and translation matrix, and multiply them all together to combine them into a single 4X4 matrix. We will also use the extra bit to compute perspective in the next article.

But what does a four-dimensional vector entail? Actually, this isn't anything special, we are again just exploiting the order that matrices multiply with vectors. Depending on row-major or column-major convention, the translation part of the 4d matrix will occupy the bottom row, or the right-most column, respectively. If the fourth component of the vector is 0, then it is not affected by the translation part. If the fourth component is 1, then it will be translated. This is convenient for us, because we want our points to be translated, but not our directions (we never move a direction, but we do want them to be rotated).

Commonly Used Vector Types

- 2d coordinate - [x, y] (2d point) or [s, t] (texture coordinate) - GLSL type `vec2`
- 3d point - [x, y, z] - GLSL type `vec3`
- 3d direction - [x, y, z] - GLSL type `vec3`
- 4d point - [x, y, z, 1.0] - GLSL type `vec4`
- 4d direction - [x, y, z, 0.0] - GLSL type `vec4`

Note that, a 4X4 matrix can only transform a 4d vector, and a 3X3 matrix can only transform a 3d vector. In OpenGL and GLSL, we tend to do a lot of re-casting of vectors between 3 and 4 components because of this. **Accidentally**

mixing up 4d and 3d vectors and matrices is the most common source of error in shader programmes.

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Indices of Column-Order Memory

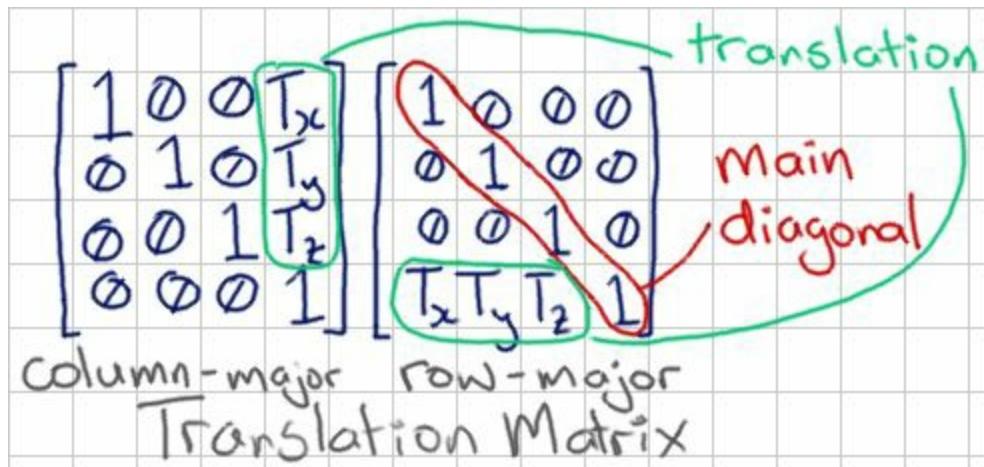
We generally store a 4x4 matrix as a regular, one-dimensional array of `float`s; i.e. `float matrix[16];`. The indices in your array are counted in either down columns (as above), or across rows. OpenGL expects to be given the array in column order, as above, but you can use row-order and transpose it when calling `glUniform...` if you prefer.

Commonly Used Matrix Types

- 3X3 - column order - `mat3` - transforming surface normals
- 4X4 - column order - `mat4` - affine transforms, virtual camera transforms

The memory order in the table above is worth noting - when we copy a matrix from C to be used in a shader as a uniform variable, we don't give it a 2d array, as we might expect, we give it a simple 1d array. The order counts down the left-most column, then down the second column, and so on. If your memory is in row order, then you can call the appropriate `glUniform` function, and set the transposition parameter to `GL_TRUE`, and it will re-arrange it for you.

Row Order and Column Order



When you are computing vectors and matrices in C code, you need to stick to a layout convention. The two layouts are **row major** and **column major**. You can use either convention, but it needs to be consistent. For no particular reason, OpenGL people tend to favour column-major matrices, and Direct3D people row-major. You can swap between row-major and column-major by **transposing** a matrix. This flips it along the **main diagonal** (top-left to bottom-right diagonal). It's easy to spot if a 4X4 matrix is in row order because the transformation components will be on the bottom-row, and there will be zeros in the right-most column (as in the example drawn above). Column-major matrices will have the transformation in the right-most column. The layout convention affects the order of multiplication, as detailed in below:

Operation Order	
$\vec{v}' = T \times R \times \vec{v}$	rotate, then translate
$\vec{v}' = R \times T \times \vec{v}$	rotate <u>around</u> a point
$\vec{v}' = \vec{v} \times T \times R$ column-major	\times invalid
	$\vec{v}' = T \times R \times \vec{v}$ row-major

Where R is a rotation matrix, T is a translation matrix, v is a vector and v' is the resulting vector.

So, there is a specific order for multiplying matrices together, or vectors by matrices. We also see that 4d matrices are **non-commutative** - we get a different result if we multiply them in a different order. We will do vector and matrix multiplication in both C and in shaders, so it's worth remembering which convention is being used. In the column-order examples, you can see we are going right-to-left; e.g. input vector, rotate, then translate.

Matrix * Vector

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

*Formula for multiplying a matrix by a vector. We will exploit this order to create multi-purpose matrices. This notation is in **column-major** form, which I use. In row order we write the vector on the left of the matrix, and in a row. The letters in row order would also go down the columns instead of across the rows. You can see how the translation matrix, in the earlier image, would only affect vectors which have a 1 in the w component.*

A minor point - in maths notation, row-major vectors are drawn in a row, and column-major vectors are drawn in a column. Matrices are expressed as a capital letter, and vectors a lower-case letter, either in **bold-face**, or with a harpoon/arrow symbol above them. Normalised (unit) vectors have a "hat" \hat{v} rather than an arrow.

So, what have we learned? We can write a 4X4 matrix directly into a 1d array in C, and send that to a shader using `glUniform`, where it will appear as a `uniform mat4`. We can then use it to transform our vertex points using an affine transformation. We already have the layout of a translation matrix, so let's try that.

Matrix Translation Demo

Okay, let's try to get some stuff moving. Start a simple project. You can use the Hello Triangle demo if you don't have one yet. Let's define a matrix that moves the triangle 0.5 units to the right. Remember that the edge of the view is -1 on the left and 1 on the right, so we don't want to move it completely off the screen. Our matrix will look like this in column-major layout:

```
float matrix[] = {  
    1.0f, 0.0f, 0.0f, 0.0f, // first column  
    0.0f, 1.0f, 0.0f, 0.0f, // second column  
    0.0f, 0.0f, 1.0f, 0.0f, // third column  
    0.5f, 0.0f, 0.0f, 1.0f // fourth column  
};
```

It looks transposed to the way we would write it down in matrix notation when we code it up like this, but I don't want to have to transpose it when I give it to GL.

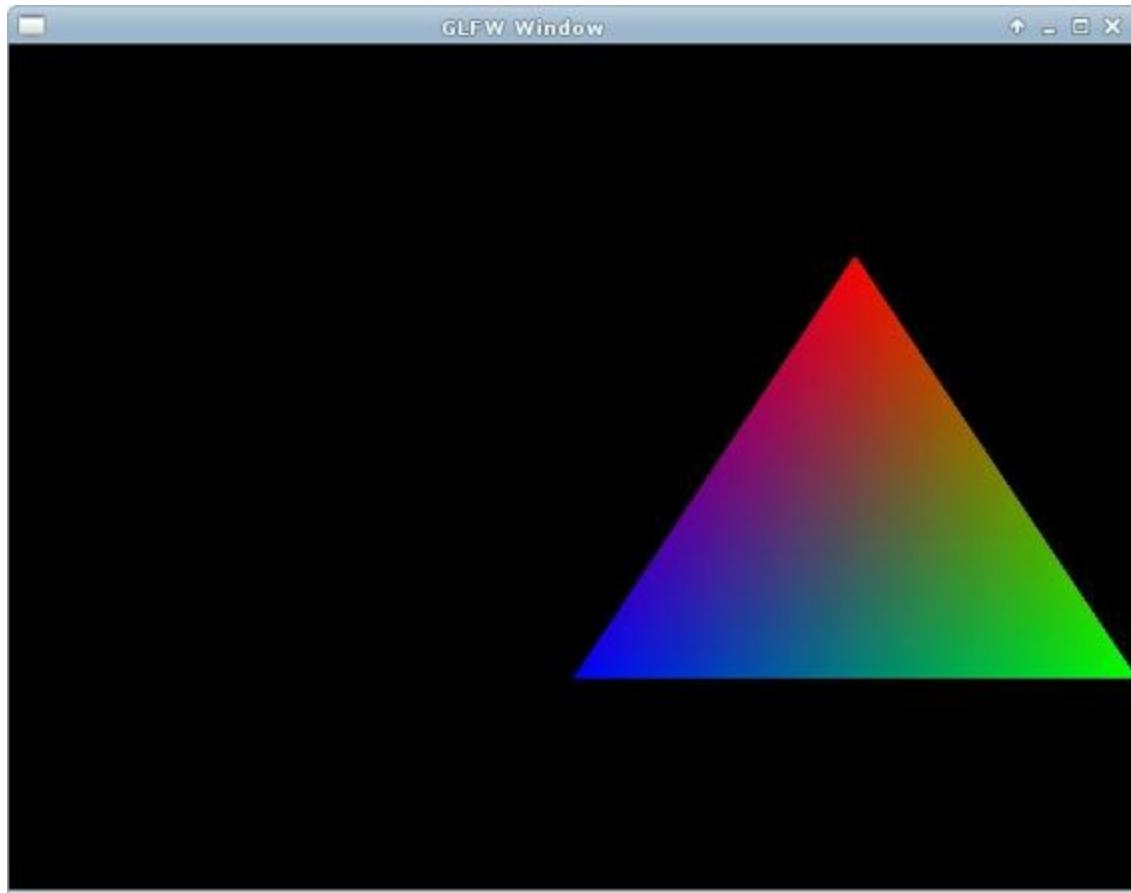
Now we need to add the uniform variable into the vertex shader, and actually do the multiplication with the vertex point. The vertex shader will look something like this:

```
in vec3 vertex_position;  
in vec3 vertex_colour;  
  
uniform mat4 matrix; // our matrix  
  
out vec3 colour;  
  
void main () {  
    colour = vertex_colour;  
    gl_Position = matrix * vec4 (vertex_position, 1.0);  
}
```

Note that the multiplication here is in column-major order as well. Okay, now we should be able to retrieve the location of the new `matrix` uniform. Do this somewhere after where you link the shader programme, but before entering the main loop. Once we have that, we can "use" the shader programme, and send in our matrix.

```
int matrix_location = glGetUniformLocation (shader_programme, "matrix");  
glUseProgram (shader_programme);  
glUniformMatrix4fv (matrix_location, 1, GL_FALSE, matrix);
```

Now we can compile it, and see what happens. The triangle should be moved to the right of the screen. If you put the wrong name into `glGetUniformLocation` then it will return a value of -1, so you can check for this.



Make it Move Back and Forth!

A lot of transformation that we do in 3d involves some sort of animated movement. To get this to work we need a timer and a speed (or velocity for a 3d speed). The translation matrix is then re-created every frame with *time since last movement * speed + last position*. We can set this up with GLFW's timer, and keep track of the last position. I'll just add some code to the main loop here - you will recognise this from previous tutorials.

```
float speed = 1.0f; // move at 1 unit per second
float last_position = 0.0f;
while (!glfwWindowShouldClose (window)) {
    // add a timer for doing animation
    static double previous_seconds = glfwGetTime ();
    double current_seconds = glfwGetTime ();
    double elapsed_seconds = current_seconds - previous_seconds;
    previous_seconds = current_seconds;

    // reverse direction when going to far left or right
    if (fabs(last_position) > 1.0f) {
        speed = -speed;
    }

    // update the matrix
    matrix[12] = elapsed_seconds * speed + last_position;
    last_position = matrix[12];
    glUseProgram (shader_programme);
    glUniformMatrix4fv (matrix_location, 1, GL_FALSE, matrix);

    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glUseProgram (shader_programme);
    glBindVertexArray (vao);
    glDrawArrays (GL_TRIANGLES, 0, 3);
    glfwPollEvents ();
    glfwSwapBuffers (window);
    if (GLFW_PRESS == glfwGetKey (window, GLFW_KEY_ESCAPE)) {
        glfwSetWindowShouldClose (window, 1);
    }
}
```

I made the movement change direction when it reached either edge of the screen (plus or minus 1). The `fabs()` function (floating point absolute value) is in `math.h` if you want to include that too. We were able to just modify our matrix here because we know it already had sensible values. Note that I only changed index 12 in the matrix. Can you tell why?

Try Some More

Rotation and Scaling

The next thing to try is rotation around different axes, and scaling. You can find the matrix layouts for these on my 3d maths cheat sheet:

http://antongerdelan.net/teaching/3dprog1/mathсs_сheat_sheet.pdf. Remember to transpose them if you are using row-order layout, as my sheet is in column-order. Note that rotation is limited to rotating around either the x , y , or z axis. To get an in-between orientation you can multiply all three matrices together, but be careful, as these are also non-commutative, and an x -axis rotation followed by a y -axis rotation will give you a different result to the y -axis rotation, followed by the x -axis rotation.

The Model Matrix

If we wanted to translate, rotate, and scale our points, we could send in 3 separate uniform matrices, and do the multiplication using the built-in GLSL functionality, but it is more conventional to send in a single, combined, matrix. Doing this means that we need a function for multiplying 4d matrices together in C. This combined matrix is usually referred to as the **model matrix**, or "world matrix".

There are no longer any matrix functions in OpenGL. You have a choice - you can write your own matrix multiplication code, or you can use somebody else's. I prefer to write my own functions, which makes it much easier to debug later on. You'll need to know a bit about vector and matrix maths to code it, but you will definitely get stuck with 3d programming if you don't know how to multiply a matrix, or what a dot product is, so it's worth teaching yourself as you code. The matrix multiplication code should be about 10 lines, and it doesn't take very long to write all of the 3d maths code that you will eventually need (perhaps a day and a just a single page of code), but if you would prefer a pre-packaged deal to start with you can use my C++ maths library; http://antongerdelan.net/opengl/mathсs_funcs.h, and

http://antongerdelan.net/opengl/math_funcs.cpp. I designed it to resemble GLSL, so there are `structs` called `vec3` and `mat4`. They just contain arrays of `floats`, which you can get to with `my_vec.v`, and `my_mat.m`. The GLM library; <http://glm.g-truc.net/> library is very popular, but I find it very hard to read the template programming style, which is not really necessary for such simple maths.

If You Haven't Done Matrix Maths for 15 Years

Okay, it's not that hard to pick up, but it's worth working out all the major functions on paper so that you know what they do. Remember that it all hinges on the order that matrix multiplication works in. I suggest this approach:

- make a list of functions that you need to know; `matrix*vector`, `matrix*matrix`, create identity matrix, create x, y, and z rotation matrices, create translation matrix, etc.
- get a simplified theory for each one, try it on paper (particularly get the order of matrix multiplication)
- start building a 3d maths cheat sheet, similar to mine, but with the bits that you need to remember
- quiz yourself
 - what happens if you multiply a vector by an identity matrix?
 - by a matrix of all zeros?
 - what does a transposed version of a matrix look like?
 - can you rotate a vector using a rotation matrix?
 - what happens mathematically when you multiply a transformation matrix with a 4d vector ending in 0 or 1?

Problem?

The most common mistakes:

- ***My shape disappeared!*** - check how far you are moving it; maximum -1 to 1 on x and y. Also check your matrix is being sent to the shader - all uniform variables are zero by default.
- ***Shader compilation error!*** - Trying to multiply a `vec3` by a `mat4`. Recast like this for a point: `vec4 result = matrix * vec4 (input_vector, 1.0);`
- ***My transformation is inside-out!*** - Row and column-order multiplication are mixed up. [Check the order of multiplication](#), and compare to the layout of your matrix.
- ***My translation isn't doing anything!*** - `vec4` point has a 0 for the last component, when it should have a 1.
- ***My translation still isn't doing anything!*** - check that your matrix uniform location is correct (see Shaders article for checking uniform locations), and is being sent with `glUniform`.

Next

Next we will go further with matrices, and create a representation of a virtual camera, that can move around the scene. You can probably guess how we will do that part, given what we have looked at so far, but we will also add some perspective to the scene to create the illusion of distance and 3d space.

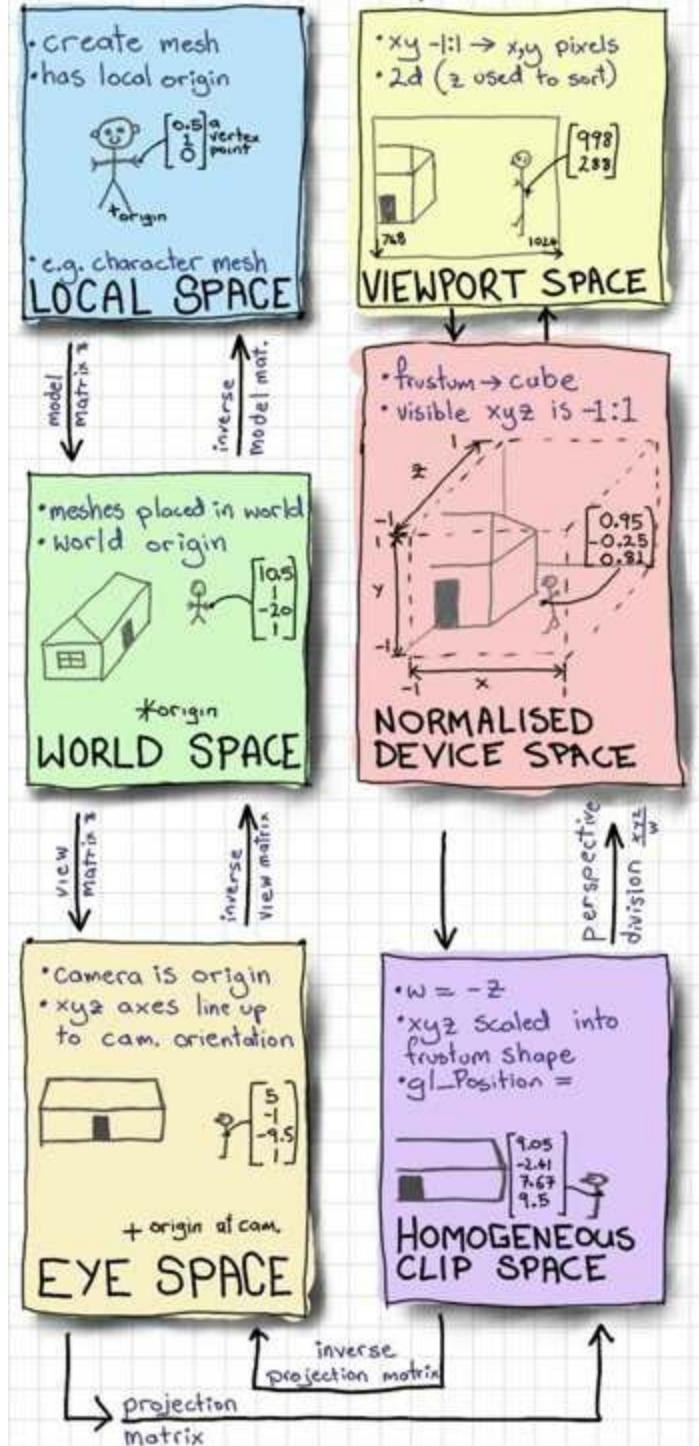
Virtual Camera

We would like to be able to navigate through our scene in 3d. To do this we need to set up a **transformation pipeline** which will move all of our points relative to the position and orientation of a virtual camera, or view point. We are also going to define how to project our scene with a **perspective**, rather than the orthogonal (squared) view that we have been using so far. This is going to make it look much more realistic, and create an illusion of depth. We can also add a few more keyboard controls for our camera.

The Transformation Pipeline

We have already seen how we can position our objects using affine transforms, or a "model matrix". When we first create our object, or load a mesh, the vertex points are given, relative to some [0,0,0] **local origin**. In this coordinate system, we talk about the points being in **local coordinate space**. If we had 2 triangles, for example, created from the same points, and moved them both to different positions by multiplying them with a **model matrix**, then we can imagine that they are now relative to some **world origin**, and are therefore in a **world coordinate space**. In world space, we can compare the distances or angles between objects. You can imagine world space being a kind of description of the layout of all of the scenery in a computer game world. In order to "move around" in this world we need to rearrange to coordinates relative to a camera position and orientation. We would also like to create the illusion of depth with perspective. This adds a few new steps to our transformations before they are output from the vertex shader.

3d Transformation Pipeline



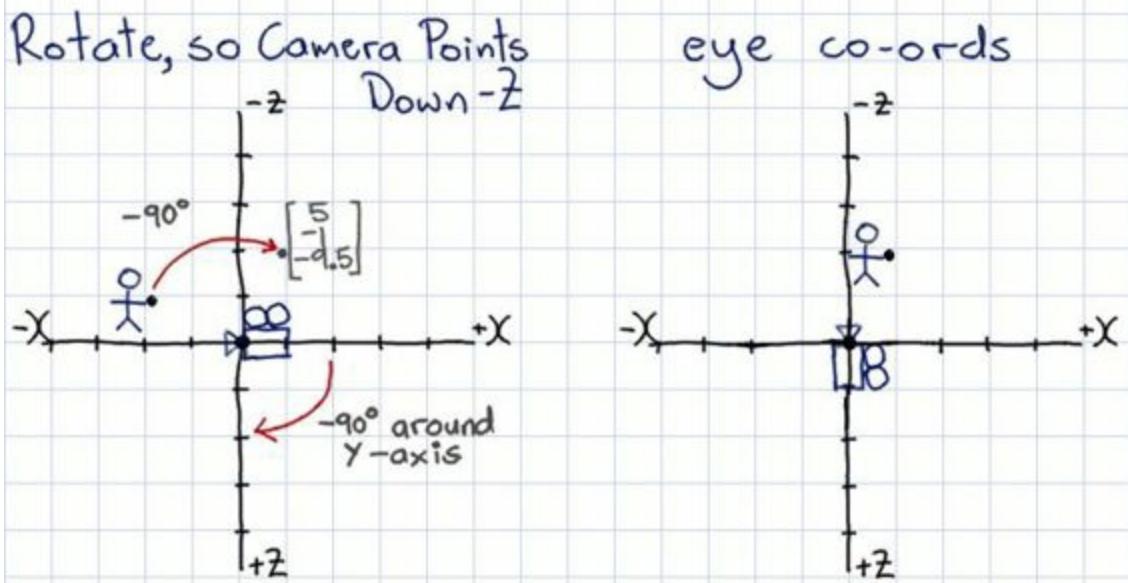
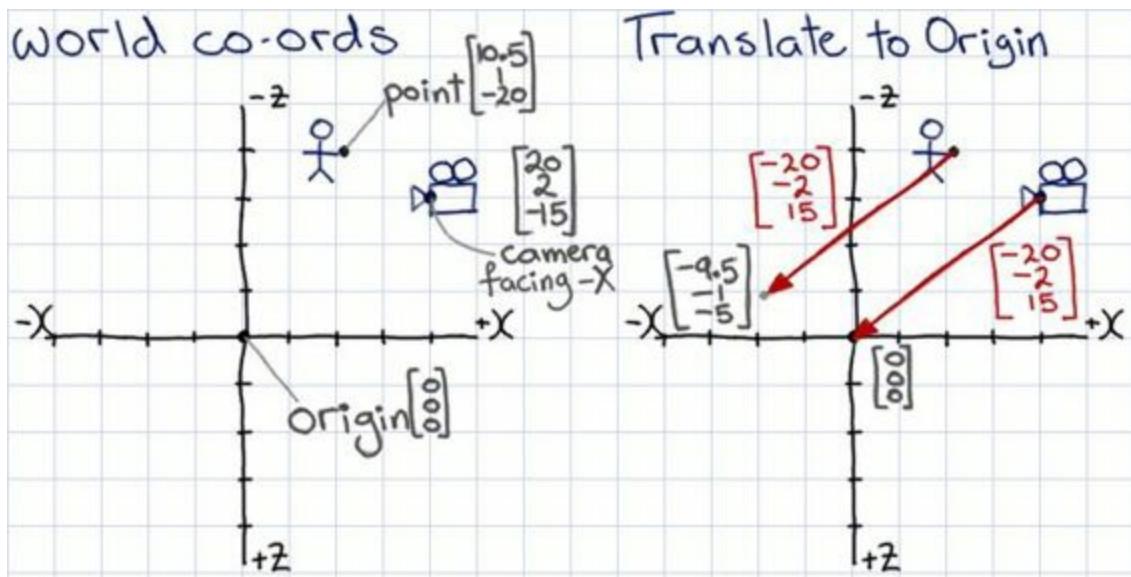
We typically manipulate the transformation pipeline in the vertex shader. When rendering each vertex of a mesh we position and orient it in the world by multiplying it with a model matrix, then transform it relative to the location and orientation of the camera by multiplying it with the view matrix, then add some depth perception by multiplying it again with the perspective matrix. We usually output it from the vertex shader at this point, and the remaining steps are done automatically.

Local Space to World Space

In the image above, the first 2 boxes describe our transition from local coordinates to world coordinates. The example uses a character mesh, rather than a triangle, and it has a local origin at the base of the feet. There is a vertex point on one of the hands with value [0.5, 1, 0]. All of the points are multiplied by a model matrix, which moves our character into the world (relative to the world origin). In the second box, you can see that the point on the hand now has the value of [10.5, 1, -20, 1]. This means that it was translated by the model matrix by distance [10, 0, -20]. And, the 1 on the end tells us that we used a 4X4 homogeneous matrix to do this - remember that the 1 means "this is a point that should be translated".

World Space to Eye Space

Now imagine that we want to view the world from the side, rather than straight-on, as we have been doing. This is the first part of our virtual camera idea. What do we need to do to achieve this? We need to give the camera a position and orientation in world coordinates.



A break-down of the transition between world space and eye space. These steps are usually combined into a single view matrix, often by using a `lookAt()` style of function.

In the diagram, the camera has a world position of [20, 2, -15]. In world coordinates, the camera is oriented 90 degrees on the Y axis. According to the camera, in eye coordinates, its own forward direction is the -Z axis, and the point on the character is 5 units to the right, 1 unit below, and 9.5 units ahead, or [5, -1, -9.5]. To map between these two spaces you can see that we just need to subtract the camera's world position, giving us [-9.5, -1, -5], and rotate in the opposite direction (-90 degrees around the y-axis), giving us [5,

-1, -9.5].

We can combine these operations into a **view matrix**, and write a function replicating the old OpenGL `lookAt()`, where we give it the camera position, a target to look at, and the "up" direction, which tells us how the camera should be pitched or rolled. Most maths libraries for graphics will have a function that does this.

$$V = \begin{bmatrix} R_x & R_y & R_z & -P_x \\ U_x & U_y & U_z & -P_y \\ F_x & F_y & F_z & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Column-Major View Matrix

Where U is a unit vector pointing up-wards, F forwards, R right, and P is the position of the camera, all in world coordinates. The U vector must change as the camera pitches forwards and back, or rolls to either side.

Now, to construct a matrix combining rotations and a translations, we generally can not simply plug the rotation and translation components into a single matrix. It's okay to do this in the special case where all of the rotations come before the translations though - try printing the resulting matrix for $M = T * R$ and $M = R * T$ to see the difference. In general it's a good idea to create separate matrices, and multiply them together to combine them.

To combine our 2 matrices into a final view matrix, in column-major layout, we do not say $V = T \times R$, because this would rotate the points around the world origin, and then move them such that the camera position is at the origin - remember that the order of operations in column-major notation is right-to-left. We want to translate such that the camera is at the origin, and then rotate them around the camera, so we say $V = R \times T$. In row-major layout, this is the other way around, of course. The example below shows how to do this for a bird's eye (overhead) view camera.

$$\begin{array}{c}
 \text{Orientation} \quad \text{Position} \\
 \hline
 R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 \hline
 V = R \times T \\
 \text{Bird's Eye View Matrix}
 \end{array}$$

An example of creating a bird's eye view matrix. The forward vector points -Y, and is negated to +Y, the up vector now points -Z, and the position will need to be set every time that the camera moves.

lookAt()

Older OpenGL implementations had a built-in viewing transform, so we didn't need to create a matrix manually. A common function for building the transform was `gluLookAt()` which took a camera position called `eye`, a target to look at called `center`, and a vector pointing upwards, `up`, which determined the roll of the camera.

If you would like to construct your own `lookAt (camera_position, target_position, up_direction)` function, read on.

First construct a translation matrix T , remembering that the translation is the **negated** world position of the camera.

Next work out the 3d distance \mathbf{d} , between the view target and the camera by subtracting the camera position from the target position:

$$\mathbf{d} = [\text{target x} - \text{camera x}, \text{target y} - \text{camera y}, \text{target z} - \text{camera z}]$$

If we **normalise** this distance vector, we turn it into our forwards direction vector with a length of 1 unit. To normalise a vector, divide each component

by the magnitude of the vector. The magnitude is:

$$m = \sqrt{x * x + y * y + z * z}$$
$$f = [x / m, y / m, z / m]$$

Note that you **should never normalise a 4d vector** because the w component (1 or 0) isn't really part of the geometric vector - we just use it to differentiate between points and directions.

Our right-pointing direction vector \hat{r} is the vector cross product of the forward vector and the upwards-pointing vector, \hat{u} . You can find the formula for the cross product on my 3d maths cheat sheet if you need it.

$$\hat{r} = f \times \hat{u}$$

We re-calculate the up vector for consistency:

$$\hat{u} = \hat{r} \times f$$

And we can now plug the right, up, and negated forwards vectors into a new rotation matrix, R . The bottom row and right-most column should be zeros, except the bottom-right cell, which should be 1.

We can now combine our two matrices together to form a single view matrix.

Note: do not forget to update the "up" input parameter as your camera pitches and rolls. If you do you can warp the perspective slightly, or if you let the forward vector become equal to the up vector, then you lose the ability to rotate sensibly around the 3 axes.

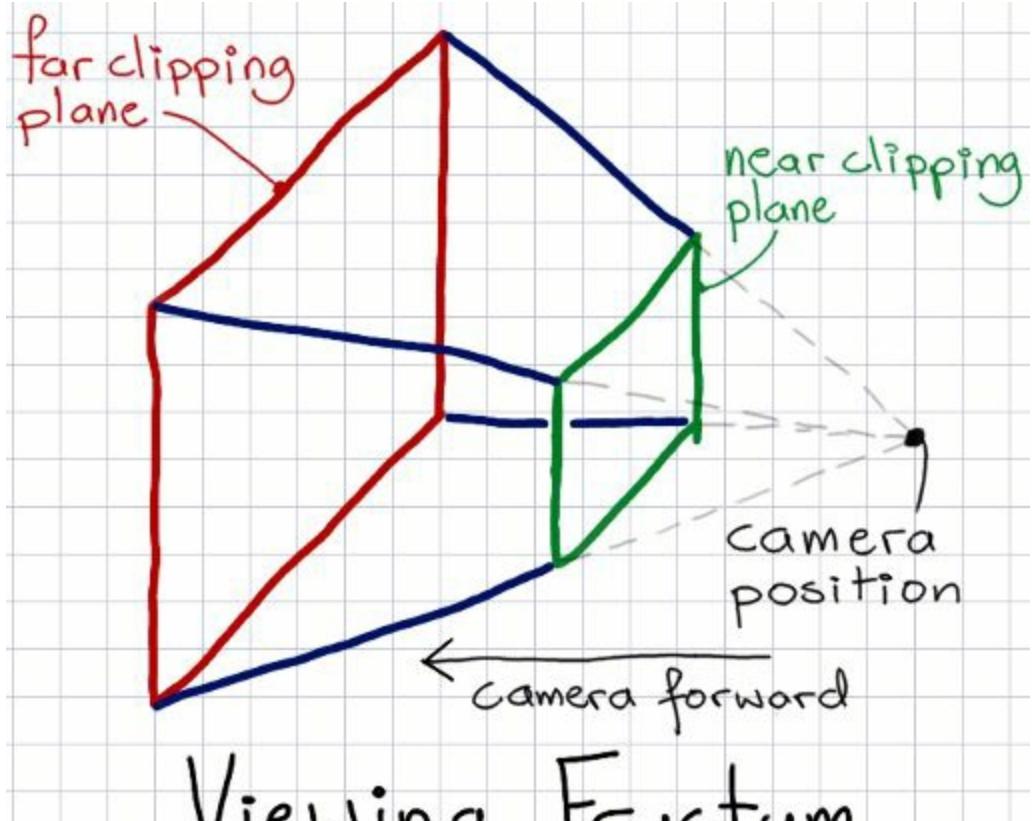
Eye Space to Homogeneous Clip Space

If the first part of our virtual camera is the view matrix, then the second part is the **projection matrix**. The projection matrix does 3 things;

1. Define **near** and **far clipping planes** - this defines the visible range along the Z axis. Anything in front of the "near" distance or behind the

"far" distance is out of the visible range. Geometry outside this range will be "**clipped**" - removed from the hardware pipeline in a later stage.

2. Define a **field of view** (FOV). This is actually not a field at all, it's an angle. You might remember that games like Quake used a 90 degree **horizontal** field of view on the standard 4:3 displays. Our field of view is actually for the vertical range, so it's going to be a bit smaller - most modern games use 66 or 67 degrees for this. The horizontal angle will be calculated by our matrix when we give it the **aspect ratio** of the viewport - how wide compared to how high. Note that on a 4:3 display $67 * 4/3 = 89.33$ degrees.
3. With the clipping planes, and the angle of view, we have created a volume - the shape of a pyramid with the top chopped off. This volume is called the **viewing frustum**. Any geometry outside the frustum will be out of the visible range. Interestingly, there is more area visible in the end than in the front. How can we represent that on the 2d screen?
4. The last job is to tell the next stage of the hardware pipeline how to calculate perspective. The fat ends of the frustum will be pushed together into a box shape - the large collection of things visible at wide end will be squeezed together towards the middle of the screen. This gives us our perspective. Imagine looking along railway tracks and observing as the rails become smaller, and closer together in the distance.



Viewing Frustum

The frustum covers a volume of the scene, and will be compressed into a box to give us perspective. The angle up and down is determined by the **field of view** given, the position along the Z axis of the near and far planes by a distance each, and the width and height of the near plane by the **aspect ratio** of the view.

The aspect ratio is the width of the viewport divided by the height of the viewport. Near and far clipping distances are just eye coordinate distances in the direction of the camera. Typical values are 0.1 for near and 100.0 for far. If your units are in meters then this means that you can see for 99.9 meters. You can not set the near clip distance to 0. In practise, you want to make the planes as close together as possible without ruining the scene, as it will give you more numerical accuracy and reduce artifacts (errors) with several effects that use depth comparison.

$$P = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & P_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Projection Matrix
(column-major)

Note that most of what the projection matrix does is scaling (the main diagonal is used), but taking field [angle] and aspect of the view into account. The P_z component adjusts points' range between clipping planes, and the -1 on the bottom row swaps the vector's fourth (w) component, with its negated third (z) component, which will later be used for perspective division.

The image above gives us the layout of a projection matrix, where:

$$S_x = (2 * \text{near}) / (\text{range} * \text{aspect} + \text{range} * \text{aspect})$$

$$S_y = \text{near} / \text{range}$$

$$S_z = -(far + near) / (far - near)$$

$$P_z = -(2 * far * near) / (far - near)$$

$$\text{range} = \tan(\text{fov} * 0.5) * \text{near}$$

Note that the input field of view to the matrix is actually the angle from the horizon to the top of the frustum, so we divide our full fov by 2 here.

Additional Stages in the Transformation Pipeline

We now have all we need to create a virtual camera that can move through a scene. We can optionally use geometry shaders and tessellation shaders here to further modify our geometry (see Hardware Pipeline diagram, earlier), but we will come back to these in later chapters. After the vertex/geometry/tessellation shaders finish there are a few additional transformation stages. These are fixed function stages - we do not write any code to define them; they will be computed automatically. Why do we need

to know about these extra steps? When we go in reverse, of course. For example, we might want to click with the mouse cursor, and determine if it intersects with an object in world space.

Perspective Division

Perspective division is not generally explained very well, so I think it's worth doing here, because it's actually quite simple. You may have heard that it is "divide by w", and been left scratching your head because didn't we make w equal to 1? Yes we did, but something has happened to it after multiplying it by the projection matrix.

If you look at the bottom row of the projection matrix you see it has (0, 0, -1, 0). None of our other 4X4 matrices had anything in the bottom row, except the 1 in the corner - we just used it to indicate if it was a point or a direction vector. Now we see a second use of the fourth vector component, and the 4X4 matrix. Recall the matrix*vector function:

Matrix * Vector

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mw + ny + oz + pw \end{bmatrix}$$

Our matrix' bottom row is 0, 0, -1, 0. This means that multiplication gives us $0x + 0y + -1 * z + 0 * w$. This has the effect of making the fourth component equal to -z. A positive version of the **depth**.

After the vertex shader finishes, the hardware pipeline takes our point, and computes a perspective division. Perspective division is:

$$v' = [x / w, y / w, z / w];$$

We are dividing the x, y, and z values by a positive depth value. The larger

the depth, the smaller the x and y values will become, pushing them closer to the centre of the view, and creating our railway tracks illusion. Because our projection matrix scaled our x and y values into a range +- the range of between the clipping planes, then this step will also scale our x,y,z coordinates into a range where -1:1 are the boundaries of the view - it has changed from a frustum shape to a cube.

You can try this out - load up a working demo, and change the value of `g1_Position`. So far, we have left the w component as 1.0, which will have no effect (divide by 1). Try making it smaller - perhaps 0.5. Your geometry should stretch out, giving it a "zoomed in" look. Make it 2.0 and your shape should be squished down, making it look farther away.

NDS

The diagram at the beginning of this article gives us 2 additional stages that happen after clip space. We already know about perspective division. The points are then normalised into a box shape with dimensions [-1 to 1, -1 to 1, -1 to 1]. The w component is discarded. This is called **Normalised Device Space** (NDS). The dimensions of the box are unit-sized (hence the "normalised" name), so they can be multiplied to scale to any viewport dimensions.

2d

The last major step is to completely flatten the normalised box, so that the objects farther away are drawn behind objects closer to the camera. The dimensions need to be scaled from the normalised box X and Y to the actual pixels of the 2d viewport size. We now have a 2d area called **Viewport Space**.

Don't Mix Up Coordinate Systems

You can not compare vectors from different coordinate spaces. It is not valid to get the distance between a point in eye space, and a point in world space. Raise the world space point to eye space first. This is a very common

source of error in shader programmes. The best way to avoid this is to postfix or prefix variable names with the space that they are in. For example:

```
vec4  
point_eye = view_matrix * point_world;.
```

Reversing

You can reverse any transformation by multiplying a point by the inverse of a matrix. That is, to go from eye space to world space, you can multiply a point in eye space by the inverse view matrix. You can work your way all the way back down the pipeline from viewport space. We do this for things like casting a ray by clicking with the mouse. There is a GLSL function `inverse()` to do this, but generally we would compute the inverses in C, so we would need an inverse function in our matrix maths library.

Virtual Camera Demo with Keyboard Controls

Start with a minimal demo. You can build on to an existing one, but it will look a bit confusing if your triangle is moving as well as the camera. First, let's define a speed of movement for the camera, and a speed that it will rotate at.

```
float cam_speed = 1.0f; // 1 unit per second  
float cam_yaw_speed = 10.0f; // 10 degrees per second
```

We also need to keep track of the world position and world orientation of the camera. I'm going to simplify this demo by saying that the camera can only rotate around the Y axis. I'm being very careful not to start my camera at world position (0,0,0). This is because our triangle shape is at Z position 0, and we will have a near cut-off distance, so better to start the camera back a bit, or the shape will be clipped from view.

Camera Variables

```
float cam_pos[] = {0.0f, 0.0f, 2.0f}; // don't start at zero, or we will be too close  
float cam_yaw = 0.0f; // y-rotation in degrees
```

I'm going to create an initial view matrix. At this point I used my little maths library ; **maths_funcs.h**, **maths_funcs.cpp**, but if you have your own matrix * matrix code, and you are happy making a Y rotation matrix and a translation matrix, then you can use that instead. The matrices here are just 1d arrays of floats. I could use a `lookAt()` style of function, but in this case I'm going to manually create a translation matrix, and a rotation matrix, and combine them together to build my view matrix.

```
mat4 T = translate (identity_mat4 (), vec3 (-cam_pos[0], -cam_pos[1], -cam_pos[2]));  
mat4 R = rotate_y_deg (identity_mat4 (), -cam_yaw);  
mat4 view_mat = R * T;
```

Remember that we are negating both the camera position, and its orientation to build this matrix. This is for column-order matrices. Row-order has

multiplication around the other way. Don't mix up the order of multiplication here, or you'll get weird results when you start moving and rotating the camera.

Next we can set up a projection matrix. We need to define some input variables for that, and work out the values that get plugged-in to the matrix. You would usually use a maths library `projection()` function to do this, but I think it's worth doing by hand here as an example. Keep in mind that the tangent maths function uses radians, not degrees, so we need to do a conversion here. I made a pre-processor definition to do that, so I don't have to work it out every time. When I calculate the aspect ratio, I make sure to cast the view dimensions as `floats`, because we don't want to do integer division here.

```
#define ONE_DEG_IN_RAD (2.0 * M_PI) / 360.0 // 0.017444444
// input variables
float near = 0.1f; // clipping plane
float far = 100.0f; // clipping plane
float fov = 67.0f * ONE_DEG_IN_RAD; // convert 67 degrees to radians
float aspect = (float)width / (float)height; // aspect ratio
// matrix components
float range = tan (fov * 0.5f) * near;
float Sx = (2.0f * near) / (range * aspect + range * aspect);
float Sy = near / range;
float Sz = -(far + near) / (far - near);
float Pz = -(2.0f * far * near) / (far - near);
```

Now we can plug our values in to an array. Remember that it's going to look a bit funny because the array indices are counting down each column:

```
float proj_mat[] = {
    Sx, 0.0f, 0.0f, 0.0f,
    0.0f, Sy, 0.0f, 0.0f,
    0.0f, 0.0f, Sz, -1.0f,
    0.0f, 0.0f, Pz, 0.0f
};
```

Modify Vertex Shader

Add uniforms to use the 2 new matrices:

```
uniform mat4 view, proj;
```

Then multiply your vertex point by the projection and view matrices. The

order here is very important. Once again, this is column-major multiplication. Row order is reversed.

```
gl_Position = proj * view * vec4 (vertex_position, 1.0);
```

If you are also using a model matrix to move your geometry, then it should look like this:

```
gl_Position = proj * view * model * vec4 (vertex_position, 1.0);
```

Where `model` should be replaced with the name of your existing model matrix.

Update Uniforms

By default, matrix uniforms are zero in the shaders. We don't want that, it will kill our geometry. Let's initialise the uniforms to our initial matrix values. Note that I was using my maths library for the view matrix. The `view_mat.m` there is retrieving the array of floats from the `mat4` structure that I made. The projection matrix was just an array of floats, so no need to do that there.

```
int view_mat_location = glGetUniformLocation (shader_programme, "view");
glUseProgram (shader_programme);
glUniformMatrix4fv (view_mat_location, 1, GL_FALSE, view_mat.m);
int proj_mat_location = glGetUniformLocation (shader_programme, "proj");
glUseProgram (shader_programme);
glUniformMatrix4fv (proj_mat_location, 1, GL_FALSE, proj_mat);
```

Keep track of the location of each matrix. We won't need to change the projection matrix unless the view is resized. In a larger programme, you should use GLFW's window resize call-back to tell you when the view has changed size, and recalculate your projection matrix. If you forget to do this the view will become warped when the window resizes.

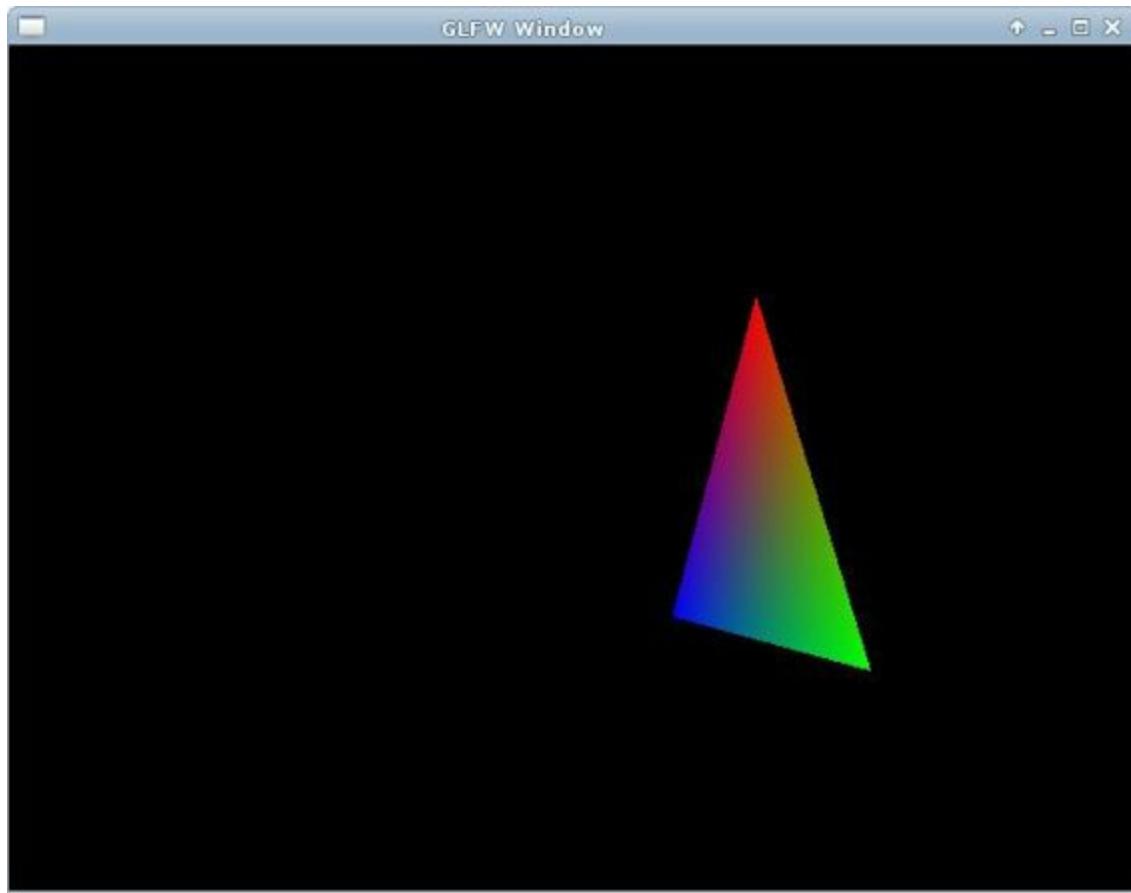
You can test it now. It should look very much the same as before, because we have not moved or rotated the camera. If you do not see a triangle any more, you can remove either the projection or view matrix from the `gl_Position = proj * view * vec4 (vertex_position, 1.0);` line in the vertex shader to isolate which matrix is the problem.

Keyboard Controls

We can use GLFW's keyboard manager to see if a button was pressed. I am going to use WASD for moving forwards, backwards, and sliding left and right. I will use the left and right arrow keys to turn around. It's not going to change the direction of movement as you turn, but you could calculate this too if you like. I added pgup/pgdn for moving vertically. I don't recalculate my view matrix straight away, but I raise a flag. This is just a small optimisation in case more than 1 button is pressed. This code should go inside the main loop. I assume that you have a counter that records the time elapsed since the last frame in `elapsed_seconds`. GLFW has a very accurate `glfwGetTime()` function to do this.

```
// control keys
bool cam_moved = false;
if (glfwGetKey (window, GLFW_KEY_A)) {
    cam_pos[0] -= cam_speed * elapsed_seconds;
    cam_moved = true;
}
if (glfwGetKey (window, GLFW_KEY_D)) {
    cam_pos[0] += cam_speed * elapsed_seconds;
    cam_moved = true;
}
if (glfwGetKey (window, GLFW_KEY_PAGE_UP)) {
    cam_pos[1] += cam_speed * elapsed_seconds;
    cam_moved = true;
}
if (glfwGetKey (window, GLFW_KEY_PAGED_DOWN)) {
    cam_pos[1] -= cam_speed * elapsed_seconds;
    cam_moved = true;
}
if (glfwGetKey (window, GLFW_KEY_W)) {
    cam_pos[2] -= cam_speed * elapsed_seconds;
    cam_moved = true;
}
if (glfwGetKey (window, GLFW_KEY_S)) {
    cam_pos[2] += cam_speed * elapsed_seconds;
    cam_moved = true;
}
if (glfwGetKey (window, GLFW_KEY_LEFT)) {
    cam_yaw += cam_yaw_speed * elapsed_seconds;
    cam_moved = true;
}
if (glfwGetKey (window, GLFW_KEY_RIGHT)) {
    cam_yaw -= cam_yaw_speed * elapsed_seconds;
    cam_moved = true;
}
// update view matrix
if (cam_moved) {
    mat4 T = translate (identity_mat4 (), vec3 (-cam_pos[0], -cam_pos[1], -cam_pos[2]));
    // cam translation
    mat4 R = rotate_y_deg (identity_mat4 (), -cam_yaw); //
```

```
mat4 view_mat = R * T;  
glUniformMatrix4fv (view_mat_location, 1, GL_FALSE, view_mat.m);  
}
```



Common Errors

3d programming tip: The first thing to do is draw a diagram of your scene on paper. Where is the camera? Where is the geometry? Is the camera in front or behind the geometry? How big is the geometry? Is it pointing the right way? Then you can compare your hand-calculations with what you wrote in the code. This will help you spot 9/10 mistakes in 3d programming. If you can't draw the problem, yet you've started coding then you need to go back to the theory.

The most common mistakes from the lab:

- Mysteriously doesn't draw properly - or a white shape is drawn. Shader didn't link properly - check shader logs. Check if your `glGetUniformLocation()` calls returned less than 0.
- Error in matrix layout. Perhaps the wrong cell was used, or a small typo. Test by substituting your code with a function from a library.
- Camera is too close to the object (i.e. both at Z position 0). Remember the near plane will cut off anything in front of a certain distance. If your shape is at 0, try starting the camera at 2.
- Camera is pointing backwards. Check your -Z and Z values.
- Camera is above or below or too far either side of object. Check camera position versus extents of object.
- Matrix rotations in wrong order. There are several matrix rotations. Check the projection, model, view in the shader, as well as the view matrix in C.
- Object is front/back face culling and camera is looking at it from the culled side. Disable face culling to test.
- Shader code error. Check logs and run shader validation.
- Mix up between degrees and radians.
- Matrix layout given manually, but in wrong column/row order.
- Screen aspect ratio is not using actual view dimensions. Has your view changed? Use values updated from the resize callback.
- Field of view angle is too big (no shape) or too small (shape covers entire screen). This should be the angle from horizontal to the top of the

display (usually 67 degrees). Test with 67 degrees first.

- Upwards vector not correct in view matrix calculation. If camera pitches or rolls, this should be re-calculated using vector cross products. If your up vector is wrong the camera may spin on the wrong axis when you rotate.
- Have you attempted to write a view matrix by plugging in the position and rotations directly? Naughty! This will warp the view. Calculate the rotation and translation separately, then multiply them together.

Going Further

Keyboard Call-Back

GLFW has a keyboard call-back which presents a tidier interface than checking every key in the main loop.

Mouse Controls

You could also use the mouse feedback functions from GLFW to rotate the camera. This isn't going to be suitable if you plan to use the mouse cursor to click on things, but would suit some programmes.

Optimisation

The clipping process can save quite a lot of GPU processing time by removing geometry from the hardware pipeline. This reduces the number of fragment shaders that need to run. But, notice that the vertices outside the frustum have all been calculated already. Various algorithms exist to remove out-of-scene geometry before rendering. You might investigate **scene graphs**, **quad-trees**, **oct-trees**, (which partition the world into a hierarchy of geometry), and **frustum culling**, which calculates collisions between a frustum-shaped box, and shapes approximating the geometry. Shapes that do not collide with the frustum-shape are **culled** (not drawn).

Quaternion Quick-Start

Introduction

You've probably tried to do a more complex rotation for a free-moving camera, a bouncing ball, or an aircraft, and **got really frustrated with how hard it is** to contrive an "any axis" rotation using a combination of rotation matrices. The Internet forums have then suggested "quaternions", and you've gone to Wikipedia, or a game maths textbook, and been **utterly horrified**, and closed it immediately. There's a lot of general half-information around on how to actually use them for rotation, which can be a bit frustrating when you're looking for somewhere to start. **I have read all of the boring articles for you**, and present here a quaternion quick-start guide (without any complex number theory background).

I hope that this is a somewhat helpful guide to getting started with using quaternions. If you're using a maths library then you barely have to do anything at all - it's just hard to get a decent introduction into the steps involved in using quaternions for rotation;

1. describe a rotation by making up an arbitrary axis and an angle around that axis
2. generate a quaternion from the angle and the axis
3. multiply or interpolate quaternions to get combined or in-between rotations, respectively
4. normalise the quaternion
5. convert the quaternion to a matrix

The first problem that I find with quaternions, is that they are usually explained by hard-core mathematicians, who have a total inability (as a species) of conveying a small amount of information in a simple way, and tend to start with extended proofs (which we don't usually care about), and end there without any human-readable explanation (yes, I just implied that mathematicians are inhuman). The second point of confusion is the old-

fashioned notation that is used " $q = q_0 + iq_1 + jq_2 + kq_3$ ". The plus symbols can be a bit confusing, until you find out that it can be expressed in more familiar 4d vector. Most of the articles that I've read actually attempt to draw something symbolising a 4-dimensional hypersphere, where "the radius means the angle" or something abstract - what?! I find this sort of thing totally un-intuitive, and that this makes the topic seem more confusing than it actually is.

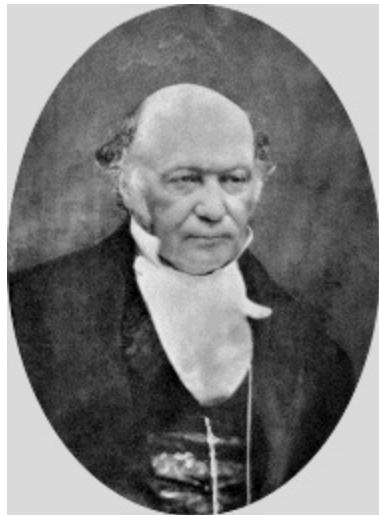
I'm going to explain how to build and use quaternions. This is going to solve the more-tricky matrix rotations in a simple way. There are some nice tools for interpolating rotations for animation problems when using quaternions, and this is where they become really useful. The only drawback is that the spherical interpolation (SLERP) can be a bit expensive, so we'll only use these things where we need them - sometimes a plain matrix or a linear interpolation will do.

The good news is that quaternions (despite being 4-dimensional geometry which **your limited human brain can't even visualise**) are very easy to code up and use. You give them an arbitrary XYZ axis, and an angle to rotate around that axis. Simple. You can then combine a whole series of them easily, without all the mess of rotation matrices. They use the 4th dimension to calculate a nice, clean rotation blend. You can then convert the resulting quaternion into a rotation matrix, and it's ready to use.

The neat things about quaternions that interest us are:

- Implementation is very elegant or simple, compared to matrices
- We can represent a rotation between any two poses with a single quaternion
- There are some nice methods for interpolating between quaternions, which is going to be very nice when we do animation with skeleton-hierarchy animation, or inverse kinematics.

A Short and Somewhat Silly Background to Quaternions



Studied where I work. People are generally afraid of trying to understand his work. Hyper-spheres and non-Euclidean geometry that warp the brain in ways that Nature never intended. But we'll see that the representations are actually much more simple and elegant than the matrices that we have been using.

The building next door to me is named for the inventor of quaternions; Sir William Rowan Hamilton. It is notable for its terrible cafeteria. In 1843 Hamilton had been working on his "quaternion" idea for a while; an extension of complex number theory, and one day on his way to the Royal Society had a stroke of genius and realised he could use the fourth dimension to multiply quaternions together, which would combine rotations in a much tidier, and more powerful way than other methods. He carved the formula into Brougham Bridge in Dublin.

$$i^2 = j^2 = k^2 = ijk = -1$$

quaternion multiplication

This is the quaternion multiplication formula that Hamilton carved into Brougham Bridge. It won't make any sense to you if you're not a mathematician, because we don't use that style of notation (vector notation didn't come about until afterwards). It does much the same thing as multiplying rotation

matrices - combines them - but in a sequential way; one rotation followed by another, which is very handy for computing freely rotating motion.

Limitations of X, Y, and Z-Axis Rotation Matrices

X, Y, and Z rotation matrices are fine for most things, but sometimes we want to do a rotation that is very difficult to construct by combining rotation matrices. You've probably noticed:

- The order of the matrices changes the result
- Rotations are always around global orientation axes, not relative to the way your object is facing
- Rotating around an axis other than X Y or Z is very difficult to work out

Make a paper aeroplane, and try rotating the following 2 examples with each matrix set to an angle of 90 degrees:

```
R = Rz * Ry * Rx;
```

```
R = Rx * Ry * Rz;
```

You get two different final orientations! So, the order of multiplication is important. Also notice that the first rotation is "local" to the object, so Rx is a "pitch the nose up", but after that the nose is no longer pointing along the Z axis, so Ry is not going to be a yaw left and right. Rats! If we wanted to combine a pitch and a yaw we would need to re-order the matrices, or do a complex stack of matrix transformations - very complicated!

But fear not! There is an easy alternative. Quaternions are just as easy to code up and use as the rotation matrices that we have used before, and can actually save a lot of hair-pulling when trying to compute more involved rotations.

No, You Didn't Have "Gimbal Lock"

The Internet seems to think "quaternions because gimbal lock"; that using rotation matrices will lead to something like gimbals locking in a gyroscope - that's not really true, as a gyroscope reads a hierarchy of local orientations, and we haven't actually computed a hierarchy of local orientation matrices yet (and there's no need to). You will get the correct result every time if you use our rotation matrices to rotate points because they don't keep track of the local orientation axes. It's just very cumbersome to compute more complex rotations. I've never encountered a gimbal lock problem in the wild.

Most of the examples of axes "losing a degree of freedom" that I've seen are actually accidental mis-use of functions like `lookAt()` that rely on a cross-product of a "forward" and "up" vector. Programmers usually set "up" to $(0, 1, 0)$ (the Y axis), but forget to change this as the camera pitch changes up and down. If the camera "forward" is the same as "up" then you get and an incorrect "right" vector as the cross product; if the camera is pointing straight up, then the "up" vector should be $(0, 0, 1)$ - you need to do a cross-product to re-calculate the "up" vector as the camera pitches up and down. This isn't a gimbal lock, this is just that the 3 axes of the co-ordinate system are not perpendicular to each other. A better solution for free-look cameras is not to use `lookAt`, but to use a quaternion rotation instead.

Steps Involved With Rotation Quaternions

A quaternion used for rotation is called a **versor**. The notation used for versors **predates vector calculus**, so it looks a bit strange, but we can rewrite it in vector notation.

$$q = q_0 + i q_1 + j q_2 + k q_3 \quad (\text{versor})$$

$\underbrace{}$

scalar (θ) x, y, z vector (axis)

Here we create a versor, q , from an angle (usually just a float), and 3 numbers that represent an axis (usually a normalised vector). Note that this is just 4 floats.

Most maths libraries for 3d will have quaternion functions, but you can also write your own pretty easily. If you want to use a library, it will be something like:

1. `quat q (angle, x, y, z);` - creates a versor which we use for calculations
2. `quat q = slerp (q1, q2, t);` - interpolate between 2 key-frames' orientations to create current orientation. Rotation is in-between $q1$ (when $t = 0$), and $q2$ (when $t = 1$)
3. produce final rotation matrix `mat4 R = q.get_matrix ();`
4. model/world matrix created as normal e.g. `mat4 M = T * R * S;`

First Step: Build a Verson

$$\hat{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right)x \\ \sin\left(\frac{\theta}{2}\right)y \\ \sin\left(\frac{\theta}{2}\right)z \end{bmatrix}$$

building
a verson

A verson is a 4d vector that is used for rotation. Create it like this. The inputs are an angle, θ , and the x , y , and z components of your axis. C and C++ \sin and \cos use radians, so you probably want to convert the angle to radians first.

We know the components of our quaternion but we can't use it in equations yet. A **verson** is the expression of the rotation in a 4d rotation vector. Making a verson is a little bit like making a rotation matrix. Now we can do various maths operations with our quaternion. I code up my versors using the formula, above, and store them in a little struct as a `float[4]`. Note the "hat" on the q - this implies that we have a unit vector (**a unit quaternion**). We'll have a look at this again shortly.

When Ready: Quaternion to Matrix

In 3d graphics we want to rotate vectors using a matrix, so we'll convert our quaternions to matrices first.

$$q = [w, (x, y, z)] \text{ where } w^2 + x^2 + y^2 + z^2 = 1$$

$$M = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy & -2wz & 2xz + 2wy & 0 \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx & 0 & 0 \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

This is how to convert a versor into a rotation matrix. You can write this as a small function. Note: w, x, y, z here are the versor components q_0, q_1, q_2, q_3 ; not the original input axis and angle. I didn't use q_0, q_1, q_2, q_3 here because it would be confusing to read. If you're using row-major notation you'll need to transpose this matrix.

This function is the last step (we combine the rotations with multiplication or interpolation whilst still in quaternion form), but I think we can code it now, so we can test that our versor creation worked. Floating point errors can accumulate when you use your quaternion, so it's a good idea to check that it's still a unit quaternion first, before converting it to a matrix. You can do by squaring each component of the versor, adding them all together, and checking that the sum is equal to 1 (as in the top of the diagram). If so, then it is still a unit quaternion, if not then we just need to normalise it first (we'll do that shortly). Your rotation matrix is now ready to use.

If Required: Re-Normalise a Quaternion

You may need to re-normalise your versor before converting it to a matrix. This is almost the same as vector normalisation. It is a component-wise operation, except that we normalise all 4 components instead of 3. As mentioned earlier, we only need to re-normalise a quaternion if the bit under the square root does not add up to 1 (no point computing square roots if we don't need to).

$$q = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \quad \hat{q} = \frac{q}{\|q\|}$$
$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

normalise versor

Much like normalising a vector, we first get the magnitude ("length") of the versor by doing a square root. We then just divide the versor by its magnitude.

Combine Rotations with Quaternion Multiplication

$$t = q \times r \quad \text{quaternion multiplication}$$

$$t = \begin{bmatrix} r_0 q_0 - r_1 q_1 - r_2 q_2 - r_3 q_3 \\ r_0 q_1 + r_1 q_0 - r_2 q_3 + r_3 q_2 \\ r_0 q_2 + r_1 q_3 + r_2 q_0 - r_3 q_1 \\ r_0 q_3 - r_1 q_2 + r_2 q_1 + r_3 q_0 \end{bmatrix}$$

It looks like a matrix, but this is just a 4d vector; another versor. Very simple, just lots of combinations of the 2 input versors.

Okay, now we're up to the bizarre-looking Brougham Bridge formula from the beginning. Like matrices, quaternions are **non-commutative**. In the example above, it means "rotate r first, then rotate q ". We won't need to combine rotations very often, unless we have a hierarchy of them, as we might for skeletal animation. But what if we want to **interpolate** a rotation part-way between two key framed orientations?

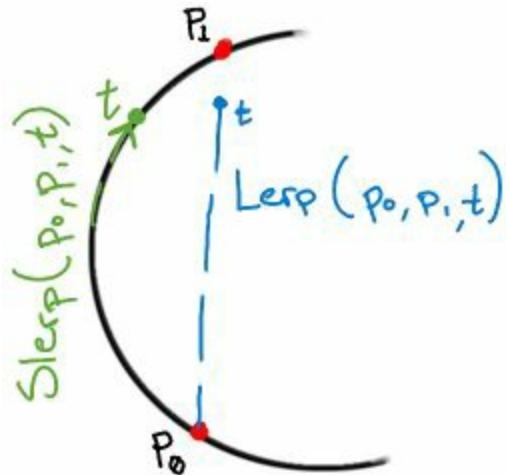
Now We Can Do Pitch/Yaw/Roll

If you maintain a "current orientation" quaternion for your aircraft, then you can use it to work out which way the "forward", "right", and "up" vectors for it are currently facing. If you want to pitch by 10 degrees, then just use the "right" vector as the axis input to a versor, and 10 degrees as your angle. If you multiply your "current orientation" by the new versor then you have your pitched result. Much less painful!

Interpolate with LERP and SLERP

Animation key-frames have position, scale, rotation keys. Rotations tend to be stored as quaternions, and scale and translations and matrices. Time between key-frames can be expressed as a factor between 0 to 1.

- scale and translation are linear changes, so we interpolate between them using linear interpolation (sometimes called LERP)
- rotations are usually circular (spherical) arcs, so we can interpolate between them with spherical linear interpolation (SLERP).



The different paths between Lerp and Slerp functions. Imagine a door opening - the end of the door should trace an arc. If you used Lerp for this, it would look like the door is shrinking and growing again in-between key-frames.

Ken Shoemake introduced this technique to graphics with "Animating Rotation with Quaternion Curves", at SIGGRAPH '85. Slerp will find the most direct path connecting the 2 quaternions, and give you a result along that based on the factor that you give it. The path is going to be a circular arc with uniform velocity (most of the time this is what you want).

$$Lerp(v_0, v_1, t) = v_0 + t(v_1 - v_0) \quad (\text{points})$$

$$Slerp(q_i, q_f, t) = q_i (q_i^{-1} q_f)^t \quad (\text{quaternions})$$

$$Slerp(p_0, p_1, t) = \frac{\sin(1-t)\Omega}{\sin(\Omega)} p_0 + \frac{\sin(t\Omega)}{\sin(\Omega)} p_1$$

$$\text{where } \Omega = \arccos(p_0 \cdot p_1)$$

A comparison of a Lerp function with 2 vectors (top), a Slerp function with 2 quaternions (middle), and the most popular method of Slerp with 2 quaternions, using a dot product (bottom).

Coding Slerp ()

The most common encoding for Slerp is the bottom formula, in the previous picture. We can first work out the dot product, and do some tests. We take the dot product of the 2 versors in the same way as we do a dot product for 2 vectors. My `versor` struct here justs hold a `float[4]` called `q`:

```
float dot (const versor& q, const versor& r) {
    return q.q[0] * r.q[0] + q.q[1] * r.q[1] + q.q[2] * r.q[2] + q.q[3] * r.q[3];
}

versor slerp (versor& q, versor& r, float t) {
    float dp = dot (q, r);
    ...

    if (fabs (dp) >= 1.0f) {
        return q;
    }
    ...
}
```

If the versors are equal then the dot product is going to be roughly equal to 1.0. If this is the case then we can skip the rest of the computation and just return the first versor, otherwise we can work out omega, and keep going.

```
...
if (fabs (dp) >= 1.0f) {
    return q;
}
...
```

We work out the $\sin(\Omega)$; part above. We can now add a test to see if the angle between the versors is 180 degrees. This means a dot product of approximately 0. If so then our slerp method would do a divide by zero (look back at the formula) and is therefore not properly defined, so we do a linear interpolation instead. Note that the method used here to get the `sin_omega` value

gives the same result as if we do `sin (acos (dp));`:

```
...
float sin_omega = sqrt (1.0f - dp * dp);
versor result;
if (fabs (sin_omega) < 0.001f) {
    for (int i = 0; i < 4; i++) {
        result.q[i] = (1.0f - t) * q.q[i] + t * r.q[i];
    }
    return result;
}
...
```

And finally we can work out Ω and the main formula. I split the sides of the formula into 2 variables called a , and b .

```
...
float omega = acos (dp);
float a = sin ((1.0f - t) * omega) / sin_omega;
float b = sin (t * omega) / sin_omega;
for (int i = 0; i < 4; i++) {
    result.q[i] = q.q[i] * a + r.q[i] * b;
}
return result;
}
```

Fix Animations That Flick Around the Wrong Way

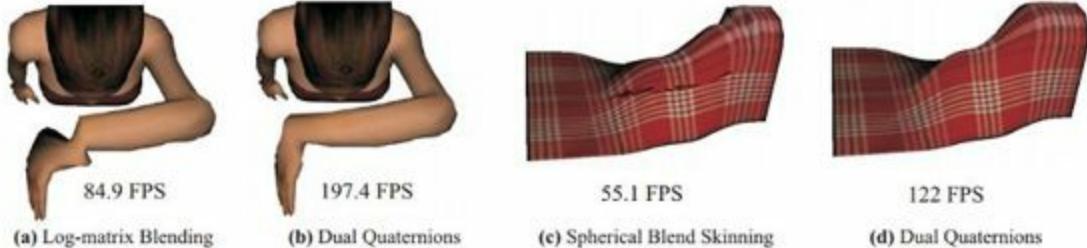
Remember that the Slerp will always take the most direct path. Sometimes this can be the opposite direction to which we intend. We see this when rotating wheels 360 degrees; you'll observe a "flick around" in the animation. To correct for this, check if the dot product of the 2 versors is negative, before calling Slerp. If it is, then negate one of the versors.

```
float dp = dot (q, r);
if (dp < 0.0f) {
    for (int i = 0; i < 4; i++) {
        q.q[i] *= -1.0f;
    }
}
versor s = slerp (q, r);
```

Alternatives to Slerp

Dual Quaternions for Rigid Transformation Blending

Ladislav Kavan*
Trinity College Dublin / CTU in Prague Steven Collins
Trinity College Dublin Carol O'Sullivan
Trinity College Dublin Jiri Zara
CTU in Prague



Slerp isn't perfect, and can be quite expensive to calculate, especially if you have lots of animated meshes in a scene. Several alternatives exist, such as this one.

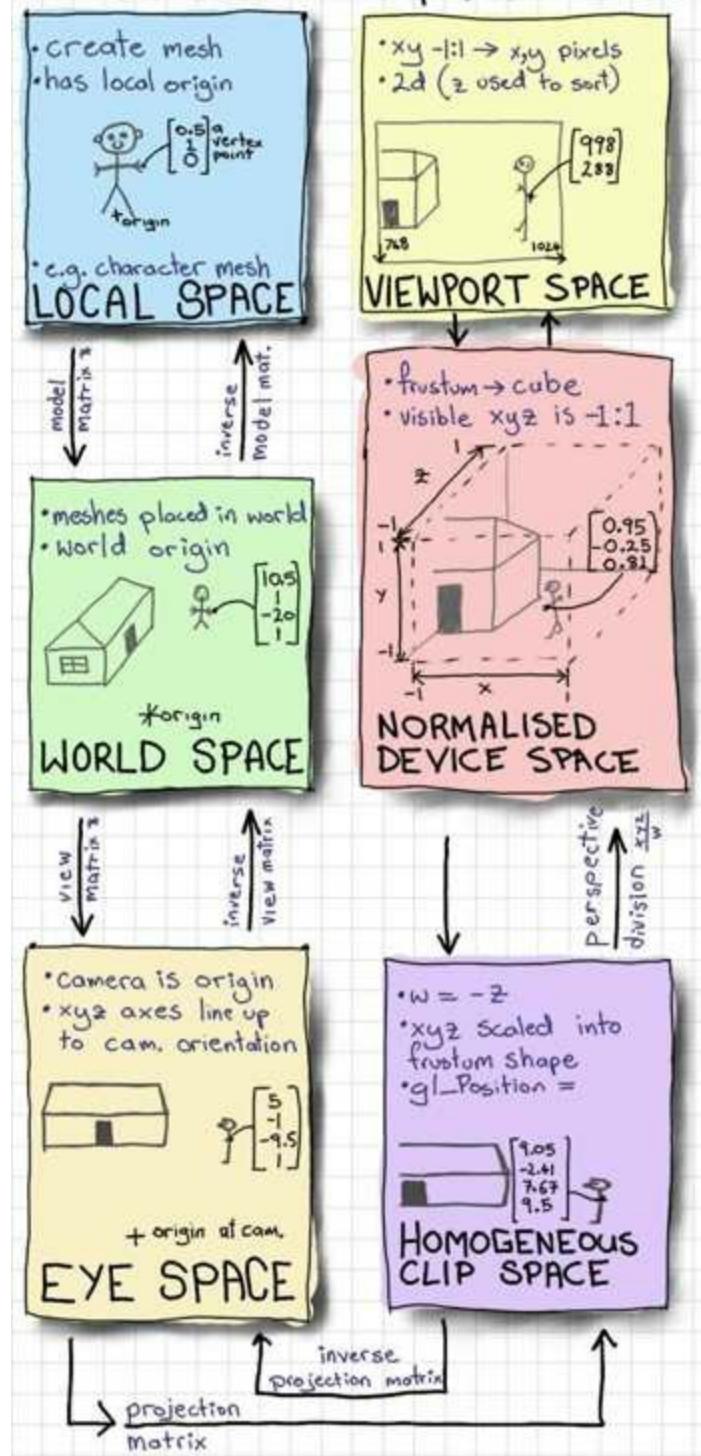
Slerp is not the only interpolation function for quaternions, and is certainly not perfect; it can be more computationally expensive than necessary, especially if you are computing Slerp for every joint in a every animated skeleton, every frame. It can also produce some "artifacts" by twisting meshes in unnatural, or unintended directions. For a good discussion of some faster alternatives to Slerp, I suggest Johnathan Blow's "Understanding Slerp, Then Not Using It", which you can find at <http://number-none.com/>. For a recent hot topic check out "Dual Quaternions".

Mouse Picking with Ray Casting

Overview

It can be useful to click on, or "pick" a 3d object in our scene using the mouse cursor. **One way of doing this** is to project a 3d ray from the mouse, through the camera, into the scene, and then check if that ray intersects with any objects. This is usually called **ray casting**. This is an entirely mathematical exercise - we don't use any OpenGL code or draw any graphics - this means that it will apply to any 3d application the same way. The mathematical subject is usually called geometric **intersection testing**.

3d Transformation Pipeline



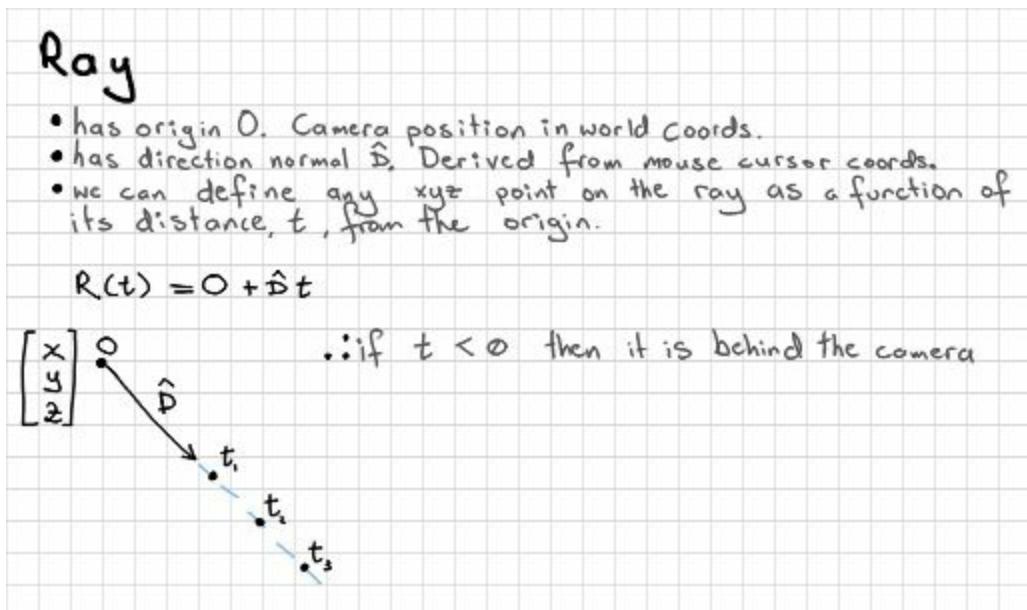
Instead of starting with a mesh in local space, we are starting with a 2d mouse cursor position in viewport space. We work backwards through the transformation by using inverse matrices, and arrive with a ray in world space.

With ray picking we usually simplify a scene into bounding spheres or boxes.

This makes the calculation a bit easier than testing all of the individual triangles. We don't need to create a 3d sphere out of triangles that can be rendered; we just represent the sphere as a simple function. The premise is that we have a mathematical formula for the points along a ray, and a mathematical formula for the points on a sphere. If we substitute the points on the sphere with the equation for points on a ray, then we get the intersection of points that are common to both. It's interesting to do this technique now because it shows us how we can use the transformation pipeline in reverse; from 2d screen to a 3d world space by using the inverse of our matrices e.g ...`inverse (view_matrix) * inverse (projection_matrix)`.... In a later tutorial we will look at an alternative technique using unique colours to determine where the mouse is hovering or clicking.

Calculating a Ray from the Mouse

All ray casting starts with a ray. In this case it has an origin \mathbf{o} at the position of the camera. We can do ray intersections in any space (world, eye, etc.), but everything needs to be in the same space - let's assume that we are doing our calculations in world space. This means that our ray origin is going to be the world x, y, z position of the camera.



This function expresses all the points, t , along an infinite ray projected from our camera location, O , in the direction D , which we will work out by modifying and normalising the mouse cursor position.

Step 0: 2d Viewport Coordinates

`range [0:width, height:0]`

We are starting with mouse cursor coordinates. These are 2d, and in the **viewport coordinate** system. First we need to get the mouse x,y pixel coordinates. You might have set up a call-back function (with e.g. GLFW or GLUT) something like this:

```
void mouse_click_callback (int b, int s, int mouse_x, int mouse_y);
```

This gives us an x in the range of `0:width` and y from `height:0`. Remember that 0 is at the top of the screen here, so the y-axis direction is opposed to that in other coordinate systems.

Step 1: 3d Normalised Device Coordinates

range [-1:1, -1:1, -1:1]

The next step is to transform it into 3d **normalised device coordinates**. This should be in the ranges of x [-1:1] y [-1:1] and z [-1:1]. We have an x and y already, so we scale their range, and reverse the direction of y.

```
float x = (2.0f * mouse_x) / width - 1.0f;
float y = 1.0f - (2.0f * mouse_y) / height;
float z = 1.0f;
vec3 ray_nds = vec3 (x, y, z);
```

We don't actually need to specify a z yet, but I put one in (for the craic).

Step 2: 4d Homogeneous Clip Coordinates

range [-1:1, -1:1, -1:1, -1:1]

We want our ray's z to point forwards - this is usually the negative z direction in OpenGL style. We can add a w, just so that we have a 4d vector.

```
vec4 ray_clip = vec4 (ray_nds.xy, -1.0, 1.0);
```

Note: we do not need to reverse perspective division here because this is a **ray** with no intrinsic depth. Other tutorials on ray-casting will, incorrectly, tell you to do this. Ignore the false prophets! We would do that only in the special case of points, for certain special effects.

Step 3: 4d Eye (Camera) Coordinates

range [-x:x, -y:y, -z:z, -w:w]

Normally, to get into clip space from eye space we multiply the vector by a projection matrix. We can go backwards by multiplying by the inverse of this matrix.

```
vec4 ray_eye = inverse (projection_matrix) * ray_clip;
```

Now, we only needed to un-project the x,y part, so let's manually set the z,w part to mean "forwards, and not a point".

```
ray_eye = vec4 (ray_eye.xy, -1.0, 0.0);
```

Step 4: 4d World Coordinates

range [-x:x, -y:y, -z:z, -w:w]

Same again, to go back another step in the transformation pipeline. Remember that we manually specified a -1 for the z component, which means that our ray isn't normalised. We should do this before we use it.

```
vec3 ray_wor = (inverse (view_matrix) * ray_eye).xyz;  
// don't forget to normalise the vector at some point  
ray_wor = normalise (ray_wor);
```

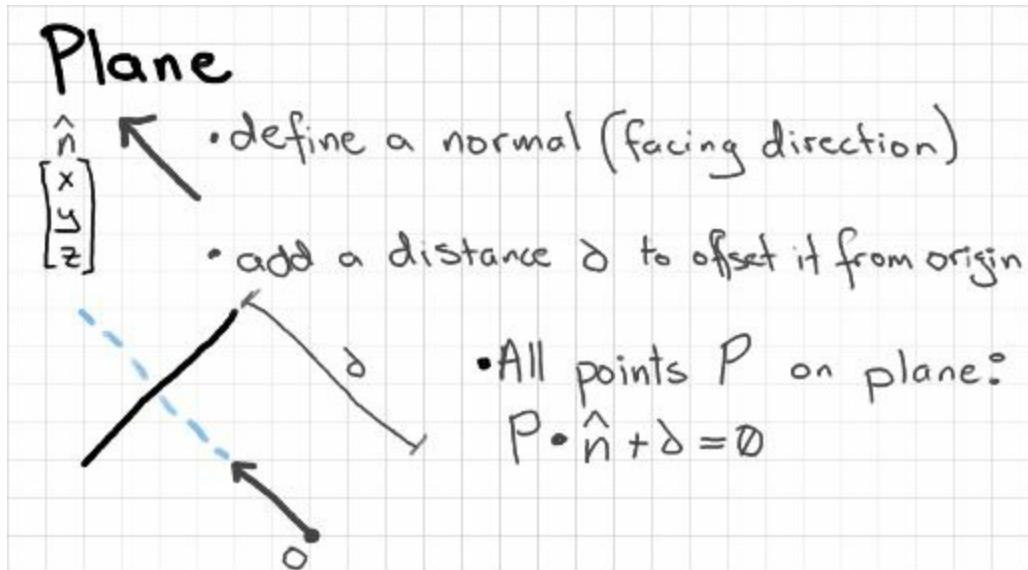
This should balance the up-and-down, left-and-right, and forwards components for us. So, assuming our camera is looking directly along the -Z world axis, we should get $[0, 0, -1]$ when the mouse is in the centre of the screen, and less significant z values when the mouse moves around the screen. This will depend on the aspect ratio, and field-of-view defined in the view and projection matrices. We now have a ray that we can compare with surfaces in world space.

Shortcut

If you're happy with this theory then you can condense all of these instructions into one line.

Ray vs. Plane

We can describe any surface as a plane. It's often useful to do this before doing more detailed (and more computationally expensive) intersection tests. We can test, for example, if a point is inside a region bounded by planes - if not, then no need to check versus individual triangles inside the region. In my case, I wanted to create an imaginary ground plane that I could use to help me click-and-drag boxes around scenery. I do 2 ray-plane intersections to get the top-left and bottom-right corners of the box in xyz world coordinates.



Ray vs. Plane

- Substitute points P on plane with points $O + \hat{d}t$ on ray:

$$(O + \hat{d}t) \cdot \hat{n} + d = 0$$

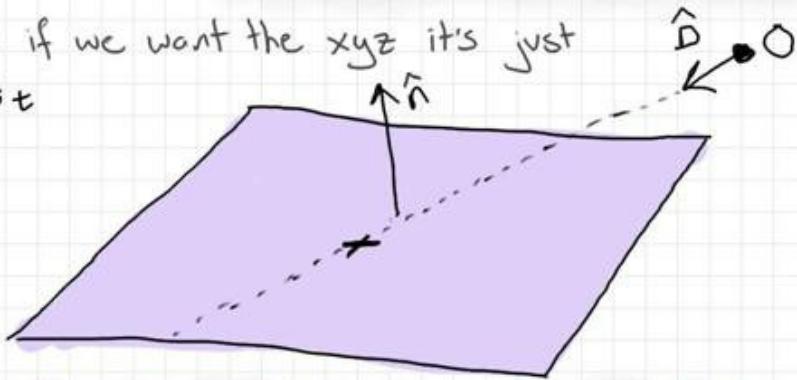
(all points on ray **and** plane)

- We just want the distance t from ray origin:

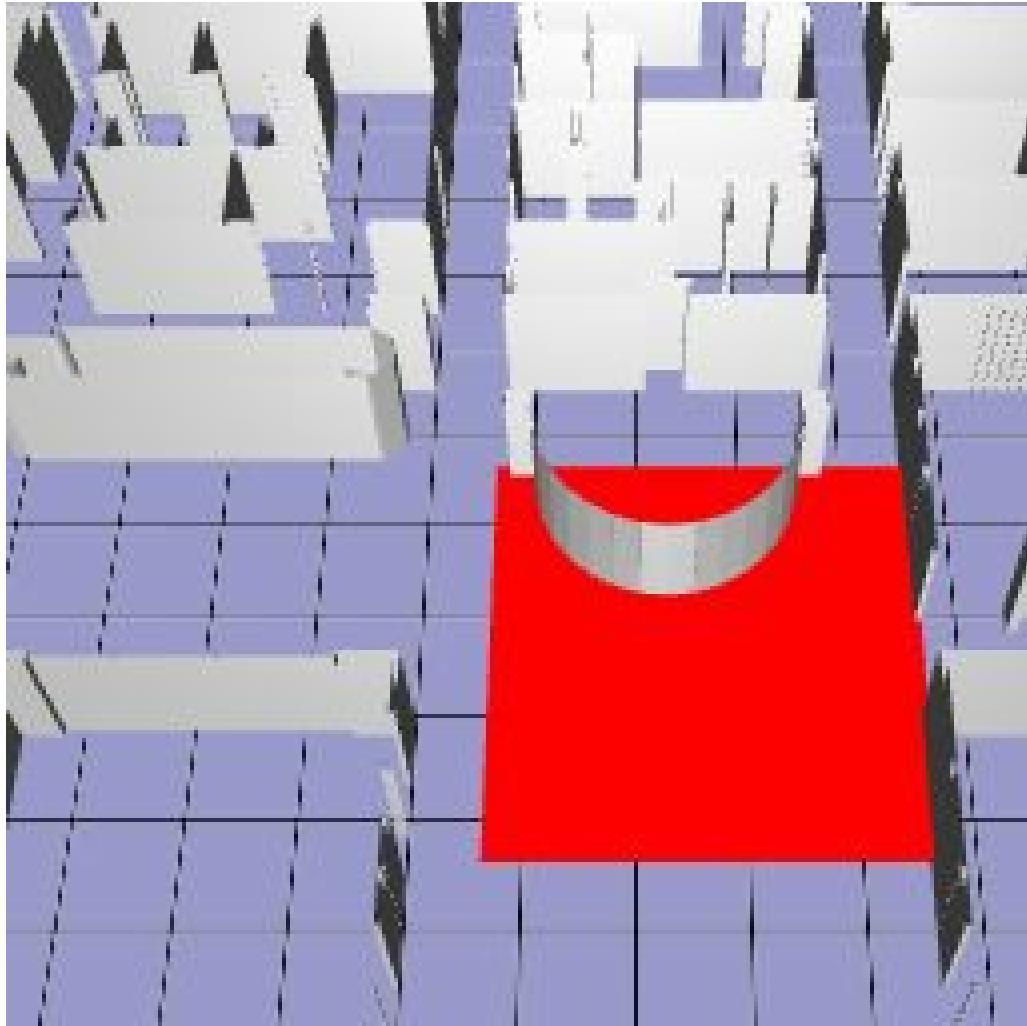
$$t = -\frac{O \cdot \hat{n} + d}{\hat{d} \cdot \hat{n}}$$
 ↗ if zero then miss! (perpendicular)

- then if we want the xyz it's just

$$O + \hat{d}t$$



*if $t < 0$ = miss (intersected behind O).



I used ray-plane intersection to allow a user to drag out a "zone" on the floor. This might be useful in strategy games where you want to select a group of units. Unless you're on a globe. In that case, read on. Note that the square is projected into depth - it isn't a true square on the 2d screen, but will be if looking from directly above it.

Ray vs. Sphere

Probably the easiest method for selecting objects in a 3d scene. If you know a bounding radius, and centre point, of each object then we have the definition of its sphere. This might not be the best method to use for large, or irregularly shaped objects, which will have very large spheres that might overlap with smaller objects nearby.

Sphere

All points 'p' on sphere:
 $\|P - C\| - r = 0$

All points on sphere and ray:
 (replace 'p' with ray equation)
 $\|O + \hat{D}t - C\| - r = 0$

This re-arranges into a quadratic:
 $t^2 + 2t b + c = 0$ } quadratics have 2 solutions
 $t = -b \pm \sqrt{b^2 - c}$ } solutions

where $b = \hat{D} \cdot (O - C)$
 $c = (O - C) \cdot (O - C) - r^2$

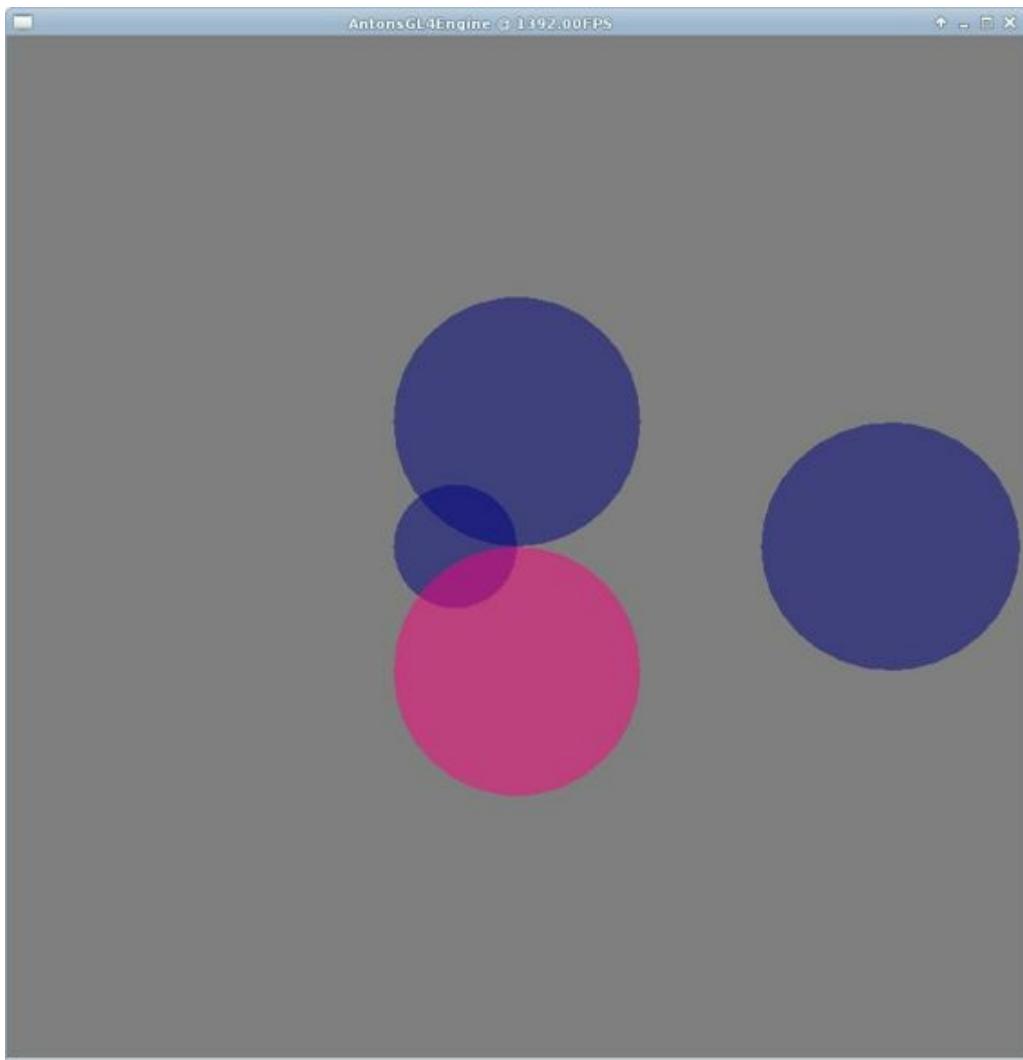
Case 1	Case 2	Case 3
$b^2 - c < 0 = \text{"Imaginary"}$ $= \text{miss}$	$b^2 - c > 0$	$b^2 - c = 0$

Here I have an equation for all the points, P on a sphere's surface, which says "anything that is radius r distance from centre point C ". The idea is to replace P with the points-along-a-ray equation, and solve for t . I skipped the re-arrangement steps, but if you're interested in a full discussion have a look at Akenine-Möller et al. "Real-Time Rendering".

We can actually get 2 output intersections from a ray-sphere intersection

because the ray can hit both the back and the front of the sphere. In this case we need to check for the closest intersection (smallest value of t). The ray can also miss the sphere, hit the very edge of the sphere (both t values are the same), or be cast from inside the sphere (one t value is negative).

Be sure to check the $b^2 - c$ term before evaluating the expensive square root operator. If you miss the sphere entirely you will not get a sensible result for t . It is also possible that the ray will be cast from in front of or inside of the sphere, so you will need to check that your t values are positive.



Here I have a collection of spheres in 3d space. I generated these in Blender and scaled them to match the radius of my bounding sphere. They are all the same size, but one is farther away and behind the others - this was to test that only the closest intersection was being selected. When a sphere is clicked on, I changed its colour to pink. This helped me determine that the ray was being calculated correctly, and the intersection was returning the correct result. You can imagine each sphere surrounding a mesh or object that is being selected.

Optimisations

You probably won't need to unless you're doing **a lot** of clicking on spheres, but you can reduce the cost of this algorithm by adding a few checks before getting to the square root. Compare squared distances between the ray origin and the sphere centre. You can project this as a 2d distance along the ray direction (dot product) and compare the end point's squared distance to the sphere origin (use Pythagoras' theorem) with the squared radius.

Round-Up

Generally, if you want to draw on a shape (e.g. drawing a click-and-drag selection box) then ray-casting is a good choice. If you want inaccuracy i.e. letting the user click in the general area of a fiddly little mesh to select it then ray-casting is a good choice. If you want accuracy then you're almost certainly better off using a colour-based picking method, which has the cost of one extra rendering pass, rather than 1-5 ray casts per selectable object in the scene.

Further Intersection Tests

You can use ray casting to test against different shapes including boxes, and even the individual triangles of a mesh, using barycentric coordinates. This sort of intersection testing become more useful for related problems of physical collisions and ray-traced rendering. Generally speaking, because ray-casting is quite computationally expensive, it's advisable to test a ray versus a bounding sphere first, before testing against more complex surfaces - no point checking against all of the individual triangles in a mesh if the ray will miss the mesh entirely!

For a more in-depth discussion of the algorithms I suggest reading Chapter 2 of the book "An Introduction to Ray Tracing", Academic Press, London, 1989. The chapter is called "Essential Ray Tracing Algorithms", and is written by Eric Haines. I can also highly recommend the textbook "Real Time Rendering", by Akenine-Möller *et al.*, as it has a whole part dedicated to different intersection tests, with lots of pseudo-code and explanations. We used this book as a reference for a class assignment on building a ray-tracer.

Phong Lighting

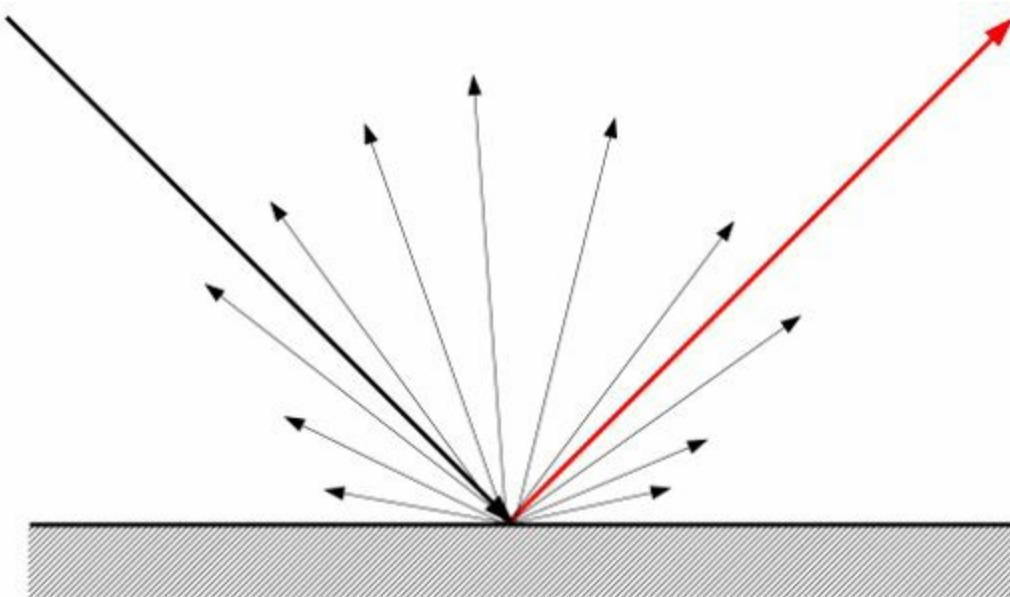
Older APIs had pre-built (fixed-function) lighting functions for different types of light; point, directional, and spot. The built-in lights were based on B.T. Phong's "Illumination for Computer Generated Pictures" (1975). These pre-sets no longer exist in newer APIs, but we can rebuild them in programmable shaders with a little bit of extra effort. This means that we have to do all of the angle calculations ourselves but it gives us more flexibility to modify the equations.

If you have never studied **Optics**; the physics of light rays, it is worth while picking up an entry-level physics text book (or Wikipedia) and familiarising yourself with the basic concepts first. You will need at least a passing understanding of **rays**, **reflection**, and **refraction**. For a bit of historical background you can browse Sir Isaac Newton's "Opticks" - still quite readable for 1704. In any case, you should get the gist that it's all about rays shooting from a light source, bouncing off various objects, and into the eye of the observer. Different surface angles cause the light rays to reflect at different angles, and some surfaces cause the light to scatter in different directions. We want to approximate this realistic lighting model in real-time rendering, which means that we are going to cheat and shortcut wherever possible.

Phong is a Sum of 3 Lighting Approximations

Phong lighting is a combination of 3 very rough reflection approximations that looks quite good when summed together. It is popular because it is much cheaper to render in dynamic scenes than the more accurate **global illumination** techniques like ray-tracing. Phong makes the following simplifications to the optics lighting model:

1. We already know which objects are visible, thanks to the transformation pipeline, so we only calculate light for those surfaces
2. Light from a source that reflects off a surface, directly into the eye of the viewer, is modelled with a **specular light** term. Surface smoothness, which factors specularity, is approximated per-surface with a **specular reflectivity** term.
3. Surface roughness (which scatters light) is not directly modelled, but we approximate this per-surface with a **diffuse reflectivity** term.
4. We don't model object-to-object reflections - we are going to assume that light only goes from the light source, directly to each object in view, and directly to the eye. Reflections hitting multiple objects (mirror-like surfaces etc.) are not modelled, but we will broadly approximate the influence of background light with an **ambient light** term.
5. Refraction is not modelled.
6. Shadows are not created.



Two of the Phong reflection models are illustrated; the large, direct arrows explain specular light, which should contribute more to the colour of the surface as the camera is in the reflected path. The scattering arrows explain diffuse light, which doesn't depend on the viewer's position, but will be more intense as the surface is perpendicular to the light source. Ambient light will always colour the the surface with the same base colour.

To calculate Phong reflection we do 3 light calculations; {diffuse light intensity, I_d , specular light intensity, I_s , ambient light intensity, I_a } per surface (usually per-fragment). Each intensity gives us an **rgb** colour. We sum them together to get a final light intensity, I , which we use as the colour of the surface:

$$I = I_s + I_d + I_a \text{ (Phong Reflection)}$$

A Virtual Light Source

Let's define a virtual light source. Let's assume a single **point source** (the simplest kind of light) which has a 3d position and shines light equally in all directions. It needs:

- a 3d world position
- an rgb colour, L_s , for its specular light colour
- an rgb colour, L_d , for its diffuse light colour
- an rgb colour, L_a , for its ambient light colour

These can all be the same colour, of course. You might like to set a constant ambient light for all of your virtual light sources, depending on the nature of your scene. That's it! You can create these 4 variables directly in your shader, but if the light needs to move or change colour then you should use **uniform** variables instead.

```
// fixed point light properties
vec3 light_position_world = vec3 (10.0, 10.0, 10.0);
vec3 Ls = vec3 (1.0, 1.0, 1.0); // white specular colour
vec3 Ld = vec3 (0.7, 0.7, 0.7); // dull white diffuse light colour
vec3 La = vec3 (0.2, 0.2, 0.2); // grey ambient colour
```

Surface Properties

Every object in your scene should have some surface properties. These properties define the factor of each lighting approximation that should be reflected by the surface. Again, these are rgb colours, which means that your object can have its own, unique, colour.

- a 3d world position
- an rgb colour, K_s , for its specular reflectance factor
- an rgb colour, K_d , for its diffuse reflectance factor
- an rgb colour, K_a , for its ambient reflectance factor
- a surface **normal**; the facing direction of the surface

A rough surface should have a very low specular reflectance factor (rough surfaces scatter light in different directions). A shiny, smooth, surface will have a high specular reflectance factor (for example, polished chrome). The diffuse reflectance factor is usually used to give the object an unique, base colour; e.g. $\{0.5, 0.0, 0.0\}$ for a dark red object. We can set the ambient reflectance factor to $\{1,1,1\}$ at this stage, but we will look at manipulating it in later articles to create some nice shading effects.

```
// surface reflectance
vec3 Ks = vec3 (1.0, 1.0, 1.0); // fully reflect specular light
vec3 Kd = vec3 (1.0, 0.5, 0.0); // orange diffuse surface reflectance
vec3 Ka = vec3 (1.0, 1.0, 1.0); // fully reflect ambient light
float specular_exponent = 100.0; // specular 'power'
```

Vertex Normals

If you are carrying-on from previous examples and using a triangle, then we can add a new vertex buffer for the normals. We will no longer use the vertex colours, so if you have them then you can replace them with normals.

```
GLfloat normals[] = {
    0.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f,
};

GLuint normals_vbo = 0;
glGenBuffers (1, &normals_vbo);
 glBindBuffer (GL_ARRAY_BUFFER, normals_vbo);
 glBufferData (GL_ARRAY_BUFFER, sizeof (normals), normals, GL_STATIC_DRAW);
```

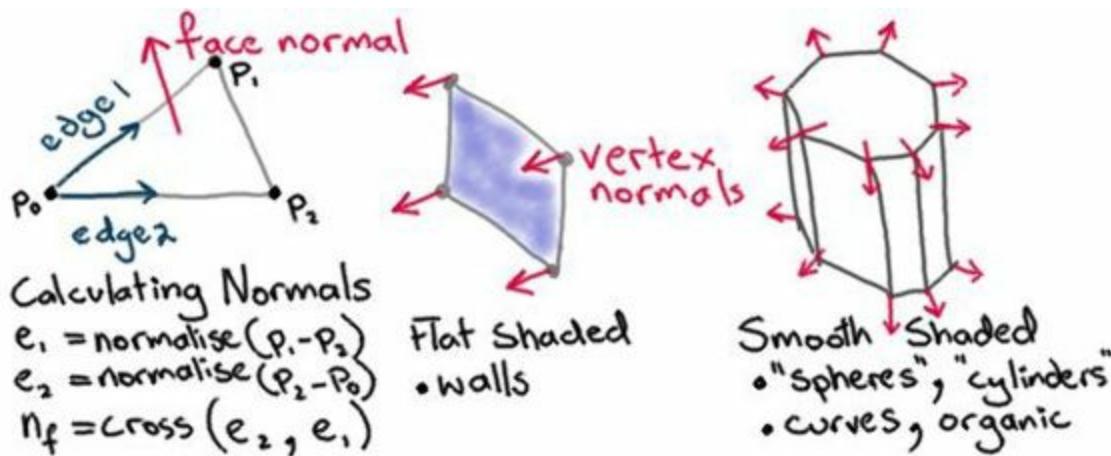
The are all 3d directions; one for each vertex in the triangle, and they all point along the Z axis. If we are not rotating the triangle then this means "towards the camera". Your vertex array set-up should now look something like this:

```
GLuint vao = 0;
 glGenVertexArrays (1, &vao);
 glBindVertexArray (vao);
 glBindBuffer (GL_ARRAY_BUFFER, points_vbo);
 glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glBindBuffer (GL_ARRAY_BUFFER, normals_vbo);
 glVertexAttribPointer (1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray (0);
 glEnableVertexAttribArray (1);
```

How We Generally Calculate/Get Normals

With more complex shapes, we can calculate a normal for each face by taking the **cross product** of two of its edge directions. To get an edge, subtract one point from another (this gives you the 3d distance between them), then normalise that. For faces that should appear flat (like a wall in a house) we should copy this normal to each of the face's vertices (so they all point the same way). For faces that should appear curved (like a segment of a sphere), then we can calculate each vertex' normal as an average of the surrounding faces' normals. This will give us the illusion of a curved surface when we apply lighting. Most 3d editing tools will give you the option of exporting smoothed or flat normals with your mesh - so we don't normally need to

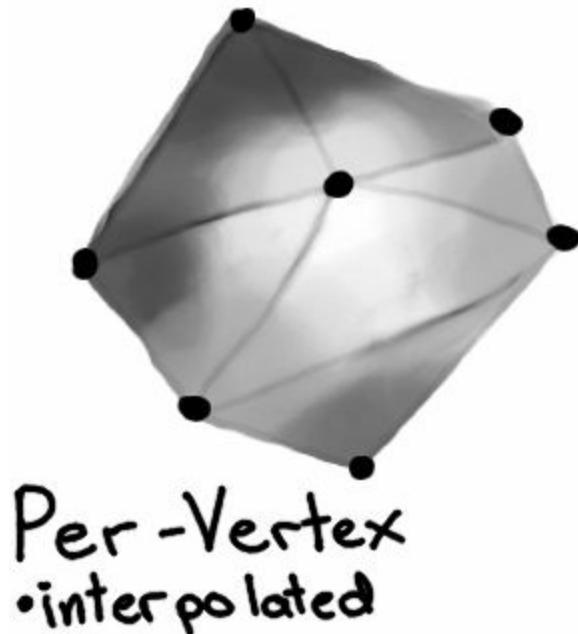
calculate them at all.



In this demo I used flat-shaded normals (they all point the same way as the face). In more complex meshes, you can calculate the normal for each face by taking the cross product of two of its sides. You can copy this to each vertex to achieve flat shading, or each vertex can get an average of the surrounding faces' normals for more smooth or organic lighting. We do this when creating terrain from a height-map, for example.

Per-Vertex Lighting and Per-Fragment Lighting

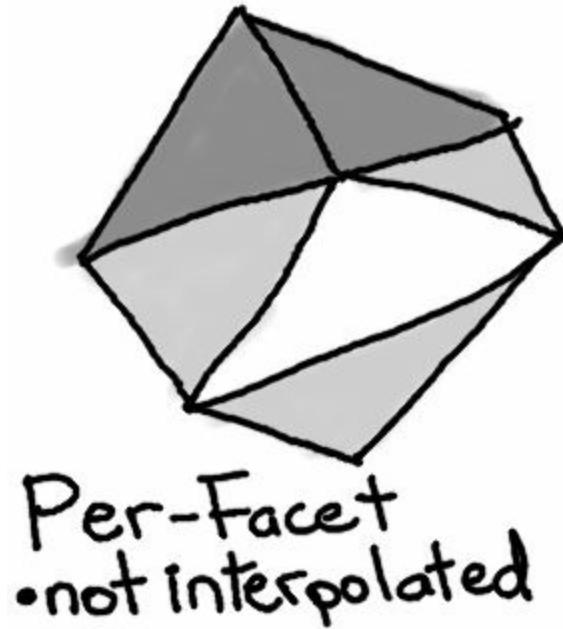
Older APIs with a software-rendering pipeline (and 3d games in the late '90s) calculated lighting **per-vertex** on an object, and interpolated this to the in-between fragments; H. Gouraud's "Continuous shading of curved surfaces" (1971), usually called "Gouraud Shading". We could replicate this by calculating the Phong equations in the vertex shader; using vertex position as the surface position, and out-putting the final intensity to the fragment shaders. This is going to be less accurate than calculating the lighting per-fragment, and we will lose detail like specular highlights in the surface centres.



Gouraud shading worked out lighting at each vertex, and interpolated it across polygons. This is okay, but we don't get any lighting detail that should be in the centre of a surface but doesn't touch a vertex.

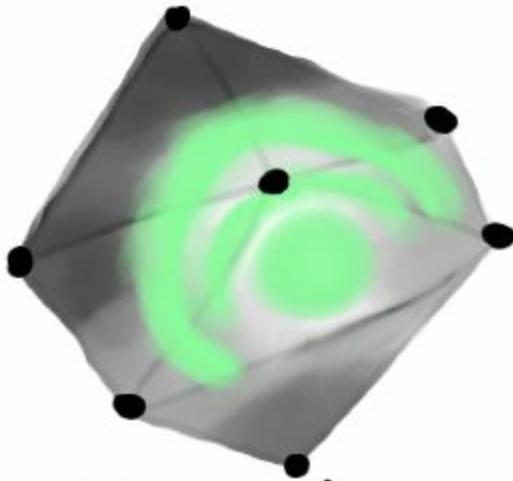
If you really wanted to, you could go back even further in time and replicate flat, non-interpolated, **per-facet** lighting (as in games from the early '90s) - this might be useful in a CAD application where you want to clearly distinguish polygons in a model. You can use the GLSL keywords to prevent

interpolation from the vertex shader to the fragment shader to replicate this.



With old-fashioned per-facet shading the faces perpendicular to the light will be brighter, but each face is a solid colour.

These days, because we finally have powerful-enough hardware to compute algorithms from the 1970s in real-time; (GPUs with fragment shaders), we usually calculate lighting per-fragment. This is sometimes called "Phong Shading", and gives us a much more accurate shading system. In this case, we will output the vertex positions, and vertex normals, so that they are interpolated to the fragment shaders; giving us a position, and surface normal for each fragment.



Per-Fragment
• more accuracy

With *per-fragment* shading we interpolate surface properties to each fragment, and work out the lighting models in the fragment shaders. Note that green "dot" (specular highlight) in the image doesn't touch any vertices.

When combined with texture sampling we can create some interesting per-fragment lighting effects (as indicated by the bands around the dot).

Remember that we will have lots more fragment shaders than vertex shaders running, so per-fragment lighting code may have a bigger impact on the time that it takes to render things.

Shader Code

Diffuse light should vary only based on the angle between the surface of an object (the facing of the surface is given by the **normal**) and the light position. The dot product of the normal, and the direction from the surface to the light position gives us an approximation of an angle. I'm going to assume that we will use per-fragment shading, so your vertex shader needs to output (interpolate) the vertex positions and normals to the fragment shaders. I'm also going to raise them to eye space to save me some time in the fragment shader:

Vertex Shader

```
in layout (location = 0) vec3 vertex_position;  
in layout (location = 1) vec3 vertex_normal;  
uniform mat4 projection_mat, view_mat, model_mat;  
out vec3 position_eye, normal_eye;  
  
void main () {  
    position_eye = vec3 (view_mat * model_mat * vec4 (vertex_position, 1.0));  
    normal_eye = vec3 (view_mat * model_mat * vec4 (vertex_normal, 0.0));  
    gl_Position = projection_mat * vec4 (position_eye, 1.0);  
}
```

The projection and view matrix uniforms come from a virtual camera. Sometimes the matrices are combined into a single MVP matrix, or P and MV - really this depends on your preference and how regularly you update each matrix.

Note that I raise everything to eye space - it doesn't really matter which space you use as long as it's consistent. We will see shortly that eye space might be a little handier for lighting. Remember that we need to raise 3d vectors to 4d before we can multiply them with a 4X4 matrix. This means adding w component. The normal should not be translated (moved) so I gave it a 0.0 as

the fourth component. I cast these as 3d vectors again when the calculation is done.

Important note: If we are doing any scaling in the model matrix where the axes are different (e.g. a horizontal stretch) then the model matrix will incorrectly scale the normal. In this case we would need to use a new "normal matrix" which is the **inverse transpose** of the `view matrix * model matrix` to correct the problem. I didn't bother here because I don't intend to do any scaling of my objects yet.

Ambient Intensity Term

The fragment shader gets the interpolated values from the vertex shader. I will also add in the lighting properties and the surface properties here. You can make these uniform variables if you want to change them during run-time. I'll need to raise some variables from world space to eye space, so I'll include the uniform for the view matrix here too.

```
in vec3 position_eye, normal_eye;

// fixed point light properties
vec3 light_position_world = vec3 (0.0, 0.0, 2.0);
vec3 Ls = vec3 (1.0, 1.0, 1.0); // white specular colour
vec3 Ld = vec3 (0.7, 0.7, 0.7); // dull white diffuse light colour
vec3 La = vec3 (0.2, 0.2, 0.2); // grey ambient colour

// surface reflectance
vec3 Ks = vec3 (1.0, 1.0, 1.0); // fully reflect specular light
vec3 Kd = vec3 (1.0, 0.5, 0.0); // orange diffuse surface reflectance
vec3 Ka = vec3 (1.0, 1.0, 1.0); // fully reflect ambient light
float specular_exponent = 100.0; // specular 'power'

out vec4 fragment_colour; // final colour of surface

void main () {
    // ambient intensity
    vec3 Ia = La * Ka;

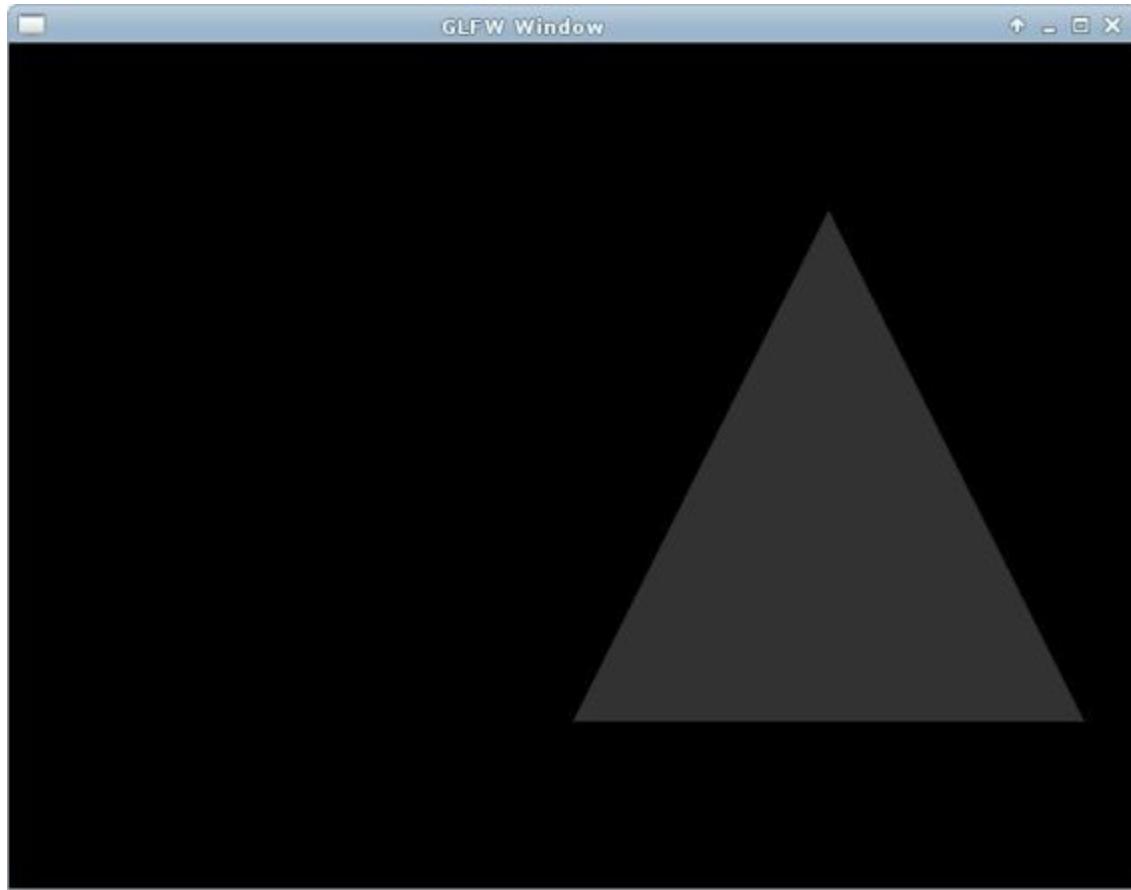
    // diffuse intensity
    vec3 Id = vec3 (0.0, 0.0, 0.0); // replace me later

    // specular intensity
    vec3 Is = vec3 (0.0, 0.0, 0.0); // replace me later

    // final colour
    fragment_colour = vec4 (Is + Id + Ia, 1.0);
}
```

So, we should be able to test this shader now - it should uniformly colour the object with the ambient (dark grey) colour. We will add in the calculations for the diffuse and specular terms in the next sections.

Note that the output colour expects a 4d vector, but our intensities are rgb (3d) colours. The final rgb colour is just a sum of the 3 intensity models. I added a 1.0 to the fourth (alpha) component. We're not using this yet.



Ambient lighting should look something like this on your triangle, and will be a "base" level of illumination when there is no other lighting.

Diffuse Intensity Term

Diffuse light should be brightest when the surface is facing the light (factor of 1.0), and not lit at all when the surface is perpendicular to the light (factor of 0.0). This is a job for the dot product - if we give it the facing direction of the surface (the normal) and the direction from the surface to the light it will

compare them and produce the factor that we are looking for. This means that, as the object rotates, the light on its surface will change.

We have the normal already, so we just need to get the actual straight-line direction between the surface and the light. We can get the 3d distance by subtracting one from the other, and just normalise that to turn it into a unit vector (length 1.0); a direction. But watch out! The light position is in world space; let's raise it to eye space so that it matches our surface position. We will need to include the uniform for the view matrix to the top of the fragment shader.

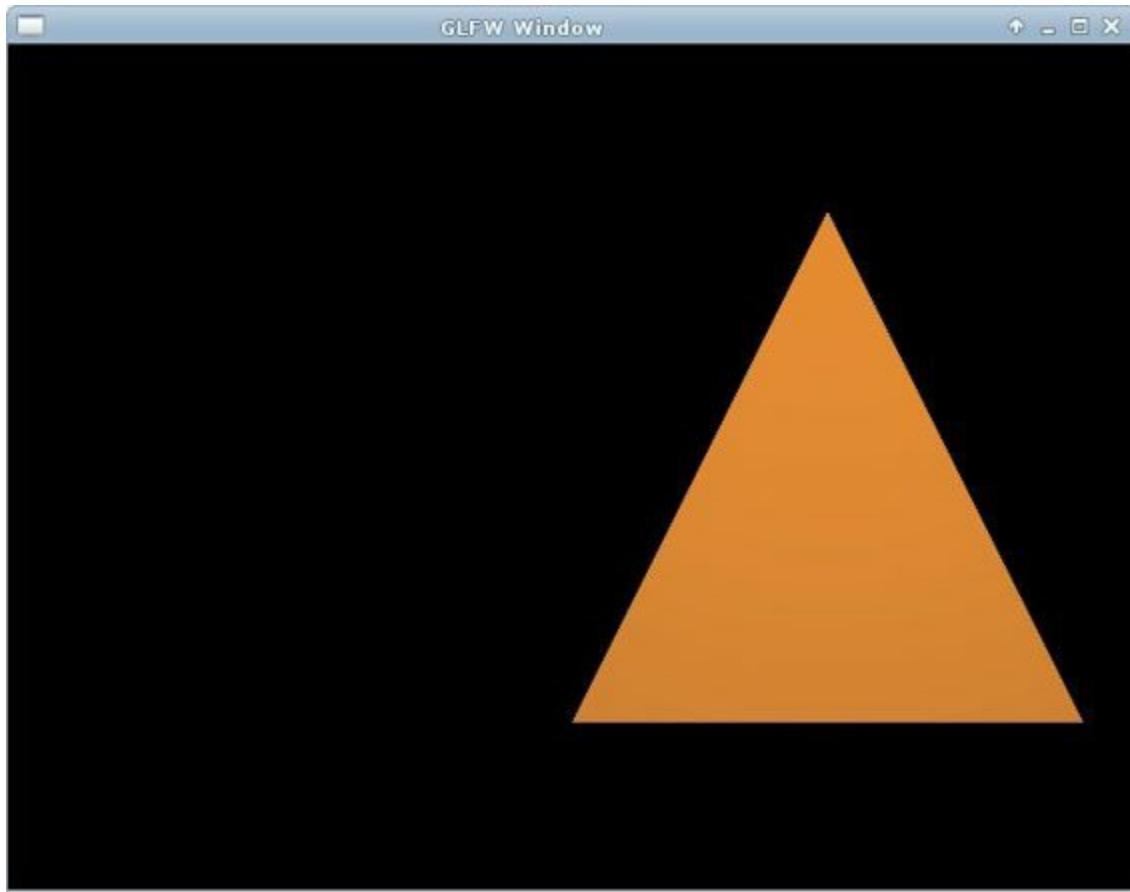
```
uniform mat4 view_mat;
```

Now we can work out the dot product, and use it to factor our diffuse light intensity. You can replace the `Id = ...` line from the previous shader code.

```
...
// raise light position to eye space
vec3 light_position_eye = vec3 (view_mat * vec4 (light_position_world, 1.0));
vec3 distance_to_light_eye = light_position_eye - position_eye;
vec3 direction_to_light_eye = normalize (distance_to_light_eye);
float dot_prod = dot (direction_to_light_eye, normal_eye);
dot_prod = max (dot_prod, 0.0);
vec3 Id = Ld * Kd * dot_prod; // final diffuse intensity
...
```

It is possible to produce a negative dot product. A negative colour will interfere with our other colours - we don't want that! This will appear as though black bands are "eating" our shape, at certain angles. We can get around this by making the dot product a minimum of 0.0; I used the `max()` function to do this.

You should be able to test this now - the parts of the surface parallel to the light should be more orange than grey now. If it didn't work draw the light world position on paper and compare this to the expect world facing of the surface. Perhaps the light needs to be in a different position to have more effect?



Diffuse light changes over the surface of the shape as its angle to the light changes. Try rotating the shape with a model matrix to see this as a dynamic effect.

Specular Intensity Term

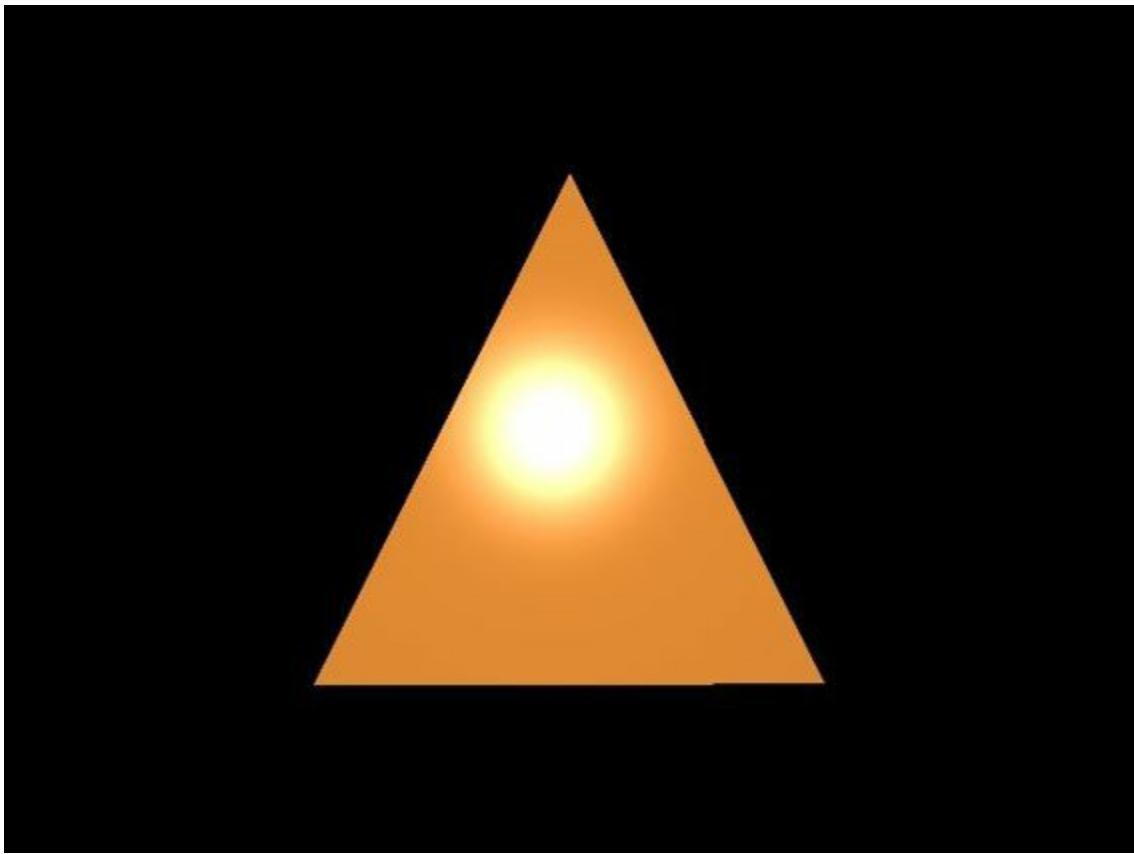
Specular light should directly reflect around the surface normal. There is a built-in GLSL function called `reflect()` to calculate this for us. Then we can compare the angle between the viewer and the surface with this new, reflected direction. If they are the same then the specular colour should be factored by 1.0. If they are perpendicular then there should be no specular light observed. Once again - the dot product. The main difference between specular and diffuse light is that it now depends on the angle between the light, the surface, and the observer. This means that, as the object or the camera rotate, the specular highlight will change. The specular highlight should also affect only a focused area of the surface - we can determine how big or small the highlight should be by raising it to a power given by the

specular_exponent variable.

```
...
vec3 reflection_eye = reflect (-direction_to_light_eye, normal_eye);
vec3 surface_to_viewer_eye = normalize (-position_eye);
float dot_prod_specular = dot (reflection_eye, surface_to_viewer_eye);
dot_prod_specular = max (dot_prod_specular, 0.0);
float specular_factor = pow (dot_prod_specular, specular_exponent);
vec3 Is = Ls * Ks * specular_factor; // final specular intensity
...
```

To get the direction from the light to the surface I just negated the previously used direction from surface to light. The second line is curious. To get the direction from the surface to the viewer (camera) I would normalise the distance. In this case, I know that the camera position is at the origin {0,0,0} in eye space, so instead of having a camera position variable - position_eye I can just say -position_eye. Now you can see why I raised everything to eye space - it saved me a uniform variable. I made sure that my dot product was not negative here also. Finally, I raise the dot product to a power using the built-in `pow()` function.

You should be able to test this now and see Phong lighting (using per-fragment shading) in action! Try changing the specular exponent to a larger or smaller value - it should change focus.



Specular light appears as a bright spot on the surface when the surface angle directly reflects light into the eye of the viewer. To test this, make the light and the camera position the same, and face the shape to the viewer. A rotating shape will show this dynamically. To make the "spot" bigger, reduce the specular exponent amount.

Common Mistakes

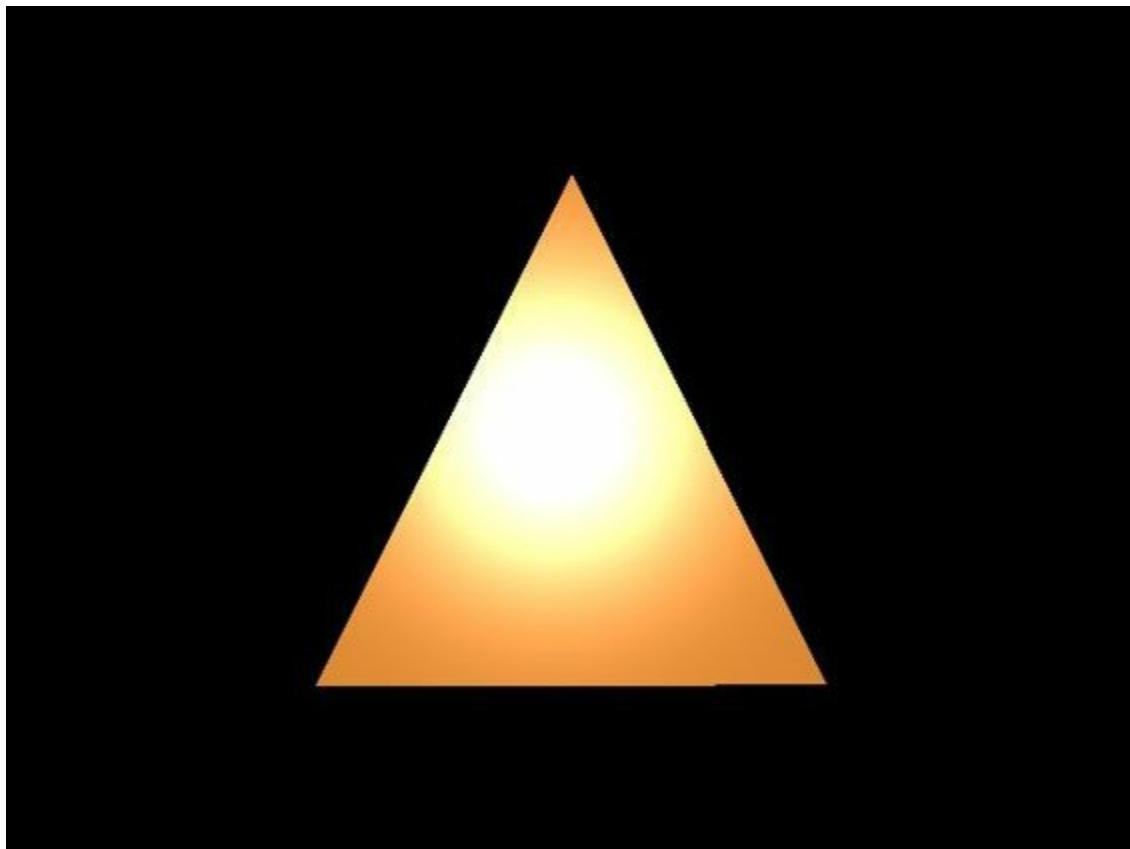
- comparison between vectors in different spaces (world coordinate light position versus local coordinate vertex)
- xyzw direction vector where the w component is set to 1
- xyzw position vector where the w component is set to 0
- dot product where one vector is a direction and the other is a distance (normalise to turn into direction)
- dot product where one vector is backwards (negate)
- normals are not transformed to correct space by use of inverse transpose matrix or uniform scale world view matrix
- vertex buffer contains wrong data
- vertex buffer stride is incorrect - vectors are formed from parts of other vectors
- shader has data from vertex buffer in wrong location order (normals from positions and positions from normals)
- invalid conversion between vec3 and vec4 or vec4 and vec3
- uniform variables are not set correctly or to correct name in main programme
- shader programme is using a hard-coded value so change to uniform variable has no effect
- colours use values not in range 0..1, but 0..255 or something - creates over-saturated colour result

Blinn-Phong

We can slightly improve the rendering rate of our fragment shaders by using J.F. Blinn's 1977 extension to Phong's reflectance model; "Models of light reflection for computer synthesized pictures". We can replace the expensive `reflect()` function with an approximation that uses a "half way" direction; half way between the surface-to-camera direction and the surface-to-light direction.

```
vec3 half_way_eye = normalize (surface_to_viewer_eye + direction_to_light_eye);
float dot_prod_specular = dot (half_way_eye, normal_eye);
float specular_factor = pow (dot_prod_specular, specular_exponent);
```

This has the effect of weakening your specular power by about half - try doubling the specular exponent to get roughly the same appearance.



With Blinn-Phong the specular highlight is cheaper to calculate (I tested it, it is), but ends up about half as focused.

Quiz

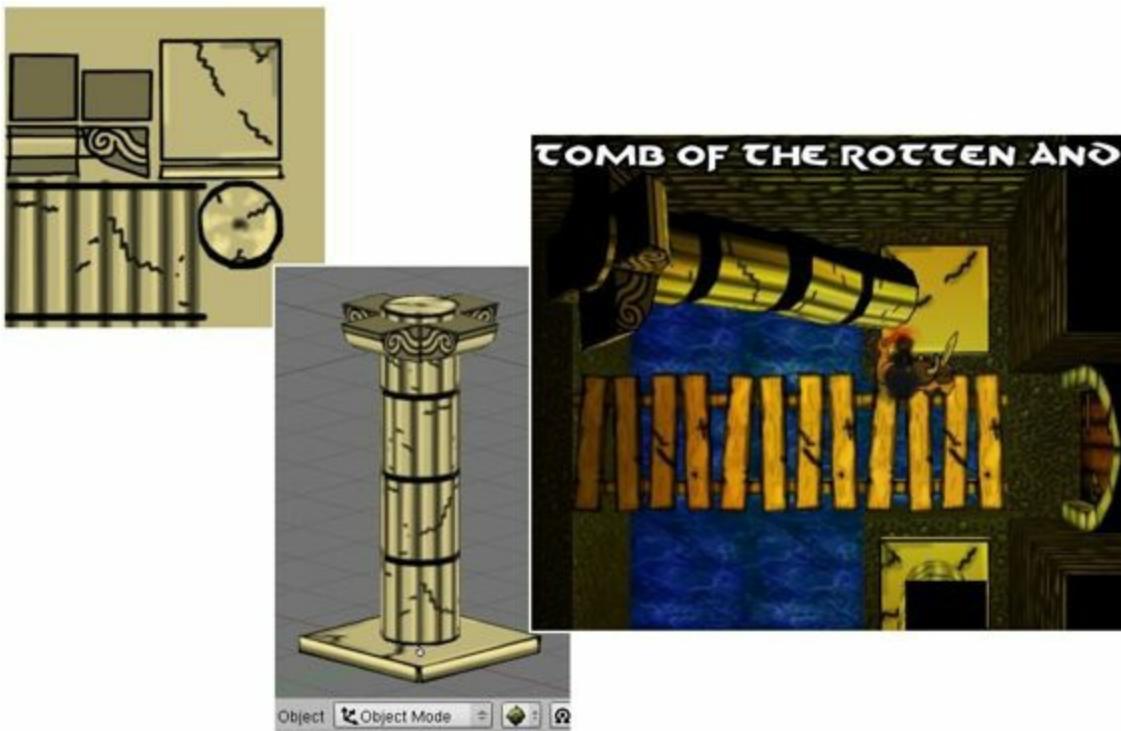
- If a surface has a diffuse reflectance of {1.0, 0.0, 0.0}, and a diffuse light has a colour of {0.0, 1.0, 0.0}, what will the final, diffuse, colour intensity of the surface be?

Multiple Light Sources

As your camera moves through a complex scene, you might like to have a number of dynamic lights in the world. You can guess that it is going to get a little bit complicated to manage a variable number of lights in your shaders. The simple option is - "always use the closest light source", or "always use the closest 3 light sources", and add their intensities together. The second option is going to be 3 times more expensive to calculate. To produce scenes that combine more than 2 or 3 lights, you might look at using **deferred rendering** to reduce the computational cost.

Texture Mapping

Texture mapping is another algorithm from the 1970s that real-time rendering technology wasn't ready for until the 1990s. Edwin Catmull (currently president of Pixar) is credited with creating the idea in 1974. The appropriate reference is his PhD thesis; "Computer display of curved surfaces". In real-time rendering history, it was adopted in 1990 by the team building the [still awesome] Ultima Underworld. There's a nice article about that development on Wikipedia. It was beaten to market by Catacomb 3-D (id Software). The idea is, that instead of using simple colours for surfaces, we will load the surface colours from an image file, and map them to the surface. Not copy, but map - because our surfaces are not going to be the same size as the image, and some maths are required to match them up.



The texturing process; (left) a square image is loaded from a file as a texture. (centre) A designer will assign every vertex on a mesh with a point on the texture; creating a texture coordinate for each vertex. (right) As the vertices are transformed in 3d, the rendering software will work out the texture coordinate of every fragment on the surface by interpolating the texture coordinates of its surrounding vertices, and sampling the colour of the corresponding point in the image.

We have some special terminology to avoid confusion between on-screen pixels, those from the original image, and the final coloured elements on a surface:

- **Texture**, an image (usually loaded from a file) that can be mapped onto a surface, and drawn to match the 3d perspective of the surface.
- **Pixels**, or "picture elements", are the final on-screen, coloured spots rendered to the viewport. Your GL viewport might have 1024x768 pixels.
- **Fragments** are pixel-sized areas of a surface (the visible surface is divided into pixel-sized 2d fragments). As a 3d triangle gets closer to the camera, its surface will contain more and more fragments. Fragments can overlap, and we can blend them together for techniques like partial-transparency, but there is only 1 final pixel rendered.
- **Texels**, or "texture elements", are pixels loaded from an image. If we map a 512x512 texture to a surface that occupies only 30 pixels of the viewport, we have more texels than fragments.

Loading Textures

OpenGL doesn't have any image file loading functionality. For simple image formats (raw bytes, or header-less TGA) it's really easy to write a function that reads an image file byte-by-byte, and copies this directly into an OpenGL texture. That's not really useful for anything more than a demo - we want to use a compressed file format like PNG to save hard-disk space when we have lots of image files. A popular choice is, of course, the **libPNG** library with **libZ**. That's okay, but the image-loading interface is actually pretty horrible, and then you have to link both libraries into your project. A simpler option is Sean Barrett's little **stb_image** library, which you can download from his website at <http://nothings.org/>. It will load PNGs, as well as a variety of other image formats (read the comments in the top of the code for a list). The interface is more sensible, and it comes as a file of source code that you can drop into your project. The only downside is that there are a few hacky bits of code that throw some compiler warnings. There are a few libraries around like **SOIL** that wrap **stb_image** and return a GL texture handle, but actually we can do better than that, and it's good to see how the GL texture creation functions work now, because we'll use them many times for different tasks. Whatever your preference of library, the functionality of all of these image loading options is much the same:

1. open image file
2. use the appropriate algorithm to reconstruct a 2d image in memory
3. return an array of bytes to us that we can copy into a GL texture

Now, I'm going to show you how to do this with the **stb_image** library. You can substitute the call to the **stb_image** library with any image loader of your choice - the rest of the code will be much the same. The only assumption that I make here is that I have a 4-channel image, which means RGBA. In other words - an image with a transparency channel. Some colour images are just RGB. As a graphics programmer, you really need to know what format of textures you are going to use, because we are going to make assumptions about this when we sample them later - we can't sample RGBA texels out of a texture that only has 3 channels! I want to use RGBA textures later, so I

actually tell `stb_image` to force any RGB images loaded to be packed-out to RGBA.

Using `stb_image`

Firstly, I separated the `stb_image` source into a `.h` header and a `.c` source file. If you look at the source code you can see that there are comments for `begin header file` and `end header file`. You can just cut this part out and paste it into a `.h`. Remember to `#include` the header in the source file. Now you can just drop these files into your project, and no linking is required.

```
int x, y, n;
int force_channels = 4;
unsigned char* image_data = stbi_load (file_name, &x, &y, &n, force_channels);
if (!image_data) {
    fprintf (stderr, "ERROR: could not load %s\n", file_name);
}
```

Great, now we have code to load an image into an array of bytes - remember that the size of a `char` is 1 byte. You can see where I told `stb_image` to force the loaded image to use RGBA (four bytes per pixel). That's it! Now, we can copy the image into an OpenGL texture.

Checking for Wonky Texture Dimensions

Older graphics cards could only cope with textures that had dimensions that were a power of 2 wide and high. Most newer cards can handle unusual texture dimensions, but it's worth printing a warning message if they are an unusual size - at least this way you have an idea that it might not display on all machines. Libraries like SOIL will automatically convert these images into power-of-two sizes, but that's a terribly disruptive waste of time, and will either reduce your image quality when sizing down or waste GPU memory when sizing them up. It's a much better idea to just print a warning to your artist so they make power-of-two sized textures to begin with. Image loading will be the slowest part of your programme - let's keep it as quick as possible.

```
// NPOT check
if ((x & (x - 1)) != 0 || (y & (y - 1)) != 0) {
    fprintf (
        stderr, "WARNING: texture %s is not power-of-2 dimensions\n", file_name
```

```
    );
}
```

The `stb_image` loading function gave me the dimensions of the loaded image, in my `x`, and `y` variables. Here the bitwise AND operator is used to check for non-power-of-two numbers.

Flip Image Upside-Down

If you were to copy your image into a texture now, you might notice that it's upside-down. This is because OpenGL expects the 0 on the Y-axis to be at the bottom of the texture, but images usually have Y-axis 0 at the top. We can add some code to flip the image upside-down before creating the texture. We know the width of each row in the image in bytes, because `stb_image` gave us the width in pixels, and each pixel is 4 bytes big. We can don't need to copy the whole image; we can just swap the top and the bottom half over. To do this we will put a pointer at the top row, and a pointer at the bottom row. We can swap the data of these rows over, then move the top pointer down a row, and the bottom pointer up a row. We just need to stop when we get to the middle. You'll need to be familiar with how pointer arithmetic works in C to follow this code:

```
int width_in_bytes = x * 4;
unsigned char *top = NULL;
unsigned char *bottom = NULL;
unsigned char temp = 0;
int half_height = y / 2;

for (int row = 0; row < half_height; row++) {
    top = image_data + row * width_in_bytes;
    bottom = image_data + (y - row - 1) * width_in_bytes;
    for (int col = 0; col < width_in_bytes; col++) {
        temp = *top;
        *top = *bottom;
        *bottom = temp;
        top++;
        bottom++;
    }
}
```

The interesting thing to note is that each time that we move a pointer we multiply the move by 4, because a pixel is 4 bytes wide (RGBA remember). All of our image data is contiguous, so that's okay; our current address + 4 will be the location of the next RGBA pixel. We use a `temp` variable to

remember the value at each location as we swap their values over. Remember that, in C, `top` is a pointer and just holds a number representing an address in memory. To get to the value in that address we "dereference" the pointer:

`*top.`

Copy Image Data into OpenGL Texture

The grunt-work is finished; now we can just copy our image data into an OpenGL texture. We need to generate a new texture object, which will give us an unsigned integer to refer to it with later. Before we can copy data into the texture we have to bind it into focus. Any time that we bind a texture it will be moved into one of several active texture slots. We don't need to worry about this yet, but it's good to specify a slot now, otherwise it will use the most recently activated slot, which might create unexpected interference with another part of your programme - remember GL is a state machine.

```
GLuint tex = 0;
 glGenTextures (1, &tex);
 glBindTexture (GL_TEXTURE_2D, tex);
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_RGBA,
    x,
    y,
    0,
    GL_RGBA,
    GL_UNSIGNED_BYTE,
    image_data
);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
 glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Look up the `glTexImage2D` function in the official documentation. It has a lot of parameters, but all of them should relate to how we loaded our image. At this stage we are just using bog-standard 2d textures, but we have to specify this whenever we bind or create a texture anyway. We aren't going to set up any custom levels-of-detail texture, so we just put 0 for this parameter. There are lots of options for texture formats - it's a good idea to have a look at the tables of options. We know for sure that our loaded image is RGBA format now, so we can say `GL_RGBA`. We give it the width and height in pixels, which were

returned by our image loader. The next "0" is an unused parameter. Again, we can put RGBA as our internal format. It's possible to have the bytes of each pixel ordered differently. Finally, we know that our image data is in `unsigned char` (byte) format, and we give it the pointer to our data. The last four lines are a good, safe, default "wrapping" mode to use for loaded textures, and a good default for anti-aliasing. We'll get onto both of these topics later.

Active Texture Slots

The texture is now loaded, and we can use it whenever we want to render by activating a texture slot, and binding the texture. It will stay in that slot until another texture is bound into it, so if your programme only has 1 texture then you can just do that once.

OpenGL can load a large number of textures into memory, but your GPU can only read from a small number of textures at one time (most GPUs will read 8 or so). To manage this, the state machine has a number of what it calls "active textures". We need to swap our textures in and out of these active texture slots as we render different objects. Yes, we have to do this low-level tedium ourselves, and it's very easy to make mistakes. The default active texture slot is number 0.

Adding a Sampler Uniform

To get hold of the contents of this active texture slot, we can represent it as a "sampler" variable inside our fragment shader. Let's add one in (I'll assume you've got a basic fragment shader working already):

```
uniform sampler2D basic_texture;  
...  
void main () {  
...}
```

By default, a sampler will use active texture slot number 0, so we don't really have to do anything else here. If you have OpenGL 4.2 on newer, then you can also use the "binding" layout to explicitly state that the sampler variable should read from active texture slot 0:

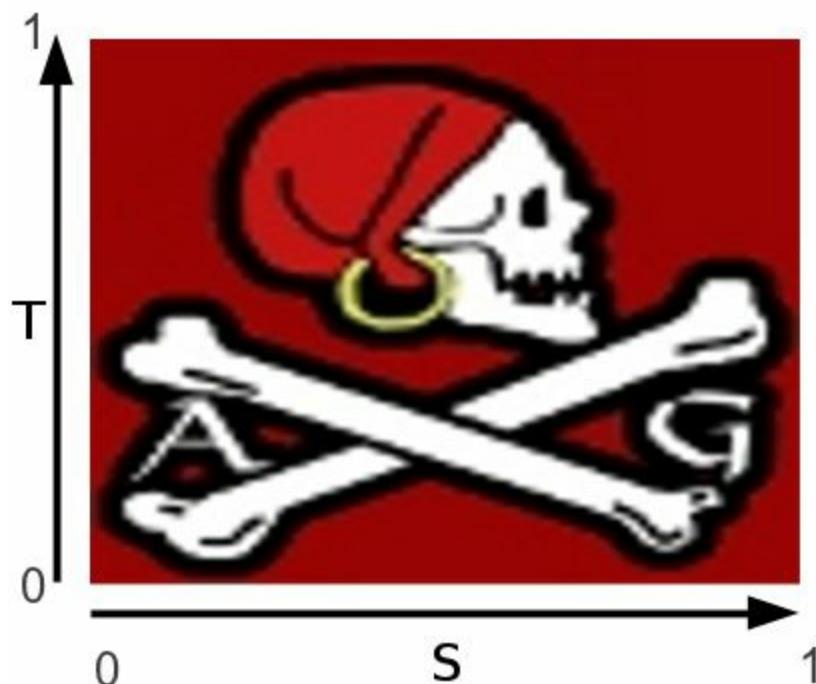
```
layout (binding = 0) uniform sampler2D basic_texture;  
...  
void main () {  
...}
```

The other way to do this is to get the location of the texture uniform in the normal way, and then update the value of the uniform with an integer representing the texture slot to use:

```
int tex_loc = glGetUniformLocation (shader_programme, "basic_texture");  
glUseProgram (shader_programme);  
glUniform1i (tex_loc, 0); // use active texture 0
```

Texture Co-ordinates

The main problem is that the surfaces are not going to have the same number of fragments as the image has texels, and this will change as the surface is viewed at viewpoints and perspectives. Have a look at the Catacomb screen capture; the wall closest to the screen is maybe 60 pixels high, and the farther wall is maybe 20 pixels high, but both are using the same texture. The obvious solution is to calculate a very simple interpolation function for each fragment of the wall, to find out the closest-matching texel from the image.



GL texture co-ordinates for s (horizontal) from left to right, and t (vertical) from bottom to top. Hint: test your texture mapping software with an image that has text in it; this will help you spot if you've got one of the axes reversed.

To keep interpolation of texels independent of the image size, we use **texture co-ordinates**, which are a horizontal and vertical value between 0.0 and 1.0. In OpenGL 0.0 is the left or bottom of a texture, and 1.0 is the right, or top of a texture. Direct3D has 0.0 at the top, and 1.0 at the bottom - don't mix them up!

So we can make another per-vertex variable to keep track of what part of the

image each vertex should map to; a texture co-ordinate. This will be 2 floats for each vertex, and we will make another vertex buffer to store these. Our previous vertex buffers have all had 3 floats - don't accidentally tell GL that it is 3d, or it will get the memory "stride" (ordering) wrong, and you'll have mixed-up values.

We use x,y,z,w to refer to 3d space, and r,g,b,a for colours. In most 3d libraries u,v are used to refer to 2d space, but OpenGL prefers s,t. The GLSL language vector data-types (`vec2`, `vec3`, `vec4`) have built-in short-cuts called **swizzle operators**. You may have used something like `vec3 pos = result.xyz;` or `float red = Kd.r;` already. You can also use `.s`, `.t`, and `.st`.

Creating Texture Co-ordinates

For a wall mesh, made up of 2 triangles, I might created the following 6 texture coordinates, and put them in a vertex buffer. Note: If you interleave all of your per-vertex attributes into a single buffer it should slightly speed up your programme, but it's a little trickier to set up. I'll assume that you've already got a vertex array object and vertex positions in a vertex buffer.

```
GLfloat texcoords[] = {
    0.0f, 1.0f,
    0.0f, 0.0f,
    1.0, 0.0,
    1.0, 0.0,
    1.0, 1.0,
    0.0, 1.0
};
GLuint vt_vbo;
glGenBuffers (1, &vt_vbo);
 glBindBuffer (GL_ARRAY_BUFFER, vt_vbo);
 glBufferData (
    GL_ARRAY_BUFFER,
    sizeof (texcoords),
    texcoords,
    GL_STATIC_DRAW
);
// note: this is your existing VAO
GLuint vao;
 glGenVertexArrays (1, &vao);
 glBindVertexArray (vao);
 glBindBuffer (GL_ARRAY_BUFFER, vt_vbo);
// note: I assume that vertex positions are location 0
dimensions = 2; // 2d data for texture coords
glVertexAttribPointer (1, dimensions, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray (1); // don't forget this!
```

If you're using normals for lighting, then you will have 3 per-vertex variables; 3d positions, 3d normals, and 2d texture co-ordinates. Make sure that there is a vertex pointer with an unique location for each one; 0, 1, and 2. It's quite an easy (and common) mistake to get this wrong too. In the above code I assumed that the vertex positions would be location 0, and made the texture coordinates position 1. If you are also using normals - change them to location 2.

Interpolating Texture Co-ordinates

We want to have an unique colour from the texture for each fragment, so we will output the texture co-ordinates to the fragment shader. Remember that this will automatically interpolate the values to each fragment. So a fragment half-way up our wall will get a vertical texture co-ordinate of 0.5. In the vertex shader:

```
...
layout (location=1) in vec2 vt; // per-vertex texture co-ords
out vec2 texture_coordinates;

void main () {
    texture_coordinates = vt;
...
}
```

Note that I specified the location of my per-vertex texture coordinates, `vt`, to match the one that we set up in the VAO's attribute pointer. This will reduce the chance of mix-ups between attributes. Usually the location is the order that you write them in the shader, but it's easy to lose track if you are modifying the shader. You can also query the location of attributes in C, as we did in the shaders article.

Texture Sampling

Now we have the correct, proportional, texture co-ordinates per-fragment. For each fragment we want to convert each co-ordinate from the 0-1 range into an actual texel inside the size of the image. e.g. if our fragment's vertical texture co-ordinate is 0.6, and the original image is 512 texels high, then $0.6 \times 512 = 307.2$. We might round down, and use the 307th texel from the

bottom of the image (the **nearest neighbour**). This process is called **texture sampling**, and GLSL has built-in functions to do this calculation for us.

```
in vec2 texture_coordinates;
uniform sampler2D basic_texture;
out vec4 frag_colour;

void main () {
    vec4 texel = texture (basic_texture, texture_coordinates);
    frag_colour = texel;
}
```

The `texture ()` function knows how to retrieve the matching texel in the texture from the sampler, and returns an `rgba` value. I used this as my fragment colour. To get better compatibility with OpenGL 3.2 you can replace the `texture ()` function with it's more specific ancestor: `texture2D ()`.

If you were to sample a texture using a coordinate value bigger than 1.0, or smaller than 0.0 what happens? It depends on your textures' wrapping mode. We set ours to "clamp to edge", which will force the texture coordinates back into the 0.0-1.0 range. In my case, the edges of my texture are white, so anything outside 0.0-1.0 is just going to be white. You can also set your texture to repeat, or to use a custom "border" colour that you can set up, which will come in handy for some more advanced effects.

Summary



2 triangles textured with the image from this web page (you might use this too, as you know it will load correctly now, and has text on it to make sure that your texture co-ordinates and not back-to-front.

The currently used shader programme will access the attributes from the currently bound VAO, so remember to use one, and bind the other before drawing.

Common Mistakes

If it doesn't work - the first thing to check is your actual texture file.

- Make sure that the format is one supported by the loader library (e.g. PNG). Some libraries do not support greyscale. The code that we wrote should at least support both RGB and RGBA format images.
- That the dimensions are sensible - i.e. a power of 2 like 256x256.
- You can try printing the first 4 bytes of loaded image data, to see if you get sensible values between 0 and 255 for the first RGBA pixel:

```
printf ("first 4 bytes are: %i %i %i %i\n",
       image_data[0], image_data[1], image_data[2], image_data[3]
);
```

I get "first 4 bytes are: 154 4 4 255" for a red pixel in the top-left.

- Check that you activated texture slot 0, and bound your texture into it before drawing.
- OpenGL does not properly set defaults for texture wrapping and anti-aliasing. You might get a black, or poorly-mapped texture if you leave out the `glTexParameteri()` calls after creating the texture.

Next check the texture coordinates. Draw the triangles on paper. Do the texture co-ordinates match up to the vertices in the same winding direction?

- Are you copying the texture coordinates array into the correctly bound VBO?
- Are you using the same VAO that you are drawing the points with - i.e. did you accidentally make a second VAO?
- Did you accidentally make your data too big or too small, because you forgot that texture coordinates are 2d and not 3d for the `glBufferData()` size parameter?
- Did you give a valid location number (1 or 2) for the buffer after binding the VAO - `glVertexAttribPointer(1, ...)`?
- Did you explicitly enable the same attribute number after binding the VAO? Only attribute 0 is enabled by default.

You can check your texture coordinates independently of the texture. In your fragment shader temporarily set the colour with `frag_colour = vec4(texture_coordinates, 0.0, 1.0);`. You should get red where S is close to 1.0, and green where T is close to 1.0.

Next, check the shaders - it's easy to make a small mistake here. If your texture has loaded, but the coordinates are wrong, then you will probably see one texel's colour from the texture used for every fragment (usually the bottom-left, which is 0,0).

- Did you add the input attribute to your vertex shader as a `vec2`?
- Did you remember to output this to the fragment shader?
- Get the location of the texture coordinate attribute in the shader (see Shaders article), and make sure that it matches the number that you gave it as an attribute. Sometimes these are not in the order that you expect, and the shader has mixed up your input variables.
- Did you use the `texture` function correctly, and store the result int a `vec4`?
- Did you remember to set an output value from your fragment shader, and make this equal to the sampled texel?

Aliasing



Magnification aliasing - the walls on the side have a texture that is lower resolution than the screen; creating giant blocks where the same texel is used for a block of pixels. Image from Tomb Raider, Eidos Interactive, 1996.

Sampled, 2d, textures have a fundamental problem when stretched over a 3d surface - **aliasing**. When the image is very close to the camera you have an aliasing effect called **magnification**; when there are more pixels on screen than there are texels. This results in giant blocky bits covering several pixels, and was quite common in older games. The obvious solution to this problem is to use higher-resolution textures for those surfaces, but this isn't always practical, so we'll look at a generic solution too.



The trees in the distance are noticeably aliased from minification; the sampler is choosing the nearest neighbour which is sometimes the transparent background, and sometimes a green leaf. As the camera moves this inaccuracy will create a flickering effect as the sampling almost randomly switches between the two colours. Image from Combat Mission, Battlefront.com, 1999.

The other problem is when the texture is farther away from the camera, and there are more texels than there are pixels to display them on. The **nearest neighbour** algorithm will pick the closest texel, but this produces a sharp, "pixellated" look, and can create a flickering effect as the camera moves, and different colours are rapidly sampled from more detailed textures.

Anti-Aliasing

Filters

A solution to aliasing problems is to filter the sampling process. We can tell OpenGL to apply a filter to a texture; this will force the sampler to actually take several samples - usually the 4 nearest texels - **bi-linear filtering** to the texture co-ordinate, and compute a weighted average based on the nearness to each one. So if the texture co-ordinate is bang-on to a texel, it will choose that colour, but if it is in-between 2 red and 2 blue texels, then the resulting colour will be purple.

Let's look back at these lines after where we first created the texture:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

The min and mag filters are:

- GL_NEAREST - Nearest-neighbour (NN) algorithm
- GL_LINEAR - Bi-linear (BL) filtering. This will blur any aliased textures.
- GL_NEAREST_MIPMAP_NEAREST - NN with mip-maps. (min filter only)
- GL_LINEAR_MIPMAP_NEAREST - BL with mip-maps. (min filter only)
- GL_NEAREST_MIPMAP_LINEAR - NN with mip-maps and tri-linear (TL) filtering between mip-maps. (min filter only)
- GL_LINEAR_MIPMAP_LINEAR - BL and TL filtering with mip-maps. (min filter only)

We've mentioned the first 2 filters in the table, and we will look at the various mip-map options next.

MIP Maps

Multim im parvo - meaning "many in one". Created by Lance Williams in 1983. This technique creates several copies of each texture, at different sizes. If your original image is 512x512 pixels, then it will create a 256x256 version, a 128x128 version ... down to a single pixel version. When used, the sampler will work out what size resolution is needed to texture a surface, and choose the next-smallest image from the MIP-maps. This will drastically reduce minification artifacts, and create a noticeably blurred look at different distances from the camera. It will also drastically increase real-time performance of texture mapping.



Mip-mapping is used here to reduce aliasing artifacts. Surfaces at acute angles to the camera are very blurred, because the largest square mip-map has been chosen for sampling. This can be improved with an-isotropic (non square) filtering.

The disadvantages to mip-mapping are that the extra images will cost 1/3 more memory than storing the original texture - not a lot more. Bands of different blurring levels will be visible over large surfaces using a repeated texture (such as floors). To counter the banding effect there is yet another filter called **tri-linear filtering** which will blend between mip-maps, but requires several additional texture samples. A further disadvantage occurs

when surfaces are roughly perpendicular to the camera - more wide than high, in other words. These will create a rectangular area of fragments to texture - e.g. 128x64. In this case mip-mapping would select the 64x64 mip-map to sample, which will be appropriate for the depth of the texture, but appear very blurred horizontally. A solution to this is to use **an-isotropic filtering**, which will create additional mip-maps at rectangular sizes such as 128x64. This will increase the memory cost.



The same image, with an-isotropic mip-maps generated. The walls appear less blurred, but still do not show aliasing artifacts. This increases the memory cost of mip-mapping.

Existing libraries generate mip-maps on the CPU, which is an horrendously slow process. In the past, to avoid this slow process, games used DDS (Microsoft Direct Draw Surface) images, which had mip-maps pre-built in them. In modern graphics we can use the GPU to do this - which is extremely efficient. Add this line after where you called `glTexImage2D`:

```
glGenerateMipmap (GL_TEXTURE_2D);
```

Make sure that you are using one of the mip-map filters for minification. The

magnification filter can be set to bi-linear filtering. Mixing these up is a very common mistake, and will explain why your mip-mapping isn't working!

An-isotropy is an extension that you can set right after the other texture filters. An-isotropy has a factor, to determine how many different sizes of rectangle are stored. You can work out the maximum an-isotropy supported by the system too.

```
// work out maximum an-isotropy
GLfloat max_aniso = 0.0f;
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &max_aniso);
// set the maximum!
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, max_aniso);
```

Screen Capture Function

If you've tried using the `PrtScn` button to capture a screen to an image from the O/S, then you've probably noticed tearing or horizontal lines in the images, especially in scenes with animation. We get bits of different rendering frames in different parts of the same shot. We can write our own screen dump function that reads a complete rendering frame from the **framebuffer** and writes it to a file. It's quite easy to do, and will teach us how the framebuffers work (useful for advanced effects), and to read/write from image files.

The easiest option is to export to a raw image file format, and convert this to a PNG image afterwards using a tool like ImageMagick. Alternatively, you can use a library that can create a formatted image in memory, and write a PNG, or other file type directly. I provided some code in the Texture Maps chapter that used Sean Barrett's **stb_image** to read a PNG, so for consistency I'll provide some code that uses **stb_image_write** to write a PNG. You could, of course, replace this with the interface to **libPNG** or another library.

Get the Pixels from the Framebuffer

First, allocate some memory. We are going to do a basic colour image with 8 bits per channel (RGB), so 24 bits-per-pixel or 3 bytes in total. We have `g_gl_width * g_gl_height` pixels on the screen. Get these values from a window resize call-back if you have one, as the window may have been resized slightly to fit onto the display.

We can get the pixels from the GL framebuffer (containing whatever was drawn last), and copy them into our buffer of memory. I'm using C memory allocation, so you'll need to include `stdlib.h`. I cast this to point to unsigned characters, just because they are 1 byte each.

```
unsigned char* buffer = (unsigned char*)malloc (g_gl_width * g_gl_height * 3);
glReadPixels (0, 0, g_gl_width, g_gl_height, GL_RGB, GL_UNSIGNED_BYTE, buffer);
```

Now we have an image in memory - not so tricky! You could use this to create a texture if you wanted to.

Write a Raw Image

"Raw" means that the image just contains the bytes; one after the other. So "RGBRGB...RGB". I wanted to have a time-stamp in my image file names so that they didn't over-write each other when I restarted the programme. You'll need to include `time.h` for this. You'll also need `stdio.h` for string `printf` and the file functions.

```
char name[1024];
long int t = time (NULL);
sprintf (name, "Screenshot_%ld.raw", t);
```

Now we can open the file in "write binary" ("wb") output mode. I'm going to use the C file output functions. You can, of course, use C++ file functions instead.

```
FILE* fp = fopen (name, "wb");
if (!fp) {
    fprintf (stderr, "ERROR: could not open %s for writing\n", name);
}
```

Right Side Up

If you skip this step, you'll find that the image will be written flipped upside-down. To get around this, I wrote it to the file in backwards **row order** (where a row is each horizontal line of pixels across the screen). Notice that I just go in a loop, and `fwrite()` a whole row of bytes at a time, with no spaces or line-breaks or anything like that.

```
int bytes_in_row = g_gl_width * 3;
int bytes_left = g_gl_width * g_gl_height * 3;
while (bytes_left > 0) {
    int start_of_row = bytes_left - bytes_in_row;
    fwrite (&buffer[start_of_row], 1, bytes_in_row, fp);
    bytes_left -= bytes_in_row;
}
fclose (fp);
free (buffer);
```

Convert to PNG

More complicated image types have 2 major differences from our raw image;

- A file header, containing some details about the file contents
- Compressed data i.e. a clever method of not writing every pixel, or using less than 3 bytes per-pixel

Formats like Targa TGA are basically just a raw file with a header, and an optional 4th channel (so 32-bits per pixel). You can do this in about 5 more lines of code, but the only real advantage over raw is that it opens natively in most image viewers.

What we really want is a high-quality (loss-less), compressed image - PNG. This is more complicated to write. You can use libpng, or the ImageMagick library to export to PNG, but actually it's much easier to just invoke the ImageMagick command-line utility, which is, by default, available on most Linux/Unix systems. Try typing "convert" into a console. If it's not there - www.imagemagick.org/.

I'm telling ImageMagick that each channel is 8-bits, and what dimensions to expect. I also tell it to take input from my time-stamped screen-shot, which has 3 channels (rgb), and to write to a PNG file with the same name, and a different extension. The `system` function will run this command as if it were in a console, so if you don't have ImageMagick on the local path, you'll need to sort that out. The last command is the Unix code for "remove this file without prompting". If you're on Windows then you'll need to use `del`, or just leave this command out.

```
// invoke imagemagick
char command[2048];
sprintf (
    command,
    "convert -depth 8 -size %ix%i rgb:screenshot_%ld.raw screenshot_%ld.png",
    g_gl_width,
    g_gl_height,
    t,
    t
);
system (command);
sprintf (command, "rm -f %s", name);
system (command);
```

Writing a PNG Directly with `stb_image_write`

If you're using `stb_image` to load image files, then you probably want something of similarly little fuss for writing them. You can get `stb_image_write` at http://nothings.org/stb/stb_image_write.h. You'll need to cut and paste the bottom part into a `.c` file (you'll see where), and include the header file in it, as well as providing the `#define` that it's looking for. Other than that it's no fuss at all to get working.

```
unsigned char* buffer = (unsigned char*)malloc (g_gl_width * g_gl_height * 3);
glReadPixels (0, 0, g_gl_width, g_gl_height, GL_RGB, GL_UNSIGNED_BYTE, buffer);
char name[1024];
long int t = time (NULL);
printf (name, "screenshot_%ld.png", t);
unsigned char* last_row = buffer + (g_gl_width * 3 * (g_gl_height - 1));
if (!stbi_write_png (name, g_gl_width, g_gl_height, 3, last_row, -3 * g_gl_width)) {
    fprintf (stderr, "ERROR: could not write screenshot file %s\n", name);
}
```

So, it's the same code as before, but with a different file name. A nice thing about `stbi_write_png` is that we can get it to flip the image the right way up for us. The fifth parameter takes a pointer to the top-left pixel of the image, so I gave it a pointer to the bottom row of the image in memory. The last parameter takes the "stride", or number of bytes to increment to get to the next row. I gave it a negative row width, so that it went backwards.

Going Further

Now we know how to read from a framebuffer, and write images to files. Writing to a frame buffer is the same process in reverse, and is used for things like post-processing effects, where you can draw anything you like over the top of an image, or blur parts of it. Reading from a file is just as easy, but you probably want some assistance from a library to read PNG files.

You can write a `record_video()` function that calls this one at fixed intervals. Writing lots of images will completely slow down anything that you're running, so it might not work well in an interactive demo. A clever way to get around this for interactive demos/games is to save a list of things moving in memory whilst in interactive mode, then replay it later in fixed-steps for saving to images. You can use various software to take the images and create an MPEG or GIF from them. The trick is to get the intervals between images exact.

Potential Problems

If you get the view dimensions wrong then you'll get a mis-aligned image. You might end up being 1 or 2 bytes too wide, which will leave some "undefined" pixels on the edges, which will probably show as flickering/noise in an image viewer.

Video Capture Function

How do I record a video of my demo to show off to my friends without using Fraps? This is how!

There are tools available that will capture video from real-time rendering applications, but none of them are very good. If you need a demo for a presentation, and you're not happy with Fraps on Windows, it can be a rather frustrating experience. If you want high-quality recordings then you need to write a function to read your framebuffer at a fixed interval, store each one in memory, then write each one to an image later. You can turn the series of images into a video using one of the many available video editing tools.

Reserve a Big Chunk of Memory

You don't want to do memory allocation during the recording process, because it's going to slow you down and ruin your video. Do it before-hand. This means you need to know:

- how many seconds the video will run for (at maximum)
- how many frames per second the video should display

Do a quick calculation - each pixel has 3 bytes (1 for red, blue, and green). If you have an 800x800 display then you need 1.92 megabytes per frame. Pick a video playback rate; commonly used rates are 29.97 frames per second, 25, and 24. I used 25. For every second you'll need 25 frames - that's 48 MB per second of video. You might want to limit your videos to a certain maximum time - I used 10 seconds - 480MB per video. You'll need a decent computer to record video at this quality.

If your videos won't fit in memory then you can reduce the video capture resolution (you'll need to do some maths to pick the nearest pixel from the framebuffer, for example), or you can record, say, only every 2nd frame. Most video editing software will let you use one image to count for 2 frames - a small drop in quality. Better solution - use a more powerful computer.

```
unsigned char* g_video_memory_start = NULL;
unsigned char* g_video_memory_ptr = NULL;
int g_video_seconds_total = 10;
int g_video_fps = 25;

void reserve_video_memory () {
    // 480 MB at 800x800 resolution 230.4 MB at 640x480 resolution
    g_video_memory_ptr = (unsigned char*)malloc (
        g_gl_width * g_gl_height * 3 * g_video_fps * g_video_seconds_total
    );
    g_video_memory_start = g_video_memory_ptr;
}
```

I use a pointer here to keep track of the start of my giant chunk of memory. And another one that I'll use to iterate along the memory for each frame.

Grab Each Video Frame

Be aware that, in normal use, your programme's frame rate will jump all over the place. We therefore can not save every OpenGL frame and use it as a video - video editing software expects us to have evenly spaced images of frames - in my case, 1 image for every 1/25 seconds. We can add a little time-step manager that looks at the elapsed time since the last OpenGL frame, and only stores the frame in memory if 1/25 second has passed since it stored the last frame. If 2/25 of a second have passed I'll make sure that 2 copies of the same frame will be saved - this should fix any inconsistencies in the video playback. I have a video capture "mode" that I enable with a key press. I'll check if this is on right after the frame is drawn to screen with `SwapBuffers()`.

```
// initialise timers
bool dump_video = false;
double video_timer = 0.0; // time video has been recording
double video_dump_timer = 0.0; // timer for next frame grab
double frame_time = 0.04; // 1/25 seconds of time
// rendering main loop
while (true) {
    ...
    if (dump_video) {
        // elapsed_seconds is seconds since last loop iteration
        video_timer += elapsed_seconds;
        video_dump_timer += elapsed_seconds;
        // only record 10s of video, then quit
        if (video_timer > 10.0) {
            break;
        }
    }
    ...
    glfwPollEvents ();
    glfwSwapBuffers (window); // whereGL displays current frame to screen
    if (dump_video) { // check if recording mode is enabled
        while (video_dump_timer > frame_time) {
            grab_video_frame (); // 25 Hz so grab a frame
            video_dump_timer -= frame_time;
        }
    }
}
```

Okay, so the above code is just an example of how to update some timers inside your main loop that accumulate the elapsed time, and grab a frame whenever this accumulates to 1/25 of a second. We'll write the `grab_video_frame()` function now.

```
void grab_video_frame () {
```

```
// copy framebuffer into 24-bit rgbrgb...rgb image
glReadPixels (
    0, 0, g_gl_width, g_gl_height, GL_RGB, GL_UNSIGNED_BYTE, g_video_memory_ptr
);
// move video pointer along to the next frame's worth of bytes
g_video_memory_ptr += g_gl_width * g_gl_height * 3;
}
```

This just reads the framebuffer, pixel by pixel, and stores them at the location in memory indicated by our pointer. This will start at the beginning of the memory that we reserved. We can move this along to the end of the memory that we just copied. We know that our memory addresses are contiguous for our big allocated chunk, so I just added the size of 1 frame to the address that it points to ($\text{width} * \text{height} * \text{bytes per pixel}$).

That's all we need to do to record video into memory, and it should be fast enough to do in real time (unless your programme is already running quite slowly). When the timer gets to 10 seconds it will stop recording (otherwise it will write over memory that we haven't allocated!).

Dumping Frames to Images

This is the same as writing a screenshot to an image - but we have a series of them. It's going to make it easier on the video editing software if we suffix their file name with a number. You can call `dump_video_frames()` when the programme finishes/user quits, as it will take quite a long time.

```
bool dump_video_frames () {
    // reset iterating pointer first
    g_video_memory_ptr = g_video_memory_start;
    for (int i = 0; i < g_video_seconds_total * g_video_fps; i++) {
        if (!dump_video_frame ()) {
            return false;
        }
        g_video_memory_ptr += g_gl_width * g_gl_height * 3;
    }
    free (g_video_memory_start);
    printf ("VIDEO IMAGES DUMPED\n");
    return true;
}
```

First we can rewind our iterating pointer to the start of our allocated memory. Remember that I was clever enough to keep an extra pointer, remembering where this was. Now we can just loop through for every frame that we recorded (25 per second), write each frame to an image file, and increment the pointer along to the next frame. I freed the allocated memory at the end of this. The following function is almost identical to the screenshot saving code, except that it numbers each image. I write a .raw image, then convert each one to a .png with ImageMagick.

```
bool dump_video_frame () {
    static long int frame_number = 0;
    // write into a file
    char name[1024];
    // save name will have number
    sprintf (name, "video_frame_%03ld.raw", frame_number);
    FILE* fp = fopen (name, "wb");
    if (!fp) {
        fprintf (stderr, "ERROR: writing video frame image %s\n", name);
        return false;
    }
    int bytes_in_row = g_gl_width * 3;
    int bytes_left = g_gl_width * g_gl_height * 3;
    while (bytes_left > 0) {
        int start_of_row = bytes_left - bytes_in_row;
        fwrite (&g_video_memory_ptr[start_of_row], 1, bytes_in_row, fp);
        bytes_left -= bytes_in_row;
    }
}
```

```
}

fclose (fp);

// invoke ImageMagick to convert from .raw to .png
char command[2048];
sprintf (
    command,
    "convert -depth 8 -size %ix%i rgb:video_frame_%03ld.raw video_frame_%03ld.png",
    g_gl_width,
    g_gl_height,
    frame_number,
    frame_number
);
printf ("%s\n", command);
system (command);
// delete the .raw
sprintf (command, "rm -f %s", name);
system (command);

frame_number++;
return true;
}
```

Of course, you don't need to export a raw image - you could use a library to export a formatted image file, as we did in the Screen Capture chapter.

Compile Into Video

Image Sequence to Video

At this point it should be easy enough to use a tool to create and edit your video. Experience has taught me that using command-line tools like `ffmpeg` and `mencoder` can be quite tricky and frustrating, and actually being able to annotate your videos visually, or synch an audio track, is essential for a lot of tasks. Almost any free or commercial video editing suite will do the trick. I just used OpenShot (a free Linux editor), and it worked splendidly. Adobe Premier Pro is also excellent. Remember to browse over your image output first, and get rid of any duds at the beginning or end of the sequence. **Don't forget to set the matching frame rate when exporting (e.g. 25 fps or 29.7 fps).** Observe the sterling quality of video - no nasty lossy video, and no horrible, blocky, blurry, intermediate frames! And, you wrote it yourself, so it's free!

Dealing With Audio

Most audio libraries will let you stream to a file, rather than the speakers. You can then drag the audio file into your video feed. You will need to tweak it a bit to get it to synch properly with the video. This is almost impossible to do with command line video software, so it's good to use an editing suite for this.

Exporting Formats and Codecs

Exporting, you probably want a web-portable video. In which case, you probably want to consider the new HTML5 `video` tag primarily, so that you can upload it to your blog/website, and maybe also a Youtube-friendly format. The video tag allows you to have a stack of different formats, and uses the first one that your browser supports. It's a good idea to have a

couple, just in case. In OpenShot, go to the "advanced" tab when exporting. These 3 should do the trick for everything:

- *format - video codec - audio codec*
- ogg (Mozilla/Google/Opera) - libtheora - libvorbis
- WebM (HTML5; Mozilla, Opera, Google) - libvpx (VP8) - libvorbis
- mp4 (Quicktime/Safari) - libx264 - libfaac

For other formats, there's a list of codecs used by OpenShot:

http://en.wikipedia.org/wiki/OpenShot_Video_Editor#Video_formats_and_codecs

In HTML5 you can use all 3 of these as fall-backs inside the video tag (I left out the .ogg here):

```
<video width="512" controls>
  <source src="rtvideo_test.mp4" type="video/mp4">
  <source src="rtvideo_test.webm" type="video/webm">
    Your browser is not HTML5-compliant.
</video>
```

Debugging Shaders

The most annoying type of bug in graphics programming is when you don't see anything. **Where is my triangle?**

Unfortunately you can't print a number from a shader, but you can "print" a colour, so most of our value-checking is going to be by setting the fragment shader outputs to a value that we want to test and then looking at it to guess the number.

There are one or two GLSL "debuggers" around, but I haven't had any success with any of them in OpenGL 4.

Much like debugging code, the trick is to isolate each variable, and test one at a time. If you try testing the whole thing you will pull your hair out and waste a lot of time. This is the general process:

1. allow one variable to change value
2. make all other variables some constant value
3. work out the expected result on paper
4. check the value of the variable being tested to see if it matches expected (paper) result
5. repeat process for next variable

Step 0: Set Up and Check Your Shader Logs

As in the shaders article, make sure that you are printing your compile and linking logs somewhere. This is an absolute must, as they will tell you when compilation (bad code) or linking (mismatch between in and out variables) fails, as well as the file and line number that caused the problem.

Step 1: Get a White Shape Drawn on Screen

Set your fragment shader output to `= vec4 (1.0, 1.0, 1.0, 1.0);` (white, with alpha 1.0). If you can't see your shape at all, then the problem isn't in your fragment shader, but somewhere else:

1. Disable back-face culling before drawing the shape. Perhaps your triangles are defined in frack to bunt winding order?
2. I often forget to set a matrix uniform, or set it to the wrong location. When this happens the value is 0 by default - and will "disappear" anything that you try to draw. You can test this by **removing your matrix uniforms** in the vertex shader i.e. `gl_Position = proj * view * model * vec4 (vp, 1.0);` becomes `gl_Position = vec4 (vp, 1.0);` You should see something on the screen if your triangle covers the clip space area of (-1:1, -1:1, -1:1)
3. Look at the first 3 points from your mesh. This may require opening a mesh file or printing them out when you read them. What are the values? Do they make a sensible triangle?
4. If you still get no shape on screen, then you are probably not sending your per-vertex attributes to the correct locations. If you have positions, normals, and texture-coordinates, for example, your vertex shader may look like this:

```
in vec3 vp; // position
in vec3 vn; // normal
in vec2 vt; // texture coordinate

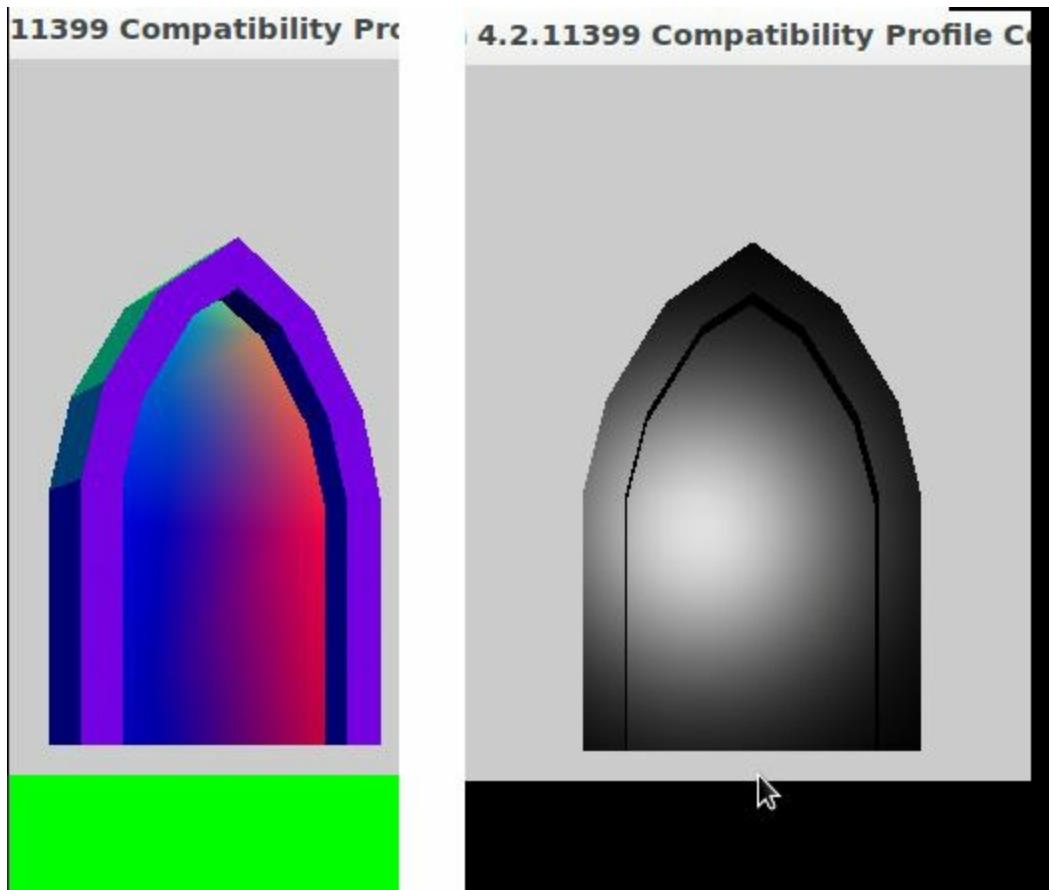
void main () {
    ... // rest of shader
}
```

There is no guaranty that `vp` is location 0. On occasion, the GLSL linker will put them in a different order. Check your shader logs (as described in the shaders article) to see which location it has assigned to each. The best solution to this muddle is to force the attributes to take a location that you specify. In GLSL 4.0.0 you can add a `layout (location = 0)` tag before `in vec3 vp;`, but to be backward-compatible with GLSL 1.5.0 the best idea is to bind them in C after compiling the shaders, but before

linking, with `glBindAttribLocation`. You can check your shader logs again to ensure that this worked.

5. Now that we know what the attribute locations are, make sure that you are specifying the correct location for each attribute pointer (`glVertexAttribPointer`) when you are setting up the VAO. I occasionally set all 3 to location 0 without realising.
6. Re-introduce your matrices, but set them all equal to an identity matrix. If the shape suddenly disappears again then it means that you are not sending the uniform values to the correct location. Check your `glGetUniformLocation` call. Make sure that this value is the same one that you are using in `glUniformMatrix`.... I often mix them up when copy-pasting from other variables. Make sure that your shader is in use before the `glUniformMatrix` call.
7. Get out a piece of paper and draw where your shape should be in relation to your view point. Check some actual vertex point values, and check the value that you are giving to your view matrix. Is the camera actually pointing at the shape? Is the shape behind the camera? Is the camera inside the shape? You can test your view and model matrices separately.
8. Are you combining your matrices in the wrong order? For column-major multiplication: `gl_Position = projection * view * model * point;`

Step 2: Use Colours to Check the Values in Your Shaders



Here I check the value of surface normals (left) and can see that the normals on the edges have been accidentally smoothed (because the colours change from blue to red). On the right I check the values of a specular lighting equation dot product, where I could see that the spot calculation was too small to light up the whole side of the mesh at once.

Do some expected calculations on paper, then **step-by-step**, check each value in the shader, and see if it matches your expected result. The first time that you will need to do this is probably Phong lighting that doesn't work properly.

Assuming we have our white shape, then we can safely override the colour output. We have a red, blue, and green channel which allows values from 0 to 1. Negative values will give you a 0.0 result i.e. `vec4 (-1.0, -1.0, -1.0, 1.0)` will

give you a black colour.

Test A Normal Vector

In a Phong lighting fragment shader, the first value that you'll want to test is your surface normal. You probably multiplied this by your model and view matrices in the vertex shader, to get it into eye space.

```
frag_colour = vec4 (normal_eye, 1.0);
```

This should colour your shape in shades of red, blue, green, and black. Fragments on right-facing parts of the shape should be mostly red (X values of 0:1 map to red values of 0:1). Left-facing parts should be black, because they have negative values. Mostly upwards-facing should be green (Y maps to G), and facing towards the viewer should be blue (positive Z in eye space). If this is not the case then you probably want to check if your normals are backwards (try negating the normal), if the conversion to eye space was correct, or if you have mixed up some variables. I would isolate this by outputting the normals in local space first, and checking that these colours are as expected (independent of the viewer).

You can use much the same process to check the positions of the light in eye space, and the vertex positions in eye space. Values greater than 1.0 will show as full brightness, so you shouldn't need to normalise them. You will also want to check the direction from the light to the surface.

Test a Floating Point Value

The next thing that you will want to check is the dot product, which takes two of the vectors that we tested above, and returns a floating point number. We can test this by setting all of the colour channels to the value, which will give you a greyscale gradient of results.

```
float dp = dot (direction_to_light_eye, normal_eye);
frag_colour = vec4 (dp, dp, dp, 1.0);
```

Being able to visualise your problem like this is a very powerful tool. It takes

the guess work out of your algorithm parameters. You will need to do this more often when setting up more advanced techniques. If you read academic literature from computer graphics you will see that it's quite common to show a rendering of the depth buffer, or of some intermediate calculation like this, as well as showing the before and after comparison shots.

Textures

Textures commonly take a bit of fiddling before they display. **Try displaying the texture coordinates as colours first.** It's quite easy to accidentally set them to a `vec3` instead of a `vec2`, either in the shader inputs, or when setting up the vertex attribute pointer. In this case they will read the wrong memory from the vertex buffers, and give you slightly-wrong results. The other common error here is mixing up the attribute locations with points - which might work but give you mangled textures. We resolved this issue earlier though!

If you have more than 1 texture input make sure that you have set a uniform integer value for each sampler. Otherwise they all default to value 0, which means "read from active texture slot 0". This usually only needs to be done once per shader, not every frame that your render. You don't need to send the texture itself into the shader as a uniform - just bind it after activating the appropriate slot number.

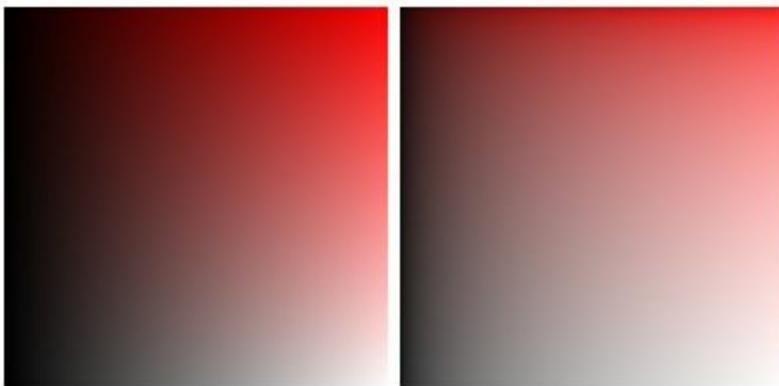
If you are using textures for special effects like specular mapping, try rendering the fragment with the sampled colours - this should tell you if the above set-up worked or not.

If you are generating any sort of texture from a shader (for shadow mapping or something similar), make sure that you can test this output by texturing a simple object with it. Usually rendering it to a screen-sized quad that you can switch on and off will do the trick.

Most Common Shader Bugs

- Trying to convert a `vec4` to a `vec3`: use the swizzle operator; `vec3 value = biggervalue.xyz;`
- Trying to convert a `vec3` to a `vec4`; add a W component; 0.0 for directions and 1.0 for points; `vec4 biggervalue = vec4 (value, 1.0);`
- Directions have a 1.0 as the W component of a `vec4`. Put 0.0
- Positions have a 0.0 as the W component of a `vec4`. Put 1.0
- Two vectors are compared or combined, but they are in different spaces (e.g. one in world space, one in eye space). Post-fix your vectors with their space to help you remember.

Gamma Correction



The range of colour for red with no correction (left), and with the gamma correction to output voltages (right). If you're looking at this in black and white - have a look at my on-line tool; <http://antongerdelan.net/colour/>.

In computer graphics we tell the video driver to display a colour as a mixture of red, green, and blue. Each colour component has a factor of 0 to 1. This is a linear range of colour, and translates to a linear range of voltages delivered to the computer's monitor or display. In reality the linear range of voltage does not correspond to a linear range of output colours. Without modification the colour range becomes mostly black through the lower ranges. We can correct this by anticipating the non-linear output of the monitor and correcting it. In previous years, monitors had a larger range of response values, called the **gamma** value, but all modern displays have a gamma value of 2.2. Any software that loads and displays images does this correction and stores it in the image, using a range of values called sRGB (standard red, green, blue), that also takes normal computer room lighting into account. We can manually add a correction for the gamma response:

```
frag_colour.rgb = pow (frag_colour.rgb, vec3 (1.0 / 2.2, 1.0 / 2.2, 1.0 / 2.2));
```

This gives us a much richer range of colours in our output to the display. The reader can see an interactive demonstration of this with an on-line tool: <http://antongerdelan.net/colour/>.

Image files already have this correction, so if you're sampling a texture in your shader the above correction will be horribly wrong. In this case, you can

either first convert your image to a linear range:

```
texel.rgb = pow (texel.rgb, vec3 (2.2, 2.2, 2.2));
```

Or, you can use the rather nifty built-in sRGB colour range when you create your texture. This will give you a kind of automatic colour correction:

```
glTexImage2D (  
    GL_TEXTURE_2D,  
    0,  
    GL_SRGB_ALPHA,  
    x,  
    y,  
    0,  
    GL_RGBA,  
    GL_UNSIGNED_BYTE,  
    image_data  
);
```

If you are also using a framebuffer that writes to a texture, make sure to use the sRGB colour range there too - then you get automatic gamma correction and you don't have to do any manual correction. With either of these approaches you should see a much richer range of colours, and you no longer have to mix in shades of blue and green to get a lighter red.

OpenGL Extensions and the Debug Callback

New OpenGL features are released as "extensions" - optional plug-ins - before they are incorporated into the new core OpenGL releases. This means that you can access some cutting-edge features early. It also means that if you are limited to an older version of OpenGL, you still might be able to use features of the current latest version, which is often the case for older graphics chips.

If you know of functionality that you want to use, but are not sure if it is available for your target versions of OpenGL, then you can look it up on the [OpenGL Wiki](#), which gives you a quick summary of what version of OpenGL it was adopted by. If it has been adopted in your version of OpenGL then you don't need to do anything - it's not a separate extension any more. If your version is earlier than that given, you can get the extension name string there, and we can check if that extension exists on your system.

Extensions are created by the architecture review board, as well as by various different vendors. The name of each extension is prefixed by the group that released it. You can find a full list of extensions at <http://www.opengl.org/registry/>. If you click on a link in the extensions list, you will get a text file description of the extension. This gives you:

- The name string for the extension (we need this).
- Who the authors and contact people are.
- Minimum version of OpenGL and any other extensions required.
- A brief description of what it does.
- Declarations of new functions introduced.
- A technical discussion, and examples of usage.

Check for Extension - the Debug Callback

If you're using the GLEW library (GL Extension Wrangler) then we can finally use it for its namesake purpose. Somewhere near the start of your programme, after initialising GLEW, you can check for all of the extensions of interest.

As an example, let's check for the [KHR_debug](#) output extension, which is incredibly useful, but has only recently been adopted in OpenGL 4.3. If your drivers don't support 4.3 yet, then it's definitely worth your while to try to use the extension anyway. If you do have 4.3 or newer, it doesn't do any harm to check for this extension anyway - it should return success.

There are several ways to check for, or get a list of, extensions supported by your system. A convenient way to is ask GLEW, which creates a list of `typedefs` for the names of all of the supported extensions on your system, prefixed with `GLEW_`.

```
if (GLEW_KHR_debug) {
    int param = -1;
    printf ("KHR_debug extension found\n");
    glDebugMessageCallback (debug_gl_callback, &param);
    glEnable (GL_DEBUG_OUTPUT_SYNCHRONOUS);
    printf ("debug callback engaged\n");
} else {
    printf ("KHR_debug extension NOT found\n");
}
```

You need to enable a debug context for this extension to work. Before creating your window with GLFW, call this hint: `glfwWindowHint (GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);`. In my game I don't always want to run in (potentially slow) debug mode, so I have a switch to skip all these commands unless I explicitly enable debug mode.

The Callback

When the extension was found we give it the name of a callback function to use whenever an error is detected. This is far more useful than the clunky checks for GL errors that you would have to do after each and every GL function is called. Now it will just call our function, give it a message, a code, and a severity level. If we want to, we can filter these by severity or type (read the extension documentation or the 4.3 Core Specification for details). I'm just going to print everything. I set up a few arrays so that I can turn the code numbers into strings of text to make it a bit more readable.

```
const char* sev_str[] = {
    "severity: HIGH",
    "severity: MEDIUM",
    "severity: LOW"
};

const char* source_str[] = {
    "source: API",
    "source: WINDOW SYSTEM",
    "source: SHADER COMPILER",
    "source: THIRD PARTY",
    "source: APPLICATION",
    "source: OTHER"
};

const char* type_str[] = {
    "type: ERROR",
    "type: DEPRECATED BEHAVIOUR",
    "type: UNDEFINED BEHAVIOUR",
    "type: PORTABILITY",
    "type: PERFORMANCE",
    "type: OTHER"
};
```

And the actual callback:

```
void debug_gl_callback (
    unsigned int source,
    unsigned int type,
    unsigned int id,
    unsigned int severity,
    int length,
    const char* message,
    void* userParam
) {
    int src_i = source - 0x8246;
    int typ_i = type - 0x824C;
    int sev_i = severity - 0x9146;
```

```
fprintf (
    stderr,
"%s %s id: %u %s length: %i %s userParam: %i\n",
source_str[src_i],
type_str[typ_i],
id,
sev_str[sev_i],
length,
message,
*(int*)userParam
);
}
```

Introduce Some Errors to Test

You can try mangling a few of your OpenGL function parameters to generate some errors (it shouldn't be too hard). I found this nice collection of sample errors at <http://www.lighthouse3d.com/cg-topics/error-tracking-in-opengl/>, and they definitely work:

```
/* calling invalid enum */
glEnable (GL_LINE);
/* value out of allowed range */
glEnableVertexAttribArray (GL_MAX_VERTEX_ATTRIBS + 1);
/* invalid buffer id */
glBindBuffer (GL_ARRAY_BUFFER, -1);
```

If you don't see any output, double-check that you are creating a debug profile properly - maybe the function call is in the wrong place.

Error Messages

The actual errors thrown, and the messages that go with them, depend on the implementation. I get different messages on my work computer with AMD, than with my home computer with Nvidia. Some of the error messages are a bit vague and unhelpful, or not a real error, but most of them are very helpful.

```
source: API type: ERROR id: 1001 severity: HIGH length: 71 glEnable parameter  
has an invalid enum '0x1b01' (GL_INVALID_ENUM) userParam: 1
```

If you've used the older error-checking functions before then this will be a huge relief. The only problem is narrowing down which line in your code actually fired the error callback. Your best bet is to sprinkle around a few printouts or break-points at different points in your code. With the synchronous output mode enabled (see first code block) it should be no trouble narrowing this down.

Other Extensions

If you're running an older version of OpenGL, for example, 2.1, and still want to use the modern interface, these extensions probably exist, and are worth checking for too:

- `GLEW_ARB_vertex_array_object` - VAO support
- `GLEW_ARB_framebuffer_object` - check for framebuffer extension and `glGenerateMipmap`
- `GLEW_ARB_uniform_buffer_object` - UBO support

In my video game, I support OpenGL 2.1 and later, but only if it has VAO support. This means that I can support about 95 percent of OpenGL machines without changing my core code from the modern paradigm.

The framebuffer extension allows post-processing effects, but it also tells you if you can use the super-efficient hardware mipmap generation. Earlier functions (see the way that the SOIL library creates mipmaps) were incredibly slow.

UBO support is fantastic if available. Unfortunately a poll of my friends' computers that I want to run my game on only gives me about 80% support, so I can't really use it. It would take a huge amount of structural code to support both UBOs and simple uniforms.

There are a number of sites around the web which collect statistics on extension support over different systems. You can also find tools to query all available extensions if you don't feel like writing this code yourself.

Uniform Buffer Objects and Buffer Mapping Functions

A **uniform buffer object** (UBO) is an alternative to the uniform updates that we have been using thus far. They have the following advantages:

- Can share the same data between multiple shader programmes.
- Can conveniently group collections of uniform variables.
- Can efficiently swap different sets of uniform variables.
- Tidier than managing collections of individual uniform locations.

In short, this is going to make your OpenGL code a lot tidier and more sane, and will give you a potentially huge performance boost if you use it for things like camera matrices and lights' variables, which you would otherwise have to resend to all your different shaders. You'll probably not bother using an UBO for individual uniforms that are only used in one shader.

The UBO is available in OpenGL 3.1 onwards. There is a special layout version of the UBO called `std140`, which is even tidier to manage, but a little tricky to set up correctly. UBO has pretty good support on earlier versions of OpenGL too - you can ask GLEW if the `ARB_Uniform_Buffer_Object` extension is available on the system. I'm building a video game that supports OpenGL 2.1 onwards, and I can almost get away with using it - about 8/10 of the machines that I target support the extension. You won't find support on other implementations of GL quite yet, so for broad platform support you'll probably be stuck with the old, ugly uniforms.

Queries

It's a good idea to query for available support first.

If we want to know if we can use UBO on a machine using an earlier version of the GL:

```
if (GLEW_ARB_uniform_buffer_object) {  
    printf ("GLEW_ARB_uniform_buffer_object = YES\n");  
} else {  
    printf ("GLEW_ARB_uniform_buffer_object = NO\n");  
}
```

We should get the total number of available binding points for our different UBOs:

```
GLint blocks = 0;  
glGetIntegeriv (GL_MAX_UNIFORM_BUFFER_BINDINGS, &blocks);  
printf ("GL_MAX_UNIFORM_BUFFER_BINDINGS = %i\n", blocks);
```

There are several other related variables that you can query with `glGetIntegeriv()` if you want to know the maximum block size available, and that kind of thing.

Creating the Buffer

Okay, I'm going to modify my skybox/environment mapping demo. There I had two shader programmes - `monkey_sp`, which renders a Suzanne mesh, and `cube_sp`, which renders the skybox. Both use the same virtual camera matrices. We will make one UBO "block" to share between them.

I'm using my own code to represent a `mat4` here - it's just a `struct` containing an array of 16 `floats`. You might use another representation. My camera UBO will have two 4X4 matrices in it - 32 `floats` in total.

```
GLuint cam_block_buffer;
 glGenBuffers (1, &cam_block_buffer);
 glBindBuffer (GL_UNIFORM_BUFFER, cam_block_buffer);
 glBufferData (GL_UNIFORM_BUFFER, sizeof (float) * 32, NULL, GL_DYNAMIC_DRAW);
```

So, generating the buffer object is exactly the same as with VBOs. Here I want to store 2 4X4 matrices in the buffer; the view and projection matrices - a total of 32 `floats`.

Each UBO block needs its own binding point number - we found out the maximum number earlier. Our first UBO we will make number 0, which is the default, but if we want to use more UBOs we would need to inform both shader programmes of each binding point:

```
int block_id = 0;
GLuint uniform_block_index_monkey =
    glGetUniformLocation (monkey_sp, "cam_block");
glUniformBlockBinding (monkey_sp, uniform_block_index_monkey, block_id);
GLuint uniform_block_index_cube_sp =
    glGetUniformLocation (cube_sp, "cam_block");
glUniformBlockBinding (cube_sp, uniform_block_index_cube_sp, block_id);
```

This is another horrible, clunky, confused, OpenGL convention, with too many integer identifiers. You see that we first retrieve an unique "index" from each shader programme by giving it the name of the UBO in the shader. Once we have that index, we use that to bind our block number of 0 to it.

Buffer Layout

The UBO works a lot like a vertex buffer object. You have a big block of memory that you copy your data into. In the basic version, you still have to get the location of each and every uniform variable within the buffer though - limiting! In the `std140` layout, however, you don't need to do this - the layout is in a fixed order, and the order of appearance in the shader code mirrors this. It would be really great if all shader variables were assigned locations based on order of appearance, but I digress! We will therefore use `std140`. The actual memory layout used by `std140` is a little bit tricky, and took me a while to get right.

Using the Older Interface

Updating buffer data with `glBufferSubData` works as follows:

```
glBindBufferBase (GL_UNIFORM_BUFFER, block_id, cam_block_buffer);
glBufferSubData (
    GL_UNIFORM_BUFFER, 0, sizeof (float) * 16, proj_mat.m);
glBufferSubData (
    GL_UNIFORM_BUFFER, sizeof (float) * 16, sizeof (float) * 16, view_mat.m);
```

Using the Newer Interface

Alternatively, we can use the new "map" buffer functions, which should be more efficient as you deal with larger buffers. These have a similar interface, but instead of providing a pointer to modified data, we are given a pointer, and modify the data behind it.

```
glBindBufferBase (GL_UNIFORM_BUFFER, block_id, cam_block_buffer);
float* cam_ubo_ptr = (float*)glMapBufferRange (
    GL_UNIFORM_BUFFER,
    0,
    sizeof (float) * 32,
    GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT
);
memcpy (&cam_ubo_ptr[0], proj_mat.m, sizeof (float) * 16);
memcpy (&cam_ubo_ptr[16], view_mat.m, sizeof (float) * 16);
glUnmapBuffer(GL_UNIFORM_BUFFER);
```

The parameters at the end are of interest. We can actually set the map up to read rather than write memory. In this case we want to write, so `GL_MAP_WRITE_BIT`. The next flag hints that we are going to replace all of the data in the buffer. For basic buffer mapping, you must call `glUnmapBuffer` after the changes have been made.

In both cases, I've copied the array of floats from my projection matrix into the first part of the buffer, and the view matrix into the second half. That's fine for 4X4 matrices. But variables of other sizes - float and vectors - can be tricky. `std140` has a strict layout scheme, and expects variables to appear in multiples of 4 floats. Basically - if all your vectors are `vec4`, you will have no problem - everything will work as expected. If you have `vec3` then you will have to add a gap (the size of 1 `float`) for padding, or it will read the wrong memory for your shader variables. Any scalars will be expected to fill in these gaps. If you get into trouble here it's worth comparing to the relevant section in the OpenGL specification (it's only a couple of paragraphs) to make sure that your data matches the expected layout.

Shaders

All you need to change is to remove your existing uniform variables (I called my camera matrices `P` and `V` for projection, and view, respectively. If you put them into a new shader block - it looks like a C `struct` - then it should work now.

```
#version 400

in vec3 vp;
//uniform mat4 P, V; // old uniforms

/* new virtual camera block */
layout (std140) uniform cam_block {
    mat4 P;
    mat4 V;
};

void main () {
    gl_Position = P * V * vec4 (vp, 1.0);
}
```

Note the keywords for using `std140`. Then the GLSL compiler knows to expect memory in a fixed order, with 16 `floats` for `P` first, and `V` second. We use the variables in exactly the same way as before. Even though our block has a name, we don't need to reference this anywhere in our shader code. We did use this name when we first set up our binding points though, if you recall. You can copy and paste this uniform block into any other shaders where you use the camera, and it will reference the same memory - provided you set up the binding point for each shader programme, like we did at the beginning.

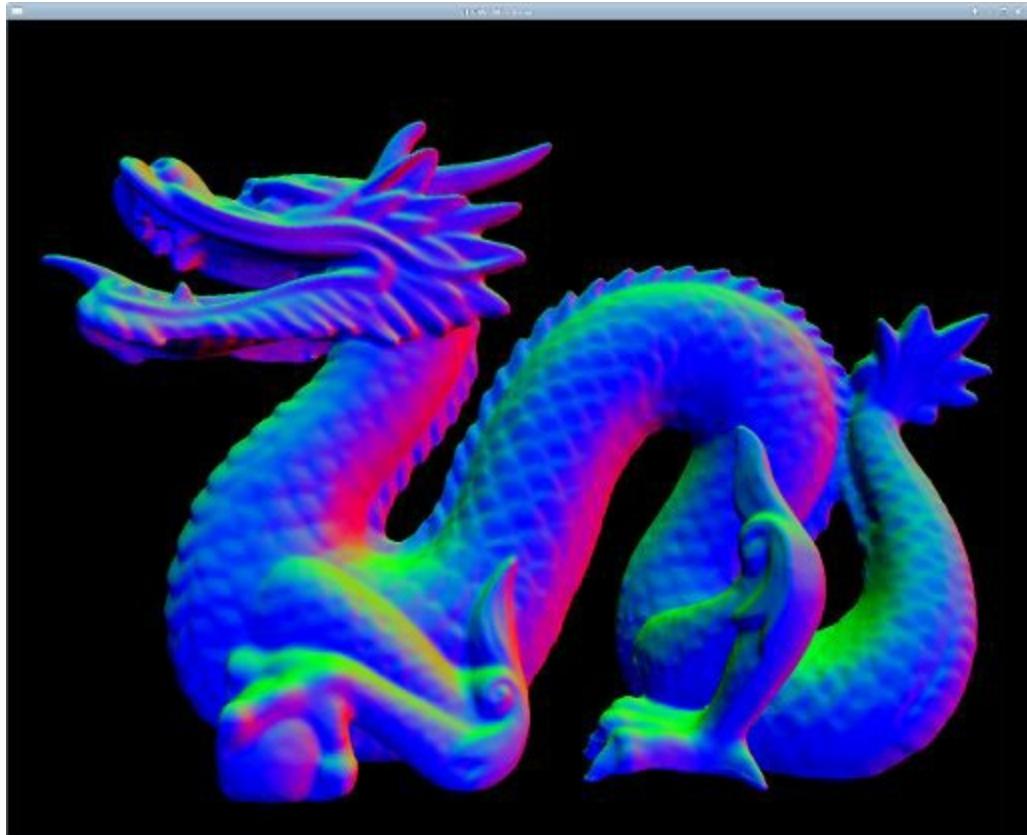
Overview

There is a little bit more initial setup than with basic uniforms, but as you share your data with more and more shaders this requires much fewer lines of code overall, as you just need to bind the new shader programmes to the block once.

I found a huge speed-up in my projects, just from eliminating all of the repeated uniform updates.

As far as I can tell, you should be able to replace all of your `glBufferData` and `glBufferSubData` function calls with the newer mapping equivalents, for vertex buffer objects too, with no drawback, other than a bit of extra code to manage. Have a look at the Quick Reference Card for a list. Replacing my older buffer data functions for UBOs with the map didn't make any difference for smaller projects, but I expect it will scale much better for larger operations. There are a whole range of different variations of these functions being introduced into OpenGL 4.4 core onwards, which significantly reduce the driver overhead for improved performance and data efficiency. For more detail on this topic, have a look at the combined presentation from Game Developers Conference 2014 - [Approaching Zero Driver Overhead](#) by experts from NVIDIA, AMD, and Intel.

Importing Meshes



The Stanford dragon. The mesh had some extra junk in it (an extra mesh) that I deleted before exporting.

I was originally going to write an article about importing a simple file format like Wavefront .obj files. Essentially all mesh files are just different ways to give you a list of vertex points that you can copy into buffers. Most of them are absolutely horrible, and there's no real consistent standard, so I don't think that there's much to gain from writing a tutorial examining any one file format. In fact, I think that we can ignore the inner details of all of the formats, and use a library to parse them all for us. The best OpenGL mesh importer is [AssImp](#). It loads a whole range of different formats, and presents them all to you in the same way, so that you can loop through and pull out the vertex data for your buffers.

Choosing a Mesh File Format

Unfortunately OpenGL doesn't have any built-in mesh loading functionality. Nor does it have a *de facto* mesh file format. So you come to a project decision - choosing a file format. In the simplest case we just want to read a list of vertex positions, normals, and texture co-ordinates from a file, and copy them into buffers. Most 3d modelling software won't export a flat file, but they will export Wavefront .obj, which isn't much more complex than that. - we could knock together a parser in half an hour. The other simple option would be an XML file - we could use existing XML parsers to read in our vertex points. But what if we want to include support for animations or complex materials? .obj files don't have that support. Consider the needs of your project, and balance these against the tools that you use. For example:

Mesh formats; *format - modellers export - animations - import complexity*

- plain file - none - difficult - simplest
- JSON - plug-in/converter - difficult - very simple
- Wavefront .obj - all - no - simple
- Collada .dae - most - yes - moderate
- various XML - most - maybe - moderate
- Autodesk .fbx - common - yes - quite complex, unreliable
- Blender .blend - Blender - yes - complex, unreliable
- Lightwave .lwo - Lightwave - yes - complex, unreliable
- Autodesk .3ds - 3ds Max - yes - complex, unreliable
- DirectX .x - common - yes - complex, out-of-date
- Quake 3 .md3 - plug-in - yes - complex, out-of-date

I've tagged a few of these formats as "unreliable" - this indicates that there might be several different versions of each format in existence. You will need to be very careful to only export a particular version of a mesh format for your project with these formats - you don't want to maintain an importer that supports a multitude of different versions. Even a well-established library like AssImp does not uniformly support all variations of each file format. Also note that not all mesh formats are well defined - some formats can actually be

interpreted ambiguously (yeah...). The more complex the format, the more nested layers there will be in them, and the longer it will take your parser to assemble the final mesh into buffers. If you're working with artists, then they are going to have a particular tool set that may influence the choice; you might consider importing the native format saved by those tools. I've listed a few of these.

I'm going to use a Collada .dae file, but you can use any format that AssImp supports. Collada is intended to be an intermediate mesh format; it's XML based so it's a bit slower to import than a more compact option, but I don't care about that right now. It supports animations, which I'll use later on, and can be exported from most modellers. I'm using the Blender modeller. The only hitch I found was that the version that installed with (2.64 or so) didn't have an exporter. I downloaded the very latest version from [the website](#) (version 2.65a), and all is good.

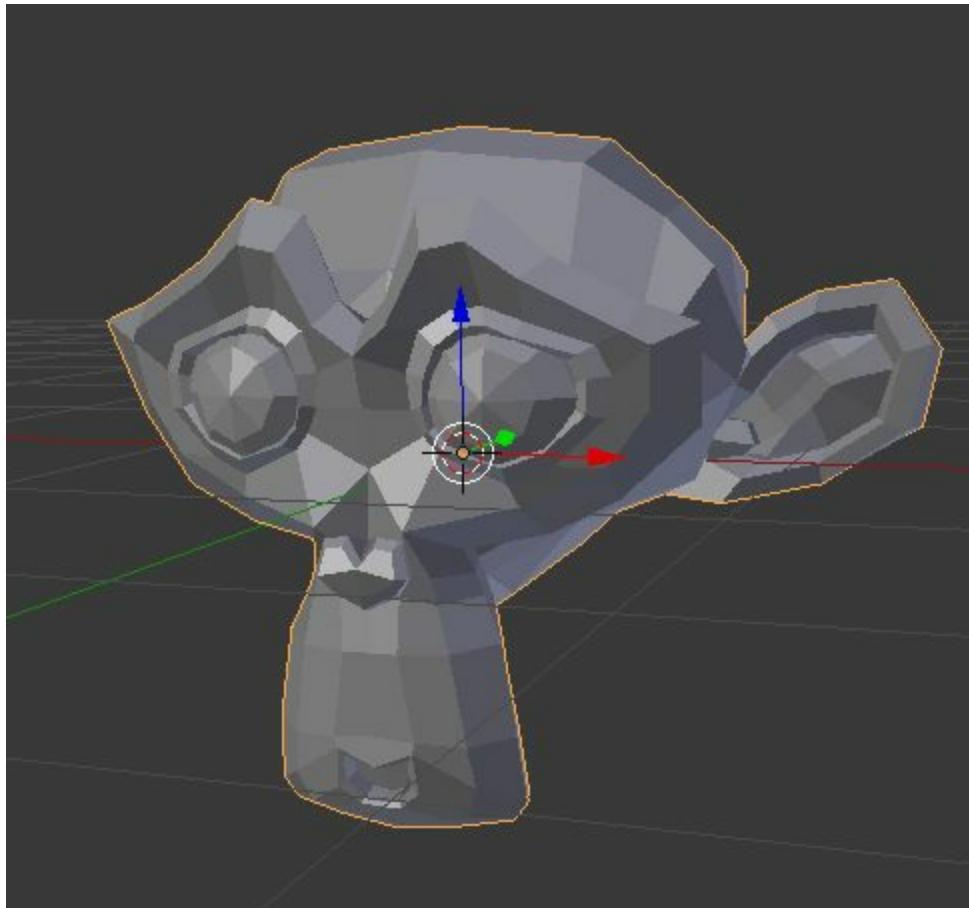
Get a Test Mesh

There are a number of fairly famous "standard" test meshes that you can use. If you're writing a research paper you would tend to use one of these so that it can be quickly visually compared to other research outputs, or benchmarked for performance. Most of these meshes have been scanned from real objects, so contain enough interesting shapes to shake out the bugs in your export/import process - you'll notice if the Stanford Bunny has an extra hole in it, but you might not notice on a mesh of your own. Note that even some of these famous scans contain some glitches - you'll see them noted on their websites. **Hint: You might want to start with a single triangle first, then a quad of two triangles, before attempting to import a larger test-mesh.** Keep in mind that some of these meshes are HUGE.

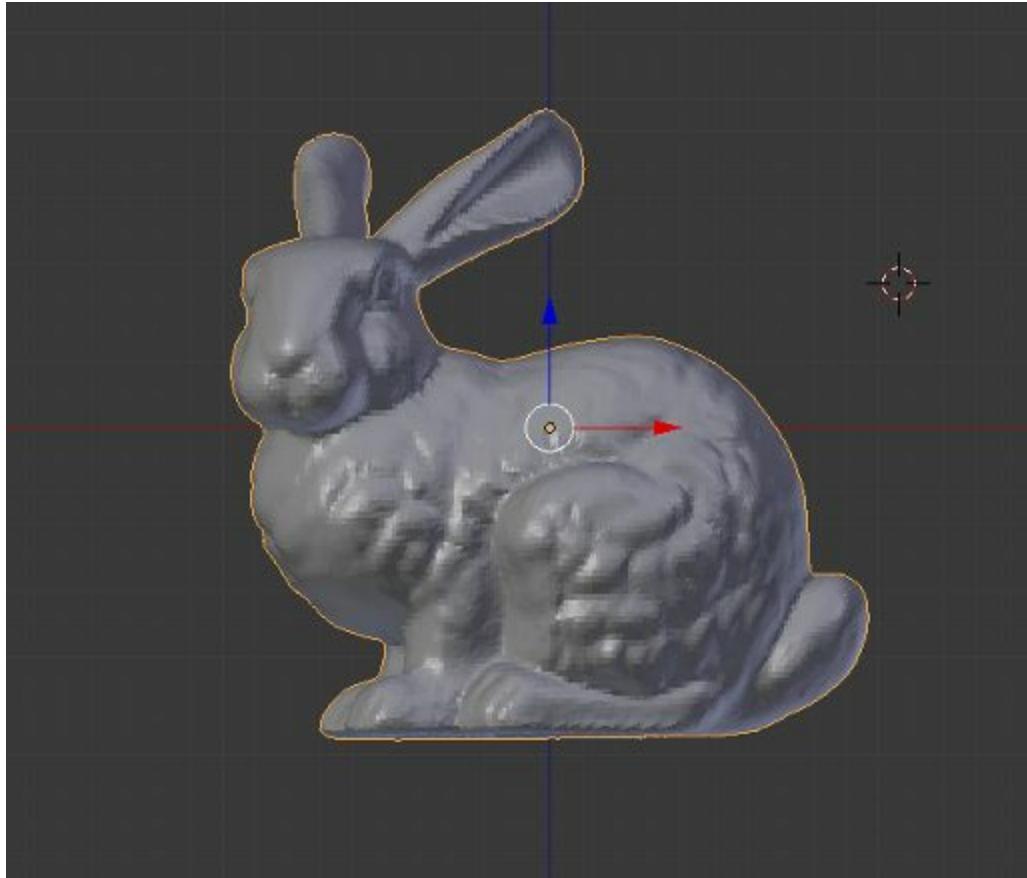
Test meshes; *name - vertices - URL:*

- Blender's "Suzanne" (chimpanzee) - ~1500 - built-in to Blender
- Eurographics "Phlegmatic Dragon" - ~20,000 - [EG'07](#)
- Stanford Bunny - 35,947 - [Stanford 3D Scanning Repository](#)
- Stanford's Happy Buddha - 543,652 - [Stanford 3D Scanning Repository](#)
- Stanford Dragon - 566,098 - [Stanford 3D Scanning Repository](#)
- Stanford's Armadillo (action figure) - ~115,000 - [Stanford 3D Scanning Repository](#)
- Stanford's Lucy (angel) - 14,027,872 - [Stanford 3D Scanning Repository](#)
- XYZ RGB Inc. Asian Dragon - 3,609,455 - [Stanford 3D Scanning Repository](#)
- XYZ RGB Inc. Thai Statue - 5,000,000 - [Stanford 3D Scanning Repository](#)
- Utah(Martin Newell's) Teapot - ? -
<http://www.sjbaker.org/teapot/teaset.tgz>
- Utah's VW Bug - ? - ?
- Cornell Box - ? - [Cornell](#)
- Skeleton Hand - 327,323 - [Georgia Institute of Technology archive](#)
- Horse - 48,485 - [Georgia Institute of Technology archive](#)
- skeleton man - ~13,000,000 - [Georgia Institute of Technology archive](#)

- Puget Sound - 16,385x16,385 - [Georgia Institute of Technology archive](#)



The "Suzanne" mesh in Blender add → mesh → monkey.



The Stanford Bunny, imported into Blender from the hi-res .ply. The mesh is scaled down to a tiny size, so I scaled it up to a -1:1 "unit bunny", and rotated it so that the +y axis was upwards, and the -z axis was forwards (I wanted to see the face when I loaded it).

Exporting

3d modelling tools have their own parlance for describing a 3d mesh. Internally, they usually organise a mesh into a hierarchy of elements. The top-level "everything" level is called the **scene**. The scene often contains all sorts of extra-fluff that you absolutely will not need in your mesh file; a (potentially scripted) **camera**, **lights**, and one or more **meshes**. Formats like .obj won't store any of this, but just about every other format will (unless you tell it not to).

```
-scene
+camera
+light
+light
+mesh
+mesh
+mesh
```

The meshes are a high-level collection of per-vertex data and, **armatures** or "skeletons" associated with a mesh. A mesh can also have **materials**, which is a high level combination of textures and shaders to use when rendering.

```
-scene
+camera
+light
+light
-mesh
+armature
+vertex points
+vertex texture coordinates
+vertex normals
+vertex tangents
-material
+shaders
+textures
+mesh
+mesh
```

You can see why an XML type of file might be convenient for exporting this structured information. In the simple case we just have 1 scene, with 1 mesh, and maybe 1 armature, so all this structure is overkill. It gets deeper; each armature contains a hierarchy of **joints** or "bones" making up a structure for skeletal animation, and can have many key-framed **animations** associated with it. Each animation has several key-frames, etc. etc. All we are really

concerned with at the moment is getting to the lists of vertex points inside the mesh, but it's good to know that this sort of structure exists first, because most file formats will store it, and we'll need to recurse this structure to some extent to get our per-vertex data out of AssImp.

Ideally, we have a relationship where our file contains 1 scene, with 1 mesh, the contents of which we can stick into one set of vertex buffers. If you have a more complex "scene", divided into several sub-meshes, then you'll probably need to divide this between several vertex buffers; not quite so efficient for rendering any more, but you might need to do this if each part uses a different material (texture or shaders). It's generally a good idea to try and merge all of your sub-meshes (called "objects" in Blender) into a single mesh, and pack all of your textures into a single image. This way you can draw the whole object with a single draw call. If you can do this, merge everything in the modeller before exporting.

In Blender, I selected "export → Collada (.dae)". On the side panel I excluded the camera and lights. That's it! We really only need the positions, normals, and texture-coordinates for this tutorial, but you might like to export other bits and pieces.

Writing a Mesh Loader Function

Loading a File with AssImp

Download AssImp and install it somewhere. I compiled AssImp, and link against the static library file, and include the headers. When coding there are 2 APIs; one for C++, and one for C. I used the C interface. Note that the header files were named a bit differently than in the AssImp documentation.

```
...
#include <assimp/cimport.h> // C importer
#include <assimp/scene.h> // collects data
#include <assimp/postprocess.h> // various extra operations
#include <stdlib.h> // memory management
#define MESH_FILE "suzanne.obj" // file to load ...
```

I wrote a function that takes a file name, gives it to the AssImp importer, and gets a "scene" structure out. We will need to use a VAO, and know the number of vertex points, when we draw later, so I use pointer parameters to pass these variables back **by reference**. If you're using C++ you might prefer the reference operator instead of pointers.

```
bool load_mesh (const char* file_name, GLuint* vao, int* point_count) {
    /* load file with assimp and print some stats */
    const aiScene* scene = aiImportFile (file_name, aiProcess_Triangulate);
    if (!scene) {
        fprintf (stderr, "ERROR: reading mesh %s\n", file_name);
        return false;
    }
    printf (" %i animations\n", scene->mNumAnimations);
    printf (" %i cameras\n", scene->mNumCameras);
    printf (" %i lights\n", scene->mNumLights);
    printf (" %i materials\n", scene->mNumMaterials);
    printf (" %i meshes\n", scene->mNumMeshes);
    printf (" %i textures\n", scene->mNumTextures);

    /* get first mesh in file only */
    const aiMesh* mesh = scene->mMeshes[0];
    printf (" %i vertices in mesh[0]\n", mesh->mNumVertices);

    /* pass back number of vertex points in mesh */
    *point_count = mesh->mNumVertices;

    /* generate a VAO, using the pass-by-reference parameter that we give to the
       function */
    glGenVertexArrays (1, vao);
    glBindVertexArray (*vao);
```

...

At the top of my function I print out some debugging information. This is so that I can check if it loaded the mesh properly. These values are in the AssImp scene data structure. You can see the documentation at http://assimp.sourceforge.net/lib_html/structai_scene.html. Note that loading any of the supported meshes only takes that 1 function call - not bad. I told it to triangulate any mesh loaded, but this didn't seem to work, so I had to do this in the editor before-hand. There are a number of [additional "post processing" flags](#) that you can give to the function there. I set the value of my point_count parameter, so that I can use it outside the function, and also generate a VAO. Note that this is a pointer, so my use of it in the generation and binding functions looks a little different than what we've done before.

Copy Loaded Data into VBOs

Once I got the number of vertices in the mesh from AssImp: *point_count = mesh->mNumVertices; I ask if there are positions, normals, and texture coordinates. I assume that there should only be 1 set of texture coordinates per vertex, so I give it "index 0" for `HasTextureCoords()`. Now, we could [almost](#) get away with copying AssImp data directly into vertex buffers, but as we know, `glBufferData()` expects the memory that we give it to be in contiguous order (like an array). Unfortunately, not all of that data that we need from AssImp is in contiguous order in memory, so we need to carefully extract each float from AssImp, and arrange it in the correct order. I allocated some memory to do this, and free it after I copy to over with `glBufferData()`.

```
...
/* we really need to copy out all the data from AssImp's funny little data
structures into pure contiguous arrays before we copy it into data buffers
because assimp's texture coordinates are not really contiguous in memory.
i allocate some dynamic memory to do this. */
GLfloat* points = NULL; // array of vertex points
GLfloat* normals = NULL; // array of vertex normals
GLfloat* texcoords = NULL; // array of texture coordinates
if (mesh->HasPositions ()) {
    points = (GLfloat*)malloc (*point_count * 3 * sizeof (GLfloat));
    for (int i = 0; i < *point_count; i++) {
        const aiVector3D* vp = &(mesh->mVertices[i]);
        points[i * 3] = (GLfloat)vp->x;
        points[i * 3 + 1] = (GLfloat)vp->y;
        points[i * 3 + 2] = (GLfloat)vp->z;
    }
}
```

```

    }
    if (mesh->HasNormals ()) {
        normals = (GLfloat*)malloc (*point_count * 3 * sizeof (GLfloat));
        for (int i = 0; i < *point_count; i++) {
            const aiVector3D* vn = &(mesh->mNormals[i]);
            normals[i * 3] = (GLfloat)vn->x;
            normals[i * 3 + 1] = (GLfloat)vn->y;
            normals[i * 3 + 2] = (GLfloat)vn->z;
        }
    }
    if (mesh->HasTextureCoords (0)) {
        texcoords = (GLfloat*)malloc (*point_count * 2 * sizeof (GLfloat));
        for (int i = 0; i < *point_count; i++) {
            const aiVector3D* vt = &(mesh->mTextureCoords[0][i]);
            texcoords[i * 2] = (GLfloat)vt->x;
            texcoords[i * 2 + 1] = (GLfloat)vt->y;
        }
    }
}
...

```

Buffers are created in exactly the same way as in the Hello Triangle example. Make sure that you use the correct number of points, multiplied by the elements in the array i.e. 3 for positions, 2 for texture co-ordinates. In the `glBufferData()` functions, we give the size of our memory array in bytes, i.e `3 * *point_count * sizeof (GLfloat)`, as well as the address of the array/memory i.e. `normals` (each of our pointers).

```

...
/* copy mesh data into VBOs */
if (mesh->HasPositions ()) {
    GLuint vbo;
    glGenBuffers (1, &vbo);
    glBindBuffer (GL_ARRAY_BUFFER, vbo);
    glBufferData (
        GL_ARRAY_BUFFER,
        3 * *point_count * sizeof (GLfloat),
        points,
        GL_STATIC_DRAW
    );
    glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray (0);
    free (points); // free our temporary memory
}
if (mesh->HasNormals ()) {
    GLuint vbo;
    glGenBuffers (1, &vbo);
    glBindBuffer (GL_ARRAY_BUFFER, vbo);
    glBufferData (
        GL_ARRAY_BUFFER,
        3 * *point_count * sizeof (GLfloat),
        normals,
        GL_STATIC_DRAW
    );
    glVertexAttribPointer (1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray (1);
    free (normals); // free our temporary memory
}
```

```

if (mesh->HasTextureCoords (0)) {
    GLuint vbo;
    glGenBuffers (1, &vbo);
    glBindBuffer (GL_ARRAY_BUFFER, vbo);
    glBufferData (
        GL_ARRAY_BUFFER,
        2 * *point_count * sizeof (GLfloat),
        texcoords,
        GL_STATIC_DRAW
    );
    glVertexAttribPointer (2, 2, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray (2);
    free (texcoords); // free our temporary memory
}
if (mesh->HasTangentsAndBitangents ()) {
    // NB: could store/print tangents here
}

/* free assimp's copy of memory */
aiReleaseImport (scene);
printf ("mesh loaded\n");

return true;
}

```

Because I already generated, and bound, a VAO, I can set up the attribute pointer, and enable the attribute variable right away after copying the data into buffers. Note that I assumed attribute locations 0, 1, and 2, here. If your shaders use different input variables, you might want to change that to suit. You can see that it's possible to set up tangents in a VBO here as well (for normal mapping techniques). After I set up all my buffer object memory, I free the memory allocated by assimp and end the function.

Using the Loader Function

For testing, it's going to be helpful if you have a virtual camera so that you can move around; sometimes the centre of the object is behind the view, or the geometry is massive or tiny.

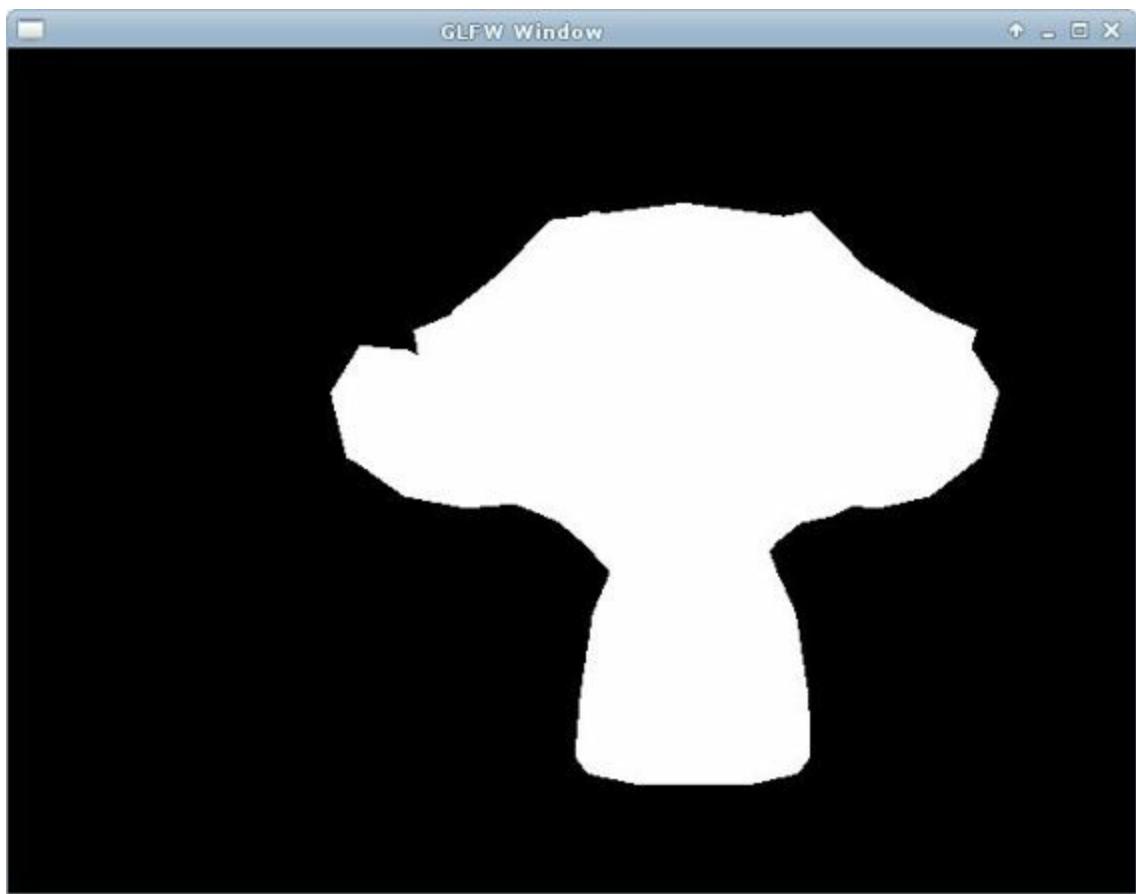
```
...
/* load the mesh using assimp */
GLuint monkey_vao;
int monkey_point_count = 0;
assert (load_mesh (MESH_FILE, &monkey_vao, &monkey_point_count));
...
while (!glfwWindowShouldClose (g_window)) {
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram (shader_programme);

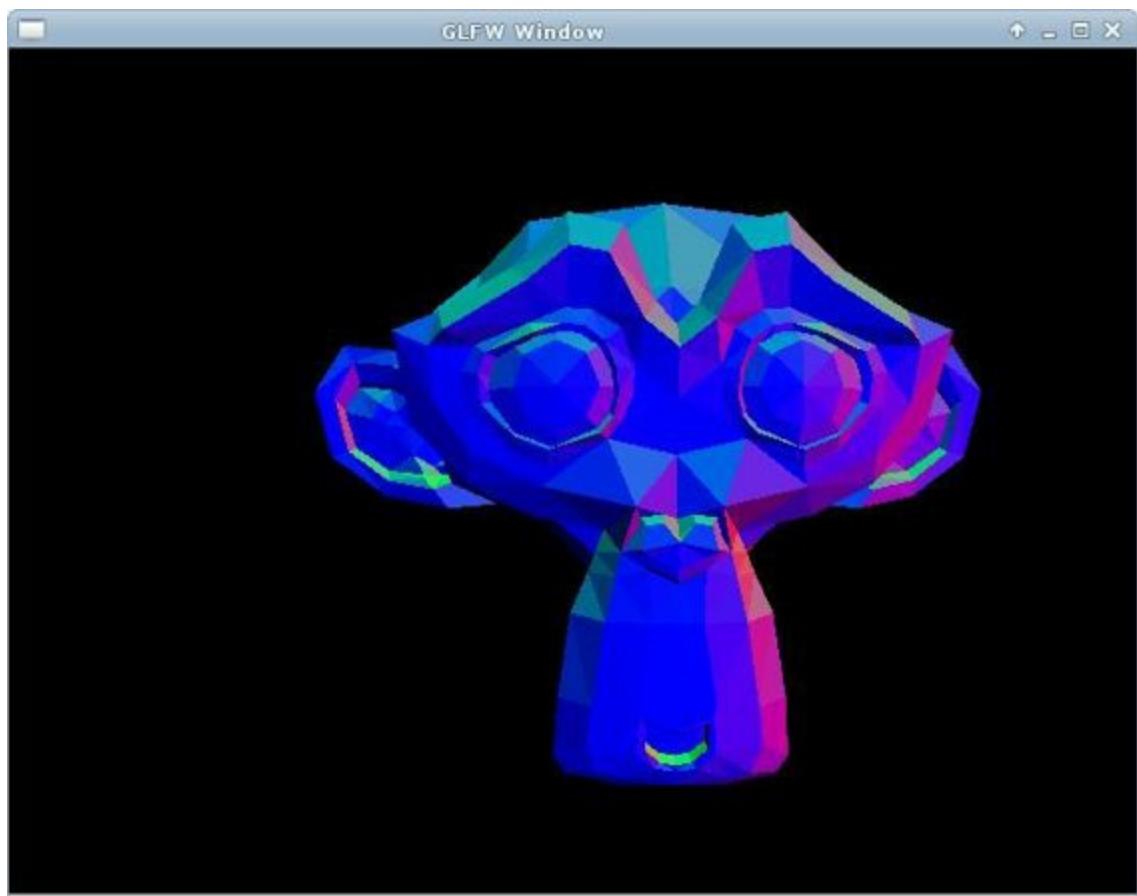
    /* set any uniforms here - camera matrices, etc. */

    glBindVertexArray (monkey_vao);
    glDrawArrays (GL_TRIANGLES, 0, monkey_point_count);

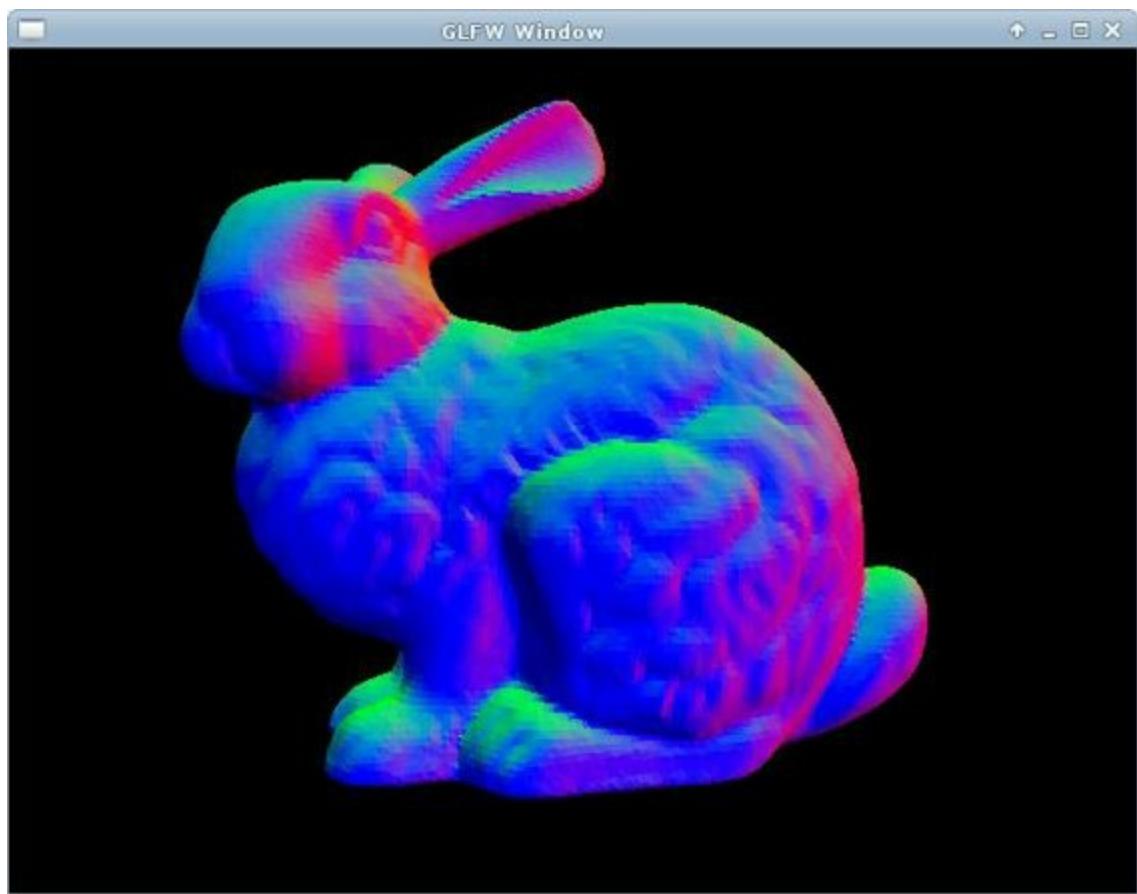
    glfwPollEvents ();
    glfwSwapBuffers (g_window);
}
...
```



I had to move the camera backwards to see my mesh, but it was there.



*I added in normals as a second vertex buffer, and sent them to the fragment shader where I used them as colours to test if they loaded properly. Because colours are not varying over each facet, we can tell that the mesh is **flat shaded**, which might not be an intended choice for a rounded mesh. You can ask the modelling tool to change the normals to **smooth shaded**.*



The Stanford Bunny, hi-res, with normals displayed. This is a good test that my export-import system is working.

Don't forget to link the AssImp library before building. If it's installed on your system path then you just need to link the `libassimp` library.

Common Mistakes

- "**There were no errors, but I don't see anything!**" - check your vertex positions; are the positions in the view of the camera? Is the camera inside the object (huge scale)? Is the object lying down, flat, to the camera? Try disabling back-face culling in case the object is inside-out. Try forcing a solid colour output (i.e. all white), in case the normals or texture co-ordinates are missing/broken.
- "**I see only 1 triangle**" - check the `glDrawArrays` function - are you giving it the correct number of vertices to draw? Are you telling it to draw triangles? Are you sure that your mesh is triangulated, and not in quads?
- "**I loaded the Suzanne mesh, but it has lots of holes in it!**" - You need to triangulate the mesh before exporting it, as we are drawing in triangles mode, not quads. **CTRL+T** in Blender.
- "**I loaded a mesh, but it looks inside-out.**" - You haven't enabled depth testing.
- "**The mesh mostly loaded, but some triangles are wrong.**" - Something is wrong with the ordering of the vertex points in the buffers. Try exporting a box, versus hard-coding a box with the same dimensions. Is the error in the export, import, and vertex-buffer packing stage? Did you give it the correct number of vertices to draw (`glDrawArrays`), and the correct size and dimensionality of each per-vertex variable (`glBufferData`)?
- "**My animations aren't loading!**" - We'll get onto this in the skinning tutorial, but try exporting to a different file format - I get mixed results with **.fbx** and **.blend** files, but the **.dae** files seem fine.

Loading More than One Mesh

If you want to draw a second copy of the same mesh, you can just re-use the same VAO, and draw it again after sending in a different model matrix uniform. But what if you want to load two different meshes? Our loader function will allocate new vertex buffer objects to hold the new mesh data, but we need to make sure that we have a new VAO to point to them, and a new point count variable.

```
/* load first mesh */
GLuint monkey_vao;
int monkey_point_count = 0;
assert (load_mesh ("suzanne.obj", &monkey_vao, &monkey_point_count));

/* load second mesh */
GLuint bunny_vao;
int bunny_point_count = 0;
assert (load_mesh ("bunny.dae", &bunny_vao, &bunny_point_count));
...
while (!glfwWindowShouldClose (g_window)) {
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ...

    /* draw monkey. also update uniform matrices here */
    glUseProgram (shader_programme);
    glBindVertexArray (monkey_vao);
    glDrawArrays (GL_TRIANGLES, 0, monkey_point_count);

    /* draw bunny. switch shader if required. update uniforms here. */
    glBindVertexArray (bunny_vao);
    glDrawArrays (GL_TRIANGLES, 0, bunny_point_count);

    glfwPollEvents ();
    glfwSwapBuffers (g_window);
}
```

Multi-Texturing

We can sample more than one texture in a fragment shader. This allows us to do a number of interesting effects. We can blend the two textures together based on some factor, or switch between them, or use the extra textures to provide some extra per-fragment detail for effects like lighting calculations.

Loading a Second Texture

Recall that the GL state machine has a number of "active" texture slots. If we are just using 1 texture then we active slot 0, and bind the texture that we want to use into it before rendering:

```
glActiveTexture (GL_TEXTURE0); // activate the first slot  
glBindTexture (GL_TEXTURE_2D, a_texture);
```

To use 2 textures at once, we active slot 1, and bind another texture into that.

```
glActiveTexture (GL_TEXTURE0); // activate the first slot  
glBindTexture (GL_TEXTURE_2D, first_texture);  
glActiveTexture (GL_TEXTURE1); // activate the second slot  
glBindTexture (GL_TEXTURE_2D, second_texture);
```

Simple enough! But of course, there's a tricky catch. If we add a second *sampler2D* into our fragment shader, then it will read from *GL_TEXTURE0* as well! Zero is the default value for any shader uniform.

```
in vec2 st;  
  
uniform sampler2D first_texture;  
uniform sampler2D second_texture;  
  
out vec4 frag_colour;  
  
void main () {  
    ...  
}
```

We need to give the second sampler uniform the value of 1 to tell it to read from *GL_TEXTURE1*. Sampler uniforms are treated as if they are integers. After linking your shader programme:

```
int first_sampler_location = glGetUniformLocation (shader_programme, "first_texture");  
int second_sampler_location = glGetUniformLocation (shader_programme,  
"second_texture");  
glUseProgram (shader_programme);  
glUniform1i (first_sampler_location, 0); // read from active texture 0  
glUniform1i (second_sampler_location, 1); // read from active texture 1
```

Blending Textures

Now that we can sample 2 textures, we can do fancy effects with them. I'm going to use the same texture co-ordinates for both textures. I could switch between them to show perhaps a building's texture in a game before, and after it has been damaged. Another fairly common effect is to use several textures for painting different areas of a landscape; grass, cobblestones, dirt, etc. On the boundaries of the different areas, most modern games will blend the grass and dirt together smoothly, to avoid a harsh transition. I'm going to do something like that; blend between 2 images based on a distance factor from left to right.

My "factor" of the first image to display is going to be 1.0 on its left-hand side, and 0.0 on its right-hand side. My texture coordinates go from 0.0 to 1.0 left-to-right, so I can just use $1.0 - st.s$ for the factor of the first texture, and $st.s$ for the factor of the second texture. My shader is going to look something like this:

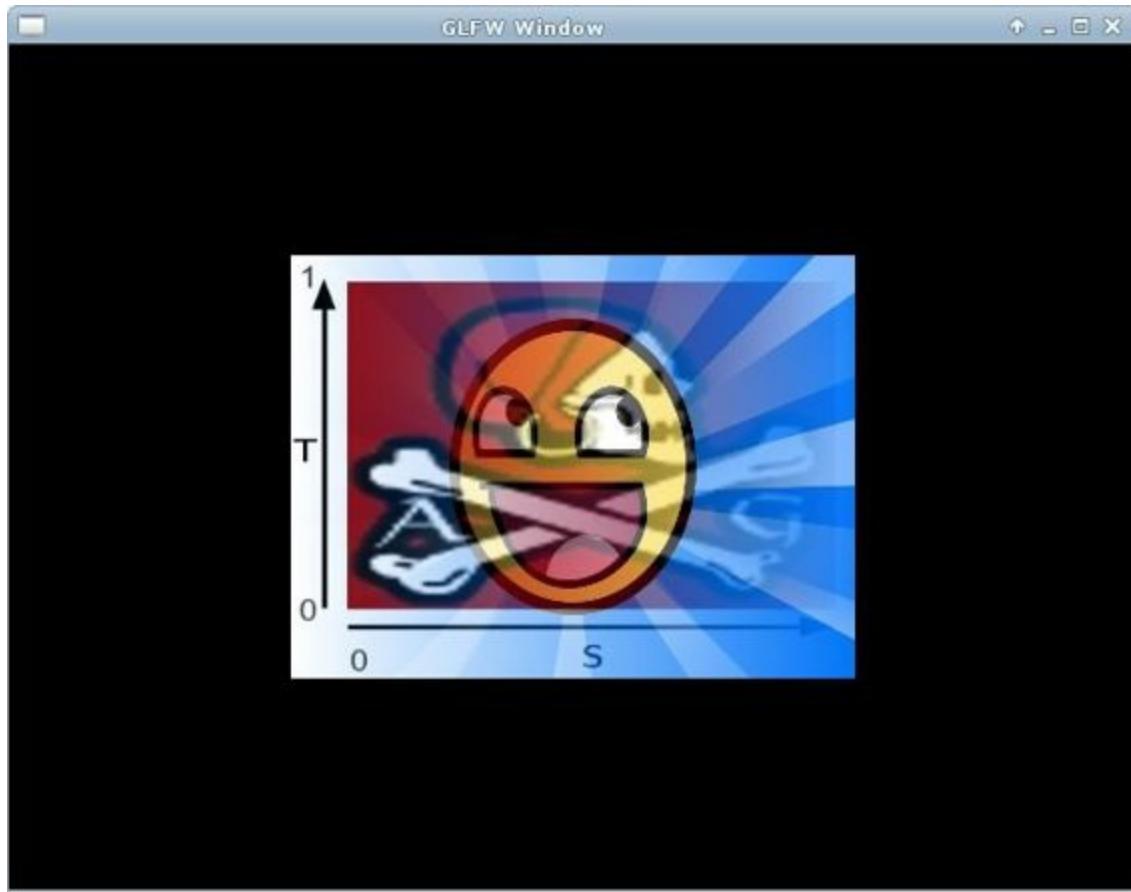
```
in vec2 st;

uniform sampler2D first_texture;
uniform sampler2D second_texture;

out vec4 frag_colour;

void main () {
    vec4 first_sample = texture (first_texture, st);
    vec4 second_sample = texture (second_texture, st);

    frag_colour = first_sample * (1.0 - st.s) + second_sample * st.s;
}
```



This kind of linear blend operation is actually very common in GLSL, so there is a function called `mix()` that we can use as a small short-cut:

```
in vec2 st;  
  
uniform sampler2D first_texture;  
uniform sampler2D second_texture;  
  
out vec4 frag_colour;  
  
void main () {  
    vec4 first_sample = texture (first_texture, st);  
    vec4 second_sample = texture (second_texture, st);  
  
    frag_colour = mix (first_sample, second_sample, st.s);  
}
```

Going Further

We don't always have to draw the extra textures - they can be exploited to store any kind of information that we want to load and use in a shader. In the next tutorial we will look at doing this to enhance Phong lighting shaders with specular maps, but textures can also store per-fragment normals (using r, g, and b channels as x, y, and z axes) or similar kinds of information.

Using Textures for Lighting Coefficients

We can combine Phong shading with multi-texturing to add some art power to our visualisations. Remember, in our Phong lighting equations we had 3 terms; diffuse, ambient, and specular light. In the terms we had properties of the light source; colour of the diffuse, specular, and ambient light, position, etc. We also had properties of the surface that the light was shining on; position, normal vector, and **coefficients of reflection** for ambient, diffuse, and specular light; K_a , K_d , and K_s , respectively. These were `vec3`s that determined what factor of each colour of light was reflected. We could set each of these as a constant per-mesh, and send it in as a shader uniform, or we can load it from a texture, so that each fragment can have its own value of K_d , K_s , and K_a .

Diffuse Maps

We can use the regular texture for a mesh as its κ_d value. I use the alpha value from the diffuse texture here as the alpha value of the fragment, but this will depend on our style.



I use the regular texture for my mesh as the "diffuse map"

My fragment shader is going to look something like the below (I'll add in the specular and ambient components later).

```
in vec3 pos_eye, norm_eye; // positions and normals in eye space
in vec2 st; // texture coordinates passed through from vertex shader
uniform sampler2D diffuse_map;
uniform vec3 light_pos_eye; // light position in eye coords
out vec4 frag_colour;
vec3 La = vec3 (0.8, 0.8, 0.8); // ambient light colour
vec3 Ld = vec3 (0.8, 0.8, 0.8); // diffuse light colour
vec3 Ls = vec3 (0.8, 0.8, 0.8); // specular light colour

void main () {
    vec4 sample = texture (diffuse_map, st);
    vec3 Kd = sample.rgb;
    vec3 surface_to_light_eye = normalize (light_pos_eye - pos_eye);
    float dp = dot (norm_eye, surface_to_light_eye);
    vec3 Id = Kd * Ld * dp;
    ...
    frag_colour = vec4 (Id + Is + Ia, sample.a);
}
```

Specular Maps and Emission Maps

Specular maps give you a lot of artistic control to creating parts of a mesh look shiny, and other parts dull.



I quickly drew this specular map image in GIMP. I wanted the hinges and the handle to shine a lot, and the stones to shimmer a little so I made them a less bright colour. Under a white specular light the handle will shine gold and the hinges will shine silver or metallic.

We need to add in a second texture here, and all of the other uniforms that we need for specular lighting.

```
in vec3 pos_eye, norm_eye; // positions and normals in eye space
in vec2 st; // texture coordinates passed through from vertex shader
uniform sampler2D diffuse_map, specular_map;
uniform vec3 light_pos_eye; // light position in eye coords
uniform float specular_exponent;
out vec4 frag_colour;
vec3 La = vec3 (0.8, 0.8, 0.8); // ambient light colour
vec3 Ld = vec3 (0.8, 0.8, 0.8); // diffuse light colour
vec3 Ls = vec3 (0.8, 0.8, 0.8); // specular light colour

void main () {
    // diffuse light intensity
    vec4 sample = texture (diffuse_map, st);
    vec3 Kd = sample.rgb;
    vec3 surface_to_light_eye = normalize (light_pos_eye - pos_eye);
    float dp = dot (norm_eye, surface_to_light_eye);
    vec3 Id = Kd * Ld * dp;

    // specular light intensity
    vec3 Ks = texture (specular_map, st).rgb;
    ...
    // rest of specular light equation goes here
```

```
...
    frag_colour = vec4 (Id + Is + Ia, sample.a);
}
```

We don't need the alpha component in the specular map, but I use it in my projects to provide an **emissive** factor - multiply the diffuse or specular map by this value and add it to the output, essentially skipping the lighting equation - creating "fully bright" areas.

```
...
// diffuse lighting here
...
vec3 sample_smap = texture (specular_map, st);
vec3 Ks = sample_smap.rgb;
float emission = sample_smap.a;
...
// specular lighting here
...
frag_colour = vec4 (Id + Is + Ia + Ks * emission, sample.a);
...
```

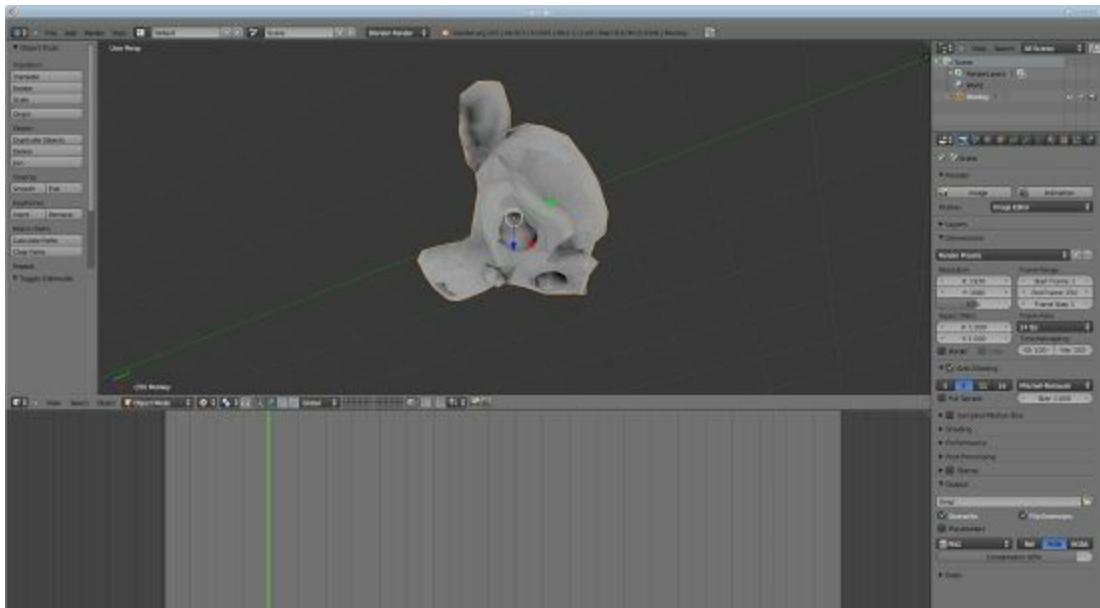
Other people prefer to use a completely separate texture for emission maps, but I try to squeeze everything I can into one texture sampling operation to make it a bit faster.



The result - as the door turns the specular highlights on the handle and hinges shine.

Ambient Occlusion Maps

You can also make a texture to store per-fragment ambient light coefficients. Remember that ambient light is supposed to model light bounced all over the scene. We can make this a little more realistic by limiting the ambient light that makes it into nooks and crannies of objects, and keeping it fully bright on exposed areas. This only works on static (non-animated) parts of meshes, and doesn't work on areas between two meshes, of course. We can use a 3d modelling programme to "bake" these textures for us from our mesh.



Baked AO map in Blender. Note the crevices around the eyes and ears receive a darker value. You can save this baked texture to an image.

In Blender you will need to create a new material, new (blank) texture, and new Image (in the UV editor) first, or you'll get a "nothing to write to" error. Then "Properties" window → "Render" (camera) button → "Bake" submenu → "Bake Mode" → "Ambient Occlusion;"

```
...
uniform sampler2D diffuse_map, specular_map, ambient_occlusion_map;
uniform La, Ld, Ls; // light colours
...
vec3 Ka = texture (ambient_occlusion_map, st).rgb;
vec3 Is = La * Ka * Kd; // multiply by diffuse map to reduce wash-out look
...
```

Conventional wisdom says to add the ambient colour value to the Phong equation, but I find this can give a "washed-out" grey look to the scene, so I often multiply the diffuse map sample colour to the ambient colour as well. To save memory you can load and store the AO map as a grey-scale image and just sample the "r" channel. Be careful to set the correct parameters when creating the texture for a greyscale image.

For whole-scene ambient occlusion calculations other methods exist, but are generally more expensive to calculate. "Screen Space Ambient Occlusion" makes all of the calculations in run-time, rather than reading from a pre-baked texture. This can be quite expensive, but if computed in screen space (post processing) then it only has to calculate once per pixel on screen, which reduces the cost. Ambient occlusion should be a subtle effect, and makes a very small difference to the overall lighting composition (it is actually very hard to spot the difference in side-by-side images).

Common Problems

Assuming that your Phong shading is correct, then the most common problems are the same as the multi-texturing issues;

- Get the locations of all of the samplers that you use
- Remember to call `glUniform1i` to tell each sampler which texture slot to read from i.e. 0, 1, and 2.
- Before rendering, activate each active texture slot, and bind the correct texture id into each.
- Get the locations of each attribute (texture coordinates, normals, positions, etc.) from the vertex shader, and bind them to the correct location.

Fragment Rejection

We can tell a fragment that it shouldn't be drawn at all by using the `discard;` keyword in the fragment shader. This is a good way of "cutting out" areas of a surface that should be completely transparent. This reminds me of the way that sprite transparency used to be done; the parts of a sprite image that should be transparent were coloured with a "special colour" (usually bright pink - because who wants to use bright pink in a game?). Sprites were drawn, one after the other, over the screen image, ordered on how far away they were supposed to be from the viewer. If a pixel was going to be pink then it wouldn't be drawn, and whatever was there in the background would show through instead. This was amazing technology in the '90s - you could have see-through fences and walls!

My Sprite Image



I quickly drew this little guy over a pink background (red 1.0, green 0.0, blue 1.0).

I re-used the 2 triangle rectangle from my texturing tutorial, and used this as the texture. I set the `g1ClearColor` to green first, so that I could tell if the black hat was rendering properly.



The little guy on my 2-triangle rectangle.

Blit Code

I changed my fragment shader to check if the colour was very pink:

```
in vec2 texture_coordinates;
uniform sampler2D basic_texture;
out vec4 frag_colour;

void main () {
    vec4 texel = texture (basic_texture, texture_coordinates);
    if (texel.r > 0.99 && texel.g < 0.01 && texel.b > 0.99) {
        discard;
    }
    frag_colour = texel;
}
```



Fragments rejected! The blurred edges didn't get rejected because they were only partially pink. Note that `discard` also exits the shader.

And it does this. Note that when I drew it, the paint tool blurred the edges so that I get half-pink, half-white areas that are not rejected. If you were really

good at your paint tool you might know how to set the pink layer to "never blend with neighbouring pixels" - but who would know that these days? I turned down my threshold numbers to 0.7, 0.3, and 0.7 and get this:



Not perfect but better. If I managed to draw my picture without any blended pink areas then I would need either a post-processing blur to smooth the pixellated edges of the image, or use a built-in anti-aliasing function.

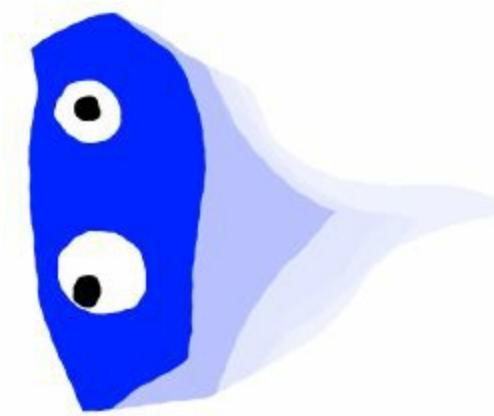
We don't need to use a special colour these days because our images can have an alpha channel. We could use this (maybe select the background colour and shrink it a bit so it gets all the blurry half pink bits). We would just test if that alpha component was over some threshold value then. Of course, you might use the `reject`; technique for something more interesting than 90s-era sprite rendering.

If I got rid of all of the blended pink pixels I would still have another problem - pixellated edges. Pixellated edges were fine in 1992, but they are going to stand out now - we would need to do some sort of post-process or anti-aliasing to smooth this sort of rough look. If we wanted to have the semi-

transparent parts, then blitting isn't the technique for us - we need to use the more involved alpha blending functions of OpenGL, which we will look at next.

Alpha Blending

We can use the alpha component in our fragment colours for a lot of things, but the most common use is to "blend" it together with the other fragments on that pixel. If you have 2 overlapping triangles, then you can make the front one seem **partially transparent** by setting its alpha value and enabling blending before drawing it. Once again, this isn't true transparency, and it has some irritating drawbacks, but it's quite easy to implement.

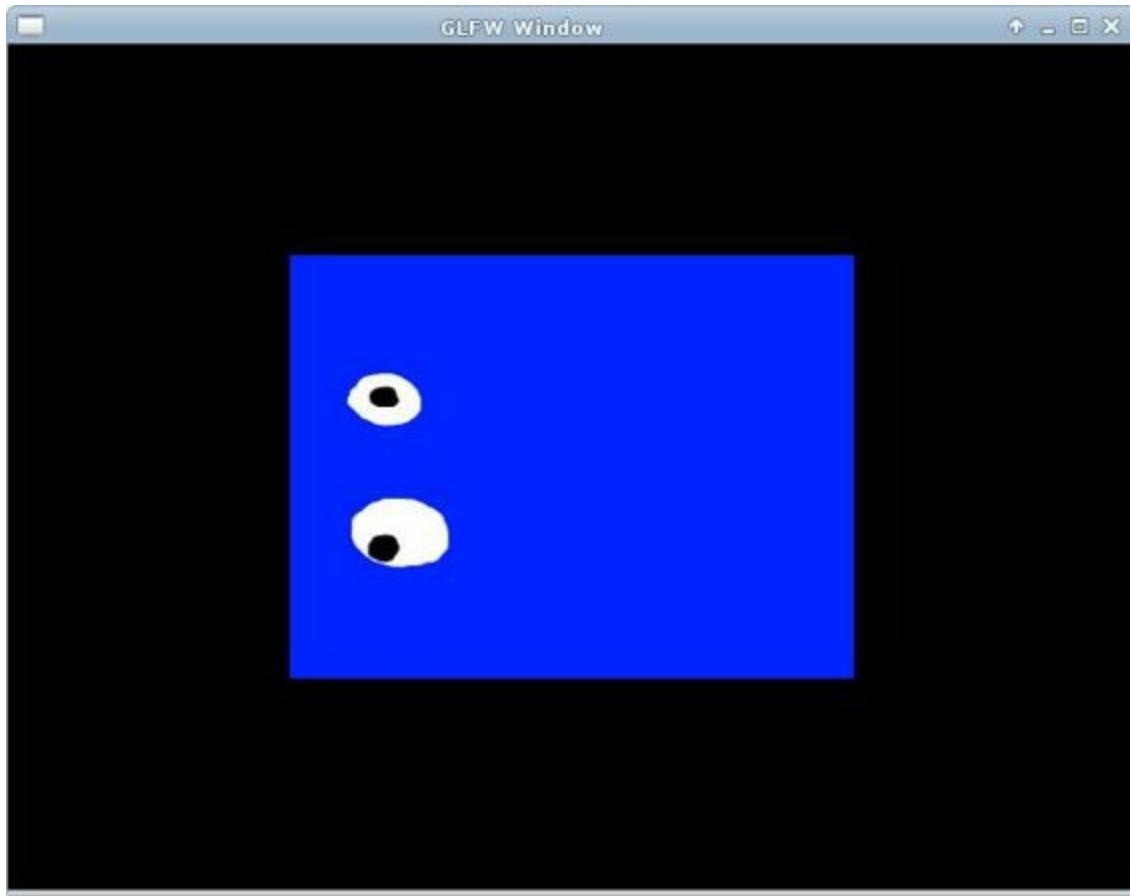


So I made a kind of blobby thing that has some totally transparent areas and some semi-transparent areas. These are stored in the alpha channel of the image - so I can map that directly to my fragment's alpha channel

Now the details - when you draw geometry you usually have 2 screen-size buffers (images) that get drawn on; the **depth buffer**, and the **colour buffer**. By default only the colour-buffer is enabled, but you may recall enabling the depth buffer with `glEnable (DEPTH_TEST);`. Normally this is very helpful, because it resolves overlaps; every time you write a fragment it checks (does the depth test function), and looks up the matching pixel in the depth buffer. If the value in the depth buffer is already smaller (closer the viewer), then it won't write the new fragment to the colour buffer. This gives us the impression of things overlapping/obscuring each other according to their distance order. If the depth test passes (i.e. the new fragment is closer than the previous one), then normally the new fragment would completely replaces the existing one in the colour buffer, and in the depth buffer.

We can hi-jack this colour-buffer writing to give the impression of transparency. Note that I said *transparent*, and not *translucent*, because it's not affecting the light calculations at all. So you can say that is a very fake way of achieving transparent effects, that will look good for some things, and not so convincing for other things such as water, where you would expect some sort of refraction to happen.

Basic Blending



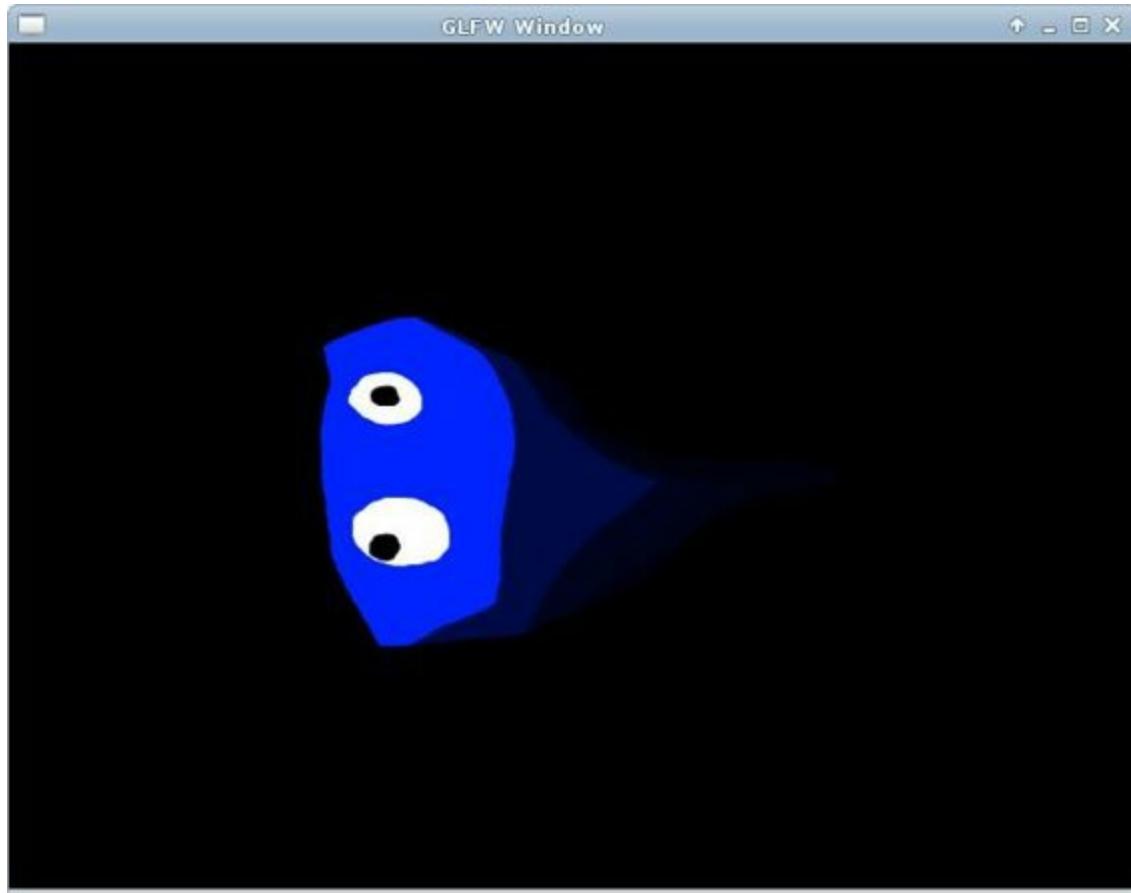
My blob drawn without blending enabled.

The way to set up blending is very simple. Before drawing, we enable blending, then we draw the object, where the alpha component of each fragment shader colour is blended with the existing one, based on some blend function that we can choose.

```
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
...
 glEnable (GL_BLEND);
 glUseProgram (my_shader_index);
 glBindVertexArray (my_vao_index);
 glDrawArrays (GL_TRIANGLES, 0, number_of_vertices);
 glDisable (GL_BLEND);
```

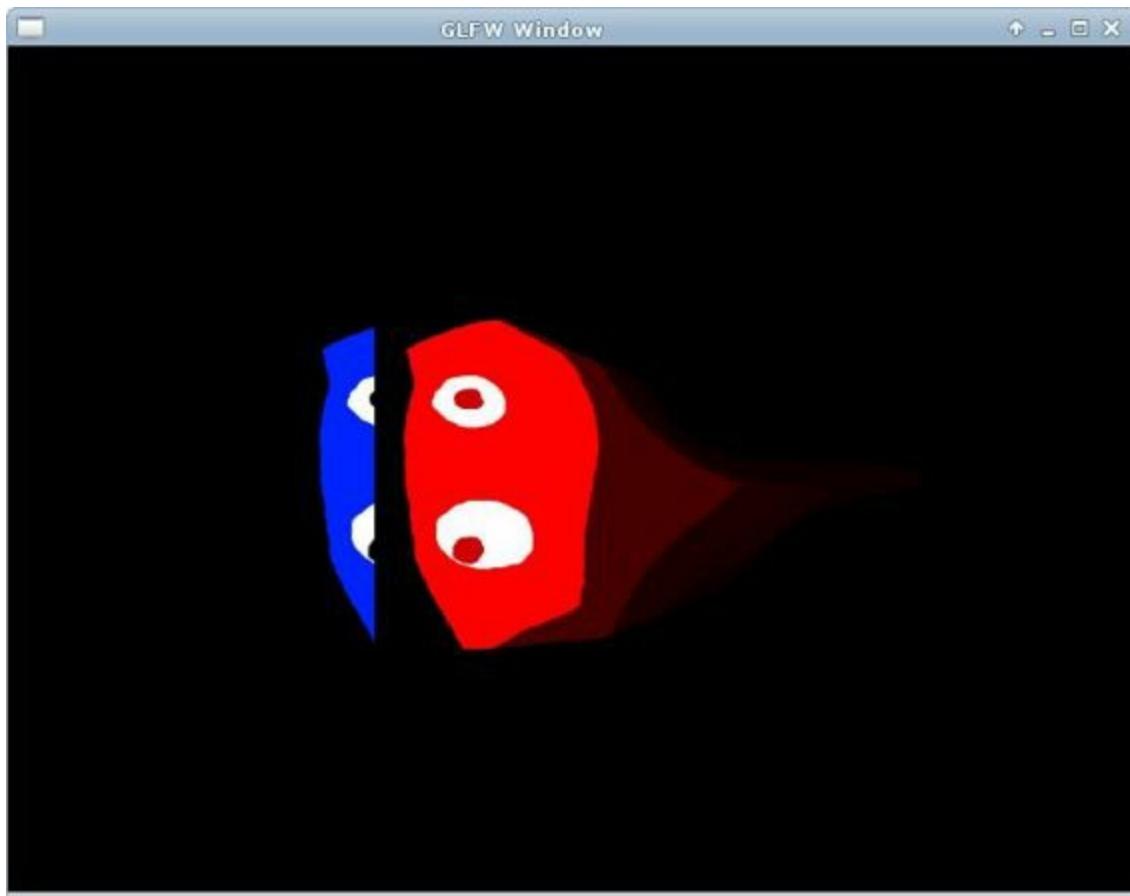
We only need to set the `blend` function once, and it stays in the state machine. The first parameter to the function asks what factor to use for blending. I

have told it `GL_SRC_ALPHA`, which means "get the factor from the alpha component in the new fragment". In other words; "use alpha blending" (you can do other types of blending that do not use the alpha channel). So if my fragment colour is `frag_colour = vec4 (1.0, 0.0, 0.0, 0.75);` (red), then my factor is 0.75. The second parameter is the equation to use. I have said `GL_ONE_MINUS_SRC_ALPHA`, which means "output pixel colour is alpha * new colour, and, $(1 - \text{alpha}) * \text{previous colour}$ ". In other words, the alpha here is actually defining the **opacity** of our fragment, so 1.0 means no transparency, and 0.0 means invisible.



My blob drawn with blending enabled.

Problems With Blending



Alpha blending does not work well with depth testing.

You might have realised that you can "stack" transparent objects on top of each other, because the next time you draw a fragment in the same pixel, the previous 2 will already have been blended, and it will just repeat the process automatically. But what if our new fragment is **behind an existing transparent pixel in the colour buffer** - whoops! The depth test will throw it away because it isn't clever enough to do sorting with transparency. So this is the down-side. We really need to disable depth testing, and sort the transparent items manually. This means working out which geometry is farther away, and drawing that before the closer geometry; the "painter's algorithm". Oh no...

```
glEnable (GL_BLEND);
glDisable (GL_DEPTH_TEST);
```

```

glUseProgram (my_shader_index);

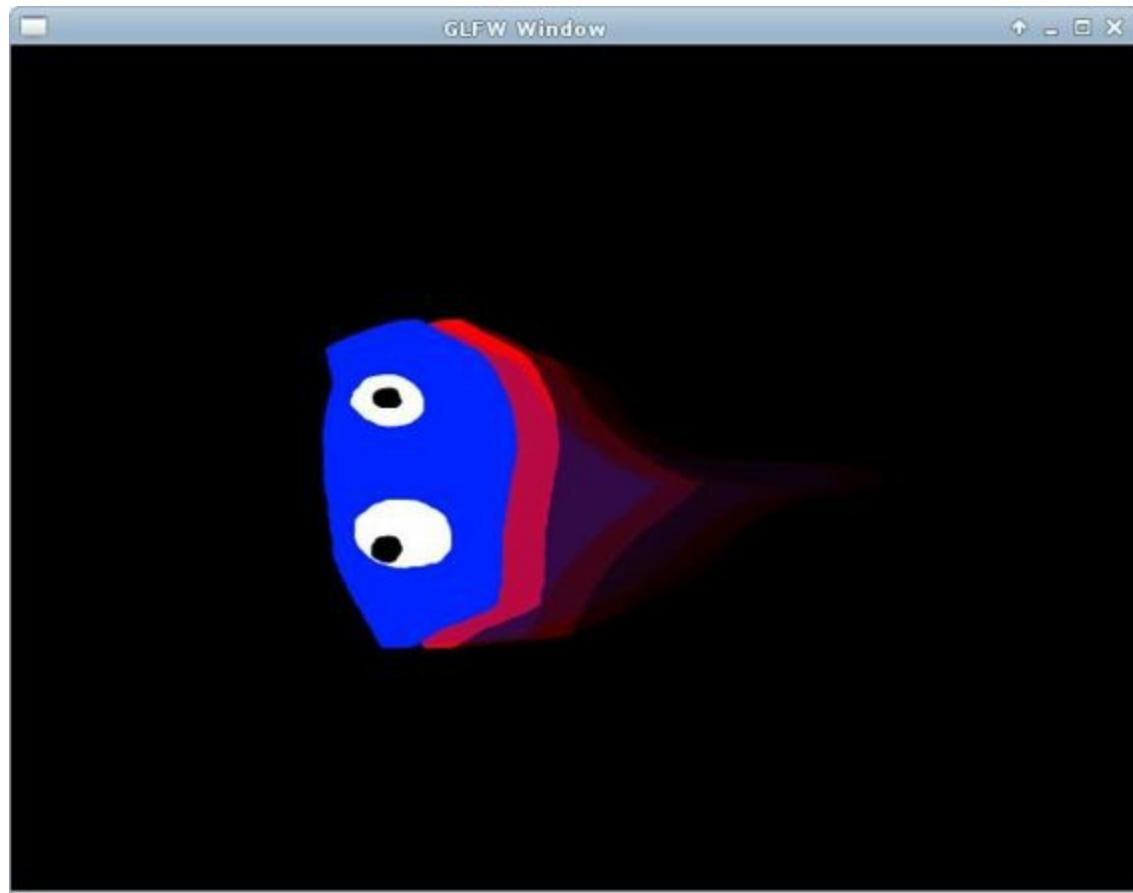
// draw an object farther away
glUniform (....
 glBindVertexArray (my_vao_index);
 glDrawArrays (GL_TRIANGLES, 0, number_of_vertices);

// draw an object closer to the viewer
glUniform (....
 glBindVertexArray (my_other_vao_index);
 glDrawArrays (GL_TRIANGLES, 0, number_of_vertices);

 glEnable (GL_DEPTH_TEST);
 glDisable (GL_BLEND);

```

Okay, that worked - all of our transparent objects blended together correctly, but we can see them, even if they are behind walls! That's perfect for GUI panels and things like that, where you actually never want them to intersect with objects in the scene, but not for objects in 3d.



Disable depth testing to draw, layered in the order of your draw calls. This is good for GUIs, but if we want to have transparent objects in a normal 3d scene (where they can be in front and behind other objects) then we can use depth masking instead.

A Cunning Plan (That Some Texts Overlook)

One sneaky trick to avoid manual depth sorting is to use `depth masking`. When depth masking is enabled it just means "write to the colour buffer, but do not write to the depth buffer". That's perfect for drawing objects where the whole object is semi-transparent. We can use depth testing for these sort of objects.

```
// prepare to draw transparent objects
glEnable (GL_BLEND);
glDepthMask (GL_FALSE);
glUseProgram (my_shader_index);

// draw an object farther away
glUniform.....
glBindVertexArray (my_vao_index);
glDrawArrays (GL_TRIANGLES, 0, number_of_vertices);

// draw an object closer to the viewer
glUniform.....
glBindVertexArray (my_other_vao_index);

glDepthMask (GL_TRUE);
glDisable (GL_BLEND);
```

We have transparent blending as well as depth testing. The down-side is that we still have to draw all the transparent objects in the correct order from back to front, and that they should be **drawn after** any objects that do not have depth masking disabled - because the blended objects do not have any depth information written and so might be drawn over.

Spotlights and Directional Lights

We have already looked at creating light sources as points in 3d space. Game engines, and older graphics APIs usually have some pre-fabricated alternatives; spotlights and directional lights are the traditional sorts.

Directional Lights

Recall in the Phong article, we calculated the direction that a point light was shining onto a surface by taking something like this:

```
vec3 light_direction_wor = normalize (vec3 (-0.5, 0.0, -1.0));
vec3 light_direction_eye = (view_matrix * vec4 (light_direction_wor, 0.0)).xyz;
vec3 direction_to_light_eye = -light_direction_eye;
```

Where I post-fixed the names with `_eye` to mean that they are all in eye coordinates. We can approximate something like the effect of the sun on a scene by just skipping this calculation and writing the direction vector by hand. Simple!

```
vec3 light_direction_eye = (view_matrix * normalize (vec3 (-0.5, 0.0, -1.0))).xyz;
vec3 direction_to_light_eye = -light_direction_eye;
```

Spotlights

Okay, for a bit of drama we can shine a light in one specific direction. It still comes from a specific point in space though. Think stage lighting, flashlights, and spinning lights on an emergency vehicle. In the real world we would just stick the light source in a cone so that it only shines one way, but remember that we are not actually calculating light rays in our GPU, so we are going to cheat again (do it in a computationally inexpensive way).

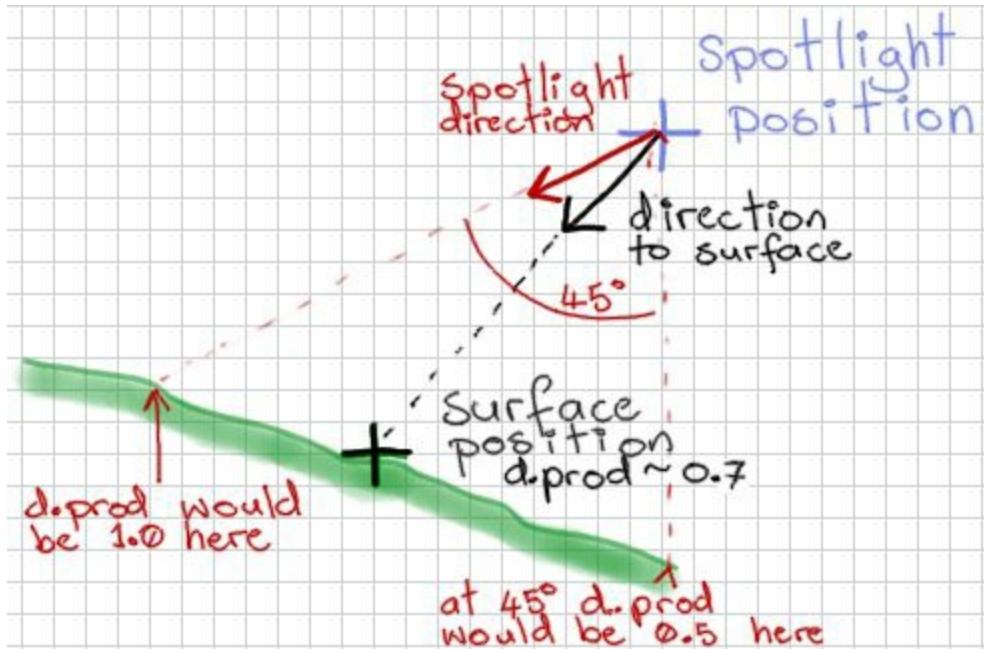
The first thing that we need to do is to define a direction for the light. We can just make up a normal for that. If it's just "point towards this other point in space" then perhaps:

```
vec3 spot_dir_eye = normalize (target_position_eye - light_position_eye);
```

Well, that was easy. Now, we still work out the direction from the point to each surface area, as we did for point lights in the Phong lighting article. This means that we have 2 direction vectors;

- Direction from light position to surface position
- Direction that the spot is shining

```
vec3 dir_to_surface_eye = normalize (surf_position_eye - light_position_eye);
```



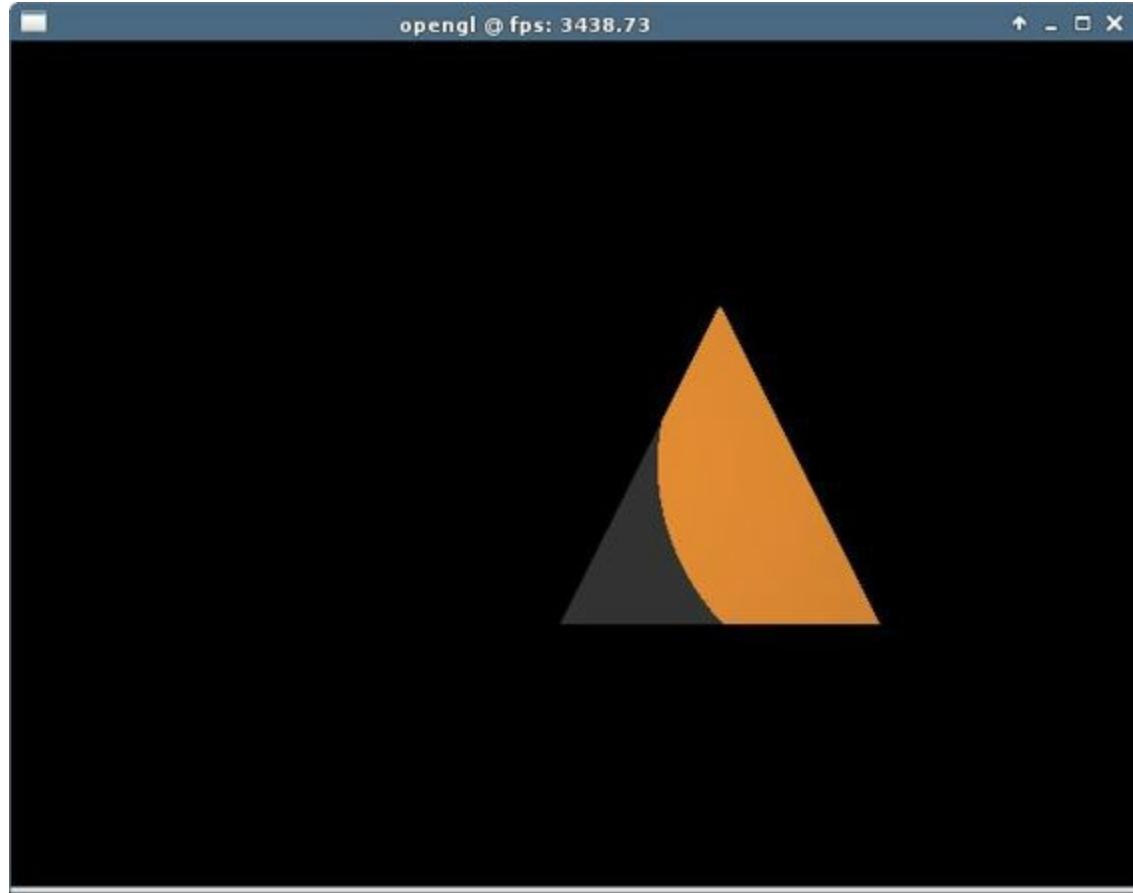
Here we work out if a surface position (vertex or fragment) is in the lit cone of a spotlight. We take the dot product of the spotlight's direction, and the direction from the light's position to the surface position. In this case we expect a value around 0.7. If our spotlight has a cone that extends 25 degrees around its direction ($1.0 - 25/90 = 0.722$) then we can see that our point should have lighting applied to it.

So, how about we get the difference between these two vectors? The dot product will give us a factor between 1.0 and 0.0; 1.0 when it's exactly the same, and 0.0 when it's 90 degrees away. That's ideal. If we want our spotlight cone to be 5 degrees, we can just make it black when the dot product is smaller than $1.0 - 5/90$, or 0.944.

```

float spot_dot = dot (spot_dir_eye, dir_to_surface_eye);
const float spot_arc = 1.0 - 5.0 / 90.0;
float spot_factor = 1.0;
if (spot_dot < spot_arc) {
    spot_factor = 0.0;
}
...
diffuse_light *= spot_factor;
speculat_light *= spot_factor;

```



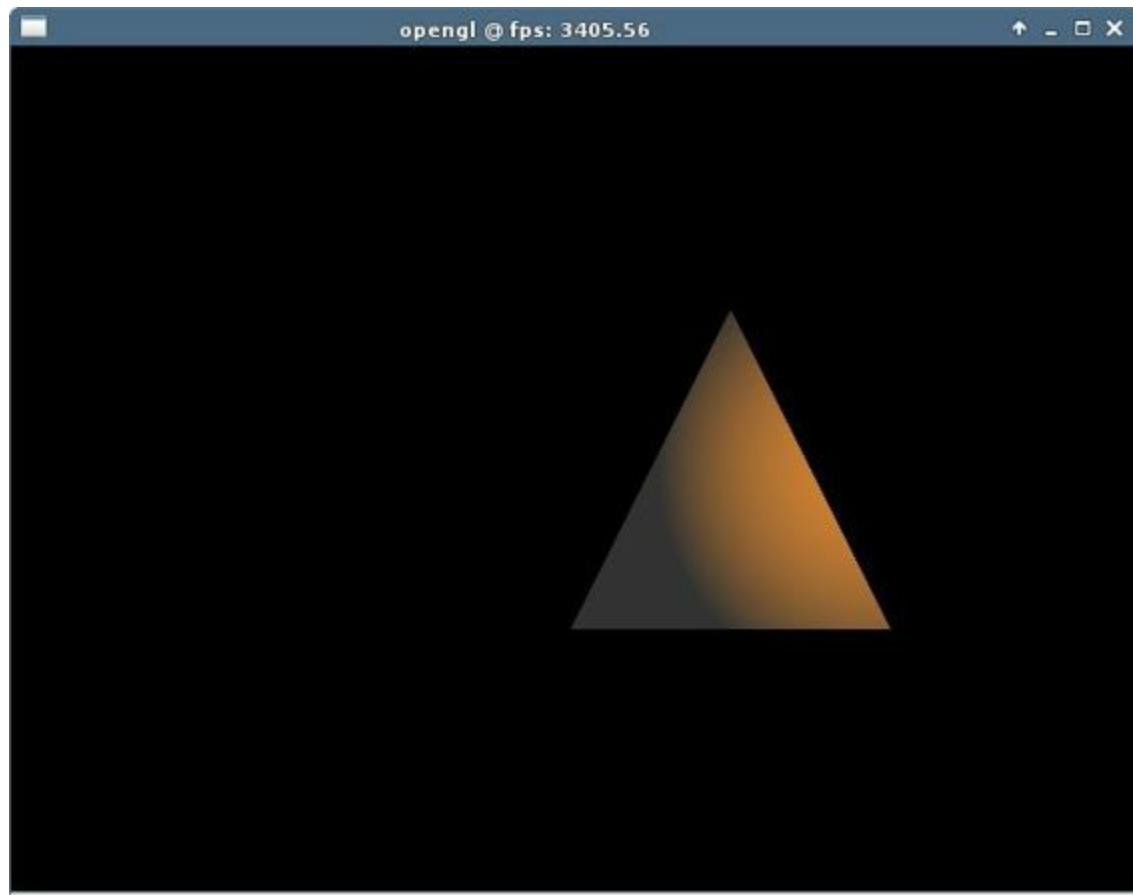
I angled my light to point slightly to the right of the screen, and gave it a radius of 5 degrees. As the triangle moves into this area the diffuse and specular lighting components receive a factor of 1. Outside the circle of light, the triangle is only lit by a grey, ambient light, which is not affected by the spotlight factor.

Attenuation

Okay, so the basic spotlight calculation will give us a uniform circle of light on a surface if we use it to factor our diffuse and specular light components. But, perhaps we want a roll-off or attenuation from centre to edge instead?

```
float spot_factor = (spot_dot - spot_arc) / (1.0 - spot_arc);  
spot_factor = clamp(spot_factor, 0.0, 1.0);
```

I just make sure that the value stays between 0 and 1 by using the `clamp()` function. You might prefer some combination of solid areas and roll-off.



A linear attenuation to give the spot a roll-off of intensity. For more interesting roll-off functions you might consider an exponential or a logarithm, but often it's hard to perceive non-linear change, so don't be too fussy.

We can, of course, use this sort of attenuation for distance from the light source too, which might be useful also for point lights. I used this idea in my

game so that the torch that the player is carrying only lights up a limited radius around the character.

Distance Fog

Fog is a very cheap and easy effect to do. Fog has a number of uses in 3d graphics:

- adding atmosphere to a scene as fog, smog, mist, steam, gas, or darkness
- enhancing the user's perception of distance
- fading scenery gradually away before the far clip plane removes it more noticeably

Linear Fog

The easiest equation to implement is simply a linear blend that has a fog colour. Usually, fog should get "thicker" on things that are farther away from the camera. It doesn't have to be the camera though; any point of interest will do. A linear fog equation will have:

1. A `vec3` fog colour
2. a minimum (starting) distance from the camera - before this point no fog colour is seen
3. a maximum distance from the camera - after this point all fragments are rendered fully as fog colour

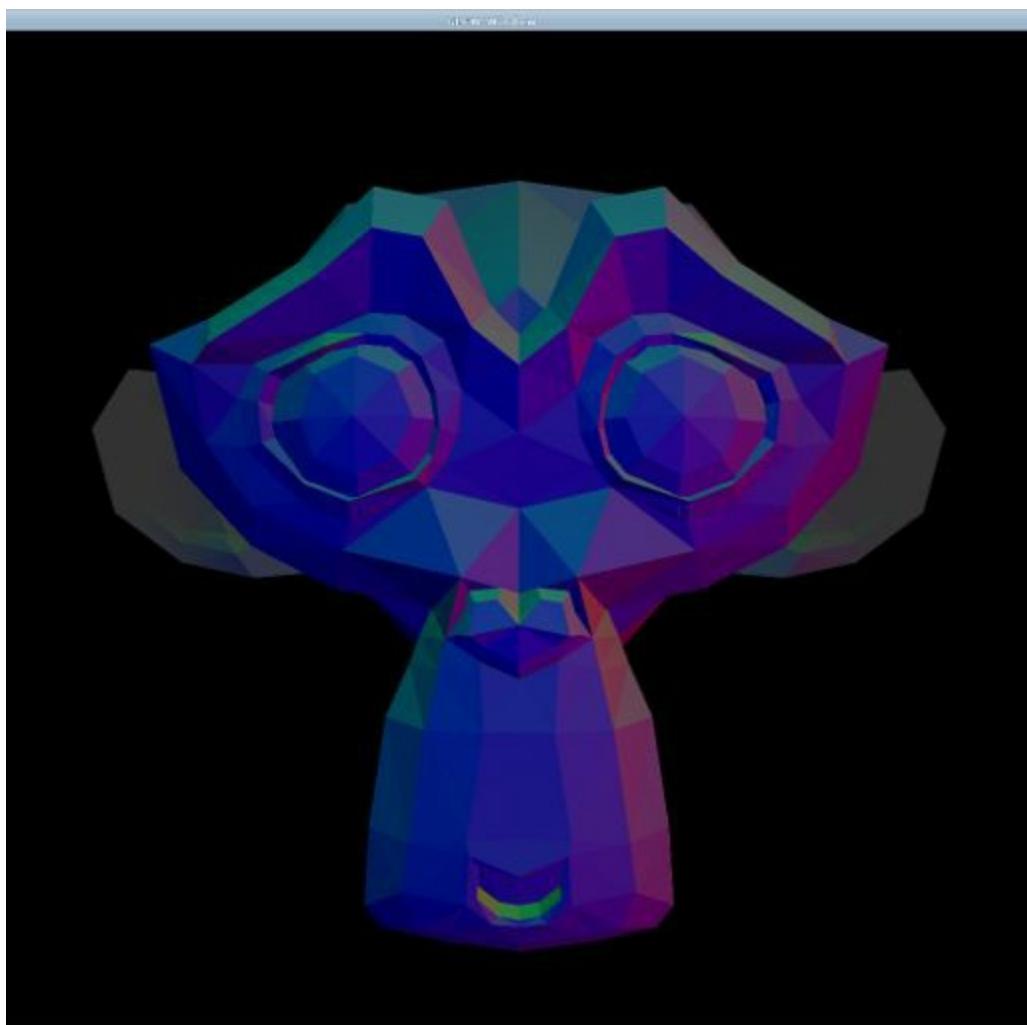
Then we just put an extra equation into our fragment shader after any lighting calculation. Remember, to get a 3d distance we subtract one `vec3` from the other i.e. `vec3 distance = cam_pos - surface_pos;`. If we do this calculation in eye space, then we have the camera position at (0, 0, 0), so our distance is just `-surface_pos_eye;`. Now, to get the distance as a 1d float, we would normally do Pythagoras' theorem; `float d = sqrt (x * x + y * y + z * z)`, where the x, y, z are the components of our 3d distance. GLSL has a built-in function to do this called `length()`. I would do this in a fragment shader, to get nicer-looking fog:

```
// required fog variables
const vec3 fog_colour = vec3 (0.2, 0.2, 0.2);
const float min_fog_radius = 3.0;
const float max_fog_radius = 10.0;

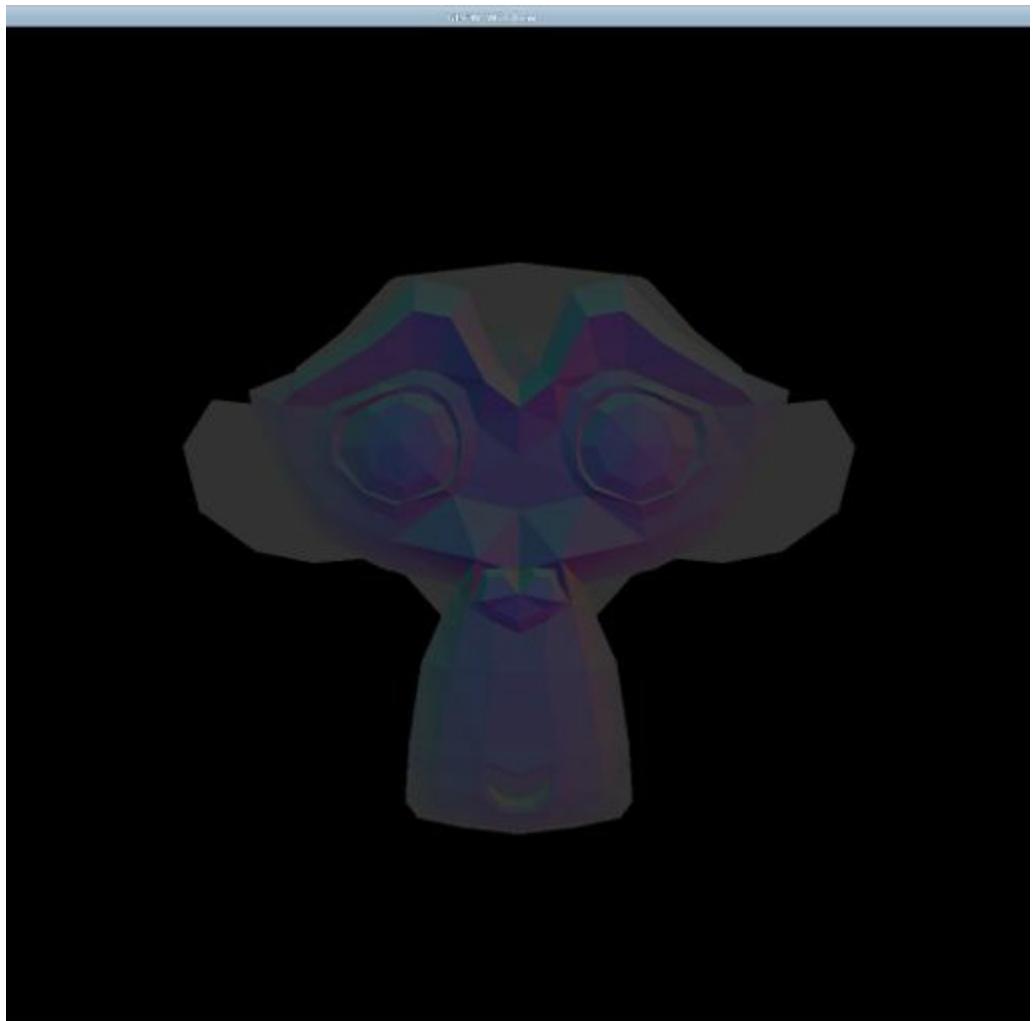
// work out distance from camera to point
float dist = length (-pos_eye);
// get a fog factor (thickness of fog) based on the distance
float fog_fac = (dist - min_fog_radius) / (max_fog_radius - min_fog_radius);
// constrain the fog factor between 0 and 1
fog_fac = clamp (fog_fac, 0.0, 1.0);

// blend the fog colour with the lighting colour, based on the fog factor
frag_colour.rgb = mix (phong_colour, fog_colour, fog_fac);
```

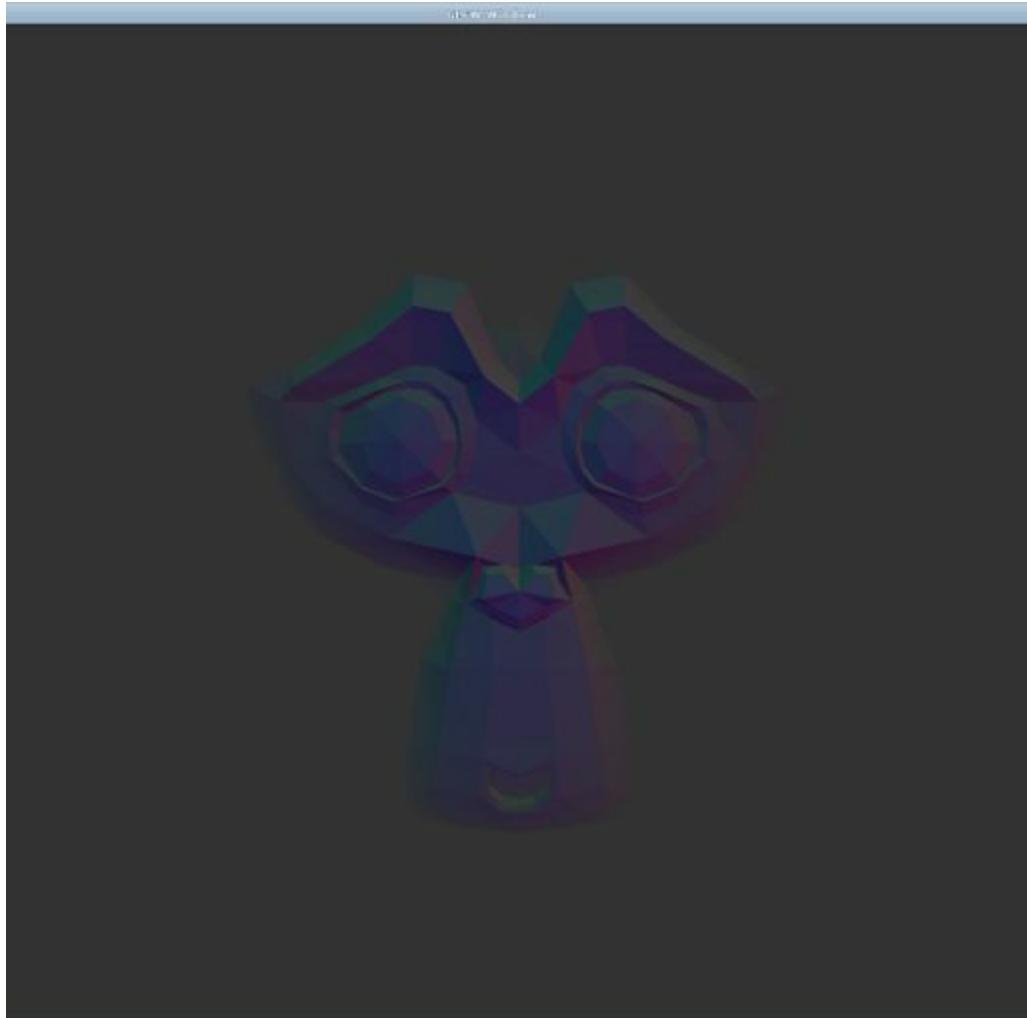
Where I assumed that the final Phong lighting colour is stored in a variable called `phong_colour`. The `mix()` function means that the fragment colour will be completely coloured with fog when the fog factor is 1, and completely based on phong lighting when the fog factor is 0.



Most of the mesh is in front of the "minimum fog distance" here so won't be fogged.



The mesh here is about half-way into the fog range. The fog colour is grey, which looks a bit strange with a black background.



If your fog is used to hide the far clip plane and cull distant scenery then we can do an old trick and set the background colour the same as the fog colour to make it look like things are disappearing into the fog.

Non-Linear Fog

Our brains are hopeless at perceiving rates of change that are non-linear, so most of the time using an exponential or a log function isn't even going to be noticed, so we won't bother. However, for very large scenes, it can be really interesting to use fog to visualise distance (think mountains fading into a blue colour). We are terrible at judging distance in 3d graphics, but adding subtle queues like colour fade can greatly improve this. If you have a really big space then a linear fog fade is going to be very obvious - you might like to manipulate this spatial awareness one way or the other by using an exponential or a logarithmic function instead.

Normal Mapping

We looked at specular mapping already, where we sampled the specular lighting reflection coefficient from a texture. This let us add a lot more detail to the surface than just interpolating per-vertex attributes to each fragment. We can do something similar with the normals; we sample a texture containing normals to **perturb** (push over a bit) the existing, interpolated normals that we usually use for lighting. This gives the appearance of a much bumpier surface. This is the most common form of **bump mapping** technique. There are some complicated problems though:

1. Creating the normal map image requires some technical artistry.
2. The sampled normals are in a coordinate space relative to the texture (imagine if the texture is rotated around), so we need to do some transformations to get them into the same coordinate space as the lighting calculations.
3. The new transformations require some new per-vertex attributes to help with the transformations. If our mesh format does not include these, then we will need to calculate them ourselves.

Bump mapping is another technique introduced by James F. Blinn in 1978 SIGGRAPH, with "Simulation of Wrinkled Surfaces". It is important to note this technique does not actually change the geometry of a mesh, it just changes the normal used in the lighting equation, so that the interior surfaces look like they have bumps. The silhouette of the mesh will not change shape.

If you're looking for a good mathematical explanation of normal mapping with triangulated meshes, then let me suggest Eric Lengyel's Mathematics for 3D Game Programming and Computer Graphics. As far as I am aware, it's the only good reference around for actually generating tangents for a mesh. Some of the other texts tend to avoid this subject and concentrate only on the shaders (the easy part), which is a bit suspicious! I was originally going to go down this same route, because I spent a great deal of time manually implementing a normal-mapping system and tangent generator in GL. It now occurs to me that I haven't seen a good, quick, description of how to use a set

of tools to get all the required bits together with minimal fuss. I went that way with the mesh loading tutorial, so I'll just extend that code for this tutorial - **keep it simple, stupid!**

Making a Normal Map Image



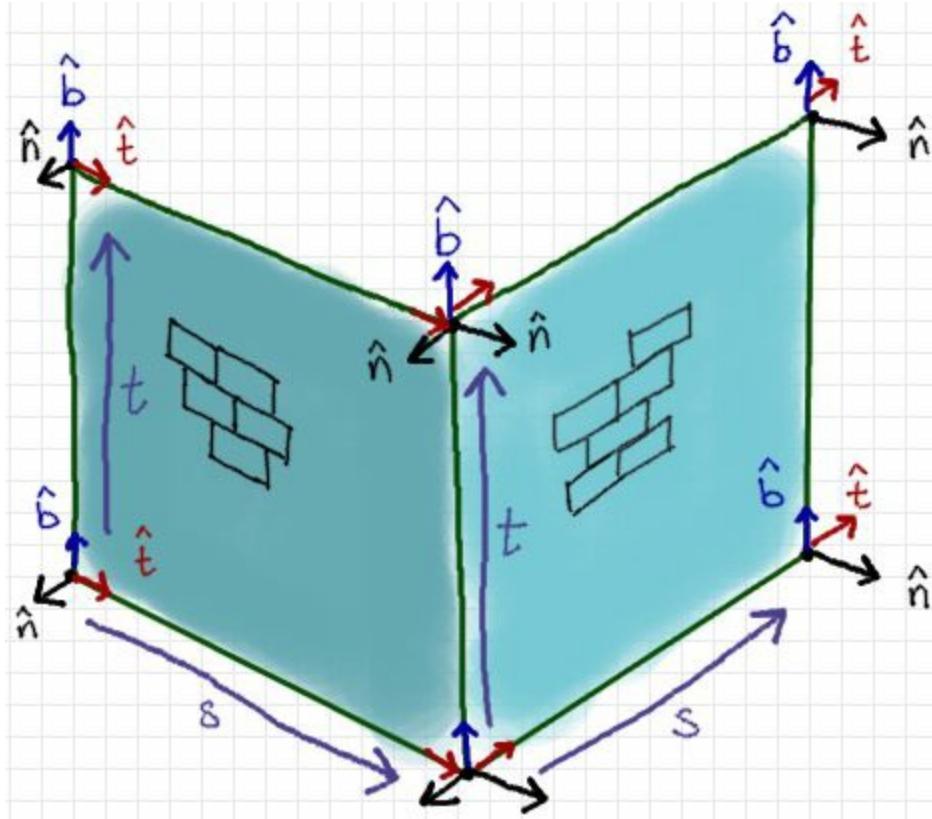
Top row are the diffuse, and specular textures for a door mesh. The middle row shows creation of a

height map (middle-left). This can be manually edited from a greyscale version of the diffuse map image, or exported from a more detailed version of the door mesh. The raised sections are lighter colours. I used a plug-in for the GIMP tool to generate a normal map from the height map (middle-right). Areas of the same colour in the height map will point forward (+z), which corresponds to a purple colour (0.5,0.5,1.0) in the normal map. Where the height map changes from grey to black it should generate normals pointing either to the right (shades of red), or to the top (shades of green). The bottom rows shows the combined diffuse, specular, and normal maps in GIMP (bottom-left), and finally in OpenGL (bottom-right).

The easiest way to create a normal map image is to use an image editing tool. Nvidia has a plug-in for Photoshop to generate a normal map from a height-map, and there is a copy of this for GIMP, which you can download (<https://code.google.com/p/gimp-normalmap/>). The trick then, is to create the height-map. We will be using the same texture coordinates as the regular textures, so I just converted one of my textures to greyscale and drew over it with a grey paintbrush to flatten-out the surfaces (see above image). You can see that this might take a bit of trial-and-error to get it looking good.

The normal map creation procedure looks for differences in height to work out which way the normals should point. These are encoded as RGB colours with a range of 0 to 1. Our normals will need to be -1 to 1, so we will remember to modify this when we sample it, later. It's also possible to write a little function to generate the normals from just a height-map, of course.

Generating Per-Vertex Tangents



Two sections of a wall that use the same normal map. Both normal maps will produce the same set of normals, but we need to be able to say that one section of wall is facing in another direction. We know which way a sampled normal with value $(0,0,1)$ is facing in local space; the exact same way as the per-vertex normal which we interpolated from the vertex shader, $n\hat{}$. But to help us work out the X and Z components we will add in a tangent, $t\hat{}$, and a bi-tangent, $b\hat{}$, which will line up with the directions of the s , and t , texture coordinate axes, respectively.

Imagine this: we have 2 sections of a wall, perpendicular to each other (as in the image, above). We sample the diffuse texture for each one using the texture coordinates s , and t . We do the same thing to sample the normal map, which gives us an XYZ normal for each fragment. But wait; the sampled normals for both parts of the wall are the same, even though the faces point different ways! We know that the Z component should point the same way as the normal that we got from the vertex shader, but we can't tell which way the X and Z components should be pointing because we don't have the texture coordinate axes any longer; just the current point. You can create the tangent, and bi-tangent manually, but we are using AssImp, and we can just ask

AssImp to do this when we load the mesh:

```
aiImportFile (file_name, aiProcess_CalcTangentSpace);
```

The only *caveat* is that the mesh must have normals in it, or AssImp will quietly ignore the tangent-generation process. When successful, this will give us a tangent, and a bi-tangent per-vertex, that we can pull out of AssImp.

We can create two more direction vectors to represent the S and T directions; we call these the **tangent**, and **bi-tangent** vectors. With the set of 3 vectors we have the axes for defining the **tangent space** of the sampled normals. This means that we can convert between tangent space, and local coordinate space (and therefore also world space and eye space in the same way that we transform vertex points). We can build a tangent matrix, T, to convert from tangent space to local space, and its inverse, for converting from local space to tangent space.

$$\begin{bmatrix} \hat{t}_x & \hat{b}_x & \hat{n}_x \\ \hat{t}_y & \hat{b}_y & \hat{n}_y \\ \hat{t}_z & \hat{b}_z & \hat{n}_z \end{bmatrix} = T$$

The Tangent Matrix

If we build a tangent matrix from our 3 per-vertex vectors (normal, tangent, and bi-tangent), then we can convert from tangent-space to local space. The problem is that our vertex normal is not necessarily perpendicular to the other two vectors, so we will orthogonalise the tangent and bi-tangent first.

In the case of the walls pictured, our matrices are fine, but for other shapes we can't be sure that our normal is exactly perpendicular to the bi-tangent, and tangent. Why? Imagine a sphere. It's made of triangles, but we want smooth lighting, so the normals at each vertex are averaged from the surrounding faces. This gives us a more spherical look when we light it. It also means that the normals are not perpendicular to the texture coordinate axes. This means that our normal-bi-tangent-tangent axes are not orthonormal. We can fix this by doing a little bit of jiggery-pokery using the

Gram-Schmidt algorithm, which will change our tangent and bi-tangent vectors a little. We can then just transpose the tangent matrix to get something usable for the inverse. The tangent is orthogonalised, and normalised, like this:

```
vec3 t_i = normalise (t - n * dot (n, t)); (make tangent perpendicular to normal)
```

$$\begin{bmatrix} \hat{t}_x & \hat{t}_y & \hat{t}_z \\ b_x & b_y & b_z \\ \hat{n}_x & \hat{n}_y & \hat{n}_z \end{bmatrix} = T^{-1}$$

Inverse Tangent Matrix

Once we are sure that our 3 vectors are orthogonal, then we can just transpose the Tangent matrix to create its inverse, which lets us transform from local space to tangent space. This is going to be more useful, as we will see shortly.

Now, you might see that this means that we are going to have a very large number of matrices. We don't want to have to send in lots of matrix uniforms, so we are just going to re-assemble the matrix inside the vertex shader. To do this we only need the normal, which we already have, and the tangent. Because we will orthogonalise our tangent first, this means that we can generate the bi-tangent inside the shader; from the cross-product of the tangent and the normal. That means that we only need to add in one more per-vertex attribute; the tangent.

When we do this cross-product, it's possible that we do it backwards to the expected order, which will make our bi-tangents point the wrong way. If you notice this (the faces will not light up consistently) then we just need to negate the bi-tangent, ($*= -1.0f$). You can detect and fix this automatically by working out the determinant of the tangent matrix, and adding this in as another attribute. This should be either -1 or 1, so if we multiply the sampled normal by this value then it is taken care of. I'll add in the code here, but you can most likely leave it out. The common tactic is to store this determinant as

the fourth (w) component of the tangent vector.

The following code is extending the AssImp vertex attribute extraction from the mesh tutorial, so if you've already got that running, then you can just fill out the tangents `if`-statement, that I left as a place-holder in that tutorial. I use my own maths library here because it has dot-product, and cross-product functions, but you can use any maths library here.

```
// a pointer to a dynamic array of floats containing the xyzw tangents
GLfloat* g_vtans = NULL;

...

bool load_mesh (const char* file_name) {
    /* ask AssImp to generate tangents after loading the mesh */
    const aiScene* scene = aiImportFile (
        file_name, aiProcess_Triangulate | aiProcess_CalcTangentSpace
    );

    ...

    if (mesh->HasTangentsAndBitangents ()) {
        printf ("mesh has tangents\n");
        g_vtans = (GLfloat*)malloc (g_point_count * 4 * sizeof (GLfloat));
    }

    /* the loop where we copy points, texture coords, etc. into memory */
    for (unsigned int v_i = 0; v_i < mesh->mNumVertices; v_i++) {

        ...

        if (mesh->HasTangentsAndBitangents ()) {
            const aiVector3D* tangent = &(mesh->mTangents[v_i]);
            const aiVector3D* bitangent = &(mesh->mBitangents[v_i]);
            const aiVector3D* normal = &(mesh->mNormals[v_i]);

            /* put the three vectors into my vec3 struct format for doing maths */
            vec3 t (tangent->x, tangent->y, tangent->z);
            vec3 n (normal->x, normal->y, normal->z);
            vec3 b (bitangent->x, bitangent->y, bitangent->z);

            /* orthogonalise and normalise the tangent so we can use it in something
               approximating a T,N,B inverse matrix */
            vec3 t_i = normalise (t - n * dot (n, t));

            /* get determinant of T,B,N 3x3 matrix by dot*cross method */
            float det = (dot (cross (n, t), b));
            if (det < 0.0f) {
                det = -1.0f;
            } else {
                det = 1.0f;
            }

            /* push back 4d vector for inverse tangent with determinant */
            g_vtans[v_i * 4] = t_i.v[0];
            g_vtans[v_i * 4 + 1] = t_i.v[1];
            g_vtans[v_i * 4 + 2] = t_i.v[2];
        }
    }
}
```

```

        g_vtans[v_i * 4 + 3] = det;
    } // endif
} // endfor
}

```

If you like, you can build a `mat3` using the 3 vectors, and get the determinant from that, but I short-cut that by just normalising an orthogonal `t_i`, and doing a dot * cross method for the determinant. After this extraction loop is over, we can of course create a vertex buffer and set up attribute pointers as we do for everything else:

```

GLuint tangents_vbo;
if (NULL != g_vtans) {
    glGenBuffers (1, &tangents_vbo);
    glBindBuffer (GL_ARRAY_BUFFER, tangents_vbo);
    glBufferData (
        GL_ARRAY_BUFFER,
        4 * g_point_count * sizeof (GLfloat),
        g_vtans,
        GL_STATIC_DRAW
    );
    glVertexAttribPointer (3, 4, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray (3);
}

```

Shaders

The general approach is this:

1. Work out the light direction (`mesh_world_pos - light_world_pos`), and convert this to **local space** by multiplying it with the **inverse model matrix**. This is the same as the light direction vector that we use for Phong, except that we convert it to local space first.
2. Work out the camera position in local space in the same way.
3. In the vertex shader, work out the bi-tangent by taking the cross product of the vertex normal and the tangent. Multiply this by the "handedness" value that we stored in the w component of the tangent vector. This just negates the bi-tangents if they are all pointing backwards. Normalise each vector before using it.
4. Form a **tangent matrix** from the tangent vector, bitangent vector, and vertex normal.
5. Multiply the view and light directions by the tangent matrix to transform them from local space to tangent space. Output these to be interpolated to the fragment shaders.
6. In the fragment shader, sample the normal map, and convert the value into the range of -1:1, and normalise it. We will use this normal, which is in tangent space, for the lighting calculations, instead of the one that we'd normally use from the per-vertex normals.
7. We do Phong lighting as usual, except everything will be done in tangent space. We will use the sampled normal, from the texture, instead of an interpolated normal. We will use the view direction, and light directions, which we get from the vertex shader (remember, these are in tangent space now too). We should normalise them before we use them though.

This means that you will need to add a couple of uniform values for the light direction, and camera position, and send these to the shader in the normal way.

Vertex Shader

```

#version 400

/* attributes are vertex positions, vertex normals, texture coordinates, and
   per-vertex tangents
*/
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;
layout(location = 2) in vec2 texture_coord;
layout(location = 3) in vec4 vtangent;

/* uniforms for the camera position and light direction. you could also use the
   light position, and work out the direction, as we did in the Phong tutorial.
*/
uniform mat4 model, view, proj;

out vec2 st;
out vec3 view_dir_tan;
out vec3 light_dir_tan;

void main() {
    gl_Position = proj * view * vec4 (vertex_position, 1.0);
    st = texture_coord;

    /* I hacked my camera position out of the view matrix. to do this properly you
       might use a uniform vec3 instead */
    vec3 cam_pos_wor = (inverse (view) * vec4 (0.0, 0.0, 0.0, 1.0)).xyz;
    vec3 light_dir_wor = vec3 (0.0, 0.0, -1.0);

    /* work out bi-tangent as cross product of normal and tangent. also multiply
       by the determinant, which we stored in .w to correct handedness
    */
    vec3 bitangent = cross (vertex_normal, vtangent.xyz) * vtangent.w;

    /* transform our camera and light uniforms into local space */
    vec3 cam_pos_loc = vec3 (inverse (model) * vec4 (cam_pos_wor, 1.0));
    vec3 light_dir_loc = vec3 (inverse (model) * vec4 (light_dir_wor, 0.0));
    // ...and work out view_direction_ in local space
    vec3 view_dir_loc = normalize (cam_pos_loc - vertex_position);

    /* this [dot, dot, dot] is the same as making a 3x3 inverse tangent matrix, and
       doing a matrix*vector multiplication.
    */
    // work out view direction in _tangent space_
    view_dir_tan = vec3 (
        dot (vtangent.xyz, view_dir_loc),
        dot (bitangent, view_dir_loc),
        dot (vertex_normal, view_dir_loc)
    );
    // work out light direction in _tangent space_
    light_dir_tan = vec3 (
        dot (vtangent.xyz, light_dir_loc),
        dot (bitangent, light_dir_loc),
        dot (vertex_normal, light_dir_loc)
    );
}

```

The job of the vertex shader is to re-construct the bi-tangent, by doing a cross-product. You can multiply this by the determinant here to ensure that the result of the cross-product is not pointing backwards to its correct

direction. Remember that we stored the determinant in the w component of the tangent vector. Once we have this, we can convert the view and light directions into local space by multiplying them with the inverse model matrix, and then into tangent space, by multiplying them with the inverse tangent matrix. We could build a 3X3 tangent matrix, but it's quicker to just do 3 dot products; a matrix*vector operation is mathematically the same as doing a dot product for each component (try it on paper to double-check). Once we have everything in tangent space, we output these variables to the fragment shaders, and we will be able to compute the whole lighting equations in tangent space.

Fragment Shader

```
#version 400

// inputs: texture coordinates, and view and light directions in tangent space
in vec2 st;
in vec3 view_dir_tan;
in vec3 light_dir_tan;

// the normal map texture
uniform sampler2D normal_map;

// output colour
out vec4 frag_colour;

void main() {
    vec3 Ia = vec3 (0.2, 0.2, 0.2);

    // sample the normal map and convert from 0:1 range to -1:1 range
    vec3 normal_tan = texture (normal_map, st).rgb;
    normal_tan = normalize (normal_tan * 2.0 - 1.0);

    // diffuse light equation done in tangent space
    vec3 direction_to_light_tan = normalize (-light_dir_tan);
    float dot_prod = dot (direction_to_light_tan, normal_tan);
    dot_prod = max (dot_prod, 0.0);
    vec3 Id = vec3 (0.7, 0.7, 0.7) * vec3 (1.0, 0.5, 0.0) * dot_prod;

    // specular light equation done in tangent space
    vec3 reflection_tan = reflect (normalize (light_dir_tan), normal_tan);
    float dot_prod_specular = dot (reflection_tan, normalize (view_dir_tan));
    dot_prod_specular = max (dot_prod_specular, 0.0);
    float specular_factor = pow (dot_prod_specular, 100.0);
    vec3 Is = vec3 (1.0, 1.0, 1.0) * vec3 (0.5, 0.5, 0.5) * specular_factor;

    // phong light output
    frag_colour.rgb = Is + Id + Ia;
    frag_colour.a = 1.0;
}
```

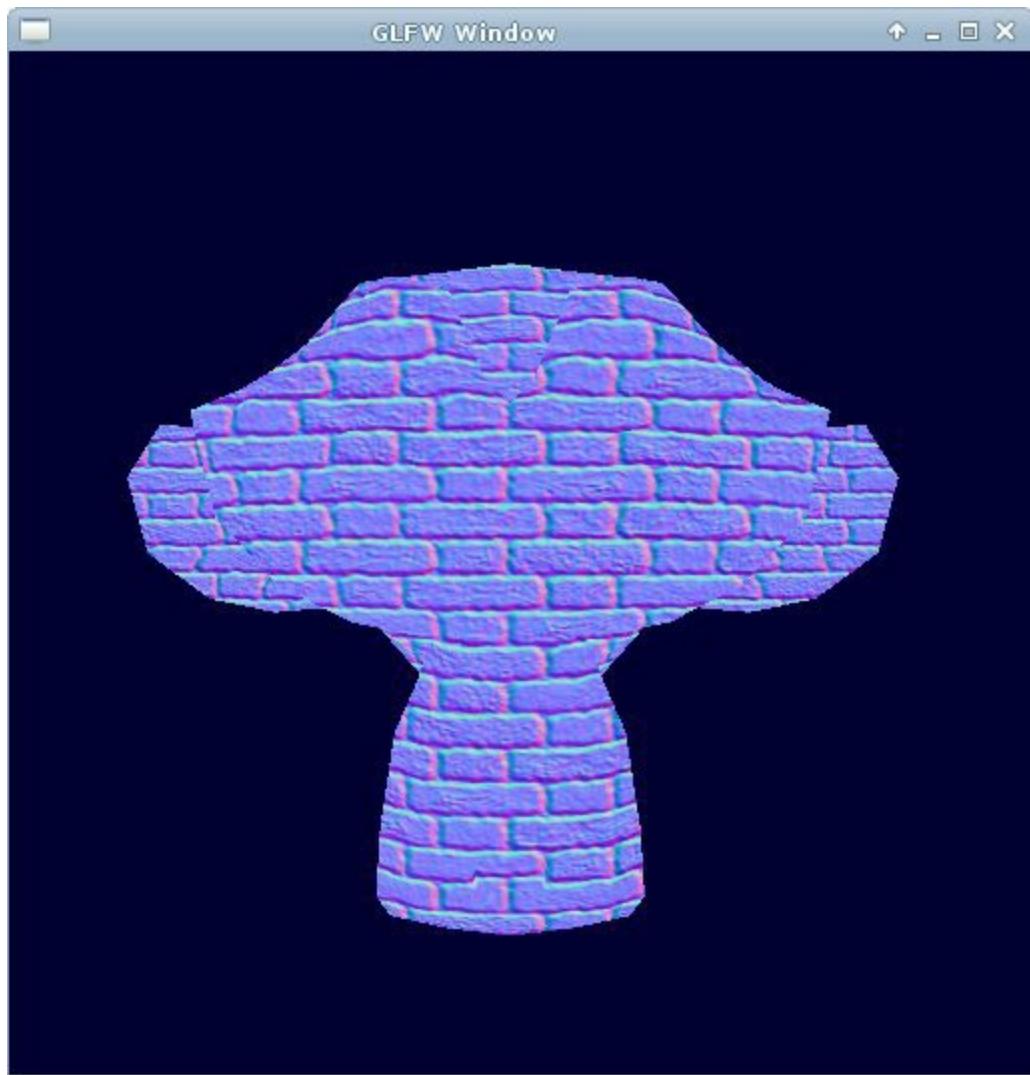
The fragment shader is the same as the Phong fragment shader, except that we use the sampled normal instead of the normals from the vertex shader, and all the other variables are in tangent space, to match. You could instead convert your sampled normals to eye space or something, but it's usually a bit quicker to put as much computation as possible into the vertex shaders.

Loading the Normal Map

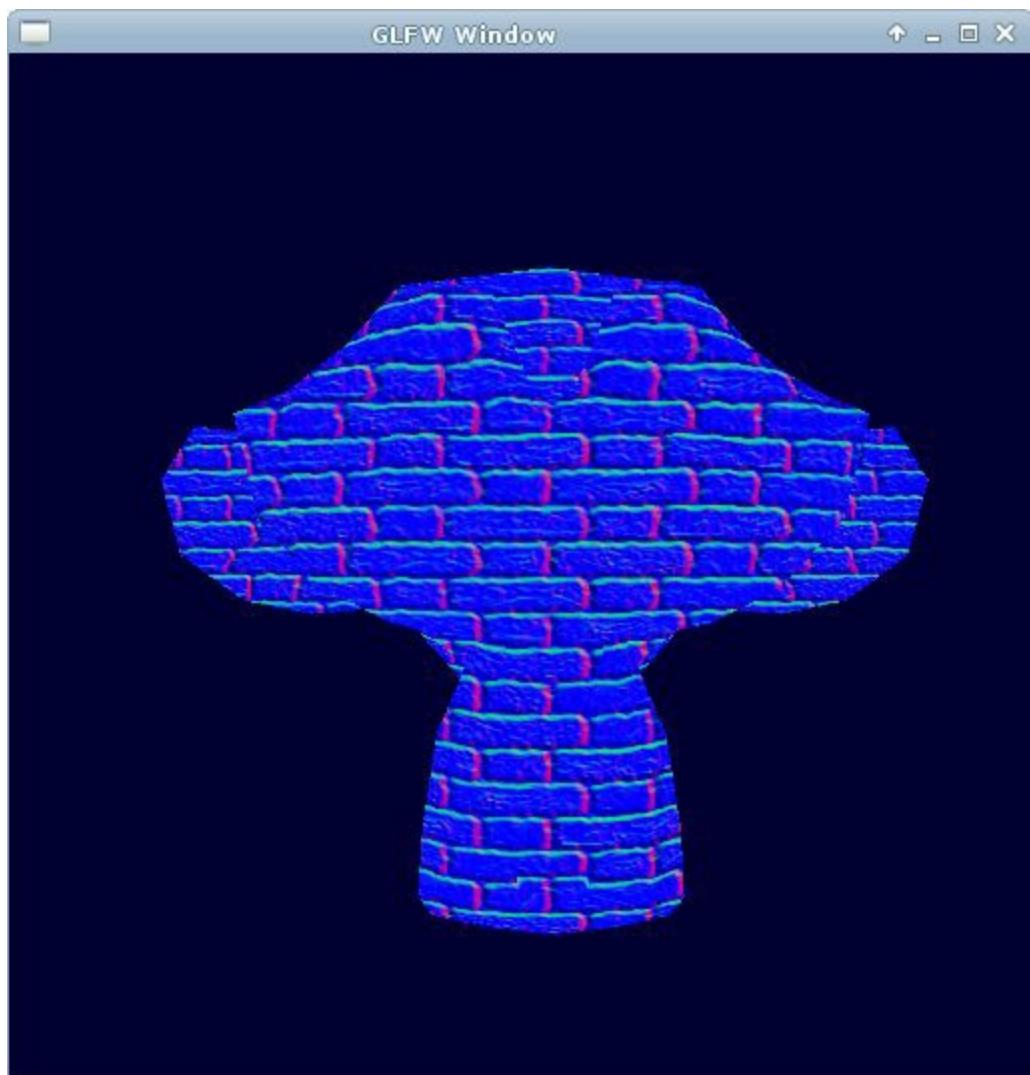
Load the normal map as if it were a normal texture. I used the same function that we looked at in the texturing tutorial. If you are using diffuse map and specular maps as well, then you will need to bind the normal map into a new texture slot, and also set the uniform value for the sampler to match this slot number (as we did in the multi-texturing tutorial).

Testing

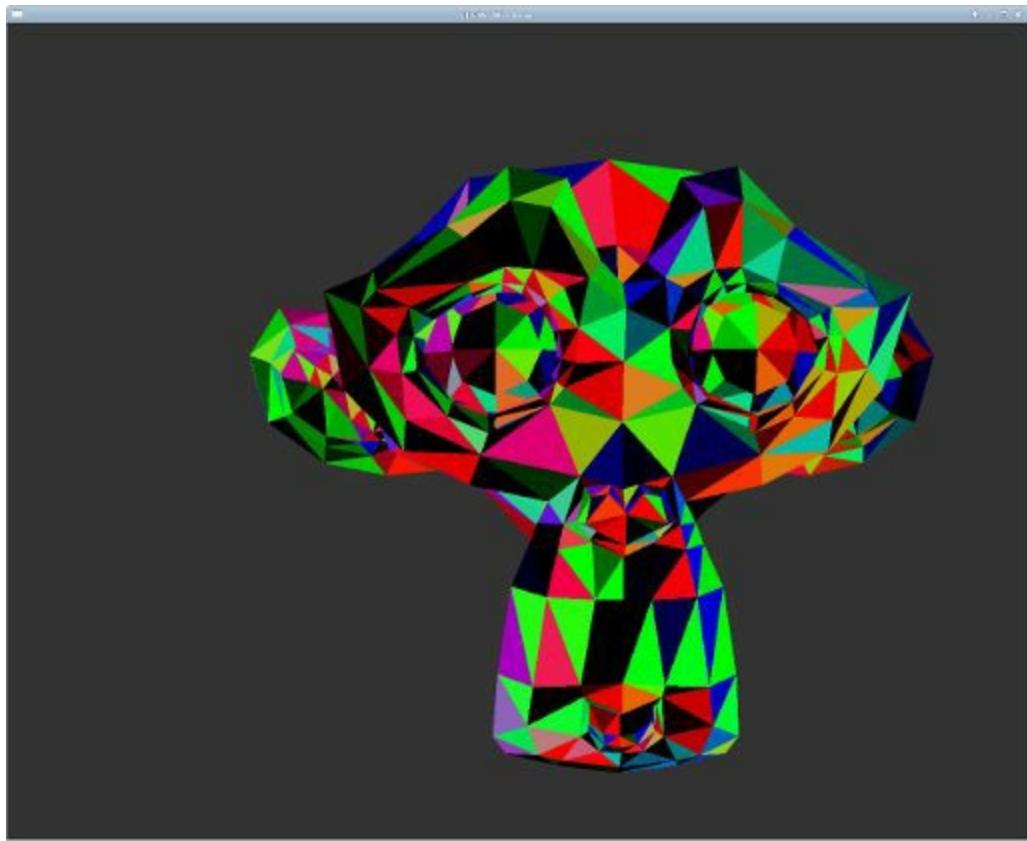
There's quite a lot of code in the shaders, so make sure that you build up slowly (add one bit at a time), and test each variable. I output each of my new variables as a colour in the fragment shader. They should look something like this:



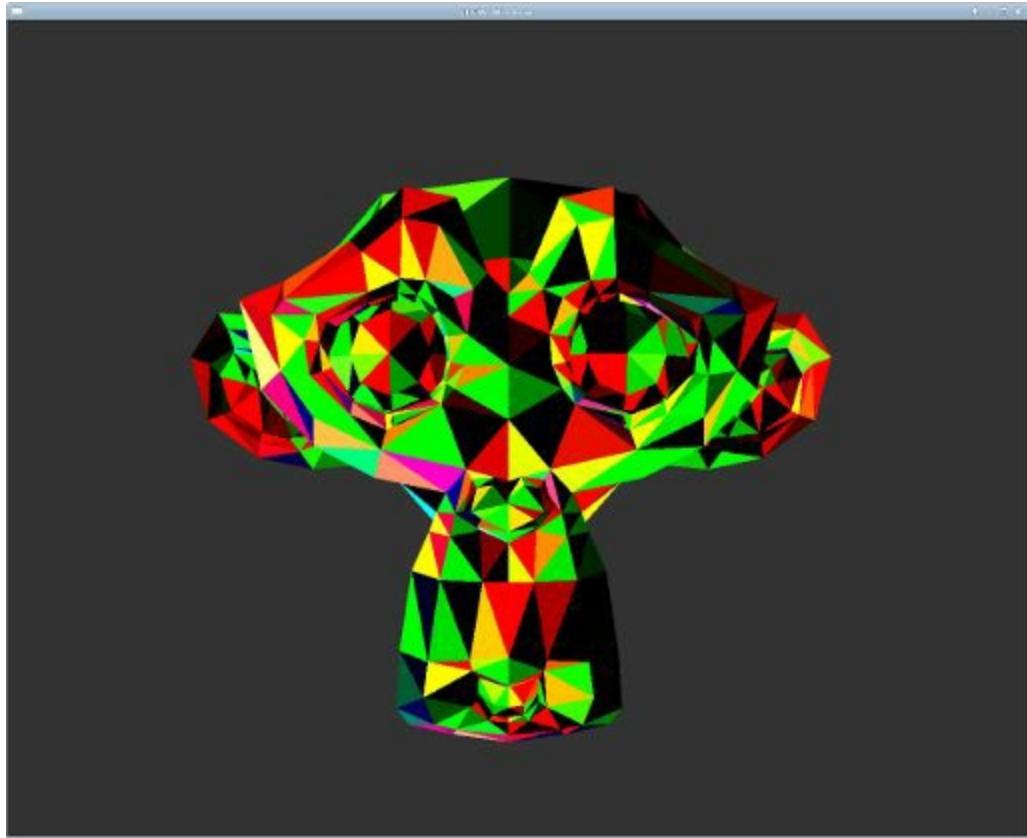
*The sampled normals from the texture, **before** conversion from 0:1 to -1:1 range.*



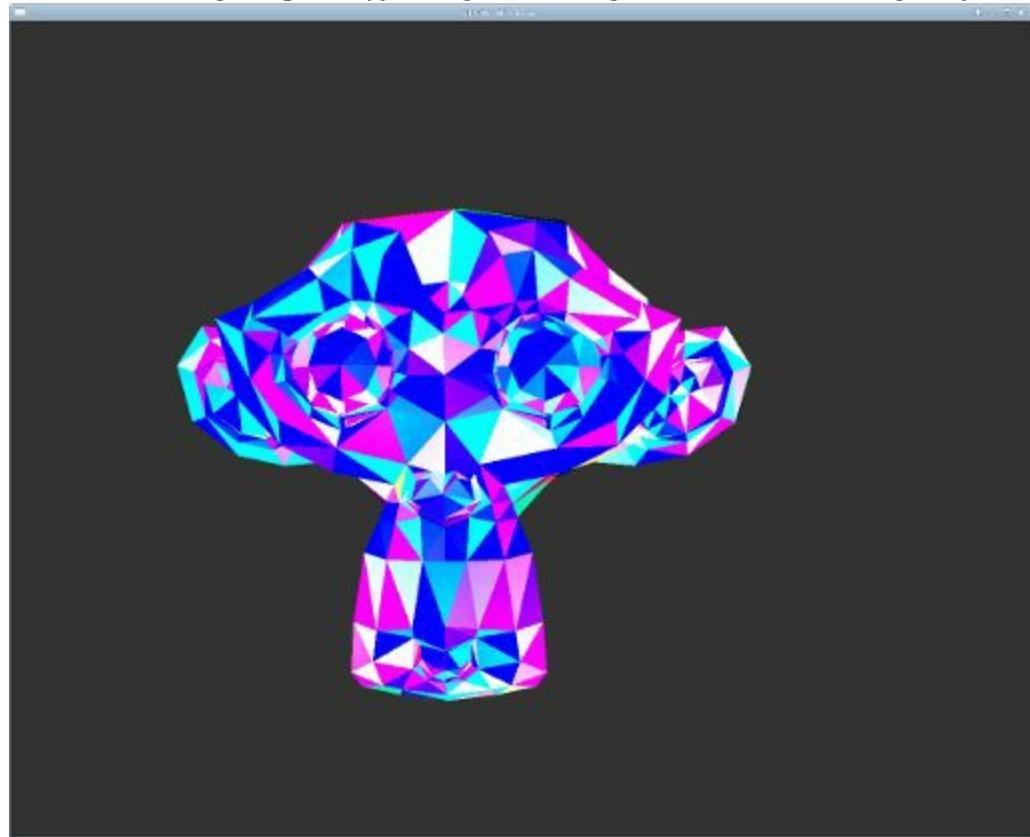
The sampled normals from the texture, **after** conversion from 0:1 to -1:1 range.



The tangents from the vertex shader, rendered as RGB colours.



The light direction in tangent space. If your light is moving, then this should change as you watch it.



The view direction in tangent space. You might observe a "shimmer" of colours as the camera moves.

Common Problems

- **The mesh imports fine, but no tangents are loaded!** - You must export normals with your mesh from the modelling programme, or AssImp will not generate any tangents.
- **My specular highlights are far too bright!** - Remember to normalise the view direction vector and light direction vector in the fragment shader when you use them.

Suggestions

Signed Data Format

When you're comfortable with loading textures from files, you might consider a small improvement for normal maps. Instead of copying unsigned bytes into your texture at `glTexture2D`, you could convert each RGBA texel to a `float` first. You can change the parameters of `glTexture2D` to expect `float` format. The advantage here is that we can normalise and scale into the range of -1.0 to 1.0 before we start drawing. You can then modify the fragment shader and remove the line: `normal_tan = normalize (normal_tan * 2.0 - 1.0);`

Generating a High-Quality Height Map

You can generate the height map instead of manually creating it in an art tool. Most modelling software will have plug-ins for generating a height-map (sometimes called a "bump map") from a more complex version of your mesh. I saw a great trick recently that you could adapt for a quick solution. You create a flat plane in your modelling software, and apply your base texture to it. Subdivide the plane as many times as you want to add detail for, and vertically extrude the surfaces where the texture should have bumps. You can then generate your height map image. You now have a high quality image that actually matches your texture, and the normal map is going to look a lot better. I found the original trick on the [Rayne3D blog](#).

Cube Maps: Sky Boxes and Environment Mapping

OpenGL has a special kind of texture for cubes that allows us to pack 6 textures into it. It has a matching sampler in GLSL that takes a 3d texture coordinate - with R, S, and T, components. The trick here is that we use a 3d direction vector for the texture coordinates. The component with the biggest magnitude tells GL which of the 6 sides to sample from, and the balance between the remaining 2 components tells GL where on that 2d texture to sample a texel from.

Two popular techniques that make use of cube maps are **sky boxes**, which provide the illusion of a 3d background behind the scenery, and **environment mapping**, which can make a very-shiny surface (think lakes, chrome, or car paint) appear to **reflect the environment** scenery. We can also use it to approximate **refraction** for translucent materials like water.

Sky Box

This technique effectively replaces the GL background colour with a more interesting 3d "painted backdrop". The idea with the sky box technique is to have a big box encase the camera as it moves around. If the box appears to be the same distance even when the camera is moving, then we get a **parallax** effect - the illusion of it being relatively very far away. We don't rotate it as the camera rotates though - this allows us to pan around. If it has been well constructed then we won't notice the seams and it will have been projected onto the box so that it appears as if we are inside a sphere. We can actually use a dome or a sphere instead of a box, but boxes are easier to texture.

The size of the box is not going to affect how big it looks. Think about it - if it's moving with the camera it will look exactly the same close up as far away. The size only matters if it intersects with the clip planes - in this case you will see a plane cutting off part of your box. We will **draw the box before anything else**, and turn off **depth-masking**. Remember that this means that it doesn't write anything into the depth buffer - any subsequent draws to the same pixel will draw on top of it. Then it is never drawn in front of your scenery, even if it is closer to the camera. You don't need to use a cube-map texture for a sky box of course; it's just a small short-hand, and only occupies 1 active texture slot.

Make a Big Cube

We need to make a big cube and put it into a vertex buffer. We don't need to add texture coordinates. We are going to be inside the cube, so make sure that you get the winding order of the vertices so that the "front" faces are on the inside. You can load a cube from a mesh file if you like, but I think that's cheating.

```
GLfloat points[] = {
    -10.0f,  10.0f, -10.0f,
    -10.0f, -10.0f, -10.0f,
    10.0f, -10.0f, -10.0f,
    10.0f, -10.0f, -10.0f,
    10.0f,  10.0f, -10.0f,
```

```

-10.0f,  10.0f, -10.0f,
-10.0f, -10.0f,  10.0f,
-10.0f, -10.0f, -10.0f,
-10.0f,  10.0f, -10.0f,
-10.0f,  10.0f,  10.0f,
-10.0f, -10.0f,  10.0f,
10.0f, -10.0f, -10.0f,
10.0f, -10.0f,  10.0f,
10.0f,  10.0f,  10.0f,
10.0f,  10.0f, -10.0f,
10.0f, -10.0f, -10.0f,
-10.0f, -10.0f,  10.0f,
-10.0f,  10.0f,  10.0f,
10.0f,  10.0f,  10.0f,
10.0f,  10.0f, -10.0f,
10.0f, -10.0f,  10.0f,
-10.0f, -10.0f, -10.0f,
10.0f,  10.0f, -10.0f,
10.0f,  10.0f,  10.0f,
10.0f, -10.0f,  10.0f,
-10.0f,  10.0f,  10.0f,
-10.0f,  10.0f, -10.0f,
-10.0f, -10.0f, -10.0f,
10.0f, -10.0f, -10.0f,
10.0f, -10.0f,  10.0f,
-10.0f, -10.0f,  10.0f,
-10.0f,  10.0f, -10.0f,
10.0f, -10.0f,  10.0f
};

GLuint vbo = 0;
glGenBuffers (1, &vbo);
 glBindBuffer (GL_ARRAY_BUFFER, vbo);
 glBindBuffer (GL_ARRAY_BUFFER, sizeof (points), &points, GL_STATIC_DRAW);

GLuint vao = 0;
 glGenVertexArrays (1, &vao);
 glBindVertexArray (vao);
 glEnableVertexAttribArray (0);
 glBindBuffer (GL_ARRAY_BUFFER, vbo);
 glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

```

Create a Cube-Map Texture

Now, we need to make or find a suitable set of textures for the sky box. There are some great examples at Humus' (Emil Persson's) site: <http://www.humus.name/index.php?page=Textures>, and many older games will have sky box textures that you can experiment with. You might need to

swap the "front" and "back" textures around if they are intended for the opposing coordinate system. Cube maps are sometimes stored inside a single texture file, but more often than not, as 6 separate textures. I'll assume that we are loading from 6 separate textures here. Note that we don't generate mip-maps for a sky box because it's always going to be much the same distance from the camera, and we can just choose an appropriately-sized texture. To test my cube I drew a different coloured border onto each textured face to show where the seams were and check if I needed to swap the front and back textures.

I have a `create_cube_map()` function which takes file names for the 6 separate image files, and generates the opengl texture. This will call another function to load the separate image for each of the 6 sides. Note that each side is going to bind to the same texture that we generated - we will pass this as a parameter. Finally we apply some texture filtering to the texture, and the texture handle is passed back as a function parameter.

```
void create_cube_map (
    const char* front,
    const char* back,
    const char* top,
    const char* bottom,
    const char* left,
    const char* right,
    GLuint* tex_cube
) {
    // generate a cube-map texture to hold all the sides
    glBindTexture (GL_TEXTURE0);
    glGenTextures (1, tex_cube);

    // load each image and copy into a side of the cube-map texture
    assert (
        load_cube_map_side (*tex_cube, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, front));
    assert (
        load_cube_map_side (*tex_cube, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, back));
    assert (
        load_cube_map_side (*tex_cube, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, top));
    assert (
        load_cube_map_side (*tex_cube, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, bottom));
    assert (
        load_cube_map_side (*tex_cube, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, left));
    assert (
        load_cube_map_side (*tex_cube, GL_TEXTURE_CUBE_MAP_POSITIVE_X, right));
    // format cube map texture
    glTexParameteri (GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri (GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri (GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    glTexParameteri (GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri (GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
}
```

The texture filtering parameters are interesting. Make sure that you set "clamp to edge" for all three components. If you don't clamp to edge then you might get a visible seam on the edges of your textures (we'll see why shortly). We have some bi-linear filtering to clean up any minor aliasing when the camera rotates. The function that loads each image is very similar to the one that we wrote in the texture mapping tutorial, and also uses Sean Barrett's image loader (stb_image):

```
bool load_cube_map_side (
    GLuint texture, GLenum side_target, const char* file_name
) {
    glBindTexture (GL_TEXTURE_CUBE_MAP, texture);

    int x, y, n;
    int force_channels = 4;
    unsigned char* image_data = stbi_load (
        file_name, &x, &y, &n, force_channels);
    if (!image_data) {
        fprintf (stderr, "ERROR: could not load %s\n", file_name);
        return false;
    }
    // non-power-of-2 dimensions check
    if ((x & (x - 1)) != 0 || (y & (y - 1)) != 0) {
        fprintf (
            stderr, "WARNING: image %s is not power-of-2 dimensions\n", file_name
        );
    }

    // copy image data into 'target' side of cube map
    glTexImage2D (
        side_target,
        0,
        GL_RGBA,
        x,
        y,
        0,
        GL_RGBA,
        GL_UNSIGNED_BYTE,
        image_data
    );
    free (image_data);
    return true;
}
```

The code is almost identical to creating a single 2d texture in OpenGL, except that we bind a texture of type `GL_TEXTURE_CUBE_MAP` instead of `GL_TEXTURE_2D`. Then `glTexImage2D()` is called 6 times, with the macro for each side as the first parameter, rather than `GL_TEXTURE_2D`.

Basic Cube-Map Shaders

Our vertex shader just needs position the cube and output texture coordinates. In this case a 3d coordinate as a `vec3`. This should be a direction, but I can use the actual local vertex points as a direction because it's a cube. If you think about it, the points along the front face are going to be interpolated with varying values of x and y , but all will have z value -10.0 (or the biggest size in the cube). You don't need to normalise these first - it will still work. The only problem is that the points exactly on the corners are not going to have a "biggest" component, and pixels may appear black as the coordinate is invalid. This is why we enabled clamp-to-edge.

Note: I've seen other tutorials that manipulate the Z and W components of `gl_Position` such that, after perspective division the Z distance will always be 1.0; maximum distance in normalised device space, and therefore always in the background. This is a bad idea. It will clash, and possibly visibly flicker or draw all-black, when depth testing is enabled. It's better to just disable depth masking when drawing, as we will do shortly, and make sure that it's the first thing that you draw in your drawing loop.

```
#version 400

in vec3 vp;
uniform mat4 P, V;
out vec3 texcoords;

void main () {
    texcoords = vp;
    gl_Position = P * V * vec4 (vp, 1.0);
}
```

The P and V matrices are my camera's projection, and view matrices, respectively. The view matrix here is a special version of the camera's view matrix that **does not contain the camera translation**. Inside my main loop I check if the camera has moved. If so I build a very simple view matrix and update its uniform for the cube map shader. This camera only can only rotate on the Y-Axis, but you might use a quaternion to generate a matrix for a free-look camera. Remember that view matrices use negated orientations so that the scene is rotated around the camera. Of course if you are using a different maths library then you will have different matrix functions here.

```
if (cam_moved) {
    mat4 R = rotate_y_deg (identity_mat4 (), -cam_heading);
    glUseProgram (shader_programme);
    glUniformMatrix4fv (V_loc, 1, GL_FALSE, R.m);
```

```
}
```

The fragment shader uses a special sampler; `samplerCube` which accepts the 3d texture coordinate. Other than that there is nothing special about the fragment shader. We use the `texture()` function here.

```
#version 400

in vec3 texcoords;
uniform samplerCube cube_texture;
out vec4 frag_colour;

void main () {
    frag_colour = texture (cube_texture, texcoords);
}
```

Rendering

Use your shaders, and bind your VAO as usual. Remember that we should bind the texture into an active slot before rendering. This time it has a different texture type; `GL_TEXTURE_CUBE_MAP`. With depth-masking disabled it won't write anything to the depth buffer; allowing all subsequently drawn scenery to be in front of the sky box.

```
glDepthMask (GL_FALSE);
glUseProgram (shader_programme);
glActiveTexture (GL_TEXTURE0);
glBindTexture (GL_TEXTURE_CUBE_MAP, tex_cube);
glBindVertexArray (vao);
glDrawArrays (GL_TRIANGLES, 0, 36);
glDepthMask (GL_TRUE);
```





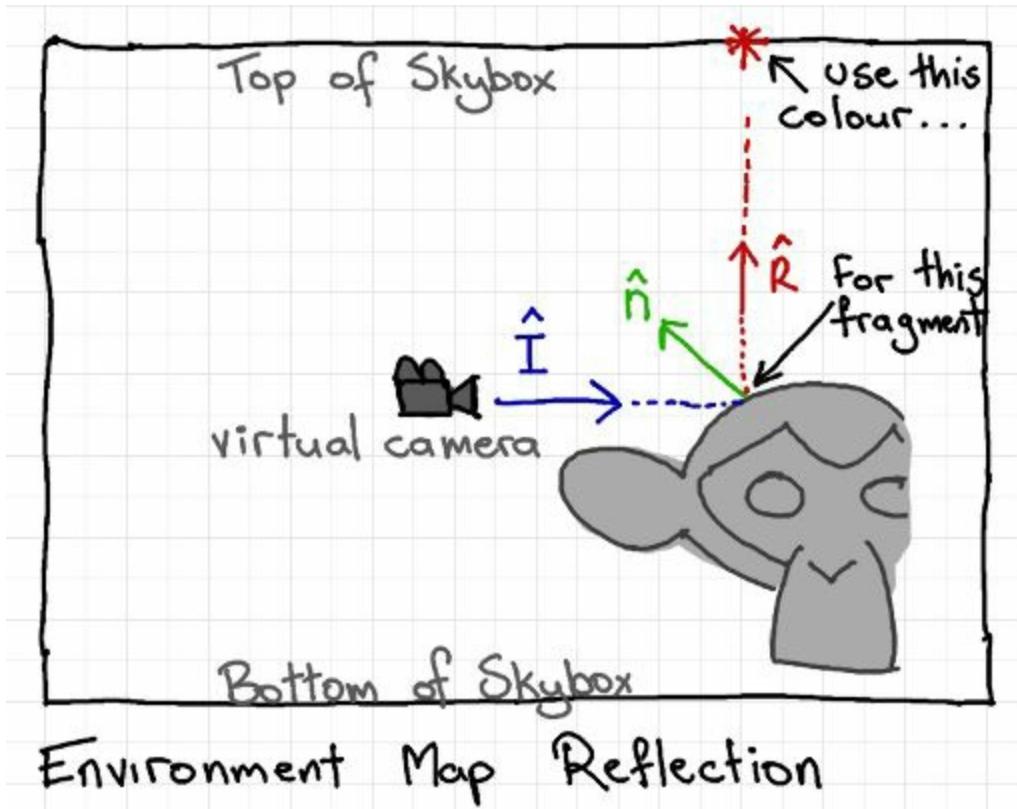
I found this set of sky box textures on Emil Perssons' website, which have a Creative Commons Attribution licence, so they are excellent for demos. (top) points to a corner of the box, viewing the left and front textures, (bottom) points to another corner, viewing the front, and right textures. I suggest testing it with a camera that can rotate.

Reflection and Refraction with Static Environment Maps

So far we have only used specular highlights to show if a surface is highly reflective, which appears as a brightly-coloured dot. Suppose we want to reflect a more detailed image of the sky box or part of the environment onto the surface. If we have a cube map with an image of a brightly-coloured square window on it, then we can use this to create a much more interesting reflection of light on the surface.

Reflecting the Sky Box

The most common type of environment map just reflects a scene's sky box onto a surface. We can say that this is a "static" environment map because it won't reflect any moving pieces of the scene. It can produce quite a convincing effect as you can see that the sky box matches the reflection as you rotate the view. You don't need to have a sky box displayed for this to work though - an indoor scene might not have any visible sky, but you might have a cube map that is not displayed, but contains an indoor-like cube map which you only use for reflections.



With the incident direction vector from the camera to the surface position, I , we calculate a reflection R around the surface normal N . We use R to sample the cube map, and the texel is then applied to the fragment of the surface.

The principle is very simple. We work out a direction vector from the view point (camera position) to the surface. We then reflect that direction off the surface based on the surface normal. We use the reflected direction vector to

sample the cube map. GLSL has a built-in `reflect()` function which takes an input vector and a normal, and produces the output vector. I had some trouble getting this to compile on all drivers, so I prefer to code it manually.

I loaded a mesh of the Suzanne monkey head, and created a vertex simple shader for it. I calculate the vertex positions and normals in eye space.

```
#version 400

in vec3 vp; // positions from mesh
in vec3 vn; // normals from mesh
uniform mat4 P, V, M; // proj, view, model matrices
out vec3 pos_eye;
out vec3 n_eye;

void main () {
    pos_eye = vec3 (V * M * vec4 (vp, 1.0));
    n_eye = vec3 (V * M * vec4 (vn, 0.0));
    gl_Position = P * V * M * vec4 (vp, 1.0);
}
```

The fragment shader calculates the direction vector from the camera to the surface as `incident_eye`; the incident direction in eye space. Note that this is normally `vec3 dir = normalize (to - from)` but the "from" here is the camera position (0,0,0) in eye space. The built-in reflection function then gets the reflected vector, which is converted to world space and used as the texture coordinates.

```
#version 400

in vec3 pos_eye;
in vec3 n_eye;
uniform samplerCube cube_texture;
uniform mat4 V; // view matrix
out vec4 frag_colour;

void main () {
    /* reflect ray around normal from eye to surface */
    vec3 incident_eye = normalize (pos_eye);
    vec3 normal = normalize (n_eye);

    vec3 reflected = reflect (incident_eye, normal);
    // convert from eye to world space
    reflected = vec3 (inverse (V) * vec4 (reflected, 0.0));

    frag_colour = texture (cube_texture, reflected);
}
```

You can reduce the number of instructions in the shader if you compute the reflection in world space, but then you need a camera world position uniform.



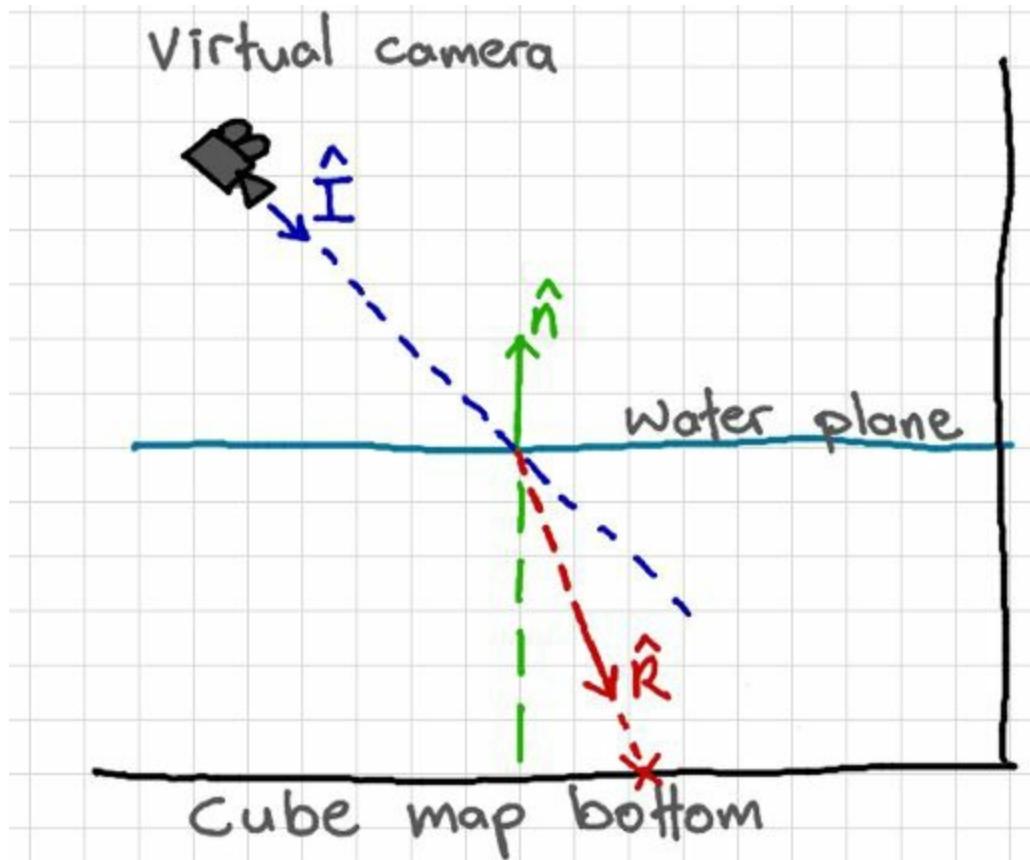


Environment maps reflecting the sky box onto a mesh. Using a highly detailed cube map texture will significantly improve the quality of the effect.

Common Mistakes

- My reflection is upside-down or the wrong side! - Check the direction of your incident and normal vectors. Normalise them.
- My mesh reflections are not smooth like your monkey! - Remember to use interpolated surface normals (set all faces to smooth normals in Blender).

Refraction



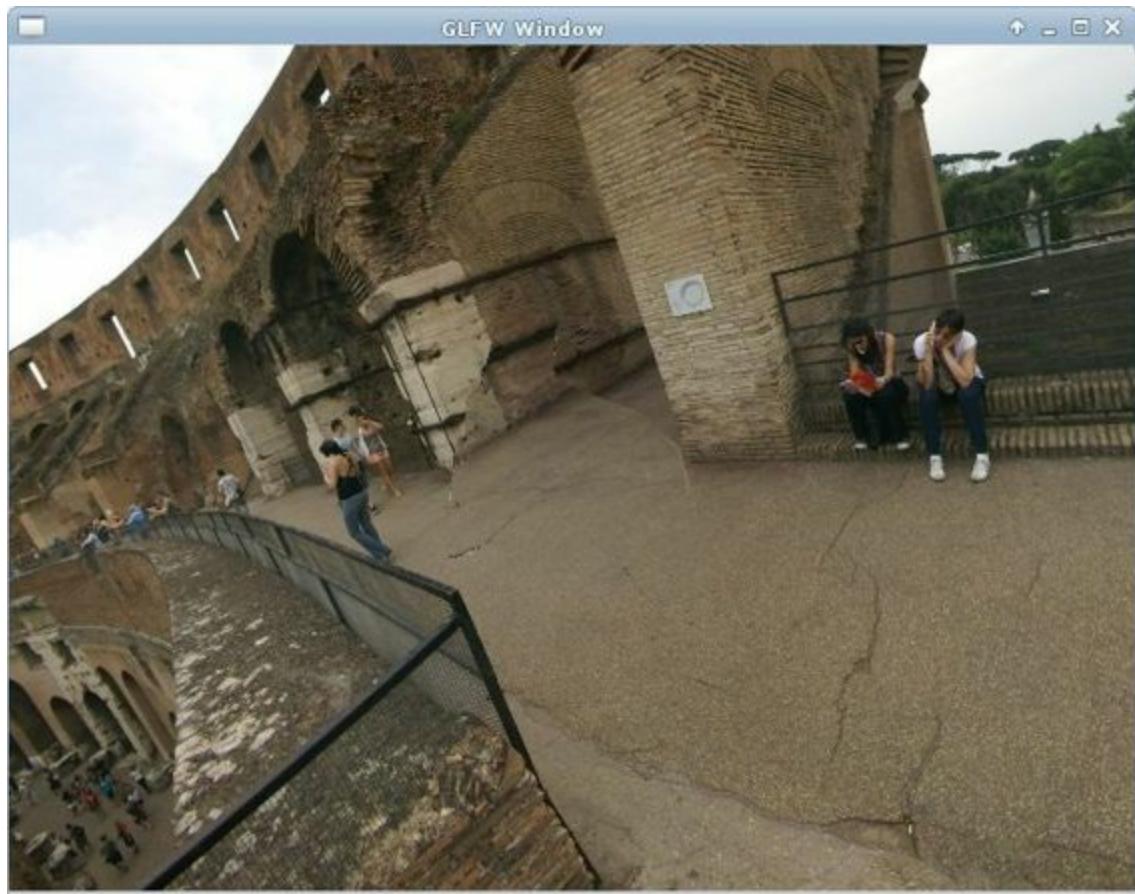
Environment Map Refraction

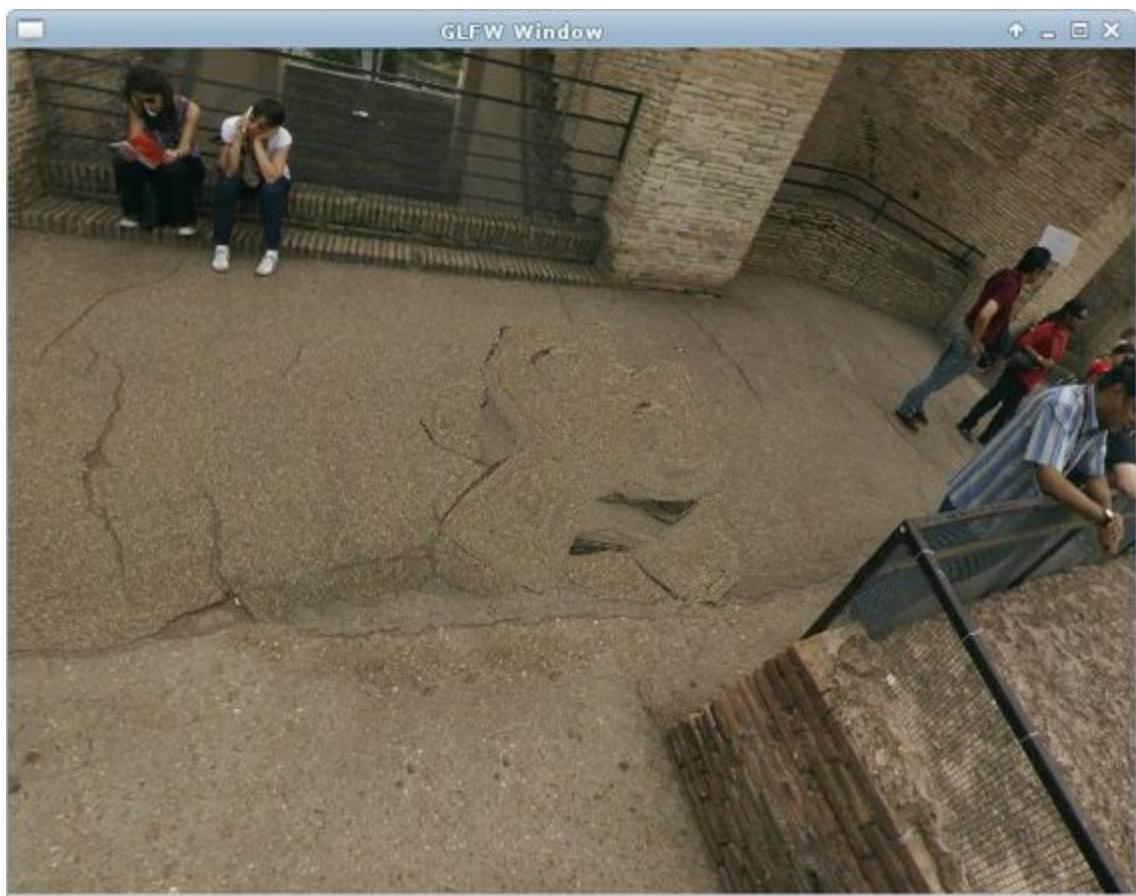
With the incident direction vector from the camera to the surface position, I , we calculate a refraction R which modifies the incident vector based on the surface normal, N , and the ratio of refractive indices, which is the first index (1.0 for air), divided by the second index (1.3333 for water). With the above example, you might use a cube map that displays the interior of a swimming pool, rather than the sky box.

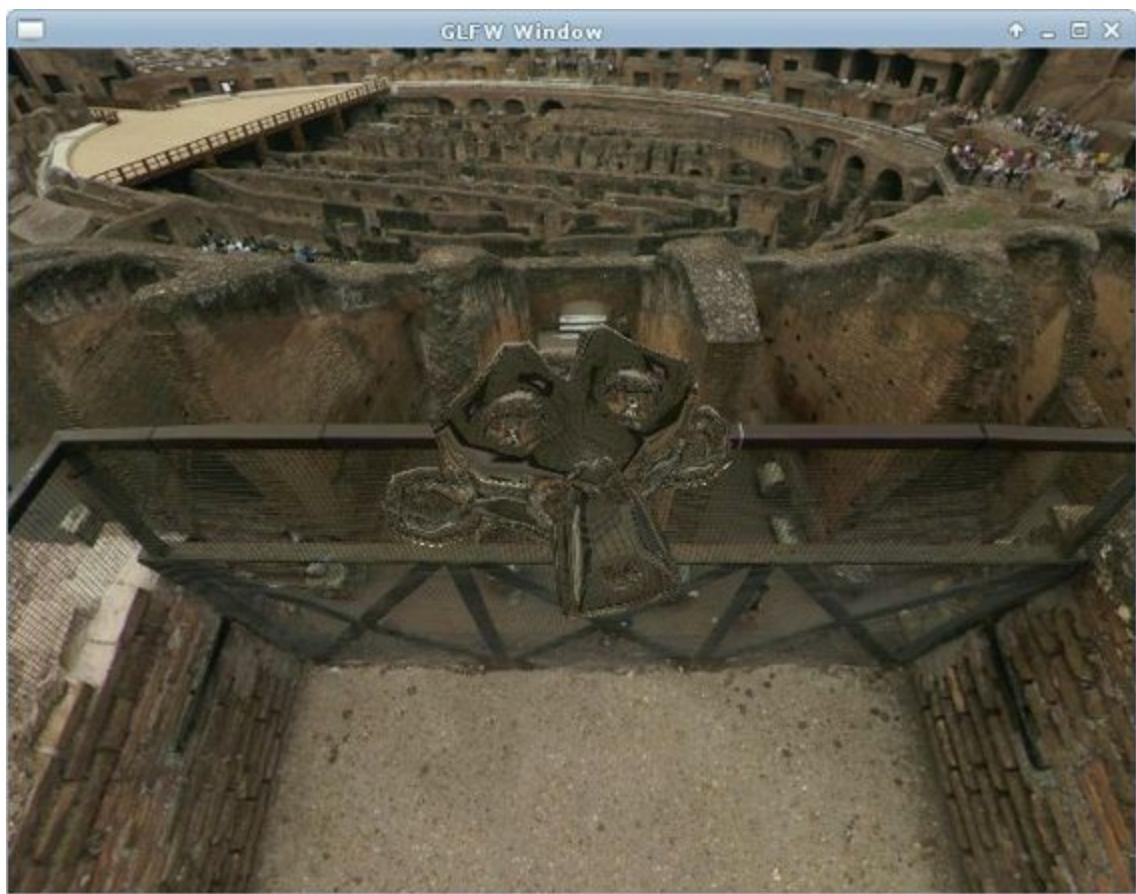
Translucent objects should **refract** light. This means that it doesn't go straight through; the ray bends when it hits the surface and changes material. Looking into a swimming pool, for example. This works much the same way as reflection, except that we perturb the eye-to-surface vector about the surface normal, instead of reflecting it. The actual change in direction is governed by Snell's Law. GLSL also has a built-in `refract()` function with a vector-based version of Snell's Law. You can modify the previous fragment shader to use this instead of `vec3 reflected`:

```
float ratio = 1.0 /1.3333;
vec3 refracted = refract (incident_eye, normal, ratio);
refracted = vec3 (inverse (V) * vec4 (refracted, 0.0));
```

Where the `ratio` is the ratio between the **refractive index** of the first material (air, 1.0), and the second material (water, 1.3333). You can find some indices for common materials on Wikipedia under "List_of_refractive_indices". Note that the monkey head is single-sided; we are ignoring the effect of the light traveling through the back of the head, where it should refract a second time. So you can say that it's not very realistic, but fine for simple surfaces.









Refraction using indices for air (top), water (second), and cubic zirconia (bottom two).

You can imagine that with the right combination of Phong, vertex displacement, and refraction environment map, that you could make a pretty convincing water visualisation.

Common Mistakes

- My refracted image looks magnified or blocky! - Try using higher-resolution cube map textures.
- My image is upside-down! - This is probably correct. Look at photos of refractions of real, similar surfaces (i.e. sphere, pool). Check that your index ratio is correct (first / second).

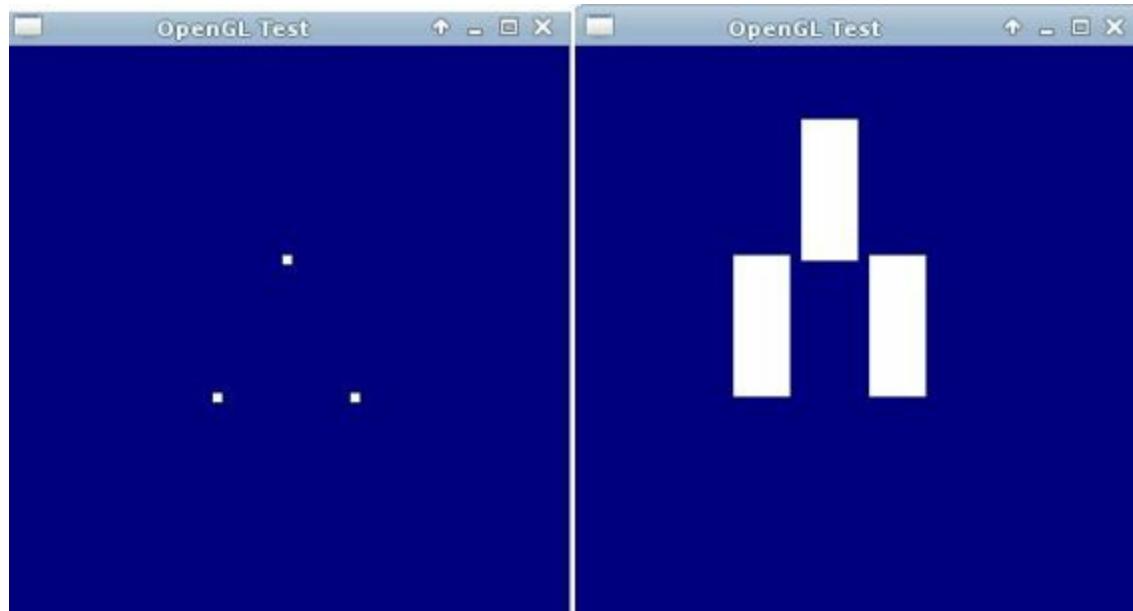
And For Experts...Dynamic Environment Maps

It is possible to use the entire scene, including all the moving objects, in an environment map. If you are already comfortable with binding textures to framebuffers (see later tutorials) then you can bind 6 textures as target outputs from 6 framebuffers. You will render the scene 6 times from the point of view of the object that is going to be environment-mapped. Once pointing up, once down, once left, etc. The output textures will be combined into the cube map. The perspective matrix should have a field of view of 90 degrees so that the 6 frustums line up exactly into a cube.

1. Create 6 colour textures
2. Create 6 depth textures
3. Create 6 framebuffers
4. Attach a colour texture and depth map to each framebuffer
5. When rendering bind each framebuffer and draw in 1 of 6 fixed directions
6. Create a cube map from 6 textures

We looked at how to manually create a cube-map at the beginning of the article. It's also possible to attach all 6 depth and colour maps to a single framebuffer and render all 6 views in a single pass. To do this you will need to use a geometry shader that redirects the vertex positions to separate `g1_Layers`. This might give you a small computational improvement if you are computing dynamic environment maps in real-time, but I find geometry shaders to be unstable across the range of drivers and would not recommend this option.

Geometry Shaders



Here we have a rendering with 3 points (left), and using a geometry shader to redraw the points as billboards made from 2 triangles each (right)

Geometry shaders are quite easy to use, but they are not available in all versions of OpenGL. I have put them late in the series for this reason only.

They were available as extensions to OpenGL 3, and at the time they were a tool for doing level of detail (LOD) techniques on hardware, such as adding or removing detail from meshes as they moved closer or farther from the view point. They were not particularly efficient at sub-dividing geometry (a process called tessellation), which led to the creation of specialised hardware and shaders for that task.

Geometry shaders can:

- gather whole, assembled geometric primitives (triangles, points, lines, etc.)
- output new geometry, but only as points, line strips, or triangle strips
- delete vertices
- split geometry into "layers"; each layer will be sent to a different frame

- buffer
- gather information about adjacent geometry

Some potential uses of geometry shaders:

- creating "imposters" (billboards) from a single input vertex
- extrusion of geometry
- culling algorithms that progressively remove only parts of a vertex buffer
- calculating facets from tessellation shaders
- generating a wire-frame mesh in one pass

There is no need to create square-shaped imposters (2d sprite replacements for 3d meshes) or billboards (geometry that always faces the viewer) using geometry shaders, because we can do that in OpenGL by rendering points and determining the size. The texture coordinates will automatically be generated for us for "point sprites". But, if we wanted to create a non-square billboard then geometry shaders are ideal for this purpose. The vertices have already been transformed into homogeneous clip space by the vertex shader, so we know that $\pm X$ will be right and left on the screen.

Aside from the obvious uses, extrusion is a useful technique for generating visible outlines around meshes, which can enhance techniques like cartoon or non-photo-realistic rendering. It is also used to generate sharp, volumetric shadows. Philip Rideout has an excellent post on that topic:

<http://prideout.net/blog/?p=54>

Usage

- You must compile your geometry shader in the same way as vertex and fragment shaders.
- You must add layout qualifiers in the shader to say what type of geometry to expect as input, in the format; `layout (points) in;`. These are limited to `{points, lines, triangles, lines_adjacency, triangles_adjacency}`
- You can use the `gl_VerticesIn` to get a count of input vertices
- You must add layout qualifiers to say what type of geometry to output, in the format; `layout (triangle_strip, max_vertices = 4) out;`. These are limited to `{points, line_strip, triangle_strip}`
- The layout qualifier `max_vertices` must be at least as large as the number of vertices emitted.
- Any "pass-through" data that goes from vertex shader to fragment shader (such as normals or texture coordinates) must also be passed through by the geometry. shader.
- `gl_Position` must be set before emitting each vertex with `EmitVertex()`
- Any vertices not emitted are deleted from the pipeline. You can therefore delete whole triangles with a geometry shader.
- In GLSL 4.0.0 and newer we can also specify that a geometry shader should run a number of times per input vertex by adding an input layout qualifier.

Example: Billboards and Imposters

I wrote the following shader to take each transformed vertex point from the vertex shader, and from it create two triangles to make a billboard:

Geometry Shader Example

```
#version 400

layout (points) in;
// convert to points, line_strip, or triangle_strip
layout (triangle_strip, max_vertices = 4) out;

// NB: in and out pass-through vertex->fragment variables must go here if used

void main () {
    for(int i = 0; i < gl_VerticesIn; i++) {
        // copy attributes
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
        // NB: pass-through of variables would go here

        gl_Position.y = gl_in[i].gl_Position.y + 5.0;
        EmitVertex();
        gl_Position.y = gl_in[i].gl_Position.y;
        gl_Position.x = gl_in[i].gl_Position.x - 2.0;
        EmitVertex();
        gl_Position.y = gl_in[i].gl_Position.y + 5.0;
        gl_Position.x = gl_in[i].gl_Position.x - 2.0;
        EmitVertex();
    }
}
```

Note that I have set layout input to `points`, and output to `triangle_strip`. I set `max_vertices` to 4, as I will need 4 points to draw both triangles in strip arrangement.

I loop through all the points in the input geometry (in this case `gl_VerticesIn` is going to be 1 because a point has 1 vertex, but we would need a loop like this for triangles or lines).

I set the `gl_Position` variable to the position of each point in the strip, then immediately call the built-in `EmitVertex()` function to output it.

OpenGL 4 Invocations

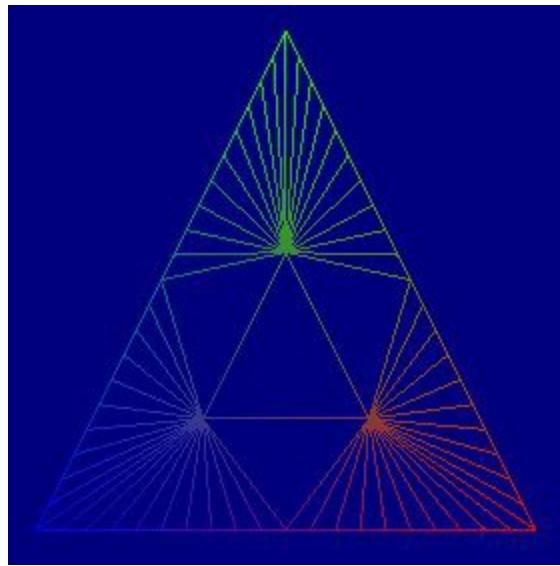
Each geometry shader must produce one primitive, but can produce a strip of primitives. OpenGL 4 gives us the ability to produce more than one primitive in parallel.

```
layout(points, invocations = 4) in;
```

We have access to a variable called `gl_InvocationID` which tells us which instance we are running. So, if we test for the invocation ID we can produce 4 separate triangles, each at a different location around the original input point.

Tessellation Shaders

Overview



This plain triangle has had more geometry detail added to it by using the tessellator unit on OpenGL 4-compatible video hardware. The amount of geometry sub-divisions are determined by tessellation shaders. You can also generate geometry with geometry shaders - these don't use the specialised hardware.

Usage

The most obvious use? Very smooth, very efficient progressive level-of-detail (LOD) for mesh geometry. Using tessellation hardware gives you a massive potential increase to complexity of models in real-time rendering compared to older LOD implementations. The main difference when compared to older LOD methods? Rather than starting with a high-poly mesh and reducing the poly-count as it moves farther away, tessellation creates geometry. You give it a very small number of vertices called "control points", then get it to progressively split these into more and more triangles based on some factors that you change.

Limitations

You must have modern hardware with OpenGL 4 support (Direct3D 11 support) in order to use tessellation, as it **uses a special piece of hardware** that only appears on newer cards. In the last year or two 3d games have been able to expect mass-market support for OpenGL 4-level hardware and we see tessellation as a common feature, but there is still usually a fall-back option to support older hardware.

Hardware Pipeline

Tessellation shaders are optional, and are executed immediately after the vertex shader. If a geometry shader is used, then the geometry shader is executed after tessellation.

References

I suggest reading this excellent blog post on the subject (one of the first reliable resources about GL4 tessellation) <http://prideout.net/blog/?p=48>. The implementation is a little over-complicated for a basic demo, so I've made a simple-as-possible demo here. I also read a great article about generating very high resolution terrain, using the height-maps approach, on Florian Boesch' blog: <http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/>

Programming Overview

The process works like this:

1. vertex points are divided into **control patches** of a specified number of points each
2. a **tessellation control shader** is run once per patch, and specifies how many intermediate points to generate
3. the **tessellator hardware** generates that number of points using a built-in tessellation algorithm

4. the **tessellation evaluation shader** deforms/moves the generated points

We have two stages of the tessellation process to specify using special shaders:

- tessellation control shader (TC)
- tessellation evaluation shader (TE)

Both shaders operate once per-vertex. When using tessellation shaders, we draw the new primitive type; `GL_PATCHES`, which can have a variable number of vertices per patch. We specify the vertices-per-patch before drawing;

```
glPatchParameteri (GL_PATCH_VERTICES, number_of_vertices);
```

If our input mesh is divided into triangles that we want to tessellate, then we would use a parameter value of 3. It doesn't have to be triangles though - we could use quads, for example. Each patch is consumed by a tessellation control shader. It is also possible to set default tessellation factors with this function.

You can query your hardware's patch size support:

```
int max_patch_vertices = 0;
glGetIntegerv (GL_MAX_PATCH_VERTICES, &max_patch_vertices);
printf ("Max supported patch vertices %i\n", max_patch_vertices);
```

Simple Example

Set up a simple programme to render 3 points of a triangle. Ensure that they are visible. Make a note of any culling modes that you use - back-face or front-face culling. In the following example we have 2 uniforms. I set these to a default of 1.0f and 1.0f, and they modified them $\pm 1.0f$ when different keys were pressed.

C Code

```
glPatchParameteri (GL_PATCH_VERTICES, 3);
...
glDrawArrays (GL_PATCHES, 0, 3);
```

My input geometry is going to be a triangle - 3 points per control patch. If I had a mesh comprising 2 triangles to tessellate, I would still use a value of 3. Note that when drawing later, I have changed the drawing mode, and I still need to specify the total number of points to draw with. If I had 2 triangles in my buffer, I would use point count value 6.

Vertex Shader

```
#version 400

in vec3 vp_loc;
out vec3 controlpoint_wor;

void main () {
    controlpoint_wor = vp_loc; // control points out == vertex points in
}
```

The vertex shader has no real job any more; it just passes the control points through to the tessellation control shader. If using a model matrix then this would be a good place to transform the points from local to world space. You would also do animation here.

Tessellation Control Shader

```

#version 400

// number of CPs in patch
layout (vertices = 3) out;

// from VS (use empty modifier [] so we can say anything)
in vec3 controlpoint_wor[];

// to evaluation shader. will be used to guide positioning of generated points
out vec3 evaluationpoint_wor[];

uniform float tessLevelInner = 4.0; // controlled by keyboard buttons
uniform float tessLevelOuter = 4.0; // controlled by keyboard buttons

void main () {
    evaluationpoint_wor[gl_InvocationID] = controlpoint_wor[gl_InvocationID];

    // Calculate the tessellation levels
    gl_TessLevelInner[0] = tessLevelInner; // number of nested primitives to generate
    gl_TessLevelOuter[0] = tessLevelOuter; // times to subdivide first side
    gl_TessLevelOuter[1] = tessLevelOuter; // times to subdivide second side
    gl_TessLevelOuter[2] = tessLevelOuter; // times to subdivide third side
}

```

The control shader is our knobs-and-dials interface to the hardware tessellator. I control my tessellation factors with uniforms so that I can change them with key-presses.

Tessellation Evaluation Shader

```

#version 400

// triangles, quads, or isolines
layout (triangles, equal_spacing, ccw) in;
in vec3 evaluationpoint_wor[];

// could use a displacement map here

uniform mat4 viewmat;
uniform mat4 projmat;

// gl_TessCoord is location within the patch
// (barycentric for triangles, UV for quads)

void main () {
    vec3 p0 = gl_TessCoord.x * evaluationpoint_wor[0]; // x is one corner
    vec3 p1 = gl_TessCoord.y * evaluationpoint_wor[1]; // y is the 2nd corner
    vec3 p2 = gl_TessCoord.z * evaluationpoint_wor[2]; // z is the 3rd corner (ignore
when using quads)
    vec3 pos = p0 + p1 + p2;
    gl_Position = projmat * viewmat * vec4 (pos, 1.0);
}

```

You use the evaluation shader to move generated points into position. It is

executed once per generated point. The evaluator can work in triangle, quad, or iso-line mode:

- **triangles** mode will create a triangle from every 3 points, and gives you `gl_TessCoord` as a 3d barycentric coordinate which you can position in terms of the surrounding control points' world positions.
- **quads** mode will give you `gl_TessCoord` as a 2d UV coordinate, so that you can position your point relative to 4 surrounding control points.
- **isolines** mode will generate 3d lines.

You specify the mode in an input layout instruction. I told it to assume the default "equal" spacing between generated points, and to assemble geometry in counter-clockwise order, which I am using as "front" faces in OpenGL.

I sent my control points down the hardware pipeline, and I get them here as an array of inputs; `in vec3 evaluationpoint_wor[];`. These will be used as the world positions of the corners of a triangle (corner X, corner Y, corner Z). My generated points are being positioned relative to these, and finally moved into clip space and output with the familiar `gl_Position` variable.

Fragment Shader

```
#version 400  
  
out vec4 fragcolour;  
  
void main () {  
    fragcolour = vec4 (1.0, 1.0, 1.0, 1.0);  
}
```

The fragment shader has no special job. I just colour all of my fragments white here.

Visualising the Tessellation Factors

It's a little hard to see what's happening when rendering solids. I used this little-known OpenGL command to render all the output in wire-frame:

```
glPolygonMode (GL_FRONT, GL_LINE);
```

Displacement Maps

When creating highly detailed meshes in Maya or other modelling software, we can export the following:

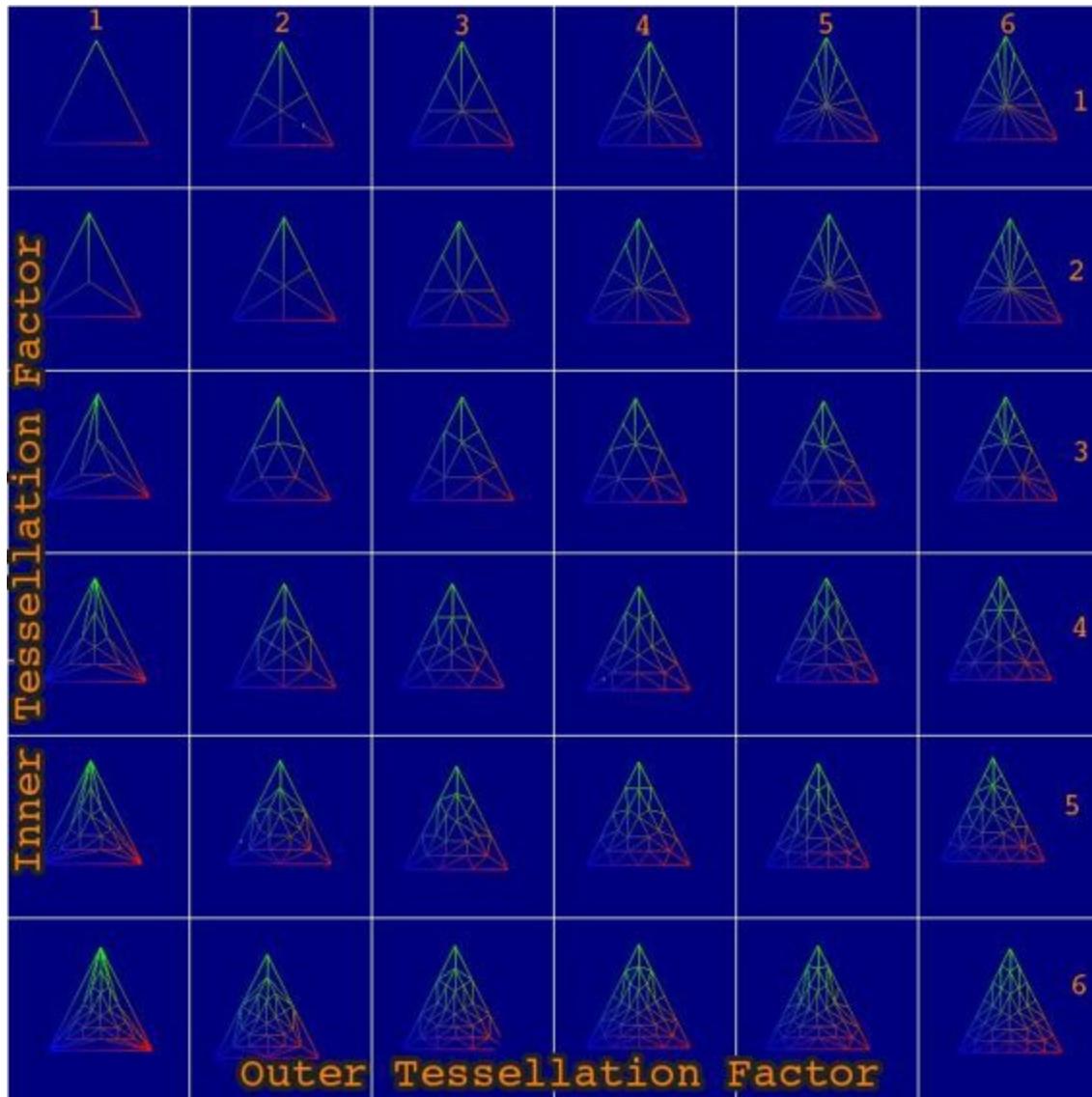
- a low-poly version of the mesh that we will use as our control points
- a displacement map, which we will sample in our evaluation shader to position each generated point

The only complication of this approach is the resolution of the displacement map; it will not map 1:1 texel:vertex, so we will need to calculate weighted averages to position each vertex relative to the surrounding texel values.

This gives us a method for "exporting" a highly detailed mesh, and rebuilding it on hardware. This is where we will get our real gains in rendering efficiency.

Effect of the 2 Tessellation Factors

The image below shows how the tessellation algorithm works when set to generate **triangles** from a set of 3 input control points. I have used evenly spaced points, and have set all 3 sides to the same outer factor value.



The weight that you give to each factor (and the overall range of weights) will depend on your end requirements. Note that the inner tessellation factor doesn't have much effect until it passes a factor of 2. Factors of less than 1 produce nothing. The range of factors can extend far past factor 6. You can

see that keeping inner and outer factors equal produces a regular-looking subdivision.

2d GUI Panels

If we're making a graphical user interface (GUI), or want to display text on the screen, display a mouse cursor, or want to debug something like a depth buffer that is not normally visible, then we want to be able to render to a 2d area of the screen that is always in view. This is very easy to do - in fact, we've done it already in the first tutorials; before we used transformation.



A screen-dump from my game; some GUI panels, and other 2d bits and pieces are on the screen that remain on the screen, even as the camera moves through the scene. This isn't terribly hard to programme; we just don't do any world, view, or perspective calculation. It gets a bit trickier if you want to handle different screen resolutions.

We also need to use this technique when using any kind of post-processing

effect. Typically we will render the first pass of the scene to a texture, then draw a quad (2 triangles) exactly covering the screen area, and apply the texture to it, which can then be modified by the new fragment shader.

Create Geometry for a Clip-Space Panel

Remember that the output of the vertex shader `gl_Position`, expects you to give it co-ordinates in **homogeneous clip space**, which has dimensions (-1:1, -1:1, -1:1, -1:1). Also recall, that after the vertex shader finishes, all points are subject to **perspective division**, which divides the X and Y components by whatever is in the W component after the projection matrix has been applied (hint: -z). Remember that perspective division moves farther-away points closer to the centre of the screen (the railway tracks illusion).

$v' = v / w;$ (perspective division)

We don't want our overlay panels to be subject to distortion, so we are going to set the W component to 1.0. We don't need the Z component, so we can leave that set to 0.0. This just leaves us to set our X and Y components to somewhere in the range of -1:1 to sit on the screen. Let's make a vertex buffer for a panel that fills the entire screen. We can use a matrix to scale it and move it on the screen later, which means that we can re-use this buffer for all panels.

```
float points[] = {
    -1.0f, -1.0f,
    1.0f, -1.0f,
    -1.0f, 1.0f,
    -1.0f, 1.0f,
    1.0f, -1.0f,
    1.0f, 1.0f
};
```

I have made 2 triangles in counter-clockwise winding order. Note that we don't need to make a corresponding array of texture coordinates, because we can work these out in the vertex shader from the points (plus one, divide by two). I'll do the usual GL binding to put this into a VAO setup. Don't forget that the points are 2d sized, **not 3d sized** in the relevant functions:

```
GLuint vp_vbo, vao;
 glGenBuffers (1, &vp_vbo);
 glBindBuffer (GL_ARRAY_BUFFER, vp_vbo);
 glBufferData (GL_ARRAY_BUFFER, sizeof (points), points, GL_STATIC_DRAW);
 glGenVertexArrays (1, &vao);
 glBindVertexArray (vao);
```

```
// note: vertex buffer is already bound  
glVertexAttribPointer (0, 2, GL_FLOAT, GL_FALSE, 0, NULL);  
glEnableVertexAttribArray (0);
```

At this point you might want to load any texture or other stuff that you want to render to the panel.

Create a Shader for Panels

I'll assume that we are not going to use a matrix to scale and position the panel, but rather modify the points inside the vertex shader. If you have several panels, it might be wise to either use a model matrix, or some uniform floats to move and scale the panels separately. If you are doing post-processing you don't need this; you can leave the panel at full-screen size. I'll also assume that we are going to use a texture to colour the panel. The vertex shader is pretty simple:

```
#version 400
in vec2 vp;
out vec2 st;
void main () {
    st = (vp + 1.0) * 0.5;
    gl_Position = vec4 (vp.x, vp.y, 0.0, 1.0);
    gl_Position.xy *= 0.5;
}
```

The only real difference here to other shaders that we have made, is that our points are not combined with a view or projection matrix; we just move the points directly. The fragment shader will be the same as the one that we used for texturing.

```
// load shaders into string here
...
GLuint gui_sp; // shader programme
GLuint gui_vs = glCreateShader (GL_VERTEX_SHADER);
glShaderSource (gui_vs, 1, &gui_vs_str, NULL);
glCompileShader (gui_vs);
GLuint gui_fs = glCreateShader (GL_FRAGMENT_SHADER);
glShaderSource (gui_fs, 1, &gui_fs_str, NULL);
glCompileShader (gui_fs);
gui_sp = glCreateProgram ();
glAttachShader (gui_sp, gui_vs);
glAttachShader (gui_sp, gui_fs);
glLinkProgram (gui_sp);
```

Basics done!

Resizing Panels

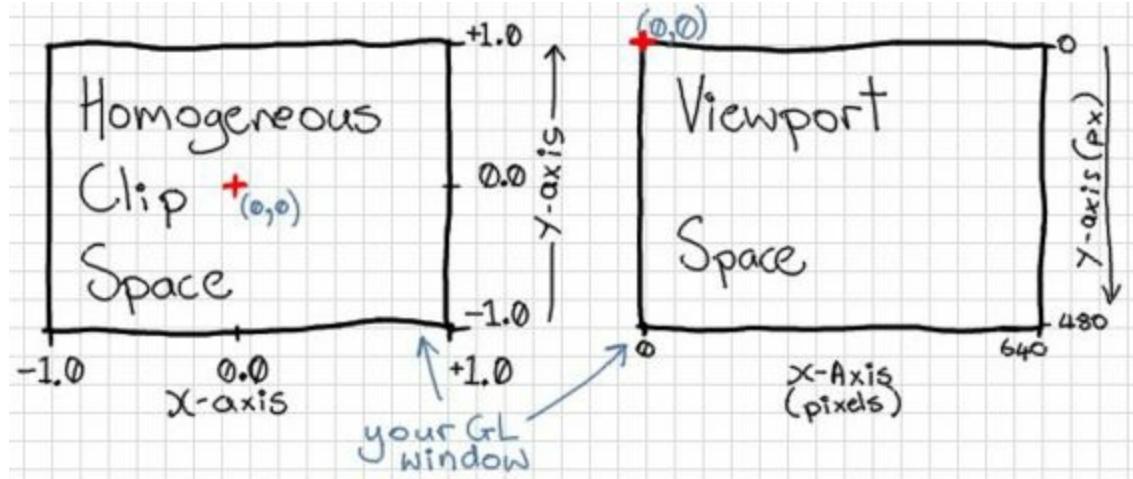
Now we have an interesting problem; we can scale, translate, and even rotate our panels *via* a matrix of uniform floats, but what happens if we are making a GUI that should support different screen resolutions? Try resizing the screen. Typically, we want to support:

- Square-window resolutions (aspect ratio 1.0)
- Wide-screen resolutions (various aspect ratios)
- Full-screen desktops (aspect ratio 3/4)

Our textures and text get all stretched and distorted as the aspect ratio changes! Rats! Sometimes you want the GUI elements to get bigger as the screen gets bigger, but they should maintain their own proportions. You could spend all day working out a system so that they are always proportional to the width of the screen, but the height is independent...what a mess! A better system is this:

- GUI elements have a fixed width and height in pixels
- Give the user the option to resize the GUI to fixed {half, regular, double, etc.} sizes.

In other words **don't resize panels automatically - let users do it as they prefer**. This should cover almost all resolutions cleanly without making a mess (and if it does make a mess, it's not your fault!). Unless you have very large, or very small panels, then you probably won't need to resize them at all. Have a look at how games handle this (Quake springs to mind). Most of the large resolutions have teeny-tiny GUI panels, and many players will prefer this.



Remember that the viewport pixels start in the top-left corner, not in the middle of the window, and that the y-axis goes top-to-bottom. I use 640x480px here as example dimensions, but this will change as your viewport resizes, of course.

Actually working this out is pretty easy - just create variables that contain the width and height of your panel in pixels (i.e. the same size as the texture), and before rendering, work out x any y scaling factors required to turn this into the corresponding size in clip space. If the the viewport is the same size as the texture, e.g. 640x480, then the factors will be 1.0. You will, of course, use GLFW's resize callback to update the viewport dimensions whenever the window is resized.

```
// absolute panel dimensions in pixels: 256x256px
const float panel_width = 256.0f;
const float panel_height = 256.0f;
.
glUseProgram (gui_sp);
// resize panel to size in pixels
float x_scale = panel_width / g_viewport_width;
float y_scale = panel_height / g_viewport_height;
glUniform2f (gui_scale_loc, x_scale, y_scale);
 glBindVertexArray (vao);
 glDrawArrays (GL_TRIANGLES, 0, 6);
```



Here, I've scaled a 2d panel to the correct texture dimensions. I can move it around the screen in the same way

My vertex shader now looks like the below:

```
#version 400
in vec2 vp;
uniform vec2 gui_scale;
out vec2 st;
void main () {
    st = (vp + 1.0) * 0.5;
    gl_Position = vec4 (vp * gui_scale, 0.0, 1.0);
}
```

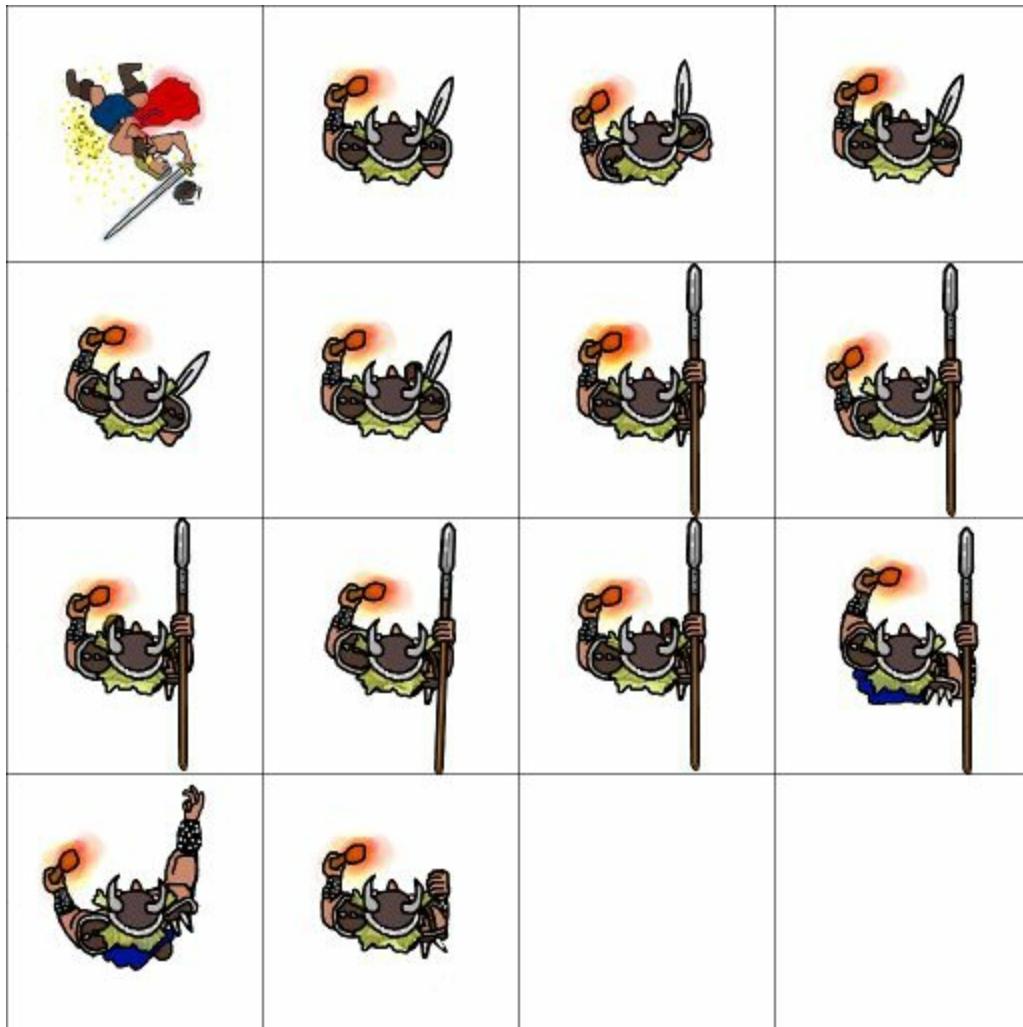
Moving the panel around on the screen, you can either use another uniform variable, or combine all the transformations into a matrix. Here you have the choice of positioning elements either in absolute pixels i.e. the right hand side of the screen -128px, or in clip space, or some combination of both.

Sprite sheets

Sprite sheets (or sprite "atlases") are a very effective way to do 2d animations or on-screen text rendering with any kind of art or outlines as desired. The main advantage is that **you don't have the overhead costs of texture switching**. The idea is very simple:

- create one huge texture
- divide texture into a grid
- put individual sprites into grid slots
- we can now use the same texture but just change texture coordinates sent to the shader as sprites need to change

Art



This is my sprite-sheet image for a little animated game character. It is one large 1024x1024 image with 256x256 sub-images. The game character actually has a lot more animation frames so 2048x2048 might be a better size.

The image here shows the image that I used as a sprite map. I wanted to refer to each sprite with an index; number 0 in the top left, number 1 next to it...down to number 15 in the bottom-right.

C function

I know that I have 4 rows and 4 columns, and that the texture coordinate values go from 0,0 in the bottom left to 1,1 in the top right. So, I can write a little C function to work out the ST values for the top-left corner for a sprite index:

```
void change_sprite (int sprite_index) {  
    const int num_cols = 2;  
    const int num_rows = 2;  
    int col = sprite_index % num_cols;  
    int row = num_rows - 1 - sprite_index / num_rows;  
    float s = (float)col / (float)num_cols;  
    float t = (float)row / (float)num_rows;  
    glUseProgram (sp);  
    glUniform2f (st_offset_loc, s, t);  
}
```

The row number, I inverted so that number 0 is the top row, not the bottom row. You might prefer not to do this, but it felt more intuitive to me. You can see that I send my s and t values to the shader programme as uniform floats (a `vec2`). I would do this whenever an animation frame changes, or before drawing each sprite, if I re-use my sprite for several characters in the scene. I already have texture coordinates worked out for the triangles that I will be drawing the texture on; I can just derive these from the vertex position values, because in this case it's a 2d shape with square dimensions from -1 to 1.

Vertex Shader

```
// shader to render a sprite from a sprite map texture
#version 400

in vec3 vp_loc; // vertex positions

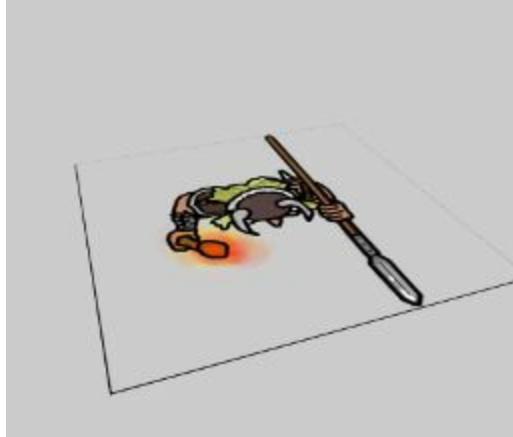
uniform mat4 model_mat;
uniform mat4 view_mat;
uniform mat4 proj_mat;
// texture coordinate for currently selected sprite
uniform vec2 st_offset;

out vec2 st;

void main() {
    // work out regular texture coordinates from vertex positions
    st = 1.0 - (vp_loc + 1.0) * 0.5;
    // now work out offset to get a sprite within the texture
    st = vec2 (st.s / 4.0 + st_offset.s, st.t / 4.0 + st_offset.t);
    gl_Position = proj_mat * view_mat * model_mat * vec4 (vp_loc, 1.0);
}
```

The interesting line is the mapping of our sprite texture coordinate offset values to the per-vertex texture coordinate values. I divided the texture coordinates by the number of columns and rows in the sprite sheet, then added our sprite s and t starting offset. The fragment shader just renders the fragment as per regular texture mapping, using our adjusted `st` for sampling.

Usage



The sprite map in use. I pressed the '6' key in my demo to display sprite index 6 from my map. The grid outlines from my map are still visible - probably I'd want to erase these before use, but it helps us see the edges of the quad here. I turned on alpha-blending so that you can see the grey background through the transparent parts of the image. Otherwise this part of the quad would be rendered black.

In a little OpenGL demo I use keyboard number keys 1 to 0 to change the sprite index for a plane. In actual use we would probably want a little animation sequence function for game animations or a text to atlas index mapping function for on-screen text. In my sprite map I had semi-transparent regions. In this case it makes sense to enable some alpha-blending before rendering:

```
// allow semi-transparent bits (torch etc)
 glEnable (GL_BLEND);
 glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // partial transparency
```

If I only had fully transparent or fully opaque texels, then it might make more sense to use a fragment rejection approach instead. This might be the case with on-screen text. To make fragment rejection work we just need to check the alpha value of the texel and use the reject; command in the fragment shader if it is greater than some value. Older games with 3-channel rgb images used this approach, but had a special colour (usually pink) that was tested - so if the colour of a texel was (1,1,0,1) then the fragment would be rejected.

If the sprites are going to be overlaid or displayed on top of each other in a 3d

scene then we need to consider other depth options to avoid transparency artifacts. I cover this in the alpha-blending tutorial.

Bitmap Font Rendering

Overview

OpenGL 4 does not have built-in functionality to render text to the screen. For text that does not need to change we can write text into an image, and use that. For text that needs to change; counters, text read from a file, user-input text, and so on, we need to do a bit of extra work.

We can use the same principle as sprite maps for rendering text; we create a big image, divide it up into squares, and put one glyph (character image) on each square. Later we will work out how to map the texture coordinates to a particular letter's glyph.



Advantages

- Dynamic text rendering in 3d or 2d
- Fonts can have colours and outlines
- Can be artistic - doesn't have to use a formalised font

Limitations

- Some technical art work required
- Fonts are not vector graphics and will not scale well to different resolutions
- Monospace (fixed-width) fonts, or fonts all in one case look best (think

Doom). It can be a bit fiddly to manage fonts that have glyphs of different sizes.

You will need to make:

1. A **font atlas**/sprite map image, containing all of your glyphs.
2. The width and position offsets of each glyph, so that you know how to space them. We will store this in a file.
3. C code that takes a string of text, and sends uniforms giving the texture coordinates and width of each character.
4. Shaders to get the texture coordinates for each character and render it to the screen.

There are 2 basic approaches; you can do all of this by hand, or you can use a font loading library to generate the atlas image and a file containing the widths of each glyph. If you are really lucky you can make a font atlas where almost all of the glyphs are the same width. Otherwise both approaches will require some manual tweaking to get right.

Now, I just got a drawing tablet, so I am quite keen to start drawing my own, unique, fonts. The method described here best suits this case. If you would rather use a [boring] font loaded from a TrueType or OpenType file, then you could also use a font rendering library like Freetype or Pango to generate your font image. The next article will discuss generating a font atlas, but we will look at hand-drawing them first, and how to map between a string of text and the glyphs that appear on the screen.

Creating a Font Sheet

How big should the image be? It can be quite big actually. We found out the maximum texture size supported in the Extended Initialisation tutorial. Remember, the texture should be a squared size (128x128, 256x256,..., 1024x1024, 2048x2048, etc.). I used a 1024x1024 pixel image, but you might want higher or lower resolution, depending on how big the letters should be on the screen.

I use the [GIMP](#) for image editing. It's all about layers this time. Add a new transparent layer. In GIMP you can add a grid pattern to the layer (under `Filters→Render→Pattern→Grid`). How many cells do we need in the grid? Well - I want to use all of the glyphs that are in the ASCII table, plus some space for extras like ö å ä. That's basically the last 3 columns here, starting with the space character, and finishing with tilde (94 glyphs).

I want a square space to fit my blockier letters like 'W', and I want to divide the atlas evenly. 8x8 is too small - it would contain only 64 of the 100 characters that I want, so I'll go for 16x16 which gives me lots of extra spaces. That's 64x64px per glyph. You can make a grid with these dimensions. Make sure to zero the intersections and offsets that GIMP likes to include.

We're going to need to define the spacing and size of each glyph later, so let's add a second-level grid (32x32 pixel) as a guide. I made another layer with a slightly different grid colour. Great! Now we have 256 boxes divided into quarters.

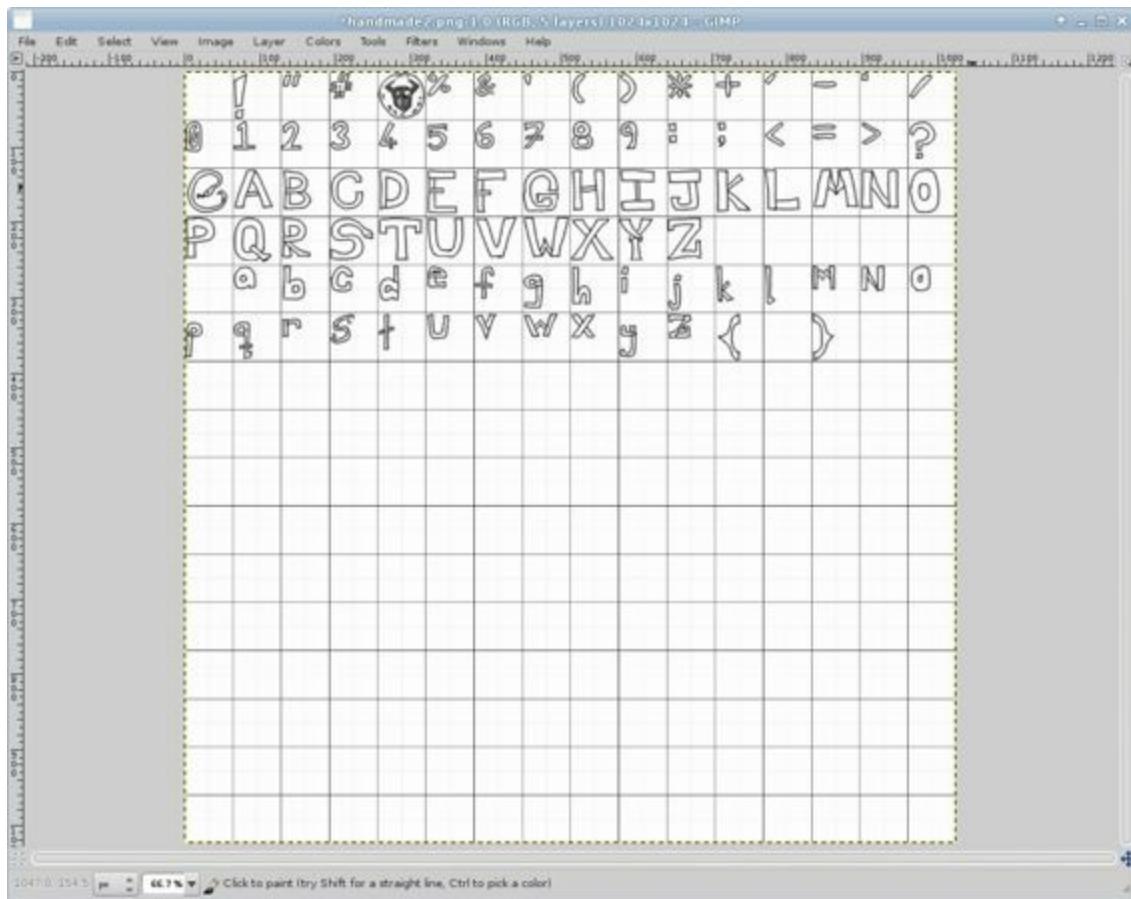
Now, we can add another transparent layer over the grid, and write our glyphs onto it. Remember to leave the top-left box blank for the *space* character.

Drawing Glyphs

Let's set up some conventions to make this easier to work out the spacing for

later.

- Glyphs should sit on the half-way line as a base-line, unless they are too tall.
- Glyphs should start at the left of the box, leaving any unused space on the right.
- Lower-case letters occupy about 1/4 of a box.
- Upper-case letters are about 3/4 of a box long, and 3/4 of a box wide.



I got lazy and skipped a few symbols. The white area and grid will not be saved in the final atlas image, leaving me with a transparent background. Note that I sat the longer letters ('q', 'p', etc.) on the half-way line - this will make them slightly easier to position later. The positions of the glyphs aren't perfect - that's fine, we can tweak them later.

I started by drawing an outline for each glyph, then added a layer underneath, and [badly] coloured them in white. Full white is handy because we can multiply it in the fragment shader to become any colour that we want.

Fragment Shader

It's a good idea to start with a minimal demo that is already loading some geometry that has a texture. Replace the texture with the font sheet, to make sure that it loads and has no issues as a texture.

Then you can replace the geometry with front-facing quads made from 2 triangles. Okay, if so far so good then we can try to pull out one of the letters to use as the whole texture. Divide your texture coordinates by 16. You should see a blank area. Why? Because that's the space glyph that we left blank! See if you can offset this to another glyph by adding 1 before dividing.

Okay, we have the concepts working - next we will write some C code to work out the position on screen of each character, and the texture coordinates of each glyph, and store these are vertex buffers.

Mapping a String of Text to Row and Column Numbers

Okay, we need to make a function to:

1. parse a string
2. create 2 triangles for each character in the string
3. work out each character's row and column in the font atlas
4. create texture coordinates matching that row and column
5. offset texture coordinates to match size of glyph
6. copy points and texture coordinates into vertex buffers
7. return the number of vertices in the buffers

Here's a function that I made to do this. It could be simplified a bit, but this will do for a start:

```
void text_to_vbo (
    const char* str,
    float at_x,
    float at_y,
    float scale_px,
    GLuint* points_vbo,
    GLuint* texcoords_vbo,
    int* point_count
) {
    int len = strlen (str);

    float* points_tmp = (float*)malloc (sizeof (float) * len * 12);
    float* texcoords_tmp = (float*)malloc (sizeof (float) * len * 12);
    for (int i = 0; i < len; i++) {
        // get ascii code as integer
        int ascii_code = str[i];
        ...
    }
}
```

This function accepts a string, and also I give it a clip-space position `at_x`, `at_y` which I will use as the top-left of the first character in the string. A nice trick here is to give it an **absolute glyph scale in pixels**. This lets us ensure that our font won't be stretched or warped if the window aspect ratio changes. Trying to resize your text to match the window size is a small nightmare - it's better to use a fixed size. You can always check if the window is over some threshold size, and if so, swap to a higher resolution version of the font atlas

texture, and double the font scale in pixels. I also give it pointers to a buffer object that I'll store points and texture coordinates in. If you prefer interleaved or concatenated buffers you might change this part. Finally, I take a pointer to an integer that will be awarded the number of points to draw.

The first part of the function gets the number of characters in the string, and allocated memory to store 12 vertex points (2 2d triangles' worth) and 12 texture coordinates. I then start a loop, where I get each character's ASCII code. I'm going to use this to work out which row and column in the atlas this corresponds to. I made some constants `ATLAS_COLS`, `ATLAS_ROWS` to hold this information - these are just the value `16` for my atlas.

```
...      // work out row and column in atlas
int atlas_col = (ascii_code - ' ') % ATLAS_COLS;
int atlas_row = (ascii_code - ' ') / ATLAS_COLS;

// work out texture coordinates in atlas
float s = atlas_col * (1.0 / ATLAS_COLS);
float t = (atlas_row + 1) * (1.0 / ATLAS_ROWS);
...
...
```

I count my rows as being from the top of the image. Once I have the column and row, I can easily work out the texture coordinates of the glyph within the image. Remember that texture coordinates in OpenGL go from bottom to top. This first texture coordinate is going to be the one in the top-left of my glyph, so I add 1 glyph height to the value; `(atlas_row + 1)`, otherwise it will give me an upside-down version of my glyph when I go to draw it.

Next I work out the position of the top-left vertex. These are going to be in absolute clip-space, so I will add the offset position that I gave as function parameters.

```
...      // work out position
float x_pos = at_x;
float y_pos = at_y - scale_px / g_viewport_height *
    glyph_y_offsets[ascii_code];
...
...
```

My Y position is adjusted by `- scale_px / g_viewport_height *
glyph_y_offsets[ascii_code]`. This is so that I can tweak the Y position of each glyph so that shorter glyphs can be brought down to line-up on their base line with taller glyphs. The `scale_px / g_viewport_height` makes sure that the offset is

a facot of the on-screen size of the glyph, where `g_viewport_height` is my window height in pixels. We will set up the `glyph_y_offsets` array later.

We want to tell the character after this one how wide it was, so that it can start a further along on the X axis. I'm just going to modify my `at_x` offset by adding the width of the glyph. I'll fetch this width from another array which we will define later, and also make sure that this is a factor of the maximum on-screen width of the glyph.

```
...
    // move next glyph along to the end of this one
    if (i + 1 < len) {
        // upper-case letters move twice as far
        at_x += glyph_widths[ascii_code] * scale_px / g_viewport_width;
    }
...

```

Now we can just plug in vertex positions for the 2 triangles into the memory that we allocated. Each character has 12 vertex points. The size of each triangle is the maximum width and height of a glyph, which we gave in pixels. We are going to have triangles that overlap when the actual glyph is small, but that won't matter because we will make those bits transparent. Note that the distance here is also in pixels, divided by the viewport dimensions so that we don't get a stretched font. I also work out the corresponding texture coordinates here too, and we reach the end of our per-character loop.

```
...
    // add 6 points and texture coordinates to buffers for each glyph
    points_tmp[i * 12] = x_pos;
    points_tmp[i * 12 + 1] = y_pos;
    points_tmp[i * 12 + 2] = x_pos;
    points_tmp[i * 12 + 3] = y_pos - scale_px / g_viewport_height;
    points_tmp[i * 12 + 4] = x_pos + scale_px / g_viewport_width;
    points_tmp[i * 12 + 5] = y_pos - scale_px / g_viewport_height;

    points_tmp[i * 12 + 6] = x_pos + scale_px / g_viewport_width;
    points_tmp[i * 12 + 7] = y_pos - scale_px / g_viewport_height;
    points_tmp[i * 12 + 8] = x_pos + scale_px / g_viewport_width;
    points_tmp[i * 12 + 9] = y_pos;
    points_tmp[i * 12 + 10] = x_pos;
    points_tmp[i * 12 + 11] = y_pos;

    texcoords_tmp[i * 12] = s;
    texcoords_tmp[i * 12 + 1] = 1.0 - t + 1.0 / ATLAS_ROWS;
    texcoords_tmp[i * 12 + 2] = s;
    texcoords_tmp[i * 12 + 3] = 1.0 - t;
    texcoords_tmp[i * 12 + 4] = s + 1.0 / ATLAS_COLS;
    texcoords_tmp[i * 12 + 5] = 1.0 - t;

    texcoords_tmp[i * 12 + 6] = s + 1.0 / ATLAS_COLS;
```

```

texcoords_tmp[i * 12 + 7] = 1.0 - t;
texcoords_tmp[i * 12 + 8] = s + 1.0 / ATLAS_COLS;
texcoords_tmp[i * 12 + 9] = 1.0 - t + 1.0 / ATLAS_ROWS;
texcoords_tmp[i * 12 + 10] = s;
texcoords_tmp[i * 12 + 11] = 1.0 - t + 1.0 / ATLAS_ROWS;
} // endfor
...

```

At the end of the function we copy our data into vertex buffer objects, free the allocated memory, and set the vertex point count to the length of the string in characters * 6. Note that I set the buffer data format to `GL_DYNAMIC_DRAW` so that OpenGL expects that we want to change the buffer data when our text changes.

```

...
glBindBuffer (GL_ARRAY_BUFFER, *points_vbo);
glBufferData (
    GL_ARRAY_BUFFER,
    len * 12 * sizeof (float),
    points_tmp,
    GL_DYNAMIC_DRAW
);
glBindBuffer (GL_ARRAY_BUFFER, *texcoords_vbo);
glBufferData (
    GL_ARRAY_BUFFER,
    len * 12 * sizeof (float),
    texcoords_tmp,
    GL_DYNAMIC_DRAW
);
free (points_tmp);
free (texcoords_tmp);
*point_count = len * 6;
}

```

You can create the width and y-offset arrays that we referenced, make them size 256 to hold all the characters, and set them to defaults; 1.0f for width factors, and 0.0 for y-offset factors. If you call the function and render the buffers you should get your letters on screen, but horribly out of alignment:

openOffice.org Writer

EN NOW AND SLAY HIM! yelled Ascalante
RANAN ut his back against the wall and
lifted his ax He stood like an image o
f the unconquerable rimordial legs bra
ced far apart head thrust forward one
hand clutching the wall for support the
other gripping the ax on high with the
great corded muscles standing out in i
ron ridges and his features frozen in a
death snarl of fury his eyes blazing t
erribly through the mist of blood which
veiled them The men faltered wild or
criminal and dissolute though they were y
et they came of a breed men called civi
lized with a civilized background here
was the barbarian the natural killer
They shrank back the dying tiger could
still deal death RANAN sensed their une
asertion and grinned mirthlessly and fe
rociously Who dies first he mumbled
through smashed and bloody lips

Normally it would be worse than this, but sticking closely to my grid idea saved me some mess. Notice my 'g' is floating in the air, and my upper-case letters are too big.

Tweaking Glyph Alignments

You can probably see some glitches in your image that you can tidy up at this point, but the alignment we will fix by tweaking the values in those arrays of offsets. It can get a little tedious at this point, but easier for subsequent fonts which can re-use and tweak your first font's offset values. For example, I wanted to move down my 'b' glyph, so I set `glyph_y_offsets['b'] = 0.4f;` where 0.0 would mean "no change", and 1.0 would mean "move it down by one row height". It's a good idea to write these tweaked offset values to a file that can go along with your atlas image.

A good test string is probably the ASCII characters in order. I did this, and tweaked the glyphs in order of appearance until they looked okay.

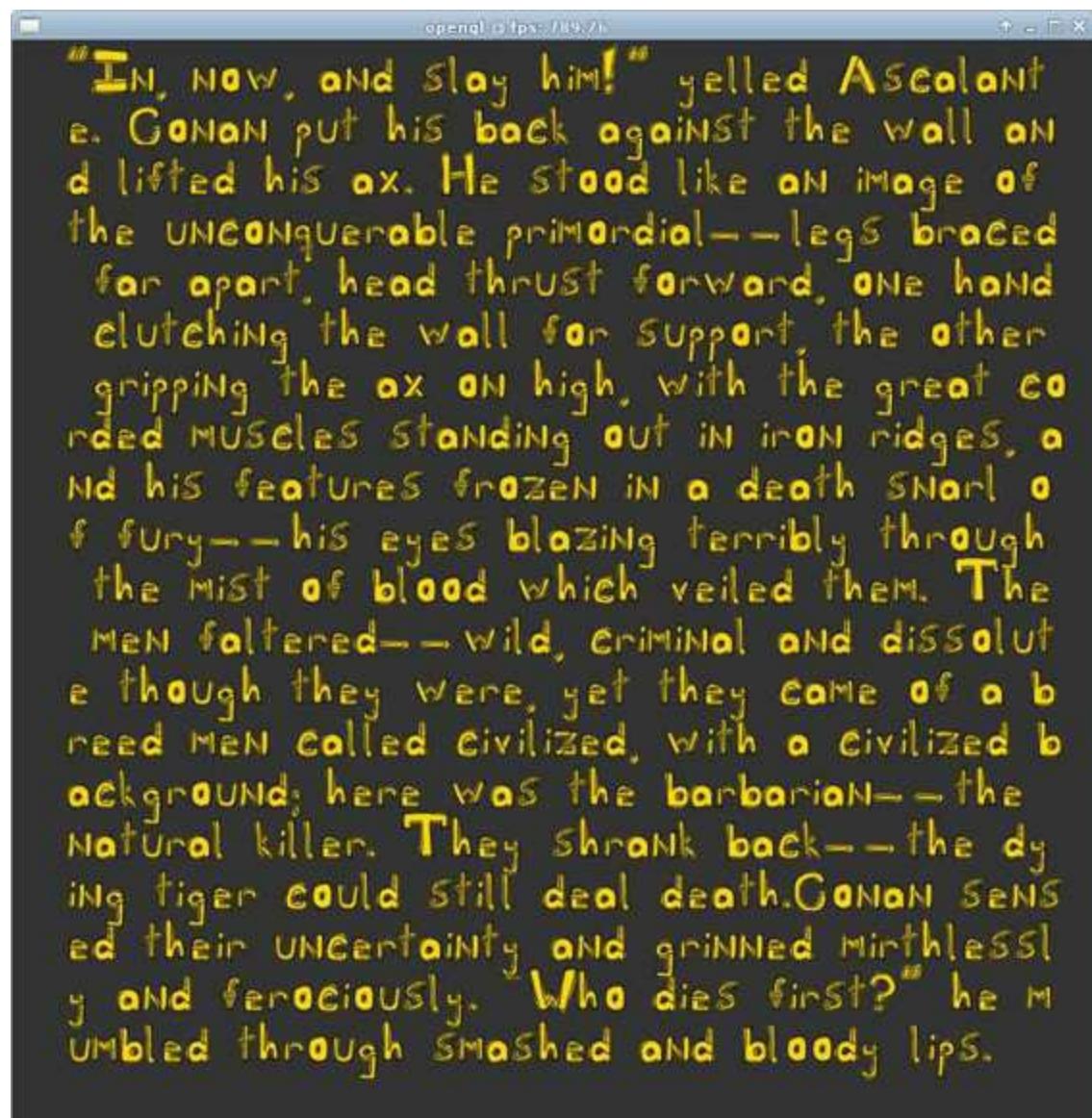
```
" !\"#$%&'()*+`-.:/0123456789;:=>?  
@ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz{}";
```



"What happened to 'p'?" I tweaked my offsets for 'p' until it lined up.

To the Front!

Now, the only natural thing to do is to test it with some, larger, "*Loem ipsum*" piece of text, except not as boring. I made a line break by watching for the sum of character widths to go past a certain point, then reset the sum to 0, and added a Y offset to my subsequent model matrices, to push them down a row. Probably breaking words would have made it easier to read, but I just wanted it to look pretty.



My resulting font looked okay. I spent about an hour drawing the font, about an hour tweaking the

meta-data file, and about 2 hours coding the shader and string parser. You can see that I could spend considerably more time tweaking the graphics and the layout.

Display Improvements

Transparency

To make the background transparent just enable alpha-blending, and set the blend mode. To colour the text I can create a new uniform variable, or just multiply the final colour with another `vec4`.

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

If glyphs' quads can overlap, or you are rendering in 2d, you might want to disable depth testing too (to remove any depth-sorting artifacts from the transparent bits). In this case, make sure that you render the text after the 3d scene.

Improving Rendering Frequency

Point-Sprite Rendering

With point-sprite rendering enabled in OpenGL we are given built-in texture coordinates that we can use in the fragment shader. This will reduce your memory usage, and number of vertex shader executions required to 1/4 or 1/6. I get some unreliable/mixed results with point-sprite rendering on different GL implementations, so I don't use it often.

```
glTexEnvf (GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE); // generate point texture
coords
glEnable (GL_POINT_SPRITE); // enable point sprites

glPointSize (scale_px * (float)g_viewport_width);
```

And finally, we change the rendering mode and number of points:

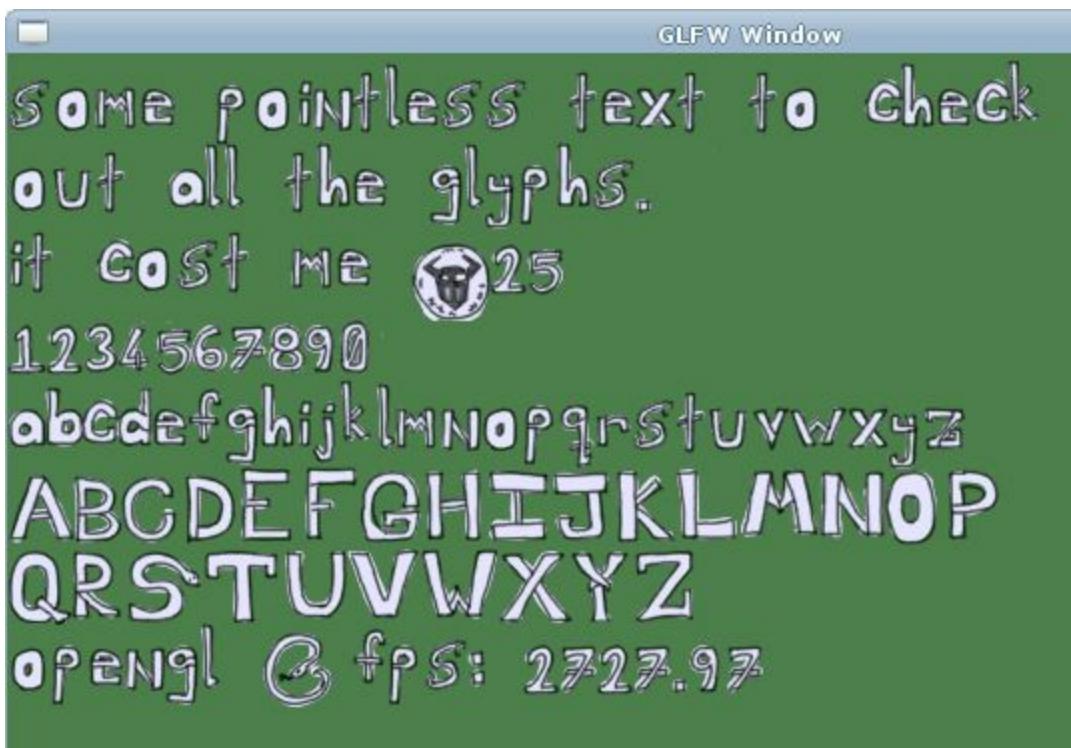
```
glDrawArrays (GL_POINTS, 0, length_of_string);
```

Changing the Text Dynamically

The problem is that it is generally expensive to modify buffers during run-time because the CPU-GPU bus is the tightest bottleneck in the pipeline. We would like to support frame counters, and user-input text as well, so in this case it's probably worth doing anyway.

To update your text, you will need to change;

- The count of glyphs to render as the length of the string changes.
- The texture coordinates of each glyph, as the text changes.
- The position of each glyph, as the width of characters change.



Here, I update a frame-rate counter at the bottom of the text. I ended up modifying the vertex buffer objects during run-time, because testing showed that it wasn't as expensive as feared.

To modify the contents of your vertex buffers, you just need to bind them and call the [glBufferData](#) function again, with the new data. The function will delete any unused memory. I used the data type "dynamic", but this didn't seem to have any noticeable impact on performance.

The trick to keeping this quick is to **delay text updates** to every 0.2 seconds, for example.

Instanced Geometry

You should be able to reduce the memory cost of your vertex buffers, get the same speed as batching geometry in a single buffer, and make the text less expensive to change by using **hardware instancing**, using the `glDrawArraysInstanced()` function. A method to do this with more sophisticated text was described in Game Programming Gems 8, Chapter 1.1; "Fast Font Rendering with Instancing". Unfortunately, I have found most drivers to be very unreliable when using the `gl_InstanceID` variable with more than 1 array access, so I have not found this technique particularly useful.

Less Blurring and Aliasing

If the text looks a bit blurred because of texturing filtering, then you need to consider changing the atlas resolution to more closely match what is shown on the screen (for 2d text). If your text is ever going to be at an angle to the camera (i.e. on walls or floor) then **an-isotropic filtering** is a good idea too, or it will be completely unreadable.

A more advanced method for rendering 3d text that looks good across a range of resolutions is to use **signed distance fields**. This is a feature of Valve's "Source" game engine, and is described in their 2007 SIGGRAPH paper *"Improved Alpha-Tested Magnification for Vector Textures and Special Effects"*. The paper also includes shader code for calculating outlines and glowing effects in the pixel [fragment] shader. Valve has a list of publications at <http://www.valvesoftware.com/company/publications.html>.

Text Formatting

My words are split over lines - you can see that I could write more sophisticated code to handle the text position in C, but this will depend on your intended layout. The easiest approach is to simply watch for the '\n' line-break code when parsing the string, and drop your text down a row when you see this. **Remember not to count the '\n'** as one of your points to render though - or you will have rendering glitches as the wrong memory is read for

the extra few characters.

Multi-Lingual Characters

For more comprehensive strings that support multi-lingual characters, you might want to consider implementing a **Unicode** system instead of an ASCII-based system. The basic Latin Unicode set is exactly the same as ASCII. We have 128 free spaces for glyphs beyond the ASCII set. This is the exact amount of space to fit one of the Unicode supplements to the Latin character set. For most western or northern European languages, the **Latin-1** supplement should do the trick. The codes there are in hexadecimal, and read top-to-bottom for some reason. The Ä glyph would be index 0x00C4, or number 196. Å is 197, and Ö is 214.

Great, so we have some extra glyphs, but C strings use `char` characters - one byte, which only stores values -128 to 127. We have 2 strategies. We could switch to `unsigned char`, which may or may not play well with all of our string parsing functions, or we could use 32-bit `int`, which is more or less the recommended way of doing Unicode in C.

You can see that we would need to make another atlas for other alphabets. There are no Magyar accents, or Latin Slavic accents in the Latin-1 supplement. There are no Greek glyphs, which is not particularly helpful if you want to render a science visualisation with German text and Greek symbols, for example. You may need to use 2 different atlas sheets for this type of scenario, but remember that I excluded all of the control symbols in ASCII - this gives you a few spare slots for special symbols.

Generating a Font Atlas with FreeType 2

Overview

If you've tried to make a font atlas in an image editing tool already, you've noticed how fiddly it is getting the letters lined up in boxes. You more or less have to do this if you're drawing your own font, but if you're using a font from a file then you can semi-automate this process to save you a lot of editing time. The way that I did this was to create a **stand-alone tool** to generate a font atlas image and write a list of font widths to a file. I used the FreeType 2 (<http://www.FreeType.org/>) library to do this, and wrote a little programme in C. This will let you convert OpenType (.otf) and TrueType (.ttf) fonts to images.

Keep in mind that many fonts are not free for all uses, and that you will have to manually add outlines, and do a bit of conversion and tidy-up to these font images. We will go over all of this, as outlines make text that is displayed over a multi-coloured scene much easier to read.

Initialise FreeType

Install the library in the normal way, then you can create a new C programme. Unfortunately, the library has an odd pre-processor macro that you need to include after the header file. We will later use the "bounding box" structure in FreeType, so we will include that header file too. Expect to do the usual sort of file I/O. I am also going to use Sean Barrett's **stb_image_write** library which we looked at in the Screenshot and Video Capture tutorials, but you can use any image writer here, including writing to a plain raw image file and converting it to a PNG with ImageMagick later.

```
#include "stb_image_write.h"
#include <ft2build.h> // FreeType header
```

```
#include FT_FREETYPE_H // unusual macro
#include <ftglyph.h> // needed for bounding box bit
#include <stdio.h>
```

I also have some definitions of different files to use for the font file, the output image, and a plain-text "meta data" file of glyph sizes. You might prefer to load these from command-line parameters.

```
/* using the FreeMono font from the GNU fonts collection. this is free and has a
"copy-left" licence. http://www.fontspace.com/gnu-freefont/freemono */
#define FONT_FILE_NAME "FreeMono.ttf"
#define PNG_OUTPUT_IMAGE "atlas.png"
#define ATLAS_META_FILE "atlas.meta"
```

Now we can initialise FreeType:

```
FT_Library ft;
if (FT_Init_FreeType (&ft)) {
    fprintf (stderr, "Could not init FreeType library\n");
    return 1;
}
```

Next, we can load a font face from a file. You can see where to put your font file name string:

```
FT_Face face;
if (FT_New_Face (ft, FONT_FILE_NAME, 0, &face)) {
    fprintf(stderr, "Could not open font\n");
    return 1;
}
```

Next we can open a file stream to write our atlas image to. We are going to open a file for binary writing, and define some dimensions for our atlas. I decided to use a big, high-resolution 1024x1024 pixel image, cut into 16x16 rows and columns. This fits 256 glyphs. We really only need about 100 glyphs, but I prefer to have extra slots for putting non-ASCII characters. If you are really clever, you can later use an algorithm to squish everything into a smaller atlas and make better use of space, but let's look at the simple case first. 16x16 gives me 64x64 pixels for each glyph. I'm going to add a 3-pixel border around each one, so I'll subtract 6 (3 on each side) from the maximum glyph size.

```
int atlas_dimension_px = 1024; // atlas size in pixels int atlas_columns = 16; //
number of glyphs across atlas int padding_px = 6; // total space in glyph size for
outlines int slot_glyph_size = 64; // glyph maximum size in pixels int atlas_glyph_px
= 64 - padding_px; // leave some padding for outlines /* allocate enough bytes of
memory to store the whole 1024x1024xRGBA image */ unsigned char* atlas_buffer =
(unsigned char*)malloc ( atlas_dimension_px * atlas_dimension_px * 4 * sizeof
```

```
(unsigned char) ); /* a counter to tell us where to put the next byte of data in the  
above */ unsigned int atlas_buffer_index = 0;
```

I want to store the bitmap for each individual glyph in an array, as well as some information about the dimensions, and offsets for each glyph to get them aligned correctly. I'm going to draw the ASCII characters, and there's 128 of them in total, with around 100 actual graphical glyphs, but some fonts have some Unicode characters, up to index 255, so I'll try and draw these too - why not, we have the space. Undefined glyphs will be drawn as squares in the atlas.

I'll tell FreeType the maximum size of each glyph in pixels with

```
FT_Set_Pixel_Sizes ().
```

```
int grows[256]; // glyph height in pixels  
int gwidth[256]; // glyph width in pixels  
int gpitch[256]; // bytes per row of pixels  
int gymin[256]; // offset for letters that dip below baseline like g and y  
unsigned char* glyph_buffer[256] = { NULL }; // stored glyph images  
// set height in pixels width 0 height 48 (48x48)  
FT_Set_Pixel_Sizes (face, 0, atlas_glyph_px);
```

Draw Individual Glyph Images

Now I'm going to loop through all the graphical characters in the range (from 33 to 255, inclusive). I'll ask FreeType to **draw each one separately**, and then copy that memory into a little buffer. Don't accidentally include the *space* character (index 32) in the drawing loop, as FreeType will crash trying to render it. We can just leave the first cell in the atlas blank.

```
for (int i = 33; i < 256; i++) {  
    if (FT_Load_Char (face, i, FT_LOAD_RENDER)) {  
        fprintf(stderr, "Could not load character %i\n", i);  
        return 1;  
    }  
    /* draw glyph image anti-aliased */  
    FT_Render_Glyph (face->glyph, FT_RENDER_MODE_NORMAL);  
    // get dimensions of bitmap  
    grows[i] = face->glyph->bitmap.rows;  
    gwidth[i] = face->glyph->bitmap.width;  
    gpitch[i] = face->glyph->bitmap.pitch;  
    /* copy glyph data into memory because it seems to be reused/lost later */  
    glyph_buffer[i] = (unsigned char*)malloc (grows[i] * gpitch[i]);  
    memcpy (  
        glyph_buffer[i],  
        face->glyph->bitmap.buffer,  
        face->glyph->bitmap.rows * face->glyph->bitmap.pitch
```

```

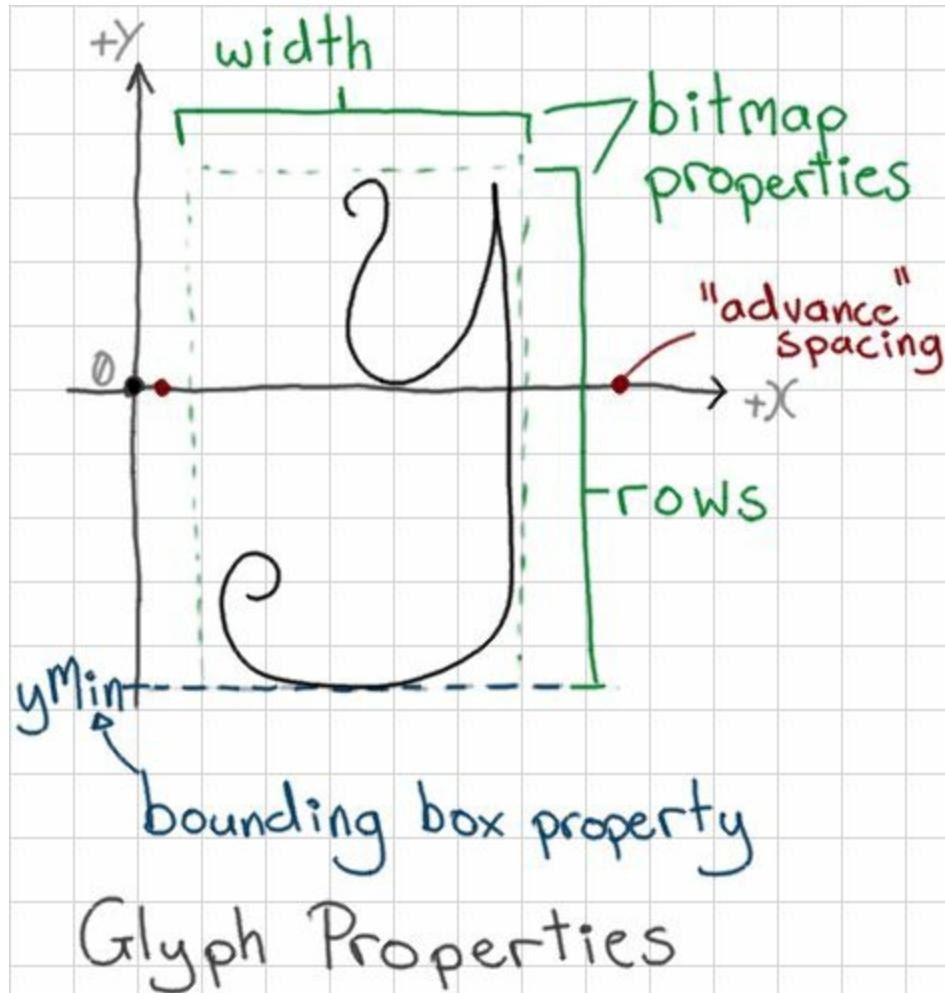
};

/* get y-offset to place glyphs on baseline. this is in the 'bounding box' */
FT_Glyph glyph; // a handle to the glyph image
if (FT_Get_Glyph (face->glyph, &glyph)) {
    fprintf(stderr, "Could not get glyph handle %i\n", i);
    return 1;
}
/* get bbox. "truncated" mode means "get dimensions in pixels" */
FT_BBox bbox;
FT_Glyph_Get_CBox (glyph, FT_GLYPH_BBOX_TRUNCATE, &bbox);
gymin[i] = bbox.yMin;
}

```

So the *i* in the loop corresponds to the ASCII value of each character.

`FT_Load_Char` will attempt to load each character, and `FT_Render_Glyph` will draw it into a "bitmap" (not a formatted bitmap, just a raw image buffer). We can get 3 useful dimensions from the `bitmap` structure; `rows`, which gives the height in pixels, `width`, which gives the width in pixels, and `pitch`, which gives the bytes used per row. I'll use `pitch` to work out the memory used by each glyph when I re-allocate it. I discovered that FreeType re-uses its memory, so if we don't store the bitmap separately it will be over-written by the next one we draw. For this reason I do a `memcpy` (memory copy), and store each image in an array. You can see that I use `rows * pitch` to tell it how many bytes to copy.



FreeType uses several data structures to describe the offsets and dimensions of each glyph. We will pull out some information so that we can line all our characters up on a common base-line.

The next part of the loop extracts the "min" height offset, which I will store to tell OpenGL how to sit each glyph properly on a common base-line. All of our glyphs will be stored in the atlas from top-to bottom, so the shorter letters like 'i' or 'n' will need to be moved down further to line up with the tall letters like 't' or 'l'. Some letters should also hang below the base-line; 'g', 'j', etc. Now, unfortunately this information is stored inside yet another data structure in FreeType, called the "bounding box". Once we have extracted that, then we can finish our glyph drawing loop.

Write Glyph Images Into Atlas Image

I'm going to write to a raw image file in *rgba* format, with one byte per channel, so 32 bits per pixel. I want the background to be transparent, which I'll define as alpha byte equal to 0. I'll loop over every pixel in the atlas, and check if part of one of the glyphs should be written into that pixel. Inside the loops, I'll start by working out which of the 16 rows, and 16 columns the pixel is in - then I know which glyph to read pixels from. Right at the end of the loops, you can see that I write a pixel value (0, 0, 0, 0) - black and transparent, if no glyph occupies that pixel. I remember to offset pixel coordinates in glyphs by the padding (3 pixels to either side). If the pixel in the atlas is outside the dimensions of the matching glyph, then I also draw a transparent pixel. Otherwise I read the matching pixel from the glyph and write that.

```
for (int y = 0; y < atlas_dimension_px; y++) {
    for (int x = 0; x < atlas_dimension_px; x++) {
        /* work out which grid slot[col][row] we are in e.g out of 16x16 */
        int col = x / slot_glyph_size;
        int row = y / slot_glyph_size;
        int order = row * atlas_columns + col;
        int glyph_index = order + 32;

        /* an actual glyph bitmap exists for these indices */
        if (glyph_index > 32 && glyph_index < 256) {
            /* pixel indices within padded glyph slot area */
            int x_loc = x % slot_glyph_size - padding_px / 2;
            int y_loc = y % slot_glyph_size - padding_px / 2;
            /* outside of glyph dimensions use a transparent, black pixel (0,0,0,0) */
            if (x_loc < 0 || y_loc < 0 ||
                x_loc >= gwidth[glyph_index] || y_loc >= grows[glyph_index]) {
                atlas_buffer[atlas_buffer_index++] = 0;
                atlas_buffer[atlas_buffer_index++] = 0;
                atlas_buffer[atlas_buffer_index++] = 0;
                atlas_buffer[atlas_buffer_index++] = 0;
            } else {
                int byte_order_in_glyph = y_loc * gwidth[glyph_index] + x_loc;
                unsigned char colour[4];
                colour[0] = colour[1] = colour[2] = colour[3] =
                    glyph_buffer[glyph_index][byte_order_in_glyph];
                /* print byte from glyph */
                atlas_buffer[atlas_buffer_index+] =
                    glyph_buffer[glyph_index][byte_order_in_glyph];
                atlas_buffer[atlas_buffer_index+] =
                    glyph_buffer[glyph_index][byte_order_in_glyph];
                atlas_buffer[atlas_buffer_index+] =
                    glyph_buffer[glyph_index][byte_order_in_glyph];
                atlas_buffer[atlas_buffer_index+] =
```

```

        glyph_buffer[glyph_index][byte_order_in_glyph];
    }
/* write black in non-graphical ASCII boxes */
} else {
    atlas_buffer[atlas_buffer_index++] = 0;
    atlas_buffer[atlas_buffer_index++] = 0;
    atlas_buffer[atlas_buffer_index++] = 0;
    atlas_buffer[atlas_buffer_index++] = 0;
} //endif
} // endfor
} // endfor

```

Now we can write the image file. It's helpful to convert it to a PNG image, which I do with `stb_image_write`:

```

// use stb_image_write to write directly to png
if (!stbi_write_png (
    PNG_OUTPUT_IMAGE,
    atlas_dimension_px,
    atlas_dimension_px,
    4,
    atlas_buffer,
    0
)) {
    fprintf (stderr, "ERROR: could not write file %s\n", PNG_OUTPUT_IMAGE);
}
/* free that buffer of glyph info */
for (int i = 0; i < 256; i++) {
    if (NULL != glyph_buffer[i]) {
        free (glyph_buffer[i]);
    }
}
/* and the memory of the main atlas image */
free (atlas_buffer);

```



Your atlas image should look something like this, although with a transparent background. This font is missing some of the special characters, which will just be left blank. Note that the letters are not vertically aligned, and some letters will need to be pushed down below the base-line (y, j, g). The letters are also different widths. We will write a meta-data file to describe all of this. We have a lot of space here for additional glyphs - you could put accented characters here, or use an algorithm that rearranges the glyphs into a smaller area.

Writing Glyph Meta-Data

We now have the atlas. If you load it up in an image viewer, you will see that it's not aligned on a common-base line, and many glyphs are different widths, so we will write a file containing this formatting information.

The first line will just be a reminder about what each column in the file is. The GLSL shaders prefer to work in texture coordinates over the entire image, so I'll convert each value into a proportion of the entire atlas. I'm going to store:

1. The ASCII value of each glyph
2. The atlas texture coordinate at the left side of its cell
3. The relative width of the glyph
4. The atlas texture coordinate at the top of its cell
5. The relative height of the glyph
6. The "y offset", or relative amount to move the glyph downwards, to make it sit on a common base-line

```
// write meta-data file to go with atlas image
FILE* fp = fopen (ATLAS_META_FILE, "w");
// comment, reminding me what each column is
fprintf (
    fp,
    "// ascii_code prop_xMin prop_width prop_yMin prop_height prop_y_offset\n");
// write an unique line for the 'space' character
fprintf (
    fp,
    "32 0 %f 0 %f 0\\n",
    0.5f,
    1.0f
);
// write a line for each regular character
for (int i = 33; i < 256; i++) {
    int order = i - 32;
    int col = order % atlas_columns;
    int row = order / atlas_columns;
    float x_min = (float)(col * slot_glyph_size) / (float)atlas_dimension_px;
    float y_min = (float)(row * slot_glyph_size) / (float)atlas_dimension_px;
    fprintf (
        fp,
        "%i %f %f %f %f\\n",
        i,
        x_min,
        (float)(gwidth[i] + padding_px) / 64.0f,
```

```
    y_min,
    (grows[i] + padding_px) / 64.0f,
    -((float)padding_px - (float)gymin[i]) / 64.0f
);
fclose (fp);
```

Modify the Viewing Code to Read the Meta-Data File

We built a bit-mapped font viewer in the previous exercise. We can modify that to load the glyph spacing data from the new meta-data file. Remember that we were setting this manually for each glyph, and adjusting it by eye. We were using just 2 variables for each glyph; an array of `glyph_y_offsets`, and an array of `glyph_widths`. I'll just set these, according to the data in the new file, but we recorded a few extra variables that you can use to tweak spacing even further if you wish.

```
bool load_meta_data (const char* meta_file) {
    FILE* fp = fopen (meta_file, "r");
    if (!fp) {
        fprintf (stderr, "ERROR: could not open file %s\n", meta_file);
        return false;
    }
    char line [128];
    int ascii_code = -1;
    float prop_xMin = 0.0f;
    float prop_width = 0.0f;
    float prop_yMin = 0.0f;
    float prop_height = 0.0f;
    float prop_y_offset = 0.0f;
    // get header line first
    fgets (line, 128, fp);
    // loop through and get each glyph's info
    while (EOF != fscanf (
        fp, "%i %f %f %f %f\n",
        &ascii_code,
        &prop_xMin,
        &prop_width,
        &prop_yMin,
        &prop_height,
        &prop_y_offset
    )) {
        glyph_widths[ascii_code] = prop_width;
        glyph_y_offsets[ascii_code] = 1.0 - prop_height - prop_y_offset;
    }
    fclose (fp);
    return true;
}
```

First, I read and discard the first line in the file (remember, this is the header/reminder strings of what the values are in each column). I use the `ascii_code`, loaded from each line of the file, to index the 2 arrays. The width is very straight forward, the y-offset is a little bit tricky. We use a combination

of the prop height and the y offset to place the glyph on its base-line. This is actually an inverse proportion to what we were using before, so I do $1.0 - \text{the value}$. Now we can display the font, exactly as we were doing before.

Adding a Border, and Tidying Up the Atlas Image

I used GIMP to edit my atlas image. To add a border I used *Select → By Color*, raised the *threshold* value of the tool, and clicked on one of the letters. Then we can grow the selection using *Select → Grow*.... I chose 2 pixels, because I left 6 pixels of padding, and I want an extra pixel of space for the next trick. Then we can create a new transparent layer; *Layer → New Layer*, and move it below the main layer in the *Layers* window. Now we can choose a suitable dark grey colour and bucket-fill one of the selection areas, adding a solid colour border to all of the letters. If it's dark grey, rather than black, it means that it will take on a darker shade of whatever we colour the letters in the fragment shader.

The last thing that you might consider doing is to blur the borders so that they are not so sharp. With the new layer still selected, de-select everything, so that the blur can spread outside the edges *Select → None*, and apply the blur *Filters → Blur → Blur*.



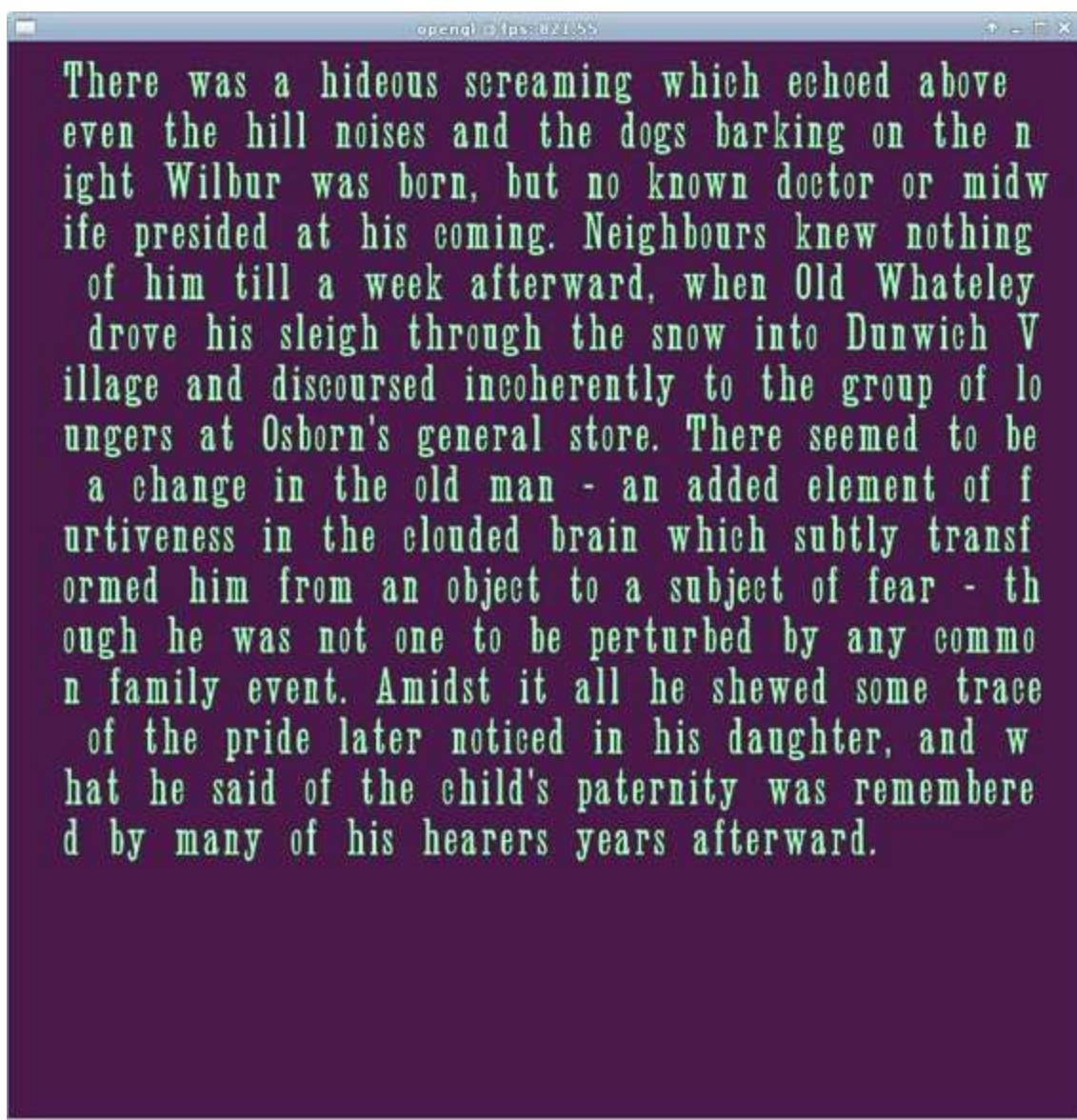
The atlas with a blurred border. Image is scaled and cropped to fit this page.

You might want bolder borders for certain projects, in which case you might make your pixel padding 8 or 10, instead of 6 pixels.

In Use

I'll avoid repeating information about mapping strings to shaders from the previous article.

So, if we read all of that meta-data from the file, and store the values in arrays corresponding to their ASCII code, we can parse any ASCII string and retrieve the matching values. Depending on the font, there may be a few Unicode characters in the extra spots.



openal : lpx:817:79

There was a hideous screaming which echoed above even the hill noises and the dogs barking on the night Wilbur was born, but no known doctor or midwife presided at his coming. Neighbours knew nothing of him till a week afterward, when Old Whateley drove his sleigh through the snow into Dunwich Village and discoursed incoherently to the group of loungers at Osborn's general store. There seemed to be a change in the old man - an added element of furtiveness in the clouded brain which subtly transformed him from an object to a subject of fear - though he was not one to be perturbed by any common family event. Amidst it all he shewed some trace of the pride later noticed in his daughter, and what he said of the child's paternity was remembered by many of his hearers years afterward.

Fonts with outlines remain readable over a variety of background colours.

Common Problems

- **I don't see my text - but substituting for another image works!** - Convert your PNG from greyscale to RGB format, or modify the texture loader to expect greyscale.
- **My letters are not vertically aligned!** - Check that your meta data file is saving sensible-looking yMax offsets, and that you are adding these to the y-offset of the letter without accidentally scaling them first.

Notes

In the code demo I used the "FreeMono" font from
<http://www.fontspace.com/gnu-freefont/freemono>, and in the screenshot
images I am using a free Lovecraft font from
<http://www.cthulhulives.org/toybox/propdocs/propfonts.html>.

Particle Systems

William Reeves published [Particle Systems: A Technique for Modeling a Class of Fuzzy Objects](#) in 1983. The idea is to render nebulous things like clouds, fire, or volumes of water, by drawing clusters of individual particles. Each particle is typically rendered as a single point (i.e. `GL_POINTS`), and scaled up to the desired size in pixels. It's very difficult to render more complex equations of motion and changes of form with the usual mesh surfaces, but we can use Newton's Laws of Motion (kinematics equations) for each particle.

The particle system technique is commonly used in computer games for **special effects**, but it is also used in **computational science** for rendering complex mathematical models of interaction. For an overview see this technical report ([CSTN-052](#)). We can also use non-realistic "pseudo-particles" for rendering surfaces of water or cloth by chaining them all together, and draping a texture over them. A related neat trick is [point cloud](#) rendering of solid surfaces; rendering points can be computationally cheaper than triangulated surfaces, and can be a nice tie-in to a level-of-detail technique. For an idea see Michéal Larkin's [Every Last Detail: Density based level of detail control for crowd rendering](#).

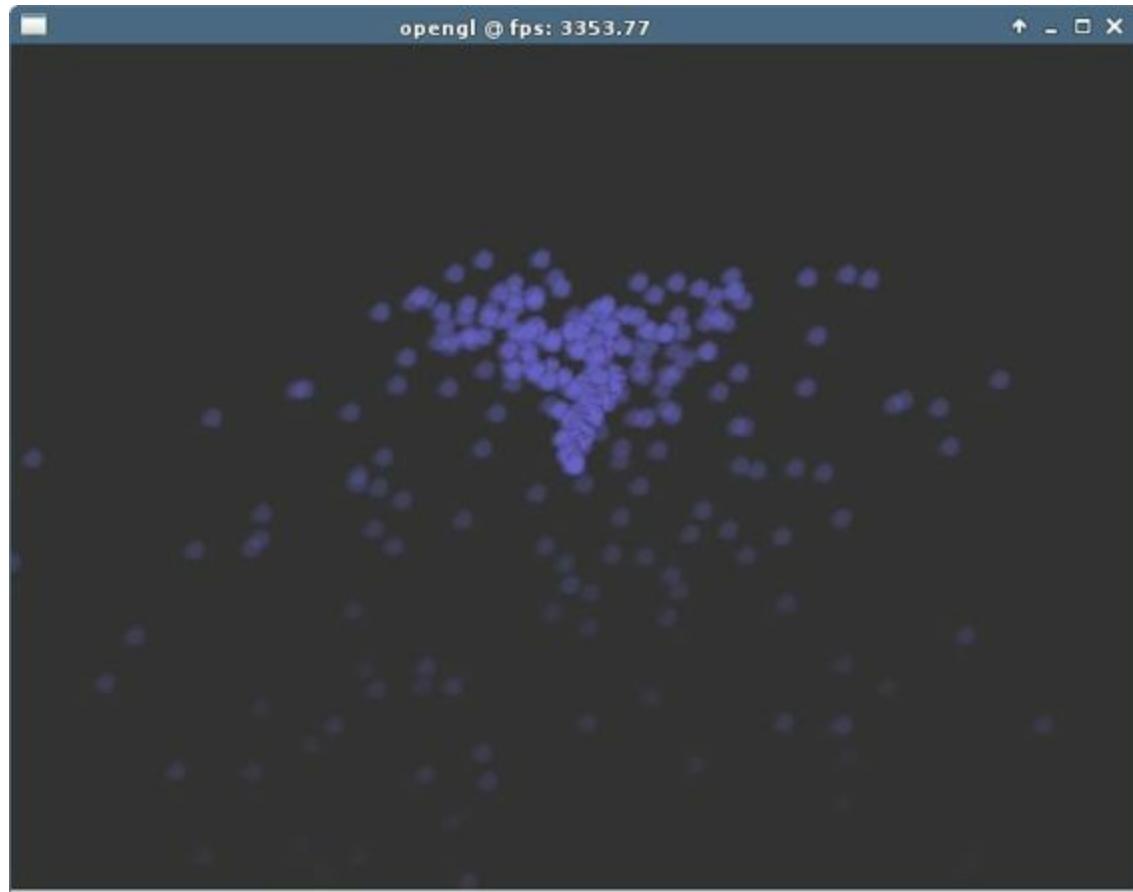
The basic principles behind particles are very simple; you have a self-contained set of points that each follow a basic kinematics equation where only the time varies. The positions are not stored - we work out the complete position each iteration based on the elapsed time of the system. In the basic case the particles do not interact with each other or the scenery. But doesn't this mean that all the particles will all follow the same path? Yes - so what we do is set up different starting conditions for each particle, so that they start at slightly different times, or have different initial velocities. Reeves gives these **particle attributes** in his paper:

1. initial position
2. initial velocity (both speed and direction)
3. initial size

4. initial colour
5. initial transparency
6. shape
7. lifetime

You might have guessed from the word "attributes" per-particle that these are going to be our attributes per-point as well, and therefore attributes per vertex (point rendering means one point is one vertex). So, instead of positions and normals and texture co-ordinates, these are going to be the input attributes to our vertex shader, and therefore also going to be our vertex buffers. This is going to be an unusual concept for programmers of older APIs, where vertex buffers had fixed purposes. Here we can put any data that we like into the vertex buffers, and have them give us data unique to each vertex.

A Simple Example



My particle system "fountain" using point sprites

I won't implement a complete particle system - let's just use a sub-set of Reeves' attributes to build a little particle sprayer. When games first started using particles they all put in "fountain" demos, so let's do a fountain. The attributes that we need are:

1. A 3d vector to represent an initial velocity. I added a little bit of randomness to each one before I put it into the VBO so that they emitted in a funnel shape, rather than all going the same way.
2. A `float` to hold a starting time so that I could stagger emission. The first particle should start at time 0.0, the second at time 0.01, etc.

If you decide on maybe 300 particles, then you can build the 2 vertex buffers.

Create a vertex array object, enable attributes 0 and 1, and provide the attribute pointers for both buffers. Remember that one is 3d and the other is 1d when giving the parameters to your attribute pointer functions.

Here's how you might generate your attributes using the `rand()` function. You can see that I am generating a random number between 0.0-1.0, and then subtracting 0.5 so that it is -0.5 to +0.5 for X and Z:

```
#define PARTICLE_COUNT 300 // tweak me to see if we need more/less particles

/* create initial attribute values for particles. return a VAO */
GLuint gen_particles () {
    float vv[PARTICLE_COUNT * 3]; // start velocities vec3
    float vt[PARTICLE_COUNT]; // start times
    float t_accum = 0.0f; // start time
    int j = 0;
    for (int i = 0; i < PARTICLE_COUNT; i++) {
        // start times
        vt[i] = t_accum;
        t_accum += 0.01f; // spacing for start times is 0.01 seconds
        // start velocities. randomly vary x and z components
        float randx = ((float)rand() / (float)RAND_MAX) * 1.0f - 0.5f;
        float randz = ((float)rand() / (float)RAND_MAX) * 1.0f - 0.5f;
        vv[j] = randx; // x
        vv[j + 1] = 1.0f; // y
        vv[j + 2] = randz; // z
        j+= 3;
    }

    GLuint velocity_vbo;
    glGenBuffers (1, &velocity_vbo);
    glBindBuffer (GL_ARRAY_BUFFER, velocity_vbo);
    glBufferData (GL_ARRAY_BUFFER, sizeof (vv), vv, GL_STATIC_DRAW);

    GLuint time_vbo;
    glGenBuffers (1, &time_vbo);
    glBindBuffer (GL_ARRAY_BUFFER, time_vbo);
    glBufferData (GL_ARRAY_BUFFER, sizeof (vt), vt, GL_STATIC_DRAW);

    GLuint vao;
    glGenVertexArrays (1, &vao);
    glBindVertexArray (vao);
    glBindBuffer (GL_ARRAY_BUFFER, velocity_vbo);
    glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
    glBindBuffer (GL_ARRAY_BUFFER, time_vbo);
    glVertexAttribPointer (1, 1, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray (0);
    glEnableVertexAttribArray (1);

    return vao;
}
```

C Code for Drawing Points and Point Sprites

By default points will be 1 pixel big, but we can resize them in the vertex shader with the build by setting the `gl_PointSize` output variable. For this to work you need to first enable it with the C function `glEnable(GL_PROGRAM_POINT_SIZE);`. After drawing the particles you can disable it again with `glDisable(GL_PROGRAM_POINT_SIZE);` if you like.



I used this texture for point-sprite rendering. Using a texture, combined with alpha-blending lets you have different shaped particles, and will suit effects like smoke or fire that should be partially transparent

You can apply a texture to points that you have scaled up. Video hardware has built-in functionality to automatically generate texture coordinates for your points, and you can access these in built-in variable `gl_PointCoord` in the fragment shader. If you are happy with plain-coloured points then there is no need for this sprite texture.

Note that, if you are not using the **core** profile, and are for some reason running a **compatibility** profile, then `gl_PointCoord` is not supported by default, and you will need to call `glEnable(GL_POINT_SPRITE);`. If you try doing this in core profile, it may throw an error or warning about using a deprecated function. You should probably be using a core profile - we looked at doing this in the first two tutorials.

Once you get that working you might want to try sorting out the transparency of the particles (see chapters on transparency techniques).

My drawing code looks like this:

```

/* Render Particles. Enabling point re-sizing in vertex shader */
glEnable (GL_PROGRAM_POINT_SIZE);
glEnable (GL_BLEND);
glActiveTexture (GL_TEXTURE0);
 glBindTexture (GL_TEXTURE_2D, tex);
glUseProgram (shader_programme);

/* update time in shaders */
glUniform1f (elapsed_system_time_loc, (GLfloat)current_seconds);

glBindVertexArray (vao);
// draw points 0-3 from the currently bound VAO with current in-use shader
glDrawArrays (GL_POINTS, 0, PARTICLE_COUNT);
glDisable (GL_BLEND);
glDisable (GL_PROGRAM_POINT_SIZE);

```

Vertex Shader

The vertex shader just needs to move each particle around based on some simple equation. I used $\mathbf{p} = \mathbf{p}_0 + \mathbf{v} * t + \mathbf{a} * t * t$ type. The rest of the code here is just to work out the current t for each particle, and to make sure that they don't move until they get to their staggered 'start time'. I take them from world to view to clip space using just a view and projection matrix. You could also use a translation matrix to move the whole system to a new spot in the world. I have some extra fluff code here to decide how to fade each particle away over time.

```

/* shader to update a particle system based on a simple kinematics function */
#version 400 core

layout (location = 0) in vec3 v_i; // initial velocity
layout (location = 1) in float start_time;

uniform mat4 V, P;
uniform vec3 emitter_pos_wor; // emitter position in world coordinates
uniform float elapsed_system_time; // system time in seconds

// the fragment shader can use this for it's output colour's alpha component
out float opacity;

void main() {
    // work out the elapsed time for _this particle_ after its start time
    float t = elapsed_system_time - start_time;
    // allow time to loop around so particle emitter keeps going
    t = mod (t, 3.0);
    opacity = 0.0;

    vec3 p = emitter_pos_wor;
    // gravity
    vec3 a = vec3 (0.0, -1.0, 0.0);
    // this is a standard kinematics equation of motion with velocity and
    // acceleration (gravity)

```

```

p += v_i * t + 0.5 * a * t * t;
// gradually make particle fade to invisible over 3 seconds
opacity = 1.0 - (t / 3.0);

gl_Position = P * V * vec4 (p, 1.0);
gl_PointSize = 15.0; // size in pixels
}

```

I put a `core` beside the version number to remind myself about the point texture coordinates needing a core profile. I used a couple of uniform values - the emitter position as a starting point, and the elapsed time for the particle system. This means that you will need to run a counter in your C code, and update this uniform every time that you draw the system. Look at what I did with my `t` value (time). I used the `mod()` function so that it loops back around from 0 after it goes past 3 seconds. This restarts the particle system so that my fountain keeps going. Neat! You should try to reduce the number of particles that we created in the vertex buffer objects down to the minimum size that you can use before it doesn't have enough to loop the animation. This should decrease your draw time.

Fragment Shader

Here I get my opacity value from the vertex shader. I'm using a texture here, but this is optional, and you can remove those parts if you prefer not to use point-sprite rendering. I set a constant here as the initial particle colour here for all of my particles - I chose a blue-ish colour. Nothing too special - I set the alpha component of the fragment colour to the opacity value. The only interesting point is the use of the built-in `gl_PointCoord` variable when sampling the texture.

```

/* shader to render simple particle system points */
#version 400 core

in float opacity;
uniform sampler2D tex; // optional. enable point-sprite coords to use
out vec4 frag_colour;

vec4 particle_colour = vec4 (0.5, 0.5, 0.8, 0.8);

void main () {
    // using point texture coordinates which are pre-defined over the point
    vec4 texel = texture (tex, gl_PointCoord);
    frag_colour.a = opacity * texel.a;
    frag_colour.rgb = particle_colour.rgb * texel.rgb;
}

```

Note that the point texture coordinate t is upside-down to how we usually use texture coordinates. I know! I flip this before using it. You can do that in the shader, but it's probably better to just change the default. Apple drivers don't respect the default, so let's explicitly state it:

```
glPointParameteri (GL_POINT_SPRITE_COORD_ORIGIN, GL_LOWER_LEFT);
```

Pros and Cons of this Implementation

The advantage to this vertex-shader based system is that we can use the parallel processing power of the GPU to compute our system. This means that we can have **lots of particles**. The downside is that the particles have no memory of their previous position; you can't move the emitter around in the world as the particles are being emitted. The particle system is a **canned coordinate system**; the particles don't know anything about the world, or about the other particles.

Common Mistakes

- It's easy to **mix up the locations of attributes** with particle systems. **Check your shader logs** so see if there is a mis-match between your attribute pointers, and the actual location of the attributes in the shader. Whilst you are at it, check that the **dimensionality of the attributes** is correct (3 for velocity, 1 for time). If you get either of these wrong then you will read the wrong data into your shader inputs, and get unusual results.
- Don't set up blending/transparency until after you are sure that it is rendering something.
- Check that the camera is actually pointing at the particle system.
- Remember to set your point size to something large enough to spot easily in pixels i.e. 10 or so.
- Set your system time to 0 first to test, then make sure that your timer is working properly.
- If you can't see your texture - check if you are in a core or compatibility GL profile. Either explicitly create a core profile, or else stick with compatibility, and `glEnable()` point sprites.

Different Implementations

Distance Attenuation

If you are using point rendering, you can reduce the size of the points as they move farther away from the camera. You could work this out yourself manually in the vertex shader, and modify the `gl_PointSize` GLSL variable. There used to be attenuation parameters that you could set with in the C interface, but it looks like they are no longer part of the interface.

Geometry Shaders

There are 2 problems with using point-rendering; the particles have to be square-shaped, and the texture coordinates generated by point-sprite rendering seem to be implemented differently in different GL incarnations. To improve on this, you can use geometry shaders to expand your points into rectangles, or any shape that you like. Geometry shaders will work in OpenGL 3.2, so will also work on modern Macs, but they can be unreliable on some cards that I've tried so I avoid using them for software that I plan to distribute.

Transform Feedback

Transform feedback is a new OpenGL feature which lets you write per-vertex shader data, such as your new particle positions, to a new set of vertex buffers. If you swap to reading these new buffers when rendering the next time, then you can add some dynamic behaviour to your particle system. For example, you can move your emitter through the world whilst emitting particles, and the old particles should be left behind in the physically-correct way. It's an attractive idea but I have found it to be:

1. Very cumbersome to implement (poorly-designed interface)
2. Not widely supported by other GL incarnations (i.e. don't expect it to

- work properly on other computers)
- 3. Not significantly more efficient than CPU-computation for commonly-used particle effects

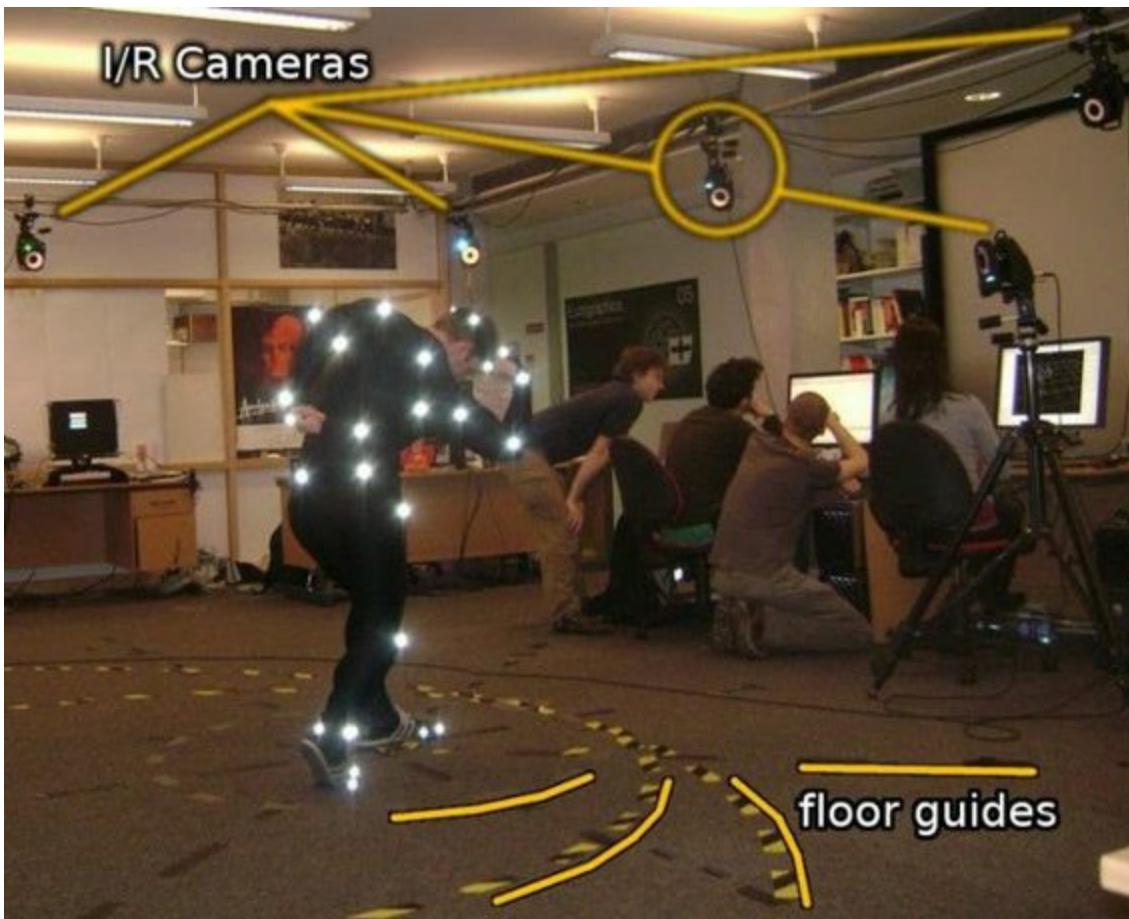
In short, this technique is not worth the effort for most simple effects.

CPU-Computed Particles

The old-fashioned way of computing particles is still useful; take the motion equation out of the vertex shader, and compute it on the CPU instead. This still seems to be the best way to calculate **dynamic** particle systems. The shortcoming is that you lose the parallel-processing power of the GPU, so the system can support a much smaller number of particles, otherwise it's not an issue.

Hardware Skinning

Hardware skinning means using **shaders** to **animate** a 3d mesh that uses an **armature** (skeleton) to deform the vertices. Instead of manually moving each vertex, we tie each vertex to a **joint** in the armature (a **bone** within the skeleton), and move just the joint, which in turn will move all the vertices that are tied to it. This is a useful tool because it allows artists to create pre-canned animations that are suitable for some types of mesh. It is often used in conjunction with **motion capture** ("mocap") animation.



Me, being motion captured. The markers are placed on points where the body can move or rotate, and each one matches a joint in the armature that will be used by a character mesh. 3d locations of each marker are captured during an animation, and these are stored at intervals called "key frames".

How We Are Going To Learn Skinning

This is a very involved and tricky technique to get working as there are lots of separate concepts that you need to get working and combine. Tutorials on the web tend to get you to do everything at once, which can be rather overwhelming. The AssImp API and its data-structures are very confusing. 3d modelling tools and file formats can be tricky. Pretty much everyone gets stuck getting skinning to work properly. I'm going to do break this topic down into little pieces that you can get working incrementally, so that you can master the concepts one-by-one. It's going to be a bit slower to get something working this way, but I've been scratching my head for a long time over this, and this is the best way that I can think of to simplify this rather complicated topic. I'm going to walk through the entire process of creating an animated mesh, importing it, and animating it in OpenGL, because students and friends get stuck on all of these rather tricky aspects of the technique.

I'm going to assume that you have already completed a mesh loader using the AssImp library (as in the earlier tutorial), and that we can build on that.

These are the steps that we are going to take before we have a fully skinned animation in OpenGL:

Part One: Bones

1. Adding bones to a mesh in Blender, and weight-painting vertices to bones
2. Importing bones and weights from a mesh with AssImp
3. Adding bone identifiers and matrices to a vertex shader
4. Using keyboard controls to manually animate the bones in a mesh

Part Two: Skeletons

1. Building a skeleton hierarchy in Blender

2. Importing the hierarchy using AssImp, and re-creating our own tree
3. Building an hierarchical animation function

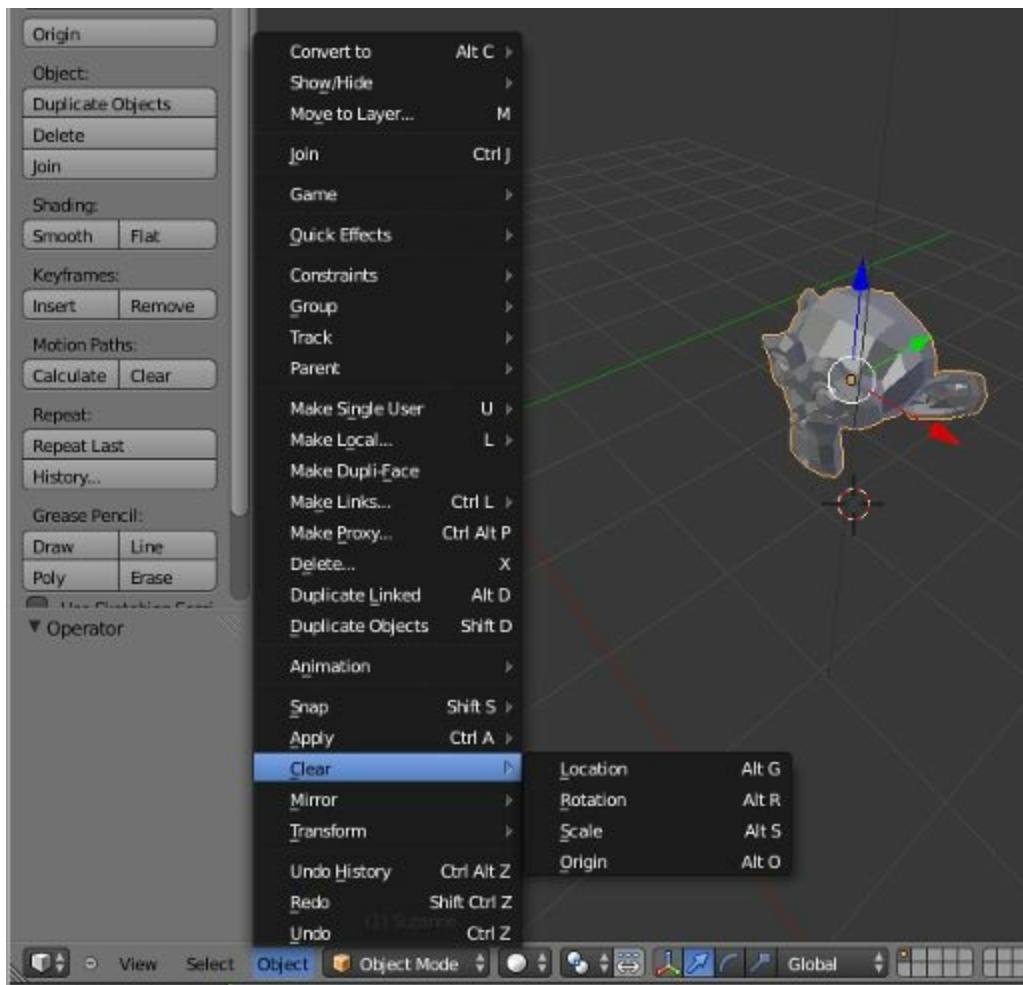
Part Three: Key-Framed Animation

1. Creating a key-framed animation in Blender
2. Importing the animation using AssImp, and re-creating our own data structure
3. Building an interpolating animation function
4. Playing the animation in OpenGL

Part One: Bones

Creating Bones in Blender

I'm going to create a new mesh in Blender. You can use any 3d modelling software. Blender has a built-in monkey-head mesh which will be interesting enough to get us started. The actual interface in Blender is kind of horrible, so I'm going to go over that because it would be a shame to get stuck at this point. I'm not an expert at Blender, and there are probably nicer ways to do some of these things, but this will at least get you started. If you're comfortable with this process, then by all means skip ahead to the next part.

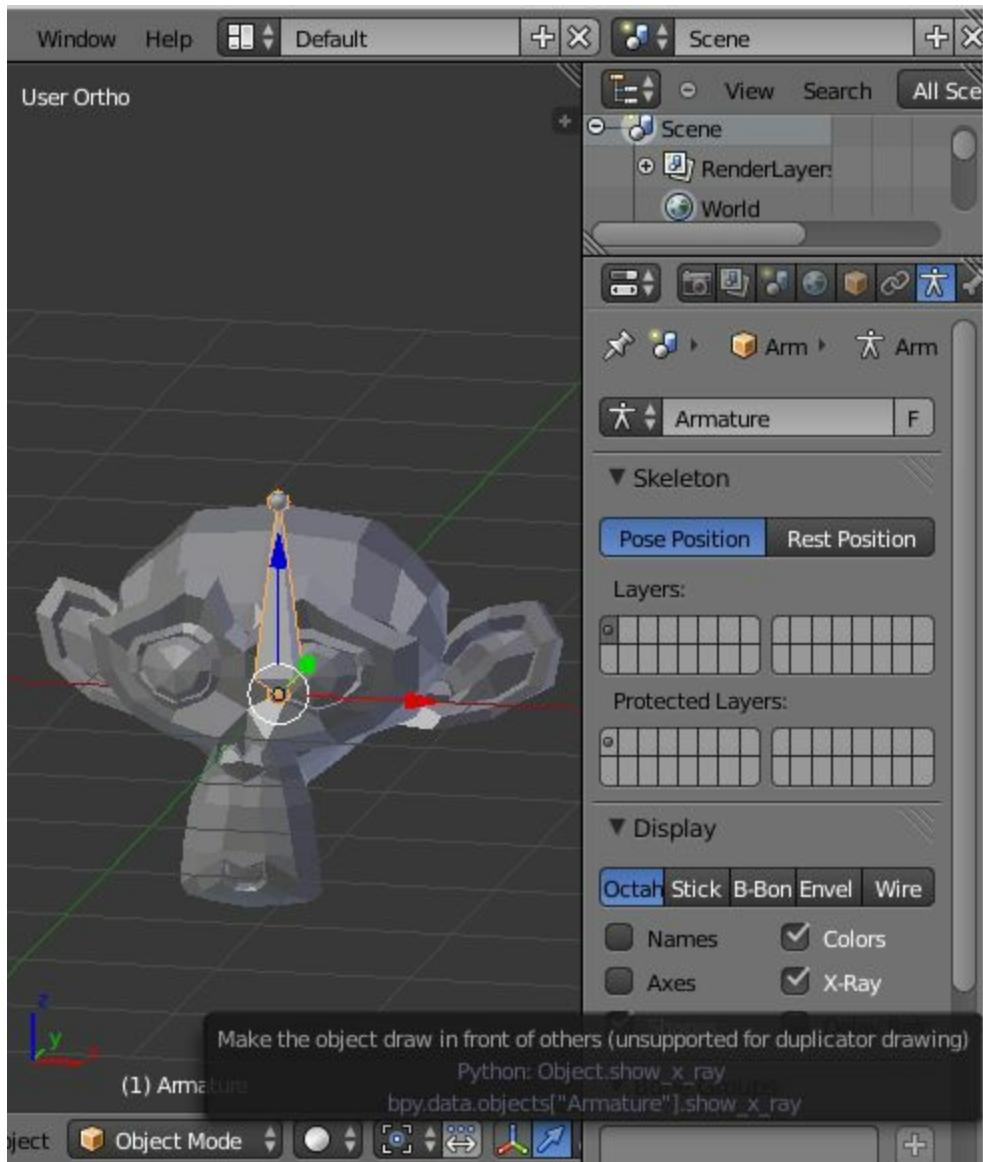


I added my monkey mesh. Blender has a nasty habit of applying orientation, and location offsets to the

high-level "object". These don't get exported, so I clear them all in Object Mode, then go into Edit Mode and rotate the mesh the right way up at the vertex level.

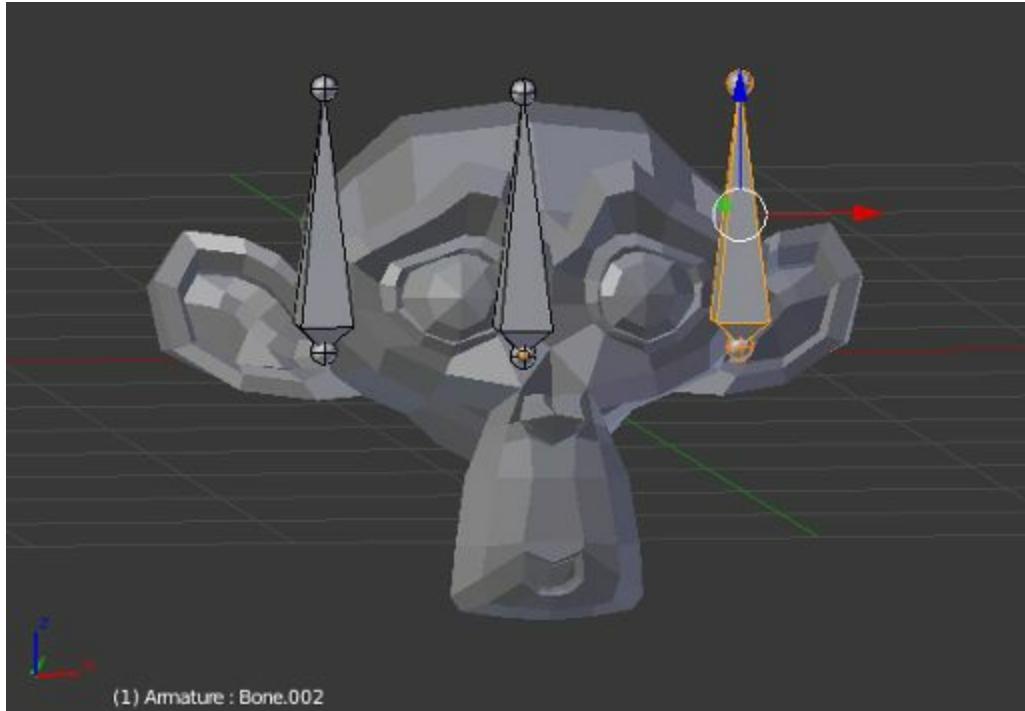
First thing to do once you've added the Suzanne mesh - clear the origin, location, and orientation of the object. Blender often applies an orientation to the high-level "object", so that it looks the right-way up in Blender, but this is not going to be exported with your vertex data, so you'll end up with an upside-down, somewhat offset animated mesh, which is really annoying. You can find the clear menu in Object Mode under Object → Clear. Clear all of these things, just in case. If your mesh is upside-down now, go into Edit Mode and correct the orientation there.

Okay, let's add a skeleton. Go back to Object Mode, and do Add → Armature → Single Bone. Blender calls skeletons "armatures", which are a kind of skeleton frame that clay sculptors use. Yes, Blender offsets and orientates these things at the "object" level as well, which will later surprise you with bizarre animations in the wrong direction. Right click on the bone when in Object Mode to select the whole armature. Clear all the different things, in the same menu as for the object.



The panel on the right hand side is set to Properties view mode. The button with the stick figure shows the armature tab, where there is an "X-Ray" option which shows our bone through the mesh.

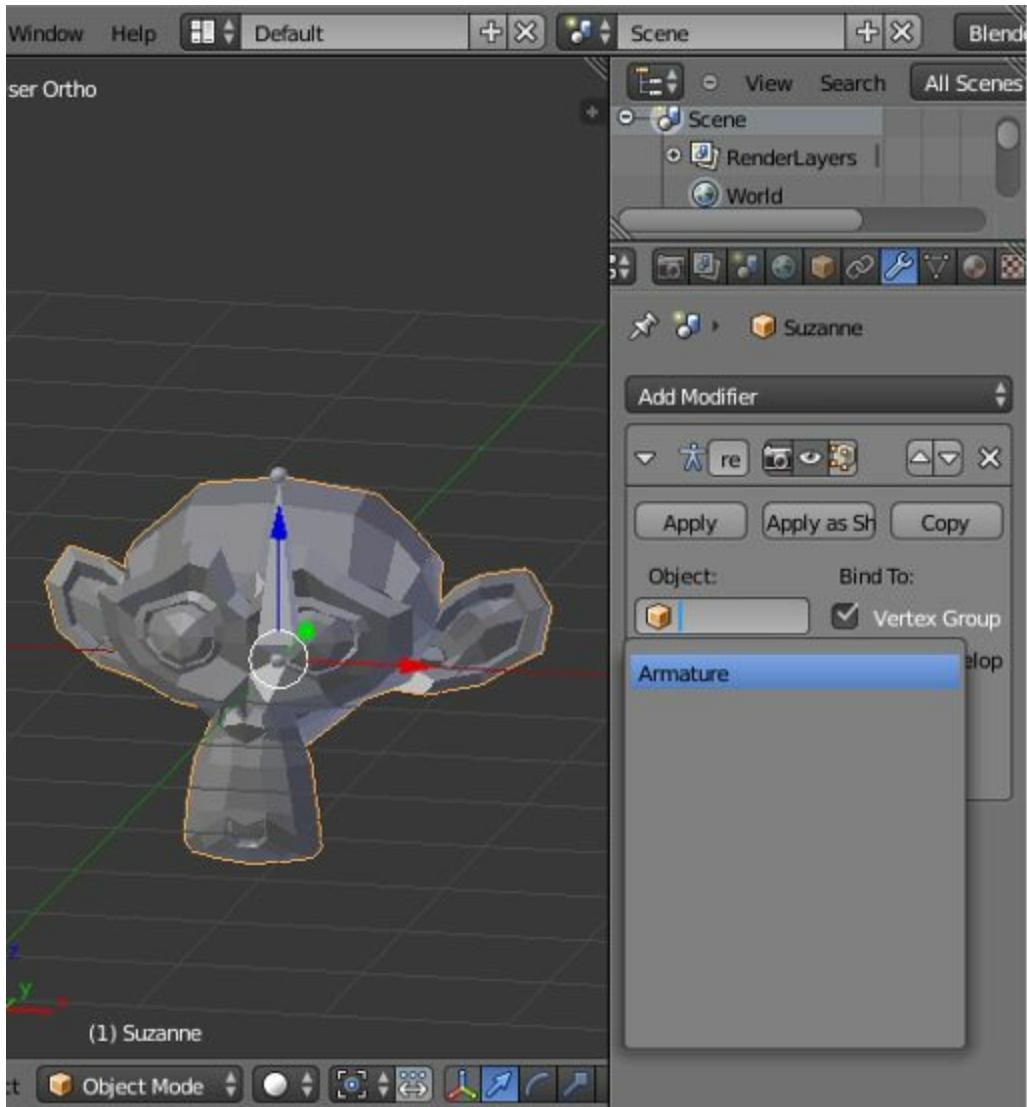
The bone is probably inside the monkey mesh, and hard to see now. Go to the Properties view - there is probably a panel open on the right hand side in Properties. This presents a range of buttons, one of which is a stick figure person for "armature". Click on this to get the armature tab. You should see a check-box called "X-Ray". Click and your bone should be visible.



In Edit Mode we can add additional bones to the skeleton by duplicating the first one. I put the new bones to either side of the head, so that they will be hinges that let me wiggle the ears.

Let's add two more bones to the armature. Select the armature, and go into Edit Mode. I want to have 3 bones in my armature; one in the middle of the head, and one for each ear. Duplicate our bone (`Shift+D`) and move the new bone to roughly where each ear joins to the head. I want to make an animation where we can rotate the ears, so put the bone at a location that would be a good hinge or pivot point. You can repeat this for a third bone. When moving the bones, select the whole bone by right-clicking in the middle of it. If you move just one of the ends then it might rotate in a strange way, and we don't want that. I think that just the bottom end of the blender bone is the actual bone position, and the top indicates an orientation offset. In any case - all your bones should be "pointing" exactly the same way.

Add the Armature as An Object Modifier



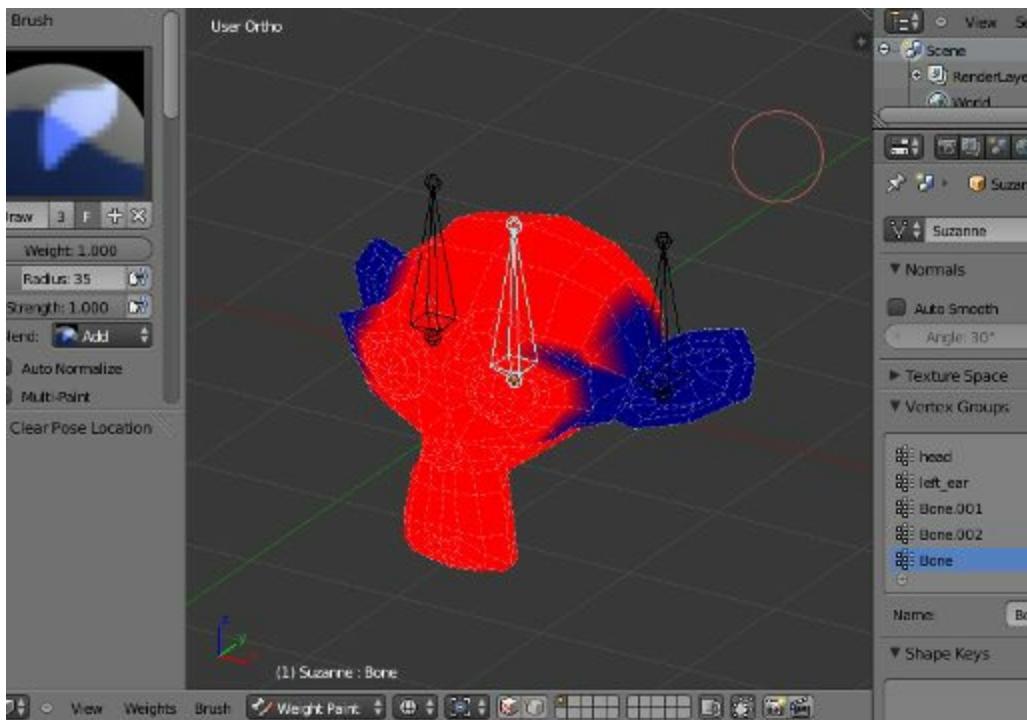
Here we tell our monkey object to use an armature modifier. The only thing to do here is make sure that our armature is selected in the "Object" field. Don't click "Apply" or anything else.

The armature isn't actually tied to our mesh yet, it's a completely unrelated object. To make the connection, we need to select the monkey object (in Object Mode). Go to the Properties view again, and click on the button that looks like a spanner/wrench. On the "Add Modifier" drop-down, choose "Armature". Now we need to tell it the name of the armature to use. Our armature should just be called the default "Armature". Click on the "Object" field and select our armature. That's it - do not click "Apply" or anything, just leave it like that.

Weight Painting

Now we can do **weight painting**, which means, telling each vertex which bone(s) should animate it. If we hadn't added the modifier, as in the previous step, then our weight painting would look okay, but not actually allow us to animate the mesh. I forget to do this all the time, and it's rather annoying.

Select the monkey object in Object Mode. This lets you get into Weight Paint mode. You should be able to select each bone separately by right-clicking on it. When you paint vertex weights, you're painting just for the selected bone, and not for the others. Blue means "not weighted to the bone", and red means "completely weighted to the bone". It's possible to have vertices partially weighted to several bones, but we don't want to deal with that just yet.



Weight painting in Blender. The red areas are vertices that are completely "weighted" to the central bone. This is really tricky to do.

We want each ear to be completely red when its bone is selected, and the rest of the head blue. Paint with the left mouse button. You'll find this to be quite tricky around the joins, so consider using the **selection masking** tools on the tool bar. You can read about that in the [Blender wiki](#). Once that's done, you can actually move the ear away from the head by selecting the bone and pressing **g**. This will make it easier to see if you forgot any vertices. Finally paint the head, and move it around to be sure that no vertices were left un-

painted. You can snap back to the original positions with `ALT+g`. You can actually get a good idea for how the mesh will animate by moving and rotating the bones now.

Export to a Suitable File Format

Finally, we have a mesh with bones in it. You can save your work, and export to a format that supports bones. The very latest version of Blender has an exporter for COLLADA .dae, which is not an ideal file format, but has animation support, and is fairly broadly supported by different software, which is what we want to begin with.

The current COLLADA export for Blender **does not have an option to correct the coordinate system to "Y is up"**, which can be a bit confusing if we are used to "Y" being up in OpenGL. It's a good idea to go into Edit Mode (not Object Mode) for both the armature and the mesh, and manually rotate them around so that "up" is now pointing up the Y-axis. If you hold down the CTRL key when rotating in Blender it will do so in fixed increments. We don't want to have to waste computation correcting this when we are animating. Alternatively, you can just change your OpenGL camera around so that it uses the same coordinate system as Blender - up to you.

Importing Bones with AssImp

Okay, let's firstly make sure that our mesh imports and displays without doing anything with bones. Pull up your AssImp-based mesh loader, change the file to our new .dae mesh, and make sure that you can see the monkey head.

In your code where you call your mesh loading function, let's add a variable to get the count of the number of bones, as well as an array of **[inverse] bone offset matrices**. These are used to rotate points around the bone during animation, but if we negate them they will also give us the offset position of each bone.

```
/* load the mesh using assimp */
GLuint monkey_vao;
mat4 monkey_bone_offset_matrices[MAX_BONES];
int monkey_point_count = 0;
int monkey_bone_count = 0;
assert (load_mesh (
    MESH_FILE,
    &monkey_vao,
    &monkey_point_count,
    monkey_bone_offset_matrices,
    &monkey_bone_count
));
printf ("monkey bone count %i\n", monkey_bone_count);
```

Of course, we have to add the new parameters to the function too:

```
/* load a mesh using the assimp library */
bool load_mesh (
    const char* file_name,
    GLuint* vao,
    int* point_count,
    mat4* bone_offset_mats,
    int* bone_count
) {
```

Inside your loader function, there are a number of if-statements checking for different attributes in the mesh. Let's add another one that checks for bones, and gets the offset matrix for each bone. We also record the name of each bone here, because AssImp uses the names as an identifier key between its different data structures. Let's get this working first, then come back later and get the bone weights per vertex:

```

/* extract bone weights */
if (mesh->HasBones ()) {
    *bone_count = (int)mesh->mNumBones;
    /* an array of bones names. max 256 bones, max name length 64 */
    char bone_names[256][64];

    for (int b_i = 0; b_i < *bone_count; b_i++) {
        const aiBone* bone = mesh->mBones[b_i];
        strcpy (bone_names[b_i], bone->mName.data);
        printf ("bone_names[%i]=%s\n", b_i, bone_names[b_i]);
        bone_offset_mats[b_i] = convert_assimp_matrix (bone->mOffsetMatrix);

        /* get weights here later */

    } // endfor
} // endif

```

I wrote a little converter function to convert AssImp's matrices to my own format. You probably want to do something similar. Note that I only use the translation part of AssImp's matrices; I don't want any bones to be oriented differently by any small accident in Blender or the import process:

```

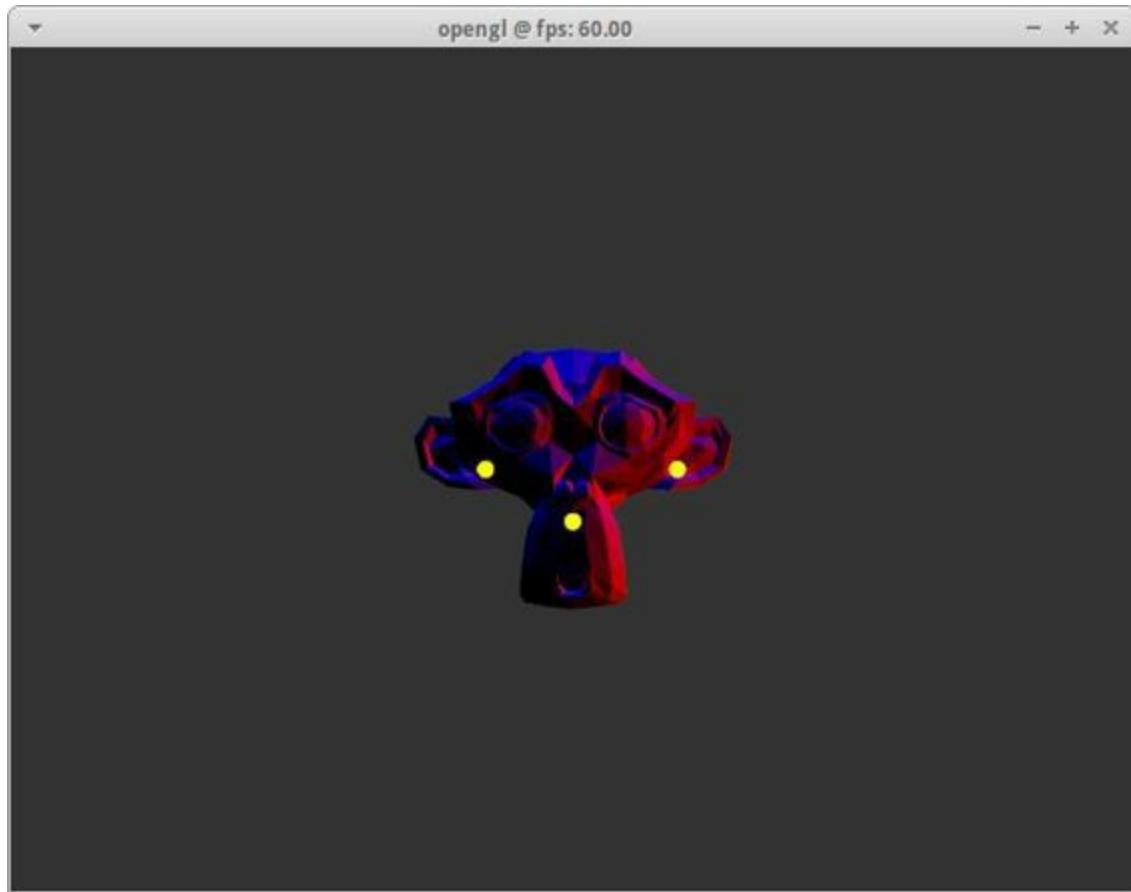
mat4 convert_assimp_matrix (aiMatrix4x4 m) {
    /* entered in columns! */
    return mat4 (
        1.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f, 0.0f,
        m.a4, m.b4, m.c4, m.d4
    );
}

```

At this point you probably want to compile, and run to make sure that the count and names of your bones are printing out correctly. I also like to print out the contents of my matrices, to make sure that they look sensible, and are in the correct column-major order.

Visualising Bone Positions

It's always a good idea to spend time visualising as much data as possible behind your algorithms. You might like to display the positions of your bones. It's easy to get these slightly wrong, or on the wrong axis due to an issue in Blender, so it's good to be able to check visually, rather than scratch your head later wondering why unusual animations are happening.



I created an array of 3d points by extracting the -x,-y,-z translation components of my 3 bone matrices and putting them in an array. I used point sprite rendering (see Particles tutorial) to render these. The first thing to check is that your bones are on the same axis as the mesh and not 90 degrees out - easy to make an orientation mistake in Blender. It would be much harder to spot this without drawing the bones on and visualising them.

One way to do this is with point sprite rendering (like we did in the Particles tutorial). You can make an array of XYZ points from the negated 12th, 13th, and 14th (XYZ translation) components of your bone offset matrices. I made a shader that used the view and projection matrices, resized the points, and coloured them yellow. Remember to disable depth testing to "X-Ray" your bones.

Importing Bone IDs

When we animate our mesh, each vertex shader will have additional inputs:

1. an **uniform array of bone animation matrices** (one for each bone)

2. an input attribute **bone_ids[4]**, saying which bones affect the vertex
3. an input attribute **bone_weights[4]**, saying what factor each of the bone's matrices has on the vertex

The next variable to try is bone IDs. We are going to assume that each vertex is only affected by one bone, so we don't need an array of these per-vertex; just a single integer per vertex will do. This also means that our weight will always be 1.0, so we can ignore this variable completely for now. Let's create a pointer to an array of bone IDs that we will allocate. This can go in the mesh loader function, under the other ones:

```
...
GLfloat* points = NULL; // array of vertex points
GLfloat* normals = NULL; // array of vertex normals
GLfloat* texcoords = NULL; // array of texture coordinates
GLint* bone_ids = NULL; // array of bone IDs
...
```

And we can allocate it inside our if-statement from earlier:

```
/* extract bone weights */
if (mesh->HasBones ()) {
...
    bone_ids = (int*)malloc (*point_count * sizeof (int));
...
}
```

Now we can fill out the `/* get weights here later */` in our if-statement to work out a single bone ID for each vertex. Assimp stores an array of "weights" inside each bone, which actually contain a vertex number, and a weight factor:

```
/* get bone weights
we can just assume weight is always 1.0, because we are just using 1 bone
per vertex. but any bone that affects a vertex will be assigned as the
vertex' bone_id */
int num_weights = (int)bone->mNumWeights;
for (int w_i = 0; w_i < num_weights; w_i++) {
    aiVertexWeight weight = bone->mWeights[w_i];
    int vertex_id = (int)weight.mVertexId;
    // ignore weight if less than 0.5 factor
    if (weight.mWeight >= 0.5f) {
        bone_ids[vertex_id] = b_i;
    }
}
```

At the end of our mesh loader function we have several if-statements that take our imported data and make vertex buffers objects out of them. Let's add

one for bone IDs.

Integer Attributes

Now, OpenGL implementations are terrible at dealing with integer attributes, and will almost certainly corrupt them if you use `glVertexAttribPointer()` when you set up your vertex attribute object. To get around this you can try using the integer-only version of that function; `glVertexAttribIPointer()`. This worked for me, but if that still doesn't work for your version/implementation then you can just use floats, and cast then to integers inside the vertex shader. I did this:

```
if (mesh->HasBones ()) {
    GLuint vbo;
    glGenBuffers (1, &vbo);
    glBindBuffer (GL_ARRAY_BUFFER, vbo);
    glBufferData (
        GL_ARRAY_BUFFER,
        *point_count * sizeof (GLint),
        bone_ids,
        GL_STATIC_DRAW
    );
    glVertexAttribIPointer(3, 1, GL_INT, 0, NULL);
    glEnableVertexAttribArray (3);
    free (bone_ids);
}
```

Remember that it's just single values for bone ids at the moment, so the size parameter to `glBufferData` does not need a `*3` multiplier like the `vec3` buffers do. I also have a "1" in the `glVertexAttribPointer` 'location' parameter - don't forget about these little traps!

Visualise Bone IDs

Great, now this should be taken care of inside our mesh's VAO. We can visualise that our bone IDs were assigned to the correct vertices by adding a bone ID input to our vertex shader, and using that to determine a colour for the fragment shader:

```
#version 400

layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;
```

```

layout(location = 2) in vec2 texture_coord;
layout(location = 3) in int bone_id;

uniform mat4 model, view, proj;

out vec3 colour;

void main() {
    colour = vec3 (0.0, 0.0, 0.0);
    if (bone_id == 0) {
        colour.r = 1.0;
    } else if (bone_id == 1) {
        colour.g = 1.0;
    } else if (bone_id == 2) {
        colour.b = 1.0;
    }

    gl_Position = proj * view * model * vec4 (vertex_position, 1.0);
}

```

And the fragment shader is fairly trivial still:

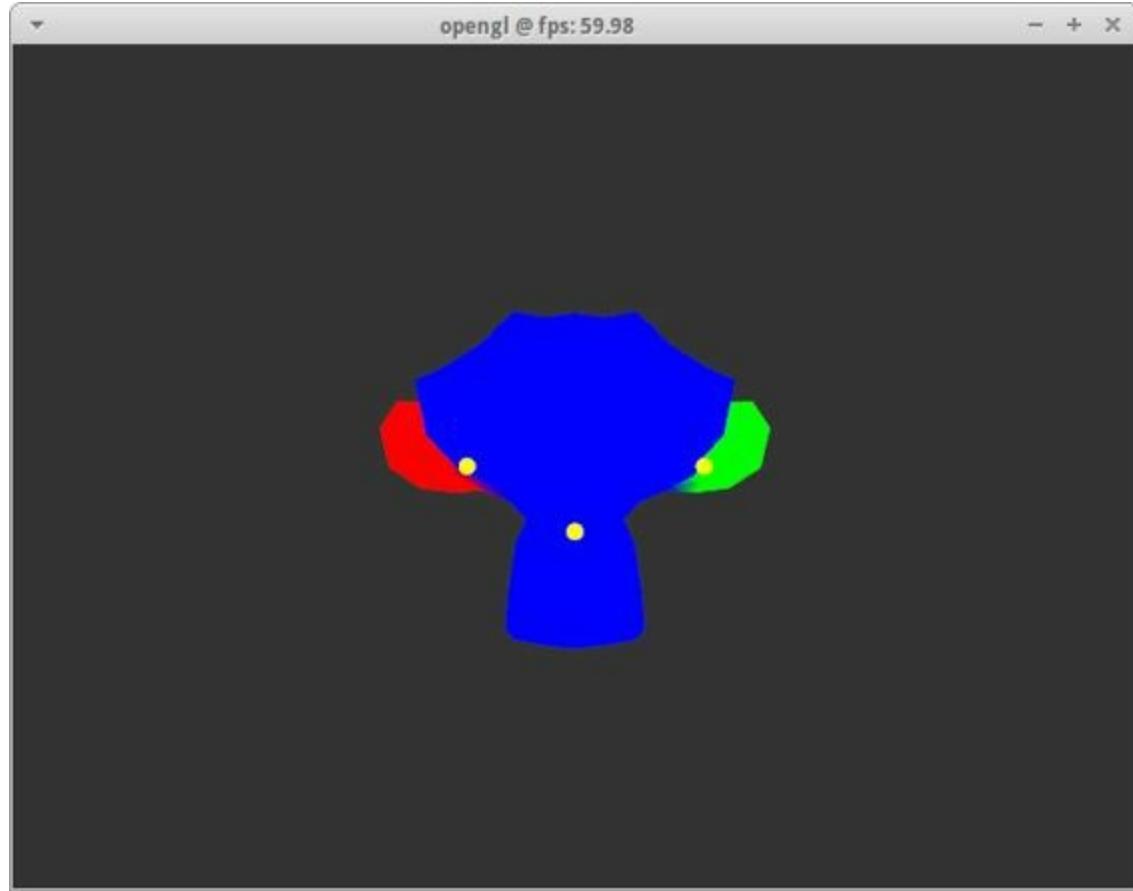
```

#version 400

in vec3 colour;
out vec4 frag_colour;

void main() {
    frag_colour = vec4 (colour, 1.0);
}

```



Using the bone IDs to determine the colour helps us check that everything loaded correctly, and will make any little weight-painting mistakes more obvious.

Bone Transformation Matrices

Animation deforms vertices, so we add an array of bone transformation matrices to the vertex shader:

```
#version 400

layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;
layout(location = 2) in vec2 texture_coord;
layout(location = 3) in int bone_id;

uniform mat4 model, view, proj;
// a deformation matrix for each bone:
uniform mat4 bone_matrices[32];

out vec3 colour;

void main() {
    colour = vec3 (0.0, 0.0, 0.0);
    if (bone_id == 0) {
```

```

        colour.r = 1.0;
    } else if (bone_id == 1) {
        colour.g = 1.0;
    } else if (bone_id == 2) {
        colour.b = 1.0;
    }

    gl_Position = proj * view * model * bone_matrices[bone_id] *
        vec4 (vertex_position, 1.0);
}

```

We also need to get the uniform locations for the bone matrices, and set these to a sensible default (like the identity matrix). We can't assume that the locations for each matrix in the array are stored contiguously - each implementation of OpenGL awards the locations differently (I recently had some problems with this), so we need to get them all.

```

int bone_matrices_locations[MAX_BONES];
float identity[] = {
    1.0f, 0.0f, 0.0f, 0.0f, // first column
    0.0f, 1.0f, 0.0f, 0.0f, // second column
    0.0f, 0.0f, 1.0f, 0.0f, // third column
    0.0f, 0.0f, 0.0f, 1.0f // fourth column
};
// reset all the bone matrices
glUseProgram (shader_programme);
char name[64];
for (int i = 0; i < MAX_BONES; i++) {
    sprintf (name, "bone_matrices[%i]", i);
    bone_matrices_locations[i] = glGetUniformLocation (shader_programme, name);
    glUniformMatrix4fv (bone_matrices_locations[i], 1, GL_FALSE, identity);
}

```

Compile and run now - it should display your original mesh as before. Next we can add a couple of buttons to let us wiggle the ears. I wrote a maths function to generate a rotation matrix for me and store it in a variable called `left_ear_mat`:

```

float theta = 0.0f;
float rot_speed = 50.0f; // 50 radians per second
mat4 left_ear_mat = identity_mat4 (); // matrix for the left ear
...
if (glfwGetKey (g_window, 'Z')) {
    theta += rot_speed * elapsed_seconds;
    left_ear_mat = rotate_z_deg (identity_mat4 (), theta);
    glUseProgram (shader_programme);
    glUniformMatrix4fv (bone_matrices_locations[0], 1, GL_FALSE, left_ear_mat.m);
}
if (glfwGetKey (g_window, 'X')) {
    theta -= rot_speed * elapsed_seconds;
    left_ear_mat = rotate_z_deg (identity_mat4 (), theta);
    glUseProgram (shader_programme);
    glUniformMatrix4fv (bone_matrices_locations[0], 1, GL_FALSE, left_ear_mat.m);
}

```

Now something should happen. You might have noticed that the ears rotate around the origin of the mesh, and not around their bone as a pivot point. To fix the pivot point problem, so that it rotates around the bone, this is where we use our offset matrices. We first multiply by the bone's offset matrix, which moves the ear so that the (0,0,0) origin is now the bone position. We then apply the rotation, which rotates around the ear, and finally multiply by the inverse of the offset matrix, which moves the ear back to the side of the head:

```
left_ear_mat = inverse (monkey_bone_offset_matrices[0]) *  
    rotate_z_deg (identity_mat4 (), theta) *  
    monkey_bone_offset_matrices[0];
```

So, from the right-hand side we first apply the bone offset matrix for bone number 0, which sets the bone position as the origin (and pivot point for rotation), next apply our actual animation - a rotation around Z, and finally apply the inverse of the offset matrix, which moves the origin back to the original position. Note that when we load key-framed animations, in the final part of this tutorial, loaded animations will already incorporate the bit that does the final `inverse()` transform, so we'll just need the first offset matrix.

Discussion

Now we have control of the low-level basics of skinned animation, which is going to make the rest of the techniques much easier to deal with. To summarise the key points:

- There is some tricky business in modelling software to set up bones
- We give each vertex one or more **bone IDs** which becomes a vertex buffer object.
- We can add a **weights** buffer too, to allow more than one bone ID per vertex.
- Bone IDs are therefore also a **vertex shader input variable** (attribute) which index...
- An (uniform variable) **array of animation matrices**; one matrix for each bone in the mesh
- We just add the animation matrix to the **transformation pipeline** before the model, view, and projection matrices
- We work out each **animation matrix in C code**
- We multiply this by the **bone offset matrix** and its **inverse** to use the bone as the pivot point for animation

I know this is a really long tutorial, but I really wanted to make this topic clearer than what is already out there, and I don't think that it's possible to do that quickly.

Common Problems

I think that I've already addressed all the common issues that people have in the text. The biggest confusions are:

- Not clearing rotations and offsets at the "object" level in Blender
- Armature not properly tied to an object in Blender (Add Modifier) before weight-painting
- Vertices exist without weights - fix in Blender, or set a default Bone ID to 0
- Multiple "objects" are not exported as a single mesh. Try joining objects in Blender
- A mesh exporter plug-in does not work properly, does not comply strictly with the file format, or does not have all the features expected.
- Mesh and animation face in different directions - you can fix this with an extra rotation matrix before you animate, but it's better to correct this in modelling software.
- Displaying bone positions is wrong - remember to use the inverse (negated version) of the bone offset matrix to get the offset positions.
- Rotations are not around the bone as a pivot point - use the offset matrix and its inverse, as in the last code sample, above.
- Animations are still wrong - make sure that you are updating the correct uniform matrix i.e. is the uniform location for the correct bone?
- Colouring my mesh by bone ID shows some incorrect regions - we chose our bone ID as the last bone with a weight over 0.5. Try adjusting this threshold.
- My rotations/translations are still not the same direction as they were in my modelling software - perhaps your individual bones were rotated in the modeller? I keep all my bones un-rotated in Blender and copy only the translation part into my matrices. You might consider copying the entire matrix instead.
- I can only index my first animation mesh! - Sounds like your bone ID integers are not working properly. This will be a problem on older versions of OpenGL. Change them to `float` instead, in both the C code, and in the vertex shader input. You can cast them to `int` in the vertex

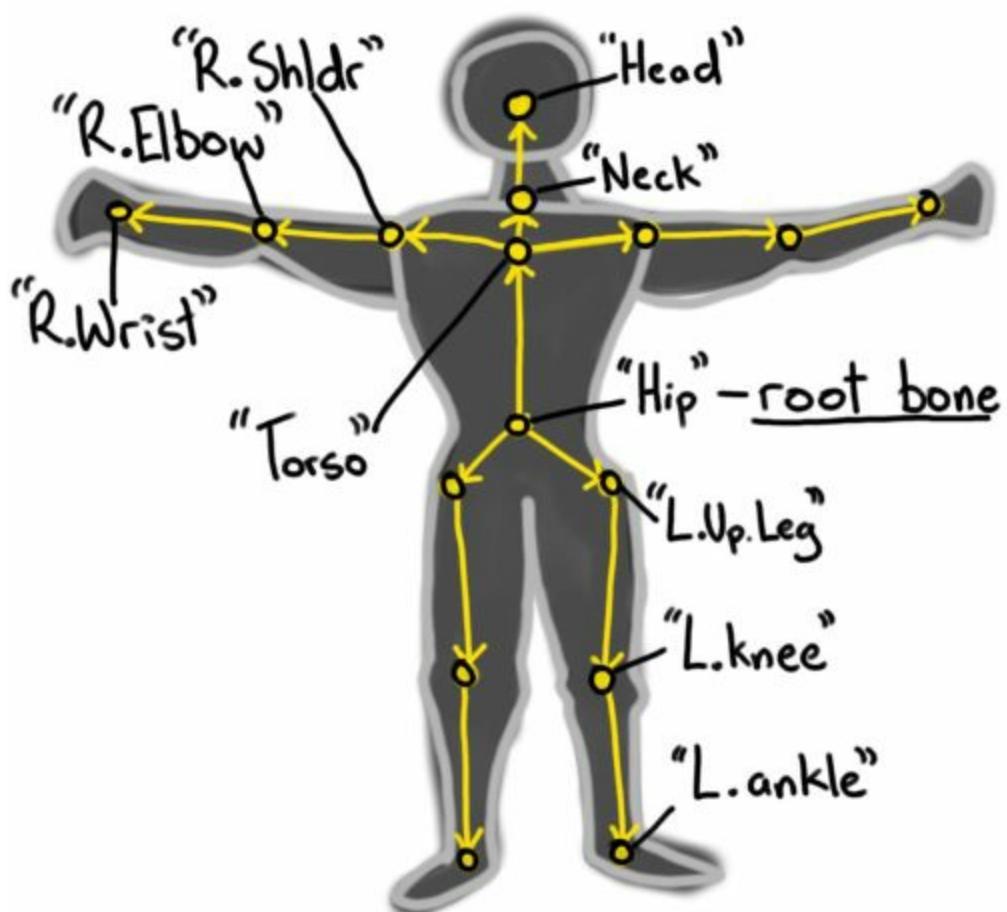
shader when you index the array.

Next

Next we can build concepts onto this; a **skeleton hierarchy**, so that moving the head bone will also move the ears, and **key-framed animations**, which we will create a function for, to smoothly **interpolate** between. These concepts are going to require some more 3d modeller work, which we will look at again, as well as some additional data structures and matrix code in C.

Part Two: Skeletons

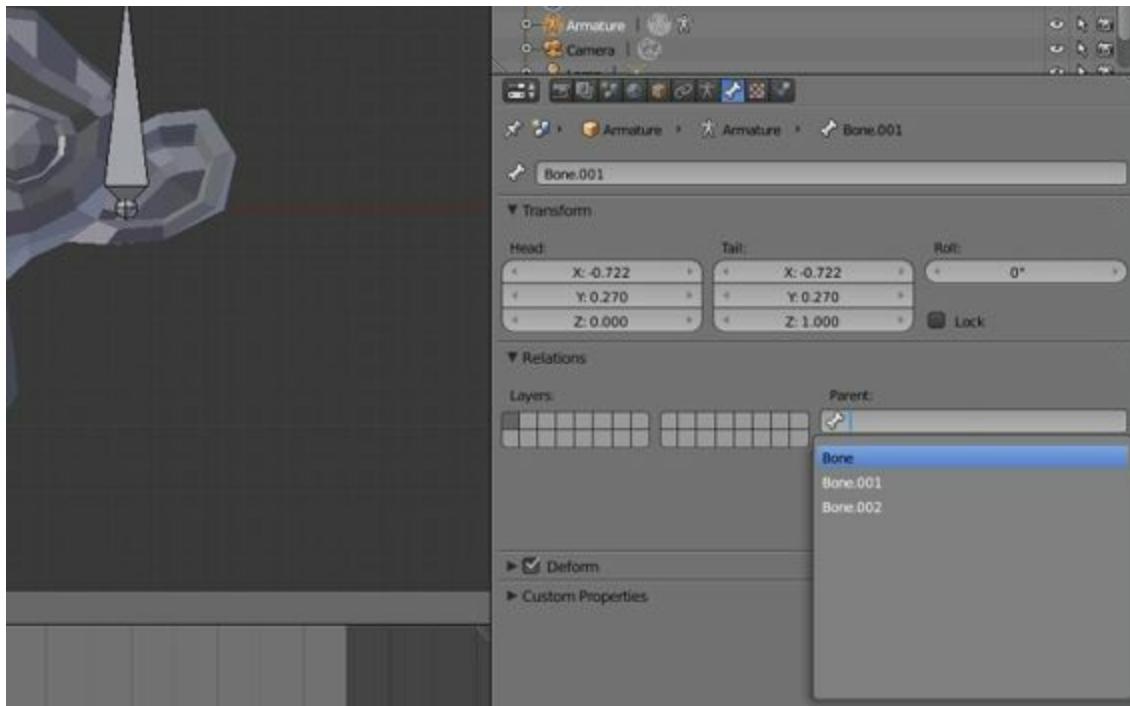
Creating Skeletons in Blender



A typical skeleton for a human mesh has a bone at each major joint in the body so that we can rotate the arms, the hips, etc. The circled dots are bones, and the arrows point from parent to child, because this is the direction that we will process the animation. When we rotate an arm at the shoulders, we also expect the lower part of the arm to move, so we make the elbow bone a child of the shoulder bone. The elbow then applies the shoulder's animation matrix to its own animation matrix. More often than not the "root" bone, where we start the animation, is at the hips of the character.

In the previous tutorial we created bones for our mesh by duplicating the first bone. Now we want to say that the ear bones are connected to the head bone.

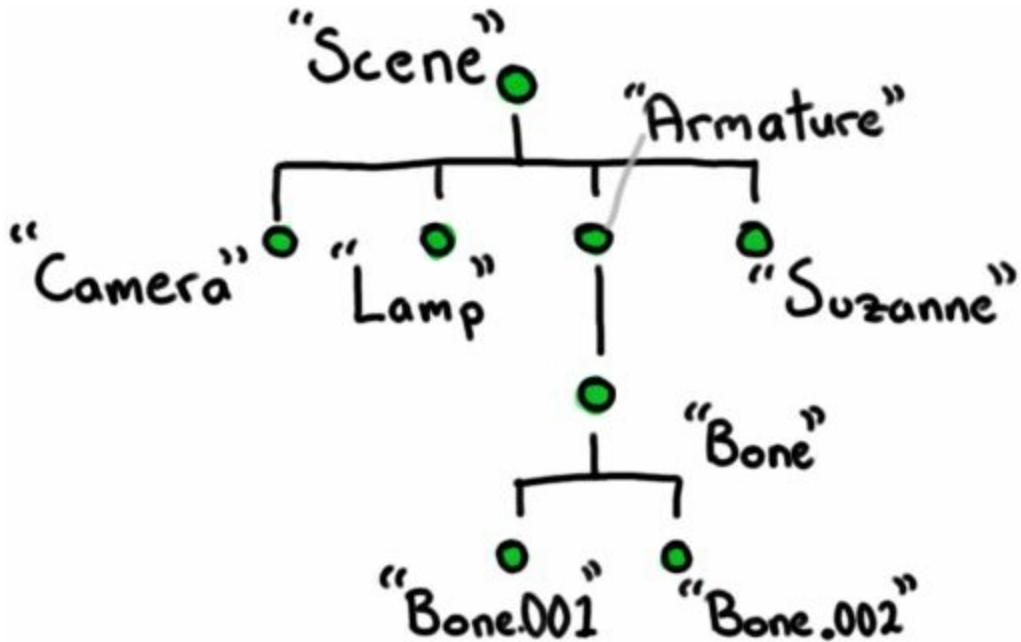
In skinning we usually just have a tree hierarchy, that is, each bone can be assigned other bones to be its "children", so that when we transform the parent, its matrix is then also applied to the other bones. This is very commonly used for animation of human and animal characters. A typical human "skeleton" is given in the diagram above. Blender has a little dialogue box to specify a parent for a bone.



In the Properties view click on the bone icon and find the "parent" drop-down list. You should see a dotted line connecting from the selected bone to the parent that you chose.

Having done that, export the mesh again. Now we are going to look at our mesh loader code again to extract the information about parentage. Unfortunately this gets tricky with AssImp, because it's stored in a separate data structure. Remember where we got the names of the bones out last time? We're going to need that array now.

Getting the Skeleton Out of AssImp



AssImp's tree of "nodes" for my exported mesh. Note that the 3 bones are in there, with the correct names and parent-child relationships, but so is a lot of other junk that we don't need. "Suzanne" is the name that Blender gave to my mesh.

AssImp doesn't just give us the skeleton, we have to extract it from a tree of "nodes", where everything in the file appears as a node, starting with a root node for the entire scene. You'll notice that the names and arrangement of these nodes corresponds to what is set up in Blender, so you may have different names for things. The bones also appear as nodes, but we have to work our way down to them. My approach is to do this, find the nodes that correspond, and recreate my own tree of nodes outside of AssImp. A diagram of the tree you will expect to get for the monkey mesh is given above. The only link that we have to our array of bones is the name string. We are going to traverse our way down Blender's tree, starting with the scene root node, and do string comparisons to find our bones. Now, you could very well just use AssImp's own tree when you're animating, but it's going to be a little bit slower, which we want to avoid, and it also means that you depend on AssImp during run-time, which we'll talk about again at the end of these tutorials.

Note that it's possible to put a bone in our skeleton in Blender that isn't weighted to any vertices, but still affects the animation because its animation is inherited by children which are weighted to vertices. That's fine - we shouldn't have to do anything special to handle this.

We may also have nodes left over in our tree that are not bones in the skeleton. To resolve the ambiguity, we will firstly refer to the nodes in our skeleton hierarchy as "nodes" and not "bones". We will indicate in each node which bone ID (index number in our array) it affects, if any. Here is the data structure that I will use for a node in the skeleton:

```
struct Skeleton_Node;
struct Skeleton_Node {
    Skeleton_Node* children[MAX_BONES];
    char name[64];
    int num_children;
    int bone_index;
};
```

I declare the `struct Skeleton_Node;` first so that it can have pointers to a number of children of its own type. Pretty simple - all the node needs to do is say:

- How many child nodes it has
- The memory address of each child node
- The index to a bone in our array, or -1 if it doesn't directly correspond to a weighted bone.

I also store its name there - I won't use that here, but it might be a good idea to give us some flexibility for more complicated animation programmes. You might have guessed already that we are going to write a recursive function to construct our skeleton tree from AssImp's node tree:

```
bool import_skeleton_node (
    aiNode* assimp_node,
    Skeleton_Node** skeleton_node,
    int bone_count,
    char bone_names[][64]
);
```

The function will work like this; we give it AssImp's root node (the "Scene" node), a NULL pointer to our skeleton's root node - the function will allocate the memory internally, and the total count of bones in the mesh, and a list of all the names of bones. This is actually a pointer to a pointer so that I can

allocate memory for it inside the function; a C programming trick. You might prefer to use the C++ reference-to-pointer version `skeleton_Node*&` instead. The last two parameters we already have from the last exercise.

I'm also going to add a parameter to the mesh loading function, so that it will pass back a pointer to the root node in our skeleton. I used the C trick of a pointer-to-a-pointer here because we will allocate its memory in the function. You may prefer to make this a global variable if you are only importing one mesh:

```
bool load_mesh (
    const char* file_name,
    GLuint* vao,
    int* point_count,
    mat4* bone_offset_mats,
    int* bone_count,
    Skeleton_Node** root_node
) {
    ...
}
```

When you call the `load_mesh` function you will first create a `Skeleton_Node* root = NULL;` and pass the address of it as a parameter to the function.

Okay, let's modify our mesh-loading function, then we will write the definition of the `import_skeleton_node` function. In our `load_mesh()` code, find where we were parsing bones inside `if (mesh->HasBones ()) {`. If you remember, we had a for-loop that created an array of `bone_ids`. After that for-loop, but still within the `HasBones` section we can get the root node of the mesh. All successfully imported meshes have a root node, even if they don't have a skeleton.

```
...
if (mesh->HasBones ()) {
    ...
    aiNode* assimp_node = scene->mRootNode;
    if (!import_skeleton_node (
        assimp_node,
        root_node,
        *bone_count,
        bone_names
    )) {
        printf ("WARNING: no skeleton found in mesh\n");
    } // endif
}
```

```
} // endif hasbones
```

You can see that the if-statement for `HasBones` closes at the end of the above fragment. We pull out the root node from AssImp - this will most likely be called one that Blender has called "Scene". Right, now let's start making the function:

```
bool import_skeleton_node (
    aiNode* assimp_node,
    Skeleton_Node** skeleton_node,
    int bone_count,
    char bone_names[][64]
) {
    // allocate memory for node
    Skeleton_Node* temp = (Skeleton_Node*)malloc (sizeof (Skeleton_Node));

    // get node properties out of AssImp
    strcpy (temp->name, assimp_node->mName.C_Str ());
    printf ("node name = %s\n", temp->name);
    temp->num_children = 0;
    printf ("node has %i children\n", (int)assimp_node->mNumChildren);
    temp->bone_index = -1;
    for (int i = 0; i < MAX_BONES; i++) {
        temp->children[i] = NULL;
    }
    ...
}
```

The first thing that I do is create a pointer to one of our skeleton nodes and allocate memory to it. I call this `temp`. It's easier to work with a pointer than with a pointer-to-a-pointer - that's why I make a new one here. I copy the name out of the AssImp node, and initialise the other variables of the node to default values. Now that I have the name of the node, I can search through our array of bone names to see if there is a match. Remember that we collected these in the last tutorial. I passed this array of names in as a function parameter.

```
...
// look for matching bone name
bool has_bone = false;
for (int i = 0; i < bone_count; i++) {
    if (strcmp (bone_names[i], temp->name) == 0) {
        printf ("node uses bone %i\n", i);
        temp->bone_index = i;
        has_bone = true;
        break;
    }
}
if (!has_bone) {
    printf ("no bone found for node\n");
}
...
```

Now we have just recorded the index number, if we found a bone with the same name as the node. Now we can recurse the function and repeat the exercise for the child nodes. If the function returns `false` then the node was something useless like a camera, and no memory was allocated for it, otherwise we increase the count of children that are stored in our node, and point its matching child pointer to the new node.

```
...
    bool has_useful_child = false;
    for (int i = 0; i < (int)assimp_node->mNumChildren; i++) {
        if (import_skeleton_node (
            assimp_node->mChildren[i],
            &temp->children[temp->num_children],
            bone_count,
            bone_names
        )) {
            has_useful_child = true;
            temp->num_children++;
        } else {
            printf ("useless child culled\n");
        }
    }
    if (has_useful_child || has_bone) {
        // point parameter to our allocated node
        *skeleton_node = temp;
        return true;
    }
    // no bone or good children - cull self
    free (temp);
    temp = NULL;
    return false;
}
```

Finally, I do a check - if a child was successfully created, we keep our node. If our node is childless, but has a matching bone, then we keep our node. If the node is to be kept, we point our original `skeleton_Node` pointer to our allocated node memory (the variable called `temp`) and return `true`. Otherwise we delete the memory that we allocated and return `false`. If you compile and run now you should get some printouts telling you which nodes are allocated to our tree, and which are discarded. We now have a tree of nodes that has an index directly to our array of matrices, rather than just a name string - efficiency. Next we will look at animating using the skeleton.

Animating with the Skeleton

Once again, we are going to write a recursive function. This time we will calculate an animation matrix for each node, starting with the root. The children will all get a copy of their parent node's matrix, and apply this at the end of their own transformation.

```
void skeleton_animate (
    Skeleton_Node* node,
    mat4 parent_mat,
    mat4* bone_offset_mats,
    mat4* bone_animation_mats
);
```

We will give the function the root node, and an identity (do-nothing) matrix to begin with. `bone_offset_mats` is where we will give it our array of bone offset matrices, which we collected in the previous tutorial. Finally, we will give it an array which will hold the final animation matrix for each of our 3 bones. The function will store matrices in this array. After the function has finished we can do one uniform update that will update all of the matrices for our bones to the vertex shader.

Somewhere at the top of your code, before you enter the main loop, we will create some new variables that will store animation information for us:

```
mat4 monkey_bone_animation_mats[MAX_BONES];
float y = 0.0; // position of head
```

It's a good idea to run a for-loop and initialise all the matrices to the identity matrix. The `y` variable I'm going to use to keep track of the vertical position of the monkey's head.

Keyboard Controls

Next time we will look at loading key-framed animations for each node, but this time we will still use keyboard controls to animate. This is a bit unusual, so I'm going to hack in a global array of matrices that we can modify from the keyboard, and read inside the animation function. You can put this at the

top of your code:

```
mat4 g_local_anims[MAX_BONES];
```

We also need to modify our keyboard controls. We will no longer send uniform updates directly after pressing the keys, but just modify the global array that we made. Remember that this code was at the end of our main update loop:

```
bool monkey_moved = false;
if (glfwGetKey (g_window, 'Z')) {
    theta += rot_speed * elapsed_seconds;
    g_local_anims[0] = rotate_z_deg (identity_mat4 (), theta);
    g_local_anims[1] = rotate_z_deg (identity_mat4 (), -theta);
    monkey_moved = true;
}
if (glfwGetKey (g_window, 'X')) {
    theta -= rot_speed * elapsed_seconds;
    g_local_anims[0] = rotate_z_deg (identity_mat4 (), theta);
    g_local_anims[1] = rotate_z_deg (identity_mat4 (), -theta);
    monkey_moved = true;
}
if (glfwGetKey (g_window, 'C')) {
    y -= 0.5f * elapsed_seconds;
    g_local_anims[2] = translate (identity_mat4 (), vec3 (0.0f, y, 0.0f));
    monkey_moved = true;
}
if (glfwGetKey (g_window, 'V')) {
    y += 0.5f * elapsed_seconds;
    g_local_anims[2] = translate (identity_mat4 (), vec3 (0.0f, y, 0.0f));
    monkey_moved = true;
}
if (monkey_moved) {
    skeleton_animate (
        monkey_root_node,
        identity_mat4 (),
        monkey_bone_offset_matrices,
        monkey_bone_animation_mats
    );
    glUseProgram (shader_programme);
    glUniformMatrix4fv (
        bone_matrices_locations[0],
        monkey_bone_count,
        GL_FALSE,
        monkey_bone_animation_mats[0].m
    );
}
```

The main difference here is that we are setting our global array. I want Z and X to rotate both of the ears up and down so they affect index number 0 and 1 - remember that these were the bone IDs of our ears. C and V will move bone ID number 2 vertically up and down.

Right after the keyboard modifiers we have an if-statement. If any of the keys were pressed, then we call the recursive function to animate. When it finishes I enable our skinning shader programme, and give it the entire array of animation matrices that have been set by the recursive function.

Recursive Animation Function

This is quite a short function, but the ordering of matrices is fairly tricky. The first thing I do is validate the skeleton node - better to crash with a warning than to have to compile in debug mode and launch a back-trace to find it later.

```
void skeleton_animate (
    Skeleton_Node* node,
    mat4 parent_mat,
    mat4* bone_offset_mats,
    mat4* bone_animation_mats
) {
    assert (node);

    /* the animation of a node after inheriting its parent's animation */
    mat4 our_mat = identity_mat4 ();
    /* the animation for a particular bone at this time */
    mat4 local_anim = identity_mat4 ();
    ...
}
```

I create 3 matrices; `our_mat` is going to be the base animation that we calculate for the node, and will be inherited by its children. `local_anim` will be either the identity (do nothing), or if there is a valid bone ID for the node, then it will be the matrix that we set with keyboard controls. We will use this in calculating our animation, but in the full skinned system this would come from a key-framed animation system.

```
... // if node has a bone...
int bone_i = node->bone_index;
if (bone_i > -1) {
    // ... then get offset matrices
    mat4 bone_offset = bone_offset_mats[bone_i];
    mat4 inv_bone_offset = inverse (bone_offset);
    // ... at the moment get the per-bone animation from keyboard input
    local_anim = g_local_anims[bone_i];

    our_mat = parent_mat * inv_bone_offset * local_anim * bone_offset;
    bone_animation_mats[bone_i] = our_mat;
} else {
    our_mat = parent_mat;
}
```

...

The real tricky bit with skinning is the ordering of animation matrices, as in the code fragment above. Everybody gets stuck on this. The key line is this one: `our_mat = parent_mat * inv_bone_offset * local_anim * bone_offset;`. Because we're taking the slow route, we've already seen most of this before. We offset the bone, apply its own transformation, using the bone position as the pivot point, and un-offset the bone. The only difference here is that we apply its parent's transformation at the end. The line after this copies the matrix to our array on bone animation matrices.

```
...
for (int i = 0; i < node->num_children; i++) {
    skeleton_animate (
        node->children[i],
        our_mat,
        bone_offset_mats,
        bone_animation_mats
    );
}
}
```

If the skeleton node didn't have an associated bone, which we indicated with a bone index of -1, then we just send down its parent matrix to its children.

You should be able to compile and run now, and move the head up and down with the keyboard. When the ears rotate they should now respect the movement of the whole head, moving up and down with it.

Discussion

By now you have most of the concepts of skinning, and even have a place-holder matrix for where we work out a key-framed animation. Next we will look at making key-framed animations in Blender, importing them with AssImp, storing them in our skeleton tree, and working out an animation matrix with **interpolation** functions. If you haven't done so yet, it's a good time to brush up on quaternion rotation, because we are going to look at using these in our animations.

In a larger programme don't forget that you should probably think about writing a function that frees the memory allocated to the skeleton. You can do this recursively.

Part Three: Key-Frame Animations

In the previous part of these skinning tutorials we created a skeleton rig with parent-child relationships in the bones. We're going to continue by loading up this model again and creating some key-framed animations for it that we can play back in our animation software.

In a key-framed animation we create a **time-line** for the animation sequence. At arbitrary points within this time we will **pose** our skeleton, and save that pose as a **key**. This will record the translation, orientation, and scale, of the bones together with the time that this should occur. We will create a few keys along the time. To animate smoothly in-between these keys we will work out an **intermediate pose** by doing an **interpolation** of the previous key and the next key, based on the current time's proportion of the total time between the two frames.

This will involve 3 jobs;

1. adding the keys in our modelling software and exporting them
2. upgrading the importer code to grab the keys and store them in our animation nodes
3. upgrading the animation function to interpolate between keys.

The actual interpolation will use different methods for translations and scales, where we will compute a linear interpolation, and rotations, where we will compute a spherical interpolation.

Creating an Animation in Blender

Once again, I'll use Blender to explain the modelling tasks. This is because it's free, so you're most likely to use this one if you're tinkering around. Also because it has the most convoluted user-interface known to man, and it's extremely difficult to learn all the little quirks and short-cuts, so this is here to reduce your trial-and-error time to get something working. Most of these tasks should be very similar across other software, but the quirks will be different of course.



If we switch to Pose Mode in Blender we can start saving key-frames. It's helpful to open a Timeline panel to set the time for the current key. One thing that you will want to do on the Timeline panel is set a duration, because we will detect this and use it for looping our animation.

If we take our mesh and go into Pose Mode in Blender we can start animating and saving key-frames. I get a Timeline panel open on the bottom to help set the time for the current key-frame. The number here are 'frames', not seconds or anything. These default to 30 frames per second, I believe, so a duration of

300 should take 10 seconds to play. You can easily scale your animations in blender or in our code to speed this up or slow it down so it's not a big deal to change later. You'll see that the Timeline has a green needle that you can move to set the "current" time within the duration.

Now, the fun part! Set the Timeline needle to frame number 0 (the extreme left). We are going to set the starting key-frame to just be the default pose. Select all bones in the Pose Mode window and press "I" to set the current pose to a key at frame 0. You get a list of different aspects to save into the key. We are going to be changing the location and orientation, so choose location and rotation. We will loop our animation, and want to finish where we started, so now drag the Timeline needle to the end of the animation I set mine to be frame 250. Set a key here too. Now we have an animation that does nothing!

I set the Timeline needle to a few in-between positions. Here I changed the position and orientation of the bones in Pose Mode and saved a key. Once you have an interesting key or two you should be able to move the Timeline needle to preview how the interpolation will look between keys. You can play it back with `alt+a`. I moved the "head" parent bone up and down, and rotated the ears. When you're happy with your animation you can export the mesh again. To get an overview of what keys are saved for which bones you can go into "Dope Sheet" mode. Here you can even select, move, and scale keys. In the Dope Sheet, notice that each bone has its own, separate, "**channel**". This is a concept that will pop-up later.

If you want to set a name for your animation you can do so in "Outliner" mode in Blender by clicking on the new "action" that has appeared under Armature and Animation. The COLLADA export script doesn't export this properly, so it's not going to be a concern for us for now. To add several animations in Blender is actually really confusing so I'll conveniently skip that bit of complexity here. The COLLADA export script doesn't support multiple animations properly in the current version. You would have to use a different mesh format (probably the most sensible option, but you'll have to find a suitable exporter plug-in), a different modelling programme, or read animations from different mesh files and combine them.

Extracting Animations from AssImp

I'm going to make the assumption that we are only loading one animation per mesh file. It's going to be clearer to see how the code works, and it's not hard to upgrade to supporting multiple animations later.

Modify Skeleton Node

We don't need a separate data structure for animations - we can store the keys directly in our skeleton nodes (remember, we made a tree out of these in the last tutorial). We have potentially 3 types of keys to store; position, rotation, and scaling keys. The keys themselves are just the time that they occur in the animation, and a 3d vector or 4d quaternion, representing the position or orientation at that time. Let's start by adding arrays of these to the `Skeleton_Node` structure. I'll use pointers so that we can allocate the memory for them dynamically.

```
struct Skeleton_Node {
    Skeleton_Node* children[MAX_BONES];

    /* key frames */
    vec3* pos_keys; /* array of XYZ positions */
    versor* rot_keys; /* array of quaternion WXYZ rotations */
    vec3* sca_keys; /* array of XYZ scales */
    double* pos_key_times; /* array of times for position keys */
    double* rot_key_times;
    double* sca_key_times;
    int num_pos_keys; /* number of position keys for node */
    int num_rot_keys;
    int num_sca_keys;

    /* existing data */
    char name[64];
    int num_children;
    int bone_index;
};
```

I used my own maths library. Once again you could use an alternative, or simply arrays of `float`. The rotations are using a 4d versor, or unit quaternion, to represent orientation at each key. Remember to initialise each of these new variables where we create the node in the `import_skeleton_node` function.

Modify Mesh Loading Function

We are going to run a timer and a loop in our main function to tell us how far through the playing animation we are. I will just add in another parameter to the `load_mesh` function to retrieve this as a single value, because I am assuming that there is only one animation in the mesh. In a fuller example you might pass back an array of these durations.

```
bool load_mesh (    const char* file_name,
                    GLuint* vao,
                    int* point_count,
                    mat4* bone_offset_mats,
                    int* bone_count,
                    Skeleton_Node** root_node,
                    double* anim_duration
) {
```

In the function, right after where we created the skeleton tree last time, we can check for animations:

```
/* we did this in the last tutorial */
import_skeleton_node (assimp_node, root_node, *bone_count, bone_names);

/* start getting animation information */
if (scene->mNumAnimations > 0) {
    // get just the first animation
    aiAnimation* anim = scene->mAnimations[0];
    printf ("animation name: %s\n", anim->mName.C_Str ());
    printf ("animation has %i node channels\n", anim->mNumChannels);
    printf ("animation has %i mesh channels\n", anim->mNumMeshChannels);
    printf ("animation duration %f\n", anim->mDuration);
    printf ("ticks per second %f\n", anim->mTicksPerSecond);

    *anim_duration = anim->mDuration;

    /* WE WILL GET KEYS HERE */
}
```

So I just check if there are any animations at all. If so I get a pointer to the first animation, and print some details about it. We are only interested in "node" animation channels for skinning - the mesh animations in AssImp are a different sort of animation. Now would be a good time to compile and run, and make sure that you have some count of animation channels in your imported mesh. The Blender export script didn't preserve the animation name for me. Once we get the duration we can pass it back to the main function. The "ticks per second" is for speeding up or slowing down the animation by

some factor. I'm going to ignore this for now. You can see where we will add more to the function later.

Modify Main Function

In our main code we can use the duration value:

```
...
double monkey_anim_duration = 0.0;
assert (load_mesh (
    MESH_FILE,
    &monkey_vao,
    &monkey_point_count,
    monkey_bone_offset_matrices,
    &monkey_bone_count,
    &monkey_root_node,
    &monkey_anim_duration
));
double anim_timer = 0.0;
...
```

Inside the main loop I just update my little timer like so:

```
...
anim_time += elapsed_seconds;
if (anim_time >= monkey_anim_duration) {
    anim_time -= monkey_anim_duration;
}
...
```

Now we are ready to go - we just need to get the keys out, and update our animation function so that it uses the keys instead of the manual keyboard input. You might remember that last time we used a global arrays of `g_local_anims` to give us some keyboard control. You can delete this now, and the references to it. You can also make sure that `skeleton_animate` is called every frame now too, because we want to continuously play this animation.

Get Keys Out of Animation

AssImp is going to give us animation keys with just a name to link back to the nodes in our skeleton tree. Let's make a function to search the tree for a matching node:

```
Skeleton_Node* find_node_in_skeleton (
```

```

Skeleton_Node* find_node_in_skeleton (Skeleton_Node* root, const char* node_name) {
    assert (root); // validate self

    // look for match
    if (strcmp (node_name, root->name) == 0) {
        return root;
    }

    // recurse to children
    for (int i = 0; i < root->num_children; i++) {
        Skeleton_Node* child = find_node_in_skeleton (root->children[i], node_name);
        if (child != NULL) {
            return child;
        }
    }

    // no children match and no self match
    return NULL;
}

```

This function is given a node, i.e. the root node in our tree, and a name to look for. It checks the node, then recurses to all of the child nodes. If the name is matched, it returns a pointer to the node. If it fails, it returns `NULL`.

Right, now we can fill out that bit of the mesh loading function where we left the `/* WE WILL GET KEYS HERE */` comment. We are going to proceed by looping over all of the **channels** in the animation. This should be one channel per animated node in our tree, and should match up to the "Dope Sheet" in Blender. In each loop iteration we will get a pointer to the node matching the channel, using our new function. We will then count the number of each type of key, and allocate the appropriate amount of memory to hold all of the keys in our node:

```

// get the node channels
for (int i = 0; i < (int)anim->mNumChannels; i++) {
    aiNodeAnim* chan = anim->mChannels[i];
    // find the matching node in our skeleton by name
    Skeleton_Node* sn = find_node_in_skeleton (
        *root_node, chan->mNodeName.C_Str ());
    assert (sn);

    sn->num_pos_keys = chan->mNumPositionKeys;
    sn->num_rot_keys = chan->mNumRotationKeys;
    sn->num_sca_keys = chan->mNumScalingKeys;

    // allocate memory
    sn->pos_keys = (vec3*)malloc (sizeof (vec3) * sn->num_pos_keys);
    sn->rot_keys = (versor*)malloc (sizeof (versor) * sn->num_rot_keys);
    sn->sca_keys = (vec3*)malloc (sizeof (vec3) * sn->num_sca_keys);
    sn->pos_key_times =
        (double*)malloc (sizeof (double) * sn->num_pos_keys);
    sn->rot_key_times =

```

```

    (double*)malloc (sizeof (double) * sn->num_rot_keys);
sn->sca_key_times =
    (double*)malloc (sizeof (double) * sn->num_sca_keys);
...

```

We finish off the loop by copying the keys and times from AssImp into our node's allocated memory. This will differ, depending on the data types that you are using for vectors and quaternions. My vectors have an internal array of `float` called `v`, or `q` for quaternions, whereas AssImp uses separate `.x .y .z float` variables. Don't forget that quaternions should contain 4 `floats`. This should be fairly straight forward:

```

...
// add position keys to node
for (int i = 0; i < sn->num_pos_keys; i++) {
    aiVectorKey key = chan->mPositionKeys[i];
    sn->pos_keys[i].v[0] = key.mValue.x;
    sn->pos_keys[i].v[1] = key.mValue.y;
    sn->pos_keys[i].v[2] = key.mValue.z;
    sn->pos_key_times[i] = key.mTime;
}
// add rotation keys to node
for (int i = 0; i < sn->num_rot_keys; i++) {
    aiQuatKey key = chan->mRotationKeys[i];
    sn->rot_keys[i].q[0] = key.mValue.w;
    sn->rot_keys[i].q[1] = key.mValue.x;
    sn->rot_keys[i].q[2] = key.mValue.y;
    sn->rot_keys[i].q[3] = key.mValue.z;
    sn->rot_key_times[i] = key.mTime;
}
// add scaling keys to node
for (int i = 0; i < sn->num_sca_keys; i++) {
    aiVectorKey key = chan->mScalingKeys[i];
    sn->sca_keys[i].v[0] = key.mValue.x;
    sn->sca_keys[i].v[1] = key.mValue.y;
    sn->sca_keys[i].v[2] = key.mValue.z;
    sn->sca_key_times[i] = key.mTime;
} // endfor
} // endfor mNumChannels

```

One thing that you might have noted is that I store key-frames completely separately for rotations, scales, and positions, with their own times, even though we know we set these together in Blender. This is just convenient because AssImp gives them to us with this separation. You could add some code to combine these where times matched if you prefer.

Writing an Interpolation Function

Firstly, let's modify our `skeleton_animate()` function so that it takes the animation time as a parameter. I'll give the entire function in this section but break it into bits:

```
void skeleton_animate (
    Skeleton_Node* node,
    double anim_time,
    mat4 parent_mat,
    mat4* bone_offset_mats,
    mat4* bone_animation_mats
) {
    assert (node);

    /* the animation of a node after inheriting its parent's animation */
    mat4 our_mat = parent_mat;

    /* the animation for a particular bone at this time */
    mat4 local_anim = identity_mat4 ();
    ...
}
```

Interpolate Translation and Scale Keys

At the top of the animation function we can work out the translation of our node. To do this we need to find the keys **before** and **after** our current time in the animation.

```
...
mat4 node_T = identity_mat4 ();
if (node->num_pos_keys > 0) {
    int prev_key = 0;
    int next_key = 0;
    for (int i = 0; i < node->num_pos_keys - 1; i++) {
        prev_key = i;
        next_key = i + 1;
        if (node->pos_key_times[next_key] >= anim_time) {
            break;
        }
    } // endfor
...
}
```

Pretty simple; we just iterate through that array of keys that we allocated. In case of being at the very start of the animation we default to using key 0 as the previous key. In case of being at the very end we use the final key as the

next key.

Next we need to work out our time, t , as a factor between 0 and 1 between the previous and next keys. When we have that then our interpolated translation is $(1.0 - t) * \text{previous vector} + t * \text{next vector}$. This kind of linear interpolation is sometimes referred to as "lerp".

```
...
    float total_t = node->pos_key_times[next_key] - node->pos_key_times[prev_key];
    float t = (anim_time - node->pos_key_times[prev_key]) / total_t;
    vec3 vi = node->pos_keys[prev_key];
    vec3 vf = node->pos_keys[next_key];
    vec3 lerped = vi * (1.0f - t) + vf * t;
    node_T = translate (identity_mat4 (), lerped);
} // endif num_pos_keys > 0
...
```

Finally, I create a transformation matrix for the translation, because we are going to combine the rotation, scale, and translation together.

The scale part is exactly the same, but with scale keys - you can copy-paste and change if you want to support scales in your animation.

Interpolate Rotation Keys

Rotations will work a little differently, because we have to interpolate quaternions instead of vectors. I cover "slerp" in the Quaternions tutorial.

```
...
mat4 node_R = identity_mat4 ();
if (node->num_rot_keys > 0) {
    // find next and previous keys
    int prev_key = 0;
    int next_key = 0;
    for (int i = 0; i < node->num_rot_keys - 1; i++) {
        prev_key = i;
        next_key = i + 1;
        if (node->rot_key_times[next_key] >= anim_time) {
            break;
        }
    }
    float total_t = node->rot_key_times[next_key] - node->rot_key_times[prev_key];
    float t = (anim_time - node->rot_key_times[prev_key]) / total_t;
    versor qi = node->rot_keys[prev_key];
    versor qf = node->rot_keys[next_key];
    versor slerped = slerp (qi, qf, t);
    node_R = quat_to_mat4 (slerped);
} // endif num_rot_keys > 0
...
```

Finalising the animation

I combine all my interpolations into a single animation node. Don't forget to throw in a `node_S` first too, if you want to support scale keys:

```
...
local_anim = node_T * node_R;
...
```

Now, if you print any of the matrices or vectors that come out of the keys you'll notice something interesting, particularly the keys from the first "default/do nothing" pose. They don't have (0,0,0), they have the bone offsets in them! That can be very confusing and unexpected. It means though, that we no longer need our `inverse(bone_offset_matrix)` in the animation function. You can take that out and change our matrix multiplication function:

```
...
// if node has a weighted bone...
int bone_i = node->bone_index;
if (bone_i > -1) {
    // ... then get offset matrices
    mat4 bone_offset = bone_offset_mats[bone_i];

    // no more inverse
    our_mat = parent_mat * local_anim;
    bone_animation_mats[bone_i] = parent_mat * local_anim * bone_offset;
} // endif
for (int i = 0; i < node->num_children; i++) {
    skeleton_animate (
        node->children[i],
        anim_time,
        our_mat,
        bone_offset_mats,
        bone_animation_mats
    );
} // endfor
} // endfunction
```

That should do it! You should be playing your animation now if all went successfully. That was a lot more involved than it first appeared wasn't it? I skipped a few pieces that you might want to add-in now that you have the basics nailed down - remember that we didn't free memory, we didn't add support for weighting vertices to multiple bones, and we didn't support more than one animation per mesh. These should be fairly easy to add in now that we have the bare bones coded. Yes, I just said that.

Common Problems

- **360 degree animations "pop" around the wrong way!** - This is an issue with Slerp. We discuss how to detect and solve this at the end of Quaternions tutorial.
- **The animations are all mangled!** - The most likely issue is that the ordering of matrices in the recursive animation function is wrong. Check this first. Have you mixed up the order of animation matrices? I'm using column-major matrices. If you're using row-major then you'll need to reverse the order of multiplication.
- **My animations are 90 degrees off from my mesh orientation!** - Go back into your modelling software and make absolutely sure that your armature is lined-up on the same axis as your mesh. It's possible that they appear lined-up, but that there are orientations in the higher "object" level that aren't respected by the export-import process. The best idea in Blender is to "clear" the Object Mode orientations, and to adjust any orientation required in Edit Mode, which will be respected by our importer. Preview the animation now before exporting, to make sure that what you are exporting looks correct. If you've re-oriented the individual bones themselves in Edit Mode, rather than just adjusting them in Pose Mode when setting key-frames, then you might be introducing additional complexity. I keep all my bones un-oriented in Edit Mode, and ignore any rotation in the bone offset matrices when I import them. If you're importing someone else's mesh then you may not have this luxury, and will need to import the rotation part of the bone matrix, as well as the translation part.
- **My animation only works on a small part of the mesh!** - Remember that we assumed that there was only one mesh stored in the file. More complex models may be split into several objects or meshes, all of which are to be animated by the skeleton. You can either merge all of your objects in Blender, or upgrade the mesh importer so that it can deal with more than one mesh. This is going to be more complex to import, and slower to draw and animate, but might be necessary if each part should have a different material and texture to draw with.
- **Some of my bones are correct, but not others!** - If you're applying a

corrective matrix (like rotating the axis) to each node, then make sure that it doesn't get inherited! Otherwise you will get a whole series of corrections stack up for lower-level child nodes.

- **My translations are backwards!** - You might have a disagreement in the "handedness" of orientations between your modeller and your animation software. In Blender I had Z for "up" and Y for "forwards", so I just rotated my mesh and armature in Blender before I exported. This was -90 degrees around the X-axis to get Y for "up" and -Z for "forwards". If you're using a different coordinate system then you will just apply a different transformation ... or you can just give up and use your modeller's coordinate system in your animation software - then you don't need to apply any correction.
- **My rotations "pop" and snap around the wrong way at some point!**
 - This is a problem with SLERP, as explained in the Quaternions article. We correct this by checking if the dot product of the 2 versors is negative before calling SLERP. If so we negate one versor.

Discussion

Why We Don't Use AssImp During Animation

The suggested use of AssImp is to load the mesh, but also to use AssImp's own data structures during animation. We didn't do that - we re-created our own data structures. There are 2 advantages to not using AssImp during animation:

AssImp uses 3 different data structures for bones, the skeleton hierarchy, and animation key-frames. As we have seen, these are linked together by a common name string. This means that we have to do searches by name on the data structures to find the matching pieces, every time that we animate. This is not efficient, in respect to both the data structure traversal, and the string comparisons. Recreating our own data structures has simplified the domain from 3 data structures into 2; the skeleton tree and the array of bone animation matrices. We also store the index of the bone in each node of the tree, so there's no time wasted with traversals and string comparisons.

You've already noticed that AssImp is a huge dependency to compile against. It may well be bigger than your entire finished programme. It takes a long time to compile with if you are static linking. It will add considerable file size to any software that you intend to distribute. It is a pain maintaining a multi-platform build that has AssImp as a library dependency. All we really used AssImp for was importing a mesh format that was available for export in our modelling software. It's really very useful for that purpose because it supports so many formats. Consider that in our end use, our data for animation is actually very simple - just an array, a skeleton, and some simple keys with times. If you're comfortable with all the concepts involved it may be worth building a **stand-alone mesh converter** with AssImp. You can convert from virtually any mesh format into a simplified format that suits your software. Your simplified output file could even be a binary data dump that you can load directly into your arrays. I did this for my video game. This means that the actual game doesn't depend on AssImp, is much smaller and simpler to distribute and build, and loads meshes much more quickly.

Multiple Animations

Eventually, you might like to upgrade your importer to support multiple animations for a single mesh; a character with walk/idle/swim/crawl/run for example. We only retrieved the first animation in the array from AssImp. You can replace this with a for-loop. You will also need to modify `Skeleton_Node` to support keys from several different (numbered) animations - use arrays of arrays. You can add an animation index parameter to your animation function so that it retrieves the correct keys. You can see that this is going to get a little bit involved so it was much easier to start with one working animation!

The Problem With COLLADA

COLLADA is okay-ish as an animated mesh format. I spent quite a lot of time looking for an export-import pipeline for these tutorials that was both simple and widely supported, and frankly there isn't one (yet). The mesh file itself is very verbose XML, which is slow, too large, and hard to manually read. The specification is not entirely clear, and as we've seen exporters don't stick to it properly anyway. There is no other commonly available animated mesh exporter that produces a reliable animated mesh file - the alternatives are worse, or are not available for the latest versions of Blender. It would be nice to not need to use AssImp, but COLLADA is too complex (and incredibly confusing) to write a simple parser for that would suit a tutorial.

You wouldn't want to use .dae files in a serious performance-driven piece of software - you'd end up making a custom in-house (probably binary-encoded) format. The advantage of using COLLADA over an interchange format from a game engine or rendering engine's format is wider support for the .dae format. I felt that this was the right way to get started from a fairly common base, even if it is a little complex.

glTF

It's worth keeping an eye on the up-and-coming glTF transmission format,

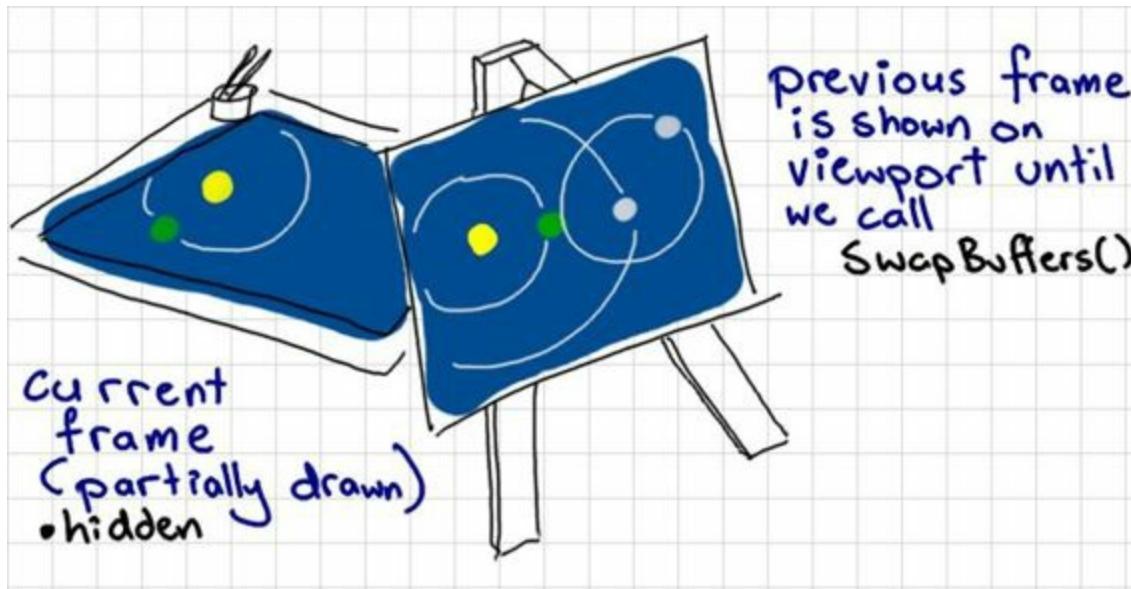
which looks like an improvement over COLLADA in many ways. It looks like it primarily takes aim at WebGL, which is very handy, but it should generalise well to all platforms. It uses a binary encoding of modern mesh data, supporting all the bits that you might want to import, as well as being very data-size and parsing-time efficient. It has a JSON interface (JavaScript Object Notation) for the node hierarchy, which is much more readable (both by humans and parsers) than an XML interface. You can also store image files and shaders directly in the format. The main downside, as I see it, is the availability of exporters. If there was a native exporter for Blender I would have used it for this example instead of COLLADA. I expect this to change soon. There is a tool to convert from COLLADA to glTF, which might be worth considering, in which case it would be possible to drop AssImp and write a much more efficient importer using just a JSON parser. I may adopt this in the near future. There is a small problem with JSON parsers being more suited to object-oriented programming than C programming, but perhaps this is worth it for simplifying the explanation of the import process. You can have a read about glTF at the Github project page, where there is a specification [README.md](#).

Framebuffers

Overview

So far we have been using a rendering loop that looks something like this:

1. `glClear ()`; - wipe the current drawing surface clean
2. `glDrawArrays ()`; - draw a new shape onto the drawing surface
3. `glDrawArrays ()`; - draw another shape onto the drawing surface
4. `glfwPollEvents ()`; - update key presses and stuff
5. `glfwSwapBuffers (window)`; - finish, and show the drawing surface on-screen

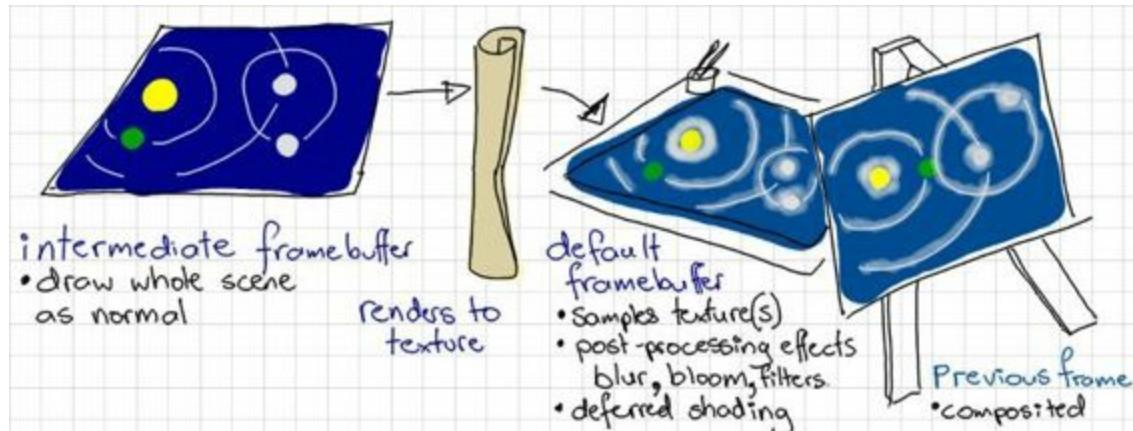


A typical double-buffering system shows the previously drawn frame until drawing the current frame is finished. In the image above the current frame has only drawn 2 of the objects - in a single-buffered system the viewer would see these pop onto the screen as they were drawn.

Each iteration of this loop is called a **frame**. The drawing surface that we are clearing and drawing to is called a **framebuffer**. It's just a 2d image with the same dimensions as the viewport. We are using a double-buffering system. This means that our in-progress drawing is hidden until we are finished drawing it, then we put it on the screen, and use a new hidden buffer for drawing the next frame. In a single-buffering system we would visibly

notice our shapes being drawn on, one at a time.

Note: "frame buffers" should not be a compound word, but I'm using it like that here because OpenGL uses it as if it was a single word in the API - otherwise I keep writing `glFrameBuffer` when it is actually `glFramebuffer`.



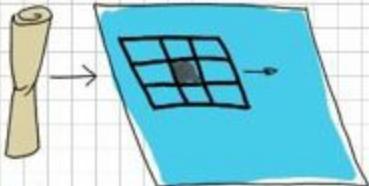
To do a post-processing effect like "bloom" we need to check the neighbours of each pixel. We can't do this for fragments in a fragment shader. We can however, do this when sampling a texture. We create a new framebuffer, and tell it to render to a texture. We render the scene as usual in this first rendering "pass". Then we bind the default framebuffer, where we draw a quad that fits the screen. In a fragment shader we sample the texture and compute any effects. We can also exploit multi-pass rendering to make optimisations such as deferred shading.

So far we have been using the default framebuffer; number 0. If we create a new framebuffer we can tell it to draw to a texture. We can then re-use that texture for drawing another rendering pass. The most common use is for **post-processing effects**, which works like this:

1. render whole scene to new framebuffer, which writes to a texture
2. render a rectangle which covers the screen using the default framebuffer, but use the texture from (1) to texture the rectangle

In the second pass we still render each fragment as usual, except we can also sample the pixels around our fragment. This lets us do fancy filtering and blurring effects, which we couldn't do before.

Advantages of Post-Processing (>1 framebuffer)

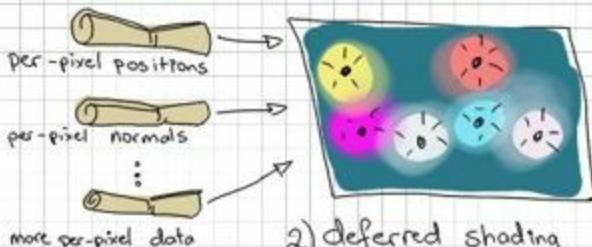


1) Image Filters with a Kernel.

- modify pixel colour based on neighbours

3) A Chain of Effects (Compositing) e.g.

1. Scene 2. edge detection 3. blur



2) deferred shading

- output fragment shader variables to textures
- depth sort only keeps 1 closest fragment's data for each pixel
- compute lighting etc. in 2nd pass using variables sampled from textures

Most commonly we use a second framebuffer to allow us to process images using a **kernel** (sliding window) filter; blurs, blooms, edge detection, etc. An optimisation technique called **deferred shading** exploits the fact that we can put any data in a texture to do calculations like lighting per-pixel instead of per-fragment. And finally, we can chain together several framebuffers to create multi-pass compositions. A lot of image processing techniques produce inaccurate or sharp edges so it is quite common to do a blur afterwards to smooth over these.

There are other uses for rendering in different passes. We can render a scene from different viewpoints; imagine creating a rear-vision mirror for a car, or a monitor displaying CCTV footage from somewhere else in the virtual world. We can use this to project shadows from a light source, using a technique called **shadow mapping**.

Simple Example

Code to Set Up a Second Framebuffer

First we can create the additional framebuffer.

```
GLuint fb;
 glGenFramebuffers (1, &fb);
```

We'll also create a blank texture, the same size as the viewport. I've used `width`, and `height` here, but you might have different variable names. I then bind the framebuffer, and attach the texture to it, so that the colour output goes to the texture instead of the viewport.

```
GLuint fb_tex;
 glGenTextures (1, &fb_tex);
 glBindTexture (GL_TEXTURE_2D, fb_tex);
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_RGBA,
    width,
    height,
    0,
    GL_RGBA,
    GL_UNSIGNED_BYTE,
    NULL
);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glBindFramebuffer (GL_FRAMEBUFFER, fb);
 glFramebufferTexture2D (
    GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, fb_tex, 0
);
```

We also want to make sure that the new render-to-texture framebuffer still uses depth sorting. One way to do this is to make something called a **renderbuffer** (yes, another ambiguous OpenGL naming convention). This is attached to the framebuffer as well, in a similar way to the colour output textures. However, this is unofficially deprecated, and we can, for all intents and purposes just attach another texture, and set it to capture depth. This has the added advantage of us being able to use it like a texture, if we want to

display it for debugging, for example. I tried both approaches in a little demo, and there is no noticeable difference to performance. I will show you both here, and you can decide how to proceed:

Alternative: Renderbuffer

```
GLuint rb = 0;
 glGenRenderbuffers (1, &rb);
 glBindRenderbuffer (GL_RENDERBUFFER, rb);
 glRenderbufferStorage (
    GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height
);
 glFramebufferRenderbuffer (
    GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rb
);
```

The last command attaches the renderbuffer to our currently-bound framebuffer.

Alternative: Depth Texture

```
GLuint depth_tex;
 glGenTextures (1, &depth_tex);
 glBindTexture (GL_TEXTURE_2D, depth_tex);
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_DEPTH_COMPONENT,
    width,
    height,
    0,
    GL_DEPTH_COMPONENT,
    GL_UNSIGNED_BYTE,
    NULL
);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 // attach depth texture to framebuffer
 glFramebufferTexture2D (
    GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depth_tex, 0);
```

A little bit more code to set up, but a bit more versatile than the renderbuffer. Whenever you see a renderbuffer later, you can also replace that with a depth texture if you like.

Finalising the Framebuffer

Finally, we need to explicitly specify the list of colour buffers that are to be used for drawing with `glDrawBuffers`. If we have multiple colour outputs from the fragment shader, then we can give it an array of buffers here. These correspond to the colour attachment numbers that we set in `glFramebufferTexture2D`. We only have 1 colour buffer, so it looks like this:

```
GLenum draw_bufs[] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers (1, draw_bufs);
```

Aside: The above command seems a bit excessive - perhaps this will go away in future versions of the API - it seems like it could have been easily merged with the `glFramebufferTexture2D` function parameters.

We can now test the framebuffer for "completeness". This will throw an error if there is a problem with the depth or texture attachments. If you need more detail on why the framebuffer is "incomplete" then you can find a full description of different error codes on the API page for `glCheckFramebufferStatus`.

```
GLenum status = glCheckFramebufferStatus (GL_FRAMEBUFFER);
if (GL_FRAMEBUFFER_COMPLETE != status) {
    fprintf (stderr, "ERROR: incomplete framebuffer\n");
}
```

Finally, we can bind the default framebuffer again, just to be on the safe side:

```
glBindFramebuffer (GL_FRAMEBUFFER, 0);
```

Set Up Screen-Space Quad

So, the first rendering pass is going to use our new framebuffer. All we will do is bind that, then draw as normal, and it will be drawn onto the texture. In the second pass, we will bind the default framebuffer, and sample the texture. But we need something to draw the texture on to. The standard way of doing this is to create 2 triangles that cover the screen in clip space (a screen space quad). We will make the usual VBO/VAO set for this. We will need points, and texture coordinates.

```
// x,y vertex positions
float ss_quad_pos[] = {
    -1.0, -1.0,
    1.0, -1.0,
    1.0,  1.0,
```

```

    1.0,  1.0,
    -1.0, 1.0,
    -1.0, -1.0
};

// per-vertex texture coordinates
float ss_quad_st[] = {
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0,
    1.0, 1.0,
    0.0, 1.0,
    0.0, 0.0
};

// create VBOs and VAO in the usual way
...

```

Note that I've used only 2d positions because we will add in a z value of 0 in the vertex shader. If you draw these positions, it will create 2 triangles covering x,z from -1,-1, to 1,1. In other words: the whole of homogeneous clip space. It will cover all of screen space if we output this: `gl_Position = vec4(vp, 0.0, 1.0);.`

Main Drawing Loop

The code in our drawing loop is going to look like the following code. The VAO from the previous section is here called `ss_quad_vao`.

```

glViewport (0, 0, width, height);
while (!glfwWindowShouldClose (window)) {
    // bind the second (render-to-texture) framebuffer
    glBindFramebuffer (GL_FRAMEBUFFER, fb);
    // clear the framebuffer's colour and depth buffers
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // render scene as normal here
    ...
    ...

    // bind default framebuffer
    glBindFramebuffer (GL_FRAMEBUFFER, 0);
    // clear the framebuffer's colour and depth buffers
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // our post-processing shader for the screen-space quad
    glUseProgram (post_process_sp);
    // bind the quad's VAO
    glBindVertexArray (ss_quad_vao);
    // activate the first texture slot and put texture from previous pass in it
    glEnableTexture (GL_TEXTURE0);
    glBindTexture (GL_TEXTURE_2D, fb_tex);
    // draw the quad
    glDrawArrays (GL_TRIANGLES, 0, 6);
}

```

```

// flip drawn framebuffer onto the display
glfwSwapBuffers (window);
glfwPollEvents ();

if (GLFW_PRESS == glfwGetKey (window, GLFW_KEY_ESCAPE)) {
    glfwSetWindowShouldClose (window, 1);
}
}

```

Don't forget to do your normal scene rendering (shaders, VAOs, etc.) in the ... bit! You can see then that post-processing can be pretty easily switched on and off, as it's just a bit of code that goes on either side of your main rendering calls.

Next we will create the shader programme that I have called `post_process_sp`.

Example Image Processing Shaders

So, you've probably realised by now that the screen space quad is going to need its own shaders to render with. You might create something basic first that colours both triangles red, for example, to test that the quad is working. When you are ready we can proceed. The following code will invert the colour of everything on the right-hand side of the screen.

Post-Processing Vertex Shader

```

#version 400

// vertex positions input attribute
in vec2 vp;

// per-vertex texture coordinates input attribute
in vec2 vt;

// texture coordinates to be interpolated to fragment shaders
out vec2 st;

void main () {
    // interpolate texture coordinates
    st = vt;
    // transform vertex position to clip space (camera view and perspective)
    gl_Position = vec4 (vp, 0.0, 1.0);
}

```

The vertex shader isn't very interesting; it just positions the 2 triangles, and passes through the texture coordinates.

Post-Processing Fragment Shader

```
#version 400

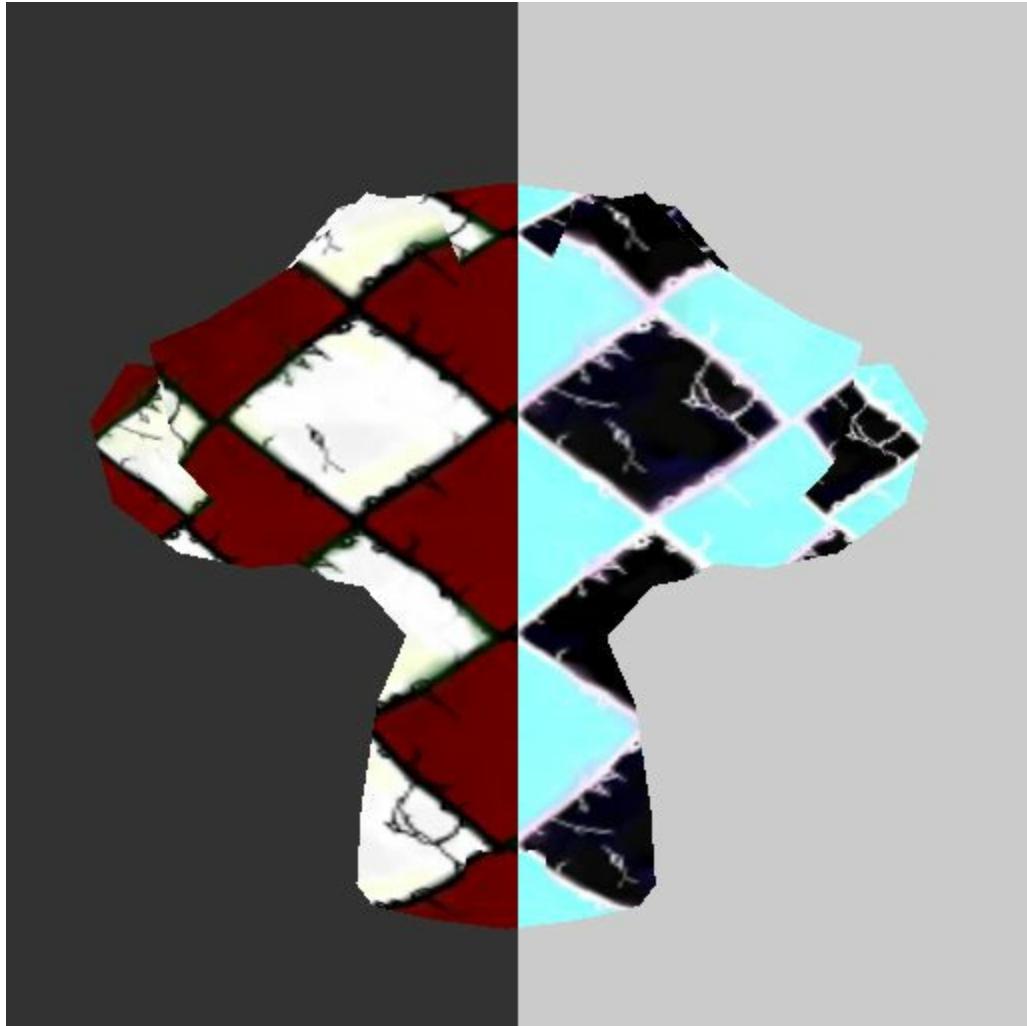
// texture coordinates from vertex shaders
in vec2 st;

// texture sampler
uniform sampler2D tex;

// output fragment colour RGBA
out vec4 frag_colour;

void main () {
    // invert colour of right-hand side
    vec3 colour;
    if (st.s >= 0.5) {
        colour = 1.0 - texture (tex, st).rgb;
    } else {
        colour = texture (tex, st).rgb;
    }
    frag_colour = vec4 (colour, 1.0);
}
```

The fragment shader samples a texture - in this case the texture that captured the output from the first rendering pass. If we draw this without modification it should have the appearance of the normal scene. Here it just modifies pixels that have texture coordinates greater than half-way along the s axis.



It's quite handy to only affect half of the original image, so that you can see what difference your filter makes in a side-by-side. Most post-processing effects are very subtle, so it can be handy to check if it's actually making an improvement outside of your imagination!

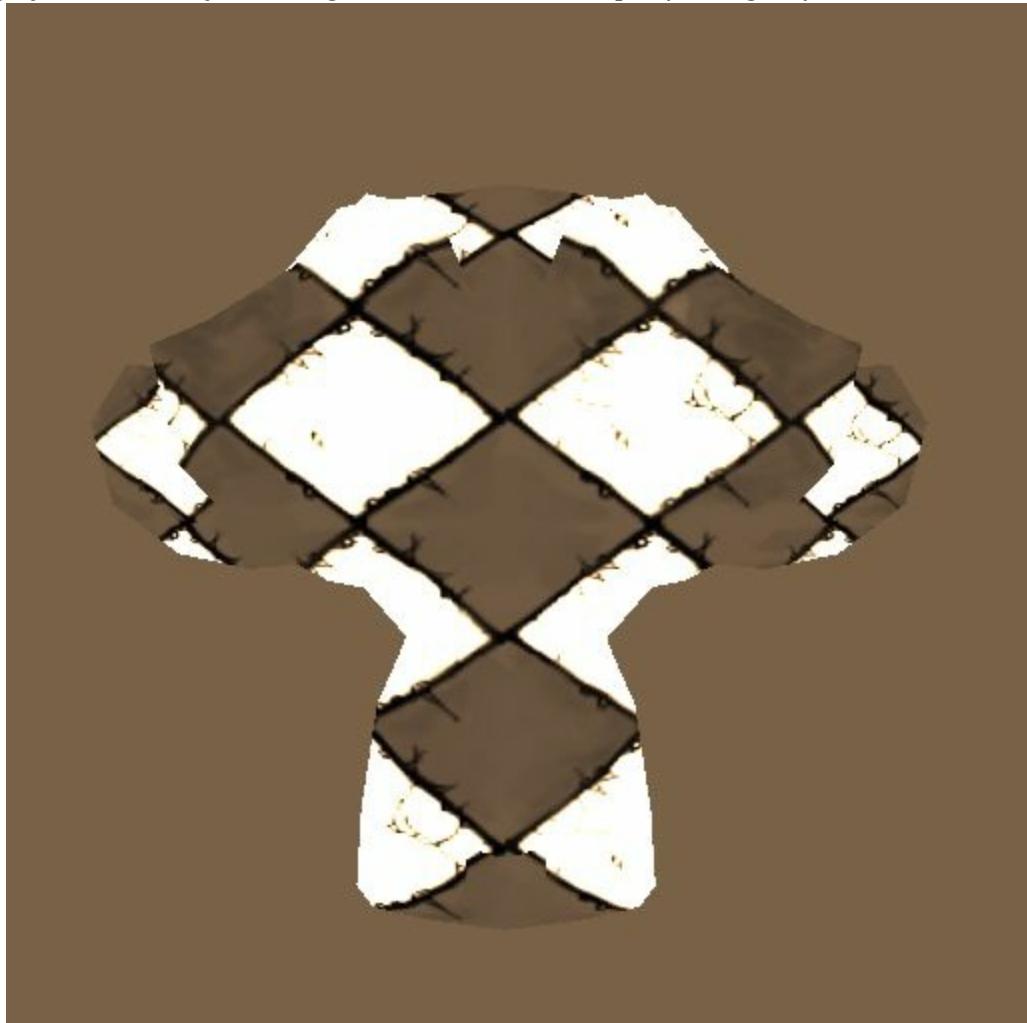
Exercises/Ideas

If that worked here are some post-processing ideas that you can try. Some simple, and some a little challenging:

- convert to greyscale (use a luminance weights)
- convert the greyscale to sepia
- flip image
- skew image
- repeat/tile image
- animated swirl or screw of image (use a time uniform)
- modify colour components based on the time and \cos (psychadelic)



A good greyscale doesn't just average the colours - it uses specific weights for each colour component.



To create a sepia filter, convert to greyscale first, and then multiply the colour with some new sepia weights.

Caveats

- Make sure that you always know which framebuffer is currently bound. It is a good habit to always re-bind the default framebuffer (number 0), when you are finished with using a secondary framebuffer.
- Be extra careful not to sample the framebuffer texture whilst its framebuffer is still bound. This kills the framebuffer - you'll get a read-write feedback loop.
- Be sure to set texture parameters for filtering and wrapping - otherwise you may be unable to write to the texture properly, and will get a black screen!

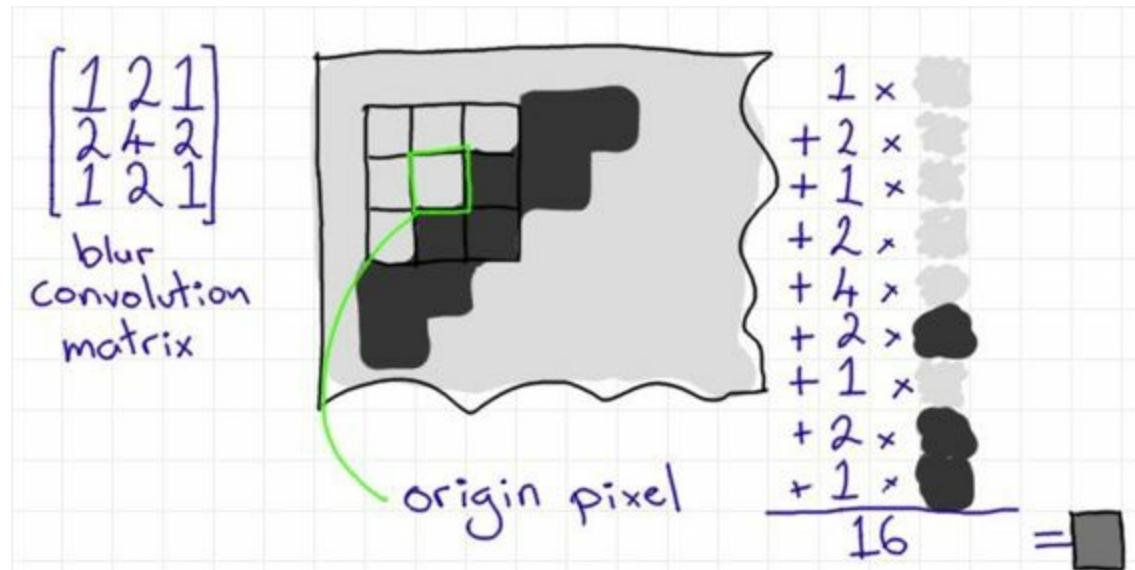
Image Processing with Kernels

Convolution

Having a render-to-texture framebuffer lets us colour a pixel based on its **neighbours**. We don't know the colours of the neighbouring pixels yet in single-pass rendering.

When colouring a pixel based on its neighbours we usually talk about a sliding window, or **kernel** - usually a 3x3 or 5x5 grid; a **convolution matrix** that says which neighbouring pixels should influence our "origin" pixel, and how much. A **weight** is given to each pixel covered by the kernel. This is the basis for many filtering algorithms.

For example; a blur filter has the following 3x3 kernel and weights:



The origin pixel is light grey, but next to a dark grey line. We convolve the surrounding pixels using the kernel's weights. This means multiplying the weights with the values of the surrounding pixels and summing them all together. The result is divided by the sum of the weights (16) to normalise it, which gives us a slightly darker grey. You can see that over the whole image this would draw a medium-grey line on either side of the dark line; which gives a smoothed or blurred appearance.

We multiply each pixel value by the weight in the kernel. All the results are

summed together. This usually gives a much higher colour intensity value, so we typically normalise it, or divide it by the sum of the weights.

Alternatively, the weights can be created such that they sum up to 1.0. The example image, above uses shades of grey. Some filter algorithms work better on a greyscale image, whilst others will run once per channel (multiply `vec3s` by the weight instead of a single colour value).

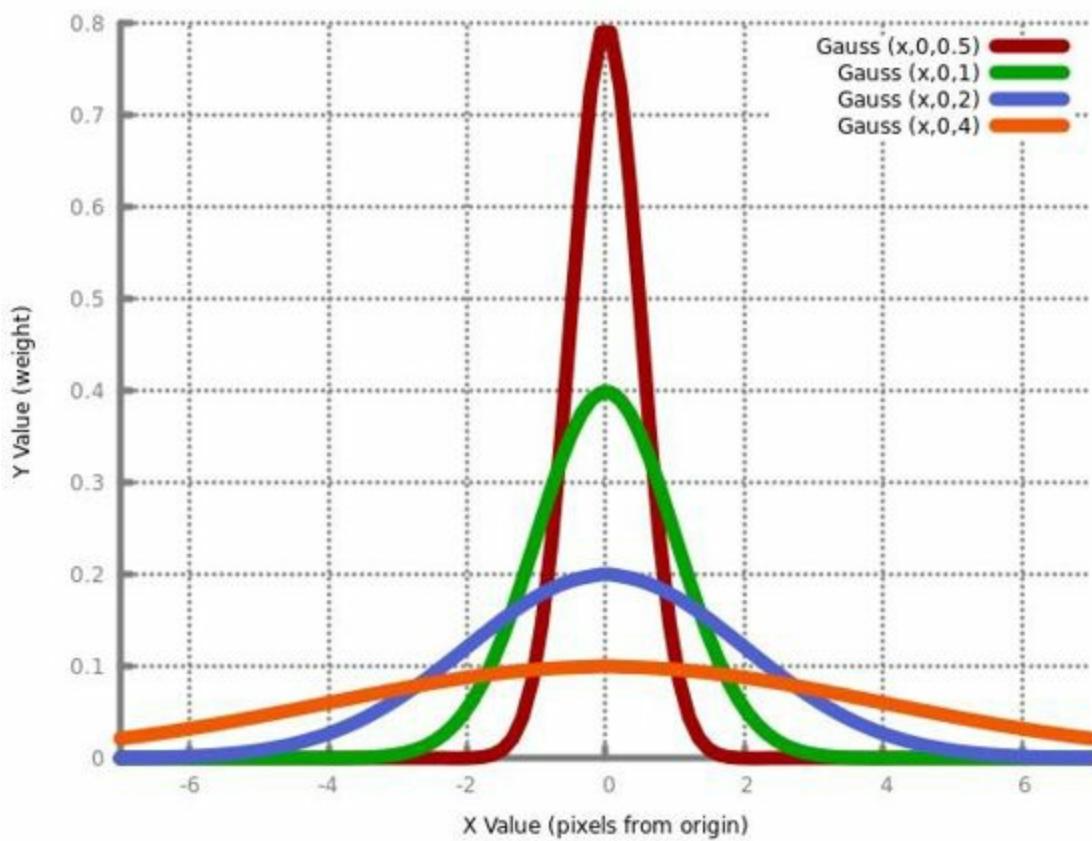
Edge Handling

What happens when the origin pixel is on the edge of the texture? The neighbours will be outside the image. We need a strategy for dealing with this. Typical options are:

1. **Prevent sampling at edges.** In the shader check if pixel being sampled has a texture coordinate less than 0 or greater than 1. If so, do not modify the pixel, and move on. This might create a visible border to your resulting image. We can slightly offset the per-vertex texture coordinates of the screen-space quad to hide this.
2. **Wrap around and sample the pixel on the opposite side.** When we create the framebuffer texture, we can set the S and T wrap parameters for the texture; `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);`, and the same for `T` - these should be the default. This might suit tiling textures, but probably isn't as useful for whole-scene filtering.
3. **Use a pre-defined "border" colour value for any edge pixels.** We can specify a border colour for the texture with `glTexParameter`, and set the clamp mode to `GL_CLAMP_TO_BORDER`. This might be some neutral value that does not give a positive feedback to feature detection algorithms.

Example: Gaussian Blur

Blur filters are very common either to create an out-of-focus effect directly, or as part of a multi-part filter to simplify the input image before processing i.e. a low-pass filter, or to clean up noise or smooth out patterns appearing in the output image. Wikipedia has some good examples under "Gaussian_blur".



Gauss functions with different values of sigma. All values of mu are set to 0. Our Gauss with sigma value 2 is going to be similar to a linear blur, but the sigma of 1 and 0.5 will produce blurs that more heavily weight the origin.

We can use a **linear blur** (similar to the first figure), but this is going to blur over much of the detail in the image. A Gaussian is a 2d **normal distribution** function that has a very big value in the middle (depending on the value of sigma used), and much smaller values to either side. This is going to preserve more of the detail in the image than a linear blur. For 2d we use the product of 2 of these functions. To increase the blurring effect, we can either repeat

the Gaussian blur, or use a bigger kernel e.g. 5x5 instead of 3x3.

Calculate Weights

We want to populate our kernel with result values from the Gaussian. We choose a sigma. We can use our kernel horizontal distance from the origin pixel as x e.g. 2,1,0,1,2 for one each row. Our kernel is 2d though, so we really want a 2d version of the Gaussian function. This is the product of 2 Gaussians, and for a symmetrical kernel it looks like this:

```
weight = exp(-(col * col + row * row) / (2.0 * sigma * sigma)) / (2.0 * pi * sigma * sigma)
```

We can calculate this weight for each cell in the kernel. No need to code this; it can be done by hand before coding. Remember to normalise the kernel - sum up all the weights and divide each weight by that total. Here's my fragment shader using a 5x5 kernel with a sigma of 0.8.

```
// GLSL version 4.0.0 corresponds to OpenGL version 4.0
#version 400

// texture coordinates from vertex shaders
in vec2 st;

// texture sampler
uniform sampler2D tex;

// size of 1 pixel in texture coordinates
uniform vec2 pixel_scale;

// output fragment colour RGBA
out vec4 frag_colour;

// Gaussian kernel weights
#define KERNEL_SIZE 25
float kernel_weights[] = float[]{
    0.00048031, 0.00500493, 0.01093176, 0.00500493, 0.00048031,
    0.00500493, 0.05215252, 0.11391157, 0.05215252, 0.00500493,
    0.01093176, 0.11391157, 0.24880573, 0.11391157, 0.01093176,
    0.00500493, 0.05215252, 0.11391157, 0.05215252, 0.00500493,
    0.00048031, 0.00500493, 0.01093176, 0.00500493, 0.00048031
};
float weights_factor = 1.01238;
...
```

See if you can follow how I calculated the weights; I got the 0.00500493 results by taking a `row` and `col` of 1 and 2, and so using the equation above:

```
= exp(-(1 * 1 + 2 * 2) / (2.0 * 0.8 * 0.8)) / (2.0 * 3.14 * 0.8 * 0.8)
```

Where `exp` is the Euler number (approximately 2.71828) with an exponent. You'll get slightly different results with different accuracies of pi and e. The sum of all of my weights was 0.98762, which will darken the image a tiny bit, so I will remember to multiply the final colour by 1.01238.

Calculate Texture Coordinate Offsets

Our texture coordinates are in the range of 0 to 1, but we want to know what proportion of this range equates to the distance between pixels. This depends on the dimensions of our texture. Remember that our texture is the same size as the viewport. We can create a `vec2` uniform in our fragment shader to hold the 1-pixel scale so that we can offset the texture coordinates to find the neighbouring pixels.

```
int pixel_scale_loc = glGetUniformLocation (sp, "pixel_scale");
float x_scale = 1.0f / width;
float y_scale = 1.0f / height;
glUniform2f (pixel_scale_loc, x_scale, y_scale);
```

I assume that we have a shader programme variable `sp`, and that the fragment shader has a `uniform vec2 pixel_scale`. Remember that if the viewport resizes during runtime then you will need to update this uniform.

In the fragment shader we can now add an array of texture offsets; one for each cell in the kernel, or neighbouring pixel that we are going to look up. I separated this array into rows of five cells from top to bottom

```
void main () {
    vec2 offset[] = vec2[](
        vec2 (-pixel_scale.s * 2.0, -pixel_scale.t * 2.0),
        vec2 (-pixel_scale.s, -pixel_scale.t * 2.0),
        vec2 (0.0, -pixel_scale.t * 2.0),
        vec2 (pixel_scale.s, -pixel_scale.t * 2.0),
        vec2 (pixel_scale.s * 2.0, -pixel_scale.t * 2.0),

        vec2 (-pixel_scale.s * 2.0, -pixel_scale.t),
        vec2 (-pixel_scale.s, -pixel_scale.t),
        vec2 (0.0, -pixel_scale.t),
        vec2 (pixel_scale.s, -pixel_scale.t),
        vec2 (pixel_scale.s * 2.0, -pixel_scale.t),

        vec2 (-pixel_scale.s * 2.0, 0.0),
        vec2 (-pixel_scale.s, 0.0),
```

```

    vec2 (0.0, 0.0),
    vec2 (pixel_scale.s, 0.0),
    vec2 (pixel_scale.s * 2.0, 0.0),

    vec2 (-pixel_scale.s * 2.0, pixel_scale.t),
    vec2 (-pixel_scale.s, pixel_scale.t),
    vec2 (0.0, pixel_scale.t),
    vec2 (pixel_scale.s, pixel_scale.t),
    vec2 (pixel_scale.s * 2.0, pixel_scale.t),

    vec2 (-pixel_scale.s * 2.0, pixel_scale.t * 2.0),
    vec2 (-pixel_scale.s, pixel_scale.t * 2.0),
    vec2 (0.0, pixel_scale.t * 2.0),
    vec2 (pixel_scale.s, pixel_scale.t * 2.0),
    vec2 (pixel_scale.s * 2.0, pixel_scale.t * 2.0)
);

...

```

If you know that your window can not be resized, then it would be desireable to hard-code the pixel scale value here rather than use a uniform. Note the unusual layout that GLSL uses for initialising arrays. This might not be so supported across drivers, especially older ones, so you may have to separate the arrays into individual variables.

Compare Results

Now we can just loop through all 25 pixels in the kernel area, sample each one from the texture using the offset coordinates, weight the result using the kernel weights, and add the whole lot together into a final colour for the pixel. I multiplied the result by my brightening factor here to make sure that it was perfect (I couldn't spot the difference though).

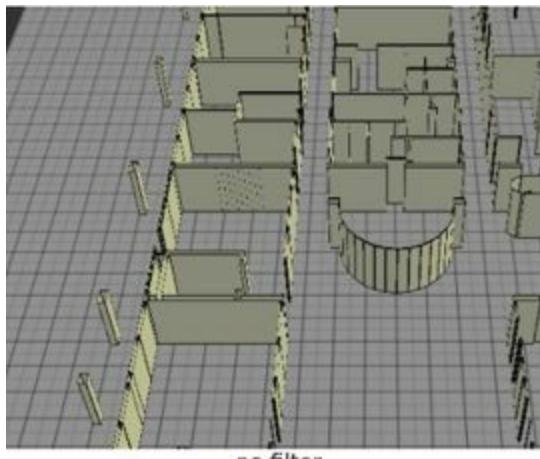
```

vec3 colour;
// only blur rhs for comparison
if (st.s >= 0.5) {
    for (int i = 0; i < KERNEL_SIZE; i++) {
        colour += texture (tex, st + offset[i]).rgb * kernel_weights[i] *
weights_factor;
    }
} else {
    colour = texture(tex, st).rgb;
}
frag_colour = vec4 (colour, 1.0);
}

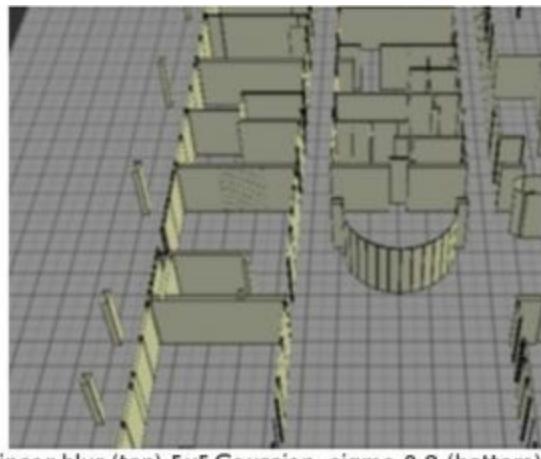
```



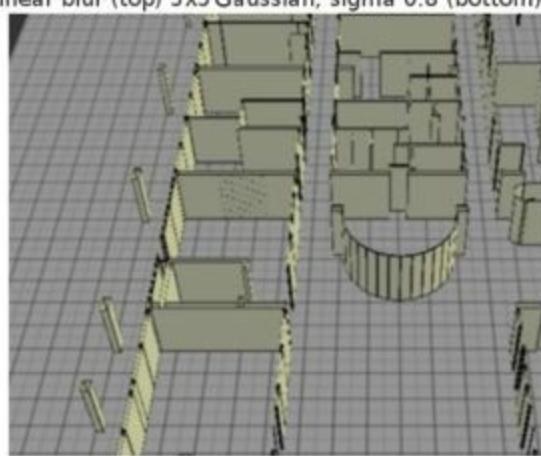
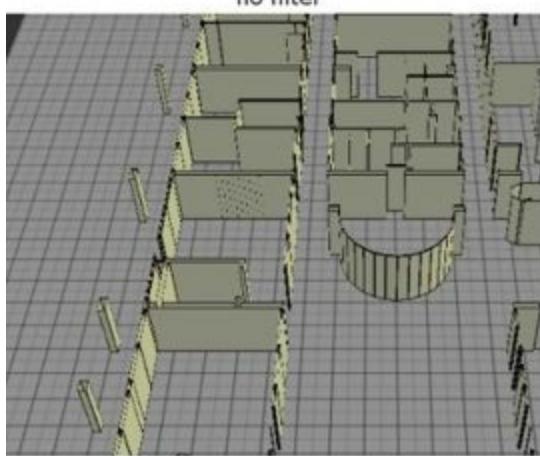
*The right half of the image above has a Gaussian blur filter applied with a sigma of 0.8 in a 5x5 kernel.
Note that the aliased edges of the mesh are noticeably smoother on the bottom-right.*



no filter



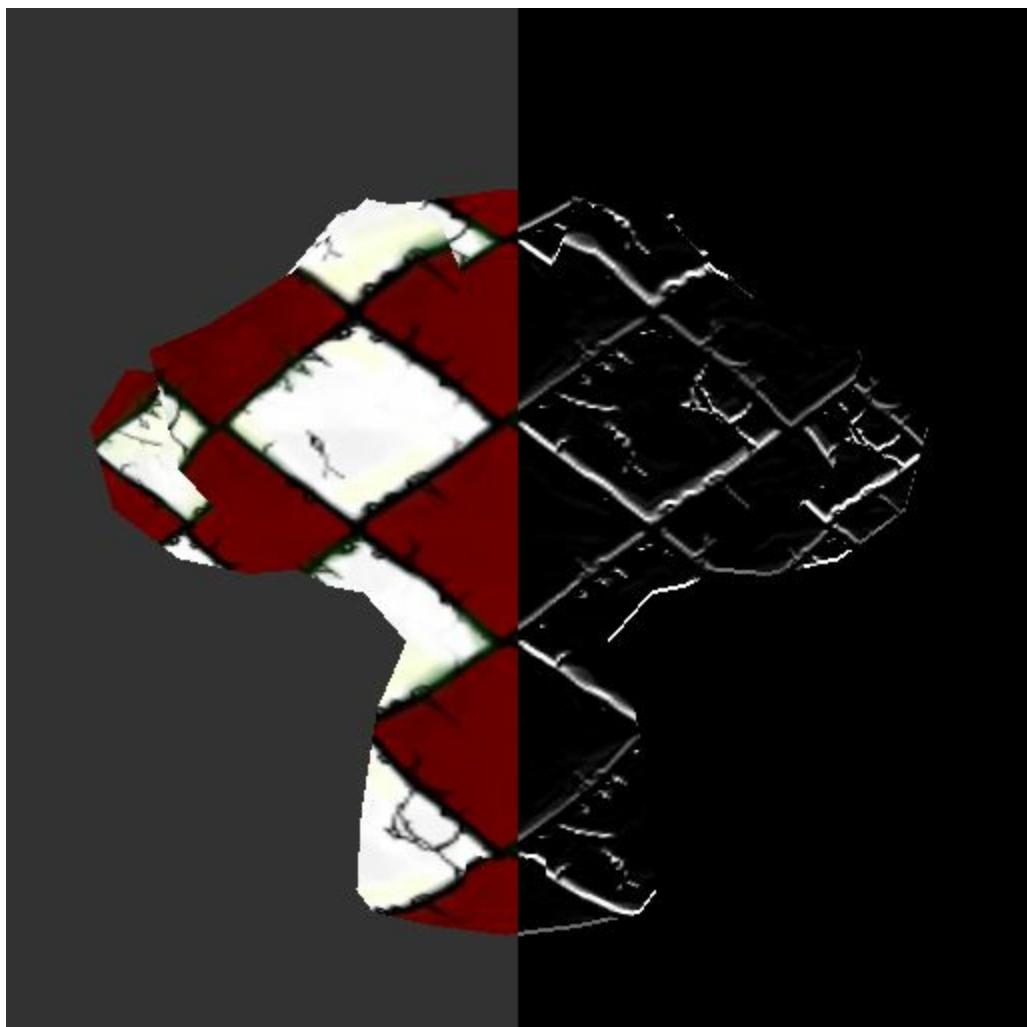
linear blur (top) 5x5 Gaussian, sigma 0.8 (bottom)



A comparison of filters between a viewport with some aliasing, compared to the effect of 5x5 linear and Gaussian blur filters.

Other Kernel Filters

Any computer vision or image processing textbook will give you a lot of ideas for filtering. Many of the techniques in computer vision are interested in **feature detection**; finding straight lines, shapes, or the edges of things in an image. The output of these techniques usually draws a greyscale image. Here's an example of an edge-detection filter. Try making a kernel with one side of the weights negated. If you sample the image and convert it to greyscale, you can use your kernel as an edge-detection filter:



A simple edge-detection filter in one direction only. You can improve these sorts of filters by blurring the image first to differing degrees - this effectively reduces the "noisy" edges so that only the main edges are shown.

Colour-Based Mouse Picking

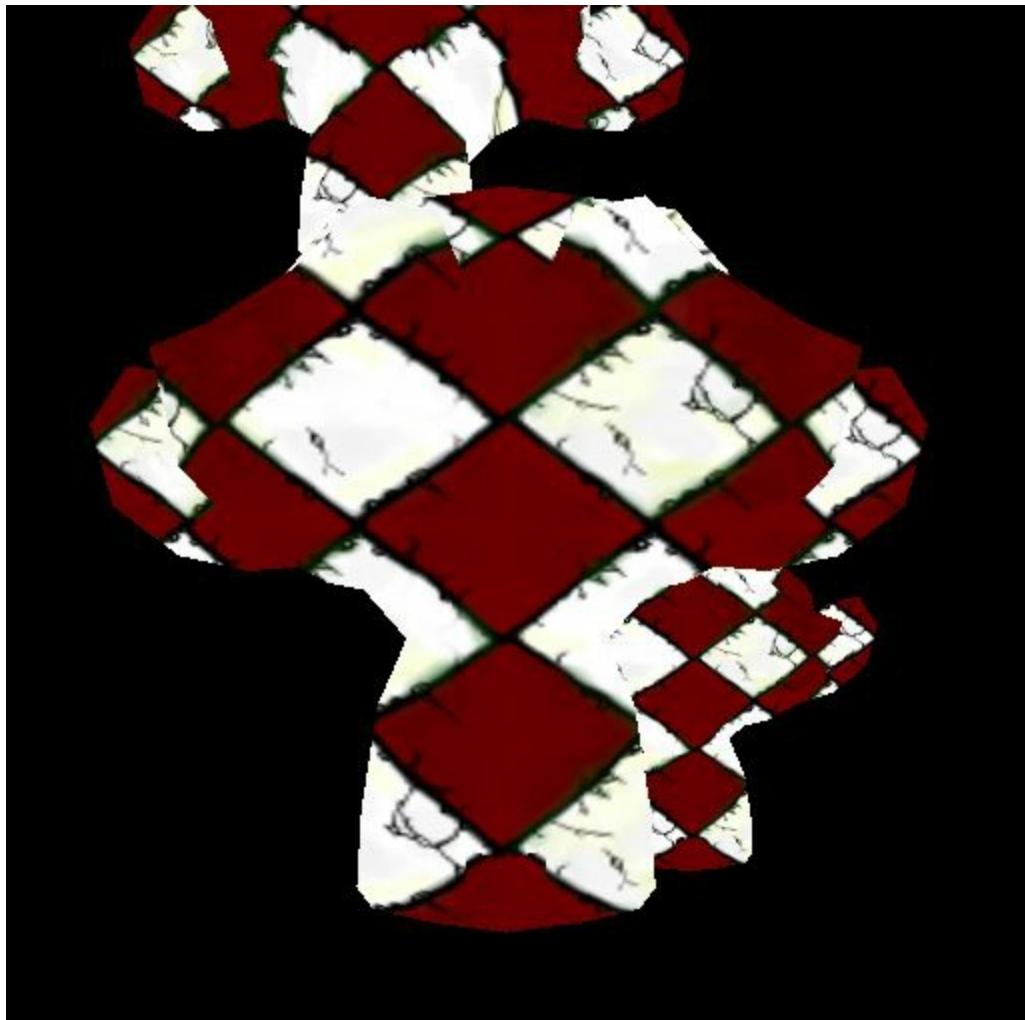
If we are doing a lot mouse cursor selection of objects in a 3d scene, then ray-based picking becomes very computationally expensive, especially if we need to select an object with more detail than a bounding sphere or box. An alternative method is to use a colour-based picking algorithm:

1. CPU: Each selectable object is given an unique, identifying, number.
2. CPU: The unique number is encoded either as an attribute or as a shader uniform.
3. GLSL: The number is encoded into a colour.
4. GLSL: The encoded colour is output from the fragment shader to a framebuffer-attached texture.
5. CPU: When the mouse is clicked or hovered, we can examine the texel in the texture that corresponds to the mouse cursor position.
6. CPU: We decode the colour back into the unique number, and mark our matching object as selected.

The most convenient method of doing this is to create an additional rendering pass. Each object that can be selected is re-drawn, using just the vertex positions (no need to compute lighting), and a uniform is updated with its unique code. The actual colour encoding will depend on the format of the texture, which we set with `glTexImage2D`. We can use any format, but let's start with using RGBA, because it's going to be easier to debug these values on the screen.

Example Scenario

Start with a simple scene that displays several objects. We are going to modify this to draw a new framebuffer with an unique colour for each object in the scene.



I drew 3 basic objects in different positions. I want to be able to click on each one separately. There is some overlap, which should be a good test. I just redrew the same VAO, but changed the model matrix uniform slightly for the additional 2 - you should be able to modify an earlier tutorial to effect this.

Create Second Framebuffer and Texture

As with the basic framebuffers article, we can create a framebuffer, and attach a texture to it. The texture must use the same format here, as elsewhere in the code - RGBA for the moment. Note that the data type is `GL_UNSIGNED_BYTE` per channel - 8 bits. This means 256 values per colour channel, or 0-255. We will need to know this later.

```
// create framebuffer
GLuint fb = 0;
 glGenFramebuffers (1, &fb);
 glBindFramebuffer (GL_FRAMEBUFFER, fb);

// attach depth texture to fb so that depth-sorting works
GLuint depth_tex;
 glGenTextures (1, &depth_tex);
 glActiveTexture (GL_TEXTURE0);
 glBindTexture (GL_TEXTURE_2D, depth_tex);
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_DEPTH_COMPONENT,
    width,
    height,
    0,
    GL_DEPTH_COMPONENT,
    GL_UNSIGNED_BYTE,
    NULL
);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// attach depth texture to framebuffer
glFramebufferTexture2D (
    GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depth_tex, 0);
// create texture to use for rendering second pass
GLuint fb_tex = 0;
 glGenTextures (1, &fb_tex);
 glBindTexture (GL_TEXTURE_2D, fb_tex);
// make the texture the same size as the viewport
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_RGBA,
    width,
    height,
    0,
    GL_RGBA,
    GL_UNSIGNED_BYTE,
    NULL
);
// textures will not work properly if you don't set up parameters
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

// attach colour texture to fb
glFramebufferTexture2D (
    GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, fb_tex, 0
);
// redirect fragment shader output 0 used to the texture that we just bound
GLenum draw_bufs[] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers (1, draw_bufs);

// bind default fb (number 0) so that we render normally next time
 glBindFramebuffer (GL_FRAMEBUFFER, 0);
```

Shaders

The shaders for rendering the colour pass are very simple - the first shader is the same as a simple positioning shader. Make sure that you set up the uniforms for each matrix as normal.

```
#version 400
in vec3 vp;

uniform mat4 P, V, M;

void main () {
    gl_Position = P * V * M * vec4 (vp, 1.0);
}
```

The fragment shader is very basic; it just takes the `unique_id` uniform, which we will normalise to a range between 0 and 1. I just set the alpha component to 1.0, so that it's easier to debug visually.

```
#version 400
uniform vec3 unique_id;
out vec4 frag_colour;

void main () {
    frag_colour = vec4 (unique_id, 1.0);
}
```

I'll assume that we have the basic shaders working, and that we compile these new shaders into a programme.

Encoding IDs as Colours

It's generally desirable to have an unique number for each selectable object. This could be its index in an array; allowing very quick random access look-up. We then need to convert the unique number (I'll assume an `int`) into an RGB colour. I won't use the alpha channel yet.

Remember that each colour channel in the texture has 1 byte, or 256 possible values. I'll use one channel as the "ones" column, one channel as the "256s" column, and the third channel as the "65536s" column. This gives us a huge number of unique IDs that we can have in just the RGB channels. I wrote a little function to do this sorting. At the end I divide each channel by 255 (the biggest value allowed), and convert it to a float in the range of 0.0 to 1.0 i.e. a proper colour value.

```
// encode an unique ID into a colour with components in range of 0.0 to 1.0
vec3 encode_id (int id) {
    int r = id / 65536;
    int g = (id - r * 65536) / 256;
    int b = (id - r * 65536 - g * 256);

    // convert to floats. only divide by 255, because range is 0-255
    return vec3 ((float)r / 255.0f, (float)g / 255.0f, (float)b / 255.0f);
}
```

I used my own `vec3` struct for returning 3 floats. Now, when I draw each object to the new framebuffer, I can convert its unique ID into a colour value, and update my uniform.

You can actually simplify this considerably if you just use the red colour channel. You can change the texture format so that it only has a red channel, with bigger size per pixel, but I suggest debugging the RGB version visually before you do this.

Drawing Unique Colours

Our drawing function to the framebuffer is going to resemble other framebuffer tutorials' code, except that we are not going to use the texture in a second pass. This function is going to redraw all my 3 objects. I need to update the matrices for the new shader programme - I do this for each one with the same translation values as in the normal scene. I'll also assume that you have successfully compiled the shader already, and called the programme `fb_sp`. Note that I use my encoding function 3 times, and update the ID uniform for each of the objects separately. Replace `point_count` with the number of vertices in your object's mesh.

```
void draw_picker_colours (mat4 P, mat4 V, mat4 M[3]) {
    glBindFramebuffer (GL_FRAMEBUFFER, fb);
    glViewport (0, 0, width, height);
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram (fb_sp);
    // set colour
    vec3 id = encode_id (255);
    glUniform3f (fb_sp_unique_id_loc, id.v[0], id.v[1], id.v[2]);
    glUniformMatrix4fv (fb_sp_P_loc, 1, GL_FALSE, P.m);
    glUniformMatrix4fv (fb_sp_V_loc, 1, GL_FALSE, V.m);
    glUniformMatrix4fv (fb_sp_M_loc, 1, GL_FALSE, M[0].m);

    glBindVertexArray (vao);
    glDrawArrays (GL_TRIANGLES, 0, point_count);

    // 2nd monkey
    id = encode_id (65280);
    glUniformMatrix4fv (fb_sp_M_loc, 1, GL_FALSE, M[1].m);
    glUniform3f (fb_sp_unique_id_loc, id.v[0], id.v[1], id.v[2]);
    glDrawArrays (GL_TRIANGLES, 0, point_count);

    // 3rd monkey
    id = encode_id (16711680);
    glUniformMatrix4fv (fb_sp_M_loc, 1, GL_FALSE, M[2].m);
    glUniform3f (fb_sp_unique_id_loc, id.v[0], id.v[1], id.v[2]);
    glDrawArrays (GL_TRIANGLES, 0, point_count);

    glBindFramebuffer (GL_FRAMEBUFFER, 0);
}
```

The `.m` notation is just the way that I access the array of floats inside my custom matrix structures. Note that I've chosen 3 different ID values to encode. Each one is a specific number; can you what colours they will

represent? I rebind the default framebuffer at the end, just to be safe. Call this function after drawing the regular pass, but before calling `SwapBuffers`.

Checking that it Worked

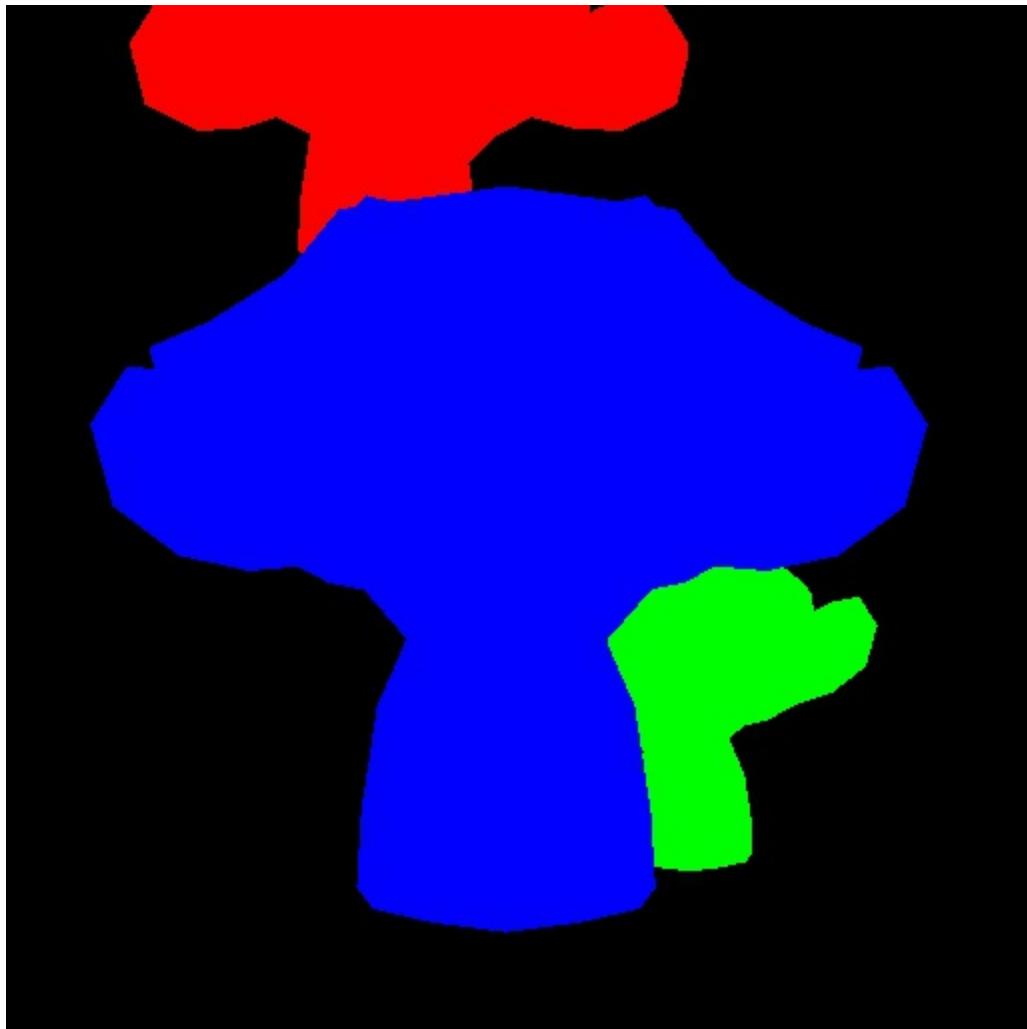
I added a global boolean flag, `bool debug_colours`; and set this to true when the space bar is held down. I'll use this as a debug mode. I added this code into my main loop to change the flag:

```
debug_colours = glfwGetKey (window, GLFW_KEY_SPACE);
```

You can modify the code at the top of the framebuffer drawing function, so that it would divert the output to the default framebuffer instead.

```
void draw_picker_colours () {
    if (debug_colours) {
        glBindFramebuffer (GL_FRAMEBUFFER, 0);
    } else {
        glBindFramebuffer (GL_FRAMEBUFFER, fb);
    }
    ...
}
```

Make sure that you get this debug view looking correct first - choose ID numbers that will definitely correspond to a colour, and check that this matches what you expect from hand calculations. According to my encoding, blue represents multiples of 1, green is multiples of 256, and red is multiples of 256*256. Each colour or column has 0-255 times its multiple. So full blue is 255*1, full green is 255*256, and full red is 255*256*256.



If I hold down the space bar then my colour picking image gets writing to the default framebuffer, instead of a texture, and shows this on-screen. I chose unique IDs that would correspond to colours red, green, and blue so that it would be easier to debug the demo.

Decoding and Selecting

Great, now we can get the mouse cursor position, read the pixel under it in the new framebuffer, and convert the values that we read back into our unique ID integer.

In GLFW you can check if the first mouse button is clicked. I'll use this in the main loop to tell when to read the buffer's texture. I'll firstly get the mouse cursor position from GLFW:

```
if (glfwGetMouseButton (window, 0)) {  
    glBindFramebuffer (GL_FRAMEBUFFER, fb);  
    double xpos, ypos;  
    glfwGetCursorPos (window, &xpos, &ypos);  
    int mx = (int)xpos;  
    int my = (int)ypos;  
    ...
```

This is using the GLFW 3.0 mouse cursor function, which returns the mouse cursor in pixels but as a `float`. Now we can use the `glReadPixels` function to sample the framebuffer pixels. This function is bit confusing. The Y-axis is almost certainly inverted, so we need to reverse the axis by subtracting the mouse Y position value from the vertical size of the viewport. We need to give it the same texture format parameters as we used when we set up the texture originally. It asks for a width and height of the pixel...erm...1,1. It also asks for a void pointer to some data. This is going to spit out 4 bytes; red, then green, then blue, then alpha. To capture this I'll first allocate an array of 4 bytes (`unsigned char` is one byte big). **Be sure to initialise this memory to zero**, or some "no hit" value, because the function will not modify the value if it does not succeed e.g. if the mouse cursor is outside the window area when the button is clicked.

```
...  
unsigned char data[4] = {0, 0, 0, 0};  
glReadPixels (mx, g_g1_height - my, 1, 1, GL_RGBA, GL_UNSIGNED_BYTE, &data);  
...
```

Okay, now we need a decode function that takes 3 bytes (we can ignore the alpha value - this will always be 255, because we output 1.0 from our fragment shader. Our function should return an integer.

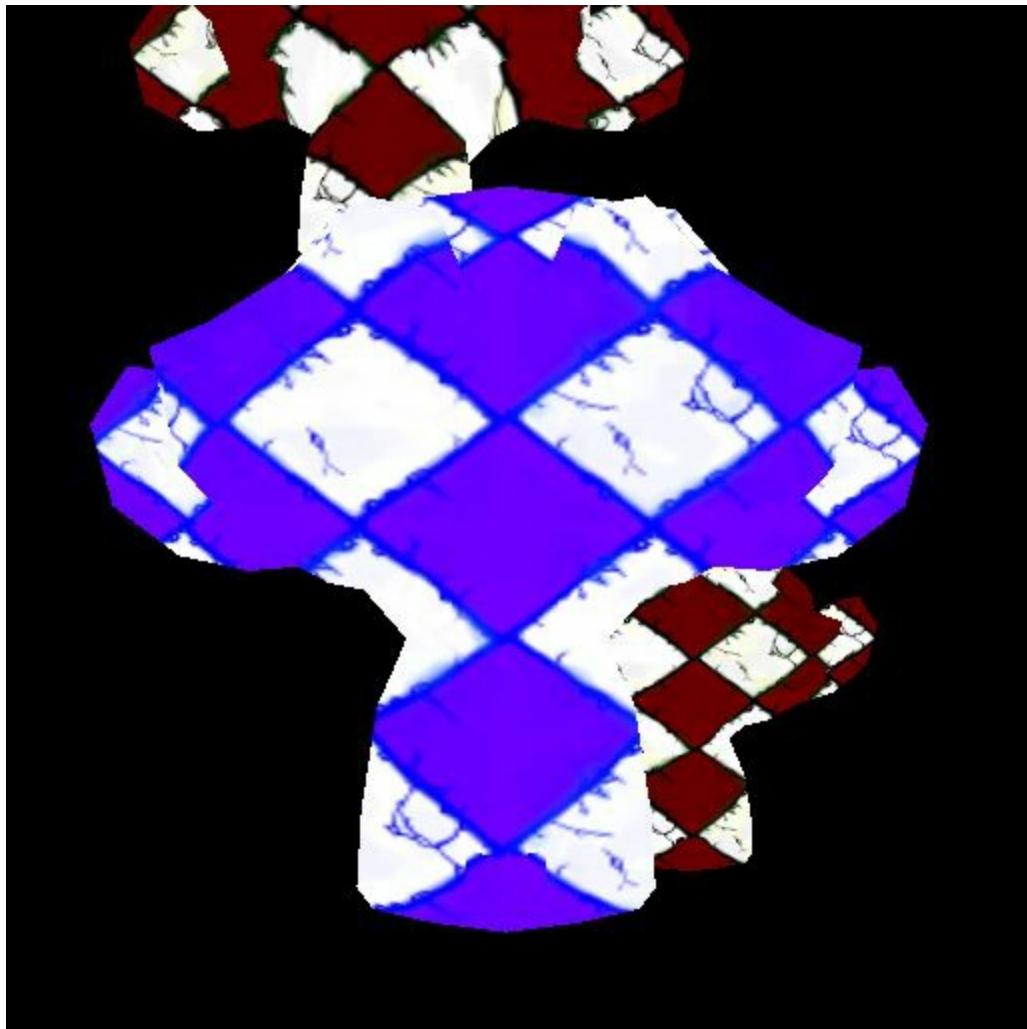
```
...
int id = decode_id ((int)data[0], (int)data[1], (int)data[2]);
printf ("%i,%i,%i means -> id was %i\n",
       data[0], data[1], data[2], id);
glBindFramebuffer (GL_FRAMEBUFFER, 0);
}

int decode_id (int r, int g, int b) {
    return b + g * 256 + r * 256 * 256;
}
```

Note that I returned the `b` (blue) coefficient, plus the `g` coefficient, multiplied by 256, plus the `r` coefficient multiplied by 256*256. If our original value is 255, we should have 0,0,255, but if it was 256, we should have 0,1,0 - the `g` is the count of 256s.



Once I had the ID number of what the mouse clicked on, I checked if this matched each item before I drew it. If so I send in a uniform value to modify the colour. In this case, I clicked on the top-left object, which returned an ID of 2, and so I sent in a modified colour uniform before drawing object number 2.



Next I clicked on an overlapping part of the middle object. It selected object number 0. In this case I colour the new object, and revert object 2 back to the default colour.

Improvements

You can try changing the memory and format of the texture to suit your application, but this probably isn't a big deal. Reducing the texture dimensions will affect your clicking accuracy. You might be able to combine both rendering passes together by using the texture as a second fragment shader output from the regular drawing pass, but this is also probably not a big deal because you only have a performance hit when you actually click in my demo code. If you want to highlight an object when the mouse hovers over it, it might be worth consider an optimisation, but I wouldn't worry unless it becomes really slow.

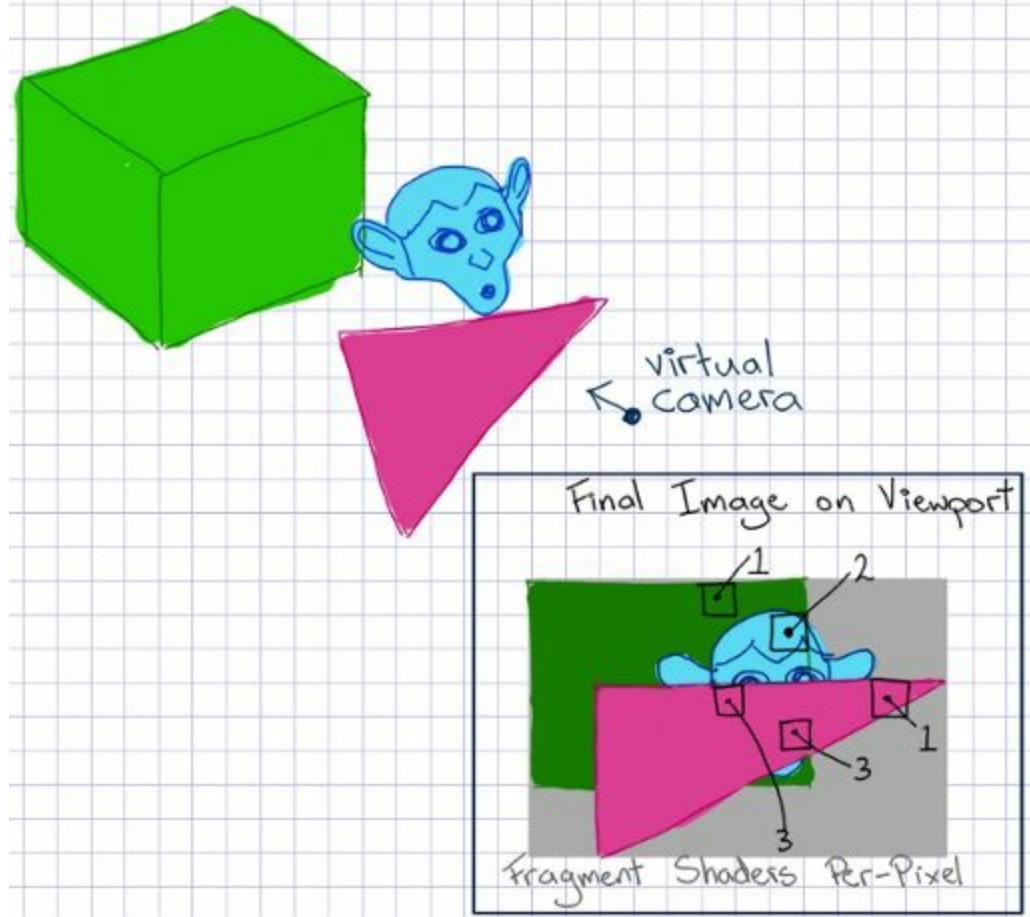
Deferred Shading

Overview

Deferred shading is an exploitation of rendering to a texture. We know that we can store outputs from a fragment shader in a texture. What if, instead of the fragment colour, we stored geometry information; things like normals and positions, in output textures? We can attach several textures to hold different outputs from a fragment shader, so we can capture several variables, which we can later use to do some more interesting post-processing techniques. This collection of textures is commonly called a "geometry buffer", or **G-buffer**. With a G-buffer we can do things like Phong lighting in a second pass - in screen space. You can see that we have **deferred** the processing to a second pass. If you're compositing together several post-processing techniques, you can share the G-buffer between all of them, which minimises your overhead costs.

The most popular deferred shading technique is **deferred lighting**. I've done quite a bit of reading and experimentation with this lately, as there's a lot of hype behind it. The web will show you demos where scenes are rendered with hundreds of light sources at once, which is amazing, because our single-pass Phong demos start slowing down after we use more than about 3 lights at once! After doing a lot of digging in to it, it **turns out to be a bit of a sham**, and I'll explain why. As we know, tricks in real-time computer graphics are not always bad, and it might be a sham that suits your project! We'll implement our own deferred lighting demo, and you can decide for yourself if the advantages outweigh the disadvantages for your virtual world.

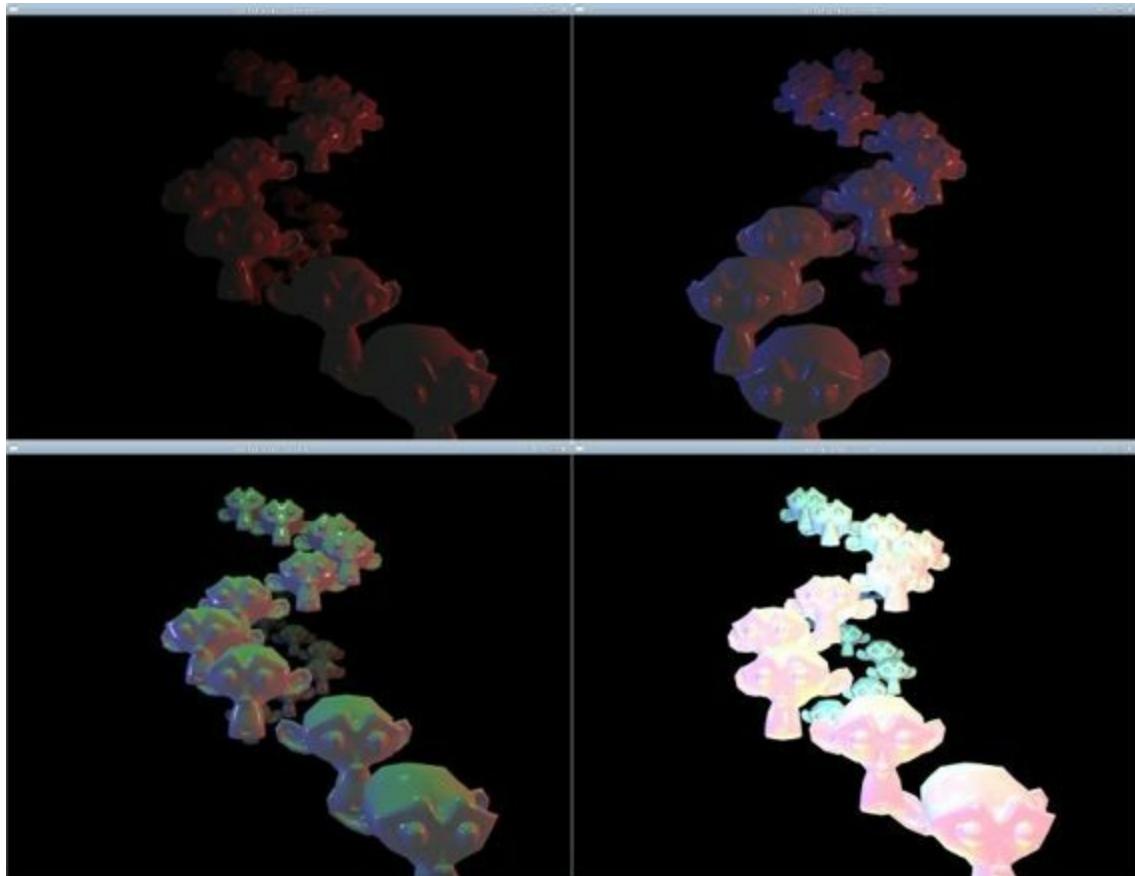
Meshes in a 3d Scene



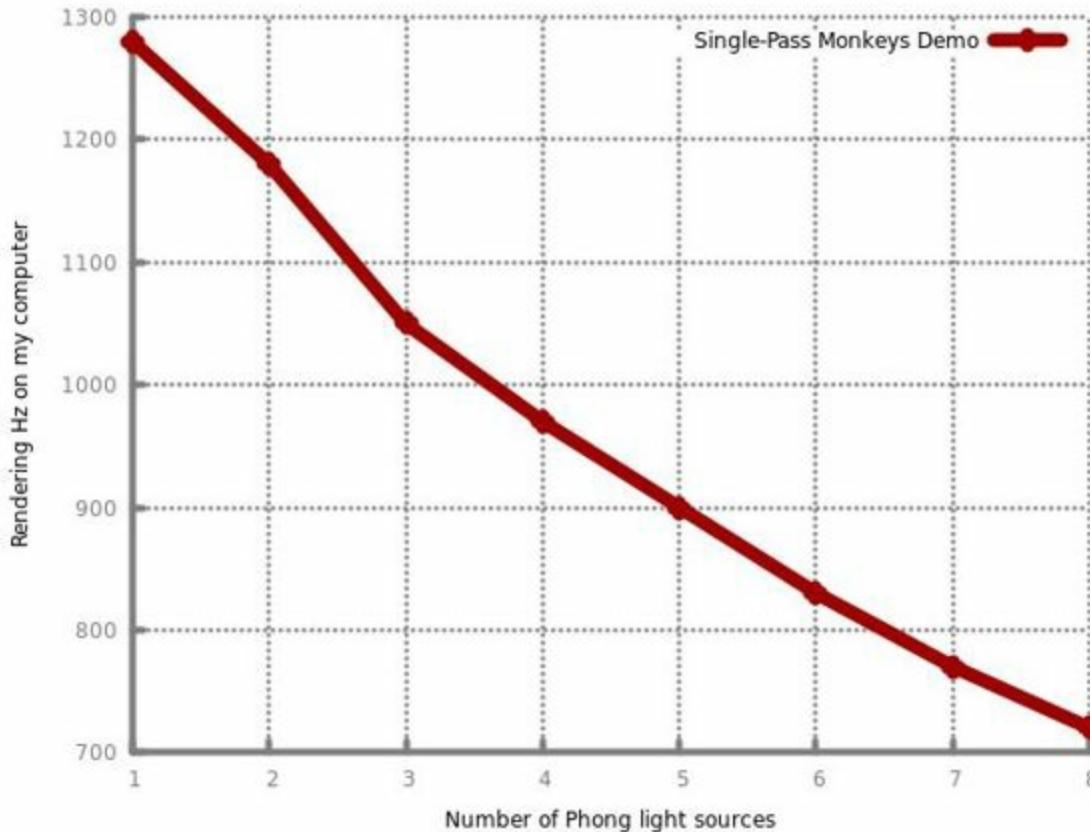
We tend to have lots of overlapped and redundant fragment shaders computed every frame. If we compute complex lighting in each fragment shader then it becomes quite costly. Deferred shading just writes all the variables used to textures, without doing the actual computation. The closest fragment's variable will overwrite the others. A second pass computes lighting once per-pixel, using the variables which we deferred from the first pass. You can see that if we can cut our overall fragment shader computation to a small fraction of the original in this way, then we can add many times the number of scene lights and run at the same speed.

The main argument for deferred shading is that you tend to compute a lot of fragment shaders that are overlapped by other geometry, and never actually contribute to the final image - a waste of processing. This can get quite expensive when fragment shaders get more complex. If we defer this processing, we minimise the cost of computing the hidden fragments. Remember that, because of depth testing, the textures that hold variables from fragments farther away will be overwritten by those of closer fragments. In theory it means we can do things like lighting only once per screen pixel, rather than once per fragment. In reality, adding a second rendering pass is

enormously expensive, and even on complex scenes you never save as much as the new overhead cost.



In a completely un-scientific test I made a demo rendering 20 Suzanne meshes. You can select the number of Phong light sources that it uses in the fragment shader. The images show 1 lights (top-left), 2 lights (top-right), 3 lights (bottom-left), and 8 lights (bottom-right). I thought I'd make a little chart of the frame-rate (below).



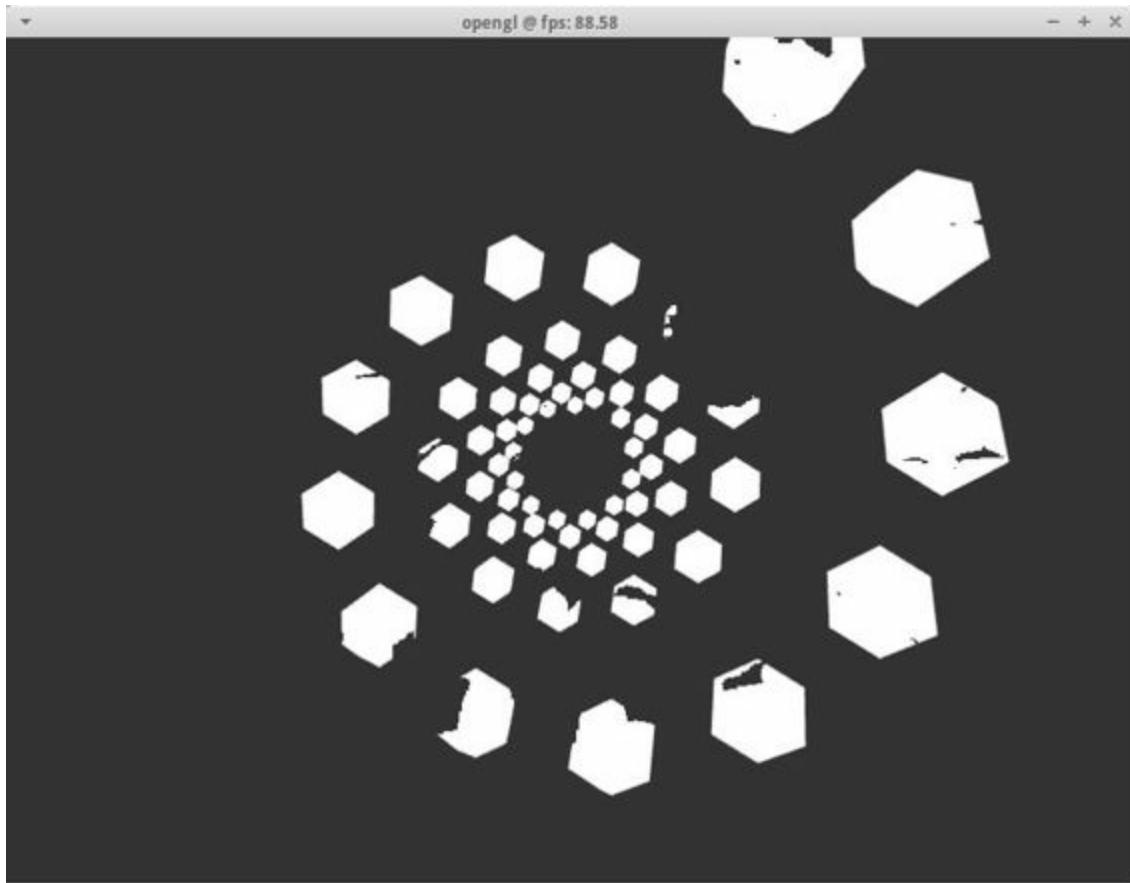
I recorded my frames-per-second (Hz) in a 1024x768px window on my (rather fast) machine for different numbers of lights. You can see that there's a linear cost that for each light added (basically just the cost of the extra `normalize()` call), so if I wanted to keep going, adding more lights, it would start to become impractically expensive once combined with other effects and techniques.

In the screen-captures above, the image with 8 lights is interesting. All of the surfaces appear whited-out. Why? Because lights are additive. If we add a red, and a blue, and a green, and another red... and so on, we end up with something equal, or greater to RGB colour (1, 1, 1) - white. But the demos that we've seen of deferred lighting tend to show a scene with a little bit of blue here, red there, purple there, green there...*ad infinitum*. Why do you think that is? Why don't they all just blend into a white colour? In fact, I implemented the exact same demo with the basic deferred lighting algorithm, and it did look exactly the same, except ran at half the speed! I didn't get an advantage with the deferred technique over the single-pass technique until I had more than 48 lights - and by that stage it was going too slowly to be useful, which means that the argument about redundant fragments is hokum - the overhead cost of the second pass is far too high. What's going on here? Where's my super-fast technique with hundreds of little coloured lights?

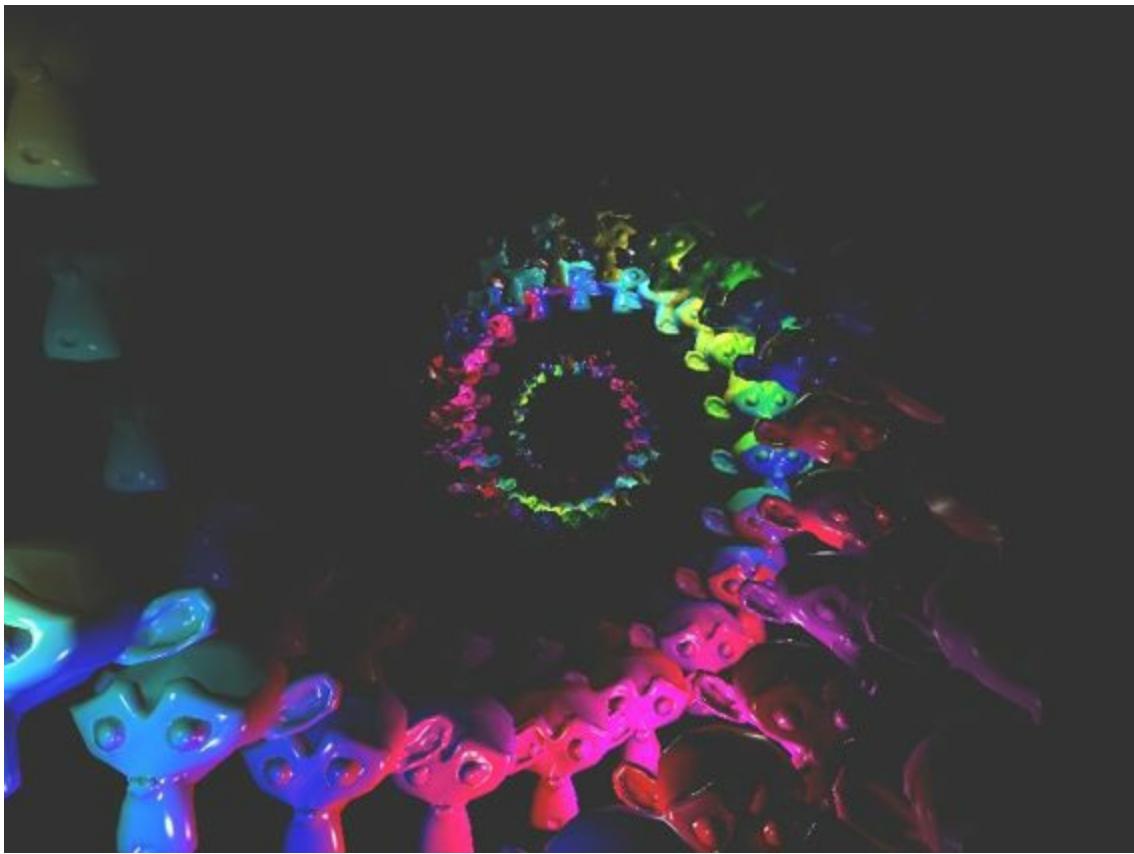
As it turns out, the key is that all light sources in deferred rendering must have a definite cut-off distance. The cut-off distance has to be small enough, in fact, that very few of our scene lights ever overlap in view. When they overlap, it starts to slow down in the same way. **The truth of the matter is that we're not rendering with hundreds of scene lights at all** - we're only rendering with 2-3 for any one particular fragment. This is the same as only having 3 scene lights in your single-pass shader, except that we can be precise to the fragment, rather than the same 3 lights for each whole mesh. The downside to this is that it's even more unrealistic than our already unrealistic lighting model - there's no such thing as a depth-limited light in reality - photons don't just run out of juice unless they hit something! Of course the lights should blend together into a whitish colour. Perhaps you can assume that your lights will be far enough apart to not contribute a significant amount to other parts of the scene. Lots of little coloured bits looks more fun, and maybe that's what you're going for. There are several methods for implementing this:

- Basic deferred shading. Compute lighting as per normal, but now that we've eliminated the cost of hidden fragments, we can add more lights. This method costs more than it saves, but might be justified if the G-buffer is shared with other post-processing shaders.
- In a second pass, draw each point light source as a sphere. Anything under the sphere gets lighting applied to it. Blend the result with any previous results from other, intersecting lights. This method suffers from the number of draw calls (1 for each light). Additive blend of overlapping lights is also a bit expensive.
- Divide the screen into a grid of 2d tiles. Use a compute shader (OpenGL 4.3) to make a list of lights that affect each tile, by checking if their radius is in the viewing area of the tile. Apply lighting to each tile using its list of lights.
- Use a stencil buffer to colour only the radius coloured by each light. I haven't tried this method, but popular wisdom says that the stencil buffer adds more overhead than it saves.

I'm going to use the spheres approach because I don't think that my driver supports OpenGL 4.3 yet.

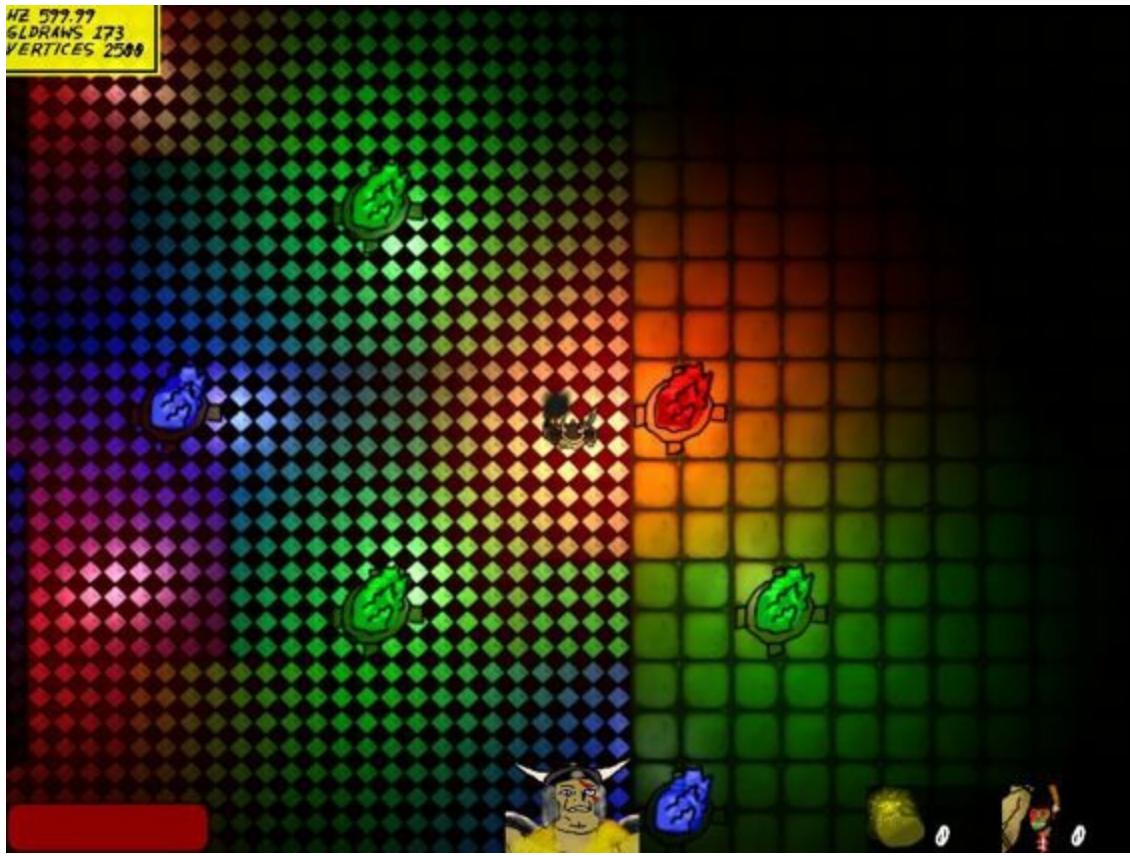


I used the "draw each light as a sphere mesh" deferred lighting approach. Here I'm drawing each "sphere" mesh in the second pass, but rendering them all full white to show where they are in the scene, and make sure that there aren't too many overlaps. I have a uniform variable for scaling the light meshes to match the intended radius. The black bits inside the spheres are parts where no monkey is visible and the background is showing through. To add colour we will run a fragment shader for each pixel covered by each light sphere. It will read the normal, surface position, etc. from the G-buffer, and compute the Phong equations in screen space.



I implemented the "draw each light as a sphere mesh" method of deferred lighting, using the same scene as in the earlier screen shots. There are 64 coloured lights here and 256 Suzanne meshes. The trick is to place the lights in such a way that very few overlap in the same place. If the lights' radius is too large, then the processing rate drops down very quickly. In a more realistic scene the lights should really blend into more of a white colour...but this is pretty.

Compare this to the more basic single-pass Doom 3 method of just using the closest 3 lights when rendering each mesh. Quickly finding the closest 3 lights can be tricky if you have hundreds of lights in a scene - we'd need to make a custom data structure to keep this fast in a larger virtual world. We can still have the same number of light sources as with deferred lighting though, and we don't need artificial cut-off distances. So we can actually say that, rather than being an optimisation, deferred lighting is actually a kind of alternative scene management tool for limiting the number of lights used for each fragment shader.



In a little game that I'm working on I implemented the "use the closest 3 lights" method, instead of deferred lighting. We can still have a large number of lights in the scene, but it requires a data-structure (I used a 2d map) to quickly work out the closest lights. If you look closely, you can spot a tile half way between the bottom green lights, that should be illuminated by the blue and red lights, but isn't.

Deferred Lighting Disadvantages

1. Very high overhead cost due to second pass and texture sampling
2. Built-in anti-aliasing of mesh edges no longer available (as with any post-processing technique)
3. Light sources must have a constrained cut-off distance (unrealistic)
4. Virtual world must be designed such that too many virtual lights will not overlap (design limitation)
5. Considerable additional code infrastructure to be built and maintained (added complexity, development cost, reliability hit).
6. Transparency does not work without additional processing

Deferred Lighting Advantages

1. Accurate at assigning lights to the pixel level, rather than the mesh level
2. Lighting is only computed for fragments that are visible
3. The scene lights from the virtual world that contribute to each fragment are managed automatically
4. Variables stored in G-buffer can be shared with other post-processing techniques that need them, minimising their overhead.

As with all post-processing techniques, OpenGL's built-in multi-sample anti-aliasing won't work, because we've lost information about where the edges of the primitives are by the time we finish rendering the second pass. The usual work-around is to implement a shader-based anti-aliasing algorithm. Remember that we are only storing the closest fragment's data in the G-buffer. Transparency requires blending of several fragments, so this won't work any longer.

If we're rendering spheres to the second pass, then any lights out of frustum can be ignored with a quick test. Clipping should get rid of any remaining light-sphere geometry that isn't in view.

Right, that was a long overview and opinion piece, but I wasn't satisfied with the quick, over-hyped, and under-explained pieces given elsewhere, so I thought I'd share all my findings. Let's build one!

Building the G-Buffer

In the first pass I have a fairly basic shader to render each Suzanne mesh.

First Pass Vertex Shader

```
#version 400

layout (location = 0) in vec3 vp;
layout (location = 1) in vec3 vn;

uniform mat4 P, V, M;

out vec3 p_eye;
out vec3 n_eye;

void main () {
    p_eye = (V * M * vec4 (vp, 1.0)).xyz;
    n_eye = (V * M * vec4 (vn, 0.0)).xyz;
    gl_Position = P * vec4 (p_eye, 1.0);
}
```

I transform the vertices into clip space as per normal, and output the variables needed for Phong lighting to the fragment shader; `p_eye`, and `n_eye` - the surface position, and surface normal, respectively, in eye space. This is nothing that we haven't done already.

First Pass Fragment Shader

```
#version 400

in vec3 p_eye;
in vec3 n_eye;

/* force output variables to use specific buffers with the 'layout' keyword */
layout (location = 0) out vec3 def_p;
layout (location = 1) out vec3 def_n;

void main () {
    def_p = p_eye;
    def_n = n_eye;
}
```

The fragment shader is interesting because it doesn't really do anything. It

just takes the 2 input variables and passes them out again. We do have something new here - we have 2 outputs from a fragment shader. Normally we'd just output the fragment colour, as a `vec4` but here we are outputting 2 `vec3` variables. We'll set up a framebuffer next, where we'll specify that each one should write to a separate RGB texture - we don't need a fourth channel here, so RGBA is not necessary.

Binding Multiple Framebuffer Textures

We will set up a secondary framebuffer in the same way as other post-processing techniques. The tricky bit here is making the correct texture line up with the correct shader outputs. Frankly OpenGL's interface for this is a confusing mess, but we can dissect it. First I'll create the framebuffer `fb`, and 2 new textures; one to hold the position outputs, which I'll call `fb_tex_p`, and one to hold the normals outputs, which I'll call `fb_tex_n`.

```
GLuint fb = 0;
 glGenFramebuffers (1, &fb);
 glGenTextures (1, &fb_tex_p);
 glBindTexture (GL_TEXTURE_2D, fb_tex_p);
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_RGB16F,
    g_gl_width,
    g_gl_height,
    0,
    GL_RGB,
    GL_UNSIGNED_BYTE,
    NULL
);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glGenTextures (1, &fb_tex_n);
 glBindTexture (GL_TEXTURE_2D, fb_tex_n);
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_RGB16F,
    g_gl_width,
    g_gl_height,
    0,
    GL_RGB,
    GL_UNSIGNED_BYTE,
    NULL
);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

We've seen this process before, except I've changed some of the parameters for the `glTexImage2D` function. I've told it to use `GL_RGB` format, instead of `GL_RGBA`, because we're only outputting `vec3` this time, instead of `vec4`.

I've also specified the data format as `GL_RGB16F`. If you read the [function documentation](#) you can see that there are a few data formats with "floating point" storage. I didn't use `GL_RGB32F` because I don't need that much accuracy for a normal vector, and I'm not concerned enough for the position's accuracy. You can use 32-bit floats but it will make your programme run a bit more slowly.

Also note that I've disabled bi-linear filtering, because we should have a 1:1 pixel match between first and second passes. Now we just have to attach these 2 textures to the framebuffer...

```
/* depth texture attachment */
GLuint depth_tex;
 glGenTextures (1, &depth_tex);
 glBindTexture (GL_TEXTURE_2D, depth_tex);
 glTexImage2D (
    GL_TEXTURE_2D,
    0,
    GL_DEPTH_COMPONENT32F,
    g_gl_width,
    g_gl_height,
    0,
    GL_DEPTH_COMPONENT,
    GL_UNSIGNED_BYTE,
    NULL
);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// attach depth texture to framebuffer
glFramebufferTexture2D (
    GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depth_tex, 0);
```

You can see that I've attached the positions texture to attachment number 0, and normals to attachment number 1. These numbers don't actually correspond to anything definite yet - they actually mean the [indices in an array](#) that we will set up shortly. Yes, this is a silly system. We add another texture set to capture the depth values. Remember that this means "enable depth-sorting for the framebuffer".

Now let's set up that array. Remember that we want positions in index 0, and normals in index 1:

```
Glenum draw_bufs[] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers (2, draw_bufs);
```

If you're lucky, your shaders will compile so that the positions are assigned as output location 0 - corresponding to `GL_COLOUR_ATTACHMENT0`, which we've just set up to correspond to our texture called `fb_tex_p`. If you'd like to play it safe, you can add a `layout(location = output index)` to your fragment shader outputs, to make sure that the outputs are not mixed up:

```
layout (location = 0) out vec3 def_p; // "go to GL_COLOR_ATTACHMENT0"
layout (location = 1) out vec3 def_n; // "go to GL_COLOR_ATTACHMENT1"
```

If you'd like to visually test that your G-buffer is being stored in the correct textures, then you'll probably want to switch your fragment shaders to output `vec4` values, and your texture format to `GL_RGBA`. Then you can switch back when you're satisfied that all is in order.

Implementing the "Sphere Meshes" Method

First we need an actual sphere (or other interesting shape) to represent our light's coverage volume. I just exported a sphere primitive from Blender that was dimensions -1:1 in all axes. There would be nothing stopping you from having a cone or even a monkey-shaped light. We can make a matrix representation for each light based on its world position, and it's scale. This will just be used in the vertex shader in place of a model matrix for the sphere mesh. I made all my lights cover a -5 to +5 volume which means that very few intersected in my scene. I have 64 scene lights, and I'll use some functions that I wrote to build a matrix of floats. I have an array called `lp` for light world positions, which I just generated from some `sin()` function calls, but you can create these any way that you like:

```
vec3 lp[64];
for (int i = 0; i < 64; i++) {
    float x = -sinf ((float)i*0.5f) * 5.0f; // between +- 10 x
    float y = cosf ((float)i*0.5f) * 5.0f;
    float z = (float)-i * 2.0f + 10.0; // 1 light every 2 meters away on z
    lp[i] = vec3 (x, y, z);
}

mat4 lM[64];
const int num_lights = 64;
const float radius = 5.0f;
for (int i = 0; i < num_lights; i++) {
    lM[i] = scale (identity_mat4 (), vec3 (radius, radius, radius));
    lM[i] = translate (lM[i], lp[i]);
}
```

I created 2 more arrays for the lights - `ld` for their diffuse colour, and `ls` for their specular colour. You can define these in any way that you like.

Right, I'll assume that you can render the first pass, using the framebuffer that we created, and that you have the 2 shader outputs stored in textures. It should look something like this:

```
glBindFramebuffer (GL_FRAMEBUFFER, fb);
glClearColor (0.0f, 0.0f, 0.0f, 1.0f);
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glDisable (GL_BLEND);
```

```

glEnable (GL_DEPTH_TEST);
glDepthMask (GL_TRUE);

glUseProgram (first_pass_sp); /* our first pass shaders */
glBindVertexArray (monkey_vao);

/* virtual camera matrices */
glUniformMatrix4fv (first_pass_P_loc, 1, GL_FALSE, g_P.m);
glUniformMatrix4fv (first_pass_V_loc, 1, GL_FALSE, g_V.m);

/* Update model matrices and draw each monkey in a loop here */
for (int i = 0; i < number_of_monkeys; i++) {
    glUniformMatrix4fv (...);
    glDrawArrays (...);
}

// deferred pass
glBindFramebuffer (GL_FRAMEBUFFER, 0);
glClearColor (0.2, 0.2, 0.2, 0.0f); // added ambient light here
glClear (GL_COLOR_BUFFER_BIT);

glEnable (GL_BLEND);
glBlendEquation (GL_FUNC_ADD);
glBlendFunc (GL_ONE, GL_ONE); // addition each time

glDisable (GL_DEPTH_TEST);
glDepthMask (GL_FALSE);

glActiveTexture (GL_TEXTURE0);
glBindTexture (GL_TEXTURE_2D, fb_tex_p);
glActiveTexture (GL_TEXTURE1);
glBindTexture (GL_TEXTURE_2D, fb_tex_n);
glUseProgram (second_pass_sp);
glBindVertexArray (sphere_mesh_vao);

const int num_lights = 64;
for (int i = 0; i < num_lights; i++) {
    glUniformMatrix4fv (def_M_loc, 1, GL_FALSE, lm[i].m); // light's model matrix
    glUniform3f (def_lp_loc, lp[i].v[0], lp[i].v[1], lp[i].v[2]); // world position
    glUniform3f (def_ld_loc, ld[i].v[0], ld[i].v[1], ld[i].v[2]); // diffuse colour
    glUniform3f (def_ls_loc, ls[i].v[0], ls[i].v[1], ls[i].v[2]); // specular colour
    glDrawArrays (GL_TRIANGLES, 0, sphere_mesh_point_count);
}

glfwSwapBuffers();

```

I start by binding the default framebuffer, and clearing the background colour. You can add an "ambient term" here by setting the background colour. Why? Because we are going to add each light's colour on top of the existing pixels' colours. Therefore the base colour can work like an ambient light term. We don't need a depth buffer at this point, so I only clear the colour buffer. Next I set up blending, with the `GL_FUNC_ADD` function and a `GL_ONE, GL_ONE` relationship. This just means that, every time we write to a pixel in the colour buffer it equals `pixel_colour = 1 * existing_colour + 1 * new_colour`.

I bind both of my textures to 2 texture slots, and then use my second-pass shader, which we will write shortly. I also bind my sphere's VAO, which I called `sphere_mesh_vao`.

The next part is interesting - I run a `for` loop, with one iteration per light in the scene. I pass in the model matrix that I made for each light. I also send in 3 more uniforms; the light position, diffuse colour, and specular colour, which I'll use in the fragment shader. You could probably extract your light position from the model matrix' uniform if you wanted to save a bit of bandwidth. Finally, I draw each sphere to the screen. As we know, drawing lots of small meshes isn't making very good use of the GPU's parallelism, so we can say that this is the downside of the "light sphere mesh" method.

Second-Pass Shaders

At this point I suggest creating some simple shaders that position your light meshes in clip space. Don't forget about sending uniforms for the view and projection matrices to your new shader. We will use the light matrix from the previous section as the model matrix. You can set your fragment shader output to some fixed colour, and test that your light meshes are a sensible position and size, as I did in the earlier screen capture (with the white spheres). There are a couple things to look out for when testing like this:

1. Are your lights in sensible positions? Tweak the matrices to suit.
2. Is the radius of each light too large - too many overlaps?
3. Are you drawing lights that aren't in view? Count them! If this is the case you'll need to consider a culling algorithm to eliminate these draw calls.

I used this vertex shader:

```
#version 400

in vec2 vp;
uniform mat4 P, V, M;

void main () {
    gl_Position = P * V * M * vec4 (vp, 0.0, 1.0);
}
```

Don't forget to get the uniform locations for the perspective, view, and light matrices, which we are using from the previous section's code. My fragment shader is as follows. You will recognise the uniforms for the view matrix, the two textures, and the light specular colour, diffuse colour, and position. Do not forget to get the uniforms for the textures, and set their uniform values to 0, and 1 - otherwise they will both sample from active texture number 0!

Easy to overlook. Note that I only have a buffer of 2d vertex positions, `vp`, for my quad - I thought I'd introduce a different method for generating texture coordinates inside the fragment shader. You could of course, also work them out in the vertex shader by scaling the vertex positions.

```
#version 400

uniform mat4 V;
uniform sampler2D p_tex;
uniform sampler2D n_tex;
uniform vec3 ls;
uniform vec3 ld;
uniform vec3 lp;

out vec4 frag_colour;

vec3 kd = vec3 (1.0, 1.0, 1.0);
vec3 ks = vec3 (1.0, 1.0, 1.0);
float specular_exponent = 100.0;

// continued...
```

I hard-coded the material properties, `kd` (diffuse reflection), `ks` (specular reflection), and the specularity exponent. Let's continue the shader, where we will write a function to do Phong lighting. This is nothing special. Ideally you'll want to diminish/attenuate your light so that it's completely black by the time that it gets to the edge of the sphere - otherwise you'll get a very visible cut-off. The `phong()` function takes 2 input parameters - which we'll sample from our textures before calling the function. These are the surface position, and normal, in eye space, as we stored in the textures. I've set the cut-off distance to match my scaled mesh radius of 5, and used `min` to clamp my attenuation factor between 0.0 and 1.0. Actually, my sphere is not a perfect shape, but more of a hacked potato shape, so I still get some visible edges - you'll need to bring your attenuation radius back a bit to get it just right. There will be fiddling!

```
vec3 phong (in vec3 p_eye, in vec3 n_eye) {
    vec3 light_position_eye = vec3 (V * vec4 (lp, 1.0));
    vec3 dist_to_light_eye = light_position_eye - p_eye;
```

```

vec3 direction_to_light_eye = normalize (dist_to_light_eye);

// standard diffuse light
float dot_prod = max (dot (direction_to_light_eye, n_eye), 0.0);
vec3 Id = ld * kd * dot_prod; // final diffuse intensity

// standard specular light
vec3 reflection_eye = reflect (-direction_to_light_eye, n_eye);
vec3 surface_to_viewer_eye = normalize (-p_eye);
float dot_prod_specular = dot (reflection_eye, surface_to_viewer_eye);
dot_prod_specular = max (dot_prod_specular, 0.0);
float specular_factor = pow (dot_prod_specular, specular_exponent);
vec3 Is = ls * ks * specular_factor; // final specular intensity

// attenuation (fade out to sphere edges)
float dist_2d = distance (light_position_eye, p_eye) / 5.0;
float atten_factor = max (0.0, 1.0 - dist_2d);

return (Id + Is) * atten_factor;
}

```

Now finally, we just call our Phong lighting after sampling the texture. Just for a change, I didn't send in any texture coordinate, so I'm pulling them out of the built-in `gl_FragCoord` variable. These are in pixels so I divide by my viewport dimensions to get between 0.0 and 1.0.

```

void main () {
    vec2 st;
    st.s = gl_FragCoord.x / 800.0;
    st.t = gl_FragCoord.y / 600.0;
    vec4 p_texel = texture (p_tex, st);
    // skip background
    if (p_texel.z > -0.0001) {
        discard;
    }
    vec4 n_texel = texture (n_tex, st);

    frag_colour.rgb = phong (p_texel.rgb, normalize (n_texel.rgb));
    frag_colour.a = 1.0;
}

```

I made a little crude escape-hatch here to abandon rendering if there was no mesh information sampled i.e. it was a fragment over the background. This doesn't automatically give you a speed-up because the GPU renders fragments in groups. If the entire group is discarded then you'll get a speed-up, which I do in my little demo! I re-normalise the surface normal here, which seems to make a difference to the final image.

Optimisations and Improvements

There are 4 major drawbacks to this approach to deferred lighting:

1. I'm sending in a lot of uniforms every frame
2. Drawing each light as a separate mesh is not making good use of the GPU's parallelism
3. Using GL blend functions to add up all the lights is quite expensive
4. You probably need to do some frustum culling or use a scene manager to optimise-out lights that won't be drawn.

You could probably reduce the number of uniforms being sent in by putting all your light variables into uniform buffers. On that note you might also be able to batch many of your lights together into a single set of vertex buffers - reducing the number of draw calls. Ultimately you might want to look at another deferred lighting method. I haven't found a good implementation of the screen-space tiling method yet, but when I do I'll write that up too!

Texture Projection Shadows

Overview

We have some efficient algorithms for calculating lighting in 3d, but these do not account for objects blocking or occluding the light - no shadows are cast. There are several algorithms for casting shadows in real-time graphics. All of them have different (major) drawbacks, so the current generation of video games tend to use a combination of effects for rendering shadows.

Modern "shadow mapping" or "texture shadows" is an implementation of Lance Williams' algorithm "Casting Curved Shadows on Curved Surfaces" (1978). The original paper is very easy to read and outlines most of the problems, some workarounds, and the main costs of the algorithm. In other words it's worth reading before you start. The idea is that we create a new virtual camera, which shows the scene from the point of view of the light. Ideally we have a spotlight so that we can build a viewing frustum pointing in a particular direction.

- baked shadow/light maps - pros: best visual quality - cons: not dynamic, memory cost of textures
- projected texture shadows - pros: dynamic, easy to soften - cons: many visual artifacts, tricky to set up, extra rendering pass
- volumetric shadows - pros: dynamic, high quality - cons: computational expense, hard to set up, sharp edges

Texture projection shadows create **another virtual camera** for each light source. If we assume a directional light, then this means another view and projection matrix. The scene is drawn for the light source, but only the depth (**Z-distance of each fragment from the light**) is written with the fragment shader. This **depth map** is captured in a texture using another framebuffer. It's going to look like a greyscale image of the scene, where the darker objects are closer to the light source. We finally draw the scene as normal, and find the texel in the depth map that matches each vertex point. We know

that if a point is farther away from the light than the matching texel value then it is blocked, and we shade it dark.

Setup

1. create virtual camera for light
2. create framebuffer with attached depth texture for shadow map
3. create shader that draws each object in scene (no colour required)
4. modify shaders for all meshes that should receive a shadow. calculate shadow map texture coordinates and check for shadow

If there are several shadow-casting lights then you will need to repeat this procedure for each light so that you have 1 shadow map for each light.

Rendering

1. bind the new framebuffer
2. use the shader as above, and send in light's camera matrices as uniforms
3. bind VAO and draw every object that should cast a shadow
4. bind the default framebuffer
5. use the appropriate shader
6. if mesh should receive shadows then activate matching texture slot and bind depth map texture
7. if the mesh should receive shadows then also send in uniforms for each light's virtual camera as well as the uniforms for the main virtual camera
8. bind VAO for every object in scene. draw as normal

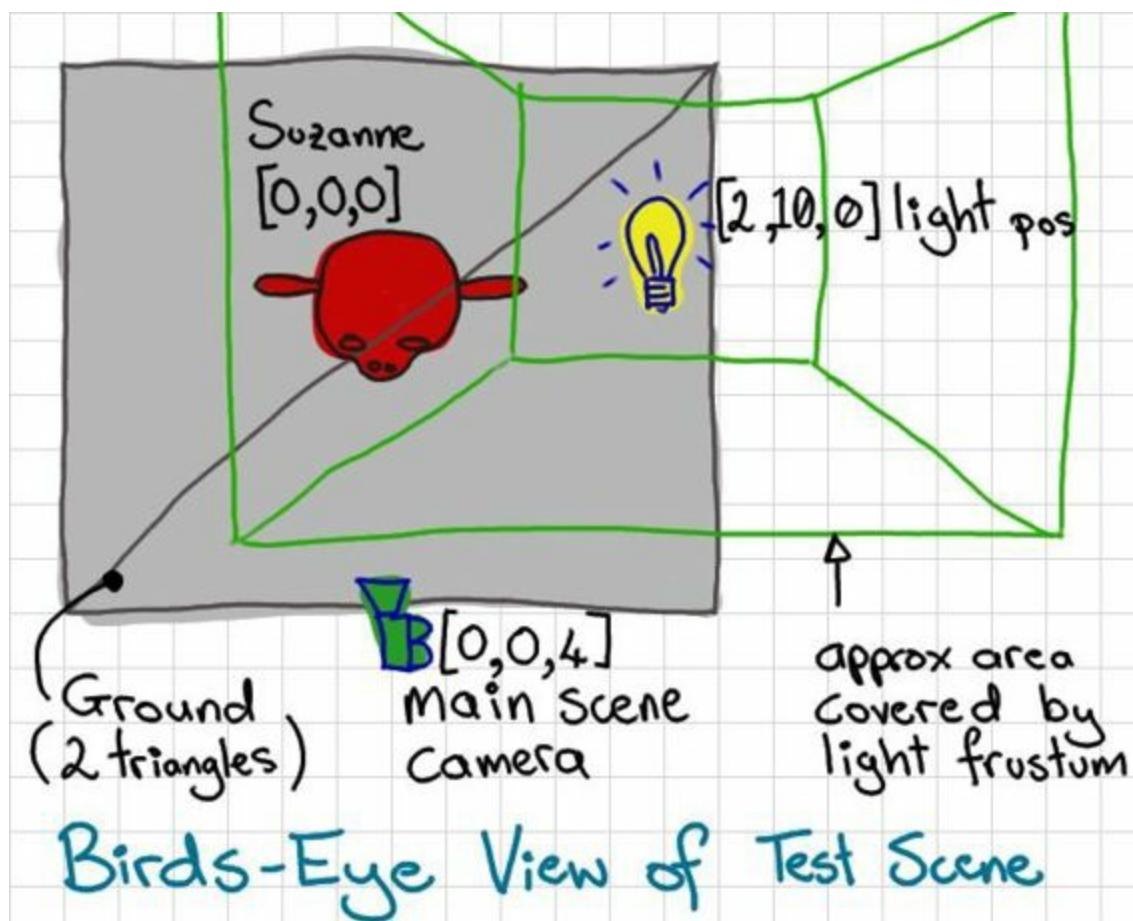
We will also add in another step to draw the depth buffer into a square on the screen. This will help us to debug any problems.

Before You Start

There are many tricky conventions to get right with texture shadows in OpenGL. I strongly suggest that you become familiar with **simple framebuffer** techniques before starting on texture shadows. You must understand how switching framebuffers works. You also need to have mastery of **virtual camera basics** (projection and view matrices), updating shader **uniforms**, and **switching between shaders**. We will also be using some **2d quad rendering**. Finally, we will be doing a lot of **texture sampling**, so you need to know how texture coordinates work. If you're unsure of any of these techniques or how to use the associated GL functions then it's going to be very difficult to implement shadow mapping.

Step 1: Set Up Light's Camera. Test It

Start with a simple demo that loads and displays a mesh. I used the Suzanne head, and also added in 2 triangles for the ground. The monkey is going to cast and receive a shadow, and the ground will just receive shadow. I used a very simple shader that just transforms them into clip space and colours them both. If you have some controls that let you move around a virtual camera, then it's going to make testing a lot easier.



I'm going to make a scene like this with a light to the right and above the Suzanne mesh. It's going to point straight down. With a 45 degree frustum it should cover an area similar to that shown by the green box. It's always a good idea to draw a couple of views on paper before you code up a scene - this makes the debugging process much easier.

I suggest integrating a mesh loader because it's going to produce more interesting shadows than a few triangles - and show you more of the difficult

issues that texture shadows creates. I will also add in a ground "plane" manually. I am going to use the same simple shader for both, and add in a uniform so that I can set a different colour for each one. I made my ground plane in the usual way, but lowered it down 1 unit so that it was below my monkey. Having it at Y height 0 would mean that it would be at eye level, and kind of invisible.

```
void init_ground_plane () {
    float gp_pos[] = {
        -20.0, -1.0, -20.0,
        -20.0, -1.0, 20.0,
        20.0, -1.0, 20.0,
        20.0, -1.0, 20.0,
        20.0, -1.0, -20.0,
        -20.0, -1.0, -20.0
    };
    ...
    // create VBO and VAO here
    ...
}
```

Once we have a basic scene rendering, we can just add in a second camera that will be positioned where the light is in world coordinates:

```
// create a view matrix for the shadow caster
vec3 light_pos (2.0f, 10.0f, 0.0f);
vec3 light_target (2.0f, 0.0f, 0.0f);
vec3 up_dir (0.0f, 0.0f, -1.0f);
mat4 caster_view_mat = look_at (light_pos, light_target, up_dir);

// create a projection matrix for the shadow caster
float near = 1.0f;
float far = 100.0f;
float fov = 45.0f;
float aspect = 1.0f;
mat4 caster_proj_mat = perspective (fov, aspect, near, far);
```

So, I'm using my own maths library here to generate a standard view matrix, and a standard projection matrix for a light that is pointing straight down, and has an arc of 45 degrees to either side. The aspect ratio is 1 because I want my frustum to be a square-shaped, and because I will be writing to a square texture. Note that the near and far clip plane distance are very important. The closer together that you can make these - the more accuracy you will have in your shadow map (more shades per unit of depth). Don't forget that your up vector is a direction and should be re-calculated, and normalised, if you change the angle that the light is pointing in.

Great, now we have an extra camera. If you set up a keyboard button to

switch between the main camera's matrices and the light's matrices, then you can test what your light can "see". It is worth doing this first - make sure that you have these matrices correct i.e. pointing in the right direction before proceeding.



This is the basic scene as viewed from the main camera at [0,0,4]. Size of monkey heads may differ from that depicted. In other words - check how big your mesh is before exporting it - I think that I made mine a about 2x2x2 "unit monkey".

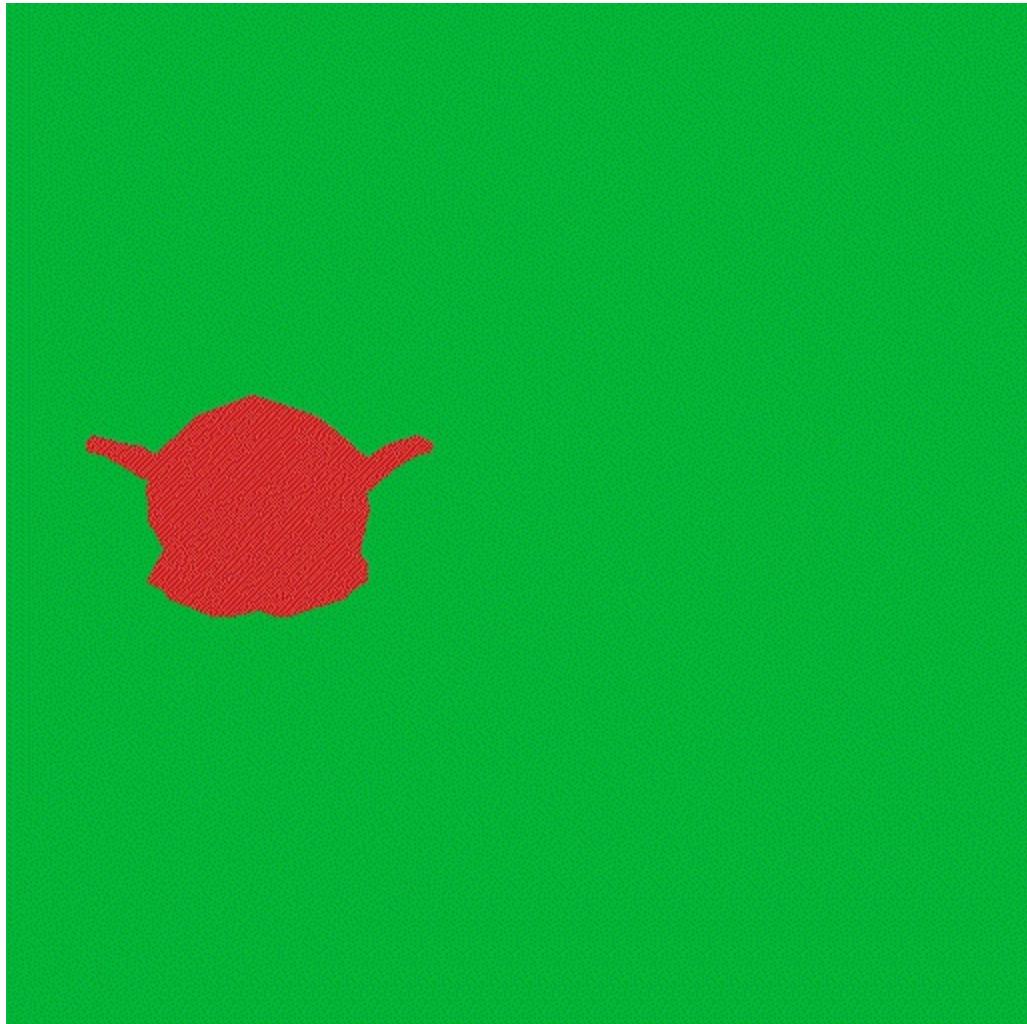
I added in some code to switch in the light's view and projection matrices. I called my uniform locations fairly obvious names for the view and projection matrices. The .m is just the convention that I use in my maths library to get hold of the arrays of floats inside my matrices. My main scene camera has matrices called v and p. I call this code in my main loop before rendering.

```
if (glfwGetKey (window, GLFW_KEY_F1)) {  
    glUniformMatrix4fv (P_loc, 1, GL_FALSE, caster_view_mat.m);
```

```

    glUniformMatrix4fv (V_loc, 1, GL_FALSE, caster_proj_mat.m);
} else {
    glUniformMatrix4fv (V_loc, 1, GL_FALSE, V.m);
    glUniformMatrix4fv (P_loc, 1, GL_FALSE, P.m);
}

```



The scene as viewed using the light's view and projection matrices. We can now confirm that the matrices are working as expected, and that the monkey is fully inside the view. If you modify the light matrices later we now have a mechanism to go back and test that they are working properly.

Great, once that's all in order I suggest giving the light position some wobble - I added a sine function to offset its position based on the elapsed time. This makes it move back and forth a little bit - dynamic shadows are going to be much more interesting to test. If you like this idea, then remember to update the light's view and projection matrices. Test this in the camera switcher thing that we just made to make sure that the wobble keeps the monkey in view at all times. Do not lose the monkey.

Step 2: Create Debugging Quad. Test It

We really want to be able to see what our depth map looks like when we are testing it. Switching views or using it as a texture on the ground is okay, but it's more convenient to just put a little 2d box on the screen where we can display it. Do this now, because you will pull your hair out if you have to do this later when something isn't working.

I just made some basic vertex points and texture coordinates for 2 triangles in clip space that will sit in the top-right on the screen:

```
// create geometry and vao for screen-space quad
float ss_quad_pos[] = {
    0.5, 0.5,
    1.0, 0.5,
    1.0, 1.0,
    1.0, 1.0,
    0.5, 1.0,
    0.5, 0.5
};

float ss_quad_st[] = {
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0,
    1.0, 1.0,
    0.0, 1.0,
    0.0, 0.0
};
// create VBOs, VAO, and set attribute locations
...
```

We also need a shader that can sample a texture:

```
#version 400

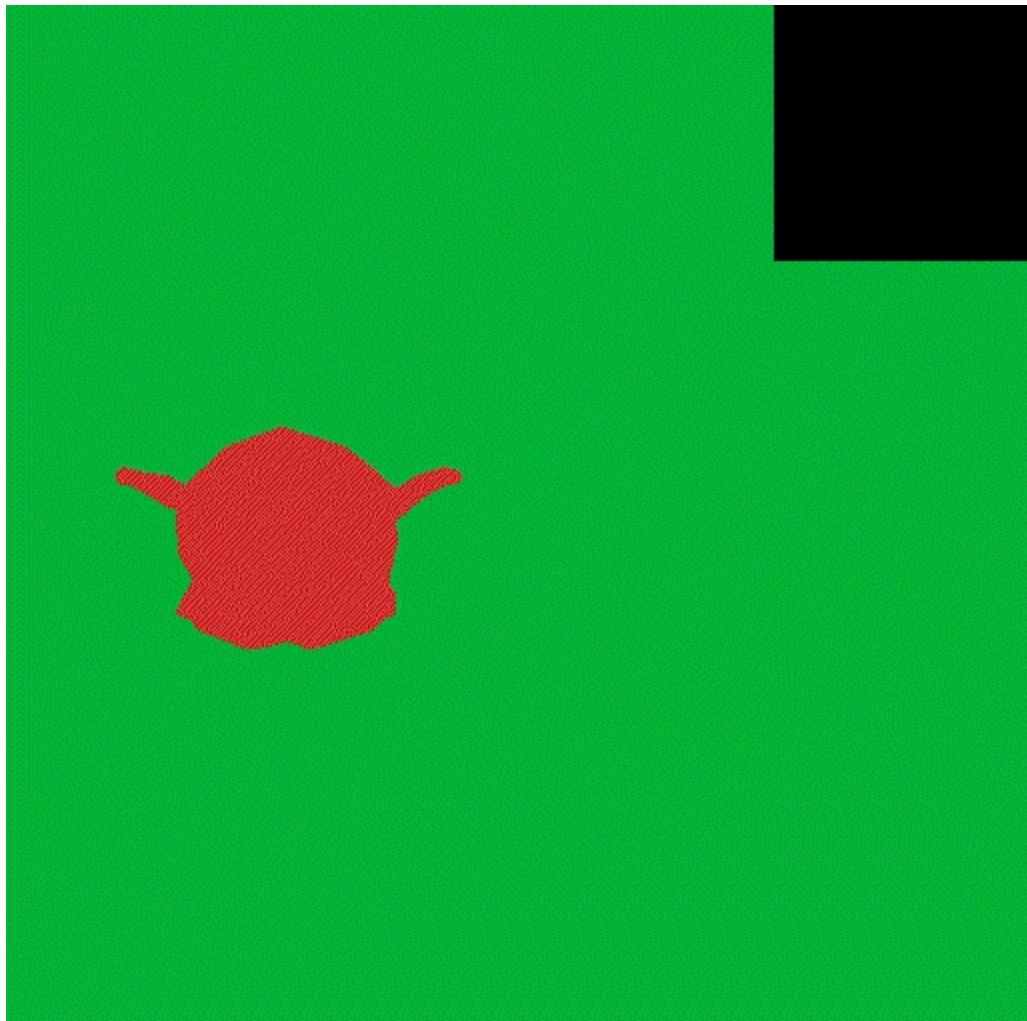
in vec2 vp, vt;
out vec2 st;

void main () {
    st = vt;
    gl_Position = vec4 (vp, 0.0, 1.0);
}

#version 400

in vec2 st;
uniform sampler2D depth_tex;
out vec4 frag_colour;
```

```
void main () {  
    frag_colour = texture (depth_tex, st);  
}
```



The quad in the corner will sample our depth texture when we have it. For now I just coloured it black;
`frag_colour = vec4 (0.0, 0.0, 0.0, 1.0);`

Step 3: Render Depth to Texture. Test It

To render to a texture we need to create another framebuffer that we will render before the scene that we have now. We will set this up with a texture to capture the depth. Now, in OpenGL 4 there are special depth texture attachments - we don't need to encode the depth as a colour, **the depth will be written to the texture automatically**.

Set Up Framebuffer and Texture

Creating the framebuffer and texture is much the same as we have done already for post-processing techniques, but the dimensions of the texture do not need to match the viewport size. As mentioned in Williams' original paper - this is the critical performance section. It is worth trying several different sizes to see the effect that this has on frame rate and visual quality. I am using 256x256 for this demo. Note that in `glTexImage2D` I don't give a colour image number, but instead give `GL_DEPTH_COMPONENT`. This tells OpenGL to automatically write the depth to the texture, and ignore any output variables from the fragment shader.

```
// dimensions of depth map
int shadow_size = 256;

// create framebuffer
GLuint fb = 0;
 glGenFramebuffers(1, &fb);
 glBindFramebuffer(GL_FRAMEBUFFER, fb);

// create texture for framebuffer
GLuint fb_tex = 0;
 glGenTextures(1, &fb_tex);
 glBindTexture(GL_TEXTURE_2D, fb_tex);
 glActiveTexture(GL_TEXTURE0);
 glTexImage2D(
    GL_TEXTURE_2D,
    0,
    GL_DEPTH_COMPONENT,
    shadow_size,
    shadow_size,
    0,
    GL_DEPTH_COMPONENT,
    GL_UNSIGNED_BYTE,
    NULL
);
```

When I create my texture I give it some interesting parameters; I'm using `GL_LINEAR` filtering. When it's working try switching this to `GL_NEAREST`. You'll see a big drop in the quality of the shadows but only a marginal improvement in frame-rate. This is therefore a computationally cheap way to smooth between blocky elements in the shadows.

```
// bi-linear filtering is very cheap and makes a big improvement over nearest-neighbour
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// clamp to edge. clamp to border may reduce artifacts outside light frustum
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

I also use `GL_CLAMP_TO_EDGE`. This means that if we sample an area of the shadow map outside of the texture, then it will use the value on the closest edge. This may not always be desirable - if you are getting harsh transitions for objects on the edges of the light's frustum, then it might be worth using a "clamp to border" parameter instead, and setting the border colour to full white/black; "completely in shadow/not in shadow" when outside the frustum. This will become clear when you debug and refer to the quad in the top-right of the screen.

```
// attach depth texture to framebuffer
glFramebufferTexture2D (GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, fb_tex,
0);

// tell framebuffer not to use any colour drawing outputs
GLenum draw_bufs[] = { GL_NONE };
glDrawBuffers (1, draw_bufs);

// bind default framebuffer again
 glBindFramebuffer (GL_FRAMEBUFFER, 0);
```

Again, at the end of the code, when attaching the depth texture to the framebuffer we specify the type of attachment as `GL_DEPTH_ATTACHMENT`.

Set Up Shadow-Casting Shaders

We also need some shaders for the depth-map creating rendering pass. The vertex shader should just take each mesh, position it in space, use the light's view and projection matrices instead of the scene camera's.

```

#version 400

in vec3 vp;
uniform mat4 P, V, M;

void main () {
    gl_Position = P * V * M * vec4 (vp, 1.0);
}

```

The fragment shader does nothing, because the depth is automatically written to the texture:

```

#version 400

void main () {
}

```

We will need to set up variables to get the uniform locations for those variables as per normal.

Rendering the Shadow-Casting

Now comes the odd bit. We need loop through and draw each object that should cast a shadow. I don't want my plane to cast a shadow, so I'm just going to draw the monkey in the first pass - the plane will not appear in the depth map because I only want it to receive shadows. I call this at the start of my main drawing loop:

```

// bind framebuffer that renders to texture instead of screen
glBindFramebuffer (GL_FRAMEBUFFER, fb);
// set the viewport to the size of the shadow map
glViewport (0, 0, shadow_size, shadow_size);
// clear the shadow map to black (or white)
glClearColor (0.0, 0.0, 0.0, 1.0);
// no need to clear the colour buffer
glClear (GL_DEPTH_BUFFER_BIT);

// bind out shadow-casting shader from the previous section
glUseProgram (depth_sp);
// send in the view and projection matrices from the light
glUniformMatrix4fv (depth_V_loc, 1, GL_FALSE, caster_view_mat.m);
glUniformMatrix4fv (depth_P_loc, 1, GL_FALSE, caster_proj_mat.m);
// model matrix does nothing for the monkey - make it an identity matrix
glUniformMatrix4fv (depth_M_loc, 1, GL_FALSE, identity_mat4 ());

// bind the monkey's vao and draw it
glBindVertexArray (monkey_vao);
glDrawArrays (GL_TRIANGLES, 0, monkey_points_count);

// bind the default framebuffer again

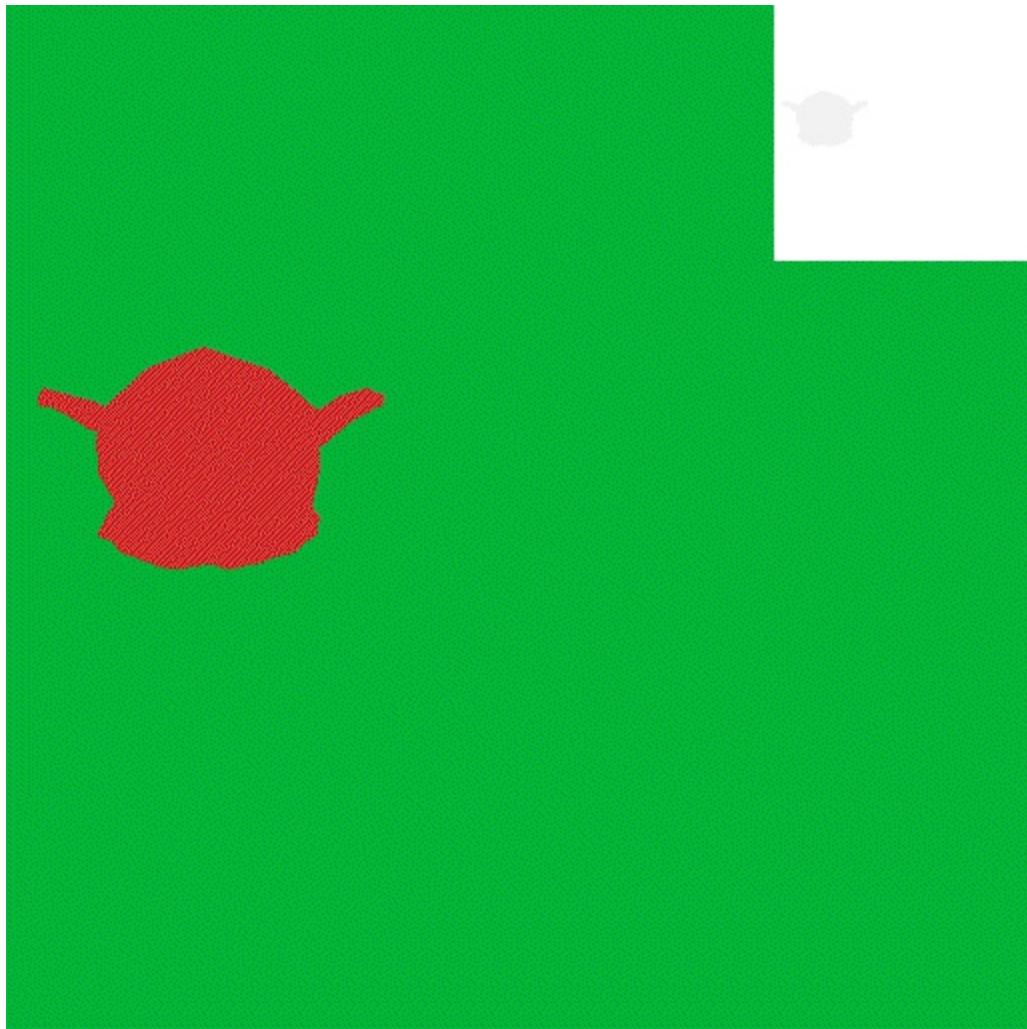
```

```
glBindFramebuffer (GL_FRAMEBUFFER, 0);
...
// start of main rendering pass here - clear viewport etc.
```

That's the end of the first pass. If I had more monkeys or shadow-casters I would draw them there too, before switching back to the default framebuffer. If you compile now it should work but do nothing special. Let's use the texture in the quad. We can modify the code where we are drawing the quad:

```
// bind quad shader
glUseProgram (quad_sp);
// bind depth texture into slot 0
glActiveTexture (GL_TEXTURE0);
glBindTexture (GL_TEXTURE_2D, fb_tex);
// bind quad VAO and draw all 6 points
glBindVertexArray (quad_vao);
glDrawArrays (GL_TRIANGLES, 0, 6);
```

Now, if we were rendering black to the quad before, we can uncomment the texture sampling line in the shader and it should look ... completely white! Why? The colours in the depth texture correspond to the range between the near and far clipping planes of the light's projection matrix. Try reducing the range and you should get a more accurate/detailed depth map image. I changed my far-clip distance from 100.0 to 20.0 and it looked like this:



This is what the quad will look like when we have our depth-buffer working. It should correspond to this view from the light's camera and display greyscale values for distance - darker=closer to the light. Remember that the colour range used corresponds to the difference between your near and far clipping planes. If you see a full-white texture, try to reduce the range between the planes.

Step 4: Modify Main Scene's Shaders. Test Depth Sampling

If you've gotten this far then we can use the depth map in our main scene shaders. We need to modify the shaders for everything that should receive a shadow. In my case I will modify the shader that I'm using for the monkey and the plane.

To get this to work we need to convert every vertex point into the space of the light's frustum. With this we can work out the corresponding texel in the depth map. We will send these to the fragment shader to get more accurate per-fragment sampling. In our case we are going to have several points that match up to the same texel in the depth map. The light is coming from the top. The very top middle of the head matches the blackest texel in our quad's image. But the very bottom-middle of the head will also map to this texel. This is the neat part - we sample the depth texture for each one. They get the same black value. Then we compare the depth of the point that we just worked out in light coordinates to the sampled texel. If it's roughly the same depth or smaller then we know that it is the first point that the light hit - no shadow then. In the case of the bottom of the head, we get the same darker texel (dark is a small number), but the point is much farther away (a bigger value) so we can say - shadow it!

```
#version 400
in vec3 vp;

// uniforms for light's matrices, and scene camera's matrices, and model matrix
uniform mat4 caster_P, caster_V, P, V, M;

// point in the light's space
out vec4 st_shadow;

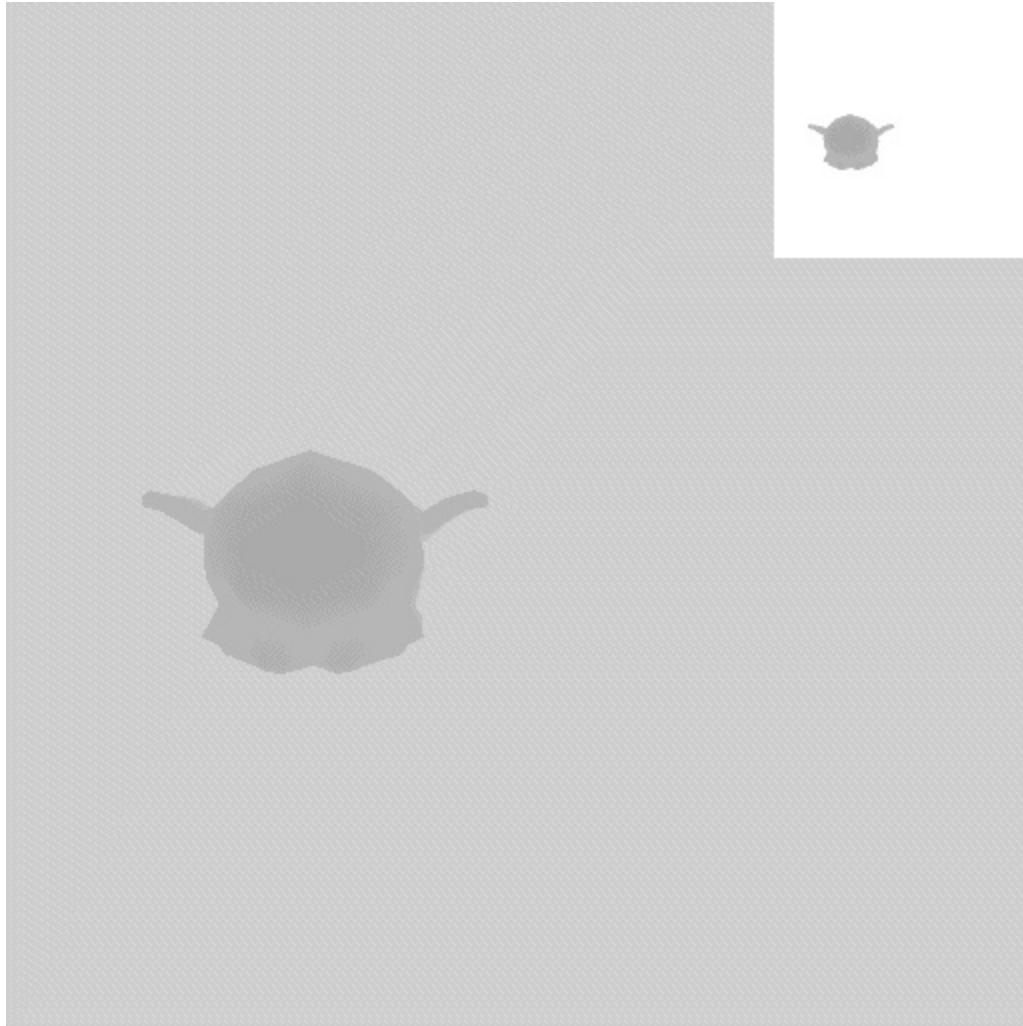
void main () {
    // transform vertex position to clip space (camera view and perspective)
    gl_Position = P * V * M * vec4 (vp, 1.0);

    // create a shadow map texture coordinate by backwards-is-ing the position.
    st_shadow = caster_P * caster_V * M * vec4 (vp, 1.0);
    st_shadow.xyz /= st_shadow.w;
    st_shadow.xyz += 1.0;
    st_shadow.xyz *= 0.5;
}
```

So, in the above vertex shader, you can see we do a normal position-setting with the same matrices as usual. However, we also create transform our vertex points in the lights space - the exact same way as we did in the depth shader; `caster_P * caster_V * M * vec4 (vp, 1.0);`. The lines after this go 1 step further and take it from homogeneous clip space into texture coordinates. Remember that after a vertex shader finishes, there's an automatic transformation step that OpenGL does - **perspective division**. Before our depth shader was written into a flat 2d texture, perspective division was applied. We do this manually here with the "divide by w" instruction. This just moves points closer to the middle of the screen as they are farther away. If we don't do this then we'll get a distorted shadow. This has the effect of converting our xy coordinates from -1:1 into 0:1, meaning that they can be used as texture coordinates, and our z value into the range of 0:1 - the same as that used in our depth map texture.

If you add an `in vec4 st_shadow;` to the top of your fragment shader, I suggest that you output this as the final colour so that we can have another look at what will happen:





Here we draw the depth value of each point in the light's coordinate space. You can see that the bottom of the head is a light colour (far away depth value), and the top is near-black (close to the light source).

When we use our xy coordinates, we will sample the corresponding texel in the actual depth map, which we are drawing in the top-right. The top of the head matches almost 1:1 to the image - we won't shadow this area. The bottom is darker and does not match the image - we will shadow these areas. Some of the ground plane under the head is also going to map to the dark texels in the depth map - you can see this in the bottom (light's perspective) image, as some of the ground is covered by ears etc. This will also be shadowed.

To make debugging easier you might like to tweak your near and far planes until you get a range of greys that is easy to visualise. In the images above, I used 5.0 and 15.0, respectively. I hope I've made it clear that testing and inspecting the details of each stage is very important in getting this technique to work without any frustration or "magic numbers". Next we can do our depth comparison and apply shadow in the fragment shader. I added in a function:

```
#version 400
```

```

// vertex points in light coordinate space
in vec4 st_shadow;

// the depth map
uniform sampler2D depth_map;
uniform vec3 colour;

out vec4 frag_colour;

// constant that you can use to slightly tweak the depth comparison
float epsilon = 0.0;

float eval_shadow (vec2 texcoords) {
    float shadow = texture (depth_map, texcoords).r;
    if (shadow + epsilon < st_shadow.z) {
        return 0.2; // shadowed
    }
    return 1.0; // not shadowed
}

void main () {
    float shadow_factor = eval_shadow (st_shadow.xy);
    frag_colour = vec4 (colour * shadow_factor, 1.0);
}

```

In the above code we do our depth comparison. This gave me the result as below:



The first attempt at using a 256x256 shadow map and unmodified depth comparison. The shadow on the ground plane is perfect - but is low resolution due to the depth map size. The self-shadowing on the monkey has many artifacts caused by inconsistent/inaccurate depth comparisons.

Step 5: Try Different Light and Camera Views

If you have keyboard controls for the camera and the light position and angle then I suggest that you try a few - find the situations where the algorithm stops working properly. Areas outside the light frustum may have transitions between shadow and light. If your scene will encounter this then consider using a wrap-to-border technique for the shadow map.

You can also add code to your fragment shader that checks if:

- texture coordinates are less than 0 or greater than 1
- depth or w value is behind the light frustum
- depth is too far away i.e. set a cut-off distance for shadow-casting

In these cases you can skip the sampling of the shadow map, and arbitrarily say "full shadow" or "no shadow".

Step 6: Improve Visual Quality

There are a whole host of methods to improve the quality of texture shadows. Some common techniques are:

- use a higher resolution depth map
- use bi-linear texture filtering
- add an epsilon value or threshold to the depth comparison
- use OpenGL's `glEnable(GL_POLYGON_OFFSET_FILL);` function with a small `glPolygonOffset(x, y);` - similar effect to using epsilon but applies equally to surfaces at different angles to the frustum.
- use a kernel of several depth-map samples to blur the shadow output

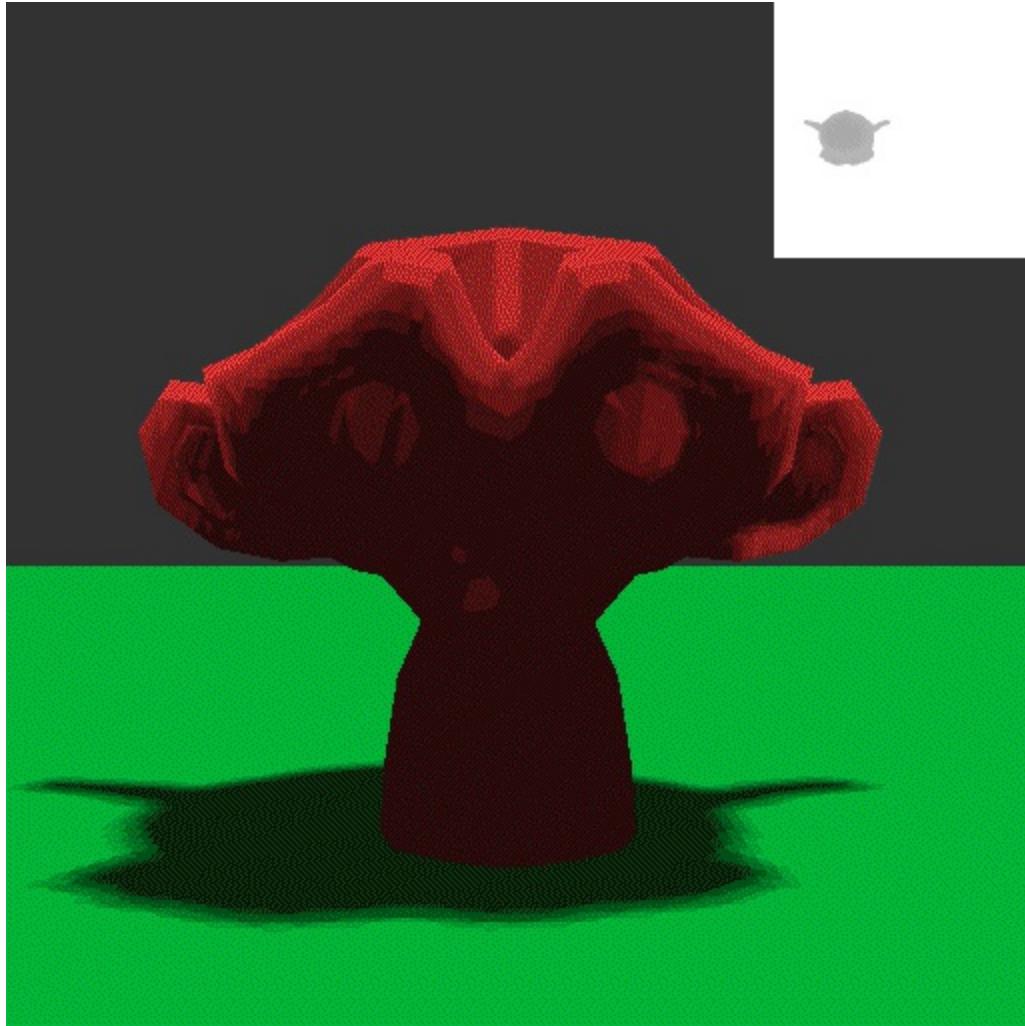
We can improve the shadow on the ground plane by increasing the resolution of the shadow map - but this will impact the performance of the algorithm. Try several sizes: 64x64, 128x128, 256x256... etc. and find a balance that suits your application. Remember to update the framebuffer viewport with this size as well.

We are already using bi-linear filtering. Try disabling it to see the difference that it makes.



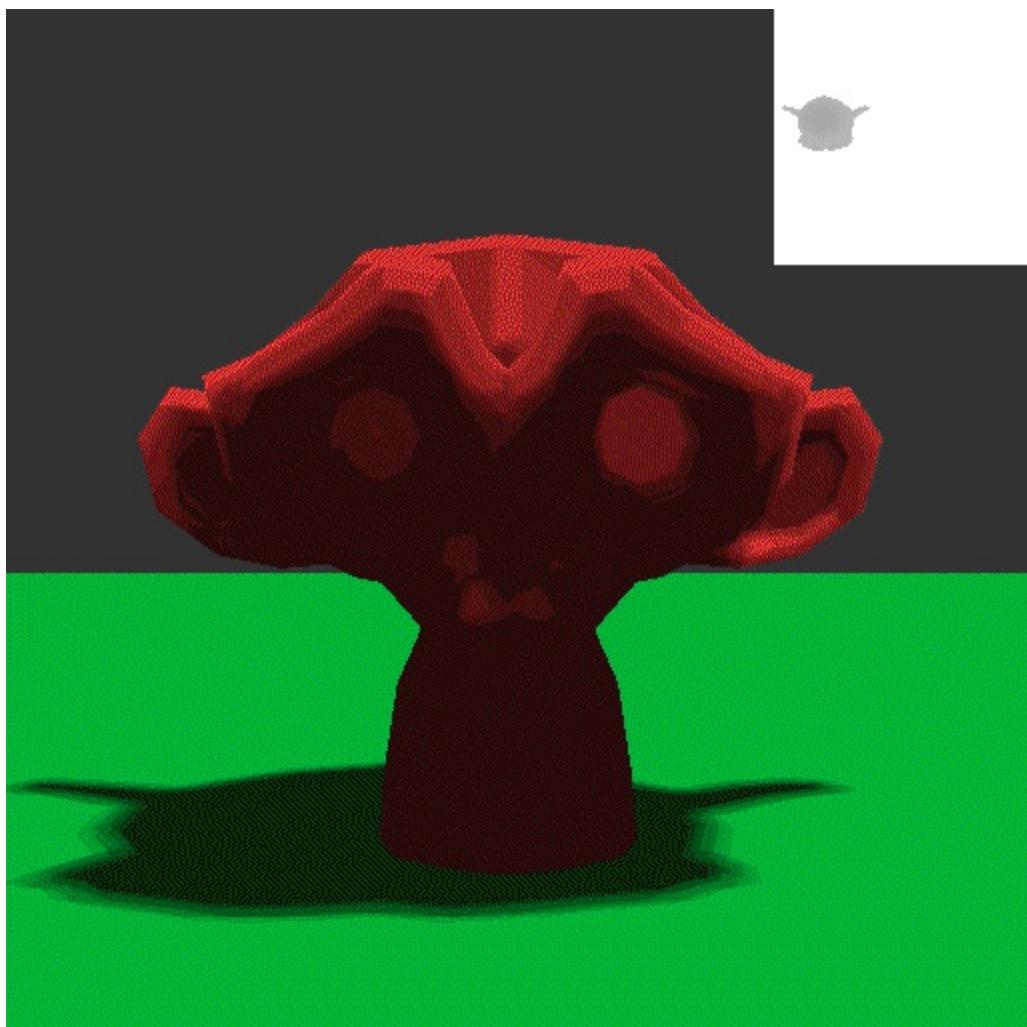
A small threshold of 0.0042 is added to the depth comparison to try to reduce self-shadowing artifacts.

Self-shadowing is a notorious problem with this algorithm. Set your epsilon value to 0.0042 - does it make a small difference to the flickering artifacts on the monkey head?



Taking several depth-map samples around the matching texel and averaging the shadow result can create a slight blur to the harder edges. Here I average 8 neighbours. I also use the epsilon of 0.0042 here.

Taking additional samples and averaging them will create a blur, but texture samples are quite computationally expensive. The trick is to take only a small number of samples, but to distribute them a little bit randomly (i.e. not in a square). This should help to break-up any repeating moire patterns in the shadows (or make them harder to spot).



Here I combine the previous 2 techniques, but also use a 4096x4096 pixel depth map. Note that I haven't used any Phong lighting - so once you combine that too then you will get much more 3d definition over the surface.

Common Mistakes

- **do not read (sample) the depth map texture whilst the framebuffer that writes to it is still bound.** This causes huge graphical distortions and flickering, and may crash your video driver. I did this to a laptop once and it never drew the screen properly again.

Other Types of Light

For point source lights, you will need to cast a shadow projection in 6 directions. Each one has a 45 degree projection frustum. Imagine making a cube of projections from 6 different shadows. This will give you 6 shadow maps. You could write these into one of OpenGL's built-in cube maps so that they are treated as a single texture. **If possible avoid casting shadows from point lights** because they are roughly 6 times more expensive than a spotlight.

For directional lights (like the sun) you will need to replace the light's projection matrix with an orthographic projection matrix instead of a perspective projection matrix. Here's an array of floats that I used to try this in my demo. The array is in column order i.e. the first 4 numbers in the array are the first column in the matrix:

```
float P_ortho[] = {  
    2.0f / (x_max - x_min),  
    0.0f,  
    0.0f,  
    0.0f,  
    0.0f,  
    2.0f / (y_max - y_min),  
    0.0f,  
    0.0f,  
    0.0f,  
    0.0f,  
    -2.0f / z_max,  
    0.0f,  
    -(x_max + x_min) / (x_max - x_min),  
    -(y_max + y_min) / (y_max - y_min),  
    -1.0f,  
    1.0f  
};
```

Where the `_max` and `_min` values are the size of the view in world units. All this does is scale our x and y values into the familiar range of -1:1, and visible z range into -1:1 before outputting from the vertex shader. The w value is left at 1, so no perspective division occurs afterwards. This is what we did manually the Hello Triangle example. I used plus and minus 5 for the xy range in my monkey head demo and checked in the debug panel. For a bigger outdoor scene you might want a bigger range. Remember that changing the z

position of the camera here and the `z_max` range isn't going to make the monkey head look any bigger or smaller, because it's orthographic, but it will affect the shade of the head in the debug panel.

For larger outdoor scenes expect to need quite high-resolution depth-map textures. You'll only need to cast shadows on the scene currently in view though, so you can adjust the orthographic projection size and light position to chase the scene camera around.

Discussion

Building Larger Programmes

One thing that is not addressed by tutorials and text books, but is still a practical concern, is how to actually put a larger programme together. We usually study a set-piece example, but most people reading have ambitions to build a game or a larger demonstration. An early whack on the nose with OpenGL is when you find out that its state machine does not scale well. It's easy to make a mess. It's easy to assume a state is set, but a small, and seemingly unrelated, change elsewhere in the programme will break everything. We find ourselves wondering:

- *What are some good general rules to follow when coding to the OpenGL interface?*
- *What is the relationship between VAOs, VBOs, shader programmes, and the "things" that I want to draw? - in other words - How would I code this in a video game that has several different character meshes to draw, some re-used?*
- *How do I code this so that it's efficient and runs expediently?*

In the next few sub-sections, I'll show you how I personally address all of these issues. Other programmers I know use different styles and layouts, but it might at least give you an insight into a minimalist approach.

My "GL Utilities" File

If you're a C++ or Java programmer, the first question that you'll ask is probably more like *How do I encapsulate all of this into classes?* My initial advice would be not to bother, but to create a few, small, helper functions, to tidy a few commonly-used things up a bit. In my video game, and other projects, I just have one small code file for "GL utility" functions. It has the following functions:

```
/* log is appended to - this clears it and prints the date */
bool restart_gl_log ();
bool gl_log (const char* message, ...);
/* same as above but also prints to stderr */
```

```

bool gl_log_err (const char* message, ...);
void log_gl_version ();
void log_driver_capabilities ();

```

So, I start with simple C log functions, as described in the early tutorials. Next, I have some generic "start OpenGL" functions:

```

bool start_glfw (bool enable_full_screen, bool g_vsync, int msaa_samples);
void stop_glfw ();
void _set_gl_window_hints (int msaa_samples);
void _window_resize_callback (GLFWwindow* window, int width, int height);
void _update_fps_counter ();
double get_elapsed_seconds ();
bool start_glew ();
void set_render_defaults ();

```

All of this is fairly self-evident, and described in the first tutorials also. The resize call-back updates the projection matrix for my scene camera (which I have only one of). The "defaults" to set are OpenGL states that I will expect to be enabled all the time, unless I explicitly disable and re-enable them again later for some special technique. These are things like setting the background "clear" colour, enabling culling of back-faces, and enabling depth-testing. You might have a few more, depending on your preference.

```

unsigned int create_vbo (int dimensionality, void* buffer, int length);
void add_attribute_to_array (
    unsigned int vao, int attribute_location, int vbo, int dimensionality);

bool parse_file_into_cstr (
    const char* file_name, char* shader_str, int max_len);
/* load shader from file, compile, check for errors */
bool compile_shader (unsigned int s, const char* file_name);
/* create a shader programme from vertex and fragment shader files */
unsigned int create_sp (const char* vs_file_name, const char* fs_file_name);
unsigned int create_sp (
    const char* vs_file_name, const char* gs_file_name, const char* fs_file_name);
bool link_sp (unsigned int sp);
int get_uniform_loc (unsigned int sp, const char* var_name);

```

Now we see my rather short list of actual OpenGL short-hands that I will use several times. These are simply to reduce repeated sequences of code down to a single function call that is re-used. You might be surprised that there isn't much here - I don't encapsulate much at all, but prefer to keep OpenGL calls un-edited in my code-base. Why? It's easier to visually debug. Okay, what I do have here are the tedious "load stuff into GL" routines as single functions. I can create a vertex buffer, which has all the binding, generation etc. taken care of - why this tedium is even part of the interface at all is beyond me. I don't have a function to generate a VAO, but I do have a function to specify a

new attribute in one - this will both set up the attribute pointer, and enable the attribute. I have some simplified function calls to create shaders from a path to an external vertex shader and fragment shader text file. Note that I make linking the shader programme a separate call, because I might want to bind attributes in-between compiling and linking. Lastly, I have a function to get the uniform location of a shader variable. This also checks if the location is valid, and prints a warning if it is not.

```
bool screenshot ();
void reserve_video_memory ();
void grab_video_frame ();
bool dump_video_frames ();

bool verify_bound_framebuffer ();
```

Not much is left - just some utilities to take screenshots and videos. The last function does a validation call of any framebuffer - `glCheckFramebufferStatus()` - but with verbose text output. It is always a good idea to check after building a new framebuffer. Other than that I just keep some global variables that will be used throughout my code:

```
extern V_GL g_gl_ver; /* the current version of GL being used */
extern char g_shaders_path[256]; /* folder to look in for shaders */
extern int g_gl_width; /* current width of viewport in pixels */
extern int g_gl_height; /* and height */
extern int g_bpp; /* bits-per-pixel of colour depth being used */
extern int g_batch_count; /* the number of "draw" calls executed in this frame */
extern int g_vertex_count; /* number of vertices drawn this frame */
extern int g_uniform_count; /* number of calls to glUniform this frame */
extern bool g_resized_view; /* if the view has been resized this frame */
extern GLFWwindow* g_window; /* a pointer to the window, used by many GLFW funcs */
```

The interesting thing here is that I keep track of the number of draw calls, vertices, and uniform calls per frame. I put this alongside where I print out my frame-rate. If any one of these numbers gets too big we should expect a corresponding dip in rendering time - and then we have a better insight as to why.

You could make a gigantic UML-designed abstract wrapper interface thing for OpenGL, as a lot of students immediately start doing, but is this really a profitable investment of your time, and in fact are you making it harder to debug?

"Un-binding"

I write long functions. If you write small functions it's easy to lose the "thread" of the GL state machine. In this case it's a good idea to practice "un-binding" at the end of each function. Most OpenGL states reserve number 0 to mean "invalid". If you bind something to 0, and try to use it later, you should get an error. This is a good thing, because having something work by accident will definitely get you into trouble later. Example:

```
glBindBuffer (GL_ARRAY_BUFFER, vbo);
glBufferData (GL_ARRAY_BUFFER, sizeof (points), points, GL_STATIC_DRAW);
glBindBuffer (GL_ARRAY_BUFFER, 0);
```

Bind, use, unbind. You will see this in code around the web. I don't do this myself because every call adds a little bit of overhead, and also because I like to live on the edge - a personal preference! Note that not all OpenGL variables have 0 meaning "invalid" - notable exceptions are textures, and uniform locations.

Incidentally, there are **direct state access** (DSA) OpenGL extensions, which remove the binding rituals altogether, and allow you to give the variable as a parameter to the main functions instead. Hopefully this catches on as the main interface in the future. See

https://www.opengl.org/registry/specs/EXT/direct_state_access.txt for examples.

The Number of VAOs, VBOs, and Shaders, vs. Game Characters

So how on earth do we relate the unusual VAO/VBO abstraction to the actual abstraction that we care about in games and simulations? This might be obvious to you by now, but I get asked this question a lot, so here's an example of the way that I would manage this in a larger programme. Perhaps we have a car racing game. We have 3 different 3d mesh files to load for the cars. There are perhaps 12 cars in the race at once though - they re-use the meshes. Each car has its own position, speed, etc. How does this relate to VAOs? In my games I would have two data structures for this type of thing:

- A car model specification (x3 instances in an array)
- An actual car (x12 instances in an array)

The specification would load a file for each type of car. It would have this kind of information:

- The mesh file to use
- The texture files to use (diffuse, specular, normal maps etc.)
- Any unique statistics like top speed, etc.
- Shaders to use

I would load all of these when the game starts, and allocate an instance of my specification data structure for each one. That data structure would have the following variables:

- The VAO to use
- The number of vertex points in the mesh
- Any VBOs used (not essential)
- The texture IDs to use
- The unique statistics
- Shader programme ID to use

So, you can gather that I also load the mesh file and create a VAO and VBOs for it when I load the specification file. I also load any textures required here, and keep track of their handles. You will probably re-use the same shaders for all of your cars, but if they are significantly different, you might also load an unique shader programme for each one too. This means that we have an analogy that for each mesh file there is one VAO.

When I start a game I probably have pre-determined how many cars will be in play, and what each type of car are. In a game with "levels" you might load a list of types and positions of monsters. In any case, now we can populate an array of these. My "actual car" data structure would have variables like:

- The index number of the specification to use
- The world position, and orientation
- The current model matrix for the car (world matrix)
- Current speed

- Other game states (number of wheels still attached, etc.)

So, when the car is moving, we can look up its specification in our index of the 3 of them, and find out what its top speed is, etc. without repeating the data for every car of the same type. We have a similar idea for drawing them - we are re-using the same mesh data, stored in vertex buffers. We might have drawing code like this:

```
/* use the cars shader (you might change this between car types, or not) */
glUseProgram (the car shader);
int last_spec_index = -1;
for (all cars on screen) {
    int spec_index = actual_cars_array[i].specification_index;
    /* update the model matrix for each car */
    glUniformMatrix4fv (model_mat_location, actual_cars_array[i].model_matrix);
    /* change texture and VAO, only if it's different to the last one */
    if (last_spec_index != spec_index) {
        /* bind the VAO */
        glBindVertexArray (car_specifications_array[spec_index].vao);
        /* bind the texture */
        glActiveTexture (GL_TEXTURE0);
        glBindTexture (GL_TEXTURE_2D, car_specifications_array[spec_index].diffuse_map);
    }
    int num_points = car_specifications_array[spec_index].num_points;
    glDrawArrays (GL_TRIANGLES, 0, num_points);
    last_spec_index = spec_index;
}
```

So, I draw all the cars. I assume that all the cars will re-use the same shader. I only update the model matrix uniform. If the car was a different type to last time, I also bind a different VAO, and texture. You could improve this code by drawing all the cars of one type first, then the next type. This would reduce the amount of calls to `glBindVertexArray` etc., but would require another data structure to keep track of them all by type.

The key ideas are to re-use data that has been loaded onto the graphics card memory, and to minimise the number of binding and shader-switching calls to keep the overhead low. Note that I don't mix my drawing updates with my simulation updates. I would have all my speed and position worked out in a separate loop, which would run at a fixed **time-step** - e.g. one step for every 0.02 seconds that pass on the computer's clock - "real-time" simulation. The graphics updates I just keep calling as fast as possible. You can have this automatically lock this to the refresh rate of the monitor to prevent any tears and flickers if you like - GLFW has `glfwSwapInterval()` for this. My video driver does an appallingly bad job of doing this automatically, so I usually disable

that in the control panel.

Measuring Time and Efficiency

The curious problem that you face when working out how to make your graphics code run faster, is that you have two different processing units, operating somewhat asynchronously. Each OpenGL function call has an overhead on the CPU, as well as a different, and not-entirely-in-step-with operating time on the GPU. You cannot put, for example, GLFW timers in your C code on either side of an OpenGL function, and expect this to give you the time taken by the function on the GPU - this will only give you the CPU overhead cost.

If you want to know how long various function calls take to execute on the GPU you can use the [ARB_timer_query](#) extension (or part of the core OpenGL if you have 3.3+ support). This is useful if you want to compare one method of rendering something to the cost of another i.e. to see if you've improved your code.

```
// GPU timer queries setup
GLuint timer_query;
GLint query_available = 0;
GLuint64 gpu_time_elapsed = 0; // this is nanoseconds according to ext spec
// NB a 32-bit query counter accuracy limit at around 4 seconds max.
glGenQueries (1, &timer_query);
```

Once you've set up your query, what you do is brace the OpenGL calls of interest with calls to begin and end the query. Then you must wait for the query to end (remember that it may not be synchronous with the CPU) before you can collect the results.

```
query_available = 0;
glBeginQuery (GL_TIME_ELAPSED, timer_query);

/* some code that we want to measure */
glBindVertexArray (multi_vao);
glDrawArrays (GL_TRIANGLES, 0, num_points);

glEndQuery (GL_TIME_ELAPSED); // wait for query to finish on GPU
while (!query_available) {
    glGetQueryObjectiv (
        timer_query, GL_QUERY_RESULT_AVAILABLE, &query_available
    );
    // could put usleep() or Sleep() here
}
```

```
gpu_time_elapsed = 0;
glGetQueryObjectui64v (timer_query, GL_QUERY_RESULT, &gpu_time_elapsed);
/* convert to milliseconds for printing */
double ms = (double)gpu_time_elapsed / 1000000.0;
```

If you combine this with a good CPU timer, which should also measure up to nano-second precision (you can query your CPU to determine this), then you have the basic tools that you need. I suggest not using the timers in `time.h`, as they are most unreliable, but rather using the timer from GLFW. The problem is that it takes a lot of work to implement timers manually, and you might want to test all of your functions to find the bottleneck areas. In this case there is an excellent tool called [apitrace](#) that will profile both your CPU and GPU OpenGL calls. It builds a profiling database, which you can investigate visually using the `qapitrace` tool. It is really worth peeking behind this curtain at some point, because OpenGL programming is absolutely a black box in terms of performance, and really everything else that's happening behind the scenes. If you like apitrace, then it will also be of interest to look at [VOGL](#) which also does OpenGL traces, with playback and debugging.

As I mentioned earlier, it's a good idea to print out the number of uniform calls, draw calls, and total vertex count per frame too. Although this depends entirely on the hardware and version of video driver that you're using, it can give you some heuristics. If you can keep your draw calls and uniform updates under a couple of hundred per frame, then you are well within bounds, and should be able to draw a lot of primitives. If you are having problems it might be worth investigating some optimisation techniques. Traditionally, you would use a scene graph, with for example, a spatial partitioning like oct-trees, or BSP (binary space partitioning), and one or more culling algorithms to remove geometry that will not be seen. However, times are changing, and algorithms are moving away from these techniques to more hardware-centric ideas. You might be interested in hardware occlusion culling, for which there is another query interface, similar to timer queries, but the latest techniques rather exploit the early-Z rejection mechanism to achieve this. See Kubish and Tavenrath's "OpenGL 4.4 Scene Rendering Techniques" presentation from the GPU Technology Conference for an example of this; [link](#) (online June 2014).

Closing Remarks, Future Techniques, and Further Reading

Future Techniques

There are several, newer OpenGL features of note that I have not covered in this volume. This is because they are very new, and I do not have the personal experience to give you sound advice on how to use them for practical video game techniques - expect this in the future!

Compute shaders are the big new feature here. These were released as part of OpenGL 4.3, and allow you to do general purpose GPU computing (GPGPU) in combination with regular usage GPU computing. The compute shaders operate somewhat outside of the hardware pipeline, and communicate with your other shaders via textures or shader storage blocks. The most talked-about application of this would be to introduce some global illumination or ray-traced rendering techniques into the pipe-line. This should allow more realistic lighting models, and more powerful screen-space special effects.

I have used the somewhat old-fashioned data buffers and uniforms throughout the text. I think that this was a good idea, because they are still part of the official OpenGL 4 API, and will also work all the way back to OpenGL 2.1, which makes life easier for students on varying hardware. The newer versions are of course uniform buffer objects for things like camera uniforms, and mapped data storage for data buffers in general. The latest versions of OpenGL have introduced several variations of this that are programmed to avoid the overhead caused by interfacing with drivers. It would be wise to keep an eye on this area in particular.

Direct3D has a shader debugger available in Visual Studio. I think that we deserve an interactive shader debugger for OpenGL. I would expect more OpenGL tools to be created in the near future, particularly given the momentum in this area created by Valve in their encouragement to port games to the Linux-based Steam OS system.

Recently we have seen the announcements of Mantle and Metal graphics APIs, which may actually supplant OpenGL as the best API to use on some of its target platforms. This might hamper the multi-platform value of OpenGL, but it was never really that good in the first place, and any competition is good.

Hardware-wise we can continue to expect bigger graphics memory, and more shader cores, particularly on mobile devices. WebGL, the on-line cut-down version of OpenGL, is becoming the most potent multi-platform graphics API. Apple recently un-locked WebGL support on the new mobile operating systems. This means that, in combination with HTML5, WebGL is an universally better choice than OpenGL ES - you can really write code once, and have it run on a multitude of different mobile and desktop platforms. JavaScript is a nasty little language, but it is very, very quick to work with - much faster than coding the same programme in C. Web browsers also have excellent debugging and profiling tools, and some even have the ability to report bugs in OpenGL shaders. I have developed two WebGL games for Ludum Dare 48-hour game jams, and found it a most expedient choice to work with - far superior to any of the 2d game creation toolkits from days past. We also have very clever integration of touch-screen controls, and better and better audio support with HTML5, native user interfaces, and an ideal content distribution mechanism as a native web browser application. JavaScript does still struggle with CPU-intensive tasks like scene management, and has a nasty habit of "invisible" errors created by small typos.

Further Reading

If you are only going to get one computer graphics textbook, let it be *Real-Time Rendering* by Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. It's generic to all APIs, it has excellent descriptions of a huge number of algorithms, and very easy-to-follow pseudo-code examples. It can be a bit academic at times, so quite a different style, but definitely it should be the flagship book to get. The website is excellent, and you can find more information at <http://www.realtimerendering.com/book.html>.

For a computer graphics textbook with a heavy mathematics focus, look at Eric Lengyel's *Mathematics for 3D Game Programming and Computer Graphics*. I don't know that I'd direct you here to learn maths for 3d programming, as it's a bit sparse on tutorial-like descriptions, but it is certainly an excellent reference to double-check theory, and it covers a lot of computer graphics algorithms. Aside from that it's quite an interesting read as Lengyel writes computer graphics engines, and really knows what he's talking about in detail.

I can't say that I own a copy myself (that latest version wasn't released when I started this material), but the most recent edition of the *OpenGL SuperBible* is the most authoritative source of OpenGL programming, by some of the key Khronos Group contributors; Graham Sellers, Richard S. Wright, Jr, and Nicholas Haemel. I have flicked through a colleague's copy, and certainly it has the go-to examples of some very advanced special effects techniques and the latest interface examples like mapped data buffers. I would not recommend it as an introductory text, however, because it immediately hides a lot of the API details, and refers to the interface of a custom framework in the text, which is not really helpful for student assignments and the like <http://www.openglsuperbible.com/>. Really OpenGL needs a simplified API, as found in the book's framework, to begin with, but that's another story.

If you're interested in WebGL, I suggest Diego Cantor and Brandon Jones' *WebGL Beginner's Guide*. It will get you up to speed with the entire JavaScript/web environment as well as basic GL programming. It's a small book, but very practical, and sticks to low-level WebGL without relying on any extra frameworks. <http://www.packtpub.com/webgl-javascript-beginners-guide/book>. Also look at Eric Haines' excellent on-line computer graphics course using WebGL <https://www.udacity.com/course/cs291>.

The GPU Gems series of books is a great source of algorithms and practical improvements to algorithms. Some of the older, and still very useful, works are available for free on-line.
http://http.developer.nvidia.com/GPUGems3/gpugems3_part01.html.

There are many more books on the topic, that I'm sure are very useful, but that I have not yet had occasion to read, so cannot recommend here yet!

I find that the most useful OpenGL information of all actually comes from informal comments by way of following some of the developers and experts on Twitter. You can find many of the authors of the top texts on that platform.

Closing Remarks

OpenGL is unnecessarily difficult to learn. Official documentation, in typical computer science [cop-out] fashion, gives you API documentation, but this isn't anything even remotely close to a user manual - and that's what you really need. There are several hulking, expensive tomes that you can buy, but the size of these volumes encumbers them both economically and in the ability to keep up to date. In the official documentation, certainly, you can look up a function and find out how to use it, but there is no information as to why to use it versus some alternative. You are expected to either already be an expert, or to absorb this basic usage information *via* osmosis over a period spanning several years. I think that OpenGL has amazing potential to motivate, to build multi-platform software, and to **unleash enormous creative power in visual arts that no other generation in the past has had access to, ever**. There is a lot of promise here, but from a teaching perspective, the API design is appallingly bad - it simply takes too long to learn. It's not robust. It's unreliable. The shaders break in different ways on all of the different driver implementations - you have to learn the quirks of 4-5 different drivers and shader compilers in order to build anything that distributes. It hasn't been designed for students, and I think that this is a major failing. There is an enormous glut of what programmers call "cruft" in the API - old, hang-on code that doesn't need to be there. There is an official "deprecation" mechanism in newer versions of the API, so we can use a "core" version that prevents us using the old stuff by accident. That's okay, but when you start talking to the experts, you also find that there's an **unofficial list of functionality that should be flagged as deprecated but isn't**. That's absolutely ridiculous, but for now we're stuck with it. I am sure that I must have unwittingly used some of this "should be deprecated but still actually official" functionality, and for that I apologise. Nevertheless, I hope that I've helped to chop a stable trail for you through this veritable jungle of uncertainty. I hope that you've enjoyed my semi-formal, unapologetic,

cynical, pragmatic attitude, my hand-drawn diagrams, and my absolute refusal to write in anything but Her Majesty's Imperial Standard English.

I've tried to price this work so that royalties equate to one double espresso. So thanks for buying me a coffee! You just enabled me to learn more material!

Dr Anton Gerdelen