Introduction to Modern 3D Graphics Programming with OpenGL

last update - February 4, 2015

Vladislav Chetrusca, FAF, UTM, 2015
Computer graphics is best learned by doing.
1. Before we start...

- Ask Questions!
- Course homepage: [http://faf.magicindie.com](http://faf.magicindie.com)
  - These slides
  - Samples
  - Labs task
  - Book links
- Level of this course: Introductory
- Required knowledge: C/C++, geometry, linear algebra.
  - Your first task for the next meeting: revise geometry(trigonometry), matrices, vectors and operations on them.
- Theory hours: 15 meetings (once per week)
- Labs: 8 meetings (twice a month), 5 laboratory tasks
- This course is based on the following books:
  - 3D Math Primer for Graphics and Game Development, 1st Edition
  - OpenGL 4.0 Shading Language Cookbook
- Notations throughout this course
  - Reference, page 34 – More details on this topic can be found in the Referenced source
  - SAMPLE1 – Denotes a sample application that can be found on the homepage of this course
- Recommended graphics hardware: OpenGL 3.3 compatible. Note: An OpenGL 2.1 fallback path is provided for most Samples.
2. The course at a glance

The mathematical background

Stetting up your first OpenGL app

Drawing and moving 3D objects

Lighting

Texturing

Image processing
3. The evolution of computer graphics

1950s

- SAGE (Semi-Automatic Ground Environment) system, which combined radar with computers created. SAGE took radar information and created computer-generated pictures. While they look rudimentary by today's standards, SAGE imaging was one of the first examples of a graphical interface and paved the way for the eventual use of 3D computer images.
1960s

- Ivan Sutherland invents Sketchpad, the first interactive computer graphics program (a predecessor to computer-aided drafting), for design and engineering applications.

- Arthur Appel at IBM introduces hidden surface removal and shadow algorithms.

- Evans & Sutherland Corps. and GE start building flight simulators with raster graphics.
• Xerox PARC develops a "paint program."

1970s

• Edward Catmull introduces the z-buffer algorithm and texture mapping.

• Arcade games Pong and Pac Man become popular.
• CGI (Computer Generated Images) was first used in movies in 1973, in the science fiction film, Westworld.

Millennium Falcon in 1977 Star Wars

James Cameron's Alien used the raster wireframe model to render the image of navigation monitors.
1980s

- Adobe markets Photoshop
- Atari's Battlezone was the first 3D arcade game. From a futuristic-looking tank, players fought in a valley surrounded by mountains and volcanoes.
Autodesk releases AutoCAD, among the earliest computer-aided design software packages for personal computers.
1990s

- Computers have 24-bit raster display and hardware support for Gouraud shading.
- Dynamical systems that allowed programmers to animate collisions, friction and cause and effects are introduced.
- Pixar is first studio to fully embrace an entirely computer-generated film with Toy Story.
2000s

- Graphic software reaches a peak in quality and user accessibility.

- 3D modeling captures facial expressions, human face, hair, water, and other elements formerly difficult to render.

- Xbox360 and PS3 gaming consoles deliver unprecedented image realism in games.
4. The "3D"

• The term three-dimensional, or 3D, means that an object being described or displayed has three dimensions of measurement: width, height, and depth.

• An example of a two-dimensional object is a piece of paper on your desk with a drawing or writing on it, having no perceptible depth. A three-dimensional object is the can of soda next to it. The soft drink can is round (width and depth) and tall (height).
For centuries, artists have known how to make a painting appear to have real depth. A painting is inherently a two-dimensional object because it is nothing more than canvas with paint applied. Similarly, 3D computer graphics are actually two-dimensional images on a flat computer screen that provide an illusion of depth, or a third dimension.

2D + Perspective = 3D. The first computer graphics no doubt appeared similar to what’s shown in Figure above, where you can see a simple three-dimensional cube drawn with 12 line segments. What makes the cube look three-dimensional is perspective, or the angles between the lines that lend the illusion of depth.
5. Common 3D Effects (a.k.a. the tricks that make the picture on our monitors resemble to real world)

- **Perspective.** Perspective refers to the angles between lines that lend the illusion of three dimensions. Figure above, shows a three-dimensional cube drawn with lines.

- In the next Figure (above), on the other hand, the brain is given more clues as to the true orientation of the cube because of hidden line removal. You expect the front of an object to obscure the back of the object from view. For solid surfaces, we call this **hidden surface removal**.
• **Color and Shading.** To further our perception, we must move beyond line drawing and add color to create solid objects. By applying different colors to each side, as shown in Figure above, we gain a lot more perception of a solid object.

• **Light and Shadows.** By shading each side appropriately, we can give the cube the appearance of being one solid color (or material) but also show that it is illuminated by a light at an angle, as shown in Figure above.
Instead of plain-colored materials, you can have wood grains, cloth, bricks, and so on. This technique of applying an image to a polygon to supply additional detail is called **texture mapping**. The image you supply is called a **texture**, and the individual elements of the texture are called texels. Finally, the process of stretching texels over the surface of an object is called filtering. Figure above shows the now-familiar cube example with textures applied to each polygon.

**There is more.**
6. Common uses for 3D Graphics

- **Real time 3D.** The computer can process input as fast as or faster than the input is being supplied. For example, talking on the phone is a real-time activity in which humans participate. In contrast, writing a letter is not a real-time activity. Examples:
  - Games
  - Military simulations
  - Data visualization in medical, scientific or business use.
  - Basically any kind of application that has 3D graphics and is interactive.

- **Non real time 3D.** The most obvious example of non real time 3D graphics is animated movies. Rendering a single frame for a movie such as Toy Story or Shrek could take hours on a very fast computer, for example.
7. Vectors and Matrices

Additional reading: 3D Math Primer for Graphics and Game Development 1st Edition, Chapters 4, 5

Additional reading: 3D Math Primer for Graphics and Game Development 1st Edition, Chapter 7
What is Computer Graphics? Who are you mr. Pixel?

- Everything you see on your computer's screen, even the text you are reading right now (assuming you are reading this on an electronic display device, rather than a printout) is simply a two-dimensional array of pixels. If you take a screenshot of something on your screen, and blow it up, it will look very blocky.

- Each of these blocks is a pixel. The word “pixel” is derived from the term “Picture Element”. Every pixel on your screen has a particular color. A two-dimensional array of pixels is called an image.

- The purpose of graphics of any kind is therefore to determine what color to put in what pixels. This determination is what makes text look like text, buttons look like buttons, and so forth.

- Since all graphics are just a two-dimensional array of pixels, how does 3D work? 3D graphics is thus a system of producing colors for pixels that convince you that the scene you are looking at is a 3D world rather than a 2D image. The process of converting a 3D world into a 2D image of that world is called rendering.
• There are several methods for rendering a 3D world. The process used by real-time graphics hardware, such as that found in your computer, involves a very great deal of fakery. This process is called **rasterization**, and a rendering system that uses rasterization is called a **rasterizer**.

• In rasterizers all objects that you see are made out of a series of adjacent triangles that define the outer surface of the object. Such series of triangles are often called **geometry**, a **model** or a **mesh**.

• The process of rasterization has several phases. These phases are ordered into a **pipeline**, where triangles enter from the top and a 2D image is filled in at the bottom.

• OpenGL, thus, is an API for accessing a hardware-based rasterizer. As such, it conforms to the model for rasterization-based 3D renderers. A rasterizer receives a sequence of triangles from the user, performs operations on them, and writes pixels based on this triangle data. This is a simplification of how rasterization works in OpenGL, but it is useful for our purposes.
Triangles and vertices

- Triangles consist of 3 vertices. A **vertex** is a collection of arbitrary data. For the sake of simplicity (we will expand upon this later), let us say that this data must contain a point in three dimensional space. It may contain other data, but it must have at least this. Any 3 points that are not on the same line create a triangle, so the smallest information for a triangle consists of 3 three-dimensional points.

- A point in 3D space is defined by 3 numbers or coordinates. An X coordinate, a Y coordinate, and a Z coordinate. These are commonly written with parenthesis, as in (X, Y, Z).

**Rasterization Overview**  
*(OpenGL Simplified Overview)*

The rasterization pipeline, particularly for modern hardware, is very complex. This is a very simplified overview of this pipeline.

- **Clip Space Transformation.** The first phase of rasterization is to transform the vertices of each triangle into a certain region of space. Everything within this volume will be rendered to the output image, and everything that falls outside of this region will not be. This region corresponds to the view of the world that the user wants to render.

The volume that the triangle is transformed into is called, in OpenGL parlance, **clip space.** The positions of the triangle's vertices in clip space are called **clip coordinates.**
Clip coordinates are a little different from regular positions. A position in 3D space has 3 coordinates. A position in clip space has four coordinates. The first three are the usual X, Y, Z positions; the fourth component (W) of clip coordinates represents the visible range of clip space for that vertex. So the X, Y, and Z component of clip coordinates must be between [-W, W] to be a visible part of the world.

Because clip space is the visible transformed version of the world, any triangles that fall outside of this region are discarded. Any triangles that are partially outside of this region undergo a process called clipping. This breaks the triangle apart into a number of smaller triangles, such that the smaller triangles are all entirely within clip space. Hence the name “clip space.”
• **Normalized Coordinates.** Clip space is interesting, but inconvenient. The extent of this space is different for each vertex, which makes visualizing a triangle rather difficult. Therefore, clip space is transformed into a more reasonable coordinate space: normalized device coordinates.

This process is very simple. The X, Y, and Z of each vertex's position is divided by W to get normalized device coordinates. That is all. The space of normalized device coordinates is essentially just clip space, except that the range of X, Y and Z are [-1, 1]. The directions are all the same. The division by W is an important part of projecting 3D triangles onto 2D images.

![Coordinate System Diagram](image)

• **Window Transformation.** The next phase of rasterization is to transform the vertices of each triangle again. This time, they are converted from normalized device coordinates to window coordinates. As the name suggests, window coordinates are relative to the window that OpenGL is running within.

Even though they refer to the window, they are still three dimensional coordinates. The X goes to the right, Y goes up, and Z goes away, just as for clip space. The only difference is that the bounds for these coordinates depends on the viewable window. It should also be noted that while these are in window coordinates, none of the precision is lost. These are not integer coordinates; they are still floating-point values, and thus they have precision beyond that of a single pixel.
Scan Conversion. After converting the coordinates of a triangle to window coordinates, the triangle undergoes a process called **scan conversion**. This process takes the triangle and breaks it up based on the arrangement of window pixels over the output image that the triangle covers.

The center image shows the digital grid of output pixels; the circles represent the center of each pixel. The center of each pixel represents a sample: a discrete location within the area of a pixel. During scan conversion, a triangle will produce a **fragment** for every pixel sample that is within the 2D area of the triangle. The image on the right shows the fragments generated by the scan conversion of the triangle. This creates a rough approximation of the triangle's general shape.

The result of scan converting a triangle is a sequence of fragments that cover the shape of the triangle. Each fragment has certain data associated with it. This data contains the 2D location of the fragment in window coordinates, as well as the Z position of the fragment. This Z value is known as the depth of the fragment.
• **Fragment Processing.** This phase takes a fragment from a scan converted triangle and transforms it into one or more color values and a single depth value. The order that fragments from a single triangle are processed in is irrelevant; since a single triangle lies in a single plane, fragments generated from it cannot possibly overlap. However, the fragments from another triangle can possibly overlap. Since order is important in a rasterizer, the fragments from one triangle must all be processed before the fragments from another triangle.

• **Fragment Writing.** After generating one or more colors and a depth value, the fragment is written to the destination image. (a image that is afterward shown on the screen)

## Colors

• Previously, a pixel was stated to be an element in a 2D image that has a particular color. A color can be described in many ways. In computer graphics, the usual description of a color is as a series of numbers on the range \([0, 1]\). Each of the numbers corresponds to the intensity of a particular reference color; thus the final color represented by the series of numbers is a mix of these reference colors.

• The set of reference colors is called a **colorspace.** The most common color space for screens is RGB, where the reference colors are Red, Green and Blue.

• So a pixel in OpenGL is defined as 3 values on the range \([0, 1]\) that represent a color in a linear RGB colorspace. By combining different intensities of this 3 colors, we can generate millions of different color shades. This will get extended slightly, as we deal with transparency later.
Shader

- A **shader** is a program designed to be run on a renderer as part of the rendering operation. Regardless of the kind of rendering system in use, shaders can only be executed at certain points in that rendering process. These shader stages represent hooks where a user can add arbitrary algorithms to create a specific visual effect.

- Shaders for OpenGL are run on the actual rendering hardware. This can often free up valuable CPU time for other tasks, or simply perform operations that would be difficult if not impossible without the flexibility of executing arbitrary code.

- There are a number of shading languages available to various APIs. The one used in during our course is the primary shading language of OpenGL. It is called, unimaginatively, the OpenGL Shading Language, or GLSL. for short. It looks deceptively like C, but it is very much not C.

Additional reading:
9. What is OpenGL?

• OpenGL stands for Open Graphics Library

• OpenGL is window and operating system independent

• OpenGL does not include any functions for window management, user interaction, and file I/O

• Host environment is responsible for window management
OpenGL as an API

- OpenGL is usually thought of as an **Application Programming Interface (API)**. The OpenGL API has been exposed to a number of languages. But the one that they all ultimately use at their lowest level is the C API.

- The API, in C, is defined by a number of typedefs, #defined enumerator values, and functions. The typedefs define basic GL types like GLint, GLfloat and so forth. These are defined to have a specific bit depth.

Complex aggregates like structs are never directly exposed in OpenGL. Any such constructs are hidden behind the API. This makes it easier to expose the OpenGL API to non-C languages without having a complex conversion layer. In C++, if you wanted an object that contained an integer, a float, and a string, you would create it and access it like this:

```cpp
struct Object
{
    int count;
    char *name;
};

// Create the storage for the object.
Object newObject;
// Put data into the object.
newObject.count = 5;
newObject.name = "Some String";
```
• In OpenGL, you would use an API that looks more like this:

```c
//Create the storage for the object
GLuint objectName;
glGenObject(1, &objectName);

//Put data into the object.
glBindObject(GL_MODIFY, objectName);
glObjectParameteri(GL_MODIFY, GL_OBJECT_COUNT, 5);
glObjectParameterf(GL_MODIFY, GL_OBJECT_OPACITY, 0.4f);
glObjectParameters(GL_MODIFY, GL_OBJECT_NAME, "Some String");
```

None of the above are actual OpenGL commands, of course. This is simply an example of what the interface to such an object would look like.

• OpenGL owns the storage for all OpenGL objects. Because of this, the user can only access an object by reference. Almost all OpenGL objects are referred to by an unsigned integer (the GLuint). Objects are created by a function of the form `glGen*`, where * is the type of the object. The first parameter is the number of objects to create, and the second is a GLuint* array that receives the newly created object names.

• To modify most objects, they must first be bound to the context. Many objects can be bound to different locations in the context; this allows the same object to be used in different ways. These different locations are called targets; all objects have a list of valid targets, and some have only one. In the above example, the fictitious target “GL_MODIFY” is the location where objectName is bound.
The **glObjectParameter** family of functions set parameters within the object bound to the given target. Note that since OpenGL is a C API, it has to name each of the differently typed variations differently. So there is `glObjectParameteri` for integer parameters, `glObjectParameterf` for floating-point parameters, and so forth.

The Structure of OpenGL

- The OpenGL API is defined as a state machine. Almost all of the OpenGL functions set or retrieve some state in OpenGL. The only functions that do not change state are functions that use the currently set state to cause rendering to happen.

- You can think of the state machine as a very large struct with a great many different fields. This struct is called the **OpenGL context**, and each field in the context represents some information necessary for rendering.

- Objects in OpenGL are thus defined as a list of fields in this struct that can be saved and restored. Binding an object to a target within the context causes the data in this object to replace some of the context's state. Thus after the binding, future function calls that read from or modify this context state will read or modify the state within the object.

- Objects are usually represented as GLuint integers; these are handles to the actual OpenGL objects. The integer value 0 is special; it acts as the object equivalent of a NULL pointer. Binding object 0 means to unbind the currently bound object. This means that the original context state, the state that was in place before the binding took place, now becomes the context state.
Let us say that this represents some part of an OpenGL context's state:

```c
struct Values
{
    int iValue1;
    int iValue2;
};
struct OpenGL_Context
{
    ...
    Values *pMainValues;
    Values *pOtherValues;
    ...
};
OpenGL_Context context;
```

To create a `Values` object, you would call something like `glGenValues`. You could bind the `Values` object to one of two targets: `GL_MAIN_VALUES` which represents the pointer `context.pMainValues`, and `GL_OTHER_VALUES` which represents the pointer `context.pOtherValues`. You would bind the object with a call to `glBindValues`, passing one of the two targets and the object. This would set that target's pointer to the object that you created.

There would be a function to set values in a bound object. Say, `glValueParam`. It would take the target of the object, which represents the pointer in the context. It would also take an enum representing which value in the object to change. The value `GL_VALUE_ONE` would represent `iValue1`, and `GL_VALUE_TWO` would represent `iValue2`. 
The Structure of OpenGL

• To be technical about it, OpenGL is not an API; it is a specification. A document. The C API is merely one way to implement the spec. The specification defines the initial OpenGL state, what each function does to change or retrieve that state, and what is supposed to happen when you call a rendering function.

• The specification is written by the **OpenGL Architectural Review Board (ARB)**, a group of representatives from companies like Apple, NVIDIA, and AMD (the ATI part), among others. The ARB is part of the Khronos Group.

• The specification is a very complicated and technical document. It describes results, not implementation. Just because the spec says that X will happen does not mean that it actually does. What it means is that the user should not be able to tell the difference. If a piece of hardware can provide the same behavior in a different way, then the specification allows this, so long as the user can never tell the difference.

• **OpenGL Implementations.** While the OpenGL ARB does control the specification, it does not control OpenGL's code. OpenGL is not something you download from a centralized location. For any particular piece of hardware, it is up to the developers of that hardware to write an **OpenGL Implementation** for that hardware. Implementations, as the name suggests, implement the OpenGL specification, exposing the OpenGL API as defined in the spec.
Who controls the OpenGL implementation is different for different operating systems. On Windows, OpenGL implementations are controlled virtually entirely by the hardware makers themselves. On Mac OSX, OpenGL implementations are controlled by Apple; they decide what version of OpenGL is exposed and what additional functionality can be provided to the user. Apple writes much of the OpenGL implementation on Mac OSX, which the hardware developers writing to an Apple-created internal driver API.

OpenGL Versions. There are many versions of the OpenGL Specification(1.0 in 1992, 4.3 in 2013). OpenGL versions are not like most Direct3D versions, which typically change most of the API. Code that works on one version of OpenGL will almost always work on later versions of OpenGL.

The only exception to this deals with OpenGL 3.0 and above, relative to previous versions. v3.0 deprecated a number of older functions(Fixed Function Pipeline), and v3.1 removed most of those functions from the API. This also divided the specification into 2 variations (called profiles): core and compatibility. The compatibility profile retains all of the functions removed in 3.1, while the core profile does not. Theoretically, OpenGL implementations could implement just the core profile; this would leave software that relies on the compatibility profile non-functional on that implementation.

As a practical matter, none of this matters at all. No OpenGL driver developer is going to ship drivers that only implement the core profile. So in effect, this means nothing at all; all OpenGL versions are all effectively backwards compatible.

Additional reading: [http://www.arcsynthesis.org/gltut/Basics/Intro%20What%20is%20OpenGL.html](http://www.arcsynthesis.org/gltut/Basics/Intro%20What%20is%20OpenGL.html)
10. More on Shaders...

- The OpenGL Shading Language (GLSL) is now a fundamental and integral part of the OpenGL API. Going forward, every program written using OpenGL will internally utilize one or several GLSL programs. These "mini-programs" written in GLSL are often referred to as shader programs, or simply shaders. A shader program is one that runs on the **GPU (Graphics Processing Unit)**, and as the name implies, it (typically) implements the algorithms related to the lighting and shading effects of a 3-dimensional image. However, shader programs are capable of doing much more than just implementing a shading algorithm. They are also capable of performing animation, tessellation, and even generalized computation.

*The field of study dubbed GPGPU (General Purpose Computing on Graphics Processing Units) is concerned with utilization of GPUs (often using specialized APIs such as CUDA or OpenCL) to perform general purpose computations such as fluid dynamics, molecular dynamics, cryptography, and so on.*

- Shader programs are designed to be executed directly on the GPU and often in parallel. For example, a fragment shader might be executed once for every pixel, with each execution running simultaneously on a separate GPU thread. The number of processors on the graphics card determines how many can be executed at one time. This makes shader programs incredibly efficient, and provides the programmer with a simple API for implementing highly parallel computation.

<table>
<thead>
<tr>
<th>Model</th>
<th>Unified Shader Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GT 430</td>
<td>96</td>
</tr>
<tr>
<td>GeForce GTS 450</td>
<td>192</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>480</td>
</tr>
</tbody>
</table>
Shader programs are intended to replace parts of the OpenGL architecture referred to as the fixed-function pipeline. The default lighting/shading algorithm was a core part of this fixed function pipeline. When we, as programmers, wanted to implement more advanced or realistic effects, we used various tricks to force the fixed-function pipeline into being more flexible than it really was.

The advent of GLSL helped by providing us with the ability to replace this "hard-coded" functionality with our own programs written in GLSL, thus giving us a great deal of additional flexibility and power.
To the left is a simplified diagram of the pipeline for OpenGL from version 3.2 forwards. Blue areas denote parts of the pipeline which can be programmed by the user in the form of Shader programs, thus the Programmable Pipeline.

- Vertex shader (OpenGL 2.0) – A vertex shader operates on individual vertices, one vertex at a time. At minimum the vertex shader must produce the computation of a clip-space vertex position; everything else is user-defined.

- Tessellation shaders (OpenGL 4.0)/Geometry shader (OpenGL 3.2) – Long story short, these shaders can be used to generate additional graphics primitives on the fly.

- Fragment shader – Processes the Fragment from the rasterization process into a set of colors and a single depth value.

12. "Hello triangle!" - Your first OpenGL App. Sample1
Let's analyze SAMPLE1. The included header files are:

```cpp
#include <glut.h>
#include <glew.h>
#include <iostream>
#include <fstream>
#include <string>
#include "cTimer.h"
using namespace std;
```

- `<glut.h>` - The header file for the FreeGLUT library. FreeGLUT is a fairly simple OpenGL initialization system. It creates and manages a single window; all OpenGL commands refer to this window. Because windows in various GUI systems need to have certain book-keeping done, how the user interfaces with this is rigidly controlled.

What that means is that you don't have to bother with creating and managing a window for your application. FreeGLUT does it for yourself. It should be noted however that FreeGLUT is not solution a for a serious project as it has it's drawbacks. We initialize FreeGLUT at the very top of the “main()” function:

```cpp
//GLUT INIT
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
glutInitWindowSize(512,512);
glutCreateWindow("Triangle Test");
```
FreeGLUT has a very useful “callback functions” mechanism. It lets you specify the function that you would like to be called in specific situations like mouse movement, keyboard input and so on. Let's have a look at the code below:

```c
glutDisplayFunc(render); // we tell FreeGLUT that we'd like it to call function "render()" for frame rendering
```

```c
- we tell FreeGLUT that we'd like it to call function "reshape()" whenever the window is resized. This allows us to make whatever OpenGL calls are necessary to keep the window's size in sync with OpenGL.
```

```c
glutTimerFunc(1, update, 0); // we tell FreeGLUT to call each millisecond the function "update()" where we can write our logic.
```

- <glew.h> - The header file for the GLEW (OpenGL Extension Wrangler) library. The OpenGL ABI (application binary interface) is frozen to OpenGL version 1.1 on Windows. Unfortunately for Windows developers, that means that it is not possible to link directly to functions that are provided in newer versions of OpenGL. Instead, one must get access to these functions by acquiring a function pointer at runtime. Doing it manually is a little painful. This is why there is GLEW.

- GLEW library requires to be initialized. In our sample we do this in the “main()” function body:

```c
    // GLEW INIT
    glewInit();
    GLenum err = glewInit();
    if (GLEW_OK != err)
    {
        /* Problem: glewInit failed, something is seriously wrong. */
        cout << "glewInit failed, aborting." << endl;
    }
```
Including the glew.h header file provides declarations for the OpenGL functions as function pointers, so all function entry points are available at compile time. At run time, the glewInit() function will scan the OpenGL library, and initialize all available function pointers. If a function is not available, the code will compile, but the function pointer will not be initialized.

The other included files are just standard header files that deal with console input/output, file reading. "cTimer.h" implements the timer functionality.

Let's continue with reviewing the body of "main()" function. At the top we initialize the FreeGLUT and the GLEW libraries, and then we detect the OpenGL version available on our machine:

```cpp
//GET GL VERSION
GLint major,minor;
glGetIntegerv(GL_MAJOR_VERSION,&major);
glGetIntegerv(GL_MINOR_VERSION,&minor);
cout << "GL VENDOR:" << glGetString(GL_VENDOR) << endl;
cout << "GL RENDERER:" << glGetString(GL_RENDERER) << endl;
cout << "GL VERSION(string):" << glGetString(GL_VERSION) << endl;
cout << "GL VERSION(integer):" << major<< "." << minor << endl;
cout << "GL VERSION:" << glGetString(GL_SHADING_LANGUAGE_VERSION) << endl;
```
Here's one of the possible outputs to the console:

```
GL_VENDOR: ATI Technologies Inc.
GL_RENDERER: ATI Mobility Radeon HD 2400 XT
GL_VERSION<string>: 3.3.11653 Compatibility Profile Context
GL_VERSION<integer>: 3.3
GL_VERSION: 3.30
```

- Knowing the version of OpenGL that your hardware is capable of is vital when designing graphical applications, because you have to activate or deactivate different rendering options depending if the target hardware has support for it.

- After determining the OpenGL version, we query and output to the console all available *OpenGL Extensions*:

  ```
  //GET GL EXTENSIONS
  GLint nExtensions;
  glGetIntegerv(GL_NUM_EXTENSIONS, &nExtensions);
  for (int i = 0; i < nExtensions; i++)
  {
    cout << "ext:" << glGetStringi(GL_EXTENSIONS, i) << endl;
  }
  ```

- OpenGL is an evolving API. New versions of the OpenGL specification are regularly released by the Khronos Group, each of which extends the API to support various new features.
• In addition to the features required by the core API, GPU vendors may provide additional functionality in the form of extensions. Extensions may introduce new functions and new constants, and may relax or remove restrictions on existing OpenGL functions. Vendors can use extensions to expose custom APIs without needing support from other vendors or the Khronos Group as a whole, which greatly increases the flexibility of OpenGL.

• Each extension is associated with a short identifier, based on the name of the company which developed it. For example, Nvidia's identifier is NV, which is part of the extension name GL_NV_half_float, the constant GL_HALF_FLOAT_NV, and the function glVertex2hNV(). If multiple vendors agree to implement the same functionality using the same API, a shared extension may be released, using the identifier EXT. In such cases, it could also happen that the Khronos Group's Architecture Review Board gives the extension their explicit approval, in which case the identifier ARB is used.

```
ext:GL_ARB_transpose_matrix
ext:GL_ARB_uniform_buffer_object
ext:GL_ARB_vertex_array_bgra
ext:GL_ARB_vertex_array_object
ext:GL_ARB_vertex_buffer_object
ext:GL_ARB_vertex_program
ext:GL_ARB_vertex_shader
ext:GL_ARB_vertex_types_2_10_10_10_rev
ext:GL_ARB_viewport_array
ext:GL_ARB_window_pos
ext:GL_ATI_draw_buffers
ext:GL_ATI_envmap_bumpmap
ext:GL_ATI_fragment_shader
ext:GL_ATI_meminfo
ext:GL_ATI_separate_stencil
ext:GL_ATI_texture_compression_3dc
ext:GL_ATI_texture_env_combine3
ext:GL_ATI_texture_float
ext:GL_ATI_texture_mirror_once
ext:GL_EXT_abgr
ext:GL_EXT_bgra
ext:GL_EXTBindable_uniform
```

• The features introduced by each new version of OpenGL are typically formed from the combined features of several widely-implemented extensions, especially extensions of type ARB or EXT.
So one way to check for an extension is to loop through all of them and see if the one you need is present on target hardware. An alternative is to use GLEW. An example of checking if an extension is available is given below:

```c
if ( ! GLEW_ARB_vertex_program )
{
    fprintf(stderr, "ARB_vertex_program is missing!\n");
    ...
}
```

Additional reading: *OpenGL 4.0 Shading Language Cookbook, pages 6-14*

- The "main()" function ends with the following:

  ```c
  //INIT GL RESOURCES
  init();

  //ENTER LOOP
  glutMainLoop();
getchar();
return 0;
  ```

- `init()` - is a call to a function which loads all the necessary graphics data for this sample. We will talk on it shortly.
glutMainLoop() - enters an infinite loop during which our window will be updated with OpenGL graphics. Inside this loop, which is managed by FreeGLUT library, the render(), reshape() and update() functions will be called at respective time by FreeGLUT given that we supplied it with pointers to them.

render() function

Earlier during FreeGLUT initialization we supplied it with a pointer to a function that will implement the actual rendering. We named this function render(). Let's see how it looks:

```c
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);

glUseProgram(theProgram);

glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

glDrawArrays(GL_TRIANGLES, 0, 3);

glDisableVertexAttribArray(0);

glUseProgram(0);

glutSwapBuffers();
```
• The first two lines clear the screen. **glClearColor** is one of those state setting functions; it sets the color that will be used when clearing the screen. It sets the clearing color to black. **glClear** does not set OpenGL state; it causes the screen to be cleared. The **GL_COLOR_BUFFER_BIT** parameter means that the clear call will affect the color buffer, causing it to be cleared to the current clearing color we set in the previous function.

• The next line sets the current shader program to be used by all subsequent rendering commands. We will go into detail as to how this works later.

• The next three commands all set state. These command set up the coordinates of the triangle to be rendered. They tell OpenGL the location in memory that the positions of the triangle will come from. The specifics of how these work will be detailed later.

• The **glDrawArrays** function is, as the name suggests, a rendering function. It uses the current state to generate a stream of vertices that will form triangles.

• The next two lines are simply cleanup work, undoing some of the setup that was done for the purposes of rendering.

• The last line, **glutSwapBuffers**, is a FreeGLUT command, not an OpenGL command. The OpenGL framebuffer, as we set up in "main()", is double-buffered. This means that the image that are currently being shown to the user is not the same image we are rendering to. Thus, all of our rendering is hidden from view until it is shown to the user. This way, the user never sees a half-rendered image. **glutSwapBuffers** is the function that causes the image we are rendering to be displayed to the user.
A word on Double Buffering

• When rendering directly to a part of the memory that is being displayed at the monitor an effect called *flickering* can appear. This effect has to do with the possibility for the user to see the image as it is being constructed.

• **Double buffering** is a technique in which there are two buffers, the one from that holds the picture we see on the screen(A) and the other, an off-screen buffer(B), is used for rendering operations. When rendering to the off-screen buffer(B) is completed, the buffer flip occurs and the two buffers change their roles. So reading already takes place from buffer B and writing takes places to buffer A.

• However, double buffering introduces another source of flicker known as *tearing*. Tearing becomes apparent when displaying fast-moving objects in an animation. The buffer flip occurs while the video display is still reading data from the framebuffer, causing the top half of the display to show the previous frame while the bottom half shows the updated frame. If you are updating the framebuffer particularly quickly you may notice three or more such "tears" in the display.

• Vertical synchronization is an option in most systems to avoid flickering, wherein the video card is prevented from doing anything visible to the display memory until after the monitor finishes its current refresh cycle. During the vertical blanking interval, the driver orders the video card to either rapidly copy the off-screen graphics area into the active display area, or treat both memory areas as displayable, and simply switch the pointers of buffers between them (*page flipping*).
Where does the triangle data come from?

- When we talked about how the rendering process in OpenGL works, the first stage in the rasterization pipeline was the transforming of vertices to clip space. Before OpenGL can do this however, it must receive a list of vertices. So the very first stage of the pipeline is sending triangle data to OpenGL. We specified this data at the top of our program:

```cpp
const float vertexPositions[] = {
    0.75f, 0.75f, 0.0f, 1.0f,
    0.75f, -0.75f, 0.0f, 1.0f,
    -0.75f, -0.75f, 0.0f, 1.0f,
};
```

- Each line of 4 values represents a 4D position of a vertex. These are four dimensional because, as you may recall, clip-space is 4D as well. These vertex positions are already in clip space. What we want OpenGL to do is render a triangle based on this vertex data. Since every 4 floats represents a vertex's position, we have 3 vertices: the minimum number for a triangle.

- Even though we have this data, OpenGL cannot use it directly. OpenGL has some limitations on what memory it can read from. You can allocate vertex data all you want yourself; OpenGL cannot directly see any of your memory. Therefore, the first step is to allocate some memory that OpenGL can see, and fill that memory with our data. This is done with something called a **buffer object.**
• A buffer object is a linear array of memory, managed and allocated by OpenGL at the behest of the user. The content of this memory is controlled by the user, but the user has only indirect control over it. Think of a buffer object as an array of GPU memory. The GPU can read this memory quickly, so storing data in it has performance advantages.

• We initialized our buffer object at the end of “init()” function:

```c
    glGenBuffers(1, &positionBufferObject);
    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

• The first line creates the buffer object, storing the handle to the object in the global variable `positionBufferObject`. Though the object now exists, it does not own any memory yet. That is because we have not allocated any with this object.

• The `glBindBuffer` function binds the newly-created buffer object to the `GL_ARRAY_BUFFER` binding target. As mentioned earlier, objects in OpenGL usually have to be bound to the context in order for them to do anything, and buffer objects are no exception.

• The `glBufferData` function performs two operations. It allocates memory for the buffer currently bound to `GL_ARRAY_BUFFER`, which is the one we just created and bound. We already have some vertex data; the problem is that it is in our memory rather than OpenGL's memory. The `sizeof(vertexPositions)` uses the C++ compiler to determine the byte size of the vertexPositions array. We then pass this size to `glBufferData` as the size of memory to allocate for this buffer object. Thus, we allocate enough GPU memory to store our vertex data.
• The other operation that `glBufferData` performs is copying data from our memory array into the buffer object. The third parameter controls this. If this value is not NULL, as in this case, `glBufferData` will copy the data referenced by the pointer into the buffer object. After this function call, the buffer object stores exactly what `vertexPositions` stores. We'll look into the fourth parameter (GL_STATIC_DRAW) later.

• The second bind buffer call is simply cleanup. By binding the buffer object 0 to GL_ARRAY_BUFFER, we cause the buffer object previously bound to that target to become unbound from it.

• This is all just to get the vertex data in the GPU's memory. But buffer objects are not formatted; as far as OpenGL is concerned, all we did was allocate a buffer object and fill it with random binary data. We now need to do something that tells OpenGL that there is vertex data in this buffer object and what form that vertex data takes.

We do this in our `render()` function:

```c
glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
```

• The first function we have seen before. It simply says that we are going to use this buffer object.

• The second function, `glEnableVertexAttribArray` will be explained a little later.
• The third function is the real key. **`glVertexAttribPointer`**, despite having the word “Pointer” in it, does not deal with pointers. Instead, it deals with buffer objects.

• When rendering, OpenGL pulls vertex data from arrays stored in buffer objects. What we need to tell OpenGL is what format our vertex array data in the buffer object is stored in. That is, we need to tell OpenGL how to interpret the array of data stored in the buffer.

• In our case, our data is formatted as follows:

  • Our position data is stored in 32-bit floating point values using the C/C++ type float.
  • Each position is composed of 4 of these values.
  • There is no space between each set of 4 values. The values are tightly packed in the array.
  • The first value in our array of data is at the beginning of the buffer object.

• The **`glVertexAttribPointer`** function tells OpenGL all of this. The third parameter specifies the base type of a value. In this case, it is **`GL_FLOAT`**, which corresponds to a 32-bit floating-point value. The second parameter specifies how many of these values represent a single piece of data. In this case, that is 4. The fifth parameter specifies the spacing between each set of values. In our case, there is no space between values, so this value is 0. And the sixth parameter specifies the byte offset from the value in the buffer object is at the front, which is 0 bytes from the beginning of the buffer object. The fourth parameter is something that we will look at in later. The first parameter is something we will look very soon.
• Once OpenGL knows where to get its vertex data from, it can now use that vertex data to render.

    glDrawArrays(GL_TRIANGLES, 0, 3);

• This function seems very simple on the surface, but it does a great deal. The second and third parameters represent the start index and the number of indices to read from our vertex data. The 0th index of the vertex array (defined with glVertexAttribPointer) will be processed, followed by the 1st and 2nd indices. That is, it starts with the 0th index, and reads 3 vertices from the arrays.

**Vertex processing, the Vertex Shader**

• Now that we can tell OpenGL what the vertex data is, we come to the next stage of the pipeline: vertex processing. This is one of two programmable stages that we will cover during our course, so this involves the use of a shader.

• A shader as already stated previously, is nothing more than a program that runs on the GPU. There are several possible shader stages in the pipeline, and each has its own inputs and outputs. The purpose of a shader is to take its inputs, as well as potentially various other data, and convert them into a set of outputs.

• Each shader is executed over a set of inputs. It is important to note that a shader, of any stage, operates completely independently of any other shader of that stage. There can be no crosstalk between separate executions of a shader. Execution for each set of inputs starts from the beginning of the shader and continues to the end.
Vertex shaders, as the name implies, operate on vertices. Specifically, each invocation of a vertex shader operates on a single vertex. These shaders must output, among any other user-defined outputs, a **clip-space position for that vertex**. How this clip-space position is computed is entirely up to the shader.

At the very beginning of our "init()" function we load two shaders, one of them is a vertex shader. It is stored in the file "basic.vert". Here's how it looks:

```glsl
#version 330
layout(location = 0) in vec4 position;

void main()
{
    gl_Position = position;
}
```

This looks fairly simple. The first line states that the version of GLSL used by this shader is **version 3.30**. A version declaration is required for all GLSL shaders. The next line defines an input to the vertex shader. The input is a variable named `position` and is of type `vec4`: a 4-dimensional vector of floating-point values. It also has a **layout location of 0**; we'll explain that a little later.

As with C, a shader's execution starts with the `main` function. This shader is very simple, copying the input position into something called `gl_Position`. This is a variable that is not defined in the shader; that is because it is a standard variable defined in every vertex shader. If you see an identifier in a GLSL shader that starts with “gl_”, then it must be a built-in identifier. You cannot make an identifier that begins with “gl_”; you can only use ones that already exist.
• **gl_Position** is defined as: `out vec4 gl_Position;

• Recall that the minimum a vertex shader must do is generate a clip-space position for the vertex. That is what **gl_Position** is: the clip-space position of the vertex. Since our input position data is already a clip-space position, this shader simply copies it directly into the output.

**Vertex Attributes.** Shaders have inputs and outputs. Think of these like function parameters and function return values. If the shader is a function, then it is called with input values, and it is expected to return a number of output values.

• Inputs to and outputs from a shader stage come from somewhere and go to somewhere. Thus, the input **position** in the vertex shader must be filled in with data somewhere. So where does that data come from? Inputs to a vertex shader are called **vertex attributes**.

• When rendering starts, vertex data in a buffer object is read based on setup work done by **glVertexAttribPointer**. This function describes where the data for an attribute comes from. Each input to a vertex shader has an index location called an **attribute index**. The input in this shader was defined with this statement:

```
layout(location = 0) in vec4 position;
```

• The layout location part assigns the attribute index of 0 to position.
In code, when referring to attributes, they are always referred to by **attribute index**. The functions `glEnableVertexAttribArray`, `glDisableVertexAttribArray`, and `glVertexAttribPointer` all take as their first parameter an **attribute index**. We assigned the **attribute index** of the **position** attribute to 0 in the vertex shader, so the call to `glEnableVertexAttribArray(0)` enables the attribute index for the position attribute.

Here is a diagram of the data flow to the vertex shader:
• Without the call to glEnableVertexAttribArray, calling glVertexAttribPointer on that attribute index would not mean much. The enable call does not have to be called before the vertex attribute pointer call, but it does need to be called before rendering. If the attribute is not enabled, it will not be used during rendering.

**Rasterization**

• All that has happened thus far is that 3 vertices have been given to OpenGL and it has transformed them with a vertex shader into 3 positions in clip-space. Next, the vertex positions are transformed into normalized-device coordinates by dividing the 3 XYZ components of the position by the W component. In our case, W for our 3 positions was 1.0, so the positions are already effectively in normalized-device coordinate space.

• After this, the vertex positions are transformed into **window coordinates**. This is done with something called the **viewport transform**. This is so named because of the function used to set it up, `glViewport`. Remember when we set up the FreeGLUT we wrote the following code:

```c
    glutReshapeFunc(reshape);
```

• We asked FreeGLUT to call the reshape() function every time the size of our window changes. And here's the body of reshape() function:

```c
    void reshape(int w, int h)
    {
        glViewport(0,0,(GLsizei)w,(GLsizei)h);
    }
```
• This tells OpenGL what area of the available area we are rendering to. In this case, we change it to match the full available area. Without this function call, resizing the window would have no effect on the rendering. Also, make note of the fact that we make no effort to keep the aspect ratio constant; shrinking or stretching the window in a direction will cause the triangle to shrink and stretch to match.

• Window coordinates are in a lower-left coordinate system. So the point (0, 0) is the bottom left of the window. This function takes the bottom left position as the first two coordinates, and the width and height of the viewport rectangle as the other two coordinates.

• Once in window coordinates, OpenGL can now take these 3 vertices and scan-convert it into a series of fragments. In order to do this however, OpenGL must decide what the list of vertices represents.

OpenGL can interpret a list of vertices in a variety of different ways. The way OpenGL interprets vertex lists is given by the draw command:

```c
       glDrawArrays(GL_TRIANGLES, 0, 3);
```

• The enum **GL_TRIANGLES** tells OpenGL that every 3 vertices of the list should be used to build a triangle. Since we passed only 3 vertices, we get 1 triangle.
The diagram below illustrates the data flow to the rasterizer. If we rendered 6 vertices, then we would get 2 triangles.

- GL_TRIANGLES is the primitive type we want to render. Other primitive types can be GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES. We'll talk about them later.
Fragment processing, the Fragment Shader

A fragment shader is used to compute the output color(s) of a fragment. The inputs of a fragment shader include the window-space XYZ position of the fragment. It can also include user-defined data, which we will see later.

Our fragment shader looks like this:

```
#version 330
out vec4 outputColor;

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

As with the vertex shader, the first line states that the shader uses GLSL version 3.30. The next line specifies an output for the fragment shader. The output variable is of type vec4.

The main function simply sets the output color to a 4-dimensional vector, with all of the components as 1.0f. This sets the Red, Green, and Blue components of the color to full intensity, which is 1.0; this creates the white color of the triangle. The fourth component is something we will see in later samples.

Though all fragment shaders are provided the window-space position of the fragment, this one does not need it. So it simply does not use it. After the fragment shader executes, the fragment output color is written to the output image.
Note: In the section on vertex shaders, we had to use the `layout(location = #)` syntax in order to provide a connection between a vertex shader input and a vertex attribute index. This was required in order for the user to connect a vertex array to a vertex shader input. So you may be wondering where the connection between the fragment shader output and the screen comes in.

OpenGL recognizes that, in a lot of rendering, there is only one logical place for a fragment shader output to go: the current image being rendered to (in our case, the screen). Because of that, if you define only one output from a fragment shader, then this output value will automatically be written to the current destination image. It is possible to have multiple fragment shader outputs that go to multiple different destination images; this adds some complexity, similar to attribute indices. But that is for another time.
Compiling the Shader

• We've covered quite a lot of our first OpenGL application. The only thing missing is how do we integrate the Shader programs in our OpenGL app. Our “init()” function does that at the very beginning. The Vertex Shader is loaded from the “basic.vert” file and the fragment shader is loaded from “basic.frag” file.

• Compiling a shader involves creating a shader object, providing the source code (as a string or set of strings) to the shader object, and asking the shader object to compile the code. So, first we load the shader code from the file:

```cpp
GLchar * shaderCode;
// Load contents of file into shaderCode here...
ifstream inFile( "shader/basic.vert", ifstream::in );
if( !inFile ) {
    fprintf(stderr, "Error opening file: shader/basic.vert\n" );
    exit(1);
}
shaderCode = (char *)malloc(10000);
int i = 0;
while( inFile.good() ) {
    int c = inFile.get();
    shaderCode[i++] = c;
}
inFile.close();
shaderCode[i++] = '\0';
```
• Next, we create the shader object. An OpenGL shader object is, as the name suggests, an object. So the first step is to create the object with `glCreateShader`.

```c
// Create the shader object
GLuint vertShader = glCreateShader( GL_VERTEX_SHADER );
if( 0 == vertShader )
{
    fprintf(stderr, "Error creating vertex shader.\n");
    exit(1);
}
```

The argument to `glCreateShader` is the type of shader, and can be one of the following: `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, `GL_TESS_EVALUATION_SHADER`, or `GL_TESS_CONTROL_SHADER`. In this case, since we are compiling a vertex shader, we use `GL_VERTEX_SHADER`. This function returns the value used for referencing the vertex shader object, sometimes called the object "handle". We store that value in the variable `vertShader`. If an error occurs while creating the shader object, this function will return 0, so we check for that and if it occurs, we print an appropriate message and terminate.
• Next step, is to load the source code into the shader object:

```c
// Load the source code into the shader object
const GLchar* codeArray[] = {shaderCode};
glShaderSource( vertShader, 1, codeArray, NULL );
free(shaderCode); // can be removed from book.
```

The `glShaderSource()` function is designed to accept an array of strings in order to support the option of compiling multiple files at once. So before we call `glShaderSource`, we place a pointer to our source code into an array named `codeArray`. The first argument to `glShaderSource` is the handle to the shader object. The second is the number of source code strings that are contained in the array. The third argument is a pointer to an array of source code strings. The final argument is an array of GLint values that contains the length of each source code string in the previous argument. In this case, we pass a value of NULL, which indicates that each source code string is terminated by a null character.

• Then, we compile the shader:

```c
// Compile the shader
glCompileShader( vertShader );
```

Of course, depending on the correctness of the source code, the compilation may fail, so the next step is to check whether or not the compilation was successful. We can query for the compilation status by calling `glGetShaderiv`, which is a function for querying the attributes of a shader object. In this case we are interested in the compilation status, so we use GL_COMPILE_STATUS as the second argument. The first argument is of course the handle to the shader object, and the third argument is a pointer to an integer where the status will be stored. The function provides a value of either GL_TRUE or GL_FALSE in the third argument, indicating whether or not the compilation was successful.
We verify the compilation status using the following code:

```c
// Check compilation status
GLint result;
glGetShaderiv( vertShader, GL_COMPILE_STATUS, &result );
if( GL_FALSE == result )
{
    fprintf( stderr, "Vertex shader compilation failed!\n" );
    GLint logLen;
    glGetShaderiv( vertShader, GL_INFO_LOG_LENGTH, &logLen );
    if( logLen > 0 )
    {
        char * log = (char *)malloc(logLen);
        GLsizei written;
        glGetShaderInfoLog(vertShader, logLen, &written, log);
        fprintf(stderr, "Shader log: \n%s", log);
        free(log);
    }
}
```

Shader objects can be deleted when no longer needed by calling `glDeleteShader`. This frees the memory used by the shader and invalidates its handle. Note that if a shader object is already attached to a program object (see Linking a shader program), it will not be immediately deleted, but flagged for deletion when it is detached from the program object.
Linking a shader program

• After we loaded both the vertex and the fragment shaders from files (fragment shader loading goes through the same steps except that you specify GL_FRAGMENT_SHADER in the glCreateShader function) we can actually install them into the OpenGL pipeline. We need to link them together into a shader program.

• Linking involves steps that are similar to those involved in compiling a shader. We attach each shader object to a new shader program object and then tell the shader program object to link (making sure that the shader objects are compiled before linking).

• We start by calling `glCreateProgram` to create an empty program object. This function returns a handle to the program object, which we store in a variable named `theProgram`. If an error occurs with program creation, the function will return 0. We check for that, and if it occurs, we print an error message and exit.

```c
// Create the program object
theProgram= glCreateProgram();
if( 0 == theProgram )
{
    fprintf(stderr, "Error creating program object.\n");
    exit(1);
}
```
• Next, we attach each shader to the program object using `glAttachShader`. The first argument is the handle to the program object, and the second is the handle to the shader object to be attached.

```c
// Attach the shaders to the program object
glAttachShader( theProgram, vertShader );
glAttachShader( theProgram, fragShader );
```

• Then, we link the program by calling `glLinkProgram`, providing the handle to the program object as the only argument. As with compilation, we check for the success or failure of the link with the subsequent query.

```c
// Link the program
glLinkProgram( theProgram );
```

• We check the status of the link by calling `glGetProgramiv`. Similar to `glGetShaderiv`, `glGetProgramiv` allows us to query various attributes of the shader program. In this case, we ask for the status of the link by providing `GL_LINK_STATUS` as the second argument. The status is returned in the location pointed to by the third argument, in this case named `status`.

```c
//Check for successful linking
GLint status;
glGetProgramiv( theProgram, GL_LINK_STATUS, &status );
if( GL_FALSE == status ) {
    fprintf( stderr, "Failed to link shader program!\n" );
    GLint logLen;
    glGetProgramiv( theProgram, GL_INFO_LOG_LENGTH, &logLen );
```
if ( logLen > 0 )
{
    char * log = (char *)malloc(logLen);
    GLsizei written;
    glGetProgramInfoLog(theProgram, logLen, &written, log);
    fprintf(stderr, "Program log: \n%s", log);
    free(log);
}

• And finally, install the program into the OpenGL pipeline:

    glUseProgram( theProgram );

With these two simple shaders compiled, linked, and installed into the OpenGL pipeline, we have a complete OpenGL pipeline and are ready to begin rendering.

If a program is no longer needed, it can be deleted from the OpenGL memory by calling glDeleteProgram, providing the program handle as the only argument. This invalidates the handle and frees the memory used by the program. Note that if the program object is currently in use, it will not be immediately deleted, but will be flagged for deletion when it is no longer in use.

Additional reading: [OpenGl 4.0 Shading Language Cookbook, pages 15-29](#)
13. Playing with colors
- Part of the fragment's data of each fragment in the fragment shader includes the position of the fragment on the screen. Thus, if we want to vary the color of a triangle across its surface, we can access this data in our fragment shader and use it to compute the final color for that fragment.
- This sample is based on SAMPLE1, with only a small change to the fragment shader:

```glsl
#version 330

out vec4 outputColor;

void main()
{
    float lerpValue = gl_FragCoord.y / 512.0f;

    outputColor = mix(vec4(1.0f, 0.0f, 0.0f, 1.0f),
                      vec4(0.0f, 1.0f, 0.0f, 1.0f),
                      lerpValue);
}
```

- `gl_FragCoord` is a built-in variable that is only available in a fragment shader. It is a vec3, so it has an X, Y, and Z component. The X and Y values are in window coordinates, so the absolute value of these numbers will change based on the window's resolution. Window coordinates put the origin at the bottom-left corner. So fragments along the bottom of the triangle would have a lower Y value than those at the top.
- The idea with this shader is that the color of a fragment will be based on the Y value of its window position. The 512.0f is the height of the window (unless you re-size the window). The division in the first line of the function simply converts the Y position to the [0, 1] range, where 1 is at the top of the window and 0 is at the bottom.
- The second line uses this [0, 1] value to perform a linear interpolation between two colors. The `mix`
function is one of the many, many standard functions that the OpenGL Shading Language provides. Many of these functions, like mix, are vectorized. That is, some of their parameters can be vectors, and when they are, they will perform their operations on each component of the vector simultaneously. In this case, the dimensionality of the first two parameters must match.

- The mix function performs a **linear interpolation**. It will return exactly the first parameter if the third parameter is 0, and it will return exactly the second parameter if the third parameter is 1. If the third parameter is between 0 and 1, it will return a value between the two other parameters, based on the third parameter.

14. Multiple vertex attributes. SAMPLE3
Algorithmically modifying the color of fragments in SAMPLE2 doesn't open a lot of possibilities. How about more control? We can supply a color attribute along each vertex. This is the purpose of SAMPLE3. What we want:

- For every position that we pass to a vertex shader, we want to pass a corresponding color value.
- For every output position in the vertex shader, we also want to output a color value that is the same as the input color value the vertex shader received.
- In the fragment shader, we want to receive an input color from the vertex shader and use that as the output color of that fragment.

In order to accomplish the above, we need to change our user supplied vertex data, to the following:

```c
const float vertexData[] = {
  0.0f,    0.5f, 0.0f, 1.0f,
  0.5f, -0.366f, 0.0f, 1.0f,
-0.5f, -0.366f, 0.0f, 1.0f,
  1.0f,    0.0f, 0.0f, 1.0f,
  0.0f,    1.0f, 0.0f, 1.0f,
  0.0f,    0.0f, 1.0f, 1.0f,
};
```

The `vertexData` variable is one large array of floats. The way we want to use it however is as two arrays. Each 4 floats is a single vec4, and the first three vec4s represents the positions. The next 3 are the colors for the corresponding vertices.
In memory, the vertexData array looks like this:

- The top two show the layout of the basic data types, and each box is a byte. The bottom diagram shows the layout of the entire array, and each box is a float. The left half of the box represents the positions and the right half represents the colors.

- The first 3 sets of values are the three positions of the triangle, and the next 3 sets of values are the three colors at these vertices. What we really have is two arrays that just happen to be adjacent to one another in memory. One starts at the memory address &vertexData[0], and the other starts at the memory address &vertexData[12].

- Feeding vertex position and color data to OpenGL is nothing we have not seen before:

```c
glGenBuffers(1, &vertexBufferObject);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```
• Now, on to rendering, bold text denote change from SAMPLE2:

```cpp
glUseProgram(theProgram);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, (void*)48);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glUseProgram(0);
    glutSwapBuffers();
```

• Since we have two pieces of data, we have two vertex attributes. For each attribute, we must call `glEnableVertexAttribArray` to enable that particular attribute. The first parameter is the attribute location set by the `layout(location)` field for that attribute in the vertex shader.

• Then, we call `glVertexAttribPointer` for each of the attribute arrays we want to use. The only difference in the two calls are which attribute location to send the data to and the last parameter. The last parameter is the byte offset into the buffer of where the data for this attribute starts. This offset, in this case, is 4 (the size of a float) * 4 (the number of floats in a vec4) * 3 (the number of vec4's in the position data).

• After this, we use `glDrawArrays` to render, then disable the arrays with `glDisableVertexAttribArray`. 
• The vertex shader for SAMPLE3 looks like this:

```glsl
#version 330

layout(location = 0) in vec4 vertexPosition;
layout(location = 1) in vec4 vertexColor;

smooth out vec4 color;

void main()
{
    color = vertexColor;
    gl_Position = vertexPosition;
}
```

• vertexPosition is assigned the attribute index 0, while vertexColor is assigned 1. That much only gets the data into the vertex shader. We want to pass this data out of the vertex shader. To do this, we must define an **output variable**. This is done using the **out** keyword. In this case, the output variable is called **color** and is of type vec4.

• The **smooth** keyword is an interpolation qualifier. We will see what this means in a bit later.

• In main function, we actually write to **color**, assigning to it the value of **vertexColor** that was given as a vertex attribute.
The data flow from the buffer object to the vertex shaders looks like this now. As before, every 3 vertices process is transformed into a triangle.
• Below is the fragment shader:

```
#version 330

smooth in vec4 color;
out vec4 fragColor;

void main()
{
    fragColor=color;
}
```

• This fragment shader defines an **input variable**. It is no coincidence that this input variable is named and typed the same as the output variable from the vertex shader. We are trying to feed information from the vertex shader to the fragment shader. To do this, OpenGL requires that the output from the previous stage have the same name and type as the input to the next stage. It also must use the same interpolation qualifier; if the vertex shader used smooth, the fragment shader must do the same.

• This is a good part of the reason why OpenGL requires vertex and fragment shaders to be linked together; if the names, types, or interpolation qualifiers do not match, then OpenGL will raise an error when the program is linked.

• So the fragment shader receives the value output from the vertex shader. The shader simply takes this value and copies it to the output. Thus, the color of each fragment will simply be whatever the vertex shader passed along.
• **Fragment interpolation.** See, our vertex shader is run only 3 times. This execution produces 3 output positions (gl_Position) and 3 output colors (color). The 3 positions are used to construct and rasterize a triangle, producing a number of fragments.

• The fragment shader is not run 3 times. It is run once for every fragment produced by the rasterizer for this triangle. The number of fragments produced by a triangle depends on the viewing resolution and how much area of the screen the triangle covers. For this sample, the number of fragments produced in the process of rasterization would count somewhere 25000 times.

• If the vertex shader outputs only 3 times, thus outputting only 3 colors, where does the other 24997 color values come from? The answer is **fragment interpolation.**

• By using the interpolation qualifier **smooth** when defining the vertex output and fragment input, we are telling OpenGL to do something special with this value. Instead of each fragment receiving one value from a single vertex, what each fragment gets is a **blend** between the three output values over the surface of the triangle. The closer the fragment is to one vertex, the more that vertex's output contributes to the value that the fragment program receives.

• Because such interpolation is by far the most common mode for communicating between the vertex shader and the fragment shader, if you do not provide an interpolation keyword, smooth will be used by default. Another interpolation qualifier available is **flat.** It actually turns interpolation off. It essentially says that, rather than interpolating between three values, each fragment of the triangle will simply get the first of the three vertex shader output variables. The fragment shader gets a flat value across the surface of the triangle, hence the term “flat.”

Additional reading: [http://www.arcsynthesis.org/gltut/Basics/Tut02%20Vertex%20Attributes.html](http://www.arcsynthesis.org/gltut/Basics/Tut02%20Vertex%20Attributes.html)
15. Blending SAMPLE3.1
Blending is the stage of OpenGL rendering pipeline that takes the fragment color outputs and combines them with the destination colors that they are adding to. Blending can be used to make objects appear transparent. This is not its only use, but it is one of its main uses.

Remember that in RGB colorspace, a color can be represented by combining the intensities of three base colors – red, green and blue. Each of these intensities can range from 0 to 1. A fragment shader output usually is declared as a vec4, which means it has 4 components rather than three. The fourth component is called alpha and denotes the transparency value for the fragment. A value of 0 means that the fragment is fully transparent, while a value of 1 means that it's opaque. However to be able to take advantage of alpha component, you have to tell OpenGL about that.

When doing blending, the color from the fragment output is called the source color. The color currently in the buffer is called the destination color.

Blending must be enabled with a call to `glEnablei(GL_BLEND, i)`, where i is an integer representing the buffer to enable blending on. Each buffer being written to by a fragment shader can have blending separately enabled and disabled. `glDisablei` can be used to disable blending. (In our course we will be using only one color buffer, so “i” should be set to 0)

The next step is to specify the equation between source and destination color. To do this, we can use the function:

```c
void glBlendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);
```
• The function takes 2 parameters. The first parameter specifies the **math operation** between the incoming (source color) and the color already present in the buffer. The second parameter specifies the **math operation** between the incoming alpha and alpha already in the buffer. The following operation are currently available:

• **GL_FUNC_ADD**: The source and destination colors are added to each other.

• **GL_FUNC_SUBTRACT**: Subtracts the destination from the source.

• **GL_FUNC_REVERSE_SUBTRACT**: Subtracts the source from the destination.

• **GL_MIN**: The output color is the component-wise minimum value of the source and dest colors.

• **GL_MAX**: The output color is the component-wise maximum value of the source and dest colors.

• If we use the **GL_FUNC_ADD** parameter our final color (and alpha value) will be computed according to this formula:

\[
\text{Final color} = \text{source color} + \text{destination color}
\]

\[
\text{Final alpha} = \text{source alpha} + \text{destination alpha}
\]
• However that doesn't give us a lot of flexibility. Therefore OpenGL pipeline provides us with 2 more parameters that come into the above equation as shown below (the example below assumes GL_FUNC_ADD is selected as the math operation between source and destination):

\[
\text{Final color} = \text{source color} \times \text{param1} + \text{destination color} \times \text{param2}
\]

\[
\text{Final alpha} = \text{source alpha} \times \text{param1} + \text{destination alpha} \times \text{param2}
\]

• Param1 and param2 can be set separately for color and alpha components using the following function:

```c
void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB,
                          GLenum srcAlpha, GLenum dstAlpha)
```

• Using parameters from function above, would transform our example in (the example below assumes GL_FUNC_ADD is selected as the math operation between source and destination fragments):

\[
\text{Final color} = \text{source color} \times \text{srcRGB} + \text{destination color} \times \text{dstRGB}
\]

\[
\text{Final alpha} = \text{source alpha} \times \text{srcAlpha} + \text{destination alpha} \times \text{dstAlpha}
\]
The following parameters are available for `glBlendFuncSeparate()`:

- `GL_SRC_COLOR`
- `GL_DST_COLOR`
- `GL_ZERO`
- `GL_ONE`
- `GL_ONE_MINUS_SRC_COLOR`
- `GL_ONE_MINUS_DST_COLOR`
- `GL_SRC_ALPHA`
- `GL_DST_ALPHA`
- `GL_ONE_MINUS_SRC_ALPHA`
- `GL_ONE_MINUS_DST_ALPHA`

If you run SAMPLE3.1 the first blending equation (use SPACE to scroll through other blending equations) that is implemented is the traditional alpha blending that let's you simulate transparent objects. Let's see the code that enables that kind of blending:

```c
glEnablei(GL_BLEND, 0);
glBlendEquationSeparate(GL_FUNC_ADD, GL_FUNC_ADD);
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);
```

The above translates into:

- **Final color** = source color* `GL_SRC_ALPHA` + destination color* `GL_ONE_MINUS_SRC_ALPHA`

- **Final alpha** = source alpha* `GL_ONE` + destination alpha* `GL_ZERO`
16. Moving the triangle SAMPLE4
The simplest way one might think to move a triangle or other object around is to simply modify the vertex position data directly. From previous samples, we learned that the vertex data is stored in a buffer object. So the task is to modify the vertex data in the buffer object.

The modifications are done in two steps. The first step is to generate the X, Y offset that will be applied to each position. The second is to apply that offset to each vertex position. The generation of the offset is done with the `ComputePositionOffset` function:

```c
void ComputePositionOffsets(float &fXOffset, float &fYOffset)
{
    const float fLoopDuration = 5.0f;
    const float fScale = 3.14159f * 2.0f / fLoopDuration;

    float fElapsedTime = glutGet(GLUT_ELAPSED_TIME) / 1000.0f;
    float fCurrTimeThroughLoop = fmodf(fElapsedTime, fLoopDuration);

    fXOffset = cosf(fCurrTimeThroughLoop * fScale) * 0.5f;
    fYOffset = sinf(fCurrTimeThroughLoop * fScale) * 0.5f;
}
```
The above function computes offsets in a loop. The offsets produce circular motion. The `glutGet(GLUT_ELAPSED_TIME)` retrieves the integer time in milliseconds since the application started. The `fmodf` function computes the remainder of the division between first and the second parameter. Thus, it returns a value on the range \([0, fLoopDuration),\) which is what we need to create a periodically repeating pattern.

The `cosf` and `sinf` functions compute the cosine and sine respectively. It is not important to know exactly how these functions work, but they effectively compute a circle of diameter 2. By multiplying by 0.5f, it shrinks the circle down to a circle with a diameter of 1.

Once the offsets are computed, the offsets have to be added to the vertex data. This is done with the `AdjustVertexData` function:

```c
void AdjustVertexData(float fXOffset, float fYOffset)
{
    int vertexDataElements = sizeof(vertexData) / 4;
    std::vector<float> fNewData(vertexDataElements);
    memcpy(&fNewData[0], vertexData, sizeof(vertexData));
    for (int iVertex = 0; iVertex < vertexDataElements; iVertex += 4)
    {
        fNewData[iVertex] += fXOffset;
        fNewData[iVertex + 1] += fYOffset;
    }
    glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertexData), &fNewData[0]);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```
• The above function works by copying the vertex data into a std::vector, then applying the offset to the X and Y coordinates of each vertex. The last three lines are the OpenGL-relevant parts.

• First, the buffer objects containing the positions is bound to the context. Then the new function `glBufferSubData` is called to transfer this data to the buffer object.

• The difference between `glBufferData` and `glBufferSubData` is that the `SubData` function does not allocate memory. `glBufferData` specifically allocates memory of a certain size; `glBufferSubData` only transfers data to the already existing memory. Calling `glBufferData` on a buffer object that has already been allocated tells OpenGL to reallocate this memory, throwing away the previous data and allocating a fresh block of memory. Whereas calling `glBufferSubData` on a buffer object that has not yet had memory allocated by `glBufferData` is an error.

• The `glBufferSubData` function can update only a portion of the buffer object's memory. The second parameter to the function is the byte offset into the buffer object to begin copying to, and the third parameter is the number of bytes to copy. The fourth parameter is our array of bytes to be copied into that location of the buffer object.

• **Buffer Object Usage Hints.** Every time we draw something in this sample, we are changing the buffer object's data. OpenGL has a way to tell it that you will be doing something like this, and it is the purpose of the last parameter of `glBufferData`. This sample changed the allocation of the buffer object slightly, replacing:

```cpp
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
```

with

```cpp
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STREAM_DRAW);
```
• GL_STATIC_DRAW tells OpenGL that you intend to only set the data in this buffer object once. GL_STREAM_DRAW tells OpenGL that you intend to set this data constantly, generally once per frame. These parameters do not mean anything with regard to the API; they are simply hints to the OpenGL implementation. Proper use of these hints can be crucial for getting good buffer object performance when making frequent changes.

• The render() function now has become this (blue denotes the addition):

```c
void render(void) {
    float fXOffset = 0.0f, fYOffset = 0.0f;
    ComputePositionOffsets(fXOffset, fYOffset);
    AdjustVertexData(fXOffset, fYOffset);
    
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    
    glUseProgram(theProgram);
    
    glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
    
    glDrawArrays(GL_TRIANGLES, 0, 3);
    
    glDisableVertexAttribArray(0);
    glUseProgram(0);
}
```

17. Moving the triangle – A better way
Imagine a scene involving millions of vertices (and no, that's not an exaggeration for high-end applications). Moving objects as we did in SAMPLE4 by copying millions of vertices from the original vertex data to the buffer object would be a terrific waste of resources. And this is all before rendering. Clearly there should be a better way of doing this.

Why just not give the vertex shader the offset and let it compute the final vertex position, since it operates on each vertex? That's exactly the idea of this sample. Here's the new vertex shader:

```glsl
#version 330

layout(location = 0) in vec4 vertexPosition;
uniform vec2 offset;

void main()
{
    vec4 totalOffset = vec4(offset.x, offset.y, 0.0, 0.0);
    gl_Position = vertexPosition+totalOffset;
}
```

After defining the input position, the shader defines a 2-dimensional vector `offset`. But it defines it with the term `uniform`, rather than in or out. This has a particular meaning.
- Variables defined as **uniform** do not change at the same frequency as variables defined as **in**. Input variables change with every execution of the shader. **Uniform** variables (called uniforms) change only between executions of rendering calls. And even then, they only change when the user sets them explicitly to a new value. Vertex shader inputs come from vertex attribute array definitions and buffer objects. By contrast, uniforms are set directly on program objects.

- In order to set a uniform in a program, we need two things. **The first is a uniform location**. Much like with attributes, there is an index that refers to a specific uniform value. Unlike attributes, you cannot set this location yourself; you must query it. In this sample, this is done in the init() function, with this line:

  ```
  offsetLocation = glGetUniformLocation(theProgram, "offset");
  ```

- The function **glGetUniformLocation** retrieves the uniform location for the uniform named by the second parameter. Note that just because a uniform is defined in a shader, GLSL does not have to provide a location for it. It will only have a location if the uniform is actually used in the program, as we see in the vertex shader; **glGetUniformLocation** will return -1 if the uniform has no location.

- Once we have the uniform location, we can set the uniform's value. However, unlike retrieving the uniform location, setting a uniform's value requires that the program be currently in use with **glUseProgram**.

- We use ComputePositionOffsets to get the offsets, and then use **glUniform2f** to set the uniform's value. The buffer object's data is never changed; the shader simply does the hard work. Which is why those shader stages exist in the first place.
Here's how the render() function now looks like:

```c
void render(void)
{
    float fXOffset = 0.0f, fYOffset = 0.0f;
    ComputePositionOffsets(fXOffset, fYOffset);

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(theProgram);

    glUniform2f(offsetLocation, fXOffset, fYOffset);

    glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray(0);
    glUseProgram(0);

    glutSwapBuffers();
}
```
The function `glUniform2f` uses a special bit of OpenGL nomenclature. Specifically, the “2f” part. Uniforms can have different types. And because OpenGL is defined in terms of C rather than C++, there is no concept of function overloading. So functions that do the same thing which take different types must have different names. OpenGL has standardized its naming convention for these.

Functions like `glUniform` have a **suffix**. The first part of the suffix is the number of values it takes. In the above case, `glUniform2f` takes 2 values, in addition to the regular parameters. The type of values it takes is defined by the second part. The possible type names and their associated types are:

![Type names and types](image)

There are also vector forms, defined by adding a “v” after the type. These take an array of values, with a parameter saying how many elements are in the array. Example:

```cpp
//Vertex Shader
uniform vec2 Exponents[5];
//In your C++ code
float Exponents[10];
glUniform2fv(MyShader, 5, Exponents);
```

18. More power to the shaders
• It is actually possible to move almost entirely the `ComputePositionOffsets()` to the vertex shader with the except of `glutGet(GL_ELAPSED_TIME)` since shaders can't call directly C/C++ functions.

• The vertex shader now looks like this:

```
#version 330

layout(location = 0) in vec4 vertexPosition;
uniform float loopDuration;
uniform float time;

void main()
{
    float timeScale = 3.14159f * 2.0f / loopDuration;
    
    float currTime = mod(time, loopDuration);
    vec4 totalOffset = vec4(
        cos(currTime * timeScale) * 0.5f,
        sin(currTime * timeScale) * 0.5f,
        0.0f,
        0.0f);

    gl_Position = vertexPosition + totalOffset;
}
```

• This shader takes two uniforms: the duration of the loop and the elapsed time. In this shader, we use a number of standard GLSL functions, like `mod`, `cos`, and `sin`. And these are just the tip of the iceberg; there are a lot of standard GLSL functions available.
• The two uniform variables are set within our C++ sample in a way that we already seen:

```cpp
glUniform1f(loopDurationLocation, 5);
float fElapsedTime = glutGet(GLUT_ELAPSED_TIME) / 1000.0f;
glUniform1f(timeLocation, fElapsedTime);
```

• Now let's make the life of the fragment shader more complicated. Moving the triangle around is nice and all, but it would also be good if we could do something time-based in the fragment shader. Fragment shaders cannot affect the position of the object, but they can control its color.

```glsl
#version 330

out vec4 fragColor;

uniform float fragLoopDuration;
uniform float time;

const vec4 firstColor = vec4(1.0f, 1.0f, 1.0f, 1.0f);
const vec4 secondColor = vec4(0.0f, 1.0f, 0.0f, 1.0f);

void main()
{
    float currTime = mod(time, fragLoopDuration);
    float currLerp = currTime / fragLoopDuration;
    fragColor = mix(firstColor, secondColor, currLerp);
}
```
The above shader interpolates between two colors based on how far it is through the loop. When it is at the start of the loop, the triangle will be firstColor, and when it is at the end of the loop, it will be secondColor.

The standard library function `mix` performs linear interpolation between two values. Like many GLSL standard functions, it can take vector parameters; it will perform component-wise operations on them. So each of the four components of the two parameters will be linearly interpolated by the 3rd parameter. The third parameter, currLerp in this case, is a value between 0 and 1. When it is 0, the return value from `mix` will be the first parameter; when it is 1, the return value will be the second parameter.

As usual fragLoopDuration uniform is set inside the C++ program through a call to `glUniform1f`.

You may be wondering how the `time` uniform for the vertex shader and fragment shader get set? One of the advantages of the GLSL compilation model, which links vertex and fragment shaders together into a single object, is that uniforms of the same name and type are concatenated. So there is only one uniform location for time, and it refers to the uniform in both shaders.

The downside of this is that, if you create one uniform in one shader that has the same name as a uniform in a different shader, but a different type, OpenGL will give you a linker error and fail to generate a program. Also, it is possible to accidentally link two uniforms into one. In the sample, the fragment shader's loop duration had to be given a different name, or else the two shaders would have shared the same loop duration.

Additional reading: [http://www.arcsynthesis.org/gltut/Positioning/Tut03%20Multiple%20Shaders.html](http://www.arcsynthesis.org/gltut/Positioning/Tut03%20Multiple%20Shaders.html)
19. The orthographic cube

- Let's move from flat to real 3D. This sample renders a rectangular prism (3D rectangle). The dimensions of the prism are 0.5x0.5x1.5, so it is longer in the Z axis by 3x the X and Y. The code in this sample should be familiar, for the most part. We simply draw 12 triangles rather than one. The rectangular faces of the prism are made of 2 triangles, splitting the face along one of the diagonals.

- The vertices also have a color. However, the color for the 6 vertices that make up a face is always the same; this gives each face a single, uniform color.

- Shaders are made of standard stuff that we covered in previous samples.

- **Face Culling.** There is one very noteworthy code change in the init() function, at the end:

  ```c
  glEnable(GL_CULL_FACE);
  glFrontFace(GL_CW);
  glCullFace(GL_BACK);
  ```

- The `glEnable` function is a multi-purpose tool. There are a lot of binary on/off flags that are part of OpenGL's state. `glEnable` is used to set these flags to the “on” position. Similarly, there is a `glDisable` function that sets the flag to “off.”
• The **GL_CULL_FACE** flag, when enabled, tells OpenGL to activate **face culling**. Up until now, we have been rendering with face culling off. Face culling is a useful feature for saving performance. Take our rectangular prism, for example. Pick up a remote control; their general shape is that of a rectangular prism. No matter how you look at it or orient it, you can never see more than 3 sides of it at once. So why bother spending all that fragment processing time drawing the other three sides?

• Face culling is a way of telling OpenGL not to draw the sides of an object that you cannot see. It is quite simple, really.

• In window space, after the transform from normalized device coordinates, you have a triangle. Each vertex of that triangle was presented to OpenGL in a specific order. This gives you a way of numbering the vertices of the triangle.

• No matter what size or shape the triangle is, you can classify the ordering of a triangle in two ways: **clockwise** or **counter-clockwise**. That is, if the order of the vertices from 1 to 2 to 3 moves clockwise in a circle, relative to the triangle's center, then the triangle is facing clockwise relative to the viewer. Otherwise, the triangle is counter-clockwise relative to the viewer. This ordering is called the **winding order**.

• The left triangle has a clockwise winding order; the triangle on the right has a counter-clockwise winding order.
• Face culling in OpenGL works based on this ordering. Setting this is a two-step process, and is accomplished by the last two statements of the initialization function.

• The `glFrontFace` defines which winding order, clockwise or counter-clockwise, is considered to be the “front” side of the triangle. This function can be given either `GL_CW` or `GL_CCW`, for clockwise and counter-clockwise respectively.

• The `glCullFace` function defines which side, front or back, gets culled. This can be given `GL_BACK`, `GL_FRONT`, or `GL_FRONT_AND_BACK`. The latter culls everything, so no triangles are rendered. This can be useful for measuring vertex shader performance but is less useful for actually drawing anything.

• This sample defines the vertices in clockwise order.
Lack of Perspective

• Here's the output of SAMPLE7:

![Sample output image]

• There's something wrong with this. Namely, that it looks like a square. Pick up a remote control again. Point it directly at your eye and position it so that it is in the center of your vision. You should only be able to see the front panel of the remote. Now, move it to the right and up, similar to where the square is. You should be able to see the bottom and left side of the remote. So we should be able to see the bottom and left side of our rectangular prism. But we cannot. Why not?

• Think back to how rendering happens. In clip-space, the vertices of the back end of the rectangular prism are directly behind the front end. And when we transform these into window coordinates (we'll talk about that shortly), the back vertices are still directly behind the front vertices. This is what the rasterizer sees, so this is what the rasterizer renders.

• There has to be something that reality is doing that we are not. This magic is called “perspective.”

20. Perspective projection
The above figure represents the generalized vertex transformation pipeline in OpenGL. In the previous sample we saw a square on the screen instead of a 3D rectangular prism. This happened because our Vertex Data(1) was taken to the shader and was copied straight to "gl_Position" which is the clip coordinate between step 3 and 4. Then to obtain the Normalized Device Coordinates, OpenGL divided our clip "xyz" coordinate by "w", which happened to be "1" resulting in no changes again. Before step 5, we thus, had in the pipeline the same coordinates as at step 1.

The question is, so how do we transform the Vertex Data(1), so that our cube would look 3D? The answer is to involve step 3 – a projection transformation – which we didn't consider until now.

A projection, in terms of the rendering pipeline is a way to transform a world from one dimensionality to another. Our initial world is three dimensional, and therefore, the rendering pipeline defines a projection from this 3D world into the 2D one that we see. It is the 2D world in which the triangles are actually rendered.
• An **orthographic projection** is a very simplistic projection. When projecting onto an axis-aligned surface, as below, the projection simply involves throwing away the coordinate perpendicular to the surface.

![Diagram of orthographic projection](image)

• A scene orthographically projected onto the black line. The gray box represents the part of the world that is visible to the projection; parts of the scene outside of this region are not seen.
• Human eyes do not see the world via orthographic projection. If they did, you would only be able to see an area of the world the size of your pupils. Because we do not use orthographic projections to see (among other reasons), orthographic projections do not look particularly real to us.

• Instead, we use a pinhole camera model for our eyesight. This model performs a **perspective projection**. A perspective projection is a projection of the world on a surface as though seen through a single point. A 2D to 1D perspective projection looks like this:
As you can see, the projection is radial, based on the location of a particular point. That point is the eye of the projection. Just from the shape of the projection, we can see that the perspective projection causes a larger field of geometry to be projected onto the surface. An orthographic projection only captures the rectangular prism directly in front of the surface of projection. A perspective projection captures a larger space of the world.

In 2D, the shape of the perspective projection is a regular trapezoid (a quadrilateral that has only one pair of parallel sides, and the other pair of sides have the same slope). In 3D, the shape is called a frustum; essentially, a pyramid with the tip chopped off.
• The frustum can be described by the following parameters: \( n \) – distance from the center of the projection to the projection plane (also known as the near clipping plane), \( f \) – distance from the center of projection to the far clipping plane, \( l, r, t, b \) – the left, right, top and bottom coordinates of the near clipping plane in a right handed coordinate system.

• The math describing how the projection is integrated into the OpenGL pipeline is not straightforward and is beyond the scope of this introductory course. Below are some great resources that explain this in detail:

Additional reading: [http://www.songho.ca/opengl/gl_projectionmatrix.html](http://www.songho.ca/opengl/gl_projectionmatrix.html)

• Here's a quick summary of how projection is integrated into the OpenGL Pipeline:

  • First, the vertex data is transformed to **clip space** (also known as the canonical view volume), given frustum parameters. Up until now, our vertex positions have been “pure” 3D vectors—that is, they only had three coordinates, or if they have a fourth coordinate, then \( w \) was always equal to 1. Transformation to clip space serves two reasons:

    • Prepare for projection. We put the proper value into \( w \) so that the homogeneous division produces the desired projection. For the typical perspective projection, this means we copy \( z \) into \( w \).

    • Apply zoom and prepare for clipping. We scale \( x, y, z \) so that they can be compared against \( w \) for clipping. This scaling takes the frustum parameters into consideration.
The vertex shader from this sample that transforms our vertex data to clip space is as follows:

```glsl
#version 330

layout(location = 0) in vec4 vertexPosition;
layout(location = 1) in vec4 colorIn;
uniform vec2 offset;
uniform float r,l,t,b,zFar,zNear;
smooth out vec4 colorForFS;

void main()
{
    vec4 cameraPos = vertexPosition + vec4(offset.x, offset.y, 0.0, 0.0);
    vec4 clipPos;

    clipPos.x = cameraPos.x * (2*zNear)/(r-l) + cameraPos.z*(r+l)/(r-l);
    clipPos.y = cameraPos.y * (2*zNear)/(t-b) + cameraPos.z*(t+b)/(t-b);
    clipPos.z = -cameraPos.z * (zFar + zNear) / (zFar - zNear);
    clipPos.z -= 2 * zFar * zNear / (zFar - zNear);
    clipPos.w = -cameraPos.z;

    gl_Position = clipPos;
    colorForFS = colorIn;
}
```

cameraPos(this "camera" notation will be explained in the next chapter) is our vertex data with an additional offset to it. Next, cameraPos is transformed via some non-trivial :) math to clip space and stored in clipPos, which is afterwards copied to gl_Position and transmitted to the next stage of the pipeline.
In our SAMPLE application we set the uniforms describing the **frustum** as follows:

```plaintext
glUniform1f(rLocation, 0.75f);  glUniform1f(lLocation, -0.75f);
glUniform1f(tLocation, 0.75f);  glUniform1f(bLocation, -0.75f);
glUniform1f(zNearLocation, 1.0f); glUniform1f(zFarLocation, 3.0f);
```

The values above were picked in such a way that our 3D prism would fall inside our frustum. If you modify the zNear parameter you can see how our 3D prism becomes smaller if zNear approaches 0 and becomes bigger when zNear grows. It is because modification of zNear leads to increase or decrease of the volume of the frustum, meaning that more or less vertex data can fill inside it.

The next stage after the vertex shader is frustum culling (clipping). The clip coordinates, x, y and z are tested by comparing with w. If each clip coordinate is less than -w, or greater than w, then the vertex will be discarded. Then, OpenGL will reconstruct the edges of the polygon where clipping occurs.
The next step after frustum culling is the perspective divide – the x,y and z are divided by w (in the vertex shader we placed some meaningful information into w) to produce the normalized device coordinates. It's called **perspective divide** because the actual perspective projection happens here. The idea is illustrated in the diagram below:

The projection of the point P onto the projection plane (near clipping plane - zNear). This plane is at an offset of Ez compared to the eye point, which is fixed at the origin. R is the projected point.

A perspective projection essentially shifts vertices towards the eye, based on the location of that particular vertex. Vertices farther in Z from the front of the projection are shifted less than those closer to the eye. And the shift also depends on how far the vertices are from the center of the plane of projection, in the X,Y direction. The problem is really just a simple geometry problem.
• What we have are two similar right triangles: the triangle formed by E, R and Ez, and the triangle formed by E, P, and Pz. We have the eye position and the position of the unprojected point. To find the location of R, we simply do this:

\[
\overline{R} = \overline{P} \begin{pmatrix} \frac{Ez}{Pz} \end{pmatrix}
\]

• Since this is a vectorized function, this solution applies equally to 2D as to 3D. Thus, perspective projection is simply the task of applying that simple formula to every vertex that the vertex shader receives.

• One question you should always ask when dealing with equations is this: can it divide by zero? And this equation certainly can; if the camera-space position of any vertex is ever exactly 0, then we have a problem.

• This is why there is a strict requirement for the near clipping plane (zNear) to be greater than zero. As we already said, before the perspective divide we have the clipping operation that cuts off geometry that is outside the clipping planes. If our near clipping plane is greater than zero, the geometry that potentially has a coordinate of “z” that is zero will be removed from further processing.

• There is one final step before our initial 3D geometry coordinates are transformed to window (screen) coordinates. We will cover this process shortly.
21. Camera space

- Our vertex positions before SAMPLE8 have been stored directly in clip space because we did not involve a projection transformation. If you remember they had to be bound to the Normalized Device Coordinates boundaries [-1;1] in XYZ directions to be visible on the screen.

- Now that in SAMPLE8 we used coordinates that exceed [-1;1] range how are they called? The answer is camera space coordinates.

- Camera space is a 3D coordinate space in which the origin is at the center of projection, one is axis parallel to the direction the camera is facing (perpendicular to the projection plane), one axis is the intersection of the top and bottom clip planes, and the other axis is the intersection of the left and right clip planes.
• Camera space is also known as "Eye space" and camera coordinates are sometime referred as "Eye coordinates" like in this diagram below:

![OpenGL vertex transformation diagram]

• Camera space is one of several Coordinate spaces that we will talk throughout this course. Others are:
  • Modeling space, sometimes referred as Object space or Local space
  • World space
  • Clip space, sometimes referred as canonical view volume space
  • Screen space

*Additional reading: 3D Math Primer for Graphics and Game Development 1st Edition, pages 354-358*
22. Welcome to the Matrix :)  

- Matrices are the heart of graphics programming.

- Let's have a look at the following matrix:

\[
\begin{pmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\
0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\
0 & 0 & -1 & 0
\end{pmatrix}
\]

- Now have a look again at our vertex shader part, where we computed the clip space position of the vertex:

```cpp
clipPos.x = cameraPos.x * (2*zNear)/(r-l) + cameraPos.z * (r+l)/(r-l);
clipPos.y = cameraPos.y * (2*zNear)/(t-b) + cameraPos.z * (t+b)/(t-b);
clipPos.z = -cameraPos.z * (zFar + zNear) / (zFar - zNear);
clipPos.z -= 2 * zFar * zNear / (zFar - zNear);
clipPos.w = -cameraPos.z;
```
• Now please have a look at the following:

\[
\begin{bmatrix}
X \\
Y \\
Z \\
W
\end{bmatrix}_{\text{clip}} =
\begin{pmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\
0 & 0 & -\frac{(f+n)}{f-n} & -\frac{2fn}{f-n} \\
0 & 0 & -1 & 0
\end{pmatrix}
\begin{bmatrix}
X \\
Y \\
Z \\
W
\end{bmatrix}_{\text{camera}}
\]

• The above is a reformulation of our vertex shader clip space position computation in terms of linear algebra.

• The two long vertical columns of XYZW labeled “clip” and “camera” are 4-dimensional vectors; namely the clip and camera space position vectors.

• The operation depicted above is a vector-matrix multiplication. A matrix of dimension NxM can only be multiplied by a vector of dimension M. The result of such a multiplication is a vector of dimension N. Since our matrix in this case is 4x4, it can only be multiplied with a 4D vector and this multiplication will produce a 4D vector.

• SAMPLE9 expresses the transformation from camera to clip space via vector-matrix multiplication. The advantages of using such a form will be clear later, for now let's look at our vertex shader:
• The OpenGL Shading Language (GLSL), being designed for graphics operations, naturally has matrices as basic types. The mat\textbf{4} is a 4x4 matrix (columns x rows). GLSL has types for all combinations of columns and rows between 2 and 4. Square matrices (matrices where the number of columns and rows are equal) only use one number, as in mat4 above. So mat3 is a 3x3 matrix. If the matrix is not square, GLSL uses notation like mat2x4: a matrix with 2 columns and 4 rows.

• Vector-matrix multiplication is such a common operation in graphics that operator \(*\) is used to perform it. So the second line of main multiplies the perspective matrix by the camera position.

```cpp
#version 330

layout(location = 0) in vec4 vertexPosition;
layout(location = 1) in vec4 colorIn;
uniform vec2 offset;
uniform mat4 perspectiveMatrix;
smooth out vec4 colorForFS;

void main()
{
    vec4 cameraPos = vertexPosition + vec4(offset.x, offset.y, 0.0, 0.0);
    gl_Position = perspectiveMatrix * cameraPos;
    colorForFS = colorIn;
}
```
• Please note the order of this operation. The matrix is on the left and the vector is on the right. Matrix multiplication is not commutative, so \( M \cdot v \) is not the same thing as \( v \cdot M \). Normally vectors are considered \( N \times 1 \) (\( N \) rows, 1 column) matrices (where \( N \) is the size of the vector). When you multiply vectors on the left of the matrix, GLSL considers it an \( 1 \times N \) (1 row, \( N \) columns) matrix; this is the only way to make the multiplication make sense. This will multiply the single row of the vector with each column of the matrix, summing the results, creating a new vector. This is not what we want to do. We want to multiply rows of the matrix by the vector, not columns of the matrix. Put the vector on the right, not the left.

• We construct the matrix inside our init() function

```cpp
GLuint perspectiveMatrixUnif = glGetUniformLocation(theProgram, "perspectiveMatrix");

float theMatrix[16];
memset(theMatrix, 0, sizeof(float) * 16);

float r=0.75f, l=-0.75f, t=0.75f, b=-0.75f, zFar=3.0f, zNear=1.0f;

theMatrix[0] = (2*zNear)/(r-l);
theMatrix[8] = (r+l)/(r-l);
theMatrix[5] = (2*zNear)/(t-b);
theMatrix[9] = (t+b)/(t-b);
theMatrix[10] = -(zFar + zNear) / (zFar - zNear);
theMatrix[14] = -2 * zFar * zNear / (zFar - zNear);
theMatrix[11] = -1.0f;

glUniformMatrix4fv(perspectiveMatrixUnif, 1, GL_FALSE, theMatrix);
```
• A 4x4 matrix contains 16 values. So we start by creating an array of 16 floating-point numbers called theMatrix. Since most of the values are zero, we can just set the whole thing to zero.

• A 4x4 matrix is technically 16 values, so a 16-entry array can store a matrix. But there are two ways to store a matrix as an array. One way is called **column-major** order, the other naturally is **row-major order**. Column-major order means that, for an MxN matrix (rows x columns), the first M values in the array are the first column (top-to-bottom), the next M values are the second column, and so forth. In row-major order, the first N values in the array are the first row (left-to-right), followed by another N values for the second row, and so forth.

• In this example, the matrix is stored in **column-major** order because this is the default way OpenGL thinks the matrix is laid out. The entire matrix is a single uniform. To transfer the matrix to OpenGL, we use the `glUniformMatrix4fv` function. The first parameter is the uniform location that we are uploading to. This function can be used to transfer an entire array of matrices (yes, uniform arrays of any type are possible), so the second parameter is the number of array entries. Since we're only providing one matrix, this value is 1.

• The third parameter tells OpenGL what the ordering of the matrix data is. If it is GL_TRUE, then the matrix data is in row-major order and a transpose of the matrix will be computed to transform it into column major order.

*Additional reading: [http://www.arcsynthesis.org/gltut/Positioning/Tut04%20The%20Matrix%20Has%20You.html](http://www.arcsynthesis.org/gltut/Positioning/Tut04%20The%20Matrix%20Has%20You.html)*
23. Orthographic projection

- Remember that an orthographic projection in comparison to a perspective projection does not change the apparent size of the object no matter where this object is located. Yet, we still have to specify the clipping planes, so that OpenGL pipeline clip geometry that is outside are viewing areas:

- The orthographics projection matrix looks like this:

\[
\begin{pmatrix}
\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\
0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\
0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

- Examples where orthographic projection is used:
  - Computer aided design (CAD)
  - User interface in 3D games
24. Window coordinates (Screen coordinates)

- Window Coordinates (Screen Coordinates) are yielded by applying normalized device coordinates (NDC) to viewport transformation. The NDC are scaled and translated in order to fit into the rendering screen.

- Here's the matrix used to transform NDC coordinates to screen coordinates:

\[
\begin{pmatrix}
    x_w \\
    y_w \\
    z_w
\end{pmatrix} = \begin{pmatrix}
    \frac{w}{2} x_{ndc} + (x + \frac{w}{2}) \\
    \frac{h}{2} y_{ndc} + (y + \frac{h}{2}) \\
    \frac{f-n}{2} z_{ndc} + \frac{f+n}{2}
\end{pmatrix}
\]

- Transformation from NDC to screen coordinates is an automatic step in the pipeline. The user has only access to setting the matrix variables via the following functions:

  - `glViewport(x, y, w, h)` - `glViewport()` command is used to define the rectangle of the rendering area where the final image is mapped

  - `glDepthRange(n, f)` - `glDepthRange()` is used to determine the z value of the window coordinates. We'll talk more about this function soon.
25. Using the GLM library for mathematics

- Because of the prevalence of matrix math, we'll start using a math library called GLM. OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specification.

- GLM takes a unique approach for a vector/matrix math library. It attempts to emulate GLSL's approach to vector operations where possible. It uses C++ operator overloading to effectively emulate GLSL. In many cases, GLM-based expressions would compile in GLSL.

- GLM is a header only library; there is nothing to build to use it which increases its cross platform capabilities. To use GLM, a programmer only has to include `<glm/glm.hpp>`. This provides all the GLSL features implemented by GLM.

- Example:

```cpp
#include <glm/glm.hpp>

int foo()
{
    glm::vec4 Position = glm::vec4(glm::vec3(0.0), 1.0);
    glm::mat4 Model = glm::mat4(1.0);
    Model[3] = glm::vec4(1.0, 1.0, 0.0, 1.0);
    glm::vec4 Transformed = Model * Position;
    return 0;
}
```
26. Aspect of the world SAMPLE10
If you run SAMPLE9 and re-size the window, the viewport re-sizes with it. Unfortunately, this also means that what was once a rectangular prism with a square front becomes elongated.

This is a problem of aspect ratio, the ratio of an image's width to its height. Currently, when you change the window's dimensions, the code calls glViewport to tell OpenGL the new size. This changes OpenGL's viewport transform, which goes from normalized device coordinates to window coordinates. NDC space has a 1:1 aspect ratio; the width and height of NDC space is 2x2. As long as window coordinates also has a 1:1 width to height ratio, objects that appear square in NDC space will still be square in window space. Once window space became non-1:1, it caused the transformation to also become not a square.

Go back to the definition of the problem. NDC space is a [-1, 1] cube. If an object in NDC space is a square, in order for it to be a square in window coordinates, the viewport must also be a square. Conversely, if you want non-square window coordinates, the object in NDC space must not be a square.
The solution is to modify the projection matrix so that it takes into account the aspect ratio of our viewport. Let's review our frustum from another angle:

Originally, we defined our projection frustum with 6 parameters: left, right, top and bottom coordinates of the projection plane, the distance from the origin where the projection plane is located (zNear) and the extent of our frustum (zFar). Fortunately there is a more intuitive way to describe it using only 4 parameters:
- fovy – specifies the field of view angle in the y direction
- aspect ratio – specifies the ratio between width and height of the projection plane
- zNear – specifies the distance from the origin where our plane is located
- zFar – specifies the extent of our frustum
• The fovy, aspect ratio and zNear are used to compute the left, right, top and bottom coordinates of our projection plane, which are afterward used to builds the perspective projection matrix:

\[
\text{range} = \tan(\text{radians}(\text{fovy} / \text{valType}(2))) \times \text{zNear};
\]

\[
\text{left} = -\text{range} \times \text{aspect};
\]

\[
\text{right} = \text{range} \times \text{aspect};
\]

\[
\text{bottom} = -\text{range};
\]

\[
\text{top} = \text{range};
\]

• The \textbf{aspect} variable solves our aspect ratio problem. It introduces in the projection matrix the data necessary to keep the projected image in correlation to our viewport aspect ratio. That implies that we need to rebuild our matrix every time we change the aspect ratio. This is how we update our projection matrix in SAMPLE10(notice how we use the GLM library and GLM functions to generate a perspective projection matrix):

```cpp
void reshape(int w, int h)
{
    width = w; height = h;

    glUseProgram( theProgram );
    GLuint perspectiveMatrixUnif = glGetUniformLocation(theProgram, "perspectiveMatrix");
    glm::mat4 theMatrix;
    theMatrix = glm::perspective(90.0f,(float)width/height,1.0f,3.0f);
    glUniformMatrix4fv(perspectiveMatrixUnif, 1, GL_FALSE, &theMatrix[0][0]);
    glUseProgram(0);

    glViewport(0,0,(GLsizei)w,(GLsizei)h);
}
```

Additional reading: [http://www.arcsynthesis.org/gltut/Positioning/Tut04%20Aspect%20of%20the%20World.html](http://www.arcsynthesis.org/gltut/Positioning/Tut04%20Aspect%20of%20the%20World.html)
27. Rendering multiple objects
• In this sample we'll look how to deal with rendering multiple objects, as well as what happens when multiple objects overlap.

• Up until now, every time we have attempted to draw anything, we needed to do certain setup work before the draw call. In particular, we have to do the following, for each vertex attribute used by the vertex shader:
  
  • Use glEnableVertexAttribArray to enable this attribute.
  
  • Use glBindBuffer(GL_ARRAY_BUFFER) to bind to the context the buffer object that contains the data for this attribute.
  
  • Use glVertexAttribPointer to define the format of the data for the attribute within the buffer object previously bound to GL_ARRAY_BUFFER.

• The more attributes you have, the more work you need to do for each object. To alleviate this burden, OpenGL provides an object that stores all of the state needed for rendering: the **Vertex Array Object (VAO)**.

• VAOs are created with the **glGenVertexArray** function. This works like glGenBuffers (and like most other OpenGL objects); you can create multiple objects with one call. As before, the objects are GLuints.

• VAOs are bound to the context with **glBindVertexArray**; this function does not take a target the way that glBindBuffer does. It only takes the VAO to bind to the context. Once the VAO is bound, calls to certain functions change the data in the bound VAO.
This is how we create and bind one of two VAO in this sample:

```c
glGenVertexArrays(1, &vaoObject1);
glBindVertexArray(vaoObject1);
```

The following functions change VAO state. Therefore, if no VAO is bound to the context (if you call `glBindVertexArray(0)` or you do not bind a VAO at all), all of these functions, will fail.

- `glVertexAttribPointer`
- Also `glVertexAttribIPointer`,
- `glEnableVertexAttribArray/glDisableVertexAttribArray`

Before analyzing SAMPLE11 code let's understand one more concept: **Indexed Drawing**

In the last sample, we drew a rectangular prism. If you looked carefully at the vertex data, you may have noticed that a lot of vertex data was frequently repeated. To draw one face of the cube, we were required to have 6 vertices; the two shared vertices (along the shared line between the two triangles) had to be in the buffer object twice.

For a simple case like ours, this is only a minor increase in the size of the vertex data. The compact form of the vertex data could be 4 vertices per face, or 24 vertices total, while the expanded version we used took 36 total vertices. However, when looking at real meshes, like human-like characters and so forth that have thousands if not millions of vertices, sharing vertices becomes a major benefit in both performance and memory size. Removing duplicate data can shrink the size of the vertex data by 2x or greater in many cases.
In order to remove this extraneous data, we must perform **indexed drawing**, rather than the **array drawing** we have been doing up until now. The `glDrawArrays` function which we used in previous samples conceptually can be expressed with the following pseudo-code:

```c
void glDrawArrays(GLenum type, GLint start, GLint count)
{
    for(GLint element = start; element < start + count; element++)
    {
        VertexShader(positionAttribArray[element], colorAttribArray[element]);
    }
}
```

- You start with a particular index into the buffers, defined by the `start` parameter, and proceed forward by `count` vertices.

- In order to share attribute data between multiple triangles, we need some way to random-access the attribute arrays, rather than sequentially accessing them. This is done with an **element array**, also known as an **index array**.
Let's assume you have the following attribute array data:

Position Array:  Pos0, Pos1, Pos2, Pos3  
Color Array:    Clr0, Clr1, Clr2, Clr3

With `glDrawArrays` you can render either the first 3 vertices as a triangle, or the last 3 vertices as a triangle (using a start of 1 and count of 3). However, with the right `element array`, you can render 4 triangles from just these 4 vertices:

```
Element Array: 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3
```

This will cause OpenGL to generate the following sequence of vertices:

```
(Pos0, Clr0), (Pos1, Clr1), (Pos2, Clr2),
(Pos0, Clr0), (Pos2, Clr2), (Pos3, Clr3),
(Pos0, Clr0), (Pos3, Clr3), (Pos1, Clr1),
(Pos1, Clr1), (Pos2, Clr2), (Pos3, Clr3),
```

Which is a total of 12 vertices, which generate 4 triangles.

**NOTE:** There is only one element array, and the indices fetched from the array are used for all attributes of the vertex arrays. So you cannot have an element array for positions and a separate one for colors; they all have to use the same element array.
Now how to set up indexed drawing in OpenGL? Indexed drawing requires two things: a properly-constructed **element array** and using a **new drawing command** to do the indexed drawing.

Element arrays, as you might guess, are stored in buffer objects. They have a special buffer object binding point, **GL_ELEMENT_ARRAY_BUFFER**.

**NOTE**: All buffer objects in OpenGL are the same, regardless of what target they are bound to; buffer objects can be bound to multiple targets. So it is perfectly legal to use the same buffer object to store vertex attributes and element arrays (and, FYI, any data for any other use of buffer objects that exists in OpenGL). Obviously, the different data would be in separate regions of the buffer.

So, in order to do indexed drawing, we must bind the buffer to **GL_ELEMENT_ARRAY_BUFFER** and then call **glDrawElements**.

```c
void glDrawElements(GLenum mode,
    GLsizei count,
    GLenum type,
    GLsizeiptr indices);
```

The first parameter is the same as the first parameter of glDrawArrays. The **count** parameter defines how many indices will be pulled from the element array. The **type** field defines what the basic type of the indices in the element array are. For example, if the indices are stored as 16-bit unsigned shorts (GLushort), then this field should be **GL_UNSIGNED_SHORT**. This allows the user the freedom to use whatever size of index they want. **GL_UNSIGNED_BYTE** and **GL_UNSIGNED_INT** (32-bit) are also allowed; indices must be unsigned. The last parameter is the **byte-offset** into the element array at which the index data begins.
• The pseudo-code for `glDrawElements` can be thought of as:

```c
GLvoid *elementArray;

void glDrawElements(GLenum type, GLint count, GLenum type, GLsizei indices)
{
    GLtype *ourElementArray = (type*)((GLbyte *)elementArray + indices);
    for(GLint elementIndex = 0; elementIndex < count; elementIndex++)
    {
        GLint element = ourElementArray[elementIndex];
        VertexShader(positionAttribArray[element], colorAttribArray[element]);
    }
}
```

• The `elementArray` represents the buffer object bound to `GL_ELEMENT_ARRAY_BUFFER`. 
• The index data is stored in our C++ program in the following way (each number from below represents an index in the buffer object):

```c
const GLshort indexData[] = {
  0, 2, 1,
  3, 2, 0,
  4, 5, 6,
  6, 7, 4,
  8, 9, 10,
  11, 13, 12,
  14, 16, 15,
  17, 16, 14,
};
```

• And here's how we initialize the element array with the above index data:

```c
// initializing the vertex buffer data
glGenBuffers(1, &vertexBufferObject);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// initializing the index buffer data
glGenBuffers(1, &indexBufferObject);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferObject);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indexData), indexData, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```
Here's how we initialize the two objects rendered on the screen using Vertex Array Objects:

```c
// initializing the first vertex array object
glGenVertexArrays(1, &vaoObject1);
glBindVertexArray(vaoObject1);

size_t colorDataOffset = sizeof(float) * 3 * numberOfVertices;

glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, (void*)colorDataOffset);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferObject);

glBindVertexArray(0);

// initializing the second vertex array object
glGenVertexArrays(1, &vaoObject2);
glBindVertexArray(vaoObject2);

size_t posDataOffset = sizeof(float) * 3 * (numberOfVertices/2);
colorDataOffset += sizeof(float) * 4 * (numberOfVertices/2);

// Use the same buffer object previously bound to GL_ARRAY_BUFFER.
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)posDataOffset);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, (void*)colorDataOffset);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferObject);

glBindVertexArray(0);
```
The above code looks complicated, but it is really just the rendering code we have seen before. The offset computations for the `glVertexAttribPointer` calls are more complex, due to having the data for 2 objects stored in a single buffer. But overall it is the same code.

**NOTE**: If you look at the vertex position attribute in our array `const float vertexData[]`, we have a 3-component position vector. But the shader still uses a vec4. This works because OpenGL will fill in any missing vertex attribute components that the shader looks for but the attribute array doesn't provide. It fills them in with zeros, except for the fourth component, which is filled in with a 1.0.

Here's the how our `render` function now looks (blue color denotes usage of VAO and indexed drawing):

```c
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClearDepth(1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(theProgram);

glBindVertexArray(vaoObject1);
glUniform3f(offsetUniform, 0.0f, 0.0f, 0.0f);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);

glBindVertexArray(vaoObject2);
glUniform3f(offsetUniform, 0.0f, 0.0f, -1.0f);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);

glBindVertexArray(0);
glUseProgram(0);

glutSwapBuffers();
```
• Now, if you execute this sample, you will see this:

• Wait a minute... there is something wrong here. One object appears smaller than the other because it is farther away, in terms of its Z distance to the camera.

    glUniform3f(offsetUniform, 0.0f, 0.0f, -1.0f);

We are using a perspective transform, so it makes sense that more distant objects appear smaller. However, if the smaller object is behind the larger one, why is it rendered on top of the one in front?
The OpenGL specification defines a rasterization-based renderer. Rasterizers offer great opportunities for optimizations and hardware implementation, and using them provides great power to the programmer. However, they're very stupid. A rasterizer is basically just a triangle drawer. Vertex shaders tell it what vertex positions are, and fragment shaders tell it what colors to put within that triangle. But no matter how fancy, a rasterization-based renderer is just drawing triangles. They draw each triangle in the order given. This means that, if there is overlap between multiple triangles in window space, the triangle that is rendered last will be the one that is seen.

This problem is called **hidden surface elimination**.

The first thing you might think of when solving this problem is to simply render the most distant objects first. This is called **depth sorting**. As you might imagine, this “solution” scales incredibly poorly. Doing it for each triangle is prohibitive, particularly with scenes with millions of triangles.

And the worst part is that even if you put in all the effort, it does not actually work. Not all the time at any rate. Many trivial cases can be solved via depth sorting, but non-trivial cases have real problems. You can have an arrangement of 3 triangles where each overlaps the other, such that there simply is no order you can render them in to achieve the right effect.
• One solution might be to tag fragments with the distance from the viewer. Then, if a fragment that is about to be written has a farther distance (i.e., the fragment is behind what was already drawn), we simply do not write that fragment to the output image. That way, if you draw a triangle behind other triangles, the fragment distances that were already written will be closer to the camera than the fragment distances of the new triangle. And thus, the particular fragments of that triangle will not be drawn. And since this works at the fragment level, it will work just as well for intersecting triangles or the 3 triangle arrangement depicted above.

• The “tag” is the window-space \( Z \) value. When triangles are transformed to window(screen) coordinates and after that proceed to scan conversion to produce fragments the \( Z \) position will range from 0 to 1, where 0 is the closest and 1 is the farthest.

• **Colors** output from the fragment shader are output into the **color image buffer**. Therefore it naturally follows that **depth values** would be stored in a **depth buffer** (also called a **z buffer**), because it stores \( Z \) values. The depth buffer is an image that is the same size as the main color buffer, that stores depth values as pixels rather than colors. Where a color is a 4-component vector, a depth is just a single floating-point value.

• Like the color buffer, the depth buffer for the main window is created automatically by OpenGL when OpenGL is initialized. OpenGL can even be created without a depth buffer. Since FreeGLUT takes care of initializing OpenGL for us, we tell it in the standard initialization code to create OpenGL with a depth buffer.

```c
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
```
• Writing the depth is not enough. The suggested idea requires stopping the fragment from writing anything if the current depth at that location is in front of this fragment's depth. This is called the **depth test**. In OpenGL, the test does not have to be in any particular direction; any of the typical numerical relation operator (greater than, less than, etc) will work fine. If the test passes, then the fragment's outputs (both color and depth) will be written to their appropriate buffer. If it fails, then they will not.

• To activate depth testing, we must call **glEnable(GL_DEPTH_TEST)**; the corresponding **glDisable** call will cause depth testing to cease. After activating testing, we need to call **glDepthFunc** to set the relation of the depth test. When the test performed by glDepthFunc will return true, the incoming fragment is written.

• The test functions can be **GL_ALWAYS** (always write the fragment), **GL_NEVER** (no fragments are written), **GL_LESS, GL_GREATER, GL_LEQUAL (<=), GL_GEQUAL (>=), GL_EQUAL, or GL_NOTEQUAL**. The test function puts the incoming fragment's depth on the left of the equation and on the right is the depth from the depth buffer. So **GL_LESS** means that, when the incoming fragment's depth is less than the depth from the depth buffer, the incoming fragment is written.

• **NOTE:** While the fragment depth value is automatically passed to the depth testing function during fragment generation, you can supply your own depth value for the fragment, inside the fragment shader using the **gl_FragDepth** output variable in a fragment shader. Actually gl_FragDepth will automatically be assigned the **gl_FragCoord.z** value, **gl_FragCoord** being a fragment shader input variable that contains the window relative coordinate (x, y, z, 1/w) values for the fragment.

• Speaking of window coordinates, there is one more issue we need to deal with when dealing with depth. The **glViewport** function defines the transform between normalized device coordinates (the range [-1, 1]) to window coordinates. But glViewport only defines the transform for the X and Y coordinates of the NDC-space vertex positions.
The window-space Z coordinate ranges from \([0, 1]\); the transformation from NDC's \([-1, 1]\) range is defined with the `glDepthRange` function. This function takes 2 floating-point parameters: the range `zNear` and the range `zFar`. These values are in window-space; they define a simple linear mapping from NDC space to window space. So if `zNear` is 0.5 and `zFar` is 1.0, NDC values of -1 will map to 0.5 and values of 1 will result in 1.0. (Do not confuse the range `zNear/zFar` with the camera `zNear/zFar` used in the perspective projection matrix computation.)

In this sample if you press the SPACE button you will trigger depth testing. Here's the code that is responsible for that:

```c
void keyboard (unsigned char key, int x, int y)
{
  if(key==32)
  {
    enableDepthTest=!enableDepthTest;
    if(enableDepthTest)
    {
      glEnable(GL_DEPTH_TEST);
      glDepthMask(GL_TRUE);
      glDepthFunc(GL_LEQUAL);
      glDepthRange(0.0f, 1.0f);
    }
    else
    {
      glDisable(GL_DEPTH_TEST);
    }
  }
  glutPostRedisplay ()
}
```
The above depth testing values are the most common depth testing parameters. It turns on depth testing, sets the test function to less than or equal to, and sets the range mapping to the full accepted range.

In the main() function we tell FreeGLUT to call our “keyboard()” function. When it will do this it will also pass the ASCII index of the pressed key (unsigned char key).

**NOTE**: The call to `glDepthMask` causes rendering to write the depth value from the fragment to the depth buffer. The activation of depth testing alone is not sufficient to actually write depth values. This allows us to have depth testing for objects where their own depth (the incoming fragment's depth) is not written to the depth buffer, even when their color outputs are written.

There is one more issue. We know what the depth value is in the depth buffer after a fragment is written to it. But what is its value before any rendering is done at all? Depth buffers and color buffers are very similar; color buffers get their initial colors from calling `glClear`. So you might imagine a similar call for depth buffers.

As it turns out, they share the same clearing call. If you recall, `glClearColor` sets the color for clearing color buffers. Similarly, `glClearDepth` sets the depth value that the depth buffer will be cleared to. In order to clear the depth buffer with `glClear`, you must use the `GL_DEPTH_BUFFER_BIT`. So, the render function's clearing, at the top of the function, happens as follows:

```c
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClearDepth(1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

This will set all of the depth values in the depth buffer to 1.0, which is our range zFar.
Additional reading:

http://www.arcsynthesis.org/gltut/Positioning/Tut05%20Overlap%20and%20Depth%20Buffering.html
28. Depth clamping

![Diagram of depth clamping](SAMPLE12)
If you run SAMPLE12 you will see this:

What we did with the top object is we moved it into $+z$ direction ($-z$ direction in OpenGL right handed coordinates system goes into monitor). It's cut off because part of it's geometry falls outside the near clipping plane and in a process called **clipping**. Remember that when we integrated a projection transformation we had to set 6 planes that will cut off geometry that we don't want to render, and one of this clipping planes was the near clipping plane.

If computer graphics is an elaborate illusion, then clipping utterly shatters this illusion. It's a big, giant hole that screams, “this is fake!” as loud as possible to the user. What can we do about this?
A solution is to use a mechanism called **depth clamping**. What this does is turn off camera (the frustum) near/far plane clipping altogether. Instead, the depth for these fragments are clamped to the [-1, 1] range in NDC space.

The code that triggers depth clamping on/off is shown below:

```c
void keyboard (unsigned char key, int x, int y)
{
    if (key == 32)
    {
        enableDepthClamping = !enableDepthClamping;
        if (enableDepthClamping)
        {
            glEnable(GL_DEPTH_CLAMP);
        }
        else
        {
            glDisable(GL_DEPTH_CLAMP);
        }
    }
    glutPostRedisplay();
}
```
• Pressing the SPACE button will execute `glEnable(GL_DEPTH_CLAMP);`  
  `/glDisable(GL_DEPTH_CLAMP);` Now it looks better:

Additional reading:
http://www.arcsynthesis.org/gltut/Positioning/Tut05%20Boundaries%20and%20Clipping.html
http://www.arcsynthesis.org/gltut/Positioning/Tut05%20Depth%20Clamping.html
28. Row vs Column vectors

• When multiplying a matrix by a vector, there are two ways you can write the multiplication. The first case is to use a column vector on the right:

\[
Mv = \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} m_{1,1}v_1 + m_{1,2}v_2 \\ m_{2,1}v_1 + m_{2,2}v_2 \end{bmatrix}
\]

• The second variant is to use a row vector on the left:

\[
v^TM = \begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} = \begin{bmatrix} v_1m_{1,1} + v_2m_{2,1} & v_1m_{1,2} + v_2m_{2,2} \end{bmatrix}
\]

• Both variants are absolutely valid, however notice that you get 2 different results for each case, because matrix multiplication is not commutative.

• Some prefer to write it in one way(OpenGL style) and others prefer to write it in another way. **So please remember one fact, if you use column vectors and then want to get the same result by using the same data expressed as a row vector you have to take the transpose of the matrix and vice versa.**
29. Transformations

- Throughout our course, we have discussed a number of different spaces. We have seen OpenGL-defined spaces like normalized device coordinate (NDC) space, clip-space, and window space. And we have seen user-defined spaces like camera space. But we have yet to formally discuss about what a space actually is.

- A **space** is a shorthand term for a **coordinate system**. For the purposes of this course, a coordinate system or space consists of the following:

  - The dimensionality of the space. 2D, 3D, 4D, etc.

  - A series of vectors in those dimensions that define the axes of the space. The directions do not have to be orthogonal (at right-angles) to one another, but there must be one axis per dimension. Each axis vector in the space has a name, like X, Y, Z, etc. These are called the **basis vectors** of a space.

  - A location in the space that defines the central **origin** point. The origin is the point from which all other points in the space are derived.

  - An area within this space in which points are valid. Outside of this range, positions are not valid. The range can be infinite depending on the space.
• A position or vertex in a space is defined as the sum of the basis vectors, where each basis vector is multiplied by a scalar value called a coordinate. Geometrically, this looks like the following:

• These are two different coordinate systems. The same coordinate, in this case (2, 2) (each basis vector is added twice) can have two very different positions, from the point of view of a neutral observer. What is interesting to note is that (2, 2) is the same value in their own coordinate system. This means that a coordinate itself is not enough information to know what it means; one must also know what coordinate system it is in before you can know anything about it.
• Take this vector (point in 3D space):

\[
\begin{bmatrix}
1 \\
-3 \\
4
\end{bmatrix} = \begin{bmatrix}
1 \\
0 \\
0
\end{bmatrix} + \begin{bmatrix}
0 \\
-3 \\
0
\end{bmatrix} + \begin{bmatrix}
0 \\
0 \\
4
\end{bmatrix}
\]

• The above vector \([1, -3, 4]\) can be interpreted as a displacement of \([1, 0, 0]\), followed by a displacement of \([0, -3, 0]\), followed by a displacement of \([0, 0, 4]\).

• In general, for any vector \(v\), we can write \(v\) in “expanded” form:

\[
v = \begin{bmatrix}
x \\
y \\
z
\end{bmatrix} = x \begin{bmatrix}
1 \\
0 \\
0
\end{bmatrix} + y \begin{bmatrix}
0 \\
1 \\
0
\end{bmatrix} + z \begin{bmatrix}
0 \\
0 \\
1
\end{bmatrix}
\]

• Let’s rewrite this expression in a slightly different form:

\[
v = x p + y q + z r
\]

• Notice that the unit vectors on the right-hand side are \(x\)-, \(y\)-, and \(z\)-axes. We have just expressed each coordinate of a vector as the signed displacement parallel to corresponding axes. Let’s rewrite the sum one more time. This time, we will define the vectors \(p\), \(q\), and \(r\) to be unit vectors pointing in the \(+x\), \(+y\), and \(+z\), directions, respectively:

\[
v = xp + yq + zr
\]
Now we have expressed the vector $v$ as a linear combination of the vectors $p$, $q$, and $r$. The vectors $p$, $q$, and $r$ are known as **basis vectors**. We are accustomed to using the cardinal axes as basis vectors, but, in fact, a coordinate space may be defined using any three vectors, provided the three vectors are linearly independent (which basically means that they don’t lie in a plane). If we construct a $3 \times 3$ matrix $M$ using $p$, $q$, and $r$ as the rows of the matrix, we get:

$$M = \begin{bmatrix} p_x & p_y & p_z \\ q_x & q_y & q_z \\ r_x & r_y & r_z \end{bmatrix}$$

Multiplying a vector by this matrix, we get (notice the use of row vectors which are placed in front of the matrix for the multiplication to be valid):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \begin{bmatrix} p_x & p_y & p_z \\ q_x & q_y & q_z \\ r_x & r_y & r_z \end{bmatrix} = \begin{bmatrix} xp_x + yq_x + zr_x \\ xp_y + yq_y + zr_y \\ xp_z + yq_z + zr_z \end{bmatrix} = x\mathbf{p} + y\mathbf{q} + z\mathbf{r}$$

This is the same as our original equation for computing $v$ after transformation. We have discovered the key idea that:

If we interpret the rows of a matrix (or columns if the multiplication vector is in form of a column on the right like in OpenGL) as the basis vectors of a coordinate space, then multiplication by the matrix performs a coordinate space transformation. If $aM = b$, we say that $M$ transformed $a$ to $b$. 
• From this point forward, the terms **transformation** and **multiplication** will be largely synonymous. The bottom line is that there’s nothing especially magical about matrices. They simply provide a compact way to represent the mathematical operations required to perform a coordinate space transformation. Furthermore, using linear algebra to manipulate matrices is a convenient way to take simple transformations and derive more complicated transformations.

• In the more recent samples, the ones dealing with perspective projections, we have been taking positions in one coordinate system (space) and putting them in another coordinate system. Our shaders implement the transformation from camera space to clip-space, and this was done using a matrix.
What Does a Matrix Look Like?

- We have stated that a matrix represents a coordinate space transformation. So when we visualize the matrix, we are visualizing the transformation, the new coordinate system. But what does this transformation look like? What is the relationship between a particular 3D transformation (i.e., rotation, shearing, etc.) and those nine numbers inside a 3×3 matrix?

- To begin to answer these questions, let’s examine what happens when the basis vectors [1, 0, 0], [0, 1, 0], and [0, 0, 1] are multiplied by an arbitrary matrix M:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
m_{11} & m_{12} & m_{13} \\
m_{21} & m_{22} & m_{23} \\
m_{31} & m_{32} & m_{33} \\
\end{bmatrix}
= 
\begin{bmatrix}
m_{11} & m_{12} & m_{13} \\
m_{21} & m_{22} & m_{23} \\
m_{31} & m_{32} & m_{33} \\
\end{bmatrix}
\]

- As you can see, when we multiply the basis vector [1, 0, 0] by M, the resulting vector is the first row of M. Similar statements can be made regarding the other two rows. This is a critical observation:

Each row of a matrix (or column in case when column vectors are used on the right side of vector matrix multiplication, like in OpenGL) can be interpreted as a basis vector after transformation.
This is the same basic idea that we discovered in the previous section, only we have come at it from a slightly different angle. This very powerful concept has two important implications:

- First, we have a simple way to take any matrix and visualize what sort of transformation the matrix represents.

- Second, we have the ability to make the reverse construction—given a desired transformation (i.e. rotation, scale, etc.). We can derive a matrix which represents that transformation. All we have to do is figure out what the transformation does to basis vectors and fill in those transformed basis vectors into the rows of a matrix.

Let’s look at a couple of examples. First we will examine a 2D example to get ourselves warmed up and then a full-fledged 3D example. Examine the following 2×2 matrix

$$M = \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix}$$

What sort of transformation does this matrix represent? First, let’s extract the basis vectors p and q from the rows of the matrix:

$$p = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad q = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$
• Figure below shows these vectors in the Cartesian plane, along with the “original” basis vectors (the x-axis and y-axis), for reference:

As Figure above illustrates, the +x basis vector is transformed into the vector labeled p above, and the y basis vector is transformed into the vector labeled q. So one way to visualize a matrix in 2D is to visualize the “L” formed by the row vectors. In this example, we can easily see that part of the transformation represented by M is a counterclockwise rotation of about 26 degrees.
• Drawing an object inside the parallelogram formed by new axes, helps better understand the idea:

- It is clear that our example matrix $M$ not only rotates the coordinate space, it also scales it.
• We can extend the techniques we used to visualize 2D transformations into 3D. In 2D, we had two basis vectors that formed an “L.” In 3D, we have three basis vectors, and they form a “tripod.” First, let’s show an object before transformation. Figure below shows a teapot, a unit cube, and the basis vectors in the “identity” position:

![Teapot, unit cube, and basis vectors before transformation](image)

• Now consider the 3D transformation matrix below:

\[
\begin{bmatrix}
0.707 & -0.707 & 0 \\
1.250 & 1.250 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
Extracting the basis vectors from the rows of the matrix, we can visualize the transformation represented by this matrix. The transformed basis vectors, cube, and teapot are shown below:

As you can see, the transformation consists of a clockwise rotation about the z-axis by about 45° and a non-uniform scale that makes the teapot “taller” than it was originally. Notice that the +z basis vector was unaltered by the transformation because the third row of the matrix is [0,0,1].
• You probably noticed the use of 3x3 matrices in previous example. Those matrices performed the so called "linear transformations". In linear transformations, if we add two vectors and then perform the transformation, we get the same result as if we perform the transformation on the two vectors individually and then add the transformed vectors. Likewise, if we scale a vector and then transform it, we should get the same resulting vector as when we transform the vector and then scale it. Mathematically, a mapping \( F(a) \) is linear if:

\[
F(a + b) = F(a) + F(b)
\]

and

\[
F(ka) = kF(a)
\]

• The downside of 3x3 matrices is that they cannot include a translation transformation.

• For the purpose of implementing translation transformation we use 4x4 matrices and we already have seen them in work in our projection samples.
Model Space

• Before we look more detailed into transformation, we must define a new kind of space: **model space**. This is a user-defined space, but unlike camera space, model space does not have a single definition. It is instead a catch-all term for the space that a particular object begins in. Coordinates in buffer objects, passed to the vertex shaders as vertex attributes are de facto in model space.

• The transformation operation being discussed in the following samples is the transform from model space to camera space. Our shaders already know how to handle camera-space data; all they need is a way to transform from model space to camera space.

  **Translation** SAMPLE13

• The simplest space transformation operation is translation. Indeed, we have not only seen this transform before, it has been used in all of the samples with a perspective projection. Recall this line from the vertex shaders:

  ```
  vec4 cameraPos = vertexPosition + vec4(offset.x, offset.y, offset.z, 0.0);
  ```

• This is a **translation transformation**. All a translation needs to do is add a vector to all of the coordinates in that space. The vector added to these values is the location of where the user wants the origin point relative to the destination coordinate system.
Here is a more concrete example. Let us say that an object which in its model space is near its origin. This means that, if we want to see that object in front of the camera, we must position the origin of the model in front of the camera. If the extent of the model is only \([-1, 1]\) in model space, we can ensure that the object is visible by adding this vector to all of the model space coordinates: \((0, 0, -3)\). This puts the origin of the model at that position in camera space.

Translation is ultimately just that simple. So let's make it needlessly complex. And the best tool for doing that: matrices. Oh, we could just use a 3D uniform vector to pass an offset to do the transformation. But matrices have hidden benefits we will explore very soon.
• Multiplying a 4D vector \((v_x, v_y, v_z, 1)\) (the vector must be a column vector and placed to the right of the matrix) by the above matrix in which we placed the desired translation\((x, y, z)\) information in the 4\(^{th}\) column will produce a new vector that is effectively translated by the \(x, y, z\) translation amount. The key for the translation to be introduced in our resulting vector is the "w" term of the initial vector. This "w" term is 1 and when it is multiplied by "\(x\)", "\(y\)" and "\(z\)" from the 4\(^{th}\) column, the result of the multiplication is added to \(v_x, v_y\) and \(v_z\) values of the initial vector.

• Here's the output of SAMPLE13:

![Sample output](image)

• The sample renders two objects in motion.
Here's their rendering code:

```cpp
translationTimer.AdvanceTime();

//first objects
glm::mat4 transformMatrix = glm::mat4(1.0f);
transformMatrix[3]=glm::vec4(sin(translationTimer.GetElapsedTime())*5,0,-5,1);
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &transformMatrix[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);

//second object
transformMatrix = glm::mat4(1.0f);
transformMatrix[3]=glm::vec4(0,sin(translationTimer.GetElapsedTime())*5,-5,1);
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &transformMatrix[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);
```

The `glm::mat4(1.0f)` constructor that takes only a single value constructs what is known as a diagonal matrix. That is a matrix with all zeros except for along the diagonal from the upper-left to the lower-right. The values along that diagonal will be the value passed to the constructor. An identity matrix is just a diagonal matrix with 1 as the value along the diagonal.

Setting the translation transformation into the matrix is done in this line:

```cpp
transformMatrix[3]=glm::vec4(sin(translationTimer.GetElapsedTime())*5,0,-5,1);
```

The above code simply replaces the 4th column of our matrix with the offset values for x, y and z.

Our vertex shader main() function now looks like this:
void main()
{
    vec4 cameraPos = modelToCameraMatrix * vertexPosition;
    gl_Position = cameraToClipMatrix * cameraPos;
    colorForFS=colorIn;
}

• Notice the transformation for model to camera space and from camera to clip space.
Another kind of transformation is scaling. In terms of our previous definition of a coordinate system, this means that our basis vectors are getting shorter or longer.

Scaling can be uniform, which means each basis vector is scaled by the same value. A non-uniform scale means that each basis can get a different scale or none at all.

Uniform scales are used to allow objects in model space to have different units from the units used in camera space. For example, a modeler may have generated the model in inches, but the world uses centimeters. This will require applying a uniform scale to all of these models to compensate for this. This scale should be 2.54, which is the conversion factor from inches to centimeters.
Given the scale factors in x, y and z direction, here's the matrix that scales the vector when multiplied with it:

\[
\text{Scale} = (x, y, z) \\
\begin{bmatrix}
  x & 0 & 0 & 0 \\
  0 & y & 0 & 0 \\
  0 & 0 & z & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

If you run SAMPLE14 you'll see two objects that are animated by modifying the scale value in into all 3 axes.
Here's the code from our render() function where we initialize the scaling matrix and render our objects:

```cpp
// prepare the scaling matrix for object 1
static float sx=0, sy=0, sz=0;
sx+=0.01f;
sy+=0.01f;
sz+=0.01f;

glm::mat4 transformMatrix = glm::mat4(1.0f);
transformMatrix[0].x = sin(sx)+2;
transformMatrix[1].y = cos(sx)+2;
transformMatrix[2].z = sin(sz)/2+cos(sz)/2+2;

// render object 1
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &transformMatrix[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);

glBindVertexArray(vaoObject2);

// prepare the scaling matrix for object 2
transformMatrix[0].x = cos(sx)+2;
transformMatrix[1].y = sin(sx)+2;
transformMatrix[2].z = cos(sz)/2+sin(sz)/2+2;

// render object 2
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &transformMatrix[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);
```

Please note how we use `transformMatrix[1].y` to access the second element of second column. An alternative would be `transformMatrix[1][1].`
• **Inversion and winding order.** Scales can be theoretically negative, or even 0. A negative scale changes the direction of an axis. This causes vertices transformed with this scale to flip across the origin in that axis's direction. This is called an inversion. This can have certain unintended consequences. In particular, it can change the winding order of vertices.

• In one of our previous samples, we introduced the ability to cull triangles based on the order in which the vertices appeared in window space. Depending on which axis you negate, relative to camera space, an inversion can flip the expected winding order of vertices. Thus, triangles that were, in model space, forward-facing now in camera space are backwards-facing. And vice-versa. Negative scaling should be used with care unless you don't turn the face culling off completely.
A rotation transformation is the result of the orientation of the initial space being different from the orientation of the destination space. The basis vectors of the space do not change orientation relative to one another, but relative to the destination coordinate system, they are pointed in different directions than they were in their own coordinate system. A rotation looks like this:

Rotations are usually considered the most complex of the basic transformations, primarily because of the math involved in computing the transformation matrix. Generally, rotations are looked at as an operation, such as rotating around a particular basis vector or some such.
A rotation matrix is just a transform that expresses the basis vectors of the input space in a different orientation. The length of the basis vectors will be the same, and the origin will not change. Also, the angle between the basis vectors will not change. All that changes is the relative direction of all of the basis vectors.

A rotation matrix is not really a “rotation” matrix; it is an orientation matrix. It defines the orientation of one space relative to another space. Remember this, and you will avoid many pitfalls when you start dealing with more complex transformations.

Below are the matrices used for rotation around main axis:
A common rotation question is to therefore compute a rotation around an arbitrary axis. Or to put it more correctly, to determine the orientation of a space if it is rotated around an arbitrary axis. The axis of rotation is expressed in terms of the initial space. Here's the required transform matrix:

\[
\begin{bmatrix}
    x^2 + (1 - x^2)C & iCxy - zS & iCxz + yS & 0 \\
iCxy + zS & y^2 + (1 - y^2)C & iCy - xS & 0 \\
iCxz - yS & iCy + xS & z^2 + (1 - z^2)C & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Here's the output of SAMPLE15:
Here's the code from the render() function:

```cpp
static float rx=0,ry=0,rz=0;
rx+=0.1f;ry+=0.1f;rz+=0.1f;

//prepare the first rotation matrix and render object
glm::mat4 transformMatrix = glm::mat4(1.0f);
transformMatrix=glm::rotate(transformMatrix,rx,glm::vec3(1,0,0));
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &transformMatrix[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);

//prepare the second rotation matrix and render object
glm::mat4 transformMatrix = glm::mat4(1.0f);
transformMatrix=glm::rotate(transformMatrix,ry,glm::vec3(0,1,0));
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &transformMatrix[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);

//prepare the third rotation matrix and render object
glm::mat4 transformMatrix = glm::mat4(1.0f);
transformMatrix=glm::rotate(transformMatrix,rz,glm::vec3(0,0,1));
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &transformMatrix[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);
```
A few notes on SAMPLE 15:

- Each object is rotated about one of 3 basis axes.
- Note that we use the glm::rotate() function to build the rotation matrix by specifying the angle of rotation and the rotation axis.
- Each object is rendered from the same vertex data but before rendering we supply to the vertex shader a different transformation matrices, as well as manually translate the object in the vertex shader via the offset uniform (we'll soon see how that translation can be integrated in the same matrix as the rotation transformation):

```cpp
void main()
{
   vec4 cameraPos = modelToCameraMatrix * vertexPosition;
   cameraPos+=vec4(offset.x, offset.y, offset.z, 0.0);
   gl_Position = cameraToClipMatrix * cameraPos;
   colorForFS=colorIn;
}
```

Additional reading:

3D Math Primer for Graphics and Game Development 1st Edition, Chapters 7,8,9
http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2006.html
http://www.arcsynthesis.org/gltut/Positioning/Tut06%20Translation.html
http://www.arcsynthesis.org/gltut/Positioning/Tut06%20Scale.html
http://www.arcsynthesis.org/gltut/Positioning/Tut06%20Rotation.html
30. Hierarchical Models SAMPLE16
In SAMPLE15 we multiplied the vertices inside the vertex shader with the modelToCameraMatrix which contained our rotation transformation, and then translated the vertex by an offset.

Actually these two operation could have been combined in a single matrix and this is where matrices excel. Successive transformations can be seen as doing successive multiplication operations. For example, if S is a pure scale matrix, T is a pure translation matrix, and R is a pure rotation matrix, then the shader can compute the result of a transformation as follows:

```
vec4 temp;
temp = T * position;
temp = R * temp;
temp = S * temp;
gl_Position = cameraToClipMatrix * temp;
```

In mathematical terms, this would be the following series of matrix operations:

Final = C*S*R*T*position, where C is the camera-to-clip space transformation matrix.

Matrix math gives us an optimization. Matrix math is not commutative: S*R is not the same as R*S. However, it is associative: (S*R)*T is the same as S*(R*T). The usual grouping for vertex transformation is this: Final = C*(S*(R*(T*position))). But this can easily be regrouped as:

Final = (((C*S)*R)*T)*position.

This would in fact be slower for the shader to compute, since full matrix-to-matrix multiplication is much slower than matrix-to-vector multiplication. But the combined matrix (((C*S)*R)*T) is fixed for all of a given object's vertices. This can be computed on the CPU, and all we have to do is upload a single matrix to OpenGL. And since we're already uploading a matrix to OpenGL for each object we render, this changes nothing about the overall performance characteristics of the rendering (for the graphics hardware).
• **This** is one of the main reasons matrices are used. You can build an incredibly complex transformation sequence with dozens of component transformations. And yet, all it takes for the GPU to use this to transform positions is a single vector/matrix multiplication.

**Order of transformation**

• As previously stated, matrix multiplication is not commutative. This means that the combined transform $S \times T$ is not the same as $T \times S$. Let us explore this further. This is what these two composite transform matrices look like:

![Order of Transformation](image)

• The transform $S \times T$ actually scales the translation part of the resulting matrix. This means that the vertices will not just get farther from each other, but farther from the origin of the destination space. It is the difference between these two transforms:
• The above multiplications are better visualized in the figure below:

• **First row**: Translation happens first, Scaling second. **Second row**: Scaling first, Translation second.

• What you must understand is that something special happens between S and T. In the first row, S is being applied to positions that are not from model space (the space the original vertices were in), but are in post translation space. This is an intermediate coordinate system defined by T. Remember: a matrix, even a translation matrix, defines a full-fledged coordinate system. So S in first row, now acts on the T-space position of the vertices.
Orientation (rotation) matrices have the same issue. The orientation is always local to the origin in the current space of the positions. So a rotation matrix must happen before the translation matrix. Scales generally should happen before orientation; if they happen afterwards, then the scale will be relative to the new axis orientation, not the model-space one. This is fine if it is a uniform scale, but a non-uniform scale will be problematic.

There are reasons to put a translation matrix first. If the model-space origin is not the point that you wish to rotate or scale around, then you will need to perform a translation first, so that the vertices are in the space you want to rotate from, then apply a scale or rotation. Doing this multiple times can allow you to scale and rotate about two completely different points.

Hierarchical models

In more complex scenes, it is often desirable to specify the transform of one model relative to the model space transform of another model. This is useful if you want one object (object B) to pick up another object (object A). The object that gets picked up needs to follow the transform of the object that picked it up. So it is often easiest to specify the transform for object B relative to object A. This is exactly what we do in this sample, we render a hierarchical model of an arm. And please note – ALL 3D PRISMS OF THIS ARM ARE JUST ONE INSTANCE OF GEOMETRY THAT IS TRANSFORMED BY A DIFFERENT MATRIX BEFORE RENDERING.
• In hierarchical models, the final transform for any of the component pieces is a sequence of all of the transforms of its parent transform, plus its own model space transform. Models in this transform have a parent-child relationship to other objects.

• For the purposes of this discussion, each complete transform for a model in the hierarchy will be called a **node**. Each node is defined by a specific series of transformations, which when combined yield the complete transformation matrix for that node. Usually, each node has a translation, rotation, and scale, though the specific transform can be entirely arbitrary. What matters is that the full transformation matrix is relative to the space of its parent, not camera space.

• So if you have a node who's translation is (3, 0, 4), then it will be 3 X-units and 4 Z-units from the origin of its parent transform. The node itself does not know or care what the parent transform actually is; it simply stores a transform relative to that.

• Technically, a node does not have to have a mesh. It is sometimes useful in a hierarchical model to have nodes that exist solely to position other, visible nodes. Or to act as key points for other purposes, such as identifying the position of the gun's muzzle to render a muzzle flash.

• This sample is interactive; the relative angles of the nodes can be changed with keyboard commands.

<table>
<thead>
<tr>
<th>Node Angle</th>
<th>Increase/Left</th>
<th>Decrease/Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Spin</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>Arm Raise</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>Elbow Raise</td>
<td>R</td>
<td>F</td>
</tr>
<tr>
<td>Wrist Raise</td>
<td>T</td>
<td>G</td>
</tr>
<tr>
<td>Wrist Spin</td>
<td>Z</td>
<td>C</td>
</tr>
<tr>
<td>Finger Open/Close</td>
<td>Q</td>
<td>E</td>
</tr>
</tbody>
</table>
• Structure of the sample:

• The class **Hierarchy** stores the information for our hierarchy of nodes. It stores the relative positions for each node, as well as angle information and size information for the size of each rectangle. The rendering code in render() function simply does the usual setup work and calls Hierarchy::Draw(), where the real work happens.

• The Hierarchy::Draw() function looks like this:

```cpp
void Draw()
{
    MatrixStack modelToCameraStack;

    glUseProgram(theProgram);
    glBindVertexArray(vao);

    modelToCameraStack.Translate(posBase);
    modelToCameraStack.RotateY(angBase);

    //Draw left base.
    {
        modelToCameraStack.Push();
        modelToCameraStack.Translate(posBaseLeft);
        modelToCameraStack.Scale(glm::vec3(1.0f, 1.0f, scaleBaseZ));
        glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &modelToCameraStack.Top()[0][0]);
        glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);
        modelToCameraStack.Pop();
    }

    //Draw right base.
    {
    }
}
```
modelToCameraStack.Push();
modelToCameraStack.Translate(posBaseRight);
modelToCameraStack.Scale(glm::vec3(1.0f, 1.0f, scaleBaseZ));
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &modelToCameraStack.Top()[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);
modelToCameraStack.Pop();

//Draw main arm.
DrawUpperArm(modelToCameraStack);

glBindVertexArray(0);
glUseProgram(0);

• The MatrixStack object created in the very first line is a class that is also a part of this project. It implements the concept of a matrix stack. The matrix stack is a method for dealing with transformations in hierarchical models.

• A stack is a particular data structure concept. Stacks store a controlled sequence of objects. But unlike arrays, linked lists, or other general data structures, there are only 3 operations available to the user of a stack: push, pop, and peek. Push places a value on the top of the stack. Pop removes the value on the top of the stack, making the previous top the current top. And peek simply returns the current value at the top of the stack.

• A matrix stack is, for the most part, a stack where the values are 4x4 transformation matrices. Matrix stacks do have a few differences from regular stacks. C++ has an object, std::stack, that implements the stack concept. MatrixStack is a wrapper around that object, providing additional matrix stack functionality.
• A matrix stack has a current matrix value. An initially constructed matrix stack has an identity matrix. There are a number of functions on the matrix stack that multiply the current matrix by a particular transformation matrix; the result becomes the new current matrix. For example, the MatrixStack::RotateX function multiplies the current matrix by a rotation around the X axis by the given angle.

• The MatrixStack::Push function takes the current matrix and pushes it onto the stack. The MatrixStack::Pop function makes the current matrix whatever the top of the stack is, and removes the top from the stack. The effect of these is to allow you to save a matrix, modify the current matrix, and then restore the old one after you have finished using the modified one. And you can store an arbitrary number of matrices, all in a specific order. This is invaluable when dealing with a hierarchical model, as it allows you to iterate over each element in the model from root to the leaf nodes, preserving older transforms and recovering them as needed.

• In the Draw code, the first thing we do is establishing the position of the arm and it's angle of rotation. Translation is applied to the stack first, followed by an X rotation based on the current angle.

```cpp
modelToCameraStack.Translate(posBase);
modelToCameraStack.RotateY(angBase);
```

• Note that the order of operations is backwards from what we said previously. That's because the matrix stack looks at transforms backwards. When we said earlier that the rotation should be applied before the translation, that was with respect to the position. That is, the equation should be T*R*v, where v is the position. What we meant was that R should be applied to v before T. This means that R comes to the right of T.
• Rather than applying matrices to vertices, we are applying matrices to each other. The matrix stack functions all perform right-multiplication; the new matrix being multiplied by the current is on the right side. The matrix stack starts with the identity matrix. To have it store T*R, you must first apply the T transform, which makes the current matrix the current matrix is I*T. Then you apply the R transform, making the current matrix I*T*R.

• Right-multiplication is necessary, as the whole point of using the matrix stack is so that we can start at the root of a hierarchical model and save each node's transform to the stack as we go from parent to child. That simply would not be possible if matrix stacks left-multiplied, since we would have to apply the child transforms before the parent ones.

• The next thing that happens is that the matrix is preserved by pushing it on the stack (we are saving the position of the whole arm and it's rotation so that we can use it later). After this, a translation and scale are applied to the matrix stack (the left base of the arm is scaled and then translated). The stack's current matrix is uploaded to the program, and a model is rendered. Then the matrix stack is popped, restoring the original transform of our arm.

```cpp
modelToCameraStack.Push();
modelToCameraStack.Translate(posBaseLeft);
modelToCameraStack.Scale(glm::vec3(1.0f, 1.0f, scaleBaseZ));
glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, &modelToCameraStack.Top()[0][0]);
glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData), GL_UNSIGNED_SHORT, 0);
modelToCameraStack.Pop();
```

• So the left base of our arm is first scaled by `scaleBaseZ`, then translated to `posBaseLeft`, then rotated by `angBase` and then translated again to `posBase`.
• At the bottom of the base drawing function is a call to draw the upper arm.

\[
\text{DrawUpperArm(modelToCameraStack)};
\]

• That function looks similar to the Draw() function: apply the model space matrix to the stack, push, apply a matrix, render, pop, call functions for child parts. All of the functions, to one degree or another, look like this. Indeed, they all looks similar enough that you could probably abstract this down into a very generalized form. And indeed, this is frequently done by scene graphs and the like. The major difference between the child functions and the root one is that this function has a push/pop wrapper around the entire thing. Though since the root creates a MatrixStack to begin with, this could be considered the equivalent.

Additional reading:
http://www.arcsynthesis.org/gltut/Positioning/Tut06%20Fun%20with%20Matrices.html
31. World in motion SAMPLE17
• In this sample we'll learn how to navigate through a world of teapots with a dynamic **camera**. Back in the perspective projection sample we defined the camera space as a 3D coordinate space in which the origin is at the center of projection. The camera always looks down the Z axis, with objects that have a negative Z being considered in front of the camera.

![Camera Diagram](image)

• Right now, the problem is that we transform all of the objects from their individual model spaces to camera space directly(figure above) which doesn't give us the flexibility to manipulate this camera. Our goal is to be able to position the camera somewhere arbitrarily on our world and to be able to see the world from that position. Like below:

![Camera Position Diagram](image)
To do that, we will introduce an intermediate space between model and camera space; let us call this space **world space**.

All objects will be transformed into world space first. The camera itself will also have a **particular position** and **orientation** in world space.

What we have to do now, in order to be able to see the world through our camera is to transform object from world space to camera space. The **world-to-camera** transformation matrix is the inverse of the camera-to-world matrix. The camera-to-world matrix is the combination of a translation to the camera's position and a rotation to the camera's orientation. Thus, if \( M \) is the 3x3 rotation matrix corresponding to the camera's orientation and \( t \) is the camera's position, then the 4x4 camera-to-world matrix is:

\[
\begin{pmatrix}
M_{00} & M_{01} & M_{02} & t_x \\
M_{10} & M_{11} & M_{12} & t_y \\
M_{20} & M_{21} & M_{22} & t_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

To achieve the **world-to-camera** matrix we have to take the inverse of the above, otherwise speaking we have to perform the inverse translation and the inverse orientation.

But, instead of computing the camera-to-world matrix and inverting it, a more efficient (and numerically stable) alternative is to calculate the world-to-camera matrix directly. To do so, just invert the camera's position (by negating all 3 coordinates) and its orientation (by negating the rotation around X,Y, Z axes, and adjusting them to be in their proper ranges).
For the sake of reference, here is a diagram of the full transform for vertex positions, from the initial attribute loaded from the buffer object, to the final window-space position.

- In SAMPLE17 we will construct 3 matrices: `modelToWorldMatrix` – transforms our objects in world space.
  
  `worldToCameraMatrix` – transforms our world space objects to camera space. Finally `cameraToClipMatrix` - transforms our camera space objects to clip space.
• The controls of this sample are as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Increase/Left</th>
<th>Decrease/Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move camera target up/down</td>
<td>E</td>
<td>Q</td>
</tr>
<tr>
<td>Move camera target horizontally</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>Move camera target vertically</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>Rotate camera horizontally around target</td>
<td>L</td>
<td>J</td>
</tr>
<tr>
<td>Rotate camera vertically around target</td>
<td>I</td>
<td>K</td>
</tr>
<tr>
<td>Move camera towards/away from target</td>
<td>U</td>
<td>O</td>
</tr>
</tbody>
</table>

• Here's our Vertex Shader:

```glsl
#version 120
layout (location = 0) in vec3 VertexPosition;
uniform mat4 cameraToClipMatrix;
uniform mat4 worldToCameraMatrix;
uniform mat4 modelToWorldMatrix;

void main()
{
    vec4 temp = modelToWorldMatrix * vec4(VertexPosition,1.0);
    temp = worldToCameraMatrix * temp;
    gl_Position = cameraToClipMatrix * temp;
}
```
• And here's our render() function:

```c
void render(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //render 4 teapots
    float dist = 0.0f;
    for( int i = 0 ; i < 4; i++ )
    {
        prog.setUniform("fragmentColor", glm::vec4(1.0f-i*0.2f,1.0f-i*0.2f,1.0f-i*0.2f,1.0f));
        modelToWorldMatrix = mat4(1.0f);
        modelToWorldMatrix *= glm::translate(vec3(dist * 0.6f - 1.0f,0.0f,-dist));
        modelToWorldMatrix *= glm::rotate(-90.0f, vec3(1.0f,0.0f,0.0f));
        setMatrices();
        teapot->render();
        dist += 7.0f;
    }

    //render the plane
    prog.setUniform("fragmentColor", glm::vec4(0,0,1,1));
    modelToWorldMatrix = mat4(1.0f);
    setMatrices();
    plane->render();

    glutSwapBuffers();
}
```
What we do in the render() function is we first draw 4 teapots and for each teapot, before drawing it, we are calling the `setMatrices()` function, where we initialize our 3 matrix transformations. Let's take a look at the `setMatrices()` function:

```cpp
void setMatrices()
{
    const glm::vec3 &camPos = ResolveCamPosition();
    worldToCameraMatrix = glm::lookAt(camPos, g_camTarget, glm::vec3(0.0f, 1.0f, 0.0f));
    prog.setUniform("modelToWorldMatrix", modelToWorldMatrix);
    prog.setUniform("worldToCameraMatrix", worldToCameraMatrix);
    prog.setUniform("cameraToClipMatrix", cameraToClipMatrix);
}
```

• The first line computes the position of the camera from the spherical coordinates that we use to identify the position of the camera in 3D space. We'll talk on them on the next page.

• On the second line we finally get our hands on the **world to camera** transformation matrix that we construct with the help of glm's library helper function `glm::lookAt`. We feed to that function the **position** of our camera, the point we **look at** and the **up** vector (the up vector is used to create a cross product with the vector from position of the camera to the point we look at) and it returns us a world to camera transformation matrix.

• The next three lines feed our matrices to the shaders. The modelToWorldMatrix is computed for each object in the render() function before setMatrices() call and the cameraToClipMatrix is computed inside the reshape() function.
• For this sample we use *spherical coordinates* (*polar coordinates*) to keep the position of our camera in 3D space. Then, we convert these spherical coordinates (which come from user's input) to Cartesian 3D coordinates with the help of function `ResolveCamPosition();`

• Spherical coordinates are three dimensional, so they have 3 values. One value, commonly given the name “r” (for radius) represents the distance of the coordinate from the center of the coordinate system. This value is on the range \([0, \infty)\). The second value, called “φ” (phi), represents the angle in the elliptical plane. This value extends on the range \([0, 360)\). The third value, called “θ” (theta), represents the angle above and below the elliptical plane. This value is on the range \([0, 180]\), where 0 means straight up and 180 means straight down. The concept is illustrated in the picture below:
Spherical coordinates is a very convenient coordinate system for positioning an object around another object, particularly if you want to move along spheres relative to another object. The transformation from spherical coordinates back to Euclidean geometric coordinates is implemented in `ResolveCamPosition()`.

```cpp
glm::vec3 ResolveCamPosition()
{
    float phi = DegToRad(g_sphereCamRelPos.x);
    float theta = DegToRad(g_sphereCamRelPos.y + 90.0f);
    float fSinTheta = sinf(theta);
    float fCosTheta = cosf(theta);
    float fCosPhi = cosf(phi);
    float fSinPhi = sinf(phi);
    glm::vec3 dirToCamera(fSinTheta * fCosPhi, fCosTheta, fSinTheta * fSinPhi);
    return (dirToCamera * g_sphereCamRelPos.z) + g_camTarget;
}
```

- The global variable `g_sphereCamRelPos` contains the spherical coordinates. The X value contains φ, the Y value contains θ, and the Z value is the radius.

- The Theta value used in our spherical coordinates is slightly different from the usual. Instead of being on the range [0, 180], it is on the range [-90, 90]; this is why there is an addition by 90 degrees before computing the Theta angle in radians.

- `g_camTarget` is another global variable in this sample which we use to store the point where our camera is looking at. The implementation of `glm::lookAt` function is non-trivial and is beyond the scope of this course.

Additional reading:
29. Primitive Drawing

- The `glDraw*` commands, whether using indexed rendering or array rendering, establish a vertex stream. A vertex stream is an ordered list of vertices, with each vertex having a specific set of vertex attributes. A vertex stream is processed by the vertex shader in order.

- In array rendering, the order is determined by the order of the vertices in the attribute arrays. In indexed rendering, the order is determined by the order of the indices.

- Once the stream is processed by the vertex shader, it must be interpreted into something meaningful by OpenGL. Every `glDraw*` command takes, as its first parameter, a value that tells OpenGL how to interpret the stream. Thus far, we have used GL_TRIANGLES, but there are many options. This parameter is called the rendering mode or primitive.

- The parameter actually determines two things. The first it determines is what kind of things the vertex stream refers to; this is the primitive type. OpenGL can render points and lines in addition to triangles. These are all different primitive types.

- The other thing the parameter determines is how to interpret the vertex stream for that primitive type. This is the primitive representation. GL_TRIANGLES says more than simply that the primitive type is triangles.

- What GL_TRIANGLES means is that a vertex stream will generate triangles as follows: (0, 1, 2), (3, 4, 5), (6, 7, 8), .... The numbers represent vertices in the vertex stream, not indexed rendering indices. Among other things, this means that the vertex stream must have a length divisible by 3. For N vertices in the stream, this representation will generate N / 3 triangles.
There are two other triangular primitive representations: \texttt{GL\_TRIANGLE\_FAN}, \texttt{GL\_TRIANGLE\_STRIP}.

\texttt{GL\_TRIANGLES} takes each independent set of 3 vertices as a single triangle. \texttt{GL\_TRIANGLE\_FAN} takes the first vertex and holds on to it. Then, for every vertices and its next vertex, a triangle is made out of these two plus the initial vertex. So \texttt{GL\_TRIANGLE\_FAN} will generate triangles as follows: (0, 1, 2), (0, 2, 3), (0, 3, 4), …. Visually, a triangle fan looks like this:

![Triangle Fan Diagram](image)

The numbers in the figure above represent the order that the vertices are in in the vertex stream. The red line shows the triangle edges that are directly specified by the vertex stream. All other edges are generated automatically by the primitive representation.

This is why it is called a “fan”. The number of vertices in a triangle fan vertex stream must be 3 or greater, but can be otherwise any number. For \( N \) vertices in a stream, triangle fans will generate \( N-2 \) triangles.
• The **GL_TRIANGLE_STRIP** primitive representation means that every 3 adjacent vertices will generate a triangle, in order. So strips generate triangles as follows: (0, 1, 2), (1, 2, 3), (2, 3, 4), …. Visually, a triangle strip looks like this:

![Triangle Strip Diagram](image)

• Like with triangle fans, the number of vertices must be 3 or greater, but can be any number otherwise. For \( N \) vertices in a stream, triangle strips will generate \( N-2 \) triangles.

• Winding Order. There is one other issue with triangle strips. This has to do with the winding order of the triangles. The winding order for the triangles in a strip looks like this:

![Winding Order Diagram](image)
• Notice how it alternates between clockwise and counter-clockwise. This means that, regardless of what face you consider front, and what face you cull, you'll always lose about half of the faces.

• However, OpenGL is rather intelligent about this. Triangle strips do face culling differently. For every second triangle, the one who's winding order is opposite from the first triangle's order, the winding order is considered backwards for culling purposes.

• So if you have set the front face to be clockwise, and have face culling cull back-facing triangles, everything will work exactly as you expect so long as the order of the first triangle is correct. Every even numbered triangle will be culled if it has a clockwise winding, and every odd numbered triangle will be culled if it has a counter-clockwise winding.

• The other primitive types available in OpenGL 3.1 and above are: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES.

Additional reading: [http://www.arcsynthesis.org/gltut/Positioning/Tut07%20Primitive%20Drawing.html](http://www.arcsynthesis.org/gltut/Positioning/Tut07%20Primitive%20Drawing.html)
• If your program involves multiple shader programs that use the same uniform variables, one has to manage the variables separately for each program. Uniform locations are generated when a program is linked, so the locations of the uniforms may change from one program to the next. The data for those uniforms may have to be re-generated and applied to the new locations.

• **Uniform blocks** were designed to ease the sharing of uniform data between programs. With uniform blocks, one can create a buffer object for storing the values of all the uniform variables, and bind the buffer to the uniform block. Then when changing programs, the same buffer object need only be re-bound to the corresponding block in the new program.

• A uniform block is simply a group of uniform variables defined within a syntactical structure known as a uniform block. For example in a shader we would define it as follows:

```glsl
uniform BlobSettings {
  vec4 InnerColor;
  vec4 OuterColor;
  float RadiusInner;
  float RadiusOuter;
};
```

• This defines a block with the name “BlobSettings” that contains four uniform variables. With this type of block definition, the variables within the block are still part of the global scope and do not need to be qualified with the block name.
• The buffer object used to store the data for the uniforms is often referred to as a **uniform buffer object**. As any buffer object, it is created in the OpenGL application, outside shaders.

• Consider the following fragment shader and note the use of uniform blocks:

```glsl
#version 400
in vec3 TexCoord;
layout (location = 0) out vec4 FragColor;

uniform BlobSettings {
    vec4 InnerColor;
    vec4 OuterColor;
    float RadiusInner;
    float RadiusOuter;
};

void main() {
    float dx = TexCoord.x - 0.5;
    float dy = TexCoord.y - 0.5;
    float dist = sqrt(dx * dx + dy * dy);
    FragColor =
        mix( InnerColor, OuterColor,
            smoothstep( RadiusInner, RadiusOuter, dist ));
}
```

• Explanation of how uniform blocks are created and initialized can be found here:


*Additional reading:* *OpenGl 4.0 Shading Language Cookbook*, pages 37-42
33. The Perils of World space

- World space is a very useful intermediary between camera space and model space. It makes it easy to position cameras and so forth. But there is a lingering issue when dealing with world space directly. Namely, the problem of large worlds and numerical precision.

- Let us say that you're trying to model a very large area down to fairly small accuracy. Your units are centimeters, and you want precision to within 0.25 centimeters, otherwise speaking you'd like to be able to position vertices one from each other with a precision no less than \(\frac{1}{4}\) of a centimeter. And you want to cover an area with a radius of 1000 kilometers.

- And here comes the 32 bit floating point precision problem. The way this floating point number is represented in today's computers is that you have a higher floating point precision on small number and less precision on bigger numbers. Please have a look at the following output of a C++ program that shows the next 10 numbers that can be represented with 32 bit floating point precision after number 1000 and after number 32000000.

<table>
<thead>
<tr>
<th>Number</th>
<th>Represented Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000.00006104</td>
<td>32000002.00000000</td>
</tr>
<tr>
<td>1000.00012207</td>
<td>32000004.00000000</td>
</tr>
<tr>
<td>1000.00018311</td>
<td>32000006.00000000</td>
</tr>
<tr>
<td>1000.00024414</td>
<td>32000008.00000000</td>
</tr>
<tr>
<td>1000.00030518</td>
<td>32000010.00000000</td>
</tr>
<tr>
<td>1000.00036621</td>
<td>32000012.00000000</td>
</tr>
<tr>
<td>1000.00042725</td>
<td>32000014.00000000</td>
</tr>
<tr>
<td>1000.00048828</td>
<td>32000016.00000000</td>
</tr>
<tr>
<td>1000.00054932</td>
<td>32000018.00000000</td>
</tr>
<tr>
<td>1000.00061035</td>
<td>32000020.00000000</td>
</tr>
</tbody>
</table>
Now if these numbers represent your game centimeters, then you have no problems for objects located at 1000 cm (10 meters) away from your world origin, but you have serious problems for those objects located at 32000000 cm (320 km) and the problem is that at 320 km you can have 2 vertices be different only at intervals of 2 cm.

One solution, if you have access to powerful hardware capable of OpenGL 4.0 or better, is to use double-precision floating point values for your matrices and shader values. Double-precision floats, 64-bits in size, give you about 16 digits of precision, which is enough to measure the size of atoms in inches at more than 60 miles away from the origin.

However, you would be sacrificing a lot of performance to do this. Even though the hardware can do double-precision math, it loses quite a bit of performance in doing so (anywhere between 25% and 75% or more, depending on the GPU). And why bother, when the real solution is much easier? Let's look at our shader again.

```glsl
#version 330

layout(location = 0) in vec4 position;

uniform mat4 cameraToClipMatrix;
uniform mat4 worldToCameraMatrix;
uniform mat4 modelToWorldMatrix;

void main()
{
    vec4 worldPos = modelToWorldMatrix * position;
    vec4 cameraPos = worldToCameraMatrix * worldPos;
    gl_Position = cameraToClipMatrix * cameraPos;
}
```
• The **position** is relatively close to the origin, since model coordinates tend to be close to the model space origin. So you have plenty of floating-point precision there. The **cameraPos** value is also close to the origin. Remember, the camera in camera space is at the origin. The world-to-camera matrix simply transforms the world to the camera's position. So there's quite a bit of precision available in **cameraPos**.

• And in **gl_Position**, everything is in clip-space, which is again relative to the camera. The only precision problem is with **worldPos**. Or rather, in the **modelToWorldMatrix**.

• The **model to world transform** would have a massive translational component. We're moving from model space, which is close to the origin, to world-space which is far away. However, almost all of that will be immediately negated, because everything you're drawing is close to the camera. The camera matrix will have another massive translational component, since the camera is also far from the origin.

• This means that, if you combined the two matrices into one, you would have one matrix with a relatively small translation component. Therefore, you would not have a precision problem.

• Now, 32-bit floats on the CPU are no more precise than on the GPU. However, on the CPU you are guaranteed to be able to do double-precision math. And while it is slower than single-precision math, the CPU is not doing as many computations. You are not doing vector/matrix multiplies per vertex; you're doing them per object. And since the final result would actually fit within 32-bit precision limitations, the solution is obvious.

• The take-home point is this: avoid presenting OpenGL with an explicit model-to-world matrix. Instead, generate a matrix that goes straight from model space to camera space. You can use double-precision computations to do this if you need to; simply transform them back to single-precision when uploading the matrices to OpenGL.
• Still shocked about floating point precision? Run this small program at home and have a look at console output.

```c
int main(int argc, char* argv[])
{
    // 32 millions
    float f=32000000;
    // add 3.0
    f+=3.0f;
    // show result
    printf("%.2f",f);
    getchar();
    return 0;
}
```

Let's first define our task. Suppose we are inside a space shuttle. Inside the cabin we have the controls to rotate around X, Y and Z axes. Otherwise said, when controlling shuttle rotation we'd like to be able to rotate it around it's model space cardinal axes, like in the picture below:

The first thought is to store 3 angles which represent rotation around cardinal axes(XYZ), then create 3 rotation matrices based on these 3 angles, then multiply all these matrices together to produce a single rotation matrix that we will use to transform our vertices.

One of the first problems you will notice is that the order you apply these rotations matter. Each transform defines a new coordinate system, and the next transform is based on an object in the new space. For example, if we apply 90 degrees X rotation first, we have now changed what the axis for the subsequent Y rotation is. Before applying a rotation of 90 degrees around X axis the axis for Y rotation was 0,1,0, after X rotation by 90 degrees, that axis has changed to 0,0,-1 and a further Y rotation around 0,1,0 does not produce the desired result. Furthermore, it actually rotates your space ship around his Z axis and vice versa the rotation around the Z axis now rotates your ship around his Y axis.
• You can use any order that you like, so long as you understand what these angles mean. If you apply rotation around X axis first, your Y rotation must be in terms of the new X rotated coordinate system, and not the original XYZ coordinate system.

• **But there is a far more insidious problem lurking in there.** SAMPLE19 illustrates this problem. Because it was first diagnosed with a physical device called *gimbal*, the problem is known as *The Gimbal Lock*.

• A gimbal is a pivoted support that provides the ability to rotate in one axis. A gimbal can be mounted within another gimbal. SAMPLE19 has a set of 3 gimbals, each with a pivot axis that is perpendicular to the other two.

• You can control the orientation of each gimbal separately. 'xX' keys control the inner gimbal(rotation around X axis), 'yY' controll the middle gimbal(Y axis rotation) and 'zZ' control the outer gimbal(Z axis rotation). The worst happens when you align the two gimbals parallel with one another:
Recall that the purpose of the three gimbals is to be able to adjust one of the three angles and orient the object in a particular direction. In a flight-simulation game, the player would have controls that would change their Y, X and Z rotation. However, if in SAMPLE19 you first rotate around Y axis by 90 degrees you reach a Gimbal Lock situation that makes it impossible to rotate around ship's X axis anymore because the outer and middle gimbals are now rotating about the same axis.

A Gimbal lock is thus the loss of one degree of freedom that occurs when the axes of two of the three gimbals are driven into the same place and cannot compensate for rotations around one axis in three dimensional space.

Now, to the code. In this sample we store the orientation as 3 angles representing separate rotations around 3 cardinal axes. We than compute 3 rotation matrices from these angles:

```cpp
//calc 3 rotation matrices
xRotMatrix=glm::rotate(glm::mat4(1.0f),angX,glm::vec3(1,0,0));
yRotMatrix=glm::rotate(glm::mat4(1.0f),angY,glm::vec3(0,1,0));
zRotMatrix=glm::rotate(glm::mat4(1.0f),angZ,glm::vec3(0,0,1));
```

We finally draw our 'space ship' by combining these 3 matrices into a single one(we also scale our geometry a little using glm::scale() function) modelToWorldMatrix:

```cpp
void drawcube1()
{
    prog.setUniform("fragModulationColor", glm::vec4(1,1,1,1));
    modelToWorldMatrix = zRotMatrix*yRotMatrix*xRotMatrix*glm::scale(glm::mat4(1.0f),glm::vec3(1,1,3));
}```
setMatrices();

size_t colorData = sizeof(cubeVertexData) / 2;
glBindBuffer(GL_ARRAY_BUFFER, cubeVertexBufferObject);
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, (void*)colorData);
glDrawArrays(GL_TRIANGLES, 0, 36);
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, 0);

• We don't show the rest of SAMPLE19 code as it's something we already seen in our previous samples.

Additional reading:
3D Math Primer for Graphics and Game Development 1st Edition, pages 153-159
http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2008.html
• So gimbals, 3 accumulated axial rotations, do not really work very well for orienting an object. Gimbals can be locked, and it is very unintuitive to control them. How do we fix these problems?

• The first solution is to keep the orientation as a matrix not as 3 angles. When the time comes to modify the orientation, we simply apply a new transformation to this matrix, storing the result as the new current orientation.

• This means that every rotation around Y, Z and X axes applied to the current orientation will be relative to that current orientation. Which is precisely what we need. If the user applies a positive rotation around Y axis, you want that Y rotation to rotate them relative to where they are currently pointing, not relative to some fixed coordinate system.
Here's the code from this sample that shows how the orientation matrix is being updated every time the player presses the rotation keys:

```cpp
void keyboard(unsigned char key, int x, int y)
{
    glm::mat4 rotMatrix(1.0f);

    switch (key)
    {
    case 'x':
        rotMatrix=glm::rotate(glm::mat4(1.0f),1.0f,glm::vec3(1,0,0));
        orientationMatrix=orientationMatrix*rotMatrix;
        break;

    case 'X':
        rotMatrix=glm::rotate(glm::mat4(1.0f),-1.0f,glm::vec3(1,0,0));
        orientationMatrix=orientationMatrix*rotMatrix;
        break;

    case 'y':
        rotMatrix=glm::rotate(glm::mat4(1.0f),1.0f,glm::vec3(0,1,0));
        orientationMatrix=orientationMatrix*rotMatrix;
        break;

    case 'Y':
        rotMatrix=glm::rotate(glm::mat4(1.0f),-1.0f,glm::vec3(0,1,0));
        orientationMatrix=orientationMatrix*rotMatrix;
        break;
    }
}
```
case 'z':
    rotMatrix=glm::rotate(glm::mat4(1.0f),1.0f,glm::vec3(0,0,1));
    orientationMatrix=orientationMatrix*rotMatrix;
    break;

    case 'Z':
    rotMatrix=glm::rotate(glm::mat4(1.0f),-1.0f,glm::vec3(0,0,1));
    orientationMatrix=orientationMatrix*rotMatrix;
    break;
}

glutPostRedisplay();

• drawcube1() function is similar to the one from SAMPLE19, with the exception that we now the combined orientation matrix to orient our object:

void drawcube1()
{
    prog.setUniform("fragModulationColor",glm::vec4(1,1,1,1));
    modelToWorldMatrix = orientationMatrix*glm::scale(glm::mat4(1.0f),glm::vec3(1,1,3));
    setMatrices();

    size_t colorData = sizeof(cubeVertexData) / 2;
    glBindBuffer(GL_ARRAY_BUFFER, cubeVertexBufferObject);
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
However, there are a few downsides to this approach. First, a 4x4 matrix is rather larger than 3 floating-point angles. But a much more difficult issue is that successive floating-point math can lead to errors. If you keep accumulating successive transformations of an object, once every 1/30th of a second for a period of several minutes or hours, these floating-point errors start accumulating. Eventually, the orientation stops being a pure rotation and starts incorporating scale and skewing characteristics.

The solution here is to re-orthonormalize the matrix after applying each transform. A coordinate system (which a matrix defines) is said to be orthonormal if the basis vectors are of unit length (no scale) and each axis is perpendicular to all of the others.

Unfortunately, re-orthonormalizing a matrix is not a simple operation. You could try to normalize each of the axis vectors with typical vector normalization, but that would not ensure that the matrix was orthonormal. It would remove scaling, but the axes would not be guaranteed to be perpendicular.

Orthonormalization is certainly possible. But there are better solutions. Such as using something called a quaternion.

Additional reading:
3D Math Primer for Graphics and Game Development 1st Edition, pages 147-152
• A quaternion is (for the purposes of this conversation) a 4-dimensional vector that is treated in a special way. Any pure orientation change from one coordinate system to another can be represented by a rotation about some axis by some angle. A quaternion is a way of encoding this angle/axis rotation:

\[
\text{Quaternion} = \begin{bmatrix}
x \times \sin\left(\frac{\theta}{2}\right) \\
y \times \sin\left(\frac{\theta}{2}\right) \\
z \times \sin\left(\frac{\theta}{2}\right) \\
\cos\left(\frac{\theta}{2}\right)
\end{bmatrix}
\]

• Assuming the axis itself is a unit vector, this will produce a unit quaternion. That is, a quaternion with a length of 1.

• Quaternions can be considered to be two parts: a vector part and a scalar part. The vector part are the first three components, when displayed in the order above. The scalar part is the last part.
• **Quaternion Math.** Quaternions are equivalent to orientation matrices. You can compose two orientation quaternions using a special operation called **quaternion multiplication**. Given the quaternions $a$ and $b$, the product of them is:

\[
\begin{align*}
    a \cdot b &= \\
    &= \begin{bmatrix}
        a_w b_x + a_x b_w + a_y b_z - a_z b_y \\
        a_w b_y + a_y b_w + a_z b_x - a_x b_z \\
        a_w b_z + a_z b_w + a_x b_y - a_y b_x \\
        a_w b_w - a_x b_x - a_y b_y - a_z b_z
    \end{bmatrix}
\end{align*}
\]

• If the two quaternions being multiplied represent orientations, then the product of them is a composite orientation. This works like matrix multiplication, except only for orientations. Like matrix multiplication, quaternion multiplication is associative $((a \cdot b) \cdot c = a \cdot (b \cdot c))$, but not commutative ($a \cdot b \neq b \cdot a$).

• The main difference between matrices and quaternions that matters for our needs is that it is easy to keep a quaternion normalized. Simply perform a vector normalization on it after every few multiplications. This enables us to add numerous small rotations together without numerical precision problems showing up.

• There is one more thing we need to be able to do: convert a quaternion into a rotation matrix, because graphics pipelines still use matrices internally and not use quaternions.
• So our goal is to compose successive rotations into a final orientation. When we want to increase the rotation around Y axis, for example, we will take the current orientation in form of a quaternion and multiply into it a quaternion that represents a rotation around Y axis of a few degrees. The result becomes the new orientation.

• But which side do we do the multiplication on? Quaternion multiplication is not commutative, so this will have an affect on the output. Well, it works exactly like matrix math.

• SAMPLE19.2 does the same thing like SAMPLE19.1, only it uses quaternions to keep the orientation information (the code below shows how we react to keystrokes that we set for rotation):

```c++
void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
    case 'x':
        OffsetOrientation(glm::vec3(1.0f, 0.0f, 0.0f), 1.0f);
        break;

    case 'X':
        OffsetOrientation(glm::vec3(1.0f, 0.0f, 0.0f), -1.0f);
        break;
    }

    orientationMatrix=glm::mat4_cast(g_orientation);
    glutPostRedisplay();
}
```
• Though GLSL does not have quaternion types or quaternion arithmetic, the GLM math library provides both. The \texttt{g\_orientation} variable is of the type \texttt{glm\::fquat}, which is a floating-point quaternion. This variable keeps our orientation information in this sample. The \texttt{glm\::mat4\_cast} function converts a quaternion into a 4x4 rotation matrix. Thus, we still feed matrices(orientationMatrix) to the graphics pipeline, but we keeping the orientation information in a quaternion.

• In response to keypresses, \texttt{g\_orientation} is modified, applying a transform to it(by multiplying \texttt{g\_orientation} with a new quaternion). This is done with the \texttt{OffsetOrientation} function.

```cpp
void OffsetOrientation(\texttt{const glm\::vec3} &\_axis, \texttt{float} fAngDeg)
{
    \texttt{float} fAngRad = \texttt{Deg2Rad}(fAngDeg);

    \texttt{glm\::vec3} axis = \texttt{glm\::normalize}(_axis);

    axis = axis * \texttt{sinf}(fAngRad / 2.0f);
    \texttt{float} scalar = \texttt{cosf}(fAngRad / 2.0f);

    \texttt{glm\::fquat} offset(scalar, axis.x, axis.y, axis.z);

    \texttt{g\_orientation} = \texttt{g\_orientation} * offset;

    \texttt{g\_orientation} = \texttt{glm\::normalize}(\texttt{g\_orientation});
}
```

\textit{Additional reading:}
\url{http://www.arcsynthesis.org/gltut/Positioning/Tut08\%20Quaternions.html}
37. Quaternion interpolation

- The *raison d'être* of quaternions in 3D math is quaternion interpolation (interpolation between two orientations). This is something we cannot do with matrices. Linearly-interpolating the components of matrices does not create anything that resembles an inbetween transformation. However, linearly interpolating a pair of quaternions does. As long as you normalize the results.

- Quaternion interpolation is beyond the scope of this course. You can find more information on this here: 3D Math Primer for Graphics and Game Development 1st Edition, pages 159-179, http://www.arcsynthesis.org/gltut/Positioning/Tut08%20Interpolation.html
One of the most important aspects of rendering is lighting. Thus far, all of our objects have had a color that
is entirely part of the mesh data, pulled from a uniform variable, or computed in an arbitrary way. This
makes all of our objects look very flat and unrealistic.
• Lighting is complicated. Very complicated. The interaction between a surface and a light is mostly well understood in terms of the physics. But actually doing the computations for full light/surface interaction as it is currently understood by scientists is prohibitively expensive.

• As such, all lighting in any real-time application is some form of approximation of the real world. How accurate that approximation is generally determines how close to photorealism one gets. Photorealism is the ability to render a scene that is indistinguishable from a photograph of reality.

  **Non-Photorealistic Rendering**

• There are lighting models that do not attempt to model reality. These are, as a group, called non-photorealistic rendering (NPR) techniques. These lighting models and rendering techniques can attempt to model cartoon styles (typically called “cel shading”), paintbrush effects, pencil-sketch, or other similar things. NPR techniques including lighting models, but they also do other, non-lighting things, like drawing object silhouettes in an dark, ink-like color.

• Developing good NPR techniques is at least as difficult as developing good photorealistic lighting models. We'll have a look a later at an example of NPR lighting.

• A lighting model is an algorithm, a mathematical function, that determines how a surface interacts with light.

• The light we see can have one of two sources. A light emitting object like the sun or a lamp can emit light that is directly captured by our eyes. Or a surface can reflect light from another source that is captured by our eyes. Light emitting objects are called light sources.
The interaction between a light and a surface is the most important part of a lighting model. It is also the most difficult to get right. The way light interacts with atoms on a surface alone involves complicated quantum mechanical principles that are difficult to understand. This is made more complicated by the fact that light itself is not one thing. There is no such thing as “white light.” Virtually all light is made up of a number of different wavelengths. Each wavelength (in the visible spectrum) represents a color. White light is made of many wavelengths (colors) of light. Colored light simply has fewer wavelengths in it than pure white light.

Surfaces interact with light of different wavelengths in different ways. As a simplification of this complex interaction, we will assume that a surface can do one of two things: absorb that wavelength of light or reflect it.

For instance, a surface looks blue under white light because the surface absorbs all non-blue parts of the light and only reflects the blue parts. If one were to shine a red light on the surface, the surface would appear very dark, as the surface absorbs non-blue light, and the red light does not have much blue light in it.
Therefore, the apparent color of a surface is a combination of the absorbing characteristics of the surface (which wavelengths are absorbed or reflected) and the wavelengths of light shone upon that surface. The very first approximation which we'll make is that not all of these wavelengths matter. Instead of tracking millions of wavelengths in the visible spectrum, we will instead track 3. Red, green, and blue.

The RGB intensity of light reflected from a surface at a particular point is a combination of the RGB light absorbing characteristics of the surface at that point and the RGB light intensity shone on that point on the surface. All of these, the reflected light, the source light, and the surface absorption, can be described as RGB colors, on the range \([0, 1]\).

The intensity of light shone upon a surface depends on (at least) two things. First, it depends on the intensity of light that reaches the surface from a light source. And second, it depends on the angle between the surface and the light.

Consider a perfectly flat surface. If you shine a column of light with a known intensity directly onto that surface, the intensity of that light at each point under the surface will be a known value, based on the intensity of the light divided by the area projected on the surface (idea is illustrated in the figure below).
• If the light is shone instead at an angle, the area on the surface is much wider. This spreads the same light intensity over a larger area of the surface; as a result, each point under the light “sees” the light less intensely.

![Light at an Angle](image)

• Therefore, the intensity of the light cast upon a surface is a function of the original light's intensity and the angle between the surface and the light source. This angle is called the **angle of incidence of the light**.
39. Diffuse lighting (per-vertex, single point light source)  

The figure above shows the output of SAMPLE21 - a torus rendered with diffuse lighting technique.
• Diffuse lighting refers to a particular kind of light/surface interaction, where the light from the light source reflects from the surface at many angles, instead of as a perfect mirror (idea is illustrated in the figure below).

![](image)

• Incoming light strikes the surface and penetrates slightly before being re-radiated in all directions. Of course, the incoming light interacts with the surface before it is scattered, causing some wavelengths to be fully or partially absorbed and others to be scattered.

• For this tutorial, we will be using the **Lambertian reflectance** model of diffuse lighting. It represents the ideal case shown above, where light is reflected in all directions equally.
The mathematical model for Lambertian reflection involves two vectors: the direction from the surface point to the light source \( (s) \), and the normal vector at the surface point \( (n) \). The vectors are represented in the following diagram.

- The physics of the situation tells us that the amount of radiation (incoming light) that reaches a point on a surface is maximal when the light arrives along the direction of the normal vector, and zero when the light is perpendicular to the normal. In between, it is proportional to the cosine of the angle between the direction towards the light source and the normal vector.

- Remember that the vector dot product is represented in the following way:

\[
\overline{a} \cdot \overline{b} = \| \overline{a} \| \times \| \overline{b} \| \times \cos(\theta)
\]

- In the above equation if \( \overline{a} \) and \( \overline{b} \) are unit vectors then their dot product is the cosine of the angle between them. We thus can express the amount of radiation striking the surface as the product of the light intensity and the dot product of \( s \) and \( n \).

\[
L_d \ s \cdot n
\]

- Where \( L_d \) is the intensity of the light source, and the vectors \( s \) and \( n \) are assumed to be normalized.
As stated previously, some of the incoming light is absorbed before it is re-emitted. We can model this interaction by using a reflection coefficient ($K_d$), which represents the fraction of the incoming light that is scattered. This is sometimes referred to as the **diffuse reflectivity**, or the **diffuse reflection coefficient**. The diffuse reflectivity becomes a scaling factor for the incoming radiation, so the intensity of the outgoing light can be expressed as follows:

$$ L = L_d \ K_d \ s \cdot n $$

Because this model depends only on the direction towards the light source and the normal to the surface, not on the direction towards the viewer, we have a model that represents uniform (omnidirectional) scattering.

**Surface Orientation.** Now that we know what we need to compute, the question becomes where do we get the components, specifically the $n$ vector? Since our mesh geometry is made of triangles, and each triangle is made of vertices, we can assign to each vertex a value which will represent the the direction of the surface at that particular location. This vector is called a **surface normal**.
• In past tutorials, we have had a position and sometimes a color associated with a vertex. To that information, we add a normal. So we will need a vertex attribute that represents the normal.

• **Gouraud Shading.** So each vertex has a normal. That is useful, but it is not sufficient, for one simple reason. We do not draw the vertices of triangles; we draw the interior of a triangle through rasterization.

• There are several ways to go about computing lighting across the surface of a triangle. The simplest to code, and most efficient for rendering, is to perform the lighting computations at every vertex, and then let the result of this computation be interpolated across the surface of the triangle. This process is called Gouraud shading.

• Gouraud shading is a pretty decent approximation, when using the diffuse lighting model. It usually looks OK so long as we remain using that lighting model, and was commonly used for a good decade or so. Interpolation of vertex outputs is a very fast process, and not having to compute lighting at every fragment generated from the triangle raises the performance substantially.

• **Normals and space.** Normals are vector directions, so like position vectors, they exist in a certain coordinate system. The obvious way is to have the normals for your vertices be in the same coordinate system as the positions in those vertices. So that means model space.

• This also means that normals must be transformed from model space to the same space that the lighting direction is in; otherwise, the two vectors cannot be compared. For our purposes, we will use camera space. So that means that the light position should also be provided in camera space.
- SAMPLE21 overview:

  - We provide vertex positions of our torus at attribute with index 0
  - We provide the vertex normals at attribute with index 1
  - The OpenGL application provides to the shaders the following information via uniform variables:
    - The standard transformation matrices – modelview(model to camera space), combined model+view+projection(model to camera space to clip space)

      ```java
      prog.setUniform("ModelViewMatrix", mv);
      prog.setUniform("MVP", projection * mv);
      ```

    - The matrix to transforms normals from model to camera space(this will be explained a bit later)

      ```java
      prog.setUniform("NormalMatrix",
                      mat3( vec3(mv[0]), vec3(mv[1]), vec3(mv[2]) ));
      ```

    - The light position(transformed in camera space)

      ```java
      prog.setUniform("LightPosition", view * vec4(5.0f,5.0f,2.0f,1.0f) );
      ```

    - Kd, and Ld variables

      ```java
      prog.setUniform("Kd", 1.0f, 0.0f, 0.0f);
      prog.setUniform("Ld", 1.0f, 1.0f, 1.0f);
      ```
Here's our vertex shader:

```glsl
#version 330

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

uniform vec4 LightPosition; // Light position in eye coords.
uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ld;            // Diffuse light intensity
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

void main()
{
  vec3 tnorm = normalize( NormalMatrix * VertexNormal);
  vec4 cameraCoords = ModelViewMatrix * vec4(VertexPosition,1.0);
  vec3 s = normalize(vec3(LightPosition - cameraCoords));

  LightIntensity = Ld * Kd * max( dot( s, tnorm ), 0.0 );

  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```
• Let's analyze the vertex shader:

\[
\text{vec3 tnorm = normalize( NormalMatrix * VertexNormal);}
\]

• In the above line we first transform the normal vector using the normal matrix, normalize it, and store the result in tnorm. Note that the normalization here may not be necessary if your normal vectors are already normalized and the normal matrix does not do any scaling.

**NOTE:** The normal matrix is typically the inverse transpose of the upper-left 3x3 portion of the model-view matrix. We use the inverse transpose because normal vectors transform differently than the vertex position. For a more thorough discussion of the normal matrix, and the reasons why it exists, refer to the following link:


• The next step converts the vertex position to eye (camera) coordinates by transforming it via the model-view matrix (model to camera matrix). Then we compute the direction towards the light source by subtracting the vertex position from the light position and storing the result in s.

\[
\text{vec4 cameraCoords = ModelViewMatrix * vec4(VertexPosition,1.0);}
\]

\[
\text{vec3 s = normalize(vec3(LightPosition - cameraCoords));}
\]
• Next, we compute the scattered light intensity using the equation described above and store the result in the output variable **LightIntensity**. Note the use of the **max** function here. If the dot product is less than zero, then the angle between the normal vector and the light direction is greater than 90 degrees. This means that the incoming light is coming from inside the surface. Since such a situation is not physically possible (for a closed mesh), we use a value of 0.0. Using max function we clamp the angle in 0 – 90 degrees range.

\[
\text{LightIntensity} = \text{Ld} \times \text{Kd} \times \max( \dot{\text{s}}, \text{tnorm} ), 0.0); 
\]

• Finally, we convert the vertex position to clip coordinates by multiplying with the model-view projection matrix, (which is: projection * view * model) and store the result in the built-in output variable **gl_Position**.

\[
\text{gl\_Position} = \text{MVP} \times \text{vec4(\text{VertexPosition},1.0)}; 
\]

• Since LightIntensity is an output variable from the vertex shader, its value is interpolated across the face and passed into the fragment shader. The fragment shader then simply assigns the value to the output fragment. The algorithm is called Gouraud shading.

```
#version 330

in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```
• SAMPLE 21 output:

• Why the torus is red? Remember that we set the Ld coefficient (the RGB wavelengths intensities that reach our object) as follows: `prog.setUniform("Ld", 1.0f, 1.0f, 1.0f);` Which means that we have a full bright white light.

• And the Kd coefficient used was: `prog.setUniform("Kd", 1.0f, 0.0f, 0.0f);` Which means that the material our torus is made off, absorbs Blue and Green color totally while fully reflecting the Red color.

• Diffuse shading is a technique that models only a very limited range of surfaces. The dark areas may look a bit too dark. In fact, those areas that are not directly illuminated are completely black. In real scenes, there is typically some light that has been reflected about the room that brightens these surfaces. In the following samples, we'll look at ways to model more surface types, as well as provide some light for those dark parts of the surface.

Additional reading:
http://www.arcsynthesis.org/gltut/Illumination/Tutorial%2009.html
OpenGl 4.0 Shading Language Cookbook, pages 50-54
40. Implementing per-vertex Ambient, Diffuse and Specular (ADS) lighting

- One of the most commonly used (and fast) lighting models found in real time graphics, models the light-surface interaction as a combination of three components: ambient, diffuse, and specular. This model was described by Bui Tuong Phong and is also referred to as Phong reflection model or Phong shading model.

- The diffuse component was discussed in the previous sample. For convenience, we present the equation for diffuse lighting here again:

\[ I_d = L_d K_d (s \cdot n) \]

- The ambient component is a technique used to approximate global illumination. Consider for example the output of a program that implements diffuse lighting:

- The unlit portions of the cylinder are completely, 100% black. This almost never happens in real life, even for objects we perceive as being “black” in color.
• The light being cast in shadows from the sun comes from many places. Part of it is an atmospheric effect; the sun is so bright that the weakly reflective atmosphere reflects enough light to shine a color. Typically, this is a pale blue. Part of the light comes from other objects. The sun gives off so much light that the light reflected from other objects is bright enough to be a substantial contributor to the overall lighting in a scene.

• This phenomenon is called **interreflection**. A lighting model that handles interreflection is said to handle **global illumination**. It represents light that bounces from object to object before hitting the eyes of the person viewing the scene. Modelling only lighting directly from a light-emitting surface is called **local illumination** or **direct illumination**, and it is what we have been doing up until this point.

• The ambient lighting “model” is quite simple. It assumes that, on every object in the scene, there is a light of a certain intensity that emanates from everywhere. It comes from all directions equally, so there is no angle of incidence in our diffuse calculation. It can be modeled simply by multiplying the light source intensity (La) by the surface reflectivity (Ka).

\[ I_a = L_a K_a \]

• **Adding shinies.** The diffuse lighting model works reasonably well for a smooth, matte surface. Few objects in reality conform to this archetype. Therefore, in order to more accurately model real objects, we need to improve upon this. Let us focus on making objects appear shiny.

• Shiny materials tend to reflect light more strongly in the opposite direction from the angle of incidence (the angle between the surface normal and the incoming light direction). This kind of reflection is called a **specular reflection**. A perfect specular reflector would be a mirror.
The physics of the situation tells us that for perfect reflection, the angle of incidence is the same as the angle of reflection and that the vectors are coplanar with the surface normal, as shown in the following diagram:

In the preceding diagram, \( r \) represents the vector of pure-reflection corresponding to the incoming light vector \((-s)\), and \( n \) is the surface normal. We can compute \( r \) by using the following equation:

\[
r = -s + 2(s \cdot n)n
\]

However, the intensity of the reflected light depends not only on the angle of incidence but also on the direction to the viewer.
To model specular reflection, we need to compute the following (normalized) vectors: the direction towards the light source ($s$), the vector of perfect reflection ($r$), the vector towards the viewer ($v$), and the surface normal ($n$). These vectors are represented in the following image:

![Image showing vectors](image)

We would like the reflection to be maximal when the viewer is aligned with the vector $r$, and to fall off quickly as the viewer moves further away from alignment with $r$. This can be modeled using the cosine of the angle between $v$ and $r$ raised to some power ($f$). So the equation to compute the specular component is:

$$I_s = L_s K_s (r \cdot v)^f$$

The larger the power($f$), the faster the value drops towards zero as the angle between $v$ and $r$ increases. Again, similar to the other components, we also introduce a specular light intensity term ($L_s$) and surface reflectivity term ($K_s$).
• The specular component creates specular highlights (bright spots) that are typical of glossy surfaces. The larger the power of $f$ in the equation, the smaller the specular highlight and the shinier the surface appears. The value for $f$ is typically chosen to be somewhere between 1 and 200.

• Putting all of this together, we have the following shading equation:

$$I = I_a + I_d + I_s$$

$$= L_a K_a + L_d K_d (s \cdot n) + L_s K_s (r \cdot v)^f$$

• As in previous sample we will provide a normal at each vertex. Here's the vertex shader code:

```glsl
#version 330

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;       // Ambient light intensity
    vec3 Ld;       // Diffuse light intensity
    vec3 Ls;       // Specular light intensity
};
uniform LightInfo Light;
```
struct MaterialInfo {
    vec3 Ka;            // Ambient reflectivity
    vec3 Kd;            // Diffuse reflectivity
    vec3 Ks;            // Specular reflectivity
    float Shininess;    // Specular shininess factor
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

void main()
{
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix * vec4(VertexPosition,1.0);
    vec3 s = normalize(vec3(Light.Position - eyeCoords));
    vec3 v = normalize(-eyeCoords.xyz);
    vec3 r = reflect(-s, tnorm);
    float sDotN = max( dot(s,tnorm), 0.0 );
    vec3 ambient = Light.La * Material.Ka;
    vec3 diffuse = Light.Ld * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Ls * Material.Ks *
            pow( max( dot(r,v), 0.0 ), Material.Shininess );
    LightIntensity = ambient + diffuse + spec;
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
• Let's analyze the shader:

  • The shader begins by transforming the vertex normal into eye coordinates(camera coordinates) and normalizing, then storing the result in tnorm.

    ```
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    ```

  • The vertex position is then transformed into the same eye coordinates(camera coordinates) and stored in eyeCoords.

    ```
    vec4 eyeCoords = ModelViewMatrix * vec4(VertexPosition,1.0);
    ```

  • Next, we compute the normalized direction towards the light source (s). This is done by subtracting the vertex position in eye coordinates from the light position and normalizing the result. We'll use this vector in computing the diffuse and specular components:

    ```
    vec3 s = normalize(vec3(Light.Position - eyeCoords));
    ```

  • The direction from the vertex to the viewer (v) is the negation of the position (normalized) because in eye coordinates the viewer is at the origin. We use this vector in specular component computation:

    ```
    vec3 v = normalize(-eyeCoords.xyz);
    ```
• We compute the direction of pure reflection (r) by calling the GLSL built-in function `reflect`, which reflects the first argument about the second. We don't need to normalize the result because the two vectors involved are already normalized.

```glsl
vec3 r = reflect( -s, tnorm );
```

• The ambient component is computed and stored in the variable `ambient`.

```glsl
vec3 ambient = Light.La * Material.Ka;
```

• The dot product of s and n is computed next. As in the preceding recipe, we use the built-in function `max` to limit the range of values to between one and zero. The result is stored in the variable `sDotN`, and is used to compute the diffuse component. The resulting value for the diffuse component is stored in the variable `diffuse`.

```glsl
float sDotN = max( dot(s,tnorm), 0.0 );
vec3 diffuse = Light.Ld * Material.Kd * sDotN;
```

• Before computing the specular component, we check the value of `sDotN`. If `sDotN` is zero, then there is no light reaching the surface, so there is no point in computing the specular component, as its value must be zero. Otherwise, if `sDotN` is greater than zero, we compute the specular component using the equation presented earlier. Again, we use the built-in function `max` to limit the range of values of the dot product to between one and zero, and the function `pow` raises the dot product to the power of the Shininess exponent (corresponding to f in our lighting equation).

```glsl
vec3 spec = vec3(0.0);
if( sDotN > 0.0 )
spec = Light.Ls * Material.Ks * pow( max( dot(r,v), 0.0 ), Material.Shininess );
```
NOTE: If we did not check \( s \cdot \text{DotN} \) before computing the specular component, it is possible that some specular highlights could appear on faces that are facing away from the light source. This is clearly a non-realistic and undesirable result. Some people solve this problem by multiplying the specular component by the diffuse component, which would decrease the specular component substantially and alter its color. The solution presented here avoids this, at the cost of a branch statement (the if statement).

- The sum of the three components is then stored in the output variable \( \text{LightIntensity} \). This value will be associated with the vertex and passed down the pipeline. Before reaching the fragment shader, its value will be interpolated in a perspective correct manner across the face of the polygon.

\[
\text{LightIntensity} = \text{ambient} + \text{diffuse} + \text{spec};
\]

- Here's the output of SAMPLE22 (white bright spots represent the specular component):
• **Per-vertex vs. Per-fragment.** Since the shading equation is computed within the vertex shader, we refer to this as per-vertex lighting. One of the disadvantages of per-vertex lighting is that specular highlights can be warped or lost, due to the fact that the shading equation is not evaluated at each point across the face. For example, a specular highlight that should appear in the middle of a polygon might not appear at all when per-vertex lighting is used, because of the fact that the shading equation is only computed at the vertices where the specular component is near zero. We'll look later at an example of per-fragment shading.

*Additional reading:*

- OpenGL 4.0 Shading Language Cookbook, pages 55-61
41. Implementing two-sided shading SAMPLE23
When rendering a mesh that is completely closed, the back faces of polygons are hidden from the viewer so there is no reason to draw them at all. However, if a mesh contains holes, it might be the case that the back faces would become visible. For instance if we remove the lid from the teapot, we'd like to see it's interiors, but if the back-face culling is active we will get the following result.

What we want is to actually turn off back-face culling, so that triangles that are facing with their back to be rendered anyway, which means for the interior of the teapot to be rendered.

But because the normal of the back-face is pointing outwards the shading applied to that triangle during the lighting equation will be wrong. This is illustrated in the picture below:
• Because the normal for back-faces is pointing outwards of the interior of the teapot mesh, the lighting equation in the picture above does not light the interior enough.

• To properly shade those back faces, one needs to invert the normal vector and compute the lighting equations based on the inverted normal.
So, let's see how to modify the vertex shader to obtain the right result:

```glsl
#version 330

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 FrontColor;
out vec3 BackColor;

struct LightInfo { ... };
uniform LightInfo Light;

struct MaterialInfo {...};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

vec3 phongModel( vec4 position, vec3 normal ) {
    // The ADS shading calculations go here
    ...
}

void main()
{
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix * vec4(VertexPosition,1.0);
    FrontColor = phongModel( eyeCoords, tnorm );
    BackColor = phongModel( eyeCoords, -tnorm );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```
In the vertex shader, we compute the lighting equation using both the vertex normal and the inverted version, and pass each resultant color to the fragment shader.

```glsl
FrontColor = phongModel( eyeCoords, tnorm );
BackColor = phongModel( eyeCoords, -tnorm );
```

The vertex shader is a slightly modified version of the vertex shader presented in the recipe Implementing per-vertex ambient, diffuse, and specular (ADS) shading. The evaluation of the shading model is placed within a function named `phongModel`. The function is called twice, first using the normal vector (transformed into eye coordinates), and second using the inverted normal vector. The combined results are stored in `FrontColor` and `BackColor`, respectively.

And here's our fragment shader:

```glsl
#version 330

in vec3 FrontColor;
in vec3 BackColor;

layout( location = 0 ) out vec4 FragColor;

void main() {
    if( gl_FrontFacing ) {
        FragColor = vec4(FrontColor, 1.0);
    } else {
        FragColor = vec4(BackColor, 1.0);
    }
}
```
• In the fragment shader, we determine which color to apply based on the value of the built-in variable `gl_FrontFacing`. This is a Boolean value that indicates whether the fragment is part of a front or back facing polygon. Note that this determination is based on the winding of the polygon, and not the normal vector. (A polygon is said to have counter-clockwise winding if the vertices are specified in counter-clockwise order as viewed from the front side of the polygon.) By default when rendering, if the order of the vertices appear on the screen in a counter-clockwise order, it indicates a front facing polygon, however, we can change this by calling `glFrontFace` from the OpenGL program.

• **Using two-sided rendering for debugging.** It can sometimes be useful to visually determine which faces are front facing and which are back facing. For example, when working with arbitrary meshes, polygons may not be specified using the appropriate winding. As another example, when developing a mesh procedurally, it can sometimes be helpful to determine which faces are oriented in the proper direction in order to help with debugging.

• We can easily tweak our fragment shader to help us solve these kinds of problems by mixing a solid color with all back (or front) faces. For example, we could change the else clause within our fragment shader to the following:

```
FragColor = mix( vec4(BackColor,1.0),
vec4(1.0,0.0,0.0,1.0), 0.7 );
```

• This would mix a solid red color with all back faces, helping them to stand out, as shown in the following image. In the image, back faces are mixed with 70% red as shown in the preceding code.
Additional reading:
OpenGl 4.0 Shading Language Cookbook, pages 65-69
42. Implementing flat-shading

• Per-vertex shading involves computation of the shading model at each vertex and associating the result (a color) with that vertex. The colors are then interpolated across the face of the polygon to produce a smooth shading effect. This is also referred to as **Gouraud shading**.

• It is sometimes desirable to use a single color for each polygon so that there is no variation of color across the face of the polygon, causing each polygon to have a flat appearance. This can be useful in situations where the shape of the object warrants such a technique, perhaps because the faces really are intended to look flat, or to help visualize the locations of the polygons in a complex mesh. Using a single color for each polygon is commonly called **flat shading**. We have already learned about that (interpolation qualifiers0 in SAMPLE3. Let's revisit them once again.

• The images below show a mesh rendered with the ADS shading model. On the left, Gouraud shading is used. On the right, flat shading is used:
• In this sample we're using the same vertex shader as in SAMPLE22. The only change is:

```glsl
out vec3 LightIntensity;
to
flat out vec3 LightIntensity;
```

• The fragment shader has also only one change in comparison to SAMPLE22:

```glsl
in vec3 LightIntensity;
to
flat in vec3 LightIntensity;
```

• Flat shading is enabled by qualifying the vertex output variable (and its corresponding fragment input variable) with the **flat** qualifier. This qualifier indicates that no interpolation of the value is to be done before it reaches the fragment shader. The value presented to the fragment shader will be the one corresponding to the result of the invocation of the vertex shader for either the first or last vertex of the polygon. This vertex is called the **provoking vertex**, and can be configured using the OpenGL function `glProvokingVertex`. For example, the call:

```glsl
glProvokingVertex(GL_FIRST_VERTEX_CONVENTION);
```

• The above indicates that the first vertex should be used as the value for the flat shaded variable. The argument `GL_LAST_VERTEX_CONVENTION` indicates that the last vertex should be used.

*Additional reading:
* OpenGL 4.0 Shading Language Cookbook, pages 69-71
43. Using subroutines to select shader functionality

• In GLSL, a subroutine is a mechanism for binding a function call to one of a set of possible function
definitions based on the value of a variable. In many ways it is similar to function pointers in C.

• Subroutines therefore provide a way to select alternate implementations at runtime without swapping
shader programs and/or recompiling, or using if statements along with a uniform variable.

• For example, a single shader could be written to provide several shading algorithms intended for use on
different objects within the scene. When rendering the scene, rather than swapping shader programs (or
using a conditional statement), we can simply change the subroutine's uniform variable to choose the
appropriate shading algorithm as each object is rendered.

NOTE: Since performance is crucial in shader programs, avoiding a conditional statement or a shader swap can
be very valuable. With subroutines, we can implement the functionality of a conditional statement or shader
swap without the computational overhead.

• Here's how subroutines are used in shaders(The first step involves declaring the subroutine type)

```glsl
#version 400
subroutine vec3 shadeModelType( vec4 position, vec3 normal);
```

• After creating the new subroutine type, we declare a uniform variable of that type named shadeModel.

```glsl
subroutine uniform shadeModelType shadeModel;
```
The above variable serves as our function pointer and will be assigned to one of the two possible functions in the OpenGL application.

We declare two functions to be part of the subroutine by prefixing their definition with the subroutine qualifier:

```cpp
subroutine( shadeModelType )
vec3 phongModel( vec4 position, vec3 norm )
{
    // The ADS shading calculations go here (see: "Using functions in shaders," and "Implementing per-vertex ambient, diffuse and specular (ADS) shading")
    ...
}

subroutine( shadeModelType )
vec3 diffuseOnly( vec4 position, vec3 norm )
{
    vec3 s = normalize( vec3(Light.Position - position) );
    return Light.Ld * Material.Kd * max( dot(s, norm), 0.0 );
}
```
We call one of the two subroutine functions by utilizing the subroutine uniform `shadeModel` within the main function:

```cpp
void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;
    // Get the position and normal in eye space
    getEyeSpace(eyeNorm, eyePosition);

    // Evaluate the shading equation. This will call one of
    // the functions: diffuseOnly or phongModel.
    LightIntensity = shadeModel( eyePosition, eyeNorm );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

The call to `shadeModel( eyePosition, eyeNorm )` will be bound to one of the two functions depending on the value of the subroutine uniform `shadeModel`, which we will set within the OpenGL application.

Within the render function of the OpenGL application, we assign a value to the subroutine uniform with the following steps. First, we query for the index of each subroutine function using `glGetSubroutineIndex`. The first argument is the program handle. The second is the shader stage. In this case, the subroutine is defined within the vertex shader, so we use `GL_VERTEX_SHADER` here. The third argument is the name of the subroutine. We query for each function individually and store the indexes in the variables adsIndex and diffuseIndex.
To select the appropriate subroutine function, we need to set the value of the subroutine uniform shadeModel. To do so, we call `glUniformSubroutinesuiv`.

```gl
glUniformSubroutinesuiv( GL_VERTEX_SHADER, 1, &adsIndex);
... // Render the left teapot

.glUniformSubroutinesuiv( GL_VERTEX_SHADER, 1, &diffuseIndex);
... // Render the right teapot
```

- This function is designed for setting multiple subroutine uniforms at once. In our case, of course, we are setting only a single uniform. The first argument is the shader stage (GL_VERTEX_SHADER), the second is the number of uniforms being set, and the third is a pointer to an array of subroutine function indexes. Since we are setting a single uniform, we simply provide the address of the GLuint variable containing the index, rather than a true array of values.

*Additional reading:*
OpenGl 4.0 Shading Language Cookbook, pages 71-76
44. Discarding fragments to create a perforated look.
• Fragment shaders can make use of the `discard` keyword to "throw away" fragments. Use of this keyword causes the fragment shader to stop execution, without writing anything (including depth) to the output buffer.

• This provides a way to create **holes in polygons** without using blending. In fact, since fragments are completely discarded, there is no dependence on the order in which objects are drawn, saving us the trouble of doing any depth sorting that might have been necessary if blending was used.

• In this sample, we'll draw a teapot, and use the discard keyword to remove fragments selectively based on **texture coordinates**. Algorithmically we'll discard fragments in such a way to create a lattice like pattern:
• But what are texture coordinates? While we'll be looking in more details on texture coordinates in later samples, here's a small diagram that explains the ideas behind them:

![Diagram showing texture coordinates and their mapping to a triangle and pyramid.](image)
The vertex position, normal, and texture coordinates must be provided to the vertex shader from the OpenGL application. The position will be provided at location 0, the normal at location 1, and the texture coordinates at location 2. As in previous examples, the lighting parameters will be set from the OpenGL application via the appropriate uniform variables.

Here's how our vertex shader looks:

```cpp
#version 330

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;

out vec3 FrontColor;
out vec3 BackColor;
out vec2 TexCoord;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;       // Ambient light intensity
    vec3 Ld;       // Diffuse light intensity
    vec3 Ls;       // Specular light intensity
};

uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;            // Ambient reflectivity
    vec3 Kd;            // Diffuse reflectivity
    vec3 Ks;            // Specular reflectivity
    float Shininess;    // Specular shininess factor
};
```
uniform MaterialInfo Material;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

vec3 phongModel( vec4 position, vec3 norm )
{
    vec3 s = normalize(vec3(Light.Position - position));
    vec3 v = normalize(-position.xyz);
    vec3 r = reflect( -s, norm );
    vec3 ambient = Light.La * Material.Ka;
    float sDotN = max( dot(s,norm), 0.0 );
    vec3 diffuse = Light.Ld * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Ls * Material.Ks *
            pow( max( dot(r,v), 0.0 ), Material.Shininess );

    return ambient + diffuse + spec;
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;
    TexCoord = VertexTexCoord;
}
// Get the position and normal in eye space
getEyeSpace(eyeNorm, eyePosition);

FrontColor = phongModel( eyePosition, eyeNorm );
BackColor = phongModel( eyePosition, -eyeNorm );

gl_Position = MVP * vec4(VertexPosition, 1.0);

• And our fragment shader:

```glsl
#version 330
ing vec3 FrontColor;
ing vec3 BackColor;
ing vec2 TexCoord;
layout( location = 0 ) out vec4 FragColor;

void main() {
    const float scale = 15.0;

    bvec2 toDiscard = greaterThan( fract(TexCoord * scale), vec2(0.2, 0.2) );

    if( all(toDiscard) )
        discard;
    else {
        if( gl_FrontFacing )
            FragColor = vec4(FrontColor, 1.0);
        else
            FragColor = vec4(BackColor/6, 1.0);
    }
}
```
Now, let's explain the code.

Since we will be discarding some parts of the teapot, we will be able to see through the teapot to the other side. This will cause the back sides of some polygons to become visible. Therefore, we need to compute the lighting equation appropriately for both sides of each face. We'll use the same technique presented earlier in the two-sided shading sample.

The vertex shader is essentially the same as in the two-sided shading sample, with the main difference being the addition of the texture coordinate.

To manage the texture coordinate, we have an additional input variable, `VertexTexCoord`, that corresponds to attribute location 2. The value of this input variable is passed directly on to the fragment shader unchanged via the output variable `TexCoord`.

```
TexCoord = VertexTexCoord;
```

The ADS shading model is calculated twice, once using the given normal vector, storing the result in `FrontColor`, and again using the reversed normal, storing that result in `BackColor`. Both values are passed down the pipeline to the fragment shader.

```
FrontColor = phongModel( eyePosition, eyeNorm );
BackColor = phongModel( eyePosition, -eyeNorm );
```

In the fragment shader, we calculate whether or not the fragment should be discarded based on a simple technique designed to produce the lattice-like pattern.
• We first scale the texture coordinate by the arbitrary scaling factor \texttt{scale}. This corresponds to the number of lattice rectangles per unit (scaled) texture coordinate. We then compute the fractional part of each component of the scaled texture coordinate using the built-in function \texttt{fract}. Each component, then, is compared to 0.2 using the built-in function \texttt{greaterThan}, and the result is stored in the bool vector \texttt{toDiscard}. The \texttt{greaterThan} function compares the two vectors component-wise, and stores the Boolean results in the \texttt{toDiscard} boolean vector.

\begin{verbatim}
bvec2 toDiscard = greaterThan( fract(TexCoord * scale), vec2(0.2,0.2) );
\end{verbatim}

• If both components of the vector \texttt{toDiscard} are true, then the fragment lies within the inside of each lattice frame, and therefore we wish to discard this fragment. We can use the built-in function \texttt{all} to help with this check. The function \texttt{all} will return true if all of the components of the parameter vector are true. If the function returns true, we execute the discard statement to reject the fragment. In the else branch, we color the fragment based on the orientation of the polygon, as in the two-sided shading recipe presented earlier.

\begin{verbatim}
if( all(toDiscard) )
    discard;
else {
    if( gl_FrontFacing )
        FragColor = vec4(FrontColor, 1.0);
    else
        FragColor = vec4(BackColor/6, 1.0);
}
\end{verbatim}

\textit{Additional reading:}
\begin{itemize}
    \item OpenGl 4.0 Shading Language Cookbook, pages 76-80
\end{itemize}
45. Shading with multiple positional lights
• The goal of this sample is to add to our scene several light sources. We'll be rendering a pig mesh with 5 light sources of different colors.

   **Note:** This sample is the first to load the geometry from an OBJ file. OBJ (or .OBJ) is a geometry definition file format first developed by Wavefront Technologies for its Advanced Visualizer animation package. The file format is open and has been adopted by other 3D graphics application vendors. For the most part it is a universally accepted format.

• When shading with multiple light sources, we need to evaluate the shading equation for each light and sum the results to determine the total light intensity reflected by a surface location. The natural choice is to create uniform arrays to store the position and intensity of each light. We'll use an array of structures so that we can store the values for multiple lights within a single uniform variable.

• How it works. Within the vertex shader, the lighting parameters are stored in the uniform array “lights”. Each element of the array is a struct of type “LightInfo”. This example uses five lights. The light intensity is stored in the “Intensity” field, and the position in eye coordinates is stored in the “Position” field.

```cpp
struct LightInfo {
    vec4 Position;  // Light position in eye coords.
    vec3 Intensity; // Light intesity (amb., diff., and spec.)
};
uniform LightInfo lights[5];
```
The vertex shader function “ads” is responsible for computing the shading equation for a given light source. The index of the light is provided as the first parameter lightIndex. The equation is computed based on the values in the lights array at that index.

```plaintext
vec3 ads( int lightIndex, vec4 position, vec3 norm )
{
    vec3 s = normalize( vec3(lights[lightIndex].Position - position) );
    vec3 v = normalize(vec3(-position));
    vec3 r = reflect( -s, norm );
    vec3 I = lights[lightIndex].Intensity;
    return
        I * ( Ka +
            Kd * max( dot(s, norm), 0.0 ) +
            Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}
```

In the vertex shader “main” function, a “for” loop is used to compute the shading equation for each light, and the results are summed into the shader output variable Color.

```plaintext
void main()
{
    vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyePosition = ModelViewMatrix * vec4(VertexPosition,1.0);

    // Evaluate the lighting equation, for each light
    Color = vec3(0.0);
```
The fragment shader is not presented here as it simply applies the interpolated color to the fragment as we already seen in the previous samples.

In the OpenGL application we set the light intensities once, at program initialization:

```cpp
prog.setUniform("lights[0].Intensity", vec3(0.0f,0.8f,0.8f) );
prog.setUniform("lights[1].Intensity", vec3(0.0f,0.0f,0.8f) );
prog.setUniform("lights[2].Intensity", vec3(0.8f,0.0f,0.0f) );
prog.setUniform("lights[3].Intensity", vec3(0.0f,0.8f,0.0f) );
prog.setUniform("lights[4].Intensity", vec3(0.8f,0.8f,0.8f) );
```

And update the light positions at each frame by moving them in a circular manner:

```cpp
char name[20];
float x, z;
for( int i = 0; i < 5; i++ ) {
    sprintf(name,"lights[%d].Position", i);
    x = 2.0 * cos((TWOPI / 5) * i * ang);
    z = 2.0 * sin((TWOPI / 5) * i * ang);
    prog.setUniform(name, view * vec4(x, 1.2f, z + 1.0f, 1.0f) );
}
```
In this sample we also render a plane below the pig. To make the plane look a little different we transmit to the vertex shader different material properties. Note that we do this every frame before rendering the objects:

```cpp
prog.setUniform("Kd", 0.4f, 0.4f, 0.4f);
prog.setUniform("Ks", 0.9f, 0.9f, 0.9f);
prog.setUniform("Ka", 0.1f, 0.1f, 0.1f);
prog.setUniform("Shininess", 180.0f);
mesh->render();

prog.setUniform("Kd", 0.1f, 0.1f, 0.1f);
prog.setUniform("Ks", 0.9f, 0.9f, 0.9f);
prog.setUniform("Ka", 0.1f, 0.1f, 0.1f);
prog.setUniform("Shininess", 180.0f);
plane->render();
```

This is why the plane appears darker:

*Additional reading:*

*OpenGl 4.0 Shading Language Cookbook, pages 82-84*
46. Shading with a directional light source
• A core component of a shading equation is the vector that points from the surface location towards the light source ($s$ in previous examples). For lights that are extremely far away, there is very little variation in this vector over the surface of an object. In fact, for very distant light sources, the vector is essentially the same for all points on a surface. (Another way of thinking about this is that the light rays are nearly parallel.)

• Such a model would be appropriate for a distant, but powerful, light source such as the sun. Such a light source is commonly called a **directional light source** because it does not have a specific position, only a direction.

  NOTE: Of course, we are ignoring the fact that, in reality, the intensity of the light decreases with the square of the distance from the source. However, it is not uncommon to ignore this aspect for directional light sources.

• If we are using a directional light source, the direction towards the source is the same for all points in the scene. Therefore, we can increase the efficiency of our shading calculations because we no longer need to recompute the direction towards the light source for each location on the surface.

• In previous versions of OpenGL, the fourth component of the light position was used to determine whether or not a light was considered directional. A zero in the fourth component indicated that the light source was directional and the position was to be treated as a direction towards the source (a vector). Otherwise, the position was treated as the actual location of the light source. In this example, we'll emulate the same functionality.

• The shaders don't changed too much since our last examples, so we'll just show the function that implements the lighting equation inside the vertex shader:
Within the vertex shader, the fourth coordinate of the uniform variable “LightPosition” is used to determine whether or not the light is to be treated as a directional light. Inside the “ads” function, which is responsible for computing the shading equation, the value of the vector “s” is determined based on whether or not the fourth coordinate of “LightPosition” is zero. If the value is zero, “LightPosition” is normalized and used as the direction towards the light source. Otherwise, “LightPosition” is treated as a location in eye coordinates, and we compute the direction towards the light source by subtracting the vertex position from LightPosition and normalizing the result.
The advantages. There is a slight efficiency gain when using directional lights due to the fact that there is no need to re-compute the light direction for each vertex. This saves a subtraction operation, which is a small gain, but could accumulate when there are several lights, or when the lighting is computed per-fragment.

Additional reading:
OpenGl 4.0 Shading Language Cookbook, pages 82-87
32. Improving realism with per-fragment shading

• Per-vertex shading (Gouraud shading) that we have been doing until now is very fast but has quite a few visual limitations that make the image look unrealistic.

• For instance if you would render a large plane consisting of 4 vertices and place a very bright light source not far above this plane, you would expect to see a bright spot right under the light source. What you would actually see would be something like this:

• Where is the bright spot?
• What we are looking for is this:

Why in the first picture the bright spot was missing? Why it doesn't work, after all, geometrically the situation looks like this:
The surface normal for the areas directly under the light are almost the same as the direction towards the light. This means that the angle of incidence is small, so the cosine of this angle is close to 1. That should translate to having a bright area under the light, but darker areas farther away. What we see is nothing of the sort. Why is that?

Well, consider what we are doing. We are computing the lighting at every triangle's vertex, and then interpolating the results across the surface of the triangle. The ground plane is made up of precisely four vertices: the four corners. And those are all very far from the light position and have a very large angle of incidence. Since none of them have a small angle of incidence, none of the colors that are interpolated across the surface are bright.

This is not the only problem with doing per-vertex lighting. Other undesirable artifacts, such as edges of polygons, may also appear when Gouraud shading is used, due to the fact that color interpolation is less physically accurate. This is illustrated in the next picture:
• In the above picture, on the left, a low-tesselated sphere rendered with per-vertex shading and on the right, the same sphere rendered with per-fragment shading. Notice the edges of the polygon clearly visible in the picture on the left.

• To improve the accuracy of our results, we can move the computation of the shading equation from the vertex shader to the fragment shader. Instead of interpolating color across the polygon, we interpolate the position and normal vector, and use these values to evaluate the shading equation at each fragment. This technique is often called **Phong shading** or **Phong interpolation**. The results from Phong shading are much more accurate and provide more pleasing results.

• SAMPLE28 provides vertex position in attribute location zero, and the normal in location one. It also provide the values for the uniform variables $Ka$, $Kd$, $Ks$, Shininess, LightPosition, and LightIntensity, the first four of which are the standard material properties (reflectivities) of the ADS shading model. The latter two are the position of the light in eye coordinates, and the intensity of the light source, respectively. Finally, the application also provide the values for the uniforms ModelViewMatrix, NormalMatrix, ProjectionMatrix, and MVP.
Here's how the vertex shader now looks:

```cpp
#version 330
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix * vec4(VertexPosition,1.0) );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

The vertex shader has two output variables: `Position` and `Normal`. In the main function, we convert the vertex normal to eye coordinates by transforming with the normal matrix, and then store the converted value in `Normal`. Similarly, the vertex position is converted to eye coordinates by transforming it by the model-view matrix, and the converted value is stored in `Position`. 
The values of Position and Normal are automatically interpolated and provided to the fragment shader via the corresponding input variables. The fragment shader then computes the standard ADS shading equation using the values provided. The result is then stored in the output variable FragColor. The fragment shader is given below:

```glsl
#version 330

in vec3 Position;
in vec3 Normal;

uniform vec4 LightPosition;
uniform vec3 LightIntensity;

uniform vec3 Kd; // Diffuse reflectivity
uniform vec3 Ka; // Ambient reflectivity
uniform vec3 Ks; // Specular reflectivity
uniform float Shininess; // Specular shininess factor

layout( location = 0 ) out vec4 FragColor;

vec3 ads( )
{
    vec3 s = normalize( vec3(LightPosition) - Position );
    vec3 v = normalize(vec3(-Position));
    vec3 r = reflect( -s, Normal );

    return
    LightIntensity * ( Ka +
    Kd * max( dot(s, Normal), 0.0 ) +
```

In SAMPLE28 you can press the space bar to change from fragment to vertex shading. Here's the comparison:

- Notice how the bottom plane on the right picture, where the vertex shading is implemented is missing the correct illumination. In the same picture, notice the how the edges of triangles are visible.
• Evaluating the shading equation within the fragment shader produces more accurate renderings. However, the price we pay is in the evaluation of the shading model for each pixel of the polygon, rather than at each vertex. The good news is that with modern graphics cards, there may be enough processing power to evaluate all of the fragments for a polygon in parallel. This can essentially provide nearly equivalent performance for either per-fragment or per-vertex shading.

• Phong shading offers a more accurate visual result, but some undesirable artifacts may still appear. Please consult the following material for a more detailed overview of phong shading:
  http://www.arcsynthesis.org/gltut/Illumination/Tut10%20Fragment%20Lighting.html

Additional reading:
http://www.arcsynthesis.org/gltut/Illumination/Tut10%20Interpolation.html
http://www.arcsynthesis.org/gltut/Illumination/Tut10%20Fragment%20Lighting.html
OpenGl 4.0 Shading Language Cookbook, pages 88-91
• A halfway vector is widely used to give a slight performance increase when computing the specular component, especially when shading is computer per-fragment.

• As covered in the SAMPLE22, the specular term in the ADS shading equation involves the dot product of the vector of pure reflection \( (r) \), and the direction towards the viewer \( (v) \).

\[
I_s = L_s K_s (r \cdot v)^f
\]

• In order to evaluate the above equation, we need to find the vector of pure reflection \( (r) \), which is the reflection of the vector towards the light source \( (s) \) about the normal vector \( (n) \).

\[
r = -s + 2(s \cdot n)n
\]

This equation is implemented by the GLSL function: \texttt{reflect}. 
• The above equation requires a dot product, an addition, and a couple of multiplication operations. We can gain a slight improvement in the efficiency of the specular calculation by making use of the following observation. When \( v \) is aligned with \( r \), the normal vector \( (n) \) must be halfway between \( v \) and \( s \).

• Let's define the **halfway vector (h)** as the vector that is halfway between \( v \) and \( s \), where \( h \) is normalized after the addition:

\[
h = v + s
\]

• The following picture shows the relative positions of the halfway vector and the others.
• We could then replace the dot product in the equation for the specular component, with the dot product of \( h \) and \( n \).

\[
I_s = L_s K_s (h \cdot n)^f
\]

• Computing \( h \) requires fewer operations than it takes to compute \( r \), so we should expect some efficiency gain by using the halfway vector. The angle between the halfway vector and the normal vector is proportional to the angle between the vector of pure reflection \( (r) \) and the vector towards the viewer \( (v) \) when all vectors are coplanar. Therefore, we expect that the visual results will be similar, although not exactly the same.

• Here's how the function to compute lighting at each fragment now looks in the fragment shader:

```cpp
vec3 ads() {
    vec3 s = normalize(vec3(LightPosition) - Position);
    vec3 v = normalize(vec3(-Position));
    vec3 h = normalize(v + s);

    return LightIntensity * (Ka +
             Kd * max(dot(s, Normal), 0.0) +
             Ks * pow(max(dot(h,Normal), 0.0), Shininess));
}
```
• We compute the halfway vector by summing the direction towards the viewer (v), and the direction towards the light source (s), and normalizing the result. The value for the halfway vector is then stored in h.

• The specular calculation is then modified to use the dot product between h and the normal vector (Normal). The rest of the calculation is unchanged.

• The halfway vector provides a slight improvement in the efficiency of our specular calculation, and the visual results are quite similar. The following images show the teapot rendered using the halfway vector (right), versus the same rendering used in SAMPLE22. The halfway vector produces a larger specular highlight, but the visual impact is not substantially different. If desired, we could compensate for the difference in the size of the specular highlight by increasing the value of the exponent Shininess.

Addional reading:
OpenGl 4.0 Shading Language Cookbook, pages 91-93
48. Simulating a spotlight
• In this sample, we'll use a shader to implement a spotlight effect.

• A spotlight source is considered to be one that only radiates light within a cone, the apex of which is located at the light source. Additionally, the light is attenuated so that it is maximal along the axis of the cone and decreases towards the outside edges. This allows us to create light sources that have a similar visual effect to a real spotlight.

• The following image shows a teapot and a torus rendered with a single spotlight. Note the slight decrease in the intensity of the spotlight from the center towards the outside edge.
• The spotlight's cone is defined by a spotlight direction (\( \mathbf{d} \) in the figure below), a cutoff angle (\( \mathbf{c} \) in the figure below), and a position (\( \mathbf{P} \) in the figure below). The intensity of the spotlight is considered to be strongest along the axis of the cone, and decreases as you move towards the edges.

![Spotlight Diagram](image)

• Our sample will be based on SAMPLE29, which implements per-fragment lighting using the halfway vector.

• The vertex shader is basically the same as for SAMPLE29. Let's go ahead and see the fragment shader.
#version 330

in vec3 Position;
in vec3 Normal;

struct SpotLightInfo {
    vec4 position;   // Position in eye coords
    vec3 intensity;  // Direction of the spotlight in eye coords.
    vec3 direction;  // Angular attenuation exponent
    float exponent;  // Cutoff angle (between 0 and 90)
    float cutoff;    // Ambient reflectivity
};
uniform SpotLightInfo Spot;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

layout( location = 0 ) out vec4 FragColor;

vec3 adsWithSpotlight( ) {
    vec3 s = normalize( vec3( Spot.position) - Position );
    vec3 spotDir = normalize( Spot.direction);
    float angle = acos( dot(-s, spotDir) );
    float cutoff = radians( clamp( Spot.cutoff, 0.0, 90.0 ) );
    vec3 ambient = Spot.intensity * Ka;
    if( angle < cutoff ) {
        float spotFactor = pow( dot(-s, spotDir), Spot.exponent );
        // Other code...
    }
}

vec3 v = normalize(vec3(-Position));
vec3 h = normalize(v + s);

return ambient +
spotFactor * Spot.intensity * ( 
  Kd * max(dot(s, Normal), 0.0) +
  Ks * pow(max(dot(h, Normal), 0.0), Shininess)
);
} else {
  return ambient;
}

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

• Let's analyze it. The structure **SpotLightInfo** defines all of the configuration options for the spotlight.

struct SpotLightInfo {
  vec4 position;   // Position in eye coords
  vec3 intensity;  // Direction of the spotlight in eye coords.
  vec3 direction;  // Direction of the spotlight in eye coords.
  float exponent;  // Angular attenuation exponent
  float cutoff;    // Cutoff angle (between 0 and 90)
};

uniform SpotLightInfo Spot;
• We declare a single uniform variable named `Spot` to store the data for our spotlight. The **position** field defines the location of the spotlight in eye coordinates. It is set from within the OpenGL program as follows:

```cpp
double angle;
vec4 lightPos = vec4(10.0f * cos(angle), 10.0f, 10.0f * sin(angle), 1.0f);
prog.setUniform("Spot.position", view * lightPos);
```

• The **intensity** field is the intensity(ambient, diffuse, and specular) of the spotlight. If desired, you could break this into three variables.

```cpp
prog.setUniform("Spot.intensity", vec3(0.9f,0.9f,0.9f) );
```

• The **direction** field will contain the direction that the spotlight is pointing, which defines the center axis of the spotlight's cone. This vector should be specified in eye coordinates. Within the OpenGL program it should be transformed by the normal matrix in the same way that normal vectors would be transformed. We could do so within the shader; however, within the shader, the normal matrix would be specified for the object being rendered. This may not be the appropriate transform for the spotlight's direction.

```cpp
mat3 normalMatrix = mat3( vec3(view[0]), vec3(view[1]), vec3(view[2]) );
prog.setUniform("Spot.direction", normalMatrix * vec3(-lightPos) );
```

• The **exponent** field defines the exponent that is used when calculating the angular attenuation of the spotlight.

```cpp
prog.setUniform("Spot.exponent", 30.0f );
```
• The intensity of the spotlight is decreased in proportion to the cosine of the angle between the vector from the light to the surface location (the negation of the variable \( s \)) and the direction of the spotlight. That cosine term is then raised to the power of the variable \( \text{exponent} \). The larger the value of this variable, the faster the intensity of the spotlight is decreased. This is quite similar to the exponent in the specular shading term.

• The cutoff field defines the angle between the central axis and the outer edge of the spotlight's cone of light. We specify this angle in degrees, and clamp its value between 0 and 90 (clamping is done inside the fragment shader).

  \[
  \text{prog.setUniform("Spot.cutoff", 15.0f );}
  \]

• The function “adsWithSpotlight()” from the fragment shader computes the standard ambient, diffuse, and specular (ADS) shading equation, using a spotlight as the light source. The first line computes the vector from the surface location to the spotlight's position (\( s \)).

  \[
  \text{vec3 s = normalize( vec3( Spot.position ) - Position );}
  \]

• Next, the spotlight's direction is normalized and stored within \( \text{spotDir} \).

  \[
  \text{vec3 spotDir = normalize( Spot.direction );}
  \]

• The angle between \( \text{spotDir} \) and the negation of \( s \) is then computed and stored in the variable \( \text{angle} \).

  \[
  \text{float angle = acos( dot(-s, spotDir) );}
  \]
The variable `cutoff` stores the value of `Spot.cutoff` after it has been clamped between 0 and 90, and converted from degrees to radians.

```cpp
float cutoff = radians( clamp( Spot.cutoff, 0.0, 90.0 ) );
```

Next, the ambient lighting component is computed and stored in the variable `ambient`.

```cpp
vec3 ambient = Spot.intensity * Ka;
```

We then compare the value of the variable `angle` with that of the variable `cutoff`. If `angle` is less than `cutoff`, then the surface point is within the spotlight's cone. Otherwise the surface point only receives ambient light, so we return only the ambient component. If `angle` is less than `cutoff`, we compute the variable `spotFactor` by raising the dot product of `-s` and `spotDir` to the power of `Spot.exponent`. The value of `spotFactor` is used to scale the intensity of the light so that the light is maximal in the center of the cone, and decreases as you move towards the edges. Finally, the ADS shading equation is computed using the halfway vector technique(SAMPLE29), with the exception that the diffuse and specular terms are scaled by `spotFactor`.

```cpp
if( angle < cutoff ) {
    float spotFactor = pow( dot(-s, spotDir), Spot.exponent );
    vec3 v = normalize(vec3(-Position));
    vec3 h = normalize( v + s );
    return ambient +
        spotFactor * Spot.intensity * (  
            Kd * max( dot(s, Normal), 0.0 ) +  
            Ks * pow( max( dot(h,Normal), 0.0 ), Shininess ));
} else {
    return ambient;
}
```

Additional reading:
OpenGL 4.0 Shading Language Cookbook, pages 94-97
49. Creating a cartoon shading effect SAMPLE31
• Toon shading (also called Celshading) is a non-photorealistic technique that is intended to mimic the style of shading often used in hand-drawn animation. There are many different techniques that are used to produce this effect. In this sample, we'll use a very simple technique that involves a slight modification to the ambient and diffuse shading model.

• The basic effect is to have large areas of constant color with sharp transitions between them. This simulates the way that an artist might shade an object using strokes of a pen or brush.

• The technique presented in this sample involves computing only the ambient and diffuse components of the typical ADS shading model, and quantizing (To quantize == to limit the possible values of a magnitude or quantity to a discrete set of values) the cosine term of the diffuse component. In other words, the value of the dot product normally used in the diffuse term is restricted to a fixed number of possible values. The following table illustrates the concept for four levels:

<table>
<thead>
<tr>
<th>Cosine of the Angle between s and n</th>
<th>Value used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between 1 and 0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>Between 0.75 and 0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Between 0.5 and 0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>Between 0.25 and 0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

• In the preceding table, s is the vector towards the light source and n is the normal vector at the surface. By restricting the value of the cosine term in this way, the shading displays strong discontinuities from one level to another (see the preceding image), simulating the pen strokes of hand-drawn cel animation.
The fragment shader is given below:

```glsl
#version 330
in vec3 Position;
in vec3 Normal;

struct LightInfo {
   vec4 position;
   vec3 intensity;
};
uniform LightInfo Light; // Diffuse reflectivity
uniform vec3 Ka; // Ambient reflectivity

const int levels = 3;
const float scaleFactor = 1.0 / levels;
layout( location = 0 ) out vec4 FragColor;

vec3 toonShade() {
   vec3 s = normalize( Light.position.xyz - Position.xyz );
   vec3 ambient = Ka;
   float cosine = max(dot( s, Normal ),0.0f);
   vec3 diffuse = Kd * floor( cosine * levels ) * scaleFactor;

   return Light.intensity * (ambient + diffuse);
}

void main() {
   FragColor = vec4(toonShade(), 1.0);
}
```
• How it works: the constant variable **levels** defines how many distinct values will be used in the diffuse calculation. This could also be defined as a uniform variable to allow for configuration from the main OpenGL application. We will use this variable to quantize the value of the cosine term in the diffuse calculation.

```c
const int levels = 3;
const float scaleFactor = 1.0 / levels;
```

• The function “toonShade()” is the most significant part of this shader. We start by computing \( s \), the vector towards the light source.

```c
vec3 s = normalize( Light.position.xyz - Position.xyz );
```

• Next, we compute the cosine term of the diffuse component by evaluating the dot product of \( s \) and \( \text{Normal} \).

```c
float cosine = max(dot( s, Normal ),0.0f);
```

• The next line quantizes that value in the following way. Since the two vectors are normalized, and we have removed negative values with the “max()” function, we are sure that the value of \( \text{cosine} \) is between zero and one. By multiplying this value(\( \text{cosine} \)) by \( \text{levels} \) and taking the `floor()`(the “floor()” function maps a real number to the largest previous integer), the result will be an integer between 0 and \( \text{levels–1} \). When we divide that value by \( \text{levels} \) (by actually multiplying by \( \text{scaleFactor} \)), we scale these integral values to be between zero and one again. The result is a value that can be one of \( \text{levels} \) possible values spaced between zero and one. This result is then multiplied by \( \text{Kd} \), the diffuse reflectivity term.

```c
vec3 diffuse = Kd * floor( cosine * levels ) * scaleFactor;
```
• Finally, we combine the diffuse and ambient components together to get the final color for the fragment.

```java
return Light.intensity * (ambient + diffuse);
```

• When quantizing the **cosine** term, we could have used “`ceil()`” instead of “`floor()`”. Doing so would have simply shifted each of the possible values up by one level. This would make the levels of shading slightly brighter. The “`ceil()`” function maps a real number to the smallest following integer.

• The typical cartoon style seen in most cel animation includes black outlines around the silhouettes and along other edges of a shape. The shading model presented here does not produce those black outlines. There are several techniques for producing them, and we'll look at one of them later.

*Additional reading:*
OpenGL 4.0 Shading Language Cookbook, pages 97–100
50. Simulating fog SAMPLE32
A simple fog effect can be achieved by mixing the color of each fragment with a constant fog color. The amount of influence of the fog color is determined by the **distance from the camera**. We could use either a **linear relationship** between the distance and the amount of fog color, or we could use a non-linear relationship such as an **exponential** one. The following image shows four teapots rendered with a fog effect produced by mixing the fog color in a linear relationship with distance:

![Teapots rendered with fog effect](image)

- To define this linear relationship we can use the following equation:

$$f = \frac{d_{\text{max}} - |z|}{d_{\text{max}} - d_{\text{min}}}$$
In the preceding equation, \( d_{\text{min}} \) is the distance from the eye where the fog is minimal (no fog contribution), and \( d_{\text{max}} \) is the distance where the fog color obscures all other colors in the scene. The variable \( z \) represents the distance from the eye of the fragment. The value \( f \) is the fog factor. A fog factor of zero represents 100% fog, and a factor of one represents no fog. Since fog typically looks thickest at large distances, the fog factor is minimal (more fog effect) when \(|z|\) is equal to \( d_{\text{max}} \), and maximal (less fog effect) when \(|z|\) is equal to \( d_{\text{min}} \).

• NOTE: Since the fog is applied by the fragment shader, the effect will only be visible on the objects that are rendered. It will not appear on any "empty" space in the scene (the background). To help make the fog effect consistent, you should use a background color that matches the maximum fog color.

• So let's see the fragment shader:

```glsl
#version 330

in vec3 Position;
in vec3 Normal;

struct LightInfo {
  vec4 position;
  vec3 intensity;
};
uniform LightInfo Light;

struct FogInfo {
  float maxDist;
  float minDist;
  vec3 color;
};
```

float dist = abs(Position.z);
float fogFactor = (Fog.maxDist - dist) / (Fog.maxDist - Fog.minDist);
fogFactor = clamp(fogFactor, 0.0, 1.0);
vec3 shadeColor = ads();
vec3 color = mix(Fog.color, shadeColor, fogFactor);
FragColor = vec4(color, 1.0);
The uniform variable **Fog** contains the parameters that define the extent and **color** of the fog.

```c
struct FogInfo {
    float maxDist;
    float minDist;
    vec3 color;
};

uniform FogInfo Fog;
```

- The field **minDist** is the distance from the eye to the fog's starting point, and **maxDist** is the distance to the point where the fog is maximal. The field **color** is the color of the fog.

- Now, to the main() function. The variable **dist** is used to store the distance from the surface point to the eye position (camera origin). The z coordinate of the camera space position is used as an estimate of the actual distance.

```c
float dist = abs( Position.z );
```

- The variable **fogFactor** is computed using the preceding equation. Since dist may not be between **minDist** and **maxDist**, we clamp the value of **fogFactor** to be between zero and one.

```c
float fogFactor = (Fog.maxDist - dist) / (Fog.maxDist - Fog.minDist);
fogFactor = clamp( fogFactor, 0.0, 1.0 );
```

- We then call the function “ads()” to evaluate the basic ADS shading model. The result of this is stored in the variable **shadeColor**.

```c
vec3 shadeColor = ads();
```
• Finally, we mix shadeColor and Fog.color together based on the value of fogFactor, and the result is used as the fragment color.

• In the above code, we used the absolute value of the z coordinate as the distance from the camera. This may cause the fog to look a bit unrealistic in certain situations. To compute a more precise distance, we could replace the line:

```c
float dist = abs(Position.z);
```

with the following:

```c
float dist = length(Position.xyz);
```

• Of course, the latter version requires a square root, and therefore would be a bit slower in practice.

• **Non-linear fog.** In this recipe we used a linear relationship between the amount of fog color and the distance from the eye. Another choice would be to use an exponential relationship. For example, the following equation could be used:

\[
f = e^{-d|z|}
\]

• In the above equation, d represents the density of the fog. Larger values would create "thicker" fog. We could also square the exponent to create a slightly different relationship (a faster increase in the fog with distance).

\[
f = e^{-(dz)^2}
\]

*Additional reading:*
OpenGL 4.0 Shading Language Cookbook, pages 100-104
51. Texturing

Right image – lighting only, left image – lighting + texturing
Textures are an important and fundamental aspect of real-time rendering in general, and OpenGL in particular. The use of textures within a shader opens up a huge range of possibilities. Beyond just using textures as sources of color information (like in the picture above), they can be used for things like additional sources of data (such as depth information), shading parameters, displacement maps, normal vectors, or other vertex data.

At their heart, however, a texture is a look-up table; an array. There is a lot of minutiae (bullshit) about accessing them, but at their core a texture is just a large array of some dimensionality that you can access from a shader.

Perhaps the most common misconception about textures is that textures are pictures: images of skin, rock, or something else that you can look at in an image editor. While it is true that many textures are pictures of something, it would be wrong to limit your thoughts in terms of textures to just being pictures. Sadly, this way of thinking about textures is reinforced by OpenGL; data in textures are “colors” and many functions dealing with textures have the word “image” somewhere in them.

The best way to avoid this kind of thinking is to have our first textures be of those non-picture types of textures. So as our introduction to the world of textures, we will rewrite a little the cartoon shading sample to produce the same result using the “texture” concept.
Creating a cartoon shading effect v.2.0

SAMPLE31.1
Let's remember how we implemented the cartoon shading computation in the first example:

```cpp
const int levels = 3;
const float scaleFactor = 1.0 / levels;

vec3 toonShade() {
    vec3 s = normalize(Light.position.xyz - Position.xyz);
    vec3 ambient = Ka;
    float cosine = max(dot(s, Normal), 0.0f);
    vec3 diffuse = Kd * floor(cosine * levels) * scaleFactor;
    return Light.intensity * (ambient + diffuse);
}

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

We algorithmically forced the value of the cosine of the angle between the normal vector to the surface and the vector to the light source (the angle of incidence) to be one of “x” values. We did this with the help of the “floor()” function and a few mathematical tricks.

In this sample we'll save the possible values of the angle of incidence inside a “look-up” table (an array) and then access this array inside our shader.
So how do we get a look-up table to the shader? We could use the obvious method; build a uniform buffer containing an array of floats. We would multiply the dot-product by the number of entries in the table and pick a table entry based on that value. By now, you should be able to code this.

But lets say that we want another alternative; what else can we do? We can put our look-up table in a texture.

A **texture** is an object that contains one or more arrays of data, with all of the arrays having some dimensionality. The storage for a texture is owned by OpenGL and the GPU, much like they own the storage for buffer objects. Textures can be accessed in a shader, which fetches data from the texture at a specific location within the texture's arrays.

The arrays within a texture are called **images**; this is a legacy term, but it is what they are called. Textures have a **texture type**; this defines characteristics of the texture as a whole, like the number of dimensions of the images and a few other special things.

In order to understand how textures work, let's follow the data from our initial generation of the lookup tables to how the GLSL shader accesses them in SAMPLE31.1. The function “BuildCartoonData()” from our C++ program, generates the data that we want to put into our OpenGL texture.

```cpp
void BuildCartoonData(std::vector<GLubyte> &textureData, int levels)
{
    textureData.resize(levels);
    std::vector<GLubyte>::iterator currIt = textureData.begin();
```
for(int iLevel = 0; iLevel < levels; iLevel++)
{
    float angle = 1.0f - 1.0f / ((float)iLevel + 1.0f);
    *currIt++ = (GLubyte)(angle * 255.0f);
}

• The above function fills a std::vector with bytes that represents our lookup table. It's a pretty simple function. The parameter levels specifies the number of entries in the table. As we iterate over the range, we add to this lookup table the values that we would like our cosine of our angle of incidence to be restricted to.

• However, the result of this computation is a float(angle variable), not a GLubyte. Yet our array contains bytes. It is here that we must introduce a new concept widely used with textures: normalized integers.

• A normalized integer is a way of storing floating-point values on the range [0, 1] in far fewer than the 32-bytes it takes for a regular float. The idea is to take the full range of the integer and map it to the [0, 1] range. The full range of an unsigned integer is [0, 255]. So to map it to a floating-point range of [0, 1], we simply divide the value by 255.

• The above code takes the angle and converts it into a normalized integer. This saves a lot of memory. By using normalized integers in our texture, we save 4x the memory over a floating-point texture. When it comes to textures, oftentimes saving memory improves performance. And since this is supposed to be a performance optimization over shader computations, it makes sense to use a normalized integer value.
The function “CreateCartoonTexture()” calls “BuildCartoonData()” to generate the array of normalized integers. The rest of that function uses the array to build the OpenGL texture object:

```cpp
GLuint CreateCartoonTexture(int levels)
{
    std::vector<GLubyte> textureData;
    BuildCartoonData(textureData, levels);

    GLuint gaussTexture;
    glGenTextures(1, &gaussTexture);
    glBindTexture(GL_TEXTURE_1D, gaussTexture);
    glTexImage1D(GL_TEXTURE_1D, 0, GL_R8, levels, 0, GL_RED,
                 GL_UNSIGNED_BYTE, &textureData[0]);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_BASE_LEVEL, 0);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAX_LEVEL, 0);
    glBindTexture(GL_TEXTURE_1D, 0);
    return gaussTexture;
}
```

The `glGenTextures()` function creates a single texture object, similar to other `glGen*` functions we have seen. `glBindTexture()` attaches the texture object to the context. The first parameter specifies the texture's type. `GL_TEXTURE_1D` means that the texture contains one-dimensional images.

The next function, `glTexImage1D()` is how we allocate storage for the texture and pass data to the texture. It is similar to `glBufferData()`, though it has many more parameters. Let's enumerate them:
• The first(GL_TEXTURE_1D) specifies the type of the currently bound texture.
• The second parameter is something we will(hopefully) talk about in later samples.
• The third(GL_R8) parameter is the format that OpenGL will use to store the texture's data.
• The fourth parameter(levels) is the width of the image, which corresponds to the length of our lookup table.
• The fifth parameter must always be 0; it represents an old feature no longer supported.

The next three parameters tell OpenGL how to read the texture data from our array(how the data is stored in our array).

• The sixth parameter(GL_RED) says that we are uploading a single component to the texture, namely the red component.
• The parameter GL_UNSIGNED_BYTE says that each component that we are uploading is stored in an 8-bit unsigned byte.
• The last parameter is the pointer to our data

• So by calling `glTexImage1D(GL_TEXTURE_1D, 0, GL_R8, levels, 0, GL_RED, GL_UNSIGNED_BYTE, &textureData[0])` we told OpenGL to create a 1D texture with "levels" texels, where each texel has only one component, namely the red one(GL_R8) with a size of 8 bits. We also told OpenGL that we will be uploading a single component – GL_RED, where each component will be of type GL_UNSIGNED_BYTE and all this data should be read from &textureData[0].

• A question that could arise is that the GL_R8 duplicates the GL_RED and GL_UNSIGNED_BYTE information. Why is that? Because GL_R8 defines how the user expects the OpenGL texture to be laid out(one component – red, 8 bits per component), but GL_RED and GL_UNSIGNED_BYTE specifies the
format of the user input data. All this means that the user input data and the OpenGL texture stored data can be different, and when they are, OpenGL will invoke a **pixel transfer** operation to convert between formats.

- Note that GL_R8 and GL_RED/GL_UNSIGNED_BYTE combination perfectly match each other in our case. We tell OpenGL to make the texture store unsigned normalized 8-bit integers, and we provide unsigned normalized 8-bit integers as the input data.

- This is not strictly necessary. We could have used GL_R16 as our format instead. OpenGL would have created a texture that contained 16-bit unsigned normalized integers. OpenGL would then have had to convert our input data to the 16-bit format. It is good practice to try to match the texture's format with the format of the data that you upload to OpenGL.

**NOTE:** GL_R8 demistified. The “R” represents the components that are stored in texel. Namely, the “red” component. Since textures used to always represent image data, the components are named after components of a color vec4. Each component takes up “8” bits. The suffix of the format represents the data type. Since unsigned normalized values are so common, they get the “no suffix” suffix; all other data types have a specific suffix. Float formats use “f”; a red, 32-bit float internal format would use GL_R32F.

- The calls to `glTexParameter()` set parameters on the texture object. These parameters define certain properties of the texture. Exactly what these parameters are doing is something that will be discussed in next samples.

**NOTE:** A **texel**, or **texture element** (also texture pixel) is the fundamental unit of texture space. Textures are represented by arrays of texels, just as pictures are represented by arrays of pixels.
Textures in Shaders

- OK, so we have a texture object, which has a texture type. We need some way to represent that texture in GLSL. This is done with something called a **GLSL sampler**. Samplers are special types in OpenGL; they represent a texture that has been bound to the OpenGL context. For every OpenGL texture type, there is a corresponding sampler type. So a texture that is of type GL_TEXTURE_1D is paired with a sampler of type sampler1D.

- The GLSL sampler type is very unusual. Indeed, it is probably best if you do not think of it like a normal basic type. Think of it instead as a specific hook into the shader that the user can use to supply a texture.

- In our fragment shader file we create the sampler in the following way:

```glsl
uniform sampler1D cartoonTexture;
```

  This creates a sampler for a 1D texture type; the user cannot use any other type of texture with this sampler.

- The process of fetching data from a texture, at a particular location, is called **sampling**. Let's have a look on how this is done in our fragment shader:

```glsl
texture3 toonShade( )
{
    vec3 s = normalize( Light.position.xyz - Position.xyz );
    vec3 ambient = Ka;
    float texCoord = dot( s, Normal );
    ...}
```
The bold font in the code above denotes the place where the texture is accessed. The function `texture()`, accesses the texture denoted by the first parameter (the sampler to fetch from). It accesses the value of the texture from the location specified by the second parameter. This second parameter, the location to fetch from, is called the **texture coordinate**. Since our texture has only one dimension, our texture coordinate also has one dimension.

The texture function for 1D textures expects the texture coordinate to be normalized. This means something similar to normalizing integer values. A normalized texture coordinate is a texture coordinate where the coordinate values range from \([0, 1]\) refer to texel coordinates (the coordinates of the pixels within the textures) to \([0, \text{texture-size}]\).

What this means is that our texture coordinates do not have to care how big the texture is. We can change the texture's size without changing anything about how we compute the texture coordinate. A coordinate of 0.5 will always mean the middle of the texture, regardless of the size of that texture.

A texture coordinate values outside of the \([0, 1]\) range must still map to a location on the texture. What happens to such coordinates depends on values set in OpenGL that we will see later.
• The return value of the `texture()` function is a `vec4`, regardless of the image format of the texture. So even though our texture's format is GL_R8, meaning that it holds only one channel of data, we still get four in the shader. The other three components are 0, 0, and 1, respectively.

• We get floating-point data back because our sampler is a floating-point sampler. Samplers use the same prefixes as vec types. A `ivec4` represents a vector of 4 integers, while a `vec4` represents a vector of 4 floats. Thus, an `isampler1D` represents a texture that returns integers, while a `sampler1D` is a texture that returns floats. Recall that 8-bit normalized unsigned integers are just a cheap way to store floats, so this matches everything correctly.

**Texture Binding**

• We have a texture object, an OpenGL object that holds our image data with a specific format. We have a shader that contains a sampler uniform that represents a texture being accessed by our shader. How do we associate a texture object with a sampler in the shader?

• The OpenGL context has an array of slots called `texture image units`, also known as `image units` or `texture units`. Each image unit represents a single texture. A sampler uniform in a shader is set to a particular image unit; this sets the association between the shader and the image unit. To associate an image unit with a texture object, we bind the texture to that unit.
• The first step is to bind our glsl sampler to one of texture units. For this example we'll use the first texture unit which has the index 0.

```cpp
GLuint cartoonTextureUnif = glGetUniformLocation(prog.getHandle(), "cartoonTexture");
glUniform1i(cartoonTextureUnif, 0);
```

• After that, we have to bind the texture object to the same texture unit as above(0).

```cpp
GLuint cartoonTextureHandle = CreateCartoonTexture(3);
glActiveTexture(GL_TEXTURE0 + 0);
glBindTexture(GL_TEXTURE_1D, cartoonTextureHandle);
```

• The `glActiveTexture()` function changes the current texture unit. All subsequent texture operations, whether `glBindTexture()`, `glTexImage()`, `glTexParameter()`, etc, affect the texture bound to the current texture unit.

• Also note the peculiar `glActiveTexture` syntax for specifying the image unit: `GL_TEXTURE0 + 0`. This is the correct way to specify which texture unit, because `glActiveTexture` is defined in terms of an enumerator rather than integer texture image units.

• Texture binding can be summarized in the following diagram:
Sampler Objects

• With the association between a texture and a program's sampler uniform made, there is still one thing we need before we render. There are a number of parameters the user can set that affects how texture data is fetched from the texture.
• In our case, we want to make sure that the shader cannot access texels outside of the range of the texture. If the shader tries, we want the shader to get the nearest texel to our value. So if the shader passes a texture coordinate of "-0.3", we want them to get the same texel as if they passed 0.0. In short, we want to clamp the texture coordinate to the range of the texture.

• These kinds of settings are controlled by an OpenGL object called a **sampler object**. The code that creates a sampler object for our textures is given below:

```c
GLuint cartoonSampler;
glGenSamplers(1, &cartoonSampler);
glSamplerParameteri(cartoonSampler, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glSamplerParameteri(cartoonSampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glSamplerParameteri(cartoonSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

• As with most OpenGL objects, we create a sampler object with `glGenSamplers()`

• In this above code, we set three parameters. The first two parameters are things we will discuss in the next sample. The third parameter, GL_TEXTURE_WRAP_S, is how we tell OpenGL that texture coordinates should be clamped to the range of the texture.

• OpenGL names the components of the texture coordinate “strq” rather than “xyzw” or “uvw” as is common. Indeed, OpenGL has two different names for the components: “strq” is used in the main API, but “stpq” is used in GLSL shaders. Much like “rgba”, you can use “stpq” as swizzle selectors for any vector instead of the traditional “xyzw”.
NOTE: The reason for the odd naming is that OpenGL tries to keep vector suffixes from conflicting. “uvw” does not work because “w” is already part of the “xyzw” suffix. In GLSL, the “r” in “strq” conflicts with “rgba”, so they had to go with “stpq” instead.

• The GL_TEXTURE_WRAP_S parameter defines how the “s” component of the texture coordinate will be adjusted if it falls outside of the [0, 1] range. Setting this to GL_CLAMP_TO_EDGE clamps this component of the texture coordinate to the edge of the texture. Each component of the texture coordinate can have a separate wrapping mode. Since our texture is a 1D texture, its texture coordinates only have one component.

• The sampler object is used similarly to how textures are associated with GLSL samplers: we bind them to a texture image unit. The API is much simpler than what we saw for textures:

```c
glBindSampler(0, cartoonSampler);
```

• In the above function we pass the texture unit directly; there is no need to add GL_TEXTURE0 to it to convert it into an enumerator. This effectively adds an additional value to each texture unit.
Let's now see how the sampler object integrates into our diagram presented earlier:

NOTE: Technically, we do not have to use a sampler object. The parameters we use for samplers could have been set into the texture object directly with `glTexParameter`. Sampler objects have a lot of advantages over setting the value in the texture, and binding a sampler object overrides parameters set in the texture. There are still some parameters that must be in the texture object, and those are not overridden by the sampler object.

Well that's it for a small introduction on using textures. While the algorithm used in this sample to limit the value of the angle of incidence to a few values is useless, it shows how textures can be viewed as something more than just pictures.

Additional reading: [http://www.arcsynthesis.org/gltut/Texturing/Tutorial%202014.html](http://www.arcsynthesis.org/gltut/Texturing/Tutorial%202014.html)
32. Applying a 2D Texture

- In this sample, we'll look at a simple example involving the application of a 2D texture to a surface as shown in the following image. We'll use the texture color to scale the color provided by the Phong(ADS) shading model. The following image shows the results of a brick texture applied to a cube. The texture is shown on the right and the rendered result is on the left.

- To achieve the above result, we must first find a way to associate points on our triangles with texels on a texture. This association is called *texture mapping*, since it maps between points on a triangle and locations on the texture. This is achieved by using *texture coordinates* that correspond with positions on the surface.

- In the last sample, the texture coordinate was a value computed based on the angle of incidence. The texture coordinate for accessing our brick texture will instead come from interpolated per-vertex parameters.
For simple cases, we could generate the texture coordinates from vertex positions and pass them down the pipeline to the fragment shader. However, in the vast majority of cases, texture coordinates for texture mapping will be part of the per-vertex attribute data. In this sample, the texture coordinates will be provided at attribute location 2.

Our brick texture is defined by a file. To load the data from file, we use a library called SOIL (Simple OpenGL Image Library - [http://www.lonesock.net/soil.html](http://www.lonesock.net/soil.html)). Let's see how we can load a texture from a file and pass it to OpenGL using SOIL.

Start by creating an empty texture object and bind it to `GL_TEXTURE_2D` target, which means our texture will have two dimensions.

```gl
glGenTextures(1, &textureHandle);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureHandle);
```

Next, we use the SOIL library to read the texture data from file into RAM.

```c
int width, height;

unsigned char* image = SOIL_load_image("texture/brick1.jpg", &width, &height, 0, SOIL_LOAD_RGB);
```
The next step is to copy the data from RAM to the OpenGL memory (on the GPU). We use `glTexImage2D` instead of the 1D version. This takes both a width and a height. But otherwise, the code is very similar to the previous version(`glTexImage1D`).

```c
int i;
for (i = 0; i < width; i++)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
```

Make sure to free the RAM data occupied by SOIL library after we finished transferring data from RAM to OpenGL memory.

```c
SOIL_free_image_data(image);
```

In the next step we create the OpenGL sampler object. Because we work with a 2D texture, we use two coordinates to access the texels: S and T. So we need to clamp both S and T in our sampler object:

```c
glGenSamplers(1, &textureSampler);  
glSamplerParameteri(textureSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);  
glSamplerParameteri(textureSampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

The next step involves setting the magnification and minimization filters for the texture object. We'll talk more on them in the next sample.

```c
glSamplerParameteri(textureSampler, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glSamplerParameteri(textureSampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```
• Next bind the sampler to texture unit 0.

    glBindSampler(0, textureSampler);

• Finally, we set the uniform variable `Tex1` from the GLSL program to zero. This is our GLSL sampler variable. Setting its value to zero indicates to the OpenGL system that the variable should refer to the texture unit 0.

    prog.setUniform("Tex1", 0);

• The vertex shader is very similar to the one used in previous examples except for the addition of the texture coordinate input variable `VertexTexCoord`, which is bound to attribute location 2.

    layout (location = 2) in vec2 VertexTexCoord;

    Its value is simply passed along to the fragment shader by assigning it to the shader output variable `TexCoord`. Vertex shader main() function is given below:

    void main()
    {
        TexCoord = VertexTexCoord;
        Normal = normalize( NormalMatrix * VertexNormal);
        Position = vec3( ModelViewMatrix * vec4(VertexPosition,1.0) );
        gl_Position = MVP * vec4(VertexPosition,1.0);
    }
Now let's see what happens in the fragment shader. Since we are using texture objects of GL_TEXTURE_2D type, we must use **sampler2D** samplers in our shader.

```glsl
uniform sampler2D Tex1;
```

In the main() function of the fragment shader, we use that variable; along with the texture coordinate (TexCoord) to access the texture.

```glsl
void main()
{
    vec3 ambAndDiff, spec;
    vec4 texColor = texture( Tex1, TexCoord );
    phongModel( Position, Normal, ambAndDiff, spec );
    FragColor = (vec4( ambAndDiff, 1.0 ) * texColor) + vec4(spec,1.0);
}
```

After accessing the texture, in the code above, the shading model is evaluated by calling **phongModel()** and the results are returned in the parameters **ambAndDiff** and **spec**. The variable **ambAndDiff** contains only the ambient and diffuse components of the shading model. A color texture is often only intended to affect the diffuse component of the shading model and not the specular. So we multiply the texture color by the ambient and diffuse components and then add the specular. The final sum is then applied to the output fragment FragColor.

Additional reading:
OpenGL 4.0 Shading Language Cookbook, pages 105-110
53. Linear filtering **SAMPLE33.1**

- In this sample we will draw a single large, flat plane. The plane will have a texture of a checkerboard drawn on it.

  ![Checkerboard Texture](image1)

- The camera will hover above the plane, looking out at the horizon as if the plane were the ground. The output of the sample is given below:

  ![Checkerboard Texture with Camera](image2)

- The first thing you can notice in the picture above is that the texture is repeated many times over the plane. The texture coordinates are well outside the [0,1] range. They span from -512 to 512 now, but the remember that texture itself is only valid within the [0, 1] range.
Recall from the last sample that the sampler object has a parameter that controls what texture coordinates outside of the [0, 1] range mean. Here's how we instruct OpenGL to deal with texture coordinates outside [0,1] range:

\[
\text{glSamplerParameteri}(\text{textureSampler}, \text{GL\_TEXTURE\_WRAP\_S}, \text{GL\_REPEAT}); \\
\text{glSamplerParameteri}(\text{textureSampler}, \text{GL\_TEXTURE\_WRAP\_T}, \text{GL\_REPEAT});
\]

We set the S and T wrap modes to GL_REPEAT. This means that values outside of the [0, 1] range wrap around to values within the range. So a texture coordinate of 1.1 becomes 0.1, and a texture coordinate of -0.1 becomes 0.9. The idea is to make it as though the texture were infinitely large, with infinitely many copies repeating over and over.

Note: It is perfectly legitimate to set the texture coordinate wrapping modes differently for different coordinates.

While this example certainly draws a checkerboard, you can see that there are some visual issues. Take a look at one of the squares in the middle of the screen. Notice how the line looks jagged as we press W and S. You can see the pixels of it sort of crawl up and down as it shifts around on the plane.
• This is caused by the discrete nature of our texture accessing. The texture coordinates are all in floating-point values. The GLSL texture function internally converts these texture coordinates to specific texel values within the texture. So what value do you get if the texture coordinate lands halfway between two texels?

• That is governed by a process called texture filtering. Filtering can happen in two directions: magnification and minification.

• Magnification happens when the texture mapping makes the texture appear bigger in screen space than its actual resolution. If you get closer to the texture, relative to its mapping, then the texture is magnified relative to its natural resolution. Minification is the opposite: when the texture is being shrunken relative to its natural resolution.

• In OpenGL, magnification and minification filtering are each set independently. That is what the GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER sampler parameters control. We are currently using GL_NEAREST for both; this is called nearest filtering. This mode means that each texture coordinate picks the texel value that it is nearest to. For our checkerboard, that means that we will get either black or white.

• Now this may sound fine, since our texture is a checkerboard and only has two actual colors. However, it is exactly this discrete sampling that gives rise to the pixel crawl effect. A texture coordinate that is halfway between the white and the black is either white or black; a small change in the camera causes an instant pop from black to white or vice-versa.
Another type of filtering is the *linear filtering*. The idea behind the linear filtering is when a fragment is generated in the scan line conversion algorithm, instead of picking a single sample for the texture coordinate provided for this fragment from the vertex shader (via interpolation), pick the nearest 4 samples and then interpolate the values based on how close they each are to this texture coordinate.

In the picture above the dot represents the fragment's texture coordinate location on the texture. The box is the area that the fragment covers. Using linear filtering the final fragment color will be a weighted average of the four nearest texels based on how far these texels are from texture coordinate (from the dot).

In SAMPLE33.1 linear filtering can be activated by pressing the 2 key (press 1 to go back to nearest filtering). The code for linear filtering activation is as follows:

```glsl
glSamplerParameteri(textureSampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(textureSampler, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```
• Below you can see the comparison between nearest and linear filtering:

![Comparison between nearest and linear filtering](image)

• That looks much better for the squares close to the camera. It creates a bit of fuzziness, but this is generally a lot easier for the viewer to tolerate than pixel crawl. Human vision tends to be attracted to movement, and false movement like dot crawl can be distracting.

Additional reading:
http://www.arcsynthesis.org/gltut/Texturing/Tutorial%2015.html
http://www.arcsynthesis.org/gltut/Texturing/Tut15%20Magnification.html
Let's talk about what is going on in the distance. If you open SAMPLE33.1, when the camera moves (try it by yourself), the more distant parts of the texture look like a jumbled mess. Even when the camera motion is paused, it still doesn't look like a checkerboard.

What is going on there is really simple. The way our filtering works is that, for a given texture coordinate, we take either the nearest texel value, or the nearest 4 texels and interpolate. The problem is that, for distant areas of our surface, the texture space area covered by our fragment is much larger than 4 texels across.

In the picture above, the inner box (green) represents the nearest texels used in linear filtering algorithm, while the outer box represents the area of the texture that is mapped to our fragment. We can see that the value we get with linear sampling will be pure white, since the four nearest values are white. But the value we should get based on the covered area is some shade of gray.
In order to accurately represent this area of the texture, we would need to sample from more than just 4 texels. The GPU is certainly capable of detecting the fragment area and sampling enough values from the texture to be representative. But this would be exceedingly expensive, both in terms of texture bandwidth and computation (maybe that is something that would become available in the future).

What if, instead of having to sample more texels, we had a number of smaller versions of our texture? The smaller versions effectively pre-compute groups of texels. That way, we could just sample 4 texels from a texture that is close enough to the size of our fragment area.

These smaller versions of an image are called **mipmaps**; they are also sometimes called **mipmap levels**. In the figure above, the image on the left is the original texture and the one from the right is a mipmap which is the same texture downscaled twice. By performing linear sampling against a lower mipmap level, we get a gray value that, while not the exact color the coverage area suggests, is much closer to what we should get than linear filtering on the large mipmap.
In OpenGL, mipmaps are numbered starting from 0. The 0 image is the largest mipmap, what is usually considered the main texture image. When people speak of a texture having a certain size, they mean the resolution of mipmap level 0. Each mipmap is half as small as the previous one. So if our main image, mipmap level 0, has a size of 128x128, the next mipmap, level 1, is 64x64. The next is 32x32. And so forth, down to 1x1 for the smallest mipmap.

For textures that are not square (which as we saw in the previous tutorial, is perfectly legitimate), the mipmap chain keeps going until all dimensions are 1. So a texture who's size is 128x16 (remember: the texture's size is the size of the largest mipmap) would have just as many mipmap levels as a 128x128 texture. The mipmap level 4 of the 128x16 texture would be 8x1; the next mipmap would be 4x1.

Note: It is also perfectly legal to have texture sizes that are not powers of two. For them, mipmap sizes are always rounded down. So a 129x129 texture's mipmap 1 will be 64x64. A 131x131 texture's mipmap 1 will be 65x65, and mipmap 2 will be 32x32.

For SAMPLE33.2 we'll be using a brick texture:
First thing, load the texture as in the previous sample:

```c
glActiveTexture(GL_TEXTURE0);
glGenTextures(1,&textureHandle);
glBindTexture(GL_TEXTURE_2D, textureHandle);

int width, height;
unsigned char* imageDataPtr =SOIL_load_image("texture/brick.jpg", &width,
&height, 0, SOIL_LOAD_RGB);

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
GL_RGB,GL_UNSIGNED_BYTE, imageDataPtr);
SOIL_free_image_data(imageDataPtr);
```

Next we're using the OpenGL function `glGenerateMipmap()` to generate the full mipmap list to the texture currently bound to the context.

```c
int mipmapLevelCount=floor(log2((float)min(width, height)));
generateMipmap(GL_TEXTURE_2D);
```

NOTE: Mipmap levels are usually stored inside the graphics files themselves. In such a case we must read the mipmap levels from disk to RAM and then transfer them to OpenGL dedicated memory.

The next thing we should do is tell OpenGL what mipmaps in our texture can be used. The `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL` parameters are used to specify that.

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, mipmapLevelCount);
```
• The base level of a texture is the largest usable mipmap level, while the max level is the smallest usable level. It is possible to omit some of the smaller mipmap levels. Note that level 0 is always the largest possible mipmap level.

• Filtering based on mipmaps is unsurprisingly named **mipmap filtering**. Setting the base and max level is not enough; the sampler object must be told to use mipmap filtering. If it does not, then it will simply use the base level.

• Mipmap filtering only works for minification, since minification represents a fragment area that is larger than the texture's resolution. To activate this, we use a special MIPMAP mode of minification filtering.

  ```glsl
  glSamplerParameteri(textureSampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glSamplerParameteri(textureSampler, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
  ```

• The **GL_LINEAR_MIPMAP_NEAREST** minification filter means the following. For a particular call to the GLSL **texture** function, it will detect which mipmap is the one that is nearest to our fragment area. This detection is based on the angle of the surface relative to the camera's view. Then, when it samples from that mipmap, it will use linear filtering of the four nearest samples within that one mipmap.
• If you press '2' in this sample you can see the effects of mipmap filtering.

• That's a lot more reasonable. It isn't perfect, but it is much better than the random motion in the distance that we have previously seen.
• **Filtering Between Mipmaps.** Our mipmap filtering has been a dramatic improvement over previous efforts. However, it does create artifacts. One of particular concern is the change between mipmap levels. It is abrupt and somewhat easy to notice for a moving scene. Perhaps there is a way to smooth that out.

• Our current minification filtering picks a single mipmap level and selects a sample from it. It would be better if we could pick the two nearest mipmap levels and blend between the values fetched from the two textures. This would give us a smoother transition from one mipmap level to the next.

• This is done by using GL_LINEAR_MIPMAP_LINEAR minification filtering. The first LINEAR represents the filtering done within a single mipmap level, and the second LINEAR represents the filtering done between mipmap levels.

• OpenGL actually allows all combinations of NEAREST and LINEAR in minification filtering. Using nearest filtering within a mipmap level while linearly filtering between levels (GL_NEAREST_MIPMAP_LINEAR) is possible but not terribly useful in practice.

Like linear and mipmap filtering, anisotropic filtering eliminates aliasing effects, but improves on these other techniques by reducing blur and preserving detail at extreme viewing angles. Below is an image filtered with mipmap filtering on the left and anisotropic filtering on the right.

Anisotropic filtering requires taking multiple samples from the various mipmaps. The control on the quality of anisotropic filtering is in limiting the number of samples used. Raising the maximum number of samples taken will generally make the result look better, but it will also decrease performance.
• This is done by setting the `GL_TEXTURE_MAX_ANISOTROPY_EXT` sampler parameter:

```glSL
    glSamplerParameteri(g_samplers[4], GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glSamplerParameteri(g_samplers[4], GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glSamplerParameterf(g_samplers[4], GL_TEXTURE_MAX_ANISOTROPY_EXT, 4.0f);
```

• This represents the maximum number of samples that will be taken for any texture accesses through this sampler. Note that we still use linear mipmap filtering in combination with anisotropic filtering. While you could theoretically use anisotropic filtering without mipmaps, you will get much better performance if you use it in tandem with linear mipmap filtering.

• There is a limit to the maximum anisotropy that we can provide. This limit is implementation defined; it can be queried with `glGetFloatv`, since the value is a float rather than an integer. To set the max anisotropy to the maximum possible value, we do this.

```glSL
    GLfloat maxAniso = 0.0f;
    glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &maxAniso);
    glSamplerParameteri(g_samplers[5], GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glSamplerParameteri(g_samplers[5], GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glSamplerParameterf(g_samplers[5], GL_TEXTURE_MAX_ANISOTROPY_EXT, maxAniso);
```

• Detailed description of the anisotropic filtering is beyond the scope of this course. You can read more about anisotropic filtering below.

Additional reading:

56. Planets sample

- A sample involving rendering and rotation of two planets can be downloaded from here.
57. To do

- Dynamic range and gamma correction:
- Lies and impostors
- Review one again the only arcsynthesis book and add skipped material
- Add the remaining OpenGL 4.0 Shading Language Cookbook samples
- Intersection tests
- Texture animation
- Visibility determination
- Multisampling
- Next semester update
  - Review the whole course and optimize it
  - Fix GL 1.20 compatibility issues
    - texture() must be replaced with texture2D()
    - GL_TESSELATION_SHADER is not available in GL 1.20
    - glSampleParameter(), glGenerateMipmaps() also not available in GL 1.20