Learning Modern 3D Graphics Programming

Jason L. McKesson

Learning Modern 3D Graphics Programming
Jason L. McKesson

Table of Contents

About this Book

Three dimensional graphics hardware is fast becoming, not merely a staple of computer systems, but an indispensable component. Many operating systems directly use and even require some degree of 3D rendering hardware. Even in the increasingly important mobile computing space, 3D graphics hardware is a standard feature of all but the lowest power devices.

Understanding how to make the most of that hardware is a difficult challenge, particularly for someone new to graphics and rendering.

Why Read This Book?

There are many physical books for teaching graphics. There are many more online repositories of knowledge, in the form of wikis, blogs, tutorials, and forums. So what does this book offer that others do not?

Programmability. Virtually all of the aforementioned sources instruct beginners using something called “fixed functionality.” This represents configurations in older graphics processors that define how a particular rendering operation will proceed. It is generally considered easiest to teach neophyte graphics programmers using the fixed function pipeline.

This is considered true because it is easy to get something to happen with fixed functionality. It's simpler to make pictures that look like something real. The fixed function pipeline is like training wheels for a bicycle.

There are downsides to this approach. First, much of what is learned with this approach must be inevitably abandoned when the user encounters a graphics problem that must be solved with programmability. Programmability wipes out almost all of the fixed function pipeline, so the knowledge does not easily transfer.

A more insidious problem is that the fixed function pipeline can give the *illusion* of knowledge. A user can think they understand what they are doing, but they're really just copy-and-pasting code around. Programming thus becomes akin to magical rituals: you put certain bits of code before other bits, and everything seems to work.

This makes debugging nightmarish. Because the user never really understood what the code does, the user is unable to diagnose what a particular problem could possibly mean. And without that ability, debugging becomes a series of random guesses as to what the problem is.

By contrast, you cannot use a programmable system successfully without first understanding it. Confronting programmable graphics hardware means confronting issues that fixed function materials often gloss over. This may mean a slower start overall, but when you finally get to the end, you truly *know* how everything works.

Another problem is that, even if you truly understand the fixed function pipeline, it limits how you think about solving problems. Because of its inflexibility, it focuses your mind along certain problem solving possibilities and away from others. It encourages you to think of textures as pictures; vertex data as texture coordinates, colors, or positions; and the like. By its very nature, it limits creativity and problem solving.

Lastly, even on mobile systems, fixed functionality is generally not available in the graphics hardware. Programmability is the order of the day for most graphics hardware, and this will only become more true in the future.

What this book offers is beginner-level instruction on what many consider to be an advanced concept. It teaches programmable rendering for beginning graphics programmers, from the ground up.

This book also covers some important material that is often neglected or otherwise relegated to “advanced” concepts. These concepts are not truly advanced, but they are often ignored by most introductory material because they do not work with the fixed function pipeline.

This book is first and foremost about learning how to be a graphics programmer. Therefore, whenever it is possible and practical, this book will present material in a way that encourages the reader to examine what graphics hardware can do in new and interesting ways. A good graphics programmer sees the graphics hardware as a set of tools to fulfill their needs, and this book tries to encourage this kind of thinking.

One thing this book is not, however, is a book on graphics APIs. While it does use OpenGL and out of necessity teach rendering concepts in terms of OpenGL, it is not truly a book that is *about* OpenGL. It is not the purpose of this book to teach you all of the ins and outs of the OpenGL API.There will be parts of OpenGL functionality that are not dealt with because they are not relevant to any of the lessons that this book teaches. If you already know graphics and are in need of a book that teaches modern OpenGL programming, this is not it. It may be useful to you in that capacity, but that is not this book's main thrust.

This book is intended to teach you how to be a graphics programmer. It is not aimed at any particular graphics field; it is designed to cover most of the basics of 3D rendering. So if you want to be a game developer, a CAD program designer, do some computer visualization, or any number of things, this book can still be an asset for you.

This does not mean that it covers everything there is about 3D graphics. Hardly. It tries to provide a sound foundation for your further exploration in whatever field of 3D graphics you are interested in.

One topic this book does not cover in depth is optimization. The reason for this is simply that serious optimization is an advanced topic. Optimizations can often be platform-specific, different for different kinds of hardware. They can also be API-specific, as some APIs have different optimization needs. Optimizations may be mentioned here and there, but it is simply too complex of a subject for a beginning graphics programmer. There is a chapter in the appendix covering optimization opportunities, but it only provides a fairly high-level look.

What You Need

This is a book for beginning graphics programmers; it can also serve as a book for those familiar with fixed functionality who want to understand programmable rendering better. But this is not a book for beginning programmers.

You are expected to be able to read C and reasonable C++ code. If “Hello, world!” is the extent of your C/C++ knowledge, then perhaps you should write some more substantial code before proceeding with trying to render images. 3D graphics rendering is simply not a beginner programming task; this is just as true for traditional graphics learning as for modern graphics learning.

These tutorials should be transferable to other languages as well. If you can read C/C++, that is enough to understand what the code is doing. The text descriptions that explain what the code does are also sufficient to get information out of these tutorials.

Any substantial discussion of 3D rendering requires a discussion of mathematics, which are at the foundation of all 3D graphics. This book expects you to know basic geometry and algebra.

The tutorials will present the more advanced math needed for graphics as it becomes necessary, but you should have at least a working knowledge of geometry and algebra. Linear algebra is not required, though it would be helpful.

The code tutorials in this book use OpenGL as their rendering API. You do not need to know OpenGL, but to execute the code, you must have a programming environment that allows OpenGL. Specifically, you will need hardware capable of running OpenGL version 3.3. This means any GeForce 8xxx or better, or any Radeon HD-class card. These are also called “Direct3D 10” cards, but you do not need Windows Vista or 7 to use their advanced features through OpenGL.

Organization of This Book

This book is broken down into a number of general subjects. Each subject contains several numbered chapters called tutorials. Each tutorial describes several related concepts. In virtually every case, each concept is demonstrated by a companion set of code.

Each tutorial begins with an overview of the concepts that will be discussed and demonstrated. At the end of each tutorial is a review section and a glossary of all terms introduced in that tutorial. The review section will explain the concepts presented in the tutorial. It will also contain suggestions for playing with the source code itself; these are intended to further your understanding of these concepts. If the tutorial introduced new OpenGL functions or functions for the OpenGL shading language, they will be reviewed here as well.

This is a book for beginning graphics programmers. Graphics is a huge topic, and this book will not cover every possible effect, feature, or technique. This book will also not cover every technique in full detail. Sometimes techniques will be revisited in later materials, but there simply isn't enough space to say everything about everything. Therefore, when certain techniques are introduced, there will be a section at the end providing some cursory examination of more advanced techniques. This will help you further your own research into graphics programming, as you will know what to search for online or in other books.

Each tutorial ends with a glossary of all of the terms defined in that tutorial.

Browser Note

This website and these tutorials make extensive use of SVG images. Basic SVG support is in all major browsers except all Internet Explorer versions before version 9. If you are content with these versions of Internet Explorer (or unable to upgrade), consider installing the Google Chrome Frame add-on for IE8. This will allow you to see the images correctly.

Conventions used in This Book

Text in this book is styled along certain conventions. The text styling denotes what the marked-up text represents.

- *defined term*: This term will have a definition in the glossary at the end of each tutorial.
- `FunctionNames`: These can be in C, C++, or the OpenGL Shading Language.
- `nameOfVariable`: These can be in C, C++, or the OpenGL Shading Language.
- `GL_ENUMERATORS`
- `Names/Of/Paths/And/Files`
- **K**: The keyboard key “K,” which is not the same as the capital letter “K”. The latter is what you get by pressing **Shift**+**K**.

Building the Tutorials

These tutorials require a number of external libraries in order to function. The specific version of these libraries that the tutorials use are distributed with the tutorials. The tutorial source distribution [http://bitbucket.org/alfonse/gltut/downloads] can be found online. This section will describe each of the external libraries, how to build them, and how to build the tutorials themselves. Windows and Linux builds are supported.

You will need minimal familiarity with using the command line in order to build these tutorials. Also, any mention of directories is always relative to where you unzipped this distribution.

File Structure

The layout of the files in the tutorial directory is quite simple. The `framework` directory and all directories of the form `Tut*` contain all of the source code for the tutorials themselves. Each `Tut*` directory has the code for the various tutorials. The `framework` directory simply contains utility code that is commonly used by each tutorial.

Each tutorial contains one or more projects; each project is referenced in the text for that tutorial.

The `Documents` directory contains the source for the text documentation explaining how these tutorials work. This source is in xml files using the DocBook 5.0 format.

Every other directory contains the code and build files for a library that the tutorials require.

Necessary Utilities

In order to build everything, you will need to download the Premake 4 [http://industriousone.com/premake] utility for your platform of choice.

Premake is a utility like CMake [http://www.cmake.org/]: it generates build files for a specific platform. Unlike CMake, Premake is strictly a command-line utility. Premake's build scripts are written in the Lua language [http://www.lua.org/home.html], unlike CMake's build scripts that use their own language.

Note that Premake only generates build files; once the build files are created, you can use them as normal. It can generate project files for Visual Studio, Code::Blocks [http://www.codeblocks.org/], and XCode, as well as GNU Makefiles. And unless you want to modify one of the tutorials, you only need to run Premake once for each tutorial.

The Premake download comes as a pre-built executable for all platforms of interest, including Linux.

Unofficial OpenGL SDK

The Unofficial OpenGL SDK [http://glsdk.sourceforge.net/docs/html/index.html] is an aggregation of libraries, unifying a number of tools for developing OpenGL applications, all bound together with a unified build system. A modified SDK distribution is bundled with these tutorials; this distro does not contain the documentation or GLFW that comes with the regular SDK.

The SDK his library uses Premake to generate its build files. So, with **premake4.exe** in your path, go to the `glsdk` directory. Type **`premake4 plat`**, where *`plat`* is the name of the platform of choice. For Visual Studio 2008, this would be “vs2008”; for VS2010, this would be “vs2010.” This will generate Visual Studio projects and solution files for that particular version.

For GNU and makefile-based builds, this is “gmake”. This will generate a makefile. To build for debug, use **`make config=debug`**; similarly, to build for release, use **`make config=release`**.

Using the generated build files, compile for both debug and release. You should build the entire solution; the tutorials use all of the libraries provided.

Note that there is no execution of **`make install`** or similar constructs. The SDK is designed to be used where it is; it does not install itself to any system directories on your machine. Incidentally, neither do these tutorials.

Tutorial Building

Each tutorial directory has a `premake4.lua` file; this file is used by Premake to generate the build files for that tutorial. Therefore, to build any tutorial, you need only go to that directory and type **`premake4 plat`**, then use those build files to build the tutorial.

Each tutorial will generally have more than one source file and generate multiple executables. Each executable represents a different section of the tutorial, as explained in that tutorial's documentation.

If you want to build all of the tutorials at once, go to the root directory of the distribution and use Premake on the `premake4.lua` file in that directory. It will put all of the tutorials into one giant project that you can build.

If you look at any of the tutorial source files, you will not find the `main` function defined anywhere. This function is defined in `framework/framework.cpp`; it and all of the other source files in the `framework` directory is shared by every tutorial. It does the basic boilerplate work: creating a FreeGLUT window, etc. This allows the tutorial source files to focus on the useful OpenGL-specific code.

Part I. The Basics

Graphics programming can be a daunting task when starting out. The rendering pipeline involves a large number of steps, each dealing with a variety of math operations. It has stages that run actual programs to compute results for the next. Mastering this pipeline, being able to use it as a tool to achieve a visual effect, is the essence of being a graphics programmer.

This section of the book will introduce the basic math necessary for 3D graphics. It will introduce the rendering pipeline as defined by OpenGL. And it will demonstrate how data flows through the graphics pipeline.

Introduction

Unlike most sections of this text, there is no source code or project associated with this section. Here, we will be discussing vector math, graphical rendering theory, and OpenGL. This serves as a primer to the rest of the book.

Vector Math

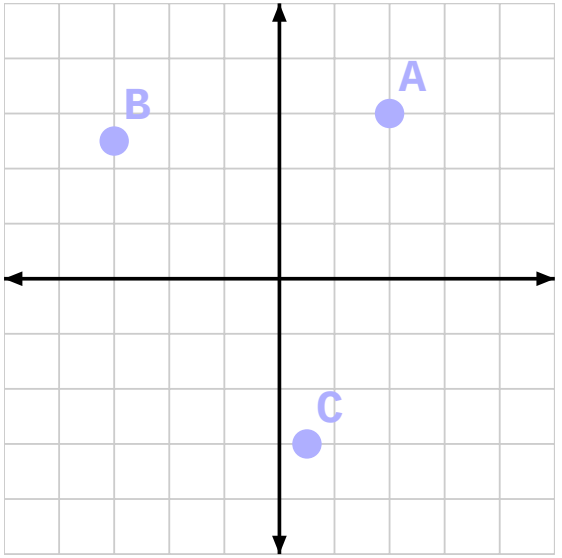
This book assumes that you are familiar with algebra and geometry, but not necessarily with vector math. Later material will bring you up to speed on more complex subjects, but this will introduce the basics of vector math.

A *vector* can have many meanings, depending on whether we are talking geometrically or numerically. In either case, vectors have dimensionality; this represents the number of dimensions that the vector has. A two-dimensional vector is restricted to a single plane, while a three-dimensional vector can point in any physical space. Vectors can have higher dimensions, but generally we only deal with dimensions between 2 and 4.

Technically, a vector can have only one dimension. Such a vector is called a *scalar*.

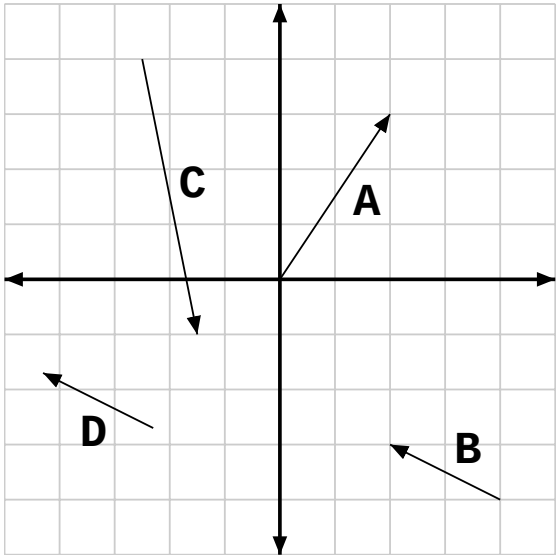
In terms of geometry, a vector can represent one of two concepts: a position or a direction within a particular space. A vector *position* represents a specific location in space. For example, on this graph, we have a vector position A:

Figure 1. Position Vectors



A vector can also represent a *direction*. Direction vectors do not have an origin; they simply specify a direction in space. These are all direction vectors, but the vectors B and D are the same, even though they are drawn in different locations:

Figure 2. Direction Vectors



That's all well and good for geometry, but vectors can also be described numerically. A vector in this case is a sequence of numbers, one for each dimension. So a two-dimensional vector has two numbers; a three-dimensional vector has three. And so forth. Scalars, numerically speaking, are just a single number.

Each of the numbers within a vector is called a *component*. Each component usually has a name. For our purposes, the first component of a vector is the X component. The second component is the Y component, the third is the Z, and if there is a fourth, it is called W.

When writing vectors in text, they are written with parenthesis. So a 3D vector could be (0, 2, 4); the X component is 0, the Y component is 2, and the Z component is 4. When writing them as part of an equation, they are written as follows:

$$\vec{a} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

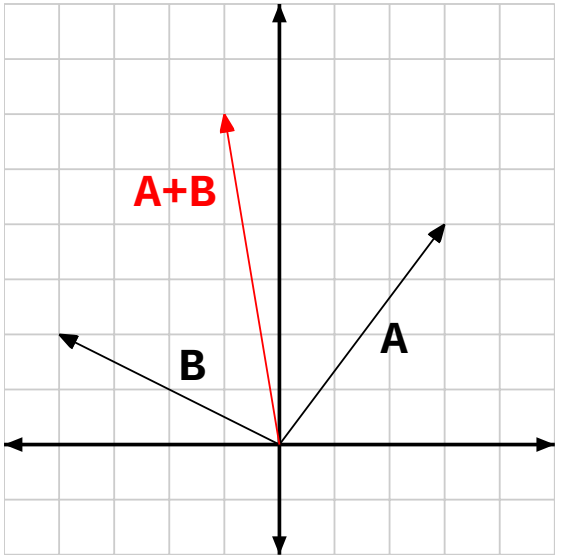
In math equations, vector variables are either in boldface or written with an arrow over them.

When drawing vectors graphically, one makes a distinction between position vectors and direction vectors. However, numerically there is *no* difference between the two. The only difference is in how you use them, not how you represent them with numbers. So you could consider a position a direction and then apply some vector operation to them, and then consider the result a position again.

Though vectors have individual numerical components, a vector as a whole can have a number of mathematical operations applied to them. We will show a few of them, with both their geometric and numerical representations.

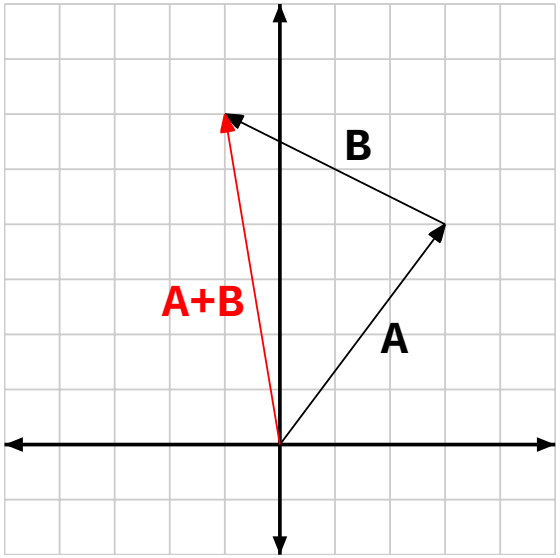
Vector Addition. You can take two vectors and add them together. Graphically, this works as follows:

Figure 3. Vector Addition



Remember that vector directions can be shifted around without changing their values. So if you put two vectors head to tail, the vector sum is simply the direction from the tail of the first vector to the head of the last.

Figure 4. Vector Addition Head-to-Tail



Numerically, the sum of two vectors is just the sum of the corresponding components:

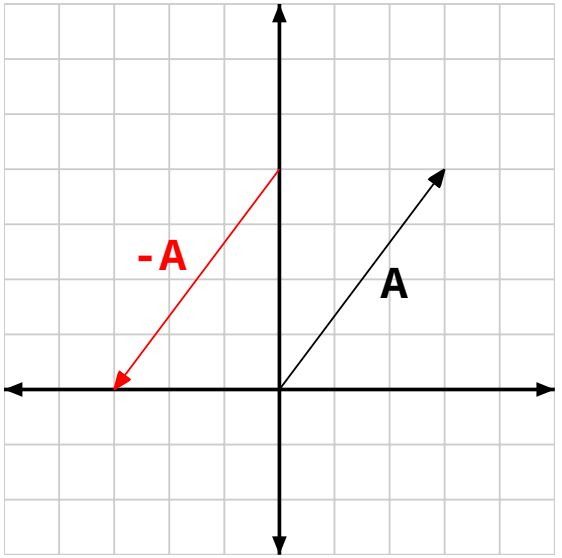
Equation 1. Vector Addition with Numbers

$$\vec{a} + \vec{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{bmatrix}$$

Any operation where you perform an operation on each component of a vector is called a *component-wise operation*. Vector addition is component-wise. Any component-wise operation on two vectors requires that the two vectors have the same dimensionality.

Vector Negation and Subtraction. You can negate a vector. This reverses its direction:

Figure 5. Vector Negation



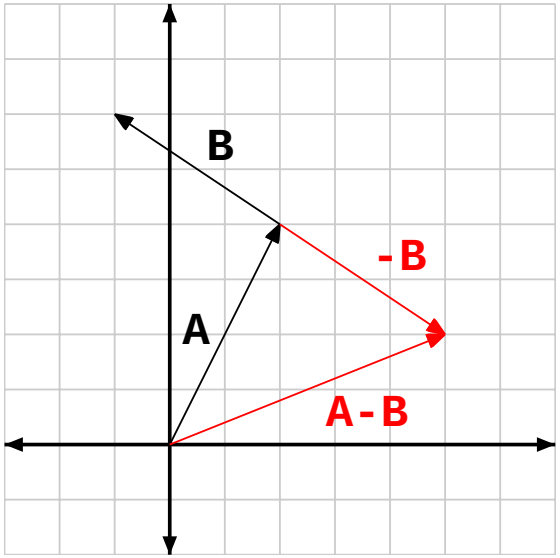
Numerically, this means negating each component of the vector.

Equation 2. Vector Negation

$$-\vec{a} = -\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} -a_x \\ -a_y \\ -a_z \end{bmatrix}$$

Just as with scalar math, vector subtraction is the same as addition with the negation of the second vector.

Figure 6. Vector Subtraction



Vector Multiplication. Vector multiplication is one of the few vector operations that has no real geometric equivalent. To multiply a direction by another, or multiplying a position by another position, does not really make sense. That does not mean that the numerical equivalent is not useful, though.

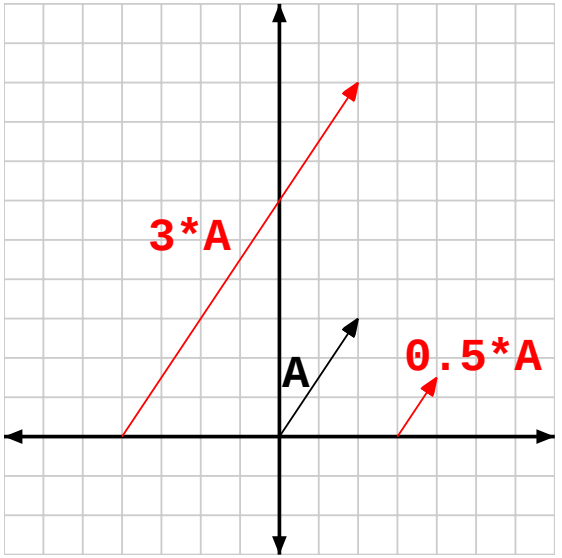
Multiplying two vectors numerically is simply component-wise multiplication, much like vector addition.

Equation 3. Vector Multiplication

$$\vec{a} * \vec{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} * \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x * b_x \\ a_y * b_y \\ a_z * b_z \end{bmatrix}$$

Vector/Scalar Operations. Vectors can be operated on by scalar values. Recall that scalars are just single numbers. Vectors can be multiplied by scalars. This magnifies or shrinks the length of the vector, depending on the scalar value.

Figure 7. Vector Scaling



Numerically, this is a component-wise multiplication, where each component of the vector is multiplied with each component of the scalar.

Equation 4. Vector-Scalar Multiplication

$$s * \vec{a} = s * \begin{bmatrix} a_x \\ a_z \\ a_z \end{bmatrix} = \begin{bmatrix} s * a_x \\ s * a_z \\ s * a_z \end{bmatrix}$$

Scalars can also be added to vectors. This, like vector-to-vector multiplication, has no geometric representation. It is a component-wise addition of the scalar with each component of the vector.

Equation 5. Vector-Scalar Addition

$$s + \vec{a} = s + \begin{bmatrix} a_x \\ a_z \\ a_z \end{bmatrix} = \begin{bmatrix} s + a_x \\ s + a_z \\ s + a_z \end{bmatrix}$$

Vector Algebra. It is useful to know a bit about the relationships between these kinds of vector operations.

Vector addition and multiplication follow many of the same rules for scalar addition and multiplication. They are commutative, associative, and distributive.

Equation 6. Vector Algebra

Commutative: $\vec{a} + \vec{b} = \vec{b} + \vec{a}$ $\quad\quad$ $\vec{a} * \vec{b} = \vec{b} * \vec{a}$

Associative: $\vec{a} + (\vec{b} + \vec{c}) = (\vec{a} + \vec{b}) + \vec{c}$ $\quad\quad$ $\vec{a} * (\vec{b} * \vec{c}) = (\vec{a} * \vec{b}) * \vec{c}$

Distributive: $\vec{a} * (\vec{b} + \vec{c}) = (\vec{a} * \vec{b}) + (\vec{a} * \vec{c})$

Vector/scalar operations have similar properties.

Length. Vectors have a length. The length of a vector direction is the distance from the starting point to the ending point.

Numerically, computing the distance requires this equation:

Equation 7. Vector Length

$$\# \vec{a} \# = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

This uses the Pythagorean theorem to compute the length of the vector. This works for vectors of arbitrary dimensions, not just two or three.

Unit Vectors and Normalization. A vector that has a length of exactly one is called a *unit vector*. This represents a pure direction with a standard, unit length. A unit vector variable in math equations is written with a ^ over the variable name.

A vector can be converted into a unit vector by *normalizing* it. This is done by dividing the vector by its length. Or rather, multiplication by the reciprocal of the length.

Equation 8. Vector Normalization

$$\hat{a} = \frac{1}{\# \vec{a} \#} * \vec{a} = \begin{bmatrix} \frac{a_x}{\# \vec{a} \#} \\ \frac{a_y}{\# \vec{a} \#} \\ \frac{a_z}{\# \vec{a} \#} \end{bmatrix}$$

This is not all of the vector math that we will use in these tutorials. New vector math operations will be introduced and explained as needed when they are first used. And unlike the math operations introduced here, most of them are not component-wise operations.

Range Notation. This book will frequently use standard notation to specify that a value must be within a certain range.

If a value is constrained between 0 and 1, and it may actually have the values 0 and 1, then it is said to be "on the range" [0, 1]. The square brackets mean that the range includes the value next to it.

If a value is constrained between 0 and 1, but it may not actually have a value of 0, then it is said to be on the range (0, 1]. The parenthesis means that the range does not include that value.

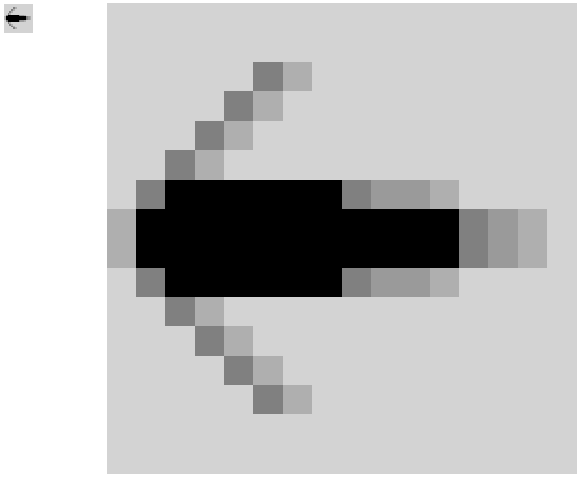
If a value is constrained to 0 or any number greater than zero, then the infinity notation will be used. This range is represented by $[0, \infty)$. Note that infinity can never be reached, so it is always exclusive. A constraint to any number less than zero, but not including zero would be on the range $(-\infty, 0)$.

Graphics and Rendering

This is an overview of the process of rendering. Do not worry if you do not understand everything right away; every step will be covered in lavish detail in later tutorials.

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.

Figure 8. An Image



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, windows look like windows, 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 empty shells. There are techniques that are used to allow you to cut open these empty shells, but this simply replaces part of the shell with another shell that shows what the inside looks like. Everything is a shell.

All of these shells are made of triangles. Even surfaces that appear to be round are merely triangles if you look closely enough. There are techniques that generate more triangles for objects that appear closer or larger, so that the viewer can almost never see the faceted silhouette of the object. But they are always made of triangles.

Note

Some rasterizers use planar quadrilaterals: four-sided objects, where all of the points lie in the same plane. One of the reasons that hardware-based rasterizers always use triangles is that all of the lines of a triangle are guaranteed to be in the same plane. Knowing this makes the rasterization process less complicated.

An object is 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*. These terms are used interchangeably.

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. This is one of the reasons why rasterization is so amenable to hardware acceleration: it operates on each triangle one at a time, in a specific order. Triangles can be fed into the top of the pipeline while triangles that were sent earlier can still be in some phase of rasterization.

The order in which triangles and the various meshes are submitted to the rasterizer can affect its output. Always remember that, no matter how you submit the triangular mesh data, the rasterizer will process each triangle in a specific order, drawing the next one only when the previous triangle has finished being drawn.

OpenGL 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

The rasterization pipeline, particularly for modern hardware, is very complex. This is a very simplified overview of this pipeline. It is necessary to have a simple understanding of the pipeline before we look at the details of rendering things with OpenGL. Those details can be overwhelming without a high level overview.

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 is called W. This last coordinate actually defines what the extents of clip space are for this vertex.

Clip space can actually be different for different vertices within a triangle. It is a region of 3D space on the range [-W, W] in each of the X, Y, and Z directions. So vertices with a different W coordinate are in a different clip space cube from other vertices. Since each vertex can have an independent W component, each vertex of a triangle exists in its own clip space.

In clip space, the positive X direction is to the right, the positive Y direction is up, and the positive Z direction is away from the viewer.

The process of transforming vertex positions into clip space is quite arbitrary. OpenGL provides a lot of flexibility in this step. We will cover this step in detail throughout the tutorials.

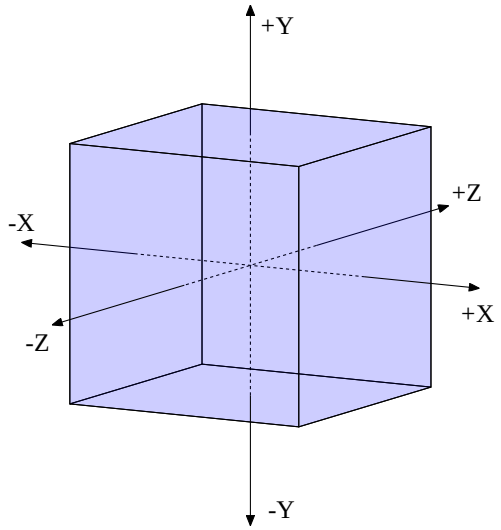
Because clip space is the visible transformed version of the world, any triangles that fall outside of this region are discarded. Any triangles that are partially outside of this region undergo a process called *clipping.* This breaks the triangle apart into a number of smaller triangles, such that the 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; we will cover that in a future tutorial.

Figure 9. Normalized Device Coordinate Space



The cube indicates the boundaries of normalized device coordinate space.

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.

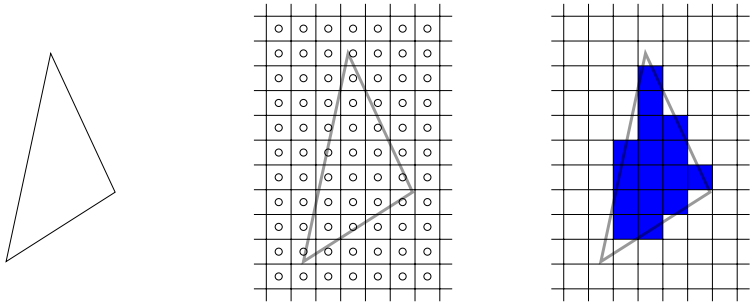
The bounds for Z are [0, 1], with 0 being the closest and 1 being the farthest. Vertex positions outside of this range are not visible.

Note that window coordinates have the bottom-left position as the (0, 0) origin point. This is counter to what users are used to in window coordinates, which is having the top-left position be the origin. There are transform tricks you can play to allow you to work in a top-left coordinate space if you need to.

The full details of this process will be discussed at length as the tutorials progress.

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.

Figure 10. Scan Converted Triangle

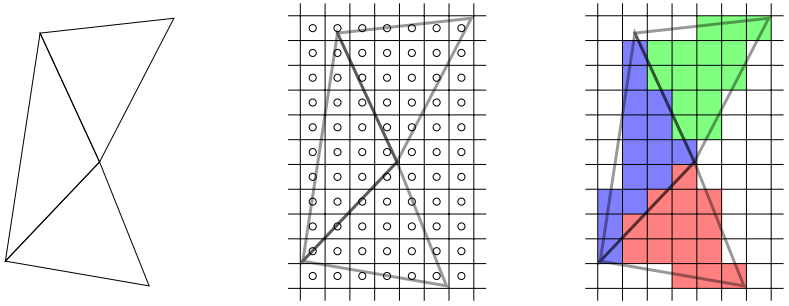


The center image shows the digital grid of output pixels; the circles represent the center of each pixel. The center of each pixel represents a *sample*: a discrete location within the area of a pixel. During scan conversion, a triangle will produce a *fragment* for every pixel sample that is within 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.

It is very often the case that triangles are rendered that share edges. OpenGL offers a guarantee that, so long as the shared edge vertex positions are *identical*, there will be no sample gaps during scan conversion.

Figure 11. Shared Edge Scan Conversion



To make it easier to use this, OpenGL also offers the guarantee that if you pass the same input vertex data through the same vertex processor, you will get identical output; this is called the *invariance guarantee*. So the onus is on the user to use the same input vertices in order to ensure gap-less scan conversion.

Scan conversion is an inherently 2D operation. This process only uses the X and Y position of the triangle in window coordinates to determine which fragments to generate. The Z value is not forgotten, but it is not directly part of the actual process of scan converting the triangle.

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. There may be other information that is part of a fragment, and we will expand on that in later tutorials.

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.

This phase is quite arbitrary. The user of OpenGL has a lot of options of how to decide what color to assign a fragment. We will cover this step in detail throughout the tutorials.

Direct3D Note

Direct3D prefers to call this stage “pixel processing” or “pixel shading”. This is a misnomer for several reasons. First, a pixel's final color can be composed of the results of multiple fragments generated by multiple *samples* within a single pixel. This is a common technique to remove jagged edges of triangles. Also, the fragment data has not been written to the image, so it is not a pixel yet. Indeed, the fragment processing step can conditionally prevent rendering of a fragment based on arbitrary computations. Thus a “pixel” in D3D parlance may never actually become a pixel at all.

Fragment Writing. After generating one or more colors and a depth value, the fragment is written to the destination image. This step involves more than simply writing to the destination image. Combining the color and depth with the colors that are currently in the image can involve a number of computations. These will be covered in detail in various tutorials.

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. Printed works tend to use CMYK (Cyan, Magenta, Yellow, Black). Since we're dealing with rendering to a screen, and because OpenGL requires it, we will use the RGB colorspace.

Note

You can play some fancy games with programmatic shaders (see below) that allow you to work in different colorspaces. So technically, we only have to output to a linear RGB colorspace.

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.

In term of rasterization as outlined above, there are several shader stages where arbitrary processing is both economical for performance and offers high utility to the user. For example, the transformation of an incoming vertex to clip space is a useful hook for user-defined code, as is the processing of a fragment into final colors and depth.

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. A downside of this is that they must live within certain limits that CPU code would not have to.

There are a number of shading languages available to various APIs. The one used in this tutorial 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.

What is OpenGL

Before we can begin looking into writing an OpenGL application, we must first know what it is that we are writing. What exactly is OpenGL?

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:

```
struct Object
{
    int count;
    float opacity;
    char *name;
};

//Create the storage for the object.
Object newObject;

//Put data into the object.
newObject.count = 5;
newObject.opacity = 0.4f;
newObject.name = "Some String";
```

In OpenGL, you would use an API that looks more like this:

```
//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 these 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 enumerators `GL_OBJECT_*` all name fields in the object that can be set. 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.

Note that all OpenGL objects are not as simple as this example, and the functions that change object state do not all follow these naming conventions. Also, exactly what it means to bind an object to the context is explained below.

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:

Example 1. OpenGL Object State

```
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 OpenGL Specification

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 [http://www.khronos.org/].

The specification is a very complicated and technical document. However, parts of it are quite readable, though you will usually need at least some understanding of what should be going on to understand it. If you try to read it, the most important thing to understand about it is this: 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. On Linux, things are... complicated.

The long and short of this is that if you are writing a program and it seems to be exhibiting off-spec behavior, that is the fault of the maker of your OpenGL implementation (assuming it is not a bug in your code). On Windows, the various graphics hardware makers put their OpenGL implementations in their regular drivers. So if you suspect a bug in their implementation, the first thing you should do is make sure your graphics drivers are up-to-date; the bug may have been corrected since the last time you updated your drivers.

OpenGL Versions. There are many versions of the OpenGL Specification. 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, and v3.1 removed most of those functions from the API[1]. 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.

Glossary

vector	A value composed of an ordered sequence of other values. The number of values stored in a vector is its dimensionality. Vectors can have math operations performed on them as a whole.
scalar	A single, non-vector value. A one-dimensional vector can be considered a scalar.
vector position	A vector that represents a position.
vector direction	A vector that represents a direction.
vector component	One of the values within a vector.
component-wise operation	An operation on a vector that applies something to each component of the vector. The results of a component-wise operation is a vector of the same dimension as the input(s) to the operation. Many vector operations are component-wise.
unit vector	A vector who's length is exactly one. These represent purely directional vectors.
vector normalization	The process of converting a vector into a unit vector that points in the same direction as the original vector.
pixel	The smallest division of a digital image. A pixel has a particular color in a particular colorspace.
image	A two-dimensional array of pixels.
rendering	The process of taking the source 3D world and converting it into a 2D image that represents a view of that world from a particular angle.
rasterization	A particular rendering method, used to convert a series of 3D triangles into a 2D image.
geometry, model, mesh	A single object in 3D space made of triangles.
vertex	One of the 3 elements that make up a triangle. Vertices can contain arbitrary of data, but among that data is a 3-dimensional position representing the location of the vertex in 3D space.
clip space, clip coordinates	A region of three-dimensional space into which vertex positions are transformed. These vertex positions are 4 dimensional quantities. 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.

[1]Deprecation only means marking those functions as to be removed in later functions. They are still available for use in 3.0.

	In clip space, positive X goes right, positive Y up, and positive Z away. Clip-space vertices are output by the vertex processing stage of the rendering pipeline.
clipping	The process of taking a triangle in clip coordinates and splitting it if one or more of its vertices is outside of clip space.
normalized device coordinates	These are clip coordinates that have been divided by their fourth component. This makes this range of space the same for all components. Vertices with positions on the range [-1, 1] are visible, and other vertices are not.
window space, window coordinates	A region of three-dimensional space that normalized device coordinates are mapped to. The X and Y positions of vertices in this space are relative to the destination image. The origin is in the bottom-left, with positive X going right and positive Y going up. The Z value is a number on the range [0, 1], where 0 is the closest value and 1 is the farthest. Vertex positions outside of this range are not visible.
scan conversion	The process of taking a triangle in window space and converting it into a number of fragments based on projecting it onto the pixels of the output image.
sample	A discrete location within the bounds of a pixel that determines whether to generate a fragment from scan converting the triangle. The area of a single pixel can have multiple samples, which can generate multiple fragments.
fragment	A single element of a scan converted triangle. A fragment can contain arbitrary data, but among that data is a 3-dimensional position, identifying the location on the triangle in window space where this fragment originates from.
invariance guarantee	A guarantee provided by OpenGL, such that if you provide binary-identical inputs to the vertex processing, while all other state remains exactly identical, then the exact same vertex in clip-space will be output.
colorspace	The set of reference colors that define a way of representing a color in computer graphics, and the function mapping between those reference colors and the actual colors. All colors are defined relative to a particular colorspace.
shader	A program designed to be executed by a renderer, in order to perform some user-defined operations.
shader stage	A particular place in a rendering pipeline where a shader can be executed to perform a computation. The results of this computation will be fed to the next stage in the rendering pipeline.
OpenGL	A specification that defines the effective behavior of a rasterization-based rendering system.
OpenGL context	A specific set of state used for rendering. The OpenGL context is like a large C-style struct that contains a large number of fields that can be accessed. If you were to create multiple windows for rendering, each one would have its own OpenGL context.
object binding	Objects can be bound to a particular location in the OpenGL context. When this happens, the state within the object takes the place of a certain set of state in the context. There are multiple binding points for objects, and each kind of object can be bound to certain binding points. Which bind point an object is bound to determines what state the object overrides.
Architectural Review Board	The body of the Khronos Group that governs the OpenGL specification.
OpenGL Implementation	The software that implements the OpenGL specification for a particular system.

Chapter 1. Hello, Triangle!

It is traditional for tutorials and introductory books on programming languages start with a program called “Hello, World!” This program is the simplest code necessary to print the text “Hello, World!” It serves as a good test to see that one's build system is functioning and that one can compile and execute code.

Using OpenGL to write actual text is rather involved. In lieu of text, our first tutorial will be drawing a single triangle to the screen.

Framework and FreeGLUT

The source to this tutorial, found in `Tut1 Hello Triangle/tut1.cpp`, is fairly simple. The project file that builds the final executable actually uses two source files: the tutorial file and a common framework file found in `framework/framework.cpp`. The framework file is where the actual initialization of FreeGLUT is done; it is also where main is. This file simply uses functions defined in the main tutorial file.

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.

The framework file expects 5 functions to be defined: `defaults`, `init`, `display`, `reshape`, and `keyboard`. The `defaults` function is called before FreeGLUT is initialized; it gives the tutorial the chance to modify the window size or the initialization parameters. The `init` function is called after OpenGL is initialized. This gives the tutorial file the opportunity to load what it needs into OpenGL before actual rendering takes place. The `reshape` function is called by FreeGLUT whenever the window is resized. This allows the tutorial to make whatever OpenGL calls are necessary to keep the window's size in sync with OpenGL. The `keyboard` function is called by FreeGLUT whenever the user presses a key. This gives the tutorial the chance to process some basic user input.

The `display` function is where the most important work happens. FreeGLUT will call this function when it detects that the screen needs to be rendered to.

Dissecting Display

The `display` function seems on the surface to be fairly simple. However, the functioning of it is fairly complicated and intertwined with the initialization done in the `init` function.

Example 1.1. The `display` Function

```
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();
```

Let us examine this code in detail.

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 `framework.cpp`, 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.

Following the Data

In the basic background section, we described the functioning of the OpenGL pipeline. We will now revisit this pipeline in the context of the code in tutorial 1. This will give us an understanding about the specifics of how OpenGL goes about rendering data.

Vertex Transfer

The first stage in the rasterization pipeline is transforming 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.

This is the data that we wish to transfer:

```
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.

The buffer object in the tutorial was created during initialization. Here is the code responsible for creating the buffer object:

Example 1.2. Buffer Object Initialization

```
void InitializeVertexBuffer()
{
    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 in the introduction [17], 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.

The fourth parameter is something we will look at in future tutorials.

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. Zero in this cases works a lot like the NULL pointer. This was not strictly necessary, as any later binds to this target will simply unbind what is already there. But unless you have very strict control over your rendering, it is usually a good idea to unbind the objects you bind.

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 the rendering code. That is the purpose of these lines:

```
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` is something we will explain in the next section. Without this function, the next one is unimportant.

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 tutorials. The first parameter is something we will look at in the next section.

One thing that appears absent is specifying which buffer object this data comes from. This is an implicit association rather than an explicit one. `glVertexAttribPointer` always refers to whatever buffer is bound to `GL_ARRAY_BUFFER` at the time that this function is called. Therefore it does not take a buffer object handle; it simply uses the handle we bound previously.

This function will be looked at in greater detail in later tutorials.

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.

The first parameter to `glDrawArrays` tells OpenGL that it is to take every 3 vertices that it gets as an independent triangle. Thus, it will read just 3 vertices and connect them to form a triangle.

Again, we will go into details in another tutorial.

Vertex Processing and Shaders

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 in this tutorial, so this involves the use of a *shader*.

A shader 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. A shader defines what its inputs and outputs are, and it is illegal for a shader to complete without writing to all of its outputs (in most cases).

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.

Shaders in OpenGL are written in the OpenGL Shading Language (GLSL). This language looks suspiciously like C, but it is very much not C. It has far too many limitations to be C (for example, recursion is forbidden). This is what our simple vertex shader looks like:

Example 1.3. Vertex Shader

```
#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*.

You might recognize something similar to the term "vertex attribute." For example, "glEnable*VertexAttrib*Array" or "gl*VertexAttrib*Pointer."

This is how data flows down the pipeline in OpenGL. 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. The connection between a particular call to `glVertexAttribPointer` and the string name of an input value to a vertex shader is somewhat complicated.

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;
```

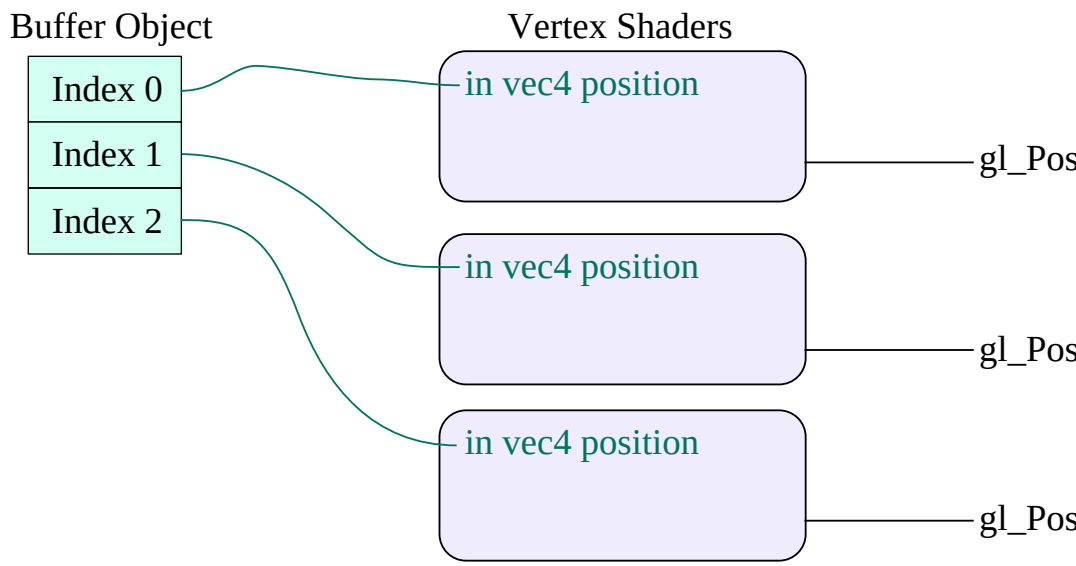
The layout location part assigns the attribute index of 0 to `position`. Attribute indices must be greater than or equal to zero, and there is a hardware-based limit on the number of attribute indices that can be in use at any one time[1].

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:

[1]For virtually all hardware since the beginning of commercial programmable hardware, this limit has been exactly 16. No more, no less.

Figure 1.1. Data Flow to 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`. The tutorial calls this function every time the window's size changes. Remember that the framework calls `reshape` whenever the window's size changes. So the tutorial's implementation of reshape is this:

Example 1.4. Reshaping Window

```
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.

Recall that 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.

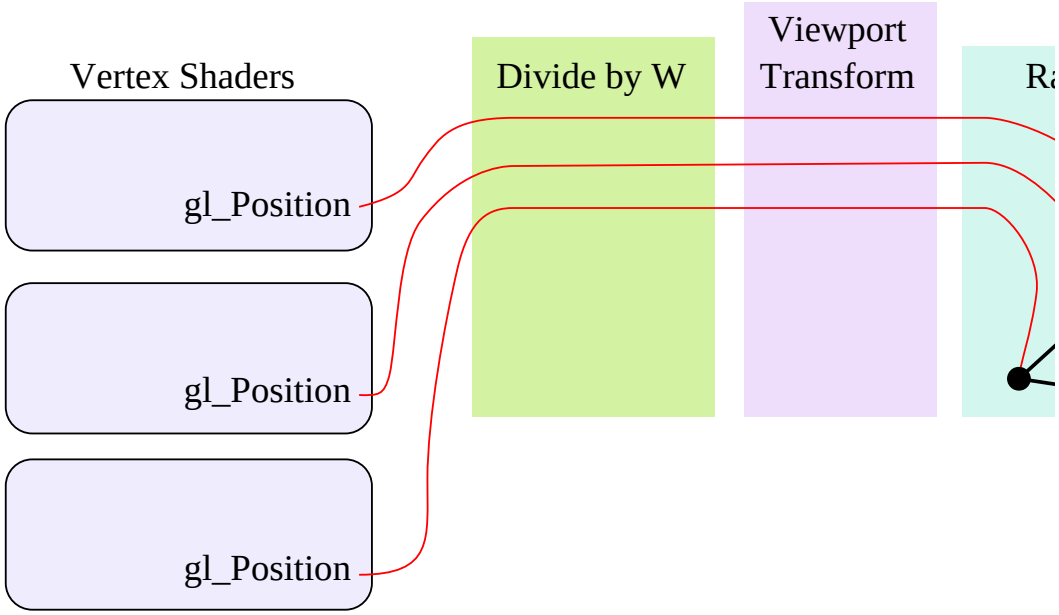
Once in window coordinates, OpenGL can now take these 3 vertices and scan-convert it into a series of fragments. In order to do this however, OpenGL must decide what the list of vertices represents.

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:

```
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.

Figure 1.2. Data Flow to Rasterizer



If we rendered 6 vertices, then we would get 2 triangles.

Fragment Processing

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, but we will get to that in later tutorials.

Our fragment shader looks like this:

Example 1.5. Fragment Shader

```
#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 tutorials.

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.

Making Shaders

We glossed over exactly how these text strings called shaders actually get sent to OpenGL. We will go into some detail on that now.

Note

If you are familiar with how shaders work in other APIs like Direct3D, that will not help you here. OpenGL shaders work very differently from the way they work in other APIs.

Shaders are written in a C-like language. So OpenGL uses a very C-like compilation model. In C, each individual .c file is compiled into an object file. Then, one or more object files are linked together into a single program (or static/shared library). OpenGL does something very similar.

A shader string is compiled into a *shader object*; this is analogous to an object file. One or more shader objects is linked into a *program object*.

A program object in OpenGL contains code for *all* of the shaders to be used for rendering. In the tutorial, we have a vertex and a fragment shader; both of these are linked together into a single program object. Building that program object is the responsibility of this code:

Example 1.6. Program Initialization

```
void InitializeProgram()
{
    std::vector<GLuint> shaderList;

    shaderList.push_back(CreateShader(GL_VERTEX_SHADER, strVertexShader));
```

```
    shaderList.push_back(CreateShader(GL_FRAGMENT_SHADER, strFragmentShader));

    theProgram = CreateProgram(shaderList);

    std::for_each(shaderList.begin(), shaderList.end(), glDeleteShader);
}
```

The first statement simply creates a list of the shader objects we intend to link together. The next two statements compile our two shader strings. The `CreateShader` function is a function defined by the tutorial that compiles a shader.

Compiling a shader into a shader object is a lot like compiling source code. Most important of all, it involves error checking. This is the implementation of `CreateShader`:

Example 1.7. Shader Creation

```
GLuint CreateShader(GLenum eShaderType, const std::string &strShaderFile)
{
    GLuint shader = glCreateShader(eShaderType);
    const char *strFileData = strShaderFile.c_str();
    glShaderSource(shader, 1, &strFileData, NULL);

    glCompileShader(shader);

    GLint status;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
    if (status == GL_FALSE)
    {
        GLint infoLogLength;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLogLength);

        GLchar *strInfoLog = new GLchar[infoLogLength + 1];
        glGetShaderInfoLog(shader, infoLogLength, NULL, strInfoLog);

        const char *strShaderType = NULL;
        switch(eShaderType)
        {
        case GL_VERTEX_SHADER: strShaderType = "vertex"; break;
        case GL_GEOMETRY_SHADER: strShaderType = "geometry"; break;
        case GL_FRAGMENT_SHADER: strShaderType = "fragment"; break;
        }

        fprintf(stderr, "Compile failure in %s shader:\n%s\n", strShaderType, strInfoLog);
        delete[] strInfoLog;
    }

 return shader;
}
```

An OpenGL shader object is, as the name suggests, an object. So the first step is to create the object with `glCreateShader`. This function creates a shader of a particular type (vertex or fragment), so it takes a parameter that tells what kind of object it creates. Since each shader stage has certain syntax rules and pre-defined variables and constants (thus making different shader stages different dialects of GLSL), the compiler must be told what shader stage is being compiled.

Note

Shader and program objects are objects in OpenGL. But they work rather differently from other kinds of OpenGL objects. For example, creating buffer objects, as shown above, uses a function of the form “glGen*” where * is “Buffer”. It takes a number of objects to create and a list to put those object handles in.

There are many other differences between shader/program objects and other kinds of OpenGL objects.

The next step is to actually compile the text shader into the object. The C-style string is retrieved from the C++ `std::string` object, and it is fed into the shader object with the `glShaderSource` function. The first parameter is the shader object to put the string into. The next parameter is the number of strings to put into the shader. Compiling multiple strings into a single shader object works analogously to compiling header files in C files. Except of course that the .c file explicitly lists the files it includes, while you must manually add them with `glShaderSource`.

The next parameter is an array of const char* strings. The last parameter is normally an array of lengths of the strings. We pass in `NULL`, which tells OpenGL to assume that the string is null-terminated. In general, unless you need to use the null character in a string, there is no need to use the last parameter.

Once the strings are in the object, they are compiled with `glCompileShader`, which does exactly what it says.

After compiling, we need to see if the compilation was successful. We do this by calling `glGetShaderiv` to retrieve the `GL_COMPILE_STATUS`. If this is `GL_FALSE`, then the shader failed to compile; otherwise compiling was successful.

If compilation fails, we do some error reporting. It prints a message to stderr that explains what failed to compile. It also prints an info log from OpenGL that describes the error; think of this log as the compiler output from a regular C compilation.

After creating both shader objects, we then pass them on to the `CreateProgram` function:

Example 1.8. Program Creation

```
GLuint CreateProgram(const std::vector<GLuint> &shaderList)
{
    GLuint program = glCreateProgram();

    for(size_t iLoop = 0; iLoop < shaderList.size(); iLoop++)
     glAttachShader(program, shaderList[iLoop]);

    glLinkProgram(program);

    GLint status;
    glGetProgramiv (program, GL_LINK_STATUS, &status);
    if (status == GL_FALSE)
    {
        GLint infoLogLength;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &infoLogLength);

        GLchar *strInfoLog = new GLchar[infoLogLength + 1];
        glGetProgramInfoLog(program, infoLogLength, NULL, strInfoLog);
        fprintf(stderr, "Linker failure: %s\n", strInfoLog);
        delete[] strInfoLog;
    }

    for(size_t iLoop = 0; iLoop < shaderList.size(); iLoop++)
        glDetachShader(program, shaderList[iLoop]);

    return program;
}
```

This function is fairly simple. It first creates an empty program object with `glCreateProgram`. This function takes no parameters; remember that program objects are a combination of *all* shader stages.

Next, it attaches each of the previously created shader objects to the programs, by calling the function `glAttachShader` in a loop over the `std::vector` of shader objects. The program does not need to be told what stage each shader object is for; the shader object itself remembers this.

Once all of the shader objects are attached, the code links the program with `glLinkProgram`. Similar to before, we must then fetch the linking status by calling `glGetProgramiv` with `GL_LINK_STATUS`. If it is GL_FALSE, then the linking failed and we print the linking log. Otherwise, we return the created program.

Note

In the above shaders, the attribute index for the vertex shader input `position` was assigned directly in the shader itself. There are other ways to assign attribute indices to attributes besides `layout(location = #)`. OpenGL will even assign an attribute index if you do not use any of them. Therefore, it is possible that you may not know the attribute index of an attribute. If you need to query the attribute index, you may call `glGetAttribLocation` with the program object and a string containing the attribute's name.

Once the program was successfully linked, the shader objects are removed from the program with `glDetachShader`. The program's linking status and functionality is not affected by the removal of the shaders. All it does is tell OpenGL that these objects are no longer associated with the program.

After the program has successfully linked, and the shader objects removed from the program, the shader objects are deleted using the C++ algorithm `std::for_each`. This line loops over each of the shaders in the list and calls `glDeleteShader` on them.

Using Programs. To tell OpenGL that rendering commands should use a particular program object, the `glUseProgram` function is called. In the tutorial this is called twice in the `display` function. It is called with the global `theProgram`, which tells OpenGL that we want to use that program for rendering until further notice. It is later called with 0, which tells OpenGL that no programs will be used for rendering.

Note

For the purposes of these tutorials, using program objects is *not* optional. OpenGL does have, in its compatibility profile, default rendering state that takes over when a program is not being used. We will not be using this, and you are encouraged to avoid its use as well.

Cleanup

The tutorial allocates a lot of OpenGL resources. It allocates a buffer object, which represents memory on the GPU. It creates two shader objects and a program object, all of which stored in memory owned by OpenGL. But it only deletes the shader objects; nothing else.

Part of this is due to the nature of FreeGLUT, which does not provide hooks for a cleanup function. But part of it is also due to the nature of OpenGL itself. In a simple example such as this, there is no need to delete anything. OpenGL will clean up its own assets when OpenGL is shut down as part of window deactivation.

It is generally good form to delete objects that you create before shutting down OpenGL. And you certainly should do it if you encapsulate objects in C++ objects, such that destructors will delete the OpenGL objects. But it is not strictly necessary.

In Review

In this tutorial, you have learned the following:

- Buffer objects are linear arrays of memory allocated by OpenGL. They can be used to store vertex data.
- GLSL shaders are compiled into shader objects that represent the code to be executed for a single shader stage. These shader objects can be linked together to produce a program object, which represent all of the shader code to be executed during rendering.
- The `glDrawArrays` function can be used to draw triangles, using particular buffer objects as sources for vertex data and the currently bound program object.

Further Study

Even with a simple tutorial like this, there are many things to play around with and investigate.

- Change the color value set by the fragment shader to different values. Use values in the range [0, 1], and then see what happens when you go outside that range.

- Change the positions of the vertex data. Keep position values in the [-1, 1] range, then see what happens when triangles go outside this range. Notice what happens when you change the Z value of the positions (note: nothing should happen while they're within the range). Keep W at 1.0 for now.
- Change the values that `reshape` gives to `glViewport`. Make them bigger or smaller than the window and see what happens. Shift them around to different quadrants within the window.
- Change the `reshape` function so that it respects aspect ratio. This means that the area rendered to may be smaller than the window area. Also, try to make it so that it always centers the area within the window.
- Change the clear color, using values in the range [0, 1]. Notice how this interacts with changes to the viewport above.
- Add another 3 vertices to the list, and change the number of vertices sent in the `glDrawArrays` call from 3 to 6. Add more and play with them.

OpenGL Functions of Note

glClearColor, glClear	These functions clear the current viewable area of the screen. `glClearColor` sets the color to clear, while `glClear` with the `GL_COLOR_BUFFER_BIT` value causes the image to be cleared with that color.
glGenBuffers, glBindBuffer, glBufferData	These functions are used to create and manipulate buffer objects. `glGenBuffers` creates one or more buffers, `glBindBuffer` attaches it to a location in the context, and `glBufferData` allocates memory and fills this memory with data from the user into the buffer object.
glEnableVertexAttribArray, glDisableVertexAttribArray, glVertexAttribPointer	These functions control vertex attribute arrays. `glEnableVertexAttribArray` activates the given attribute index, `glDisableVertexAttribArray` deactivates the given attribute index, and `glVertexAttribPointer` defines the format and source location (buffer object) of the vertex data.
glDrawArrays	This function initiates rendering, using the currently active vertex attributes and the current program object (among other state). It causes a number of vertices to be pulled from the attribute arrays in order.
glViewport	This function defines the current viewport transform. It defines as a region of the window, specified by the bottom-left position and a width/height.
glCreateShader, glShaderSource, glCompileShader, glDeleteShader	These functions create a working shader object. `glCreateShader` simply creates an empty shader object of a particular shader stage. `glShaderSource` sets strings into that object; multiple calls to this function simply overwrite the previously set strings. `glCompileShader` causes the shader object to be compiled with the previously set strings. `glDeleteShader` causes the shader object to be deleted.
glCreateProgram, glAttachShader, glLinkProgram, glDetachShader	These functions create a working program object. `glCreateProgram` creates an empty program object. `glAttachShader` attaches a shader object to that program. Multiple calls attach multiple shader objects. `glLinkProgram` links all of the previously attached shaders into a complete program. `glDetachShader` is used to remove a shader object from the program object; this does not affect the behavior of the program.
glUseProgram	This function causes the given program to become the current program. All rendering taking place after this call will use this program for the various shader stages. If the program 0 is given, then no program is current.
glGetAttribLocation	This function retrieves the attribute index of a named attribute. It takes the program to find the attribute in, and the name of the input variable of the vertex shader that the user is looking for the attribute index to.

Glossary

buffer object	An OpenGL object that represents a linear array of memory, containing arbitrary data. The contents of the buffer are defined by the user, but the memory is allocated by OpenGL. Data in buffer objects can be used for many purposes, including storing vertex data to be used when rendering.
input variable	A shader variable, declared at global scope. Input variables receive their values from earlier stages in the OpenGL rendering pipeline.

output variable
: A shader variable, declared at global scope, using the `out` keyword. Output variables written to by a shader are passed to later stages in the OpenGL rendering pipeline for processing.

vertex attribute
: Input variables to vertex shaders are called vertex attributes. Each vertex attribute is a vector of up to 4 elements in length. Vertex attributes are drawn from buffer objects; the connection between buffer object data and vertex inputs is made with the `glVertexAttribPointer` and `glEnableVertexAttribArray` functions. Each vertex attribute in a particular program object has an index; this index can be queried with `glGetAttribLocation`. The index is used by the various other vertex attribute functions to refer to that specific attribute.

attribute index
: Each input variable in a vertex shader must be assigned an index number. This number is used in code to refer to that particular attribute. This number is the attribute index.

viewport transform
: The process of transforming vertex data from normalized device coordinate space to window space. It specifies the viewable region of a window.

shader object
: An object in the OpenGL API that is used to compile shaders and represent the compiled shader's information. Each shader object is typed based on the shader stage that it contains data for.

program object
: An object in the OpenGL API that represents the full sequence of all shader processing to be used when rendering. Program objects can be queried for attribute locations and various other information about the program. They also contain some state that will be seen in later tutorials.

Chapter 2. Playing with Colors

This tutorial will show how to provide some color to the triangle from the previous tutorial. Instead of just giving the triangle a solid color, we will use two methods to provide it with varying color across its surface. The methods are to using the fragment's position to compute a color and to using per-vertex data to compute a color.

Fragment Position Display

As we stated in the overview, part of the fragment's data 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 is done in the Fragment Position tutorial, who's main file is `FragPosition.cpp`.

In this tutorial, and all future ones, shaders will be loaded from files instead of hard-coded strings in the .cpp file. To support this, the framework has the `Framework::LoadShader` and `Framework::CreateProgram` functions. These work similarly to the previous tutorial's `CreateShader` and `CreateProgram`, except that `LoadShader` takes a filename instead of a shader file.

The FragPosition tutorial loads two shaders, the vertex shader `data/FragPosition.vert` and the fragment shader `data/FragPosition.frag`. The vertex shader is identical to the one in the last tutorial. The fragment shader is very new, however:

Example 2.1. FragPosition's Fragment Shader

```
#version 330

out vec4 outputColor;

void main()
{
    float lerpValue = gl_FragCoord.y / 500.0f;

    outputColor = mix(vec4(1.0f, 1.0f, 1.0f, 1.0f),
        vec4(0.2f, 0.2f, 0.2f, 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. Recall that 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 500.0f is the height of the window (unless you resize 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.

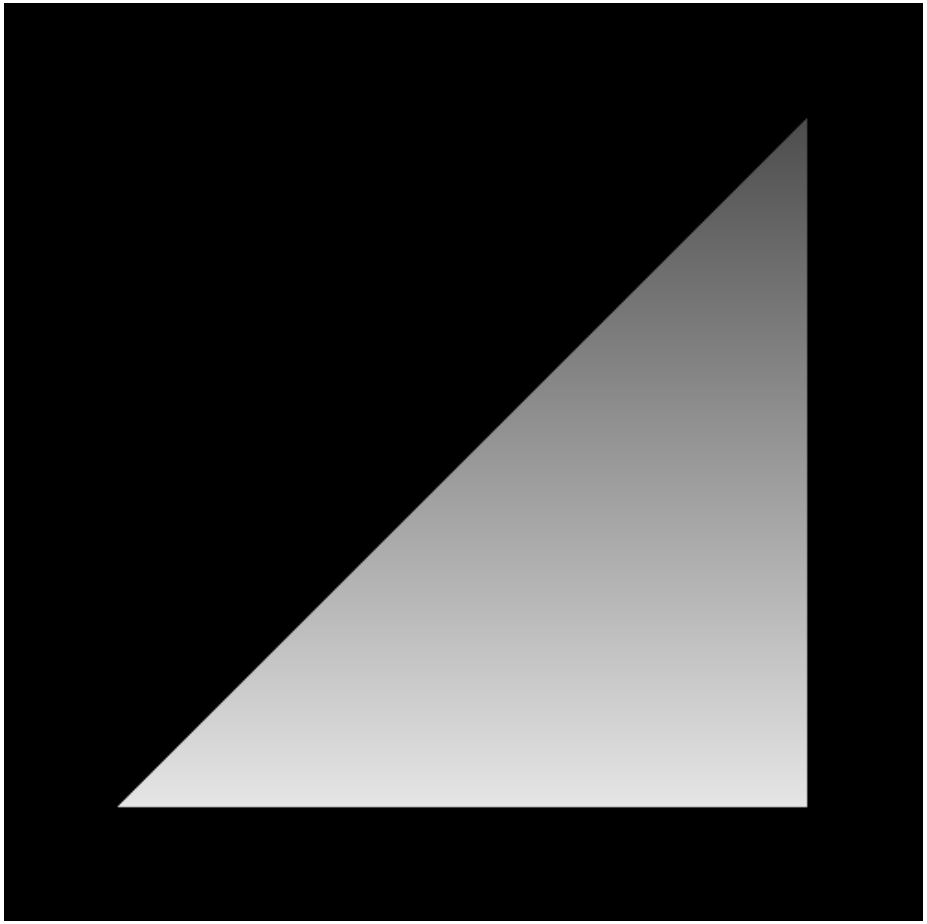
The `mix` function performs a linear interpolation. It will return exactly the first parameter if the third parameter is 0, and it will return exactly the second parameter if the third parameter is 1. If the third parameter is between 0 and 1, it will return a value between the two other parameters,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

Note

The third parameter to `mix` must be on the range [0, 1]. However, GLSL will not check this or do the clamping for you. If it is not on this range, the result of the `mix` function will be undefined. “Undefined” is the OpenGL shorthand for, “I do not know, but it is probably not what you want.”

We get the following image:

Figure 2.1. Fragment Position



In this case, the bottom of the triangle, the one closest to a Y of 0, will be the most white. While the top of the triangle, the parts closest to a Y of 500, will have the darkest color.

Other than the fragment shader, nothing much changes in the code.

Vertex Attributes

Using the fragment position in a fragment shader is quite useful, but it is far from the best tool for controlling the color of triangles. A much more useful tool is to give each vertex a color explicitly. The Vertex Colors tutorial implements this; the main file for this tutorial is `VertexColors.cpp`.

We want to affect the data being passed through the system. The sequence of events we want to happen is as follows.

1. For every position that we pass to a vertex shader, we want to pass a corresponding color value.

2. 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.

3. 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.

You most likely have some serious questions about that sequence of events, notably about how steps 2 and 3 could possibly work. We'll get to that. We will follow the revised flow of data through the OpenGL pipeline.

Multiple Vertex Arrays and Attributes

In order to accomplish the first step, we need to change our vertex array data. That data now looks like this:

Example 2.2. New Vertex Array Data

```
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,
};
```

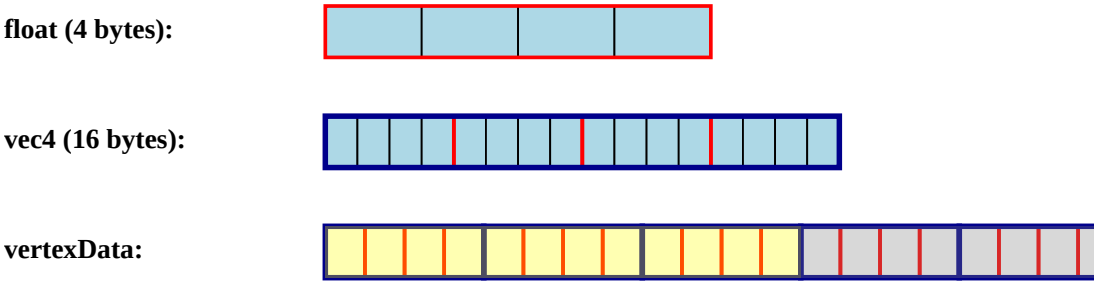
First, we need to understand what arrays of data look like at the lowest level. A single byte is the smallest addressible data in C/C++. A byte represents 8 bits (a bit can be 0 or 1), and it is a number on the range [0, 255]. A value of type float requires 4 bytes worth of storage. Any float value is stored in 4 consecutive bytes of memory.

A sequence of 4 floats, in GLSL parlance a vec4, is exactly that: a sequence of four floating-point values. Therefore, a `vec4` takes up 16 bytes, 4 floats times the size of a float.

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:

Figure 2.2. Vertex Array Memory Map



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]`

As with all vertex data, this is put into a buffer object. We have seen this code before:

Example 2.3. Buffer Object Initialization

```
void InitializeVertexBuffer()
{
    glGenBuffers(1, &vertexBufferObject);

    glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

The code has not changed, because the sizes of the array is computed by the compiler with the `sizeof` directive. Since we added a few elements to the buffer, the computed size naturally grows bigger.

Also, you may notice that the positions are different from prior tutorials. The original triangle, and the one that was used in the Fragment Position code, was a right triangle (one of the angles of the triangle is 90 degrees) that was isosceles (two of its three sides are the same length). This new triangle is an equilateral triangle (all three sides are the same length) centered at the origin.

Recall from above that we are sending two pieces of data per-vertex: a position and a color. We have two arrays, one for each piece of data. They may happen to be adjacent to one another in memory, but this changes nothing; there are two arrays of data. We need to tell OpenGL how to get each of these pieces of data.

This is done as follows:

Example 2.4. Rendering the Scene

```
void display()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    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();
    glutPostRedisplay();
}
```


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).

Note

If you're wondering why it is `(void*)48` and not just `48`, that is because of some legacy API cruft. The reason why the function name is glVertexAttrib“Pointer” is because the last parameter is technically a pointer to client memory. Or at least, it could be in the past. So we must explicitly cast the integer value 48 to a pointer type.

After this, we use `glDrawArrays` to render, then disable the arrays with `glDisableVertexAttribArray.`

Drawing in Detail

In the last tutorial, we skimmed over the details of what exactly `glDrawArrays` does. Let us take a closer look now.

The various attribute array functions set up arrays for OpenGL to read from when rendering. In our case here, we have two arrays. Each array has a buffer object and an offset into that buffer where the array begins, but the arrays do not have an explicit size. If we look at everything as C++ pseudo-code, what we have is this:

Example 2.5. Vertex Arrays

```
GLbyte *bufferObject = (void*){0.0f, 0.5f, 0.0f, 1.0f, 0.5f, -0.366f, ...};
float *positionAttribArray[4] = (float *[4])(&(bufferObject + 0));
float *colorAttribArray[4] = (float *[4])(&(bufferObject + 48));
```

Each element of the `positionAttribArray` contains 4 components, the size of our input to the vertex shader (vec4). This is the case because the second parameter of `glVertexAttribPointer` is 4. Each component is a floating-point number; similarly because the third parameter is `GL_FLOAT`. The array takes its data from `bufferObject` because this was the buffer object that was bound at the time that `glVertexAttribPointer` was called. And the offset from the beginning of the buffer object is 0 because that is the last parameter of `glVertexAttribPointer`.

The same goes for `colorAttribArray`, except for the offset value, which is 48 bytes.

Using the above pseudo-code representation of the vertex array data, `glDrawArrays` would be implemented as follows:

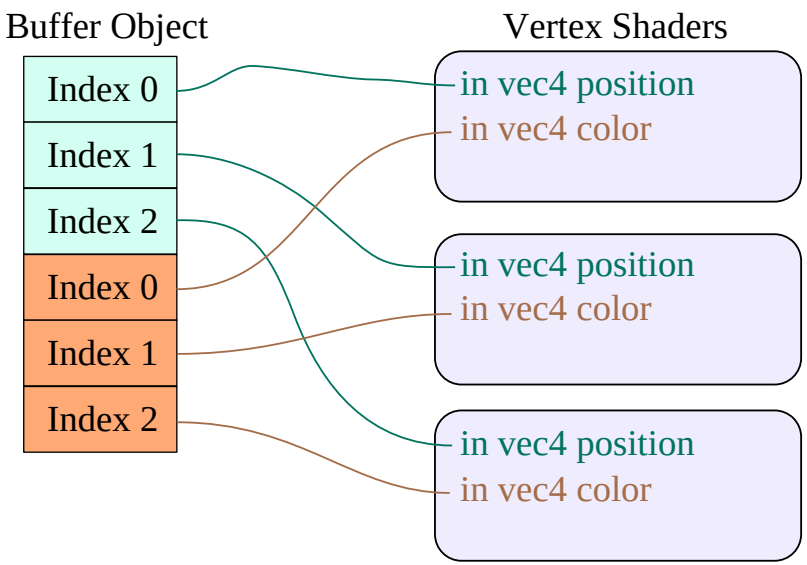
Example 2.6. Draw Arrays Implementation

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

This means that the vertex shader will be executed `count` times, and it will be given data beginning with the `start`-th element and continuing for `count` elements. So the first time the vertex shader gets run, it takes the position attribute from `bufferObject[0 + (0 * 4 * sizeof(float))]` and the color attribute from `bufferObject[48 + (0 * 4 * sizeof(float))]`. The second time pulls the position from `bufferObject[0 + (1 * 4 * sizeof(float))]` and color from `bufferObject[48 + (1 * 4 * sizeof(float))]`. And so on.

The data flow from the buffer object to the vertex shaders looks like this now:

Figure 2.3. Multiple Vertex Attributes



As before, every 3 vertices process is transformed into a triangle.

Vertex Shader

Our new vertex shader looks like this:

Example 2.7. Multi-input Vertex Shader

```
#version 330

layout (location = 0) in vec4 position;
layout (location = 1) in vec4 color;

smooth out vec4 theColor;

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

There are three new lines here. Let us take them one at a time.

The declaration of the global `color` defines a new input for the vertex shader. So this shader, in addition to taking an input named `position` also takes a second input named `color`. As with the `position` input, this tutorial assigns each attribute to an attribute index. `position` is assigned the attribute index 0, while `color` 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 `theColor` and is of type vec4.

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

Of course, this simply defines the output variable. In `main`, we actually write to it, assigning to it the value of `color` that was given as a vertex attribute. This being shader code, we could have used some other heuristic or arbitrary algorithm to compute the color. But for the purpose of this tutorial, it is simply passing the value of an attribute passed to the vertex shader.

Technically, the built-in variable `gl_Position` is defined as `out vec4 gl_Position`. So it is an output variable as well. It is a special output because this value is directly used by the system, rather than used only by shaders. User-defined outputs, like `theColor` above, have no intrinsic meaning to the system. They only have an effect in so far as other shader stages use them, as we will see next.

Fragment Program

The new fragment shader looks like this:

Example 2.8. Fragment Shader with Input

```
#version 330

smooth in vec4 theColor;

out vec4 outputColor;

void main()
{
    outputColor = theColor;
}
```

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

Now we come to the elephant in the room, so to speak. There is a basic communication problem.

See, our vertex shader is run only 3 times. This execution produces 3 output positions (`gl_Position`) and 3 output colors (`theColor`). 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. An equilateral triangle the length of who's sides are 1 has an area of ~0.433. The total screen area (on the range [-1, 1] in X and Y) is 4, so our triangle covers approximately one-tenth of the screen. The window's natural resolution is 500x500 pixels. 500*500 is 250,000 pixels; one-tenth of this is 25,000. So our fragment shader gets executed about 25,000 times.

There's a slight disparity here. If the vertex shader is directly communicating with the fragment shader, and the vertex shader is outputting only 3 total color values, where do the other 24,997 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. There are two other alternatives: `noperspective` and `flat`.

If you were to modify the tutorial and change `smooth` to `noperspective`, you would see no change. That does not mean a change did not happen; our example is just too simple for there to actually be a change. The difference between `smooth` and `noperspective` is somewhat subtle, and only matters once we start using more concrete examples. We will discuss this difference later.

The `flat` interpolation 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`."

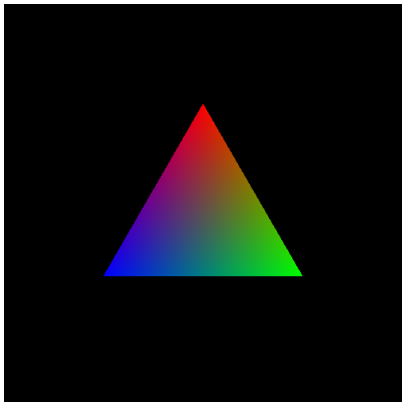
Each rasterized triangle has its own set of 3 outputs that are interpolated to compute the value for the fragments created by that triangle. So if you render 2 triangles, the interpolated values from one triangle do not directly affect the interpolated values from another triangle. Thus, each triangle can be taken independently from the rest.

It is possible, and highly desirable in many cases, to build multiple triangles from shared vertices and vertex data. But we will discuss this at a later time.

The Final Image

When you run the tutorial, you will get the following image.

Figure 2.4. Interpolated Vertex Colors



The colors at each tip of the triangle are the pure red, green, and blue colors. They blend together towards the center of the triangle.

In Review

In this tutorial, you have learned the following:

- Data is passed to vertex shaders via buffer objects and attribute arrays. This data is processed into triangles.
- The `gl_FragCoord` built-in GLSL variable can be used in fragment shaders to get the window-space coordinates of the current fragment.
- Data can be passed from a vertex shader to a fragment shader, using output variables in the vertex shader and corresponding input variables in the fragment shader.
- Data passed from the vertex shader to the fragment shader is interpolated across the surface of the triangle, based on the interpolation qualifier used for the output and input variables in the vertex and fragment shaders respectively.

Further Study

Here are some ideas to play around with.

- Change the viewport in the FragPosition tutorial. Put the viewport in the top half of the display, and then put it in the bottom half. See how this affects the shading on the triangle.
- Combine the FragPosition tutorial with the Vertex Color tutorial. Use interpolated color from the vertex shader and multiply that with the value computed based on the screen-space position of the fragment.

GLSL Functions of Note

```
vec mix(vec initial, vec final, float alpha);
```

Performs a linear interpolation between *`initial`*, *`final`*, based on *`alpha`*. An *`alpha`* value of 0 means that the *`inital`* value is returned, while an *`alpha`* of 1 means the *`final`* value is returned. The vec type means that the parameter can be a vector or float. All occurrences of vec must be the same in a particular function call, however, so *`initial`* and *`final`* must have the same type.

The *`alpha`* value can be either a scalar or a vector of the same length as *`initial`* and *`final`*. If it is a scalar, then all of the components of the two values are interpolated by that scalar. If it is a vector, then the components of *`initial`* and *`final`* are interpolated by their corresponding components of *`alpha`*.

If *`alpha`*, or any component of *`alpha`*, is outside of the range [0, 1], then the return value of this function is undefined.

Glossary

fragment interpolation	This is the process of taking 3 corresponding vertex shader outputs and interpolating them across the surface of the triangle. For each fragment generated, there will also be an interpolated value generated for each of the vertex shader's outputs (except for certain built-in outputs, like `gl_Position`.) The way that the interpolation is handled depends on the *interpolation qualifier* on the vertex output and fragment input.
interpolation qualifier	A GLSL keyword assigned to outputs of vertex shaders and the corresponding inputs of fragment shaders. It determines how the three values of the triangle are interpolated across that triangle's surface. The qualifier used on the vertex shader output must match with the one used on the fragment shader input of the same name. Valid interpolation qualifiers are `smooth`, `flat`, and `noperspective`.

Part II. Positioning

Vertex positions are perhaps the most important part of a vertex's information. The only data that a vertex shader must produce is the computation of a clip-space vertex position; everything else is user-defined.

Computing proper vertex positions can turn an assemblage of triangles into something that resembles a real object. This section of the book will detail how to make objects move around, as well as presenting them as a three-dimensional object. It covers how to manipulate vertex positions through a series of spaces, to allow for concepts like a change of viewer orientation. And it covers how to position and orient objects arbitrarily, to achieve many different kinds of movement and animation.

Chapter 3. OpenGL's Moving Triangle

This tutorial is about how to move objects around. It will introduce new shader techniques.

Moving the Vertices

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 tutorials, 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. This is what `cpuPositionOffset.cpp` does.

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:

Example 3.1. Computation of Position Offsets

```
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;
}
```

This function computes offsets in a loop. The offsets produce circular motion, and the offsets will reach the beginning of the circle every 5 seconds (controlled by `fLoopDuration`). The function `glutGet(GLUT_ELAPSED_TIME)` retrieves the integer time in milliseconds since the application started. The `fmodf` function computes the floating-point modulus of the time. In lay terms, it takes the first parameter and returns the remainder of the division between that 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:

Example 3.2. Adjusting the Vertex Data

```
void AdjustVertexData(float fXOffset, float fYOffset)
{
    std::vector<float> fNewData(ARRAY_COUNT(vertexPositions));
    memcpy(&fNewData[0], vertexPositions, sizeof(vertexPositions));

    for(int iVertex = 0; iVertex < ARRAY_COUNT(vertexPositions); iVertex += 4)
    {
        fNewData[iVertex] += fXOffset;
        fNewData[iVertex + 1] += fYOffset;
    }

    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObject);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertexPositions), &fNewData[0]);
```

```
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

This 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.

Think of `glBufferData` as a combination of `malloc` and `memcpy`, while glBufferSubData is just `memcpy`.

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.

The last line of the function is simply unbinding the buffer object. It is not strictly necessary, but it is good form to clean up binds after making them.

Buffer Object Usage Hints. Every time we draw something, 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 tutorial changed the allocation of the buffer object slightly, replacing:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, GL_STATIC_DRAW);
```

with this:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, 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. We will see more of these hints later.

The rendering function now has become this:

Example 3.3. Updating and Drawing the Vertex Data

```
void display()
{
    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, positionBufferObject);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

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



```
    glDisableVertexAttribArray(0);
    glUseProgram(0);

    glutSwapBuffers();
    glutPostRedisplay();
}
```

The first three lines get the offset and set the vertex data. Everything but the last line is unchanged from the first tutorial. The last line of the function is there to tell FreeGLUT to constantly call `display`. Ordinarily, `display` would only be called when the window's size changes or when the window is uncovered. `glutPostRedisplay` causes FreeGLUT to call `display` again. Not immediately, but reasonably fast.

If you run the tutorial, you will see a smaller triangle (the size was reduced in this tutorial) that slides around in a circle.

A Better Way

This is fine for a 3-vertex example. But imagine a scene involving millions of vertices (and no, that's not an exaggeration for high-end applications). Moving objects this way means having to copy millions of vertices from the original vertex data, add an offset to each of them, and then upload that data to an OpenGL buffer object. And all of that is *before* rendering. Clearly there must be a better way; games can not possibly do this every frame and still hold decent framerates.

Actually for quite some time, they did. In the pre-GeForce 256 days, that was how all games worked. Graphics hardware just took a list of vertices in clip space and rasterized them into fragments and pixels. Granted, in those days, we were talking about maybe 10,000 triangles per frame. And while CPUs have come a long way since then, they have not scaled with the complexity of graphics scenes.

The GeForce 256 (note: not a GT 2xx card, but the very first GeForce card) was the first graphics card that actually did some from of vertex processing. It could store vertices in GPU memory, read them, do some kind of transformation on them, and then send them through the rest of the pipeline. The kinds of transformations that the old GeForce 256 could do were quite useful, but fairly simple.

Having the benefit of modern hardware and OpenGL 3.x, we have something far more flexible: vertex shaders.

Remember what it is that we are doing. We compute an offset. Then we apply that offset to each vertex position. Vertex shaders are given each vertex position. So it makes sense to simply give the vertex shader the offset and let it compute the final vertex position, since it operates on each vertex. This is what `vertPositionOffset.cpp` does.

The vertex shader is found in `data\positionOffset.vert`. The vertex shader used here is as follows:

Example 3.4. Offsetting Vertex Shader

```
#version 330

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

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

Shaders and Granularity. Recall that, with each execution of a shader, the shader gets new values for variables defined as `in`. Each time a vertex shader is called, it gets a different set of inputs from the vertex attribute arrays and buffers. That is useful for vertex position data, but it is not what we want for the offset. We want each vertex to use the *same* offset; a “uniform” offset, if you will.

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 tutorial, this is done in the `InitializeProgram` 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`. Thus, the rendering code looks like this:

Example 3.5. Draw with Calculated Offsets

```
void display()
{
    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, positionBufferObject);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray(0);
    glUseProgram(0);

    glutSwapBuffers();
    glutPostRedisplay();
}
```

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.

Standard OpenGL Nomenclature

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 basic `glUniform` parameters are just the uniform location, so `glUniform2f` takes a uniform location and two values.

The type of values it takes is defined by the second part. The possible type names and their associated types are:

b	signed byte	ub	unsigned byte
s	signed short	us	unsigned short
i	signed int	ui	unisgned int
f	float	d	double

Note that OpenGL has special typedefs for all of these types. GLfloat is a typedef for float, for example. This is not particularly useful for simple types, but for integer types, they can be very useful.

Not all functions that take multiple types take all possible types. For example, `glUniform` only comes in `i`, `ui`, and `f` variations (OpenGL 4.0 introduces `d` variants).

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. So the function `glUniform2fv` takes a uniform location, a number of entries, and an array of that many entries. Each entry is *two floats* in size; the number of entries is not the number of floats or other types. It is the number of values divided by the number of values per entry. Since “2f” represents two floats, that is how many values are in each entry.

Some OpenGL functions just take a type without a number. These functions take a fixed number of parameters of that type, usually just one.

Vector Math. You may be curious about how these lines work:

```
vec4 totalOffset = vec4(offset.x, offset.y, 0.0, 0.0);
gl_Position = position + totalOffset;
```

The vec4 that looks like a function here is a constructor; it creates a vec4 from 4 floats. This is done to make the addition easier.

The addition of `position` to `totalOffset` is a component-wise addition. It is a shorthand way of doing this:

```
gl_Position.x = position.x + totalOffset.x;
gl_Position.y = position.y + totalOffset.y;
gl_Position.z = position.z + totalOffset.z;
gl_Position.w = position.w + totalOffset.w;
```

GLSL has a lot of vector math built in. The math operations are all component-wise when applied to vectors. However, it is illegal to add vectors of different dimensions to each other. So you cannot have a vec2 + vec4. That is why we had to convert `offset` to a vec4 before performing the addition.

More Power to the Shaders

It's all well and good that we are no longer having to transform vertices manually. But perhaps we can move more things to the vertex shader. Could it be possible to move all of `ComputePositionOffsets` to the vertex shader?

Well, no. The call to `glutGet(GL_ELAPSED_TIME)` cannot be moved there, since GLSL code cannot directly call C/C++ functions. But everything else can be moved. This is what `vertCalcOffset.cpp` does.

The vertex program is found in `data\calcOffset.vert`.

Example 3.6. Offset Computing Vertex Shader

```
#version 330

layout(location = 0) in vec4 position;
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 = position + 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`. We saw `mix` in the last tutorial. And these are just the tip of the iceberg; there are a *lot* of standard GLSL functions available.

The rendering code looks quite similar to the previous rendering code:

Example 3.7. Rendering with Time

```
void display()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(theProgram);

    glUniform1f(elapsedTimeUniform, glutGet(GLUT_ELAPSED_TIME) / 1000.0f);

    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();
    glutPostRedisplay();
}
```


This time, we do not need any code to use the elapsed time; we simply pass it unmodified to the shader.

You may be wondering exactly how it is that the `loopDuration` uniform gets set. This is done in our shader initialization routine, and it is done only once:

Example 3.8. Loop Duration Setting

```
void InitializeProgram()
{
    std::vector<GLuint> shaderList;

    shaderList.push_back(Framework::LoadShader(GL_VERTEX_SHADER, "calcOffset.vert"));
    shaderList.push_back(Framework::LoadShader(GL_FRAGMENT_SHADER, "standard.frag"));

    theProgram = Framework::CreateProgram(shaderList);

    elapsedTimeUniform = glGetUniformLocation(theProgram, "time");

    GLuint loopDurationUnf = glGetUniformLocation(theProgram, "loopDuration");
    glUseProgram(theProgram);
    glUniform1f(loopDurationUnf, 5.0f);
    glUseProgram(0);
}
```

We get the time uniform as normal with `glGetUniformLocation`. For the loop duration, we get that in a local variable. Then we immediately set the current program object, set the uniform to a value, and then unset the current program object.

Program objects, like all objects that contain internal state, will retain their state unless you explicitly change it. So the value of `loopDuration` will be 5.0f in perpetuity; we do not need to set it every frame.

Multiple Shaders

Well, 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. And this is what `fragChangeColor.cpp` does.

The fragment shader in this tutorial is loaded from the file `data\calcColor.frag`:

Example 3.9. Time-based Fragment Shader

```
#version 330

out vec4 outputColor;

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;

    outputColor = mix(firstColor, secondColor, currLerp);
}
```

This function is similar to the periodic loop in the vertex shader (which did not change from the last time we saw it). Instead of using sin/cos functions to compute the coordinates of a circle, 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.

Here is the program initialization code:

Example 3.10. More Shader Creation

```
void InitializeProgram()
{
    std::vector<GLuint> shaderList;

    shaderList.push_back(Framework::LoadShader(GL_VERTEX_SHADER, "calcOffset.vert"));
    shaderList.push_back(Framework::LoadShader(GL_FRAGMENT_SHADER, "calcColor.frag"));

    theProgram = Framework::CreateProgram(shaderList);

    elapsedTimeUniform = glGetUniformLocation(theProgram, "time");

    GLuint loopDurationUnf = glGetUniformLocation(theProgram, "loopDuration");
    GLuint fragLoopDurUnf = glGetUniformLocation(theProgram, "fragLoopDuration");


    glUseProgram(theProgram);
    glUniform1f(loopDurationUnf, 5.0f);
    glUniform1f(fragLoopDurUnf, 10.0f);
    glUseProgram(0);
}
```

As before, we get the uniform locations for `time` and `loopDuration`, as well as the new `fragLoopDuration`. We then set the two loop durations for the program.

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 tutorial, 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.

In any case, because of this, the rendering code is unchanged. The time uniform is updated each frame with FreeGLUT's elapsed time.

Globals in shaders. Variables at global scope in GLSL can be defined with certain storage qualifiers: `const`, `uniform`, `in`, and `out`. A `const` value works like it does in C99 and C++: the value does not change, period. It must have an initializer. An unqualified variable works like one would expect in C/C++; it is a global value that can be changed. GLSL shaders can call functions, and globals can be shared between functions. However, unlike `in`, `out`, and `uniforms`, non-const and `const` variables are *not* shared between stages.

On Vertex Shader Performance

These tutorials are simple and should run fast enough, but it is still important to look at the performance implications of various operations. In this tutorial, we present 3 ways of moving vertex data: transform it yourself on the CPU and upload it to buffer objects, generate transform parameters on the CPU and have the vertex shader use them to do the transform, and put as much as possible in the vertex shader and only have the CPU provide the most basic parameters. Which is the best to use?

This is not an easy question to answer. However, it is almost always the case that CPU transformations will be slower than doing it on the GPU. The only time it will not be is if you need to do the exact same transformations many times within the same frame. And even then, it is better to do the transformations once on the GPU and save the result of that in a buffer object that you will pull from later. This is called transform feedback, and it will be covered in a later tutorial.

Between the other two methods, which is better really depends on the specific case. Take our example. In one case, we compute the offset on the CPU and pass it to the GPU. The GPU applies the offset to each vertex position. In the other case, we simply provide a time parameter, and for every vertex, the GPU must compute the *exact same* offset. This means that the vertex shader is doing a lot of work that all comes out to the same number.

Even so, that does not mean it's always slower. What matters is the overhead of changing data. Changing a uniform takes time; changing a vector uniform typically takes no more time than changing a single float, due to the way that many cards handle floating-point math. The question is this: what is the cost of doing more complex operations in a vertex shader vs. how often those operations need to be done.

The second vertex shader we use, the one that computes the offset itself, does a lot of complex math. Sine and cosine values are not particularly fast to compute. They require quite a few computations to calculate. And since the offset itself does not change for each vertex in a single rendering call, performance-wise it would be best to compute the offset on the CPU and pass the offset as a uniform value.

And typically, that is how rendering is done much of the time. Vertex shaders are given transformation values that are pre-computed on the CPU. But this does not mean that this is the only or best way to do this. In some cases, it is often useful to compute the offsets via parameterized values passed to the vertex shader.

This is best done when vertex shader inputs are abstracted away. That is, rather than passing a position, the user passes more general information, and the shader generates the position at a particular time or some other parameter. This can be done for particle systems based on forces; the vertex shader executes the force functions based on time, and is able to thus compute the location of the particle at an arbitrary time.

This also has an advantage that we have seen. By passing high-level information to the shader and letting it do complex math, you can affect much more than just a simple offset. The color animation in the fragment shader would not have been possible with just an offset. High-level parameterization gives shaders a great deal of freedom.

In Review

In this tutorial, you have learned about the following:

- Buffer object contents can be updated partially with the `glBufferSubData` function. This function performs the equivalent of a `memcpy` operation.
- Uniform variables in shaders are variables that are set by code outside of GLSL. They only change between rendering calls, so they are uniform over the surface of any particular triangle.
- Uniform variable values are stored with the program object. This state is preserved until it is explicitly changed.
- Uniform variables defined in two GLSL stages that have the same name and type are considered the same uniform. Setting this uniform in the program object will change its value for both stages.

Further Study

There are several things you can test to see what happens with these tutorials.

- With `vertCalcOffset.cpp`, change it so that it draws two triangles moving in a circle, with one a half `loopDuration` ahead of the other. Simply change the uniforms after the `glDrawArrays` call and then make the `glDrawArrays` call again. Add half of the loop duration to the time before setting it the second time.
- In `fragChangeColor.cpp`, change it so that the fragment program bounces between `firstColor` and `secondColor`, rather than popping from `secondColor` back to first at the end of a loop. The first-to-second-to-first transition should all happen within a single `fragLoopDuration` time interval. In case you are wondering, GLSL supports the `if` statement, as well as the ?: operator. For bonus points however, do it without an explicit conditional statement; feel free to use a sin or cos function to do this.

- Using our knowledge of uniforms, go back to Tutorial 2's FragPosition tutorial. Modify the code so that it takes a uniform that describes the window's height, rather than using a hard-coded value. Change the `reshape` function to bind the program and modify the uniform with the new height.

OpenGL Functions of Note

glBufferSubData — This function copies memory from the user's memory address into a buffer object. This function takes a byte offset into the buffer object to begin copying, as well as a number of bytes to copy.

When this function returns control to the user, you are free to immediately deallocate the memory you owned. So you can allocate and fill a piece of memory, call this function, and immediately free that memory with no hazardous side effects. OpenGL will not store the pointer or make use of it later.

glGetUniformLocation — This function retrieves the location of a uniform of the given name from the given program object. If that uniform does not exist or was not considered in use by GLSL, then this function returns -1, which is not a valid uniform location.

glUniform* — Sets the given uniform in the program currently in use (set by `glUseProgram`) to the given value. This is not merely one function, but an entire suite of functions that take different types.

GLSL Functions of Note

```
vec mod(vec numerator, float denominator);
```

The `mod` function takes the modulus of the *`numerator`* by the *`denominator`*. The modulus can be thought of as a way of causing a loop; the return value will be on the range [0, *`denominator`*) in a looping fashion. Mathematically, it is defined as *`numerator`* - (*`denominator`* * `FLOOR`(*`numerator`* / *`denominator`*)), where `FLOOR` rounds a floating-point value down towards the smallest whole number.

The type vec can be either float or any vector type. It must be the same type for all parameters. If a vector denominator is used, then the modulus is taken for each corresponding component. The function returns a vector of the same size as its *`numerator`* type.

```
vec cos(vec angle);
```

```
vec sin(vec angle);
```

Returns the trigonometric cosine or sine [http://en.wikipedia.org/wiki/Sine#Sine.2C_cosine_and_tangent], respectively, of the given *`angle`*. The *`angle`* is given in units of radians. If the *`angle`* is a vector, then the returned vector will be of the same size, and will be the cosine or sine of each component of the *`angle`* vector.

Glossary

uniforms — These are a class of global variable that can be defined in GLSL shaders. They represent values that are uniform (unchanging) over the course of a rendering operation. Their values are set from outside of the shader, and they cannot be changed from within a shader.

Chapter 4. Objects at Rest

Thus far, we have seen very flat things. Namely, a single triangle. Maybe the triangle moved around or had some colors.

This tutorial is all about how to create a realistic world of objects.

The Unreal World

The Orthographic Cube tutorial renders a rectangular prism (a 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 tutorial 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.

The vertex shader is a combination of things we know. It passes a color through to the fragment stage, but it also takes a vec2 offset uniform that it adds an offset to the X and Y components of the position. The fragment shader simply takes the interpolated color and uses it as the output color.

Face Culling

There is one very noteworthy code change, however: the initialization routine. It has a few new functions that need to be discussed.

Example 4.1. Face Culling Initialization

```
void init()
{
    InitializeProgram();
    InitializeVertexBuffer();

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

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

The last three lines are new.

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.

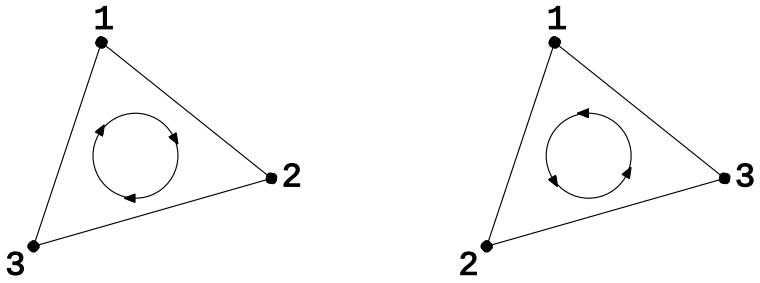
Face culling is a useful feature for saving performance. Take our rectangular prism, for example. Pick up a remote control; their general shape is that of a rectangular prism. No matter how you look at it or orient it, you can never see more than 3 sides of it at once. So why bother spending all that 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*.

Figure 4.1. Triangle 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.

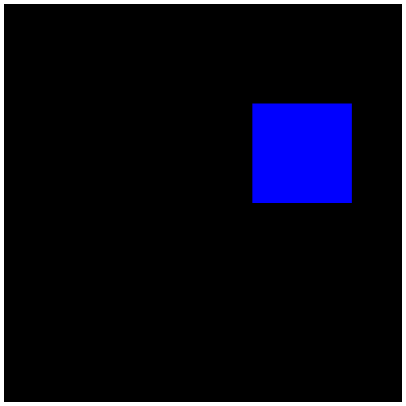
The `glCullFace` function defines which side, front or back, gets culled. This can be given `GL_BACK`, `GL_FRONT`, or `GL_FRONT_AND_BACK`. The latter culls *everything*, so no triangles are rendered. This can be useful for measuring vertex shader performance but is less useful for actually drawing anything.

The triangle data in the tutorial is specifically ordered so that the clockwise facing of the triangles face out. This prevents the drawing of the rear-facing faces.

Lack of Perspective

So, the image looks like this:

Figure 4.2. Orthographic Prism



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, 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. That something is called "perspective."

Perspective Projection

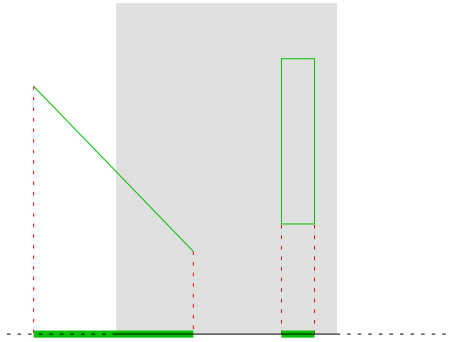
Recall that our destination image, the screen, is just a two dimensional array of pixels. The 3D rendering pipeline we are using defines transformations of vertex positions that go from clip-space to window space. Once the positions are in window space, 2D triangles are rendered.

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.

Finite projections, which are the ones we are interested in, only project objects onto a finite space of the lower dimensionality. For a 3D to 2D projection, there is a finite plane on which the world is projected. For 2D to 1D, there is a bounded line that is the result of the projection.

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.

Figure 4.3. 2D to 1D Orthographic Projection



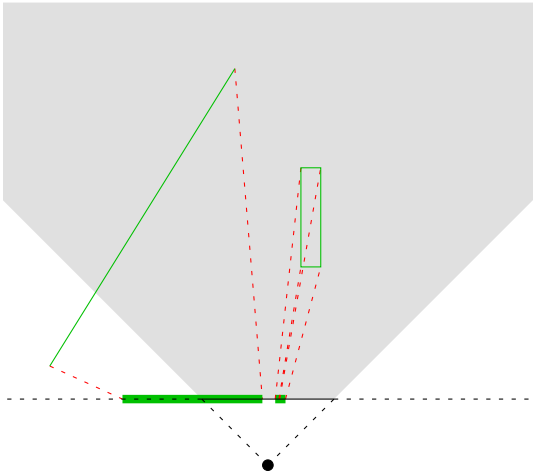
A scene orthographically projected onto the black line. The gray box represents the part of the world that is visible to the projection; parts of the scene outside of this region are not seen.

When projecting onto an arbitrary line, the math is a bit more complicated. But what makes it an orthographic projection is that the dimension perpendicular to the surface is negated uniformly to create the projection. The fact that it is a projection in the direction of the perpendicular and that it is uniform is what makes it orthographic.

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:

Figure 4.4. 2D to 1D Perspective Projection

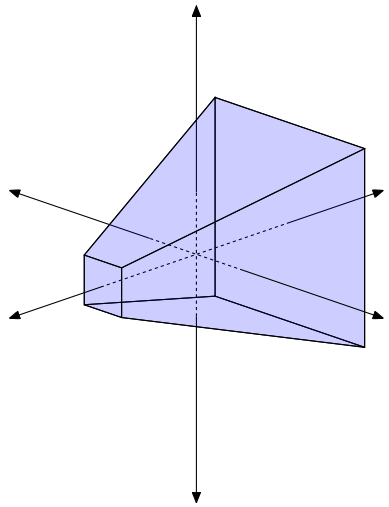


As you can see, the projection is radial, based on the location of a particular point. That point is the eye or camera of the projection.

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.

Figure 4.5. Viewing Frustum



Mathematical Perspective

Now that we know what we want to do, we just need to know how to do it.

We will be making a few simplifying assumptions:

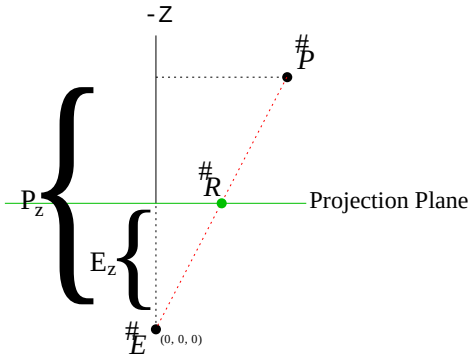
- The plane of projection is axis-aligned and faces down the -Z axis. Thus, -Z is farther away from the plane of projection.
- The eye point is fixed at the origin (0, 0, 0).
- The size of the plane of projection will be [-1, 1]. All points that project outside of this range are not drawn.

Yes, this sounds somewhat like normalized device coordinate space. No, that's not a coincidence. But let's not get ahead of ourselves.

We know a few things about how the projection results will work. 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. Here is the equivalent form in a 2D to 1D perspective projection.

Figure 4.6. 2D to 1D Perspective Projection Diagram



The projection of the point P onto the projection plane. This plane is at an offset of E_z compared to the eye point, which is fixed at the origin. R is the projected point.

What we have are two similar right triangles: the triangle formed by E, R and E_z, and the triangle formed by E, P, and P_z. We have the eye position and the position of the unprojected point. To find the location of R, we simply do this:

Equation 4.1. Perspective Computation

$$\#R = \#P\left(\frac{E_z}{P_z}\right)$$

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.

The Perspective Divide

The basic perspective projection function is simple. Really simple. Indeed, it is so simple that it has been built into graphics hardware since the days of the earliest 3Dfx card and even prior graphics hardware.

You might notice that the scaling can be expressed as a division operation (multiplying by the reciprocal). And you may recall that the difference between clip space and normalized device coordinate space is a division by the W coordinate. So instead of doing the divide in the shader, we can simply set the W coordinate of each vertex correctly and let the hardware handle it.

This step, the conversion from clip-space to normalized device coordinate space, has a particular name: the *perspective divide*. So named because it is usually used for perspective projections; orthographic projections tend to have the W coordinates be 1.0, thus making the perspective divide a no-op.

Note

You may be wondering why this arbitrary division-by-W step exists. You may also be wondering, in this modern days of vertex shaders that can do vector divisions very quickly, why we should bother to use the hardware division-by-W step at all. There are several reasons.

One we will cover in just a bit when we deal with matrices. More important ones will be covered in future tutorials. Suffice it to say that there are very good reasons to put the perspective term in the W coordinate of clip space vertices.

Camera Perspective

Before we can actually implement perspective projection, we need to deal with a new issue. The orthographic projection transform was essentially a no-op. It is automatic, by the nature of how OpenGL uses the clip space vertices output by the vertex shader. The perspective projection transform is a bit more involved. Indeed, it fundamentally changes the nature of the world.

Our vertex positions before now have been stored directly in clip space. We did on occasion add an offset to the positions to move them to more convenient locations. But for all intents and purposes, the position values stored in the buffer object are exactly what our vertex shader outputs: clip space positions.

Recall that the divide-by-W is part of the OpenGL-defined transform from clip space positions to NDC positions. Perspective projection defines a process for transforming positions *into* clip space, such that these clip space positions will appear to be a perspective projection of a 3D world. This transformation has well-defined outputs: clip space positions. But what exactly are its *input* values?

We therefore define a new space for positions; let us call this space *camera space*.[1] This is not a space that OpenGL recognizes (unlike clip-space which is explicitly defined by GL); it is purely an arbitrary user construction. The definition of camera space will affect the exact process of perspective projection, since that projection must produce proper clip space positions. Therefore, it can be useful to define camera space based on what we know of the general process of perspective projection. This minimizes the differences between camera space and the perspective form of clip space, and it can simplify our perspective projection logic.

The volume of camera space will range from positive infinity to negative infinity in all directions. Positive X extends right, positive Y extends up, and positive Z is *forward*. The last one is a change from clip space, where positive Z is away.

Our perspective projection transform will be specific to this space. As previously stated, the projection plane shall be a region [-1, 1] in the X and Y axes, and at a Z value of -1. The projection will be from vertices in the -Z direction onto this plane; vertices that have a positive Z value are behind the projection plane.

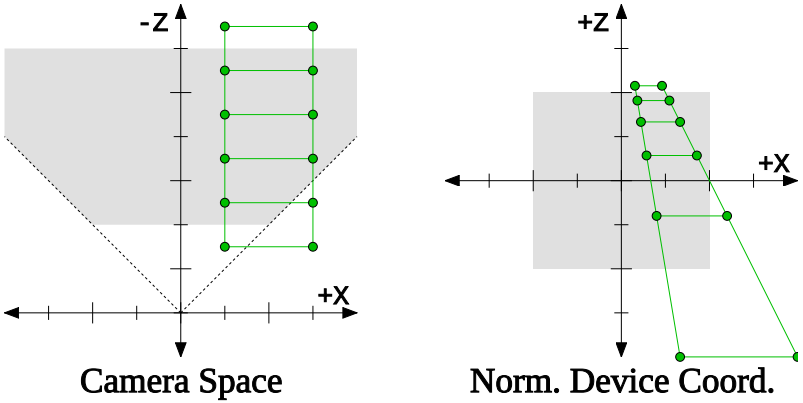
Now, we will make one more simplifying assumption: the location of the center of the perspective plane is fixed at (0, 0, -1) in camera space. Therefore, since the projection plane is pointing down the -Z axis, eye's location relative to the plane of projection is (0, 0, -1). Thus, the E_z value, the offset from the projection plane to the eye, is always -1. This means that our perspective term, when phrased as division rather than multiplication, is simply P_z/-1: the negation of the camera-space Z coordinate.

Having a fixed eye position and projection plane makes it difficult to have zoom-in/zoom-out style effects. This would normally be done by moving the plane relative to the fixed eye point. There is a way to do this, however. All you need to do is, when transforming from camera space to clip space, scale all of the X and Y values by a constant. What this does is make the world, as the camera sees it, smaller or larger in the X and Y axes. It effectively makes the frustum wider or narrower.

To compare, camera space and normalized device coordinate space (after the perspective divide) look like this, using a 2D version of a perspective projection:

[1] The reason it is called “camera” space will be discussed later.

Figure 4.7. Camera to NDC Transformation in 2D



Do note that this diagram has the Z axis flipped from camera space and normalized device coordinate (NDC) space. This is because camera space and NDC space have different viewing directions. In camera space, the camera looks down the -Z axis; more negative Z values are farther away. In NDC space, the camera looks down the +Z axis; more positive Z values are farther away. The diagram flips the axis so that the viewing direction can remain the same between the two images (up is away).

If you perform an orthographic projection from NDC space on the right (by dropping the Z coordinate), then what you get is a perspective projection of the world on the left. In effect, what we have done is transform objects into a three-dimensional space from which an orthographic projection will look like a perspective one.

Perspective in Depth

So we know what to do with the X and Y coordinates. But what does the Z value mean in a perspective projection?

Until the next tutorial, we are going to ignore the *meaning* of Z. Even so, we still need some kind of transform for it; if a vertex extends outside of the [-1, 1] box in any axis in normalized device coordinate (NDC) space, then it is outside of the viewing area. And because the Z coordinate undergoes the perspective divide just like the X and Y coordinates, we need to take this into account if we actually want to see anything in our projection.

Our W coordinate will be based on the camera-space Z coordinate. We need to map Z values from the camera-space range $[0, -\infty)$ to the NDC space range [-1, 1]. Since camera space is an infinite range and we're trying to map to a finite range, we need to do some range bounding. The frustum is already finitely bound in the X and Y directions; we simply need to add a Z boundary.

The maximum distance that a vertex can be before it is considered no longer in view is the *camera zFar*. We also have a minimum distance from the eye; this is called the *camera zNear*. This creates a finite frustum for our camera space viewpoint.

Note

It is very important to remember that these are the zNear and zFar for the *camera* space. The next tutorial will also introduce a range of depth, also using the names zNear and zFar. This is a related but fundamentally different range.

The camera zNear can appear to effectively determine the offset between the eye and the projection plane. However, this is not the case. Even if zNear is less than 1, which would place the near Z plane *behind* the projection plane, you still get an effectively valid projection. Objects behind the plane can be projected onto the plane just as well as those in front of it; it is still a perspective projection. Mathematically, this works.

What it does *not* do is what you would expect if you moved the plane of projection. Since the plane of projection has a fixed size (the range [-1, 1]), moving the plane would alter where points appear in the projection. Changing the camera zNear does not affect the X, Y position of points in the projection.

There are several ways to go about mapping one finite range to another. One confounding problem is the perspective divide itself; it is easy to perform a linear mapping between two finite spaces. It is quite another to do a mapping that remains linear *after* the perspective divide. Since we will be dividing by -Z itself (the camera-space Z, not the clip-space Z), the math is much more complex than you might expect.

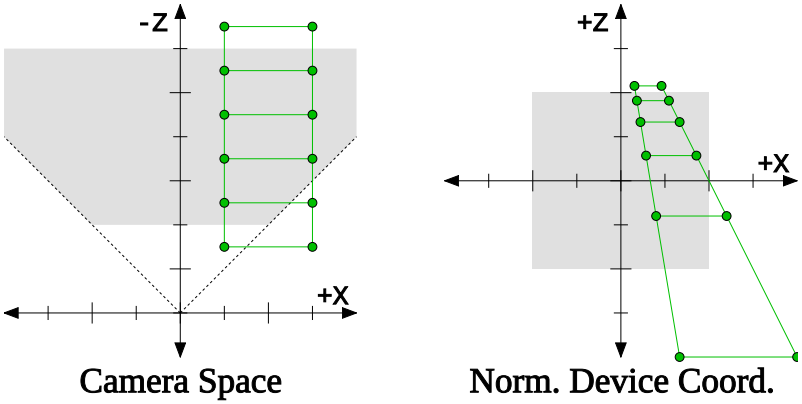
For reasons that will be better explained in the next tutorial, we will use this modestly complicated function to compute the clip-space Z:

Equation 4.2. Depth Computation

$$N = \text{zNear}$$
$$F = \text{zFar}$$
$$Z_{clip} = \frac{Z_{camera}(F + N)}{N - F} + \frac{2NF}{N - F}$$

Some important things about this equation and camera zNear/zFar. First, these values are *positive*; the equation accounts for this when performing the transformation. Also, zNear *cannot* be 0; it can be very close to zero, but it must never be exactly zero.

Let us review the previous diagram of camera-to-NDC transformation in 2D space:



The example of 2D camera-space vs. 2D NDC space uses this equation to compute the Z values. Take a careful look at how the Z coordinates match. The Z distances are evenly spaced in camera space, but in NDC space, they are non-linearly distributed. And yet simultaneously, points that are colinear in camera-space remain colinear in NDC space.

This fact has some interesting properties that we will investigate further in the next tutorial.

Drawing in Perspective

Given all of the above, we now have a specific sequence of steps to transform a vertex from camera space to clip space. These steps are as follows:

1. Frustum adjustment: multiply the X and Y value of the camera-space vertices by a constant.

2. Depth adjustment: modify the Z value from camera space to clip space, as above.

3. Perspective division term: compute the W value, where E_z is -1.

Now that we have all the theory down, we are ready to put things properly in perspective. This is done in the ShaderPerspective tutorial.

Our new vertex shader, `data\ManualPerspective.vert` looks like this:

Example 4.2. ManualPerspective Vertex Shader

```
#version 330

layout(location = 0) in vec4 position;
layout(location = 1) in vec4 color;

smooth out vec4 theColor;

uniform vec2 offset;
uniform float zNear;
uniform float zFar;
uniform float frustumScale;

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

    clipPos.xy = cameraPos.xy * frustumScale;

    clipPos.z = cameraPos.z * (zNear + zFar) / (zNear - zFar);
    clipPos.z += 2 * zNear * zFar / (zNear - zFar);

    clipPos.w = -cameraPos.z;

    gl_Position = clipPos;
    theColor = color;
}
```

We have a few new uniforms, but the code itself is only modestly complex.

The first statement simply applies an offset, just like the vertex shaders we have seen before. It positions the object in camera space, so that it is offset from the center of the view. This is here to make it easier to position the object for projection.

The next statement performs a scalar multiply of the camera-space X and Y positions, storing them in a temporary 4-dimensional vector. From there, we compute the clip Z position based on the formula discussed earlier.

The W coordinate of the clip space position is the Z distance in camera space divided by the Z distance from the plane (at the origin) to the eye. The eye is fixed at (0, 0, -1), so this leaves us with the negation of the camera space Z position. OpenGL will automatically perform the division for us.

After that, we simply store the clip space position where OpenGL needs it, store the color, and we're done. The fragment shader is unchanged.

With all of the new uniforms, our program initialization routine has changed:

Example 4.3. Program Initialization

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

frustumScaleUnif = glGetUniformLocation(theProgram, "frustumScale");
zNearUnif = glGetUniformLocation(theProgram, "zNear");
zFarUnif = glGetUniformLocation(theProgram, "zFar");

glUseProgram(theProgram);
```



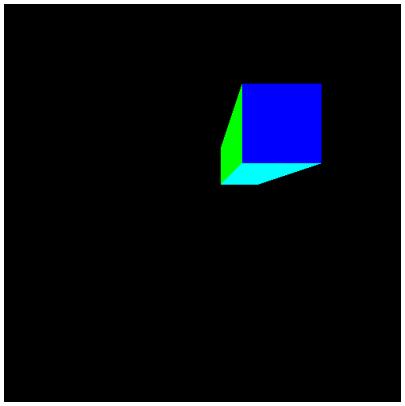
```
glUniform1f(frustumScaleUnif, 1.0f);
glUniform1f(zNearUnif, 1.0f);
glUniform1f(zFarUnif, 3.0f);
glUseProgram(0);
```

We only set the new uniforms once. The scale of 1.0 means effectively no change. We define the Z to go from -1 to -3 (remember that, in our Z equation, the zNear and zFar are positive but refer to negative values).

The location of the prism has also changed. In the original tutorial, it was located on the 0.75 range in Z. Because camera space has a very different Z from clip space, this had to change. Now, the Z location of the prism is between -1.25 and -2.75.

All of this leaves us with this result:

Figure 4.8. Perspective Prism



Now, it looks like a rectangular prism. A bright, colorful, unrealistic one.

Vector Math

We glossed over something in the vertex shader that bears more discussion. Namely, this line:

```
clipPos.xy = cameraPos.xy * frustumScale;
```

Even if you are familiar with vector math libraries in other languages, this code should be rather odd. Traditional vector libraries allow you to write selectors like `vec.x` and `vec.w` in order to get a specific field from a vector. So what does something like `vec.xy` mean?

Well, it means the obvious; this expression returns a 2D vector (vec2), since there are only two components mentioned (X and Y). This vector will have its first component come from the X component of `vec` and the second component come from the Y component of `vec`. This kind of selection is called, in GLSL parlance, *swizzle selection*. The size of the returned vector will be the number of components you mention, and the order of these components will dictate the order of the components returned.

You can do any kind of swizzle operation on a vector, so long as you keep in mind the following rules:

- You cannot select components that are not in the source vector. So if you have:

  ```
  vec2 theVec;
  ```

  You cannot do `theVec.zz` because it has no Z component.

- You cannot select more than 4 components.

These are the only rules. So you can have a vec2 that you swizzle to create a vec4 (`vec.yyyx`); you can repeat components; etc. Anything goes so long as you stick to those rules.

You should also assume that swizzling is fast. This is not true of most CPU-based vector hardware, but since the earliest days of programmable GPUs, swizzle selection has been a prominent feature. In the early programmable days, swizzles caused *no* performance loss; in all likelihood, this has not changed.

Swizzle selection can also be used on the left side of the equals, as we have done here. It allows you to set specific components of a vector without changing the other components.

When you multiply a vector by a scalar (non-vector value), it does a component-wise multiply, returning a vector containing the scalar multiplied by each of the components of the vector. We could have written the above line as follows:

```
clipPos.x = cameraPos.x * frustumScale;
clipPos.y = cameraPos.y * frustumScale;
```

But it probably would not be as fast as the swizzle and vector math version.

The Matrix has You

So, now that we can put the world into perspective, let's do it the right way. The “needlessly overcomplicated for the time being but will make sense in a few tutorials” way.

First, let us look at the system of equations used to compute clip coordinates from camera space. Given that `S` is the frustum scale factor, `N` is the zNear and `F` is the zFar, we get the following four equations.

Equation 4.3. Camera to Clip Equations

$$
\begin{aligned}
X_{clip} &= SX_{camera} \\
Y_{clip} &= SY_{camera} \\
Z_{clip} &= \frac{F+N}{N-F}Z_{camera} + \frac{2FN}{N-F} \\
W_{clip} &= -Z_{camera}
\end{aligned}
$$

The odd spacing is intentional. For laughs, let's add a bunch of meaningless terms that do not change the equation, but starts to develop an interesting pattern:

Equation 4.4. Camera to Clip Expanded Equations

$$
\begin{aligned}
X_{clip} &= (S)X_{camera} + (0)Y_{camera} + (0)Z_{camera} + (0)W_{camera} \\
Y_{clip} &= (0)X_{camera} + (S)Y_{camera} + (0)Z_{camera} + (0)W_{camera} \\
Z_{clip} &= (0)X_{camera} + (0)Y_{camera} + \left(\frac{F+N}{N-F}\right)Z_{camera} + \left(\frac{2FN}{N-F}\right)W_{camera} \\
W_{clip} &= (0)X_{camera} + (0)Y_{camera} + (-1)Z_{camera} + (0)W_{camera}
\end{aligned}
$$

What we have here is what is known as a linear system of equations. The equations can be specified as a series of coefficients (the numbers being multiplied by the XYZW values) which are multiplied by the input values (XYZW) to produce the single output. Each individual output

value is a linear combination of all of the input values. In our case, there just happen to be a lot of zero coefficients, so the output values in this particular case only depend on a few input values.

You may be wondering at the multiplication of the additive term of Z_{clip}'s value by the camera space W. Well, our input camera space position's W coordinate is always 1. So performing the multiplication is valid, so long as this continues to be the case. Being able to do what we are about to do is part of the reason why the W coordinate exists in our camera-space position values (the perspective divide is the other).

We can re-express any linear system of equations using a special kind of formulation. You may recognize this reformulation, depending on your knowledge of linear algebra:

Equation 4.5. Camera to Clip Matrix Transformation

$$
\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}_{clip} = \begin{bmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & \frac{F+N}{N-F} & \frac{2FN}{N-F} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}_{camera}
$$

The two long vertical columns of XYZW labeled “clip” and “camera” are 4-dimensional vectors; namely the clip and camera space position vectors. The larger block of numbers is a matrix. You probably are not familiar with matrix math. If not, it will be explained presently.

Generically speaking, a *matrix* is a two dimensional block of numbers (matrices with more than 2 dimensions are called “tensors”). Matrices are very common in computer graphics. Thus far, we have been able to get along without them. As we get into more detailed object transformations however, we will rely more and more on matrices to simplify matters.

In graphics work, we typically use 4x4 matrices; that is, matrices with 4 columns and 4 rows respectively. This is due to the nature of graphics work: most of the things that we want to use matrices for are either 3 dimensional or 3 dimensional with an extra coordinate of data. Our 4D positions are just 3D positions with a 1 added to the end.

The operation depicted above is a vector-matrix multiplication. A matrix of dimension `nxm` can only be multiplied by a vector of dimension `n`. The result of such a multiplication is a vector of dimension `m`. 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.

Matrix multiplication does what the expanded equation example does. For every row in the matrix, the values of each component of the column are multiplied by the corresponding values in the rows of the vector. These values are then added together; that becomes the single value for the row of the output vector.

Equation 4.6. Vector Matrix Multiplication

$$
\begin{bmatrix} m11 & m21 & m31 & m41 \\ m12 & m22 & m32 & m42 \\ m13 & m23 & m33 & m43 \\ m14 & m24 & m34 & m44 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x*m11 + y*m21 + z*m31 + w*m41 \\ x*m12 + y*m22 + z*m32 + w*m42 \\ x*m13 + y*m23 + z*m33 + w*m43 \\ x*m14 + y*m24 + z*m34 + w*m44 \end{bmatrix}
$$

This results ultimately in performing 16 floating-point multiplications and 12 floating-point additions. That's quite a lot, particularly compared with our current version. Fortunately, graphics hardware is designed to make these operations very fast. Because each of the multiplications are independent of each other, they could all be done simultaneously, which is exactly the kind of thing graphics hardware does fast. Similarly, the addition operations are partially independent; each row's summation does not depend on the values from any other row. Ultimately, vector-matrix multiplication usually generates only 4 instructions in the GPU's machine language.

We can re-implement the above perspective projection using matrix math rather than explicit math. The MatrixPerspective tutorial does this.

The vertex shader is much simpler in this case:

Example 4.4. MatrixPerspective Vertex Shader

```
#version 330

layout(location = 0) in vec4 position;
layout(location = 1) in vec4 color;

smooth out vec4 theColor;

uniform vec2 offset;
uniform mat4 perspectiveMatrix;

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

    gl_Position = perspectiveMatrix * cameraPos;
    theColor = color;
}
```

The OpenGL Shading Language (GLSL), being designed for graphics operations, naturally has matrices as basic types. The mat4 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.

Note that the shader no longer computes the values on its own; it is *given* a matrix with all of the stored values as a uniform. This is simply because there is no need for it. All of the objects in a particular scene will be rendered with the same perspective matrix, so there is no need to waste potentially precious vertex shader time doing redundant computations.

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.

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*v is not the same thing as v*M. Normally vectors are considered 1xN matrices (where N is the size of the vector). When you multiply vectors on the left of the matrix, GLSL considers it an Nx1 matrix; this is the only way to make the multiplication make sense. This will multiply the single row of the vector with each column, 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.

The program initialization routine has a few changes:

Example 4.5. Program Initialization of Perspective Matrix

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

perspectiveMatrixUnif = glGetUniformLocation(theProgram, "perspectiveMatrix");

float fFrustumScale = 1.0f; float fzNear = 0.5f; float fzFar = 3.0f;

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

theMatrix[0] = fFrustumScale;
theMatrix[5] = fFrustumScale;
theMatrix[10] = (fzFar + fzNear) / (fzNear - fzFar);
theMatrix[14] = (2 * fzFar * fzNear) / (fzNear - fzFar);
```



```
theMatrix[11] = -1.0f;

glUseProgram(theProgram);
glUniformMatrix4fv(perspectiveMatrixUnif, 1, GL_FALSE, theMatrix);
glUseProgram(0);
```

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. This works because IEEE 32-bit floating-point numbers represent a zero as 4 bytes that all contain zero.

The next few functions set the particular values of interest into the matrix. Before we can understand what's going on here, we need to talk a bit about ordering.

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 NxM matrix (columns x rows), the first N values in the array are the first column (top-to-bottom), the next N values are the second column, and so forth. In row-major order, the first M values in the array are the first row (left-to-right), followed by another M values for the second row, and so forth.

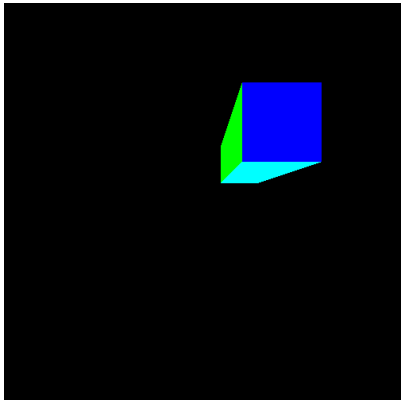
In this example, the matrix is stored in column-major order. So array index 14 is in the third row of the fourth column.

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. Since our data is column-major, we set it to `GL_FALSE`. The last parameter is the matrix data itself.

Running this program will give us:

Figure 4.9. Perspective Matrix

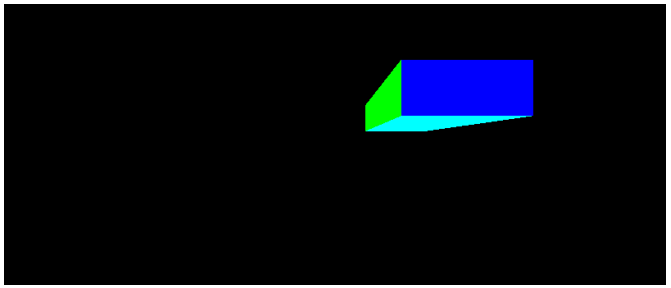


The same thing we had before. Only now done with matrices.

Aspect of the World

If you run the last program, and resize the window, the viewport resizes with it. Unfortunately, this also means that what was once a rectangular prism with a square front becomes elongated.

Figure 4.10. Bad Aspect Ratio



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.

What exactly can be done about this? Well, that depends on what you intend to accomplish by making the window bigger.

One simple way to do this is to prevent the viewport from ever becoming non-square. This can be done easily enough by changing the `reshape` function to be this:

Example 4.6. Square-only Viewport

```
void reshape (int w, int h)
{
    if(w < h)
        glViewport(0, 0, (GLsizei) w, (GLsizei) w);
    else
        glViewport(0, 0, (GLsizei) h, (GLsizei) h);
}
```

Now if you resize the window, the viewport will always remain a square. However, if the window is non-square, there will be a lot of empty space either to the right or below the viewport area. This space cannot be rendered into with triangle drawing commands (for reasons that we will see in the next tutorial).

This solution has the virtue of keeping the viewable region of the world fixed, regardless of the shape of the viewport. It has the disadvantage of wasting window space.

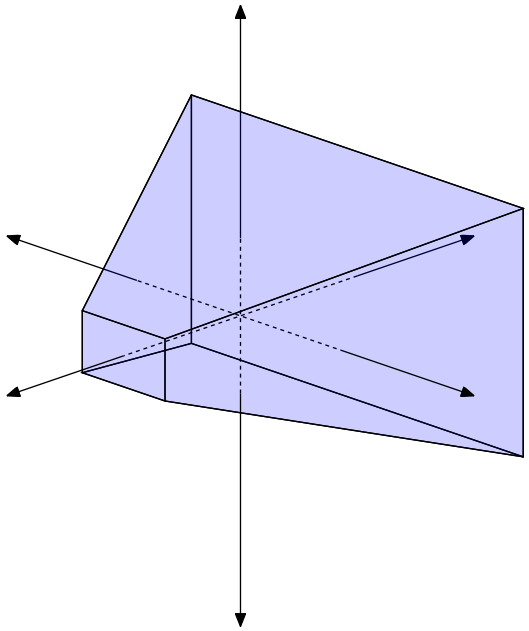
What do we do if we want to use as much of the window as possible? There is a way to do this.

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.*

So our problem is with the implicit assumption that squares in camera space need to remain squares throughout. This is not the case. To do what we want, we need to transform things into clip space such that they are the correct non-square shape that, once the perspective divide and viewport transform converts them into window coordinates, they are again square.

Currently, our perspective matrix defines a square-shaped frustum. That is, the top and bottom of the frustum (if it were visualized in camera space) would be squares. What we need to do instead is create a rectangular frustum.

Figure 4.11. Widescreen Aspect Ratio Frustum



We already have some control over the shape of the frustum. We said originally that we did not need to move the eye position from the origin because we could simply scale the X and Y positions of everything to achieve a similar effect. When we do this, we scale the X and Y by the same value; this produces a uniform scale. It also produces a square frustum, as seen in camera space. Since we want a rectangular frustum, we need to use a non-uniform scale, where the X and Y positions are scaled by different values.

What this will do is show *more* of the world. But in what direction do we want to show more? Human vision tends to be more horizontal than vertical. This is why movies tend to use a minimum of 16:9 width:height aspect ratio (most use more width than that). So it is usually the case that you design a view for a particular height, then adjust the width based on the aspect ratio.

This is done in the AspectRatio tutorial. This code uses the same shaders as before; it simply modifies the perspective matrix in the `reshape` function.

Example 4.7. Reshape with Aspect Ratio

```
void reshape (int w, int h)
{
    perspectiveMatrix[0] = fFrustumScale / (w / (float)h);
    perspectiveMatrix[5] = fFrustumScale;

    glUseProgram(theProgram);
    glUniformMatrix4fv(perspectiveMatrixUnif, 1, GL_FALSE, perspectiveMatrix);
    glUseProgram(0);

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

The matrix, now a global variable called `perspectiveMatrix`, gets its other fields from the program initialization function just as before. The aspect ratio code is only interested in the XY scale values.

Here, we change the X scaling based on the ratio of the width to the height. The Y scaling is left alone.

Also, the offset used for positioning the prism was changed from (0.5, 0.5) to (1.5, 0.5). This means that part of the object is off the side of the viewport until you resize the window. Changing the width shows more of the area; only by changing the height do you actually make the objects bigger. The square always looks like a square.

In Review

In this tutorial, you have learned about the following:

- Face culling can cause triangles to be culled (not rendered) based on the order of the vertices in window space.
- Perspective projections are used to give a scene the appearance of depth, where objects farther away appear smaller and offset compared to near ones. OpenGL hardware has special provisions for perspective projections; namely the transform from clip-space to NDC space division by W.
- The perspective transformation can be performed as a matrix multiplication operation. Matrix/vector multiplication is a way to compute multiple linear equations in a single operation.
- The proper aspect ratio for a display image can be maintained by scaling the X and Y coordinates of camera-space vertices based on the window's aspect ratio. This transformation can be folded into the perspective projection matrix.

Further Study

Try doing these things with the given programs.

- In all of the perspective tutorials, we only ever had a frustum scale of 1.0. Adjust the frustum scale and see how it affects the scene.
- Adjust the zNear distance, so that it intersects with the prism. See how this affects the rendering. Adjust the zFar distance similarly and see what happens.
- We made some simplifying assumptions in our perspective transformation algorithm. In particular, we fixed the eye point at (0, 0, 0). and the plane at (0, 0, 1). However, this was not strictly necessary; we could have altered our perspective transform algorithm to use a variable eye point. Adjust the ShaderPerspective to implement an arbitrary perspective plane location (the size remains fixed at [-1, 1]). You will need to offset the X, Y camera-space positions of the vertices by E_x and E_y respectively, but only *after* the scaling (for aspect ratio). And you will need to divide the camera-space Z term by $-E_z$ instead of just -1.
- Do the above, but in matrix form. Remember that any terms placed in the fourth column will be added to that component, due to the multiplication by W_{camera} (which is always 1.0).

OpenGL Functions of Note

glEnable/glDisable

These functions activate or inactivate certain features of OpenGL. There is a large list of possible features that can be enabled or disabled. In this tutorial, `GL_CULL_FACE` was used to enable/disable face culling.

glCullFace/glFrontFace

These two functions control how face culling works. `glFrontFace` defines which triangle winding order is considered the front. `glCullFace` defines what face gets culled. This function can also cull *all* faces, though this is not useful if you want to get rendering done.

These functions only do something useful if `GL_CULL_FACE` is currently enabled. They still set the values internally even if `GL_CULL_FACE` is not enabled, so enabling it later will use the up-to-date settings.

Glossary

face culling

The ability to cull triangles based on the winding order of the triangle. This functionality is activated in OpenGL by using `glEnable` with `GL_CULL_FACE`. Which faces get culled is determined by the `glCullFace` and `glFrontFace` functions.

winding order

The order, clockwise or counter-clockwise, that the 3 vertices that make up a triangle are received in. This is measured in window coordinates, two-dimensionally.

projection

The act of taking a series of objects in a higher dimension and presenting those objects in a lower dimension. The act of rendering a 3D scene to a 2D image requires projecting that scene from three dimensions into two dimensions.

Projection always happens relative to a surface of projection. Projecting 2D space onto a 1D space requires a finite line to be projected on. Projecting 3D space onto 2D space requires a plane of projection. This surface is defined in the higher dimension's world.

orthographic projection

A form of projection that simply negates all offsets in the direction perpendicular to the surface of projection. When doing a 3D to 2D orthographic projection, if the plane is axis aligned, then the projection can be done simply. The coordinate that is perpendicular to the plane of projection is simply discarded. If the plane is not axis aligned, then the math is more complex, but it has the same effect.

Orthographic projections are uniform in the direction of the projection. Because of the uniformity, lines that are parallel in the higher dimension space are guaranteed to remain parallel in the lower dimension space.

perspective projection

A form of projection that projects onto the surface based on a position, the eye position. Perspective projections attempt to emulate a pin-hole camera model, which is similar to how human eyes see. The positions of objects in space are projected onto the surface of projection radially based on the eye position.

Parallel lines in the higher dimension are *not* guaranteed to remain parallel in the lower dimension. They might, but they might not.

frustum

Geometrically, a frustum is 3D shape; a pyramid that has the top chopped off. The view of a 3D to 2D perspective projection, from the eye through the plane of projection has the shape of a frustum.

perspective divide

A new name for the transformation from clip space to normalized device coordinate space. This is so called because the division by W is what allows perspective projection to work using only matrix math; a matrix alone would not otherwise be able to perform the full perspective projection operation.

camera space

An arbitrarily defined, but highly useful, space from which the perspective projection can be performed relatively easily. Camera space is an infinitely large space, with positive X going right, positive Y going up, and positive Z coming towards the viewer.

In camera space, the eye position of the perspective projection is assumed to be at (0, 0, 1), and the plane of projection is a [-1, 1] plane in X and Y, which passes through the 3D origin. Thus, all points that have a positive Z are considered to be behind the camera and thus out of view. Positions in camera space are defined relative to the camera's location, since the camera has a fixed point of origin.

camera zNear, camera zFar

Normalized device coordinate (NDC) space is bounded in all dimensions on the range [-1, 1]. Camera space is unbounded, but the perspective transform implicitly bounds what is considered in view to [-1, 1] in the X and Y axis. This leaves the Z axis unbounded, which NDC space does not allow.

The camera zNear and zFar values are numbers that define the minimum and maximum extent of Z in the perspective projection transform. These values are positive value, though they represent negative values in camera space. Using the standard perspective transform, both values must be greater than 0, and zNear must be less than zFar.

swizzle selection

Swizzle selection is a vector technique, unique to shading languages, that allows you to take a vector and arbitrarily build other vectors from the components. This selection is completely arbitrary; you can build a vec4 from a vec2, or any other combination you wish, up to 4 elements.

Swizzle selections use combinations of “x,” “y,” “z,” and “w” to pick components out of the input vector. Swizzle operations look like this:

```
vec2 firstVec;
```

```
vec4 secondVec = firstVec.xyxx;
vec3 thirdVec = secondVec.wzy;
```

Swizzle selection is, in graphics hardware, considered an operation so fast as to be instantaneous. That is, graphics hardware is built with swizzle selection in mind.

matrix

A two-dimensional arrangement of numbers. Like vectors, matrices can be considered a single element. Matrices are often used to represent the coefficients in a system of linear equations; because of this (among other things), matrix math is often called linear algebra.

The size of a matrix, the number of columns and rows (denoted as NxM, where N is the number of columns and M is the number of rows) determines the kind of matrix. Matrix arithmetic has specific requirements on the two matrices involved, depending on the arithmetic operation. Multiplying two matrices together can only be performed if the number of rows in the matrix on the left is equal to the number of columns in the matrix on the right. For this reason, among others, matrix multiplication is not commutative (A*B is not B*A; sometimes B*A is not even possible).

4x4 matrices are used in computer graphics to transform 3 or 4-dimensional vectors from one space to another. Most kinds of linear transforms can be represented with 4x4 matrices.

column-major, row-major

These terms define the two ways in which a matrix can be stored as an array of values. Column-major order means that, for an NxM matrix (columns x rows), the first N values in the array are the first column (top-to-bottom), the next N values are the second column, and so forth. In row-major order, the first M values in the array are the first row (left-to-right), followed by another M values for the second row, and so forth.

Chapter 5. Objects in Depth

In this tutorial, we will look at how to deal with rendering multiple objects, as well as what happens when multiple objects overlap.

Multiple Objects in OpenGL

The first step in looking at what happens when objects overlap is to draw more than one object. This is an opportunity to talk about a concept that will be useful in the future.

An object, in terms of what you draw, can be considered the results of a single drawing call. Thus, an object is the smallest series of triangles that you draw with a single set of program object state.

Vertex Array Objects

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:

1. Use `glEnableVertexAttribArray` to enable this attribute.
2. Use `glBindBuffer(GL_ARRAY_BUFFER)` to bind to the context the buffer object that contains the data for this attribute.
3. 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. Technically, they *always* have changed the VAO's state; all of the prior tutorials have these lines in the initialization function:

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

This creates a single VAO, which contains the vertex array state that we have been setting. This means that we have been changing the state of a VAO in all of the tutorials. We just did not talk about it at the time.

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, except as noted, will fail.

- `glVertexAttribPointer`. Also `glVertexAttribIPointer`, but we have not talked about that one yet.
- `glEnableVertexAttribArray`/`glDisableVertexAttribArray`
- `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER)`: Calling this without a VAO bound will not fail.

Buffer Binding and Attribute Association

You may notice that `glBindBuffer(GL_ARRAY_BUFFER)` is not on that list, even though it is part of the attribute setup for rendering. The binding to `GL_ARRAY_BUFFER` is not part of a VAO because the association between a buffer object and a vertex attribute does *not* happen when you call `glBindBuffer(GL_ARRAY_BUFFER)`. This association happens when you call `glVertexAttribPointer`.

When you call `glVertexAttribPointer`, OpenGL takes whatever buffer is *at the moment of this call* bound to `GL_ARRAY_BUFFER` and associates it with the given vertex attribute. Think of the `GL_ARRAY_BUFFER` binding as a global pointer that `glVertexAttribPointer` reads. So you are free to bind whatever you want or nothing at all to `GL_ARRAY_BUFFER` *after* making a `glVertexAttribPointer` call; it will affect *nothing* in the final rendering. So VAOs do store which buffer objects are associated with which attributes; but they do not store the `GL_ARRAY_BUFFER` binding itself.

If you want to know why `glVertexAttribPointer` does not simply take a buffer object rather than requiring this bind+call mechanism, it is again because of legacy API cruft. When buffer objects were first introduced, they were designed to impact the API as little as possible. So the old `glVertexAttribPointer` simply changed its behavior depending on whether something was bound to `GL_ARRAY_BUFFER` or not. Nowadays, since this function will fail if nothing is bound to `GL_ARRAY_BUFFER`, it is simply an annoyance.

This allows you to setup a VAO early on, during initialization, and then simply bind it and call a rendering function to draw your object. Be advised when using a VAO in this way: VAOs are *not* immutable. Calling any of the above functions will change the data stored in the VAO.

Indexed Drawing

In the last tutorial, 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. In an earlier tutorial, we defined glDrawArrays conceptually as the following pseudo-code:

Example 5.1. Draw Arrays Implementation

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

This defines how *array drawing* works. 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
```

You can use `glDrawArrays` to 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),
```

12 vertices, which generate 4 triangles.

Multiple Attributes and Index Arrays

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.

This means that there can and often will be some duplication within a particular attribute array. For example, in order to have solid face colors, we will still have to replicate the color for every position of that triangle. And corner positions that are shared between two triangles that have different colors will still have to be duplicated in different vertices.

It turns out that, for most meshes, duplication of this sort is fairly rare. Most meshes are smooth across their surface, so different attributes do not generally pop from location to location. Shared edges typically use the same attributes for both triangles along the edges. The simple cubes and the like that we use are one of the few cases where a per-attribute index would have a significant benefit.

Now that we understand how indexed drawing works, we need to know how to set it up 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`. You can use this buffer binding point for normal maintenance of a buffer object (allocating memory with glBufferData, etc), just like `GL_ARRAY_BUFFER`. But it also has a special meaning to OpenGL: indexed drawing is only possible when a buffer object is bound to this binding point, and the element array comes from this buffer object.

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.

In order to do indexed drawing, we must bind the buffer to `GL_ELEMENT_ARRAY_BUFFER` and then call `glDrawElements`.

```
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. Index data (and vertex data, for that matter) should always be aligned to its size. So if we are using 16-bit unsigned shorts for indices, then *`indices`* should be an even number.

This function can be defined by the following pseudo-code:

Example 5.2. Draw Elements Implementation

```
GLvoid *elementArray;
```

```
void glDrawElements(GLenum type, GLint count, GLenum type, GLsizeiptr 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`.

Multiple Objects

The tutorial project Overlap No Depth uses VAOs to draw two separate objects. These objects are rendered using indexed drawing. The setup for this shows one way to have the attribute data for multiple objects stored in a single buffer.

For this tutorial, we will be drawing two objects. They are both wedges, with the sharp end facing the viewer. The difference between them is that one is horizontal and the other is vertical on the screen.

The shaders are essentially unchanged from before. We are using the perspective matrix shader from the last tutorial, with modifications to preserve the aspect ratio of the scene. The only difference is the pre-camera offset value; in this tutorial, it is a full 3D vector, which allows us to position each wedge in the scene.

The initialization has changed, allowing us to create our VAOs once at start-up time, then use them to do the rendering. The initialization code is as follows:

Example 5.3. VAO Initialization

```
void InitializeVertexArrayObjects()
{
    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);

    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);
}
```

This 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.

The code generates 2 VAOs, binds them, then sets their state. Recall that, while the `GL_ARRAY_BUFFER` binding is not part of the VAOs state, the `GL_ELEMENT_ARRAY_BUFFER` binding *is* part of that state. So these VAOs store the attribute array data and the element buffer data; everything necessary to render each object except for the actual drawing call.

In this case, both objects use the same element buffer. However, since the element buffer binding is part of the VAO state, it *must* be set into each VAO individually. Notice that we only set the `GL_ARRAY_BUFFER` binding once, but the `GL_ELEMENT_ARRAY_BUFFER` is set for each VAO.

Note

If you look at the vertex position attribute in our array, 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.

Though the initialization code has been expanded, the rendering code is quite simple:

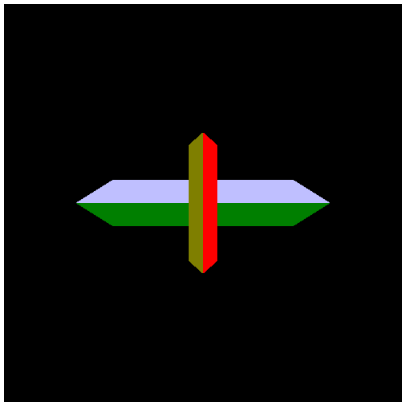
Example 5.4. VAO and Indexed Rendering Code

```
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_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();
    glutPostRedisplay();
```

We bind a VAO, set its uniform data (in this case, to position the object properly), and then we draw it with a call to `glDrawElements`. This step is repeated for the second object.

Running this tutorial will show the following image:

Figure 5.1. Overlapping Objects



The two objects are essentially flipped versions of the same one, a wedge. One object appears smaller than the other because it is farther away, in terms of its Z distance to the camera. We are using a perspective transform, so it make 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?

Before we solve this mystery, there is one minor issue we should cover first.

Optimization: Base Vertex

Using VAOs can dramatically simplify code. However, VAOs are not always the best case for performance, particularly if you use a lot of separate buffer objects.

Binding a VAO for rendering can be an expensive proposition. Therefore, if there is a way to avoid binding one, then it can provide a performance improvement, if the program is currently bottlenecked on the CPU.

Our two objects have much in common. They use the same vertex attribute indices, since they are being rendered with the same program object. They use the same format for each attribute (3 floats for positions, 4 floats for colors). The vertex data even comes from the same buffer object.

Indeed, the *only* difference between the two objects is what offset each attribute uses. And even this is quite minimal, since the difference between the offsets is a constant factor of the size of each attribute.

Look at the vertex data in the buffer object:

Example 5.5. Vertex Attribute Data Abridged

```
//Object 1 positions
LEFT_EXTENT,    TOP_EXTENT,       REAR_EXTENT,
LEFT_EXTENT,    MIDDLE_EXTENT,    FRONT_EXTENT,
RIGHT_EXTENT,   MIDDLE_EXTENT,    FRONT_EXTENT,

...

RIGHT_EXTENT,   TOP_EXTENT,       REAR_EXTENT,
RIGHT_EXTENT,   BOTTOM_EXTENT,    REAR_EXTENT,

//Object 2 positions
TOP_EXTENT,     RIGHT_EXTENT,     REAR_EXTENT,
```



```
MIDDLE_EXTENT,  RIGHT_EXTENT,     FRONT_EXTENT,
MIDDLE_EXTENT,  LEFT_EXTENT,      FRONT_EXTENT,

...

TOP_EXTENT,     RIGHT_EXTENT,     REAR_EXTENT,
TOP_EXTENT,     LEFT_EXTENT,      REAR_EXTENT,
BOTTOM_EXTENT,  LEFT_EXTENT,      REAR_EXTENT,

//Object 1 colors
GREEN_COLOR,
GREEN_COLOR,
GREEN_COLOR,

...

BROWN_COLOR,
BROWN_COLOR,

//Object 2 colors
RED_COLOR,
RED_COLOR,
RED_COLOR,

...

GREY_COLOR,
GREY_COLOR,
```

Notice how the attribute array for object 2 immediately follows its corresponding attribute array for object 1. So really, instead of four attribute arrays, we really have just two attribute arrays.

If we were doing array drawing, we could simply have one VAO, which sets up the beginning of both combined attribute arrays. We would still need 2 separate draw calls, because there is a uniform that is different for each object. But our rendering code could look like this:

Example 5.6. Array Drawing of Two Objects with One VAO

```
glUseProgram(theProgram);

glBindVertexArray(vaoObject);
glUniform3f(offsetUniform, 0.0f, 0.0f, 0.0f);
glDrawArrays(GL_TRIANGLES, 0, numTrianglesInObject1);

glUniform3f(offsetUniform, 0.0f, 0.0f, -1.0f);
glDrawArrays(GL_TRIANGLES, numTrianglesInObject1, numTrianglesInObject2);

glBindVertexArray(0);
glUseProgram(0);
```

This is all well and good for array drawing, but we are doing indexed drawing. And while we can control the location we are reading from in the element buffer by using the *`count`* and *`indices`* parameter of `glDrawElements`, that only specifies which indices we are reading from the element buffer. What we would need is a way to modify the index data itself.

This could be done by simply storing the index data for object 2 in the element buffer. This changes our element buffer into the following:

Example 5.7. MultiObject Element Buffer

```
const GLshort indexData[] =
```

```
{
//Object 1
0, 2, 1,        3, 2, 0,
4, 5, 6,        6, 7, 4,
8, 9, 10,       11, 13, 12,
14, 16, 15,     17, 16, 14,

//Object 2
18, 20, 19,     21, 20, 18,
22, 23, 24,     24, 25, 22,
26, 27, 28,     29, 31, 30,
32, 34, 33,     35, 34, 32,
};
```

This would work for our simple example here, but it does needlessly take up room. What would be great is a way to simply add a bias value to the index after it is pulled from the element array, but *before* it is used to access the attribute data.

I'm sure you'll be surprised to know that OpenGL offers such a mechanism, what with me bringing it up and all.

The function `glDrawElementsBaseVertex` provides this functionality. It works like `glDrawElements` has one extra parameter at the end, which is the offset to be applied to each index. The tutorial project Base Vertex With Overlap demonstrates this.

The initialization changes, building only one VAO.

Example 5.8. Base Vertex Single VAO

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

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);
```

This simply binds the beginning of each array. The rendering code is as follows:

Example 5.9. Base Vertex Rendering

```
glUseProgram(theProgram);

glBindVertexArray(vao);

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

glUniform3f(offsetUniform, 0.0f, 0.0f, -1.0f);
glDrawElementsBaseVertex(GL_TRIANGLES, ARRAY_COUNT(indexData),
 GL_UNSIGNED_SHORT, 0, numberOfVertices / 2);

glBindVertexArray(0);
glUseProgram(0);
```

The first draw call uses the regular glDrawElements function, but the second uses the BaseVertex version.

Note

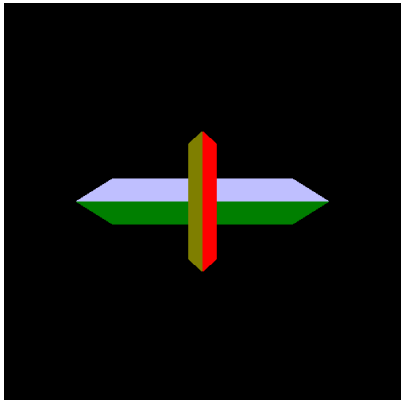
This example of BaseVertex's use is somewhat artificial, because both objects use the same index data. The more compelling way to use it is with objects that have different index data. Of course, if objects have different index data, you may be wondering why you would bother with BaseVertex when you could just manually add the offset to the indices themselves when you create the element buffer.

There are several reasons not to do this. One of these is that `GL_UNSIGNED_INT` is twice as large as `GL_UNSIGNED_SHORT`. If you have more than 65,536 entries in an array, whether for one object or for many, you would need to use ints instead of shorts for indices. Using ints can hurt performance, particularly on older hardware with less bandwidth. With BaseVertex, you can use shorts for everything, unless a particular object itself has more than 65,536 vertices.

The other reason not to manually bias the index data is to more accurately match the files you are using. When loading indexed mesh data from files, the index data is not biased by a base vertex; it is all relative to the model's start. So it makes sense to keep things that way where possible; it just makes the loading code simpler and faster by storing a per-object BaseVertex with the object rather than biasing all of the index data.

Overlap and Depth Buffering

Regardless of how we render the objects, there is a strange visual problem with what we're rendering:



If the smaller object is truly behind the larger one, why is it being rendered on top of the larger one? Well, to answer that question, we need to remember what OpenGL is.

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 render is just drawing triangles.

That's fine in general because rasterizers are very fast. They are very good at drawing triangles.

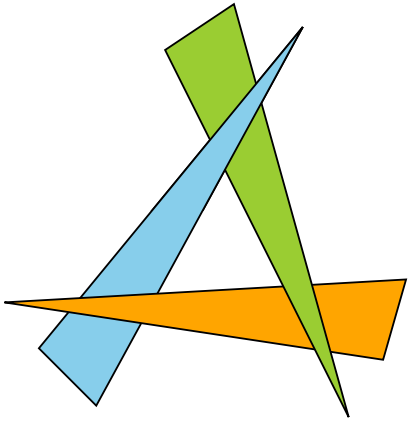
But rasterizers do exactly and only what the user says. They draw each triangle *in the order given*. This means that, if there is overlap between multiple triangles in window space, the triangle that is rendered last will be the one that is seen.

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.

Figure 5.2. Three Overlapping Triangles



Even worse, it does nothing for inter-penetrating triangles; that is, triangles that pass through each other in 3D space (as opposed to just from the perspective of the camera).

Depth sorting is not going to cut it; clearly, we need something better.

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 (ie: 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. You may recall from the introduction that the window-space Z position of a fragment ranges 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.

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 is 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.

With the fragment depth being something that is part of a fragment's output, you might imagine that this is something you have to compute in a fragment shader. You certainly can, but the fragment's depth is normally just the window-space Z coordinate of the fragment. This is computed automatically when the X and Y are computed.

Using the window-space Z value as the fragment's output depth is so common that, if you do not deliberately write a depth value from the fragment shader, this value will be used by default.

Depth and the Viewport

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.

Note

Do not confuse the range zNear/zFar with the *camera* zNear/zFar used in the perspective projection matrix computation.

The range zNear can be greater than the range zFar; if it is, then the window-space values will be reversed, in terms of what constitutes closest or farthest from the viewer.

Earlier, it was said that the window-space Z value of 0 is closest and 1 is farthest. However, if our clip-space Z values were negated, the depth of 1 would be closest to the view and the depth of 0 would be farthest. Yet, if we flip the direction of the depth test (GL_LESS to GL_GREATER, etc), we get the exact same result. Similarly, if we reverse the glDepthRange so that 1 is the depth zNear and 0 is the depth zFar, we get the same result if we use GL_GREATER. So it's really just a convention.

Z-Flip: Never Do This

In the elder days of graphics cards, calling `glClear` was a slow operation. And this makes sense; clearing images means having to go through every pixel of image data and writing a value to it. Even with hardware optimized routines, if you can avoid doing it, you save some performance.

Therefore, game developers found clever ways to avoid clearing anything. They avoided clearing the image buffer by ensuring that they would draw to every pixel on the screen every frame. Avoiding clearing the depth buffer was rather more difficult. But depth range and the depth test gave them a way to do it.

The technique is quite simple. They would need to clear the buffers exactly once, at the beginning of the program. From then on, they would do the following.

They would render the first frame with a `GL_LESS` depth test. However, the depth range would be [0, 0.5]; this would only draw to half of the depth buffer. Since the depth test is less, it does not matter what values just so happened to be between 0.5 and 1.0 in the depth buffer beforehand. And since every pixel was being rendered to as above, the depth buffer is guaranteed to be filled with values that are less than 0.5.

On the next frame, they would render with a `GL_GREATER` depth test. Only this time, the depth range would be [1, 0.5]. Because the last frame filled the depth buffer with values less than 0.5, all of those depth values are automatically "behind" everything rendered now. This fills the depth buffer with values greater than 0.5.

Rinse and repeat. This ultimately sacrifices one bit of depth precision, since each rendering only uses half of the depth buffer. But it results in never needing to clear the depth or color buffers.

Oh, and *you should never do this.*

See, hardware developers got really smart. They realized that a clear did not really have to go to each pixel and write a value to it. Instead, they could simply pretend that they had. They built special logic into the memory architecture, such that attempting to read from locations that have been "cleared" results in getting the clear color or depth value.

Because of that, this z-flip technique is useless. But it's rather worse than that; on most hardware made in the last 7 years, it actually slows down rendering. After all, getting a cleared value doesn't require actually reading memory; the very first value you get from the depth buffer is free. There are other, hardware-specific, optimizations that make z-flip actively damaging to performance.

Rendering with Depth

The Depth Buffering project shows off how to turn on and use the depth buffer. It is based on the BaseVertex rendering of the objects.

The initialization routine has all of the basic depth testing code in it:

Example 5.10. Depth Buffer Setup

```
glEnable(GL_DEPTH_TEST);
glDepthMask(GL_TRUE);
glDepthFunc(GL_LEQUAL);
glDepthRange(0.0f, 1.0f);
```

These 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.

It is common to use `GL_LEQUAL` instead of `GL_LESS`. This allows for the use of multipass algorithms, where you render the same geometry with the same vertex shader, but linked with a different fragment shader. We'll look at those much, much later.

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. We do not use this here, but a special algorithm might need this feature.

Note

Due to an odd quirk of OpenGL, writing to the depth buffer is always inactive if `GL_DEPTH_TEST` is disabled, regardless of the depth mask. If you just want to write to the depth buffer, without actually doing a test, you must enable `GL_DEPTH_TEST` and use the depth function of `GL_ALWAYS`.

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 drawing function's clearing, at the top of the function, happens as follows:

Example 5.11. Depth Buffer Clearing

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

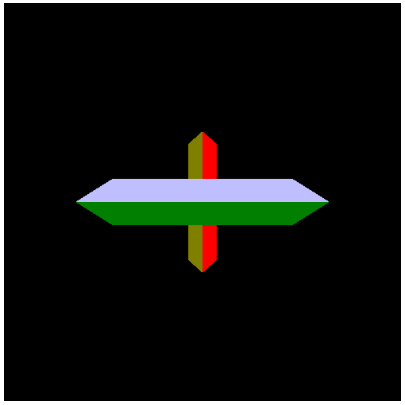
This will set all of the depth values in the depth buffer to 1.0, which is our range zFar.

Note

This is all that is necessary to do depth buffering, as far as OpenGL proper is concerned. However, in order to use depth buffering, the framebuffer must include a depth buffer in addition to an image buffer. This initialization code is platform-specific, but FreeGLUT takes care of it for us. If you do graduate from FreeGLUT, make sure that you use the appropriate initialization mechanism for your platform to create a depth buffer if you need to do depth buffering.

Our new image looks like this:

Figure 5.3. Depth Buffering



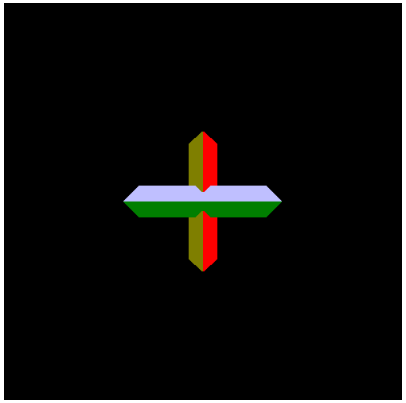
Which makes a lot more sense. No matter what order we draw the objects in, we get a reasonable result.

Let's test our depth buffering a bit more. Let's create a little overlap between the two objects. Change the first offset uniform statement in `display` to be this:

```
glUniform3f(offsetUniform, 0.0f, 0.0f, -0.75f);
```

We now get some overlap, but the result is still reasonable:

Figure 5.4. Mild Overlap

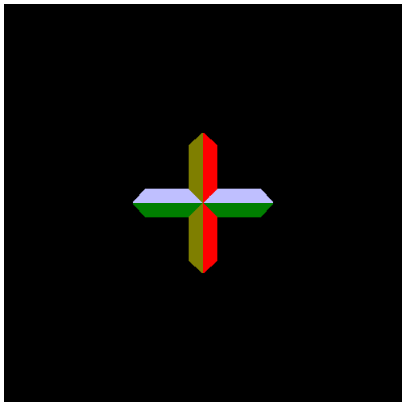


We can even change the line to cause major overlap without incident:

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

Which gives us:

Figure 5.5. Major Overlap



No amount of depth sorting will help with *that*.

Boundaries and Clipping

If you recall back to the Perspective projection tutorial, we choose to use some special hardware in the graphics chip to do the final division of the W coordinate, rather than doing the entire perspective projection ourselves in the vertex shader. At the time, it was promised that we would see why this is hardware functionality rather than something the shader does.

Let us review the full math operation we are computing here:

Equation 5.1. Perspective Computation

$$\#R = \#P\left(\frac{E_z}{P_z}\right)$$

R is the perspective projected position, P is the camera-space position, E_z is the Z-position of the eye relative to the plane (assumed to be -1), and P_z is the camera-space Z position.

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 where clip-space comes in to save the day. See, until we actually *do* the divide, everything is fine. A 4-dimensional vector that will be divided by the fourth component but has not *yet* is still valid, even if the fourth component is zero. This kind of coordinate system is called a *homogeneous coordinate system*. It is a way of talking about things that you could not talk about in a normal, 3D coordinate system. Like dividing by zero, which in visual terms refers to coordinates at infinity.

This is all nice theory, but we still know that the clip-space positions need to be divided by their W coordinate. So how to we get around this problem?

First, we know that a W of zero means that the camera-space Z position of the point was zero as well. We also know that this point *must* lie outside of the viable region for camera space. That is because of the camera Z range: camera zNear *must* be strictly greater than zero. Thus any point with a camera Z value of 0 must be in front of the zNear, and therefore outside of the visible world.

Since the vertex coordinate is not going to be visible anyway, why bother drawing it and dividing by that pesky 0? Well, because that vertex happens to be part of a triangle, and if part of the triangle is visible, we have to draw it.

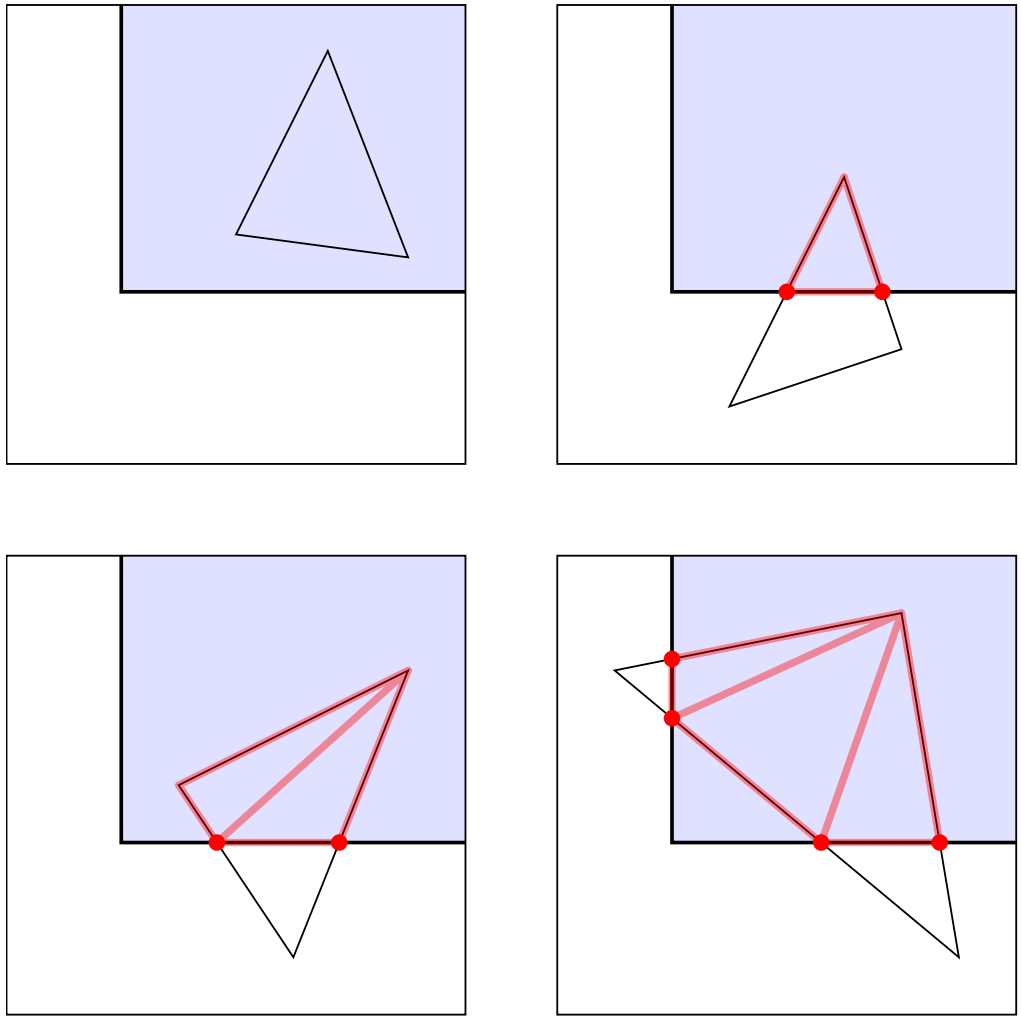
But we do not have to draw *all* of it.

Clipping is the process of taking a triangle and breaking it up into smaller triangles, such that only the part of the original triangle that is within the viewable region remains. This may generate only one triangle, or it may generate multiple triangles.

Any vertex attributes associated with that vertex are interpolated (based on the vertex shader's interpolation qualifiers) to determine the relative value of the post-clipping vertex.

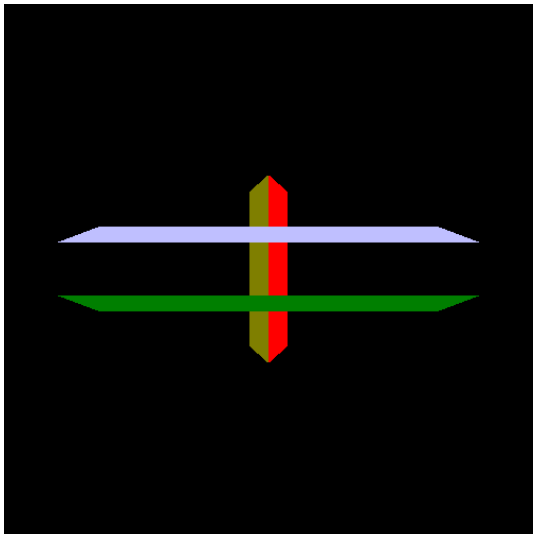
As you might have guessed, clipping happens in *clip space*, not NDC space. Hence the name. Since clip-space is a homogeneous coordinate system, we do not have to worry about those pesky zeros. Unfortunately, because homogeneous spaces are not easy to draw, we cannot show you what it would look like. But we can show you what it would look like if you clipped in camera space, in 2D:

Figure 5.6. Triangle Clipping



To see the results of clipping in action, run the Vertex Clipping tutorial. It is the same as the one for depth buffering, except one object has been moved very close to the zNear plane. Close enough that part of it is beyond the zNear and therefore is not part of the viewable area:

Figure 5.7. Near Plane Clipping



A Word on Clipping Performance

We have phrased the discussion of clipping as a way to avoid dividing by zero for good reason. The OpenGL specification states that clipping must be done against all sides of the viewable region. And it certainly appears that way; if you move objects far enough away that they overlap with zFar, then you will not see the objects.

You can also see apparent clipping with objects against the four sides of the view frustum. To see this, you would need to modify the viewport with `glViewport`, so that only part of the window is being rendered to. If you move objects to the edge of the viewport, you will find that part of them does not get rendered outside this region.

So clipping is happening all the time?

Of course not. Clipping takes triangles and breaks them into pieces using 4-dimensional homogeneous mathematics. One triangle can be broken up into several; depending on the location of the triangle, you can get quite a few different pieces. The simple act of turning one triangle into several is hard and time consuming.

So, if OpenGL states that this must happen, but supposedly OpenGL-compliant hardware does not do it, then what's going on?

Consider this: if we had not told you just now that the hardware does not do clipping most of the time, could you tell? No. And that's the point: OpenGL specifies *apparent* behavior; the spec does not care if you actually do vertex clipping or not. All the spec cares about is that the user cannot tell the difference in terms of the output.

That's how hardware can get away with the early-z optimization mentioned before; the OpenGL spec says that the depth test must happen after the fragment program executes. But if the fragment shader does not modify the depth, then would you be able to tell the difference if it did the depth test before the fragment shader? No; if it passes, it would have passed either way, and the same goes for failing.

Instead of clipping, the hardware usually just lets the triangles go through if part of the triangle is within the visible region. It generates fragments from those triangles, and if a fragment is outside of the visible window, it is discarded before any fragment processing takes place.

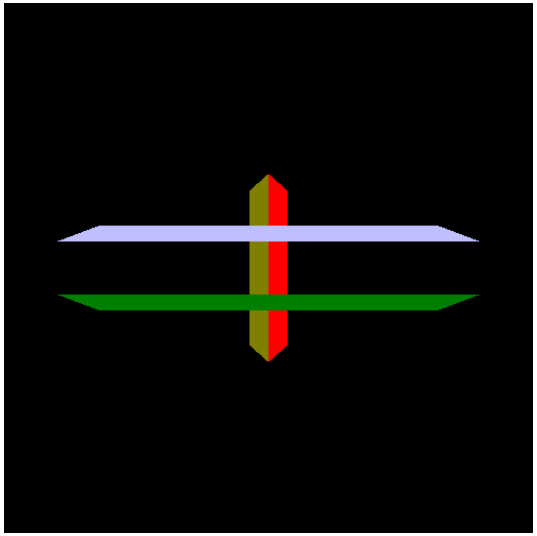
Hardware usually cannot do this however, if any vertex of the triangle has a clip-space W <= zero. In terms of a perspective projection, this means that part of the triangle is fully behind the eye, rather than just behind the camera zNear plane. In these cases, clipping is much more likely to happen.

Even so, clipping only happens if the triangle is partially visible; a triangle that is entirely in front of the zNear plane is dropped entirely.

In general, you should try to avoid rendering things that will clip against the eye plane (clip-space W <= 0, or camera-space Z >= 0). You do not need to be pedantic about it; long walls and the like are fine. But, particularly for low-end hardware, a lot of clipping can really kill performance.

Depth Clamping

That's all well and good, but this:



This is never a good thing. Sure, it keeps the hardware from dividing by zero, which I guess is important, but it looks really bad. It's showing the inside of an object that has no insides. Plus, you can also see that it has no backside (since we're doing face culling); you can see right through to the object behind it.

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?

The most common technique is to simply not allow it. That is, know how close objects are getting to the near clipping plane (ie: the camera) and do not let them get close enough to clip.

And while this can "function" as a solution, it is not exactly good. It limits what you can do with objects and so forth.

A more reasonable mechanism is *depth clamping*. What this does is turn off camera near/far plane clipping altogether. Instead, the depth for these fragments are clamped to the [-1, 1] range in NDC space.

We can see this in the Depth Clamping tutorial. This tutorial is identical to the vertex clipping one, except that the `keyboard` function has changed as follows:

Example 5.12. Depth Clamping On/Off

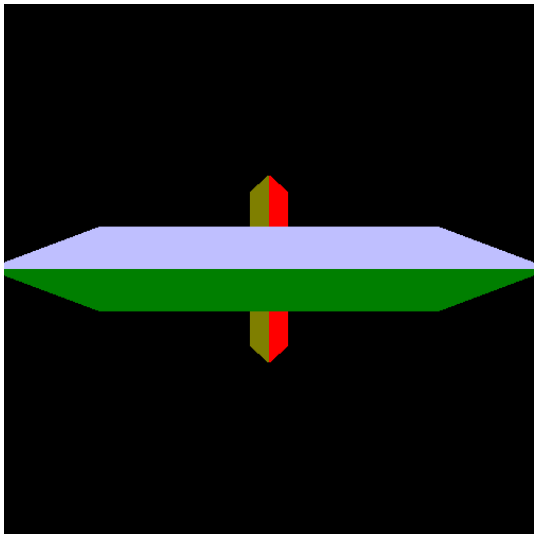
```
void keyboard(unsigned char key, int x, int y)
{
    static bool bDepthClampingActive = false;
    switch (key)
    {
    case 27:
        glutLeaveMainLoop();
        break;
    case 32:
        if(bDepthClampingActive)
            glDisable(GL_DEPTH_CLAMP);
        else
```

```
            glEnable(GL_DEPTH_CLAMP);

        bDepthClampingActive = !bDepthClampingActive;
        break;
    }
}
```

When you press the space bar (ASCII code 32), the code will toggle depth clamping, with the `glEnable`/`glDisable`(`GL_DEPTH_CLAMP`) calls. It will start with depth clamping off, since that is the OpenGL default.

When you run the tutorial, you will see what we saw in the last one; pressing the space bar shows this:

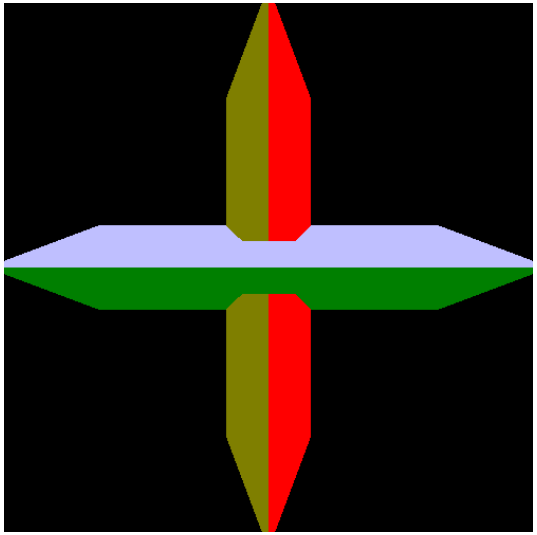
Figure 5.8. Depth Clamping



This looks correct; it appears as if all of our problems are solved.

Appearances can be deceiving. Let's see what happens if you move the other object forward, so that the two intersect like in the earlier part of the tutorial:

Figure 5.9. Depth Clamp With Overlap



Oops. Part of it looks right, just not the part where the depth is being clamped. What's going on?

Well, recall what depth clamping does; it makes fragment depth values outside of the range be clamped to within the range. So depth values smaller than depth zNear become depth zNear, and values larger than depth zFar become depth zFar.

Therefore, when you go to render the second object, some of the clamped fragments from the first are there. So the incoming fragment from the new object has a depth of 0, and some of the values from the depth buffer also have a depth of 0. Since our depth test is `GL_LESS`, the incoming 0 is not less than the depth buffer's 0, so the part of the second object does not get rendered. This is pretty much the opposite of where we started: previous rendered objects are in front of newer ones. We could change it to `GL_LEQUAL`, but that only gets us to *exactly* where we started.

So a word of warning: be careful with depth clamping when you have overlapping objects near the planes. Similar problems happen with the far plane, though backface culling can be a help in some cases.

Note

We defined depth clamping as, in part, turning off clipping against the camera near and far planes. If you're wondering what happens when you have depth clamping, which turns off clipping, and a clip-space W <= 0, it's simple. In camera space, near and far clipping is represented as turning a pyramid into a frustum: cutting off the top and bottom. If near/far clipping is not active, then the frustum becomes a pyramid. The other 4 clipping planes are still fully in effect. Clip-space vertices with a W of less than 0 are all outside of the boundary of any of the other four clipping planes.

The only clip-space point with a W of 0 that is within this volume is the homogeneous origin point: (0, 0, 0, 0); everything else will be clipped. And a triangle made from three positions that all are at the same position would have no area; it would therefore generate no fragments anyway. It can be safely eliminated before the perspective divide.

In Review

In this tutorial, you have learned about the following:

- Vertex array objects encapsulate all of the state necessary to render objects. This includes vertex attribute arrays, buffer objects to feed those arrays, and the element buffer, if present.

- Indexed rendering pulls data from the current vertex arrays by using a separate sequence of indices. The sequence of indices defines the sequence that OpenGL sees the vertices in. Indexed rendering can be performed by storing index data in a buffer object, and using `glDrawElements`.
- Indices in indexed rendering calls can be offset by a value using the `glDrawElementsBaseVertex` function.
- Hidden surface elimination can be performed by using a depth buffer and properly setting up the depth test.
- Triangles that are outside of the camera zNear and zFar range are clipped against this range. This can open up holes in models if they are too close to the camera.
- Clipping holes can be repaired to a degree by activating depth clamping, so long as there is no overlap. And as long as the triangles do not extend beyond 0 in camera space.

OpenGL Functions of Note

glGenVertexArrays/ glDeleteVertexArrays	Creates/destroys one or more vertex array objects.
glBindVertexArray	Binds a vertex array object to the `GL_VERTEX_ARRAY` target.
glDrawElements	Performs indexed rendering with the currently bound `GL_ELEMENT_ARRAY_BUFFER` (provided via the VAO) and the current attribute arrays.
glDrawElementsBaseVertex	Performs indexed rendering as `glDrawElements`, except that each element index is offset by a constant value before performing the array lookup. This is useful for minimizing the number of buffer object binds performed in a program.
glEnable/ glDisable(GL_DEPTH_TEST)	Enables/disables the per-fragment depth test. If the depth test is enabled, then the result of applying the depth function, set by `glDepthFunc`, to the incoming fragment's depth and the destination pixel's depth will determine if the incoming fragment is written or not.
glDepthMask	Sets or unsets the writing of values to the depth buffer.
glDepthFunc	Sets the depth comparison function for depth testing. Has no effect if `GL_DEPTH_TEST` is not enabled.
glDepthRange	Sets the mapping between NDC space and window space for the Z coordinate of the position. The XY counterpart to this function is `glViewport`. The range for the window-space depth must be [0, 1], though the near does not have to be less than the far. The range zNear value, the first value, is the window-space value that will map to -1 in NDC space. The range zFar is the window-space value that maps to +1 in NDC space.
glClearDepth	Sets the clear depth value. This is the value that the depth buffer will be cleared to when calling `glClear` with the `GL_DEPTH_BUFFER_BIT` bit set.
glEnable/ glDisable(GL_DEPTH_CLAMP)	Enables/disables depth clamping behavior. When enabled, clipping is deactivated, and any fragments that an object would render that are outside of the [-1, 1] range in NDC space are clamped to this range.

Glossary

vertex array object (VAO)	Vertex array objects are OpenGL Objects that store all of the state needed to make one or more draw calls. This includes attribute array setup information (from `glVertexAttribArray`), buffer objects used for attribute arrays, and the `GL_ELEMENT_ARRAY_BUFFER` binding, which is a buffer object that stores the index arrays, if needed.
array drawing	Rendering a contiguous range of vertices pulled from the currently bound attribute arrays (within the vertex array object). The vertices are sent in order from first to last in the range.
indexed drawing	Rendering an arbitrary set of vertices pulled from the currently bound attribute arrays. The set of vertices is defined by the element array. The vertices are rendered in the order specified by the element array.

element array, index array	A list of indices, stored within a buffer object, that refer to elements in the currently bound attribute arrays.
hidden surface elimination	The ability to render a scene such that objects that are behind other objects do not show through them. There are several methods available for achieving this.
depth sorting	Rendering objects or triangles in an order based on their Z-depth from the camera. An attempt at hidden surface elimination.
depth buffer, z-buffer	An image in the framebuffer that conceptually stores the distance of the pixel from the camera zNear plane. The depth buffer stores only one-dimensional values, instead of the 4-dimensional colors of the regular image buffer. Depth values are usually restricted to the range [0, 1].
depth test	The process of testing the incoming fragment's depth value against the depth value from the depth buffer for the pixel that the fragment would overwrite. If the test passes, then the fragment is written. If the test fails, the fragment is not written. This, combined with a depth buffer, can be used as a good method of hidden surface elimination.
range zNear, range zFar	The mapping from NDC-space Z coordinate [-1, 1] to window-space Z coordinates [0, 1]. This mapping is set with the `glDepthRange` function. These are specified in window-space coordinates. The -1 Z coordinate in NDC space maps to range zNear, and the 1 maps to range zFar. The range zNear does not have to be less than the range zFar.
homogeneous coordinate system	A 4-dimensional coordinate system used to represent a 3-dimensional position. To compute the 3D position, the fourth coordinate is divided into the other 3. This kind of coordinate system allows mathematics to function in the presence of what would be undefined values otherwise. Namely division by zero.
clipping	The act of breaking a single triangle into one or more smaller ones so that they all fit within the visible region. Actual clipping, generating new vertices and such, is not often done by hardware; instead they usually try to cull fragments that are outside of the viewing area.
depth clamping	A rendering mode where clipping is turned off and the Z value of fragments is clamped to the depth range. This is used to prevent clipping from punching holes in objects, though it is not a foolproof solution.

Chapter 6. Objects in Motion

In this tutorial, we will look at a number of ways to transform objects, rendering them in different locations and orientations in the world. And we will also solve the secret of why we overcomplicated everything with those matrices.

Spaces

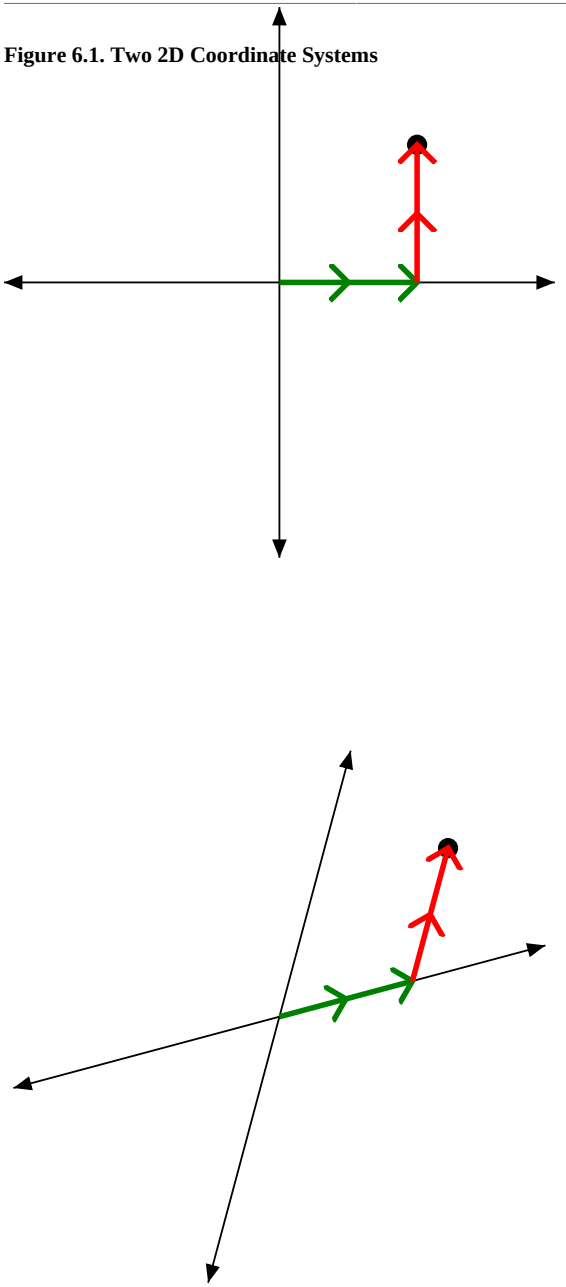
Throughout this series of tutorials, we have discussed a number of different spaces. We have seen OpenGL-defined spaces like normalized device coordinate (NDC) space, clip-space, and window space. And we have seen user-defined spaces like camera space. But we have yet to formally discuss about what a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

A *space* is a shorthand term for a *coordinate system.* For the purposes of this conversation, 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:

Figure 6.1. Two 2D Coordinate Systems



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.

The numerical version of the coordinate system equation is as follows:

Equation 6.1. Coordinate System

$$\text{Basis Vectors: } \begin{cases} (A_x, A_y, A_z) \\ (B_x, B_y, B_z) \\ (C_x, C_y, C_z) \end{cases}$$

$$\text{Origin Vectors: } (O_x, O_y, O_z)$$

$$\text{Coordinate Equation: } \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} X + \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} Y + \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} Z + \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix}$$

The geometric version is all well and good when dealing with the geometric basis vectors and origin point. The origin point is just a position, and the basis vectors are simply drawn. But what does it mean to give actual numbers to these concepts in the numerical version? A position, like the origin point, is itself a coordinate. Which means that it must be defined relative to some coordinate system. The same goes for the basis vectors.

Ultimately, this means that we cannot look numerically at a single coordinate system. Since the coordinate values themselves are meaningless without a coordinate system, a coordinate system can only be numerically expressed in relation to another coordinate system.

Technically, the geometric version of coordinate systems works the same way. The length of the basis vectors in the geometric diagrams are relative to our own self-imposed sense of length and space. Essentially, everything is relative to something, and we will explore this in the near future.

Transformation

In the more recent tutorials, the ones dealing with perspective projections, we have been taking positions in one coordinate system (space) and putting them in another coordinate system. Specifically, we had objects in camera space that we moved into clip space. The process of taking a coordinate in one space and specifying it as a coordinate in another space is called *transformation*. The coordinate's actual meaning has not changed; all that has changed is the coordinate system that this coordinate is relative to.

We have seen a number of coordinate system transformations. OpenGL implements the transformation from clip-space to NDC space and the transformation from NDC to window space. Our shaders implement the transformation from camera space to clip-space, and this was done using a matrix. Perspective projection (and orthographic, for that matter) are simply a special kind of transformation.

This tutorial will cover a large number of different kinds of transform operations and how to implement them in OpenGL.

Model Space

Before we begin, 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.

There are an infinite variety of model spaces. Each object one intends to render can, and often does, have its own model space, even if the difference between these spaces is only in the origin point. Model spaces for an object are generally defined for the convenience of the modeller or the programmer who intends to use that model.

The transformation operation being discussed in this tutorial 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.

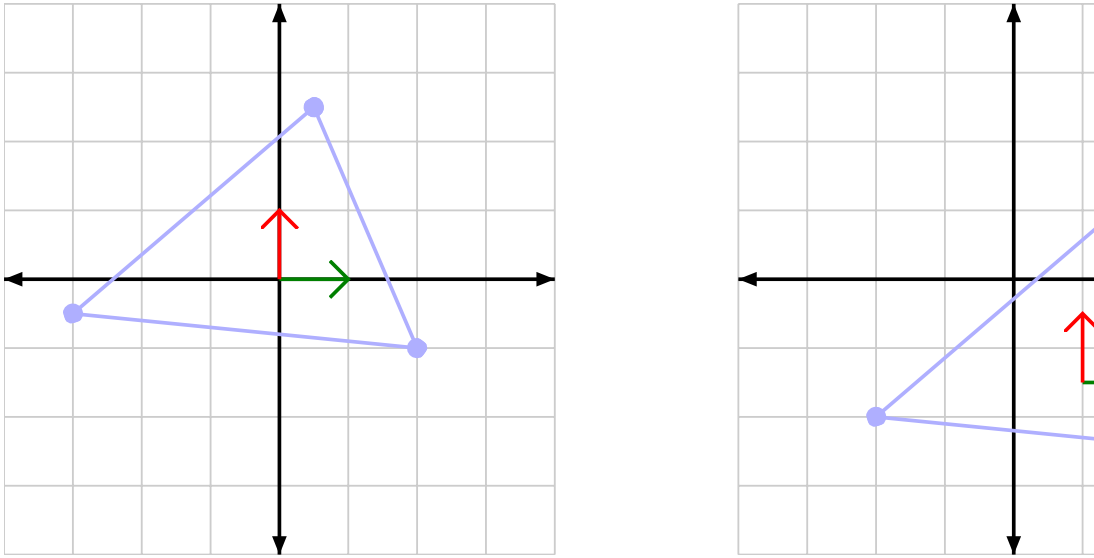
Translation

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

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

This is a *translation* transformation: it is used to position the origin point of the initial space relative to the destination space. Since all of the coordinates in a space are relative to the origin point of that space, 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.

Figure 6.2. Coordinate System Translation in 2D



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.

All of our position vectors are 4D vectors, with a final W coordinate that is always 1.0. In Tutorial 04, we took advantage of this with our perspective transformation matrix. The equation for the Z coordinate needed an additive term, so we put that term in the W column of the transformation matrix. Matrix multiplication causes the value in the W column to be multiplied by the W coordinate of the vector (which is 1) and added to the sum of the other terms.

But how do we keep the matrix from doing something to the other terms? We only want this matrix to apply an offset to the position. We do not want to have it modify the position in some other way.

This is done by modifying an *identity matrix*. An identity matrix is a matrix that, when performing matrix multiplication, will return the matrix (or vector) it was multiplied with. It is sort of like the number 1 with regular multiplication: 1*X = X. The 4x4 identity matrix looks like this:

Equation 6.2. Identity Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To modify the identity matrix into one that is suitable for translation, we simply put the offset into the W column of the identity matrix.

Equation 6.3. Translation Matrix

$$\text{Translation} = (x, y, z)$$
$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The tutorial project cleverly titled Translation performs translation operations.

This tutorial renders 3 of the same object, all in different positions. One of the objects is positioned in the center of the screen, and the other two's positions orbit it at various speeds.

Because of the prevalence of matrix math, this is the first tutorial that uses the GLM math library. So let's take a look at the shader program initialization code to see it in action.

Example 6.1. Translation Shader Initialization

```
void InitializeProgram()
{
    std::vector<GLuint> shaderList;

    shaderList.push_back(Framework::LoadShader(GL_VERTEX_SHADER,
        "PosColorLocalTransform.vert"));
    shaderList.push_back(Framework::LoadShader(GL_FRAGMENT_SHADER,
        "ColorPassthrough.frag"));

    theProgram = Framework::CreateProgram(shaderList);

    positionAttrib = glGetAttribLocation(theProgram, "position");
    colorAttrib = glGetAttribLocation(theProgram, "color");

    modelToCameraMatrixUnif = glGetUniformLocation(theProgram,
        "modelToCameraMatrix");
    cameraToClipMatrixUnif = glGetUniformLocation(theProgram,
        "cameraToClipMatrix");

    float fzNear = 1.0f; float fzFar = 45.0f;

    cameraToClipMatrix[0].x = fFrustumScale;
    cameraToClipMatrix[1].y = fFrustumScale;
    cameraToClipMatrix[2].z = (fzFar + fzNear) / (fzNear - fzFar);
    cameraToClipMatrix[2].w = -1.0f;
    cameraToClipMatrix[3].z = (2 * fzFar * fzNear) / (fzNear - fzFar);

    glUseProgram(theProgram);
    glUniformMatrix4fv(cameraToClipMatrixUnif, 1, GL_FALSE,
```



```
        glm::value_ptr(cameraToClipMatrix));
    glUseProgram(0);
}
```

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.

The matrix `cameraToClipMatrix` is defined as a glm::mat4, which has the same properties as a GLSL mat4. Array indexing of a mat4, whether GLM or GLSL, returns the zero-based *column* of the matrix as a vec4.

The `glm::value_ptr` function is used to get a direct pointer to the matrix data, in column-major order. This is useful for uploading data to OpenGL, as shown with the call to `glUniformMatrix4fv`.

With the exception of getting a second uniform location (for our model transformation matrix), this code functions exactly as it did in previous tutorials.

There is one important note: `fFrustumScale` is not 1.0 anymore. Until now, the relative sizes of objects were not particularly meaningful. Now that we are starting to deal with more complex objects that have a particular scale, picking a proper field of view for the perspective projection is very important.

The new `fFrustumScale` is computed with this code:

Example 6.2. Frustum Scale Computation

```
float CalcFrustumScale(float fFovDeg)
{
    const float degToRad = 3.14159f * 2.0f / 360.0f;
    float fFovRad = fFovDeg * degToRad;
    return 1.0f / tan(fFovRad / 2.0f);
}

const float fFrustumScale = CalcFrustumScale(45.0f);
```

The function `CalcFrustumScale` computes the frustum scale based on a field-of-view angle in degrees. The field of view in this case is the angle between the forward direction and the direction of the farmost-extent of the view.

This project, and many of the others in this tutorial, uses a fairly complex bit of code to manage the transform matrices for the various object instances. There is an `Instance` object for each actual object; it has a function pointer that is used to compute the object's offset position. The `Instance` object then takes that position and computes a transformation matrix, based on the current elapsed time, with this function:

Example 6.3. Translation Matrix Generation

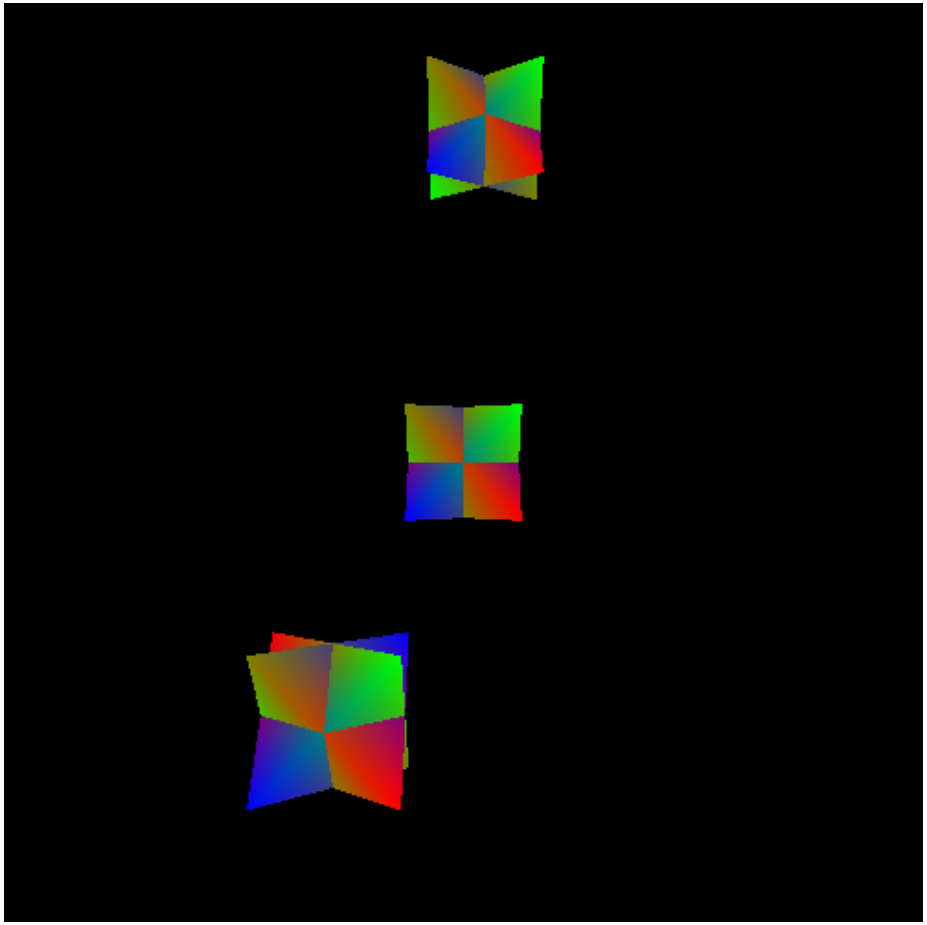
```
glm::mat4 ConstructMatrix(float fElapsedTime)
{
    glm::mat4 theMat(1.0f);

    theMat[3] = glm::vec4(CalcOffset(fElapsedTime), 1.0f);

    return theMat;
}
```

The glm::mat4 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.

This function simply replaces the W column of that identity matrix with the offset value.

This all produces the following:

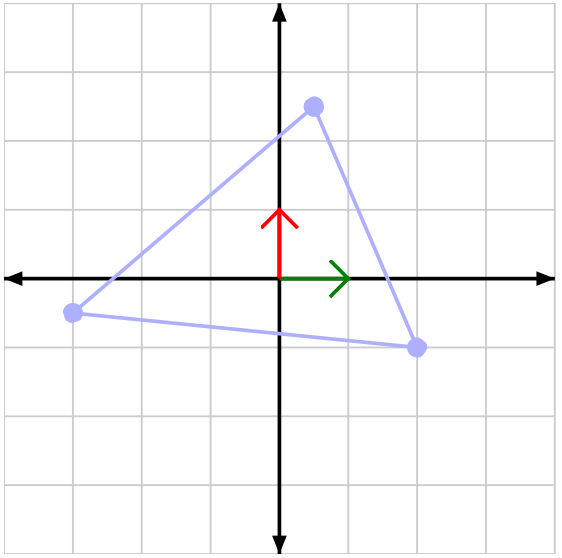
Figure 6.3. Translation Project

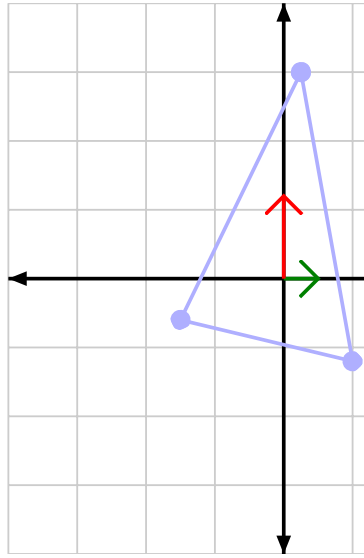


Scale

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.

Figure 6.4. Coordinate System Scaling in 2D





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 modeller 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.

Note that scaling always happens relative to the origin of the space being scaled.

Recall how we defined the way coordinate systems generate a position, based on the basis vectors and origin point:

$$X\begin{bmatrix}A_x\\A_y\\A_z\end{bmatrix}+Y\begin{bmatrix}B_x\\B_y\\B_z\end{bmatrix}+Z\begin{bmatrix}C_x\\C_y\\C_z\end{bmatrix}+\begin{bmatrix}O_x\\O_y\\O_z\end{bmatrix}$$

If you are increasing or decreasing the length of the basis vectors, this is the same as multiplying those basis vectors by the new length. So we can re-express this equation as follows:

$$\begin{bmatrix}A_x*S_a\\A_y*S_a\\A_z*S_a\end{bmatrix}\quad X\quad+\quad\begin{bmatrix}B_x*S_b\\B_y*S_b\\B_z*S_b\end{bmatrix}\quad Y\quad+\quad\begin{bmatrix}C_x*S_c\\C_y*S_c\\C_z*S_c\end{bmatrix}\quad Z\quad+\quad\begin{bmatrix}O_x\\O_y\\O_z\end{bmatrix}$$

$$\begin{bmatrix}A_x\\A_y\\A_z\end{bmatrix}*S_a*\quad X\quad+\quad\begin{bmatrix}B_x\\B_y\\B_z\end{bmatrix}*S_b*\quad Y\quad+\quad\begin{bmatrix}C_x\\C_y\\C_z\end{bmatrix}*S_c*\quad Z\quad+\quad\begin{bmatrix}O_x\\O_y\\O_z\end{bmatrix}$$

Since scalar-vector multiplication is both associative and commutative, we can multiply the scales directly into the coordinate values to achieve the same effect. So a scaled space can be reexpressed as simply multiplying the input coordinate values.

This is easy enough to do in GLSL, if you pass a vector uniform containing the scale values. But that's just not complicated enough. Obviously, we need to get matrices involved, but how?

This gets a bit technical, in terms of how a matrix multiplication works. But look back at the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix selects each coordinate in turn from the vector it is being multiplied into. Each row is multiplied with the column of the vector; all of the zeros remove the components of the vector that we do not want. The 1 value of each row multiplies into the component we do want, thus selecting it. This produces the identity result: the vector we started with.

We can see that, if the ones were some other value, we would get a scaled version of the original vector, depending on which ones were changed. Thus, a scaling transformation matrix looks like this:

Equation 6.4. Scaling Transformation Matrix

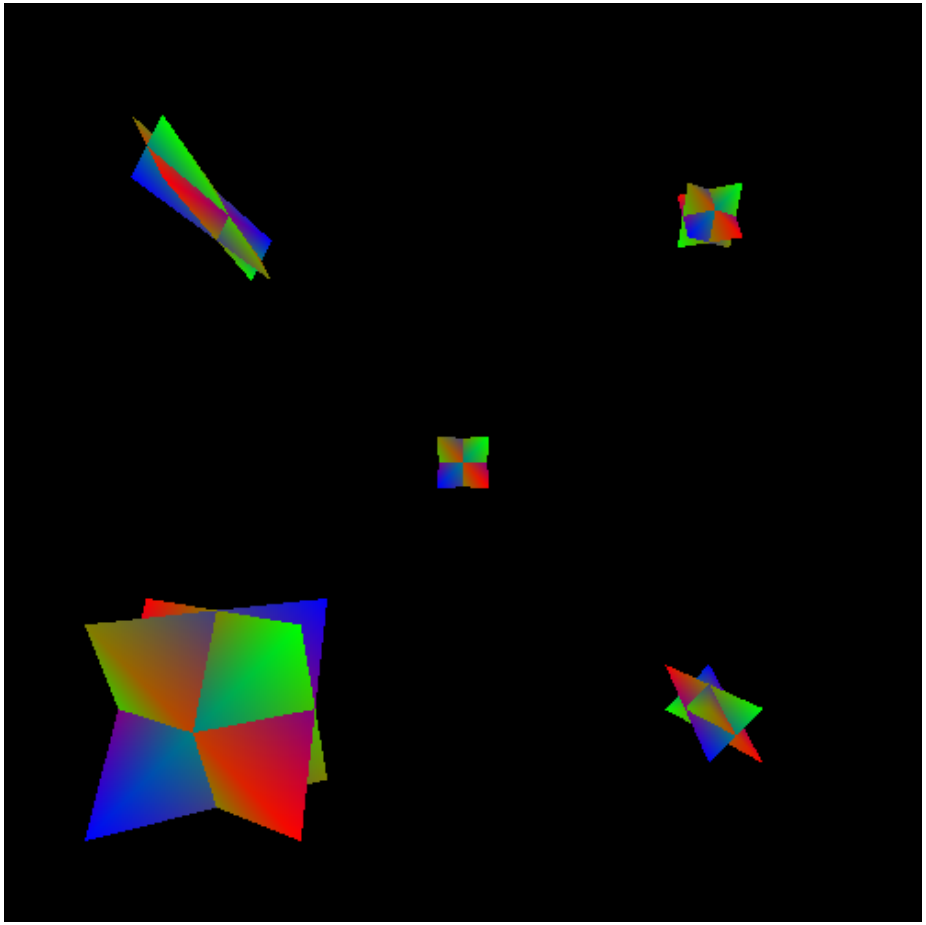
$$\text{Scale} = (x, y, z)$$

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You may start to see a pattern emerging here, something that begins to suggest why matrices are very, very useful. I will not spoil it for you yet though.

The tutorial project Scale will display 5 objects at various scales. The objects are all at the same Z distance from the camera, so the only size difference between the objects is the scale effects applied to them. The object in the center is unscaled; each of the other objects has a scale function of some kind applied to them.

Figure 6.5. Scale Project



Other than the way the tutorial builds its matrices, there is no difference between this tutorial project and the previous one. The matrix building code works as follows:

```
glm::mat4 ConstructMatrix(float fElapsedTime)
{
    glm::vec3 theScale = CalcScale(fElapsedTime);
    glm::mat4 theMat(1.0f);
    theMat[0].x = theScale.x;
    theMat[1].y = theScale.y;
    theMat[2].z = theScale.z;
    theMat[3] = glm::vec4(offset, 1.0f);
```

```
    return theMat;
}
```

As before, the scale is supplied by a number of scale functions, depending on which instance is being rendered. The scale is stored in the columns of the identity matrix. Then the translation portion of the matrix is filled in.

The `offset` variable is also a member of the `Instance` object. Unlike the last tutorial, the offset is a fixed value. We will discuss the ramifications of applying multiple transforms later; suffice it to say, this currently works.

Scaling is only slightly more complicated than translation.

Perspective and Scaling

The way we construct a scale transformation matrix may seem familiar to you. Back in Tutorial 4, the perspective transformation involved a frustum scale value. This was used to make up for the fact that our projection defined a specific location for the plane of projection and camera eye point. Using this frustum scale, we could give the appearance of having a larger or smaller viewing size. Indeed, we later used this to define the aspect ratio as well as the field of view, using the function defined earlier in this tutorial.

This frustum scale was ultimately nothing more than a scale factor applied the X and Y positions. When we constructed the perspective matrix, we used the frustum scale as a uniform scaling in the X and Y dimensions. The aspect ratio compensation code was nothing more than applying a *non*uniform scale.

Inversion and Winding Order

Scales can be theoretically negative, or even 0. A scale of 0 causes the basis vector in that direction to become 0 entirely. An basis vector with no length means that a dimension has effectively been lost. The resulting transform squashes everything in that direction down to the origin. A 3D space becomes a 2D space (or 1D or 0D, depending on how many axes were scaled).

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.

Back in Tutorial 4, 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.

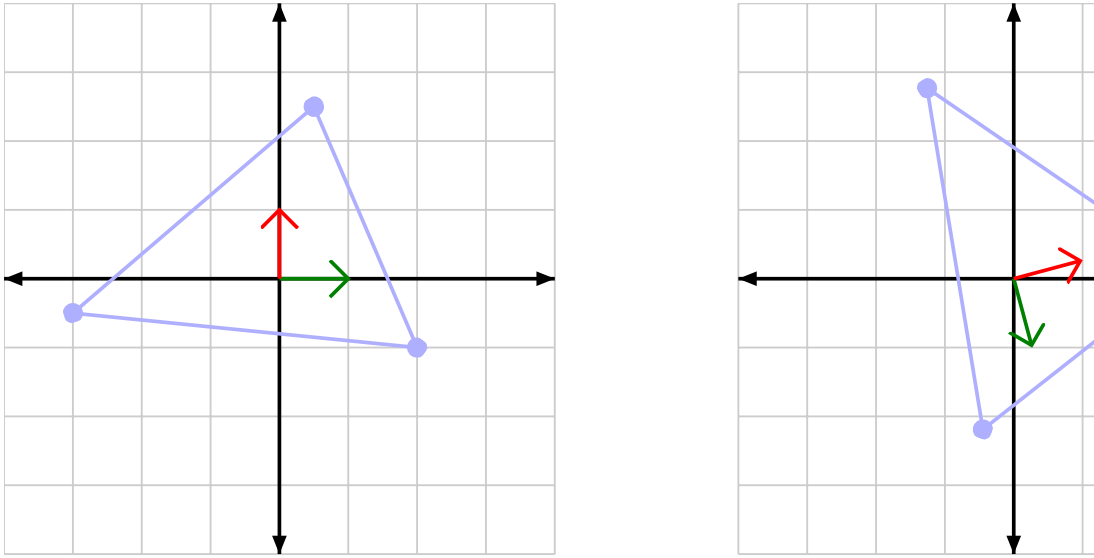
Negative scaling can have other problems as well. This is not to say that inversions cannot be used, but they should be used with care.

Rotation

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:

Figure 6.6. Coordinate Rotation in 2D



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. The prior part of the tutorial laid down some of the groundwork that will make this much simpler.

First, let's look back at our equation for determining what the position of a coordinate is relative to certain coordinate space:

$$X\begin{bmatrix}A_x\\A_y\\A_z\end{bmatrix} + Y\begin{bmatrix}B_x\\B_y\\B_z\end{bmatrix} + Z\begin{bmatrix}C_x\\C_y\\C_z\end{bmatrix} + \begin{bmatrix}O_x\\O_y\\O_z\end{bmatrix}$$

Does not this look a bit familiar? No? Maybe this look at vector-matrix multiplication will jog your memory:

$$\begin{bmatrix}m11 & m21 & m31 & m41\\ m12 & m22 & m32 & m42\\ m13 & m23 & m33 & m43\\ m14 & m24 & m34 & m44\end{bmatrix}\begin{bmatrix}x\\y\\z\\w\end{bmatrix} = \begin{bmatrix}x*m11 + y*m21 + z*m31 + w*m41\\ x*m12 + y*m22 + z*m32 + w*m42\\ x*m13 + y*m23 + z*m33 + w*m43\\ x*m14 + y*m24 + z*m34 + w*m44\end{bmatrix}$$

Still nothing? Perhaps an alternate look would help:

Equation 6.5. Vectorized Matrix Multiplication

$$\begin{bmatrix}m11 & m21 & m31 & m41\\ m12 & m22 & m32 & m42\\ m13 & m23 & m33 & m43\\ m14 & m24 & m34 & m44\end{bmatrix}\begin{bmatrix}x\\y\\z\\w\end{bmatrix} = x\begin{bmatrix}m11\\m12\\m13\\m14\end{bmatrix} + y\begin{bmatrix}m21\\m22\\m23\\m24\end{bmatrix} + z\begin{bmatrix}m31\\m32\\m33\\m34\end{bmatrix} + w\begin{bmatrix}m41\\m42\\m43\\m44\end{bmatrix}$$

Does it look familiar *now*?

What this tells us is that the columns of our transformation matrices are, and have *always* been, nothing more than the axes of a coordinate system. Except for the fourth column; because the input position has a 1 in the W, it acts as an offset.

Transformation from one space to another ultimately means this: taking the basis vectors and origin point from the original coordinate system and re-expressing them relative to the destination coordinate system. The transformation matrix from one space to another contains the basis vectors and origin of the original coordinate system, but the *values* of those basis vectors and origin are relative to the destination coordinate system.

Earlier, we said that numerical coordinates of a space must be expressed relative to another space. A matrix is a numerical representation of a coordinate system, and its values are expressed in the destination coordinate system. Therefore, a transformation matrix takes values in one coordinate system and transforms them into another. It does this by taking the basis vectors and origin of the input coordinate system and represents them relative to the output space. To put it another way, the transformation from space A to space B is what space A looks like from an observer in space B.

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.

Therefore, 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.

For any two spaces, the orientation transformation between then can be expressed as rotating the source space by some angle around a particular axis (specified in the initial space). This is true for any change of orientation.

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. In 2D, there is only one axis that can be rotated around and still remain within that 2D plane: the Z-axis.

In 3D, there are many possible axes of rotation. It does not have to be one of the initial space's basis axes; it can be any arbitrary direction. Of course, the problem is made much simpler if one rotates only around the primary axes.

Deriving these matrix equations is beyond the scope of this tutorial; so instead, we will simply provide them. To perform rotations along the primary axes, use the following matrices:

Equation 6.6. Axial Rotation Matrices

Rotation Angle $= \#$

$$\text{X Rotation} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\#) & -\sin(\#) & 0 \\ 0 & \sin(\#) & \cos(\#) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Y Rotation} \begin{bmatrix} \cos(\#) & 0 & \sin(\#) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\#) & 0 & \cos(\#) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Z Rotation} \begin{bmatrix} \cos(\#) & -\sin(\#) & 0 & 0 \\ \sin(\#) & \cos(\#) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When using the standard C/C++ library `sin` and `cos` functions, the angles must be in radians.

As useful as these are, the more generic equation for rotation by an angle about an arbitrary axis is as follows.

Equation 6.7. Angle/Axis Rotation Matrix

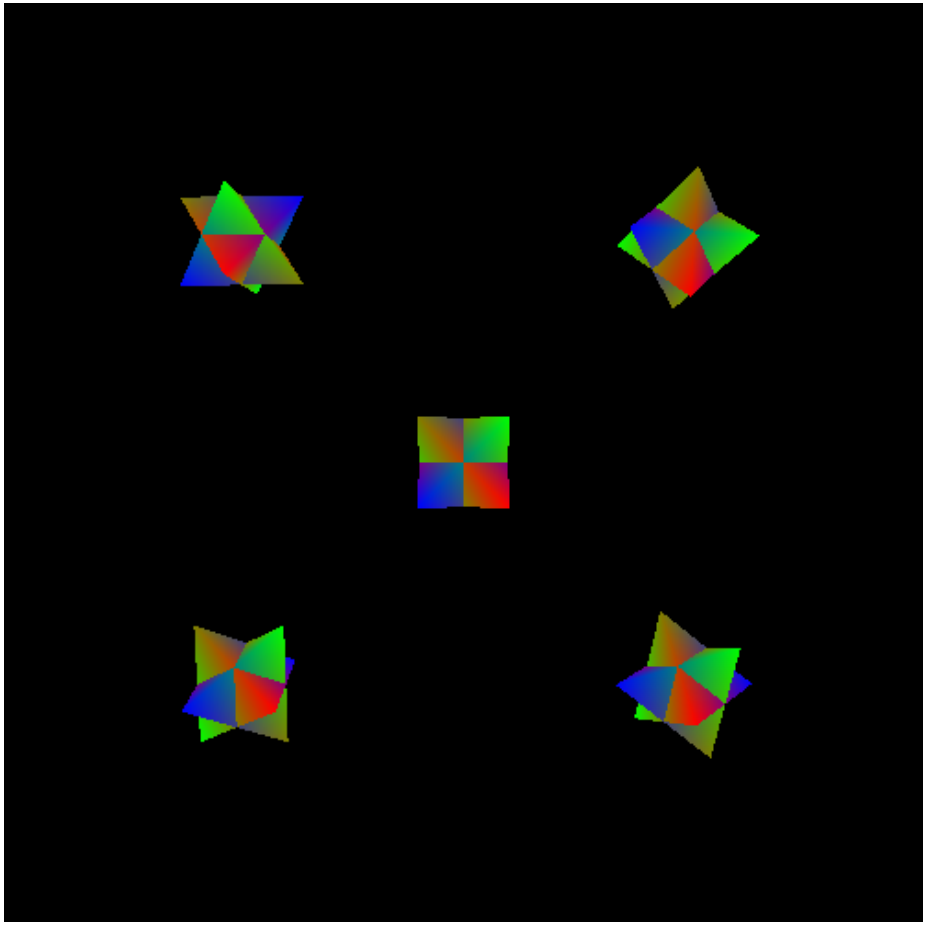
$$\text{Axis} = (x, y, z) \quad \text{Angle} = \#$$
$$C = \cos(\#) \qquad S = \sin(\#)$$
$$\text{iC} = 1 - \cos(\#) \quad \text{iS} = 1 - \sin(\#)$$

$$\begin{bmatrix} x^2 + (1 - x^2)C & \text{iC}xy - zS & \text{iC}xz + yS & 0 \\ \text{iC}xy + zS & y^2 + (1 - y^2)C & \text{iC}yz - xS & 0 \\ \text{iC}xz - yS & \text{iC}yz + xS & z^2 + (1 - z^2)C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

All of these matrices are such that, from the point of view of an observer looking down the axis of rotation (the positive direction of the axis is pointed into the eye of the observer), the object rotates counter-clockwise with positive angles.

The Rotations tutorial shows off each of these rotation matrix functions. Similar to how the others work, there are multiple instances rendered based on functions.

Figure 6.7. Rotation Project



The function that builds the transformation matrix looks like this:

Example 6.4. Rotation Transformation Building

```
glm::mat4 ConstructMatrix(float fElapsedTime)
{
    const glm::mat3 &rotMatrix = CalcRotation(fElapsedTime);
    glm::mat4 theMat(rotMatrix);
    theMat[3] = glm::vec4(offset, 1.0f);

    return theMat;
}
```


The constructor of glm::mat4 that takes a glm::mat3 generates a 4x4 matrix with the 3x3 matrix in the top-left corner, and all other positions 0 except the bottom-left corner, which is set to 1. As with much of GLM, this works in GLSL as well.

Fun with Matrices

In all of the previous examples except for the translation one, we always combined the transformation with a translation operation. So the scale transform was not a pure scale transform; it was a scale and translate transformation matrix. The translation was there primarily so that we could see everything properly.

But these are not the only combinations of transformations that can be performed. Indeed, any combination of transformation operations is possible; whether they are meaningful and useful depends on what you are doing.

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.

This is functional, but not particularly flexible; the series of transforms is baked into the shader. It is also not particularly fast, what with having to do four vector/matrix multiplications for every vertex.

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 Transforms

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

Equation 6.8. Order of Transformation

$$S = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

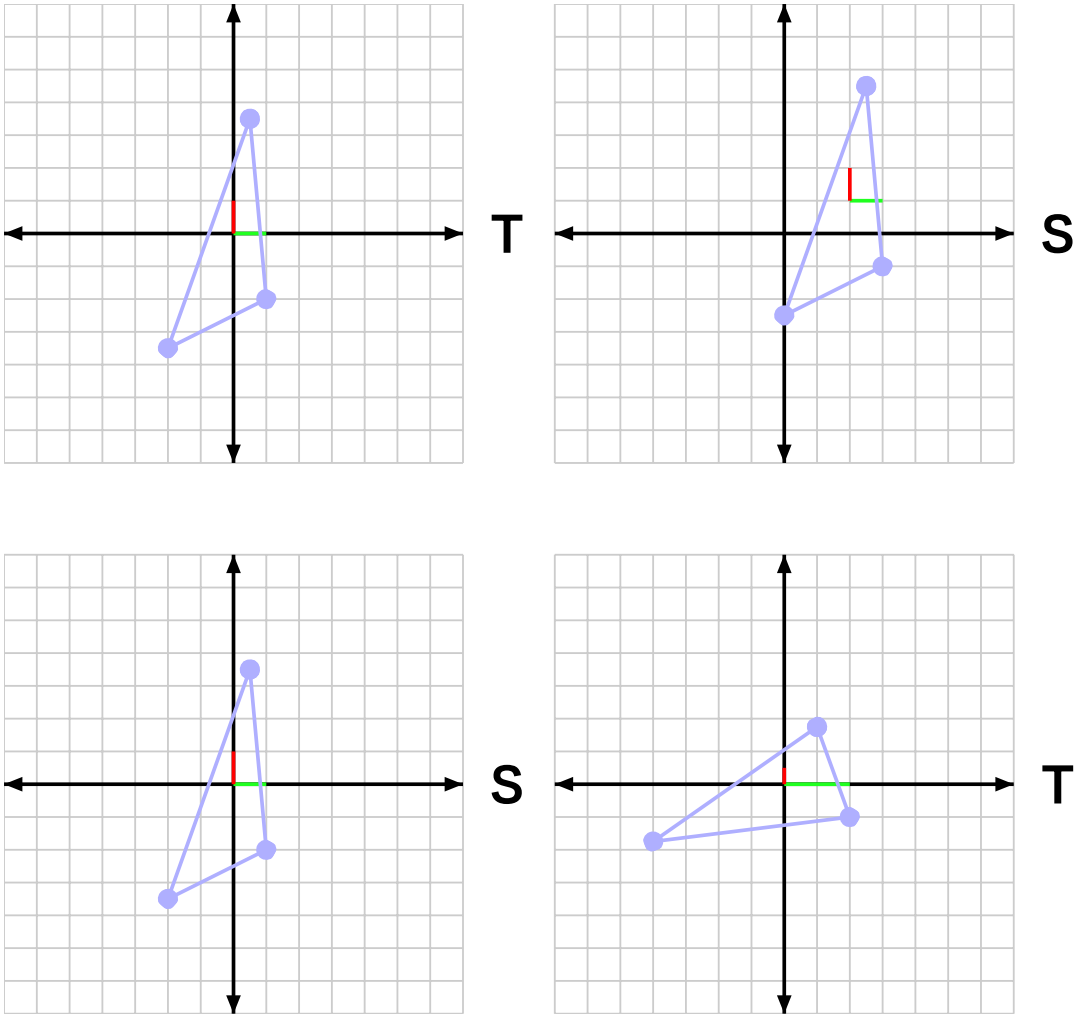
$$T = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S \times T = \begin{bmatrix} 2 & 0 & 0 & 4 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T \times S = \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 0.5 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transform S*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:

Figure 6.8. Transform Order Diagram



If you think about the order of operations, this makes sense. Even though one can think of the combined transform S*T as a single transform, it is ultimately a composite operation. The transformation T happens first; the object is translated into a new position.

What you must understand is that something special happens between S and T. Namely, that S is now 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 now acts on the T-space position of the vertices. T-space has an origin, which in T-space is (0, 0, 0). However, this origin back in model space is the translation part of the matrix T. A scaling transformation matrix performs scaling based on the origin point in the space of the vertices

being scaled. So the scaling matrix S will scale the points away from the origin point in T-space. Since what you (probably) actually wanted was to scale the points away from the origin point in *model space*, S needs to come first.

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.

A conceptually single model that is composed of multiple transforms for multiple rendered objects is called a *hierarchical model.* In such a hierarchy, 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.

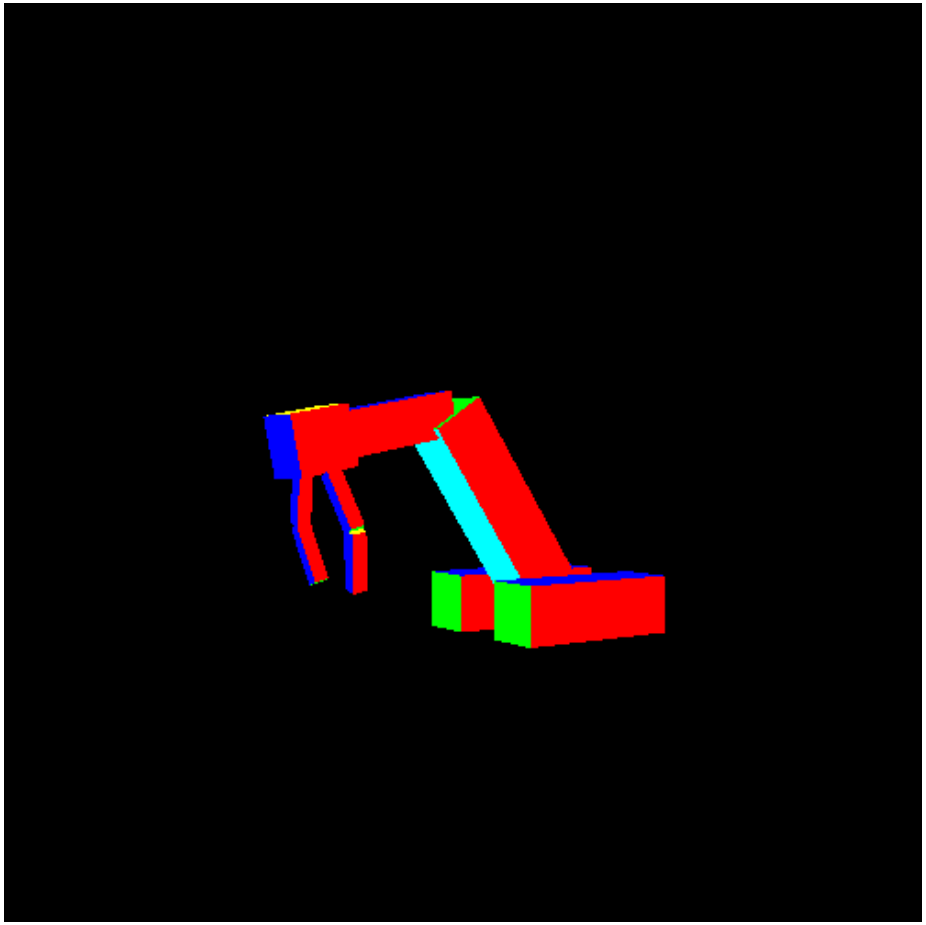
Technically, a node does not have to have a mesh. It is sometimes useful in a hierarchical model to have nodes that exist solely to position other, visible nodes. Or to act as key points for other purposes, such as identifying the position of the gun's muzzle to render a muzzle flash.

The Hierarchy tutorial renders a hierarchical model of an arm. This tutorial is interactive; the relative angles of the nodes can be changed with keyboard commands. The angles are bound within certain values, so the model will stop bending once these values are exceeded. These commands are as follows:

Table 6.1. Hierarchy Tutorial Key Commands

Node Angle	Increase/Left	Decrease/Right
Base Spin	**A**	**D**
Arm Raise	**W**	**S**
Elbow Raise	**R**	**F**
Wrist Raise	**T**	**G**
Wrist Spin	**Z**	**C**
Finger Open/Close	**Q**	**E**

Figure 6.9. Hierarchy Project



The structure of the tutorial is very interesting and shows off a number of important data structures for doing this kind of rendering.

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 `display` simply does the usual setup work and calls `Hierarchy::Draw()`, where the real work happens.

The `Draw` function looks like this:

Example 6.5. Hierarchy::Draw

```
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,
            glm::value_ptr(modelToCameraStack.Top()));
        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,
            glm::value_ptr(modelToCameraStack.Top()));
        glDrawElements(GL_TRIANGLES, ARRAY_COUNT(indexData),
            GL_UNSIGNED_SHORT, 0);
        modelToCameraStack.Pop();
    }

    //Draw main arm.
    DrawUpperArm(modelToCameraStack);

    glBindVertexArray(0);
    glUseProgram(0);
}
```

The program and VAO binding code should look familiar, but most of the code should be fairly foreign.

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 translation is applied to the stack first, followed by an X rotation based on the current angle. 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. After this, a translation and scale are applied to the matrix stack. 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. What is the purpose of this code?

What we see here is a difference between the transforms that need to be propagated to child nodes, and the transforms necessary to properly position the model(s) for rendering this particular node. It is often useful to have source mesh data where the model space of the mesh is not the same space that our node transform requires.

In our case, we do this because we know that all of our pieces are 3D rectangles. A 3D rectangle is really just a cube with scales and translations applied to them. The scale makes the cube into the proper size, and the translation positions the origin point for our model space.

Rather than have this extra transform, we could have created 9 or so actual rectangle meshes, one for each rendered rectangle. However, this would have required more buffer object room and more vertex attribute changes when these were simply unnecessary. The vertex shader runs no slower this way; it's still just multiplying by matrices. And the minor CPU computation time is exactly that: minor.

This concept is very useful, even though it is not commonly talked about to the point where it gets a special name. As we have seen, it allows easy model reuse, but it has other properties as well. For example, it can be good for data compression. There are ways to store values on the range [0, 1] or [-1, 1] in 16 or 8 bits, rather than 32-bit floating point values. If you can apply a simple scale+translation transform to go from this [-1, 1] space to the original space of the model, then you can cut your data in half (or less) with virtually no impact on visual quality.

Each section of the code where it uses an extra transform happens between a `MatrixStack::Push` and `MatrixStack::Pop`. This preserves the node's matrix, so that it may be used for rendering with other nodes.

At the bottom of the base drawing function is a call to draw the upper arm. That function looks similar to this 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.

Matrix Stack Conventions

There are two possible conventions for matrix stack behavior. The caller could be responsible for pushing and popping the matrix, or the callee (the function being called) could be responsible for this. These are called caller-save and callee-save.

In caller-save, what it is saying is that a function that takes a matrix stack should feel free to do whatever they want to the current matrix, as well as push/pop as much as they want. However, the callee *must* not pop more than they push, though this is a general requirement with any function taking a matrix stack. After all, a stack does not report how many elements it has, so you cannot know whether someone pushed anything at all.

In callee-save, what the convention is saying is that a function must be responsible for any changes it wants to make to the matrix stack. If it wants to change the matrix stack, then it must push first and pop after using those changes.

Callee-save is probably a better convention to use. With caller-save, a function that takes a matrix stack must be assumed to modify it (if it takes the object as a non-const reference), so it will have to do a push/pop. Whereas with callee-save, you only push/pop as you explicitly need: at the site where you are modifying the matrix stack. It groups the code together better.

In Review

In this tutorial, you have learned the following:

- Coordinate systems (spaces) are defined by 3 basis axes and a position.
- The transformation from one 3D space to another can be defined by a 4x4 matrix, which is constructed from the 3 basis axes and the position.
- Model space is the coordinate system that a particular model occupies, relative to camera space. Other models can have model spaces that depend on the model space of other models.
- Scale, translation, and rotation transformations have specific matrix forms.
- Transformations can be composed via matrix multiplication. All transformations for a model can be folded into a single matrix, which a vertex shader can execute at a fixed rate. Therefore, complex transforms are no slower to execute (for the graphics chip) than simple ones.
- The order that successive transforms are applied in matters. Matrix multiplication is not commutative, and neither is object transformation.
- Successive transformations can be used to build hierarchies of objects, each dependent on the accumulated transformations of lower ones. This is done using a matrix stack.

Further Study

Try doing these things with the given programs.

- In the Translation tutorial, we had two objects that rotated around a specific point. This was achieved by computing the offset for the rotated position on the CPU, not through the use of a rotation transformation. Change this code to use rotation transformations instead. Make sure that the orientation of the objects do not change as they are being rotated around; this will require applying more than one rotation transformation.
- Reverse the order of rotations on the wrist in the Hierarchy tutorial. Note how this affects the ability to adjust the wrist.
- Reimplement the Hierarchy tutorial, instead using a more generic data structure. Have each node be a struct/class that can be attached to an arbitrary node. The scene will simply be the root node. The individual angle values should be stored in the node object. The node should have a render function that will render this node, given the matrix stack. It would render itself, then recursively render its children. The node would also have a way to define the size (in world-space) and origin point of the rectangle to be drawn.
- Given the generalized Hierarchy code, remove the matrix stack. Use matrix objects created on the C++ stack instead. The node render function would take a const& to a matrix rather than a matrix stack reference.

Glossary

space, coordinate system
: This defines what the coordinates used to refer to positions actually mean. Coordinate systems have a dimensionality (the number of coordinates), an basis vector for each of the dimensions, and an origin position. A coordinate system of 3 dimensions therefore is defined by 3 vectors and a position. The X, Y and Z coordinates in that coordinate system refer to the value you get when you multiply the X, Y, Z values into the X, Y, and Z axes, then add the origin position to those values.

basis vector
: One of the vectors that define a coordinate system. The basis vectors of a coordinate system do not have to be orthogonal or of unit length.

transformation
: The process of moving objects defined in one space to be defined in another space.

model space
: The space that a particular model is expected to be in. Vertex data stored in buffer objects is expected to be in model space.

translation transform
: A transform between two spaces, where the origin of the spaces are not in the same location. This causes objects to shift as they are transformed between the two spaces.

identity matrix
: The matrix I such that the following is true: MI = M, for any matrix M. Identity matrices only exist for square matrices (a matrix with the same number of columns and rows). An identity matrix consists of a matrix with ones along the diagonal from the top-left to the lower-right, and zeros everywhere else.

scale transform
: A transform between two spaces where the axis vectors of the source space are longer or shorter than the corresponding axis vectors in the destination space. This causes objects to stretch or shrink along the axes as they are transformed between the two spaces.

scale inversion
: Performing a scale by a negative value. This is perfectly allowed, though it can change the winding order of triangles, depending on the axis being scaled.

orthogonal
: Two vectors are orthogonal if they are perpendicular to each other. Three vectors are othogonal if each vector is perpendicular to the other two.

rotation transform, orientation transform
: A transform between two spaces, where the axis vectors of the two spaces are not pointed in the same direction, but the angle between the axis vectors stay the same. This cause a reorientation of objects as they are transformed from the initial space to the destination space.

hierarchical model
: Models can be conceptually composed of multiple independent pieces in a hierarchy. The space of each component of that hierarchy is stored relative to its parent in the hierarchy.

node
: A single model space transform within a hierarchy of model transforms. The node's transform is stored relative to the transform of the node beneath it, called the parent. Nodes can have a single parent node and multiple child nodes; the child nodes' transforms are relative to this node's space.

Chapter 7. World in Motion

In this tutorial, we will show how to build a world of objects with a dynamic, moving camera.

World Space

In the perspective projection tutorial, we defined a projection matrix that transforms objects from a specific camera space to clip-space. This camera space was defined primarily to make our perspective transformation as simple as possible. The camera itself sits immobile at the origin (0, 0, 0). The camera always looks down the Z axis, with objects that have a negative Z being considered in front of the camera.

All of the tutorials we have seen since then have had model transformations that go directly to camera space. While this functions, it is not as useful as it could be. Camera space is not a particularly flexible space. If we want to have a moving camera, obviously something needs to change.

We could modify our perspective matrix generation functions, so that we can project onto a camera that has an arbitrary position and orientation. But really, that's too much work; camera space itself works just fine for our needs. It would be easier to just introduce an additional transformation.

Defining the World

Right now, the problem is that we transform all of the objects from their individual model spaces to camera space directly. The only time the objects are in the same space relative to one another is when they are in camera space. So instead, 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. The camera itself will also have a particular position and orientation in world space. And since the camera has a known space, with a known position and orientation relative to world space, we have a transformation from world space to camera space. This also neatly explains why camera space is so named: it is the space where the world is expressed relative to the camera.

So, how do we define world space? Well, we defined model space by fiat: it's the space the vertex positions are in. Clip-space was defined for us. The only space thus far that we have had a real choice about is camera space. And we defined that in a way that gave us the simplest perspective projection matrix.

The last part gives us a hint. What defines a space is not the matrix that transforms to that space, but the matrix that transforms *from* that space. And this makes sense; a transformation matrix contains the basis vector and origin of the source space, as expressed in the destination coordinate system. Defining world space means defining the world-to-camera transform.

We can define this transform with a matrix. But something said earlier gives us a more user-friendly mechanism. We stated that one of the properties of world space is that the camera itself has a position and orientation in world space. That position and orientation, expressed in world space, comprises the camera-to-world transform; do note the order: “camera-to-world.” We want the opposite: world-to-camera.

The positioning is quite simple. Given the position of the camera in world space, the translation component of the world-to-camera matrix is the negation of that. This translates world space positions to be relative to the camera's position. So if the camera's position in world space is (3, 15, 4), then the translation component of the world-to-camera matrix is (-3, -15, -4).

The orientation is a bit more troublesome. There are many ways to express an orientation. In the last tutorial, we expressed it as a rotation about an axis. For a camera, it is much more natural to express the orientation relative to something more basic: a set of directions.

What a user most wants to do with a camera is look at something. So the direction that is dead center in camera space, that is directly along the -Z axis, is one direction vector. Another thing users want to do with cameras is rotate them around the viewing direction. So the second direction is the direction that is “up” in camera space. In camera space, the up direction is +Y.

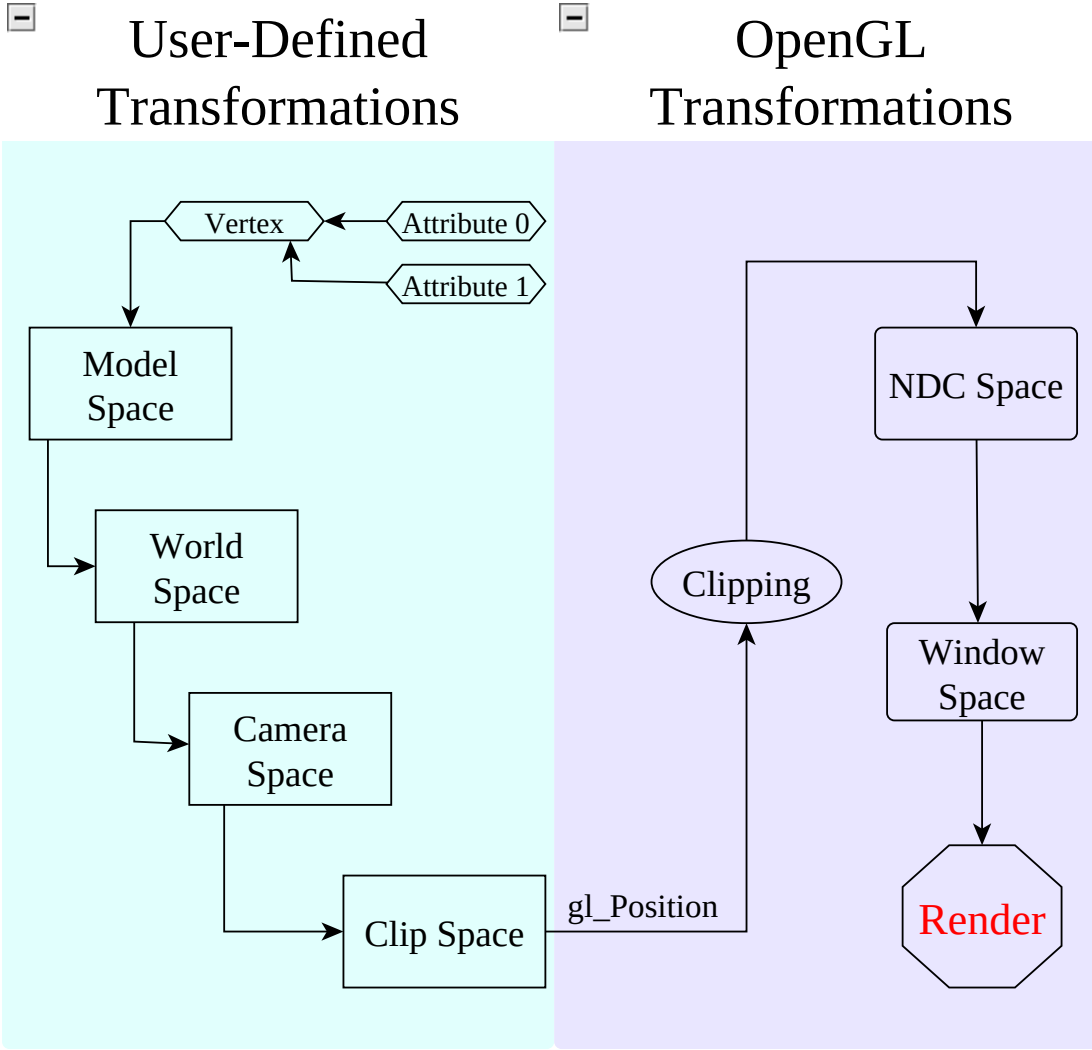
We could specify a third direction, but that is unnecessary; it is implicit based on the other two and a single assumption. Because we want this to be a pure orientation matrix, the three basis vectors must be perpendicular to one another. Therefore, the third direction is the direction perpendicular to the other two. Of course, there are two vectors perpendicular to the two vectors. One goes left relative to the camera's orientation and the other goes right. By convention, we pick the direction that goes right.

So we define the camera's orientation (in world space) as being the viewing direction and the up direction. Oftentimes, a view direction is not the most useful way to orient a camera; it is often useful to select a point in world space to look at.

Therefore, we can define the camera-to-world (again, note the order) transform based on the camera's position in the world, a target point to look at in the world, and an up direction in the world. To get the world-to-camera transform, we need to expend some effort.

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.

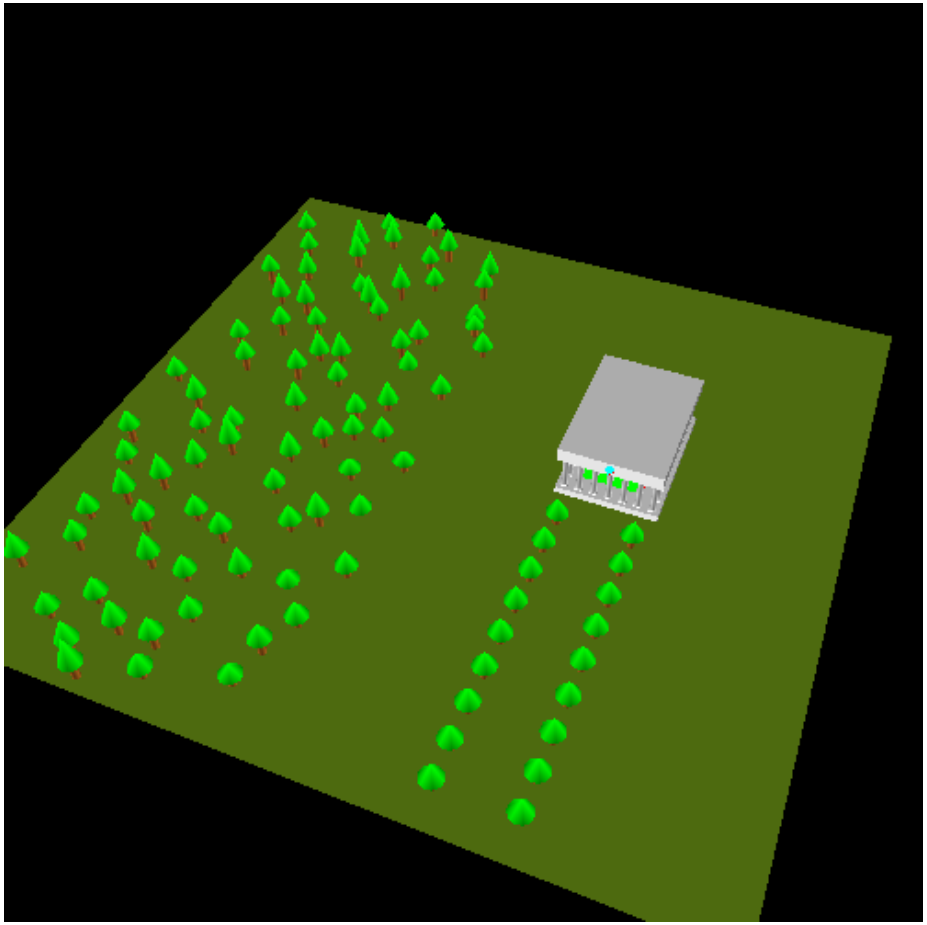
Figure 7.1. Full Vertex Transformation Pipeline



Aerial View

The tutorial project World Space demonstrates the use of a mobile camera in a world-space scene.

Figure 7.2. World Space Scene



The controls for this tutorial are as follows:

Table 7.1. World Space Controls

Function	Increase/Left	Decrease/Right
Move camera target up/down	**E**	**Q**
Move camera target horizontally	**A**	**D**
Move camera target vertically	**W**	**S**
Rotate camera horizontally around target	**L**	**J**
Rotate camera vertically around target	**I**	**K**

Function	Increase/Left	Decrease/Right
Move camera towards/away from target	**U**	**O**

In addition, if you hold down the shift key while pressing any of these keys, then the affected control will be much slower. This allows for more precision movements. The spacebar will toggle the appearance of an object indicating the position of the camera point.

This world is more complicated than anything we've seen up until now. There are a lot of objects being rendered, and most of them are composed out of multiple objects.

This tutorial is the first to incorporate some of the features of the Unofficial OpenGL SDK. Specifically, it uses the GL Util library's `glutil::MatrixStack` class, which implements a matrix stack very much like we saw in the last tutorial. The main difference is that we do not use explicit push/pop functions. To push a matrix onto the stack, we instead use a stack object, `glutil::PushStack`. The constructor pushes the matrix and the destructor automatically pops it. This way, we can never stack overflow or underflow.[1]

The tutorial also is the first to use the Framework's mesh class: `Framework::Mesh`. It implements mesh loading from an XML-based file format. We will discuss some of the functioning of this class in detail in the next section. For now, let us say that this class's `Mesh::Render` function is equivalent to binding a vertex array object, rendering with one or more `glDraw*` calls, and then unbinding the VAO. It expects a suitable program object to be bound to the context.

Multiple Programs

Speaking of suitable program objects, this will be the first tutorial that uses more than one program object. This is the perfect time to bring up an important issue.

Separate programs do not share uniform locations. That is, if you call `glGetUniformLocation` on one program object, it will not necessarily return the same value from a different program object. This is regardless of any other circumstance. You can declare the uniforms with the same name, with the same types, in the same order, but OpenGL will not *guarantee* that you get the same uniform locations. It does not even guarantee that you get the same uniform locations on different run-through of the same executable.

This means that uniform locations are local to a program object. Uniform data is also local to an object. For example:

Example 7.1. Window Resizing

```
void reshape (int w, int h)
{
    glutil::MatrixStack persMatrix;
    persMatrix.Perspective(45.0f, (w / (float)h), g_fzNear, g_fzFar);

    glUseProgram(UniformColor.theProgram);
    glUniformMatrix4fv(UniformColor.cameraToClipMatrixUnif, 1, GL_FALSE, glm::value_ptr(persMatrix.
    glUseProgram(ObjectColor.theProgram);
    glUniformMatrix4fv(ObjectColor.cameraToClipMatrixUnif, 1, GL_FALSE, glm::value_ptr(persMatrix.T
    glUseProgram(UniformColorTint.theProgram);
    glUniformMatrix4fv(UniformColorTint.cameraToClipMatrixUnif, 1, GL_FALSE, glm::value_ptr(persMat
    glUseProgram(0);

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

Here's our new function of the window reshaping function, using the `MatrixStack::Perspective` function to generate the correct perspective projection matrix. Notice that we must bind the 3 separate programs and individually update each one's uniform for the camera-to-clip matrix.

[1]This technique, using constructors and destructors to do this kind of scope-bounded work, is called Resource Acquisition Is Initialization (RAII). It is a common C++ resource management technique. You can find more information about it online [http://www.hackcraft.net/raii/]. If you are unfamiliar with it, I suggest you become familiar with it.

Attributes and Programs

Our three programs are made from 2 vertex shaders and 3 fragment shaders. The differences between these shaders is based on where they get their color information from.

We create three programs. One that expects a per-vertex color and uses that to write the fragment color. One that expects a per-vertex color and multiplies that with a uniform color to determine the fragment color. And one that does not take a per-vertex color; it simply uses the uniform color as the fragment's color. All of these do the same positional transformation, which is a series of three matrix multiplications:

Example 7.2. Position-only Vertex Shader

```
#version 330

layout(location = 0) in vec4 position;

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

void main()
{
    vec4 temp = modelToWorldMatrix * position;
    temp = worldToCameraMatrix * temp;
    gl_Position = cameraToClipMatrix * temp;
}
```

Mismatched Attributes and Programs

You may be wondering what happens if there is a mis-match between the attributes provided by a VAO and the vertex shader inputs. For example, we could use the position-only vertex shader with a mesh that provides attributes 0 and 1, with 0 being the position and 1 being the color.

OpenGL is actually very lenient about this sort of thing. It also goes through some effort to fully define what information the vertex shader gets in the event of a mismatch.

A VAO can provide attributes that a vertex shader does not use without penalty. Well, there may be a performance penalty for reading unused information, but it will still render correctly.

If a vertex shader takes attributes that the VAO does not provide, then the value the vertex shader gets will be a vector of (0, 0, 0, 1). If the vertex shader input vector has fewer than 4 elements, then it fills them in in that order. A vec3 input that is not provided by the VAO will be (0, 0, 0).

Speaking of which, if a VAO provides more components of an attribute vector than the vertex shader expects (the VAO provides 4 elements, but the vertex shader input is a vec2), then the vertex shader input will be filled in as much as it can be. If the reverse is true, if the VAO does not provide enough components of the vector, then the unfilled values are always filled in from the (0, 0, 0, 1) vector.

Camera of the World

The main rendering function implements the world-space and camera code. It begins by updating the world-to-camera matrix.

Example 7.3. Upload World to Camera Matrix

```
const glm::vec3 &camPos = ResolveCamPosition();
```



```
glutil::MatrixStack camMatrix;
camMatrix.SetMatrix(CalcLookAtMatrix(camPos, g_camTarget, glm::vec3(0.0f, 1.0f, 0.0f)));

glUseProgram(UniformColor.theProgram);
glUniformMatrix4fv(UniformColor.worldToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(camMatrix.Top(
glUseProgram(ObjectColor.theProgram);
glUniformMatrix4fv(ObjectColor.worldToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(camMatrix.Top()
glUseProgram(UniformColorTint.theProgram);
glUniformMatrix4fv(UniformColorTint.worldToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(camMatrix.
glUseProgram(0);
```

The function `ResolveCamPosition` computes the camera position, based on the user's input. `CalcLookAtMatrix` is the function that takes a camera position in the world, a point in the world to look at, and an up vector, and uses it to compute the world-to-camera matrix. We will look at that a bit later.

Speaking of which, let's look at how `ResolveCamPosition` works. The basic idea of this camera system is that there is a target point, which is mobile. The camera's position is computed relative to this target point, so if the target moves, the camera will follow it perfectly.

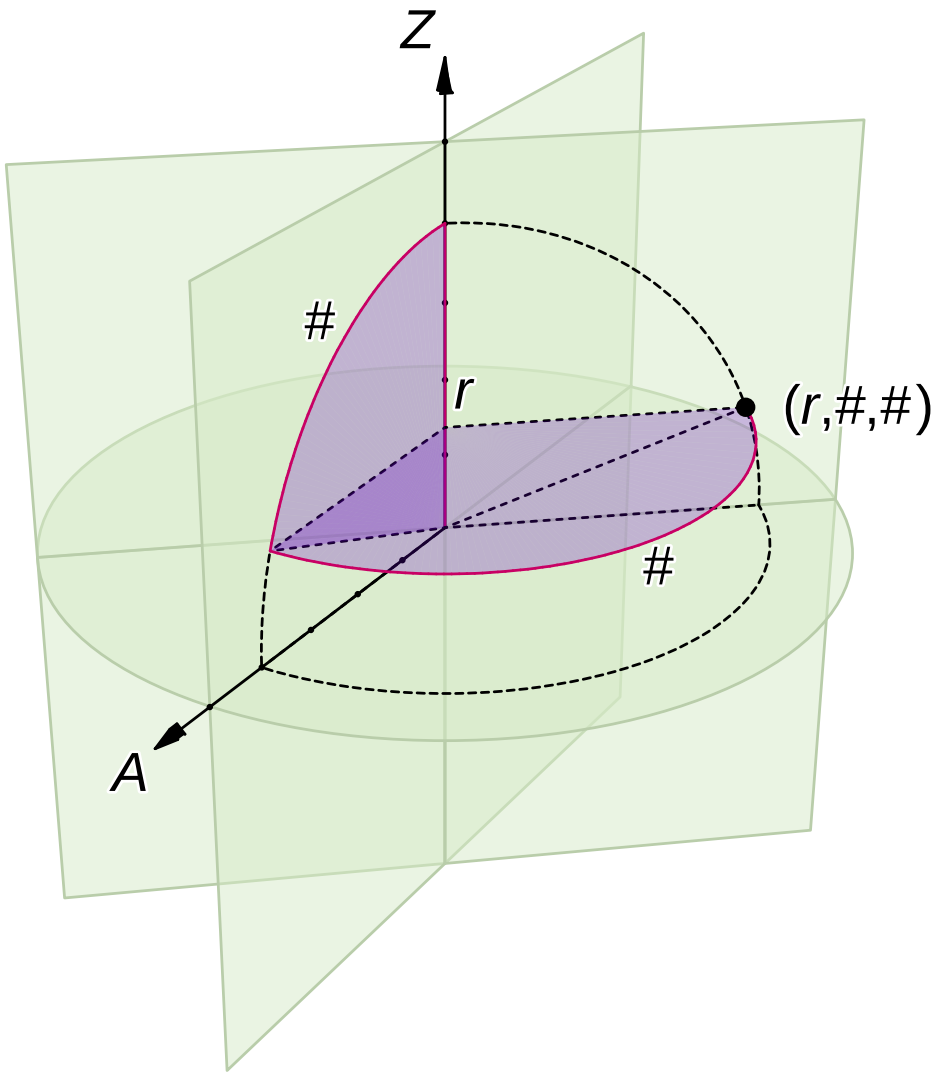
To do this, we use a special coordinate system trick. Instead of storing the relative position of the camera in a normal coordinate system, we instead use a *spherical coordinate system*, also known as *polar coordinates*.

Previously, we said that a coordinate system was defined by a series of vectors and an origin point. This was a useful simplification of the possibilities; this is true of any coordinate system that follows the rules of *Euclidean geometry*. Spherical coordinates (among many others) are non-Euclidean. For example, in Euclidean geometry, the sum of the angles of any triangle will add up to 180 degrees exactly. This is not true of spherical geometries or spherical coordinates. This is because “lines” in spherical geometries are curves when seen relative to Euclidean geometries.

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 “#” (phi), represents the angle in the elliptical plane. This value extends on the range [0, 360). The third value, called “#” (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.

This is much easier to see in diagram form:

Figure 7.3. Spherical Coordinates



This 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`.

Example 7.4. Spherical to Euclidean Transform

```
glm::vec3 ResolveCamPosition()
{
    glutil::MatrixStack tempMat;

    float phi = Framework::DegToRad(g_sphereCamRelPos.x);
    float theta = Framework::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.

The `dirToCamera` is just a direction vector. Only by scaling it by the radius (`g_sphereCamRelPos.z`) do we get the full decomposition from spherical coordinates to Euclidean. Applying the camera target as an offset is what keeps the camera's position relative to the target.

All of the above simply gets us a position for the camera and a location where the camera is looking. The matrix is computed by feeding these values into `CalcLookAtMatrix`. It takes a position for the camera, a point in the world that the camera should be looking in, and a direction in world-space that should be considered "up" based on where the camera is looking.

The implementation of this function is non-trivial. We will not go into detail explaining how it works, as it involves a lot of complex math concepts that have not been introduced. Using the function is is much easier than understanding how it works. Even so, there is one major caveat with this function (and any function of the like).

It is very important that the "up" direction is not along the same line as the direction from the camera position to the look at target. If up is very close to that direction then the generated matrix is no longer valid and unpleasant things will happen.

Since it does not make physical sense for "up" to be directly behind or in front of the viewer, it makes a degree of sense that this would likewise produce a nonsensical matrix. This problem usually crops up in camera systems like the one devised here, where the camera is facing a certain point and is rotating around that point, without rotating the up direction at the same time. In the case of this code, the up/down angle is clamped to never get high enough to cause a problem.

World Rendering

Once the camera matrix is computed, it is farmed out to each of the programs. After that, rendering is pretty simple.

The meshes we have loaded for this tutorial are unit sized. That is, they are one unit across in their major axes. They also are usually centered at the origin in their local coordinate system. This make it easy to scale them to arbitrary sizes for any particular use.

The ground is based on the unit plane mesh. This is just a square with the sides being unit length. This is rendered by the following code:

Example 7.5. Draw the Ground

```
glutil::PushStack push(modelMatrix);
```

```
modelMatrix.Scale(glm::vec3(100.0f, 1.0f, 100.0f));

glUseProgram(UniformColor.theProgram);
glUniformMatrix4fv(UniformColor.modelToWorldMatrixUnif, 1, GL_FALSE, glm::value_ptr(modelMatrix.Top
glUniform4f(UniformColor.baseColorUnif, 0.302f, 0.416f, 0.0589f, 1.0f);
g_pPlaneMesh->Render();
glUseProgram(0);
```

The unit plane mesh has no color attribute, so we use the `UniformColor` program. We apply a scale matrix to the model stack, so that the 1x1 plane becomes 100x100 in size. After setting the color, the plane is rendered.

All of the trees are drawn from the `DrawForest` function.

Example 7.6. DrawForest Function

```
void DrawForest(glutil::MatrixStack &modelMatrix)
{
    for(int iTree = 0; iTree < ARRAY_COUNT(g_forest); iTree++)
    {
        const TreeData &currTree = g_forest[iTree];

        glutil::PushStack push(modelMatrix);
        modelMatrix.Translate(glm::vec3(currTree.fXPos, 0.0f, currTree.fZPos));
        DrawTree(modelMatrix, currTree.fTrunkHeight, currTree.fConeHeight);
    }
}
```

This function iterates over a large table and draws a tree for each element in that table. The table entries determine where in world space the tree is drawn and how tall it is. The location is stored as a translation in the matrix stack (after pushing), and the tree attributes are passed to the `DrawTree` function to render.

The Parthenon is drawn from the `DrawParthenon` function. Since this draw function, like `DrawTree`, expects the matrix stack to transform it to its world-space position, the first step we see is applying a translation matrix to the stack.

Example 7.7. Call to DrawParthenon

```
glutil::PushStack push(modelMatrix);
modelMatrix.Translate(glm::vec3(20.0f, 0.0f, -10.0f));

DrawParthenon(modelMatrix);
```

The actual `DrawParthenon` function is pretty simple. It uses `DrawColumn` to draw all of the columns at the various locations around the building. It draws scaled cubes for the base and ceiling, and uses the colored version of the cube for the headpiece at the front and the interior of the building. Columns are scaled cubes and cylinders.

Non-World Rendering

The last part of the `display` function is more interesting. Pressing the **Spacebar** toggles the drawing of a representation of the camera target point. Here is how it gets drawn:

Example 7.8. Draw Camera Target

```
glDisable(GL_DEPTH_TEST);
glm::mat4 idenity(1.0f);

glutil::PushStack push(modelMatrix);
```



```
glm::vec3 cameraAimVec = g_camTarget - camPos;
modelMatrix.Translate(0.0f, 0.0, -glm::length(cameraAimVec));
modelMatrix.Scale(1.0f, 1.0f, 1.0f);

glUseProgram(ObjectColor.theProgram);
glUniformMatrix4fv(ObjectColor.modelToWorldMatrixUnif, 1, GL_FALSE,
    glm::value_ptr(modelMatrix.Top()));
glUniformMatrix4fv(ObjectColor.worldToCameraMatrixUnif, 1, GL_FALSE,
    glm::value_ptr(idenity));
g_pCubeColorMesh->Render();
glUseProgram(0);
glEnable(GL_DEPTH_TEST);
```

The first thing that happens is that the depth test is turned off. This means that the camera target point will always be seen, no matter where it is. So if you move the target point inside the building or a tree, you will still see it. This is a useful technique for UI-type objects like this.

The next important thing is that the world-to-camera matrix is set to identity. This means that the model-to-world matrix functions as a model-to-camera matrix. We are going back to positioning objects in front of the camera, which is what we actually want. The cube is translated down the -Z axis, which positions it directly in front of the camera. It positions the square at the same distance from the camera as the camera would be from the target point.

For the last few tutorials, we have been building up a transformation framework and hierarchy. Model space to world space to camera space to clip space. But the important thing to remember is that this framework is only useful to you if it does what you want. If you need to position an object directly in front of the camera, then simply remove world space from the equation entirely and deal directly with camera space.

We could even turn the depth test back on, and the camera target would interact correctly with the rest of the world. It is a part of the world, even though it seems like it goes through a different transform pipe.

Indeed, you could render part of a scene with one perspective matrix and part with another. This is a common technique for first-person shooter games. The main world is rendered with one perspective, and the part of the first-person character that is visible is rendered with another matrix.

Do not get so caught up in “the way things ought to be done” that you forget what you could have done if you broke free of the framework. Never hold so tightly to one way of doing something that it prevents you from seeing how to do something you need to much easier. For example, we could have applied the reverse of the camera matrix to the model-to-world matrix. Or we could just get rid of that matrix altogether and make everything work very easily and simply.

Primitive Drawing

We skipped over how the `Mesh::Render` function and mesh loading works. So let's cover that now.

The XML-based mesh files define a number of vertex attribute arrays, followed by a number of rendering commands. The format fully supports all features of OpenGL, including options not previously discussed. One of these options deals with how vertex data is interpreted by OpenGL.

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. They are both used in the cylinder mesh, so let's take a look at that.

Example 7.9. Cylinder Mesh File

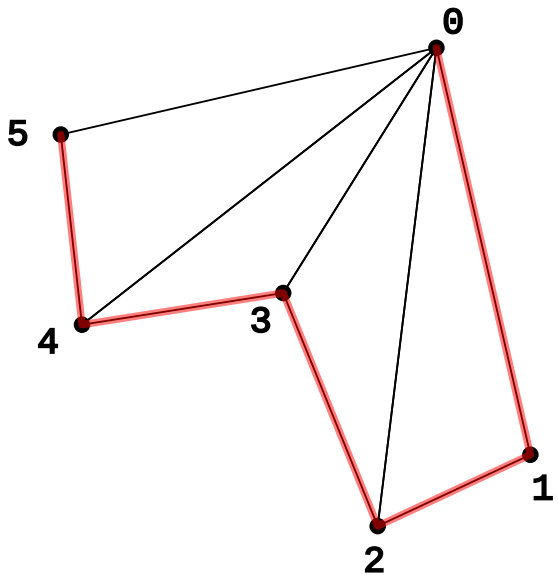
```
<indices cmd="tri-fan" type="ushort" >0 1 3 5 7 9 11 ...</indices>
<indices cmd="tri-fan" type="ushort" >61 60 58 56 54 ...</indices>
<indices cmd="tri-strip" type="ushort" >1 2 3 4 5 6 7 8 ...</indices>
```

Each “indices” element maps to a call to `glDrawElements` with the given index array. The “cmd” attribute determines the primitive that will be passed to `glDrawElements`. The value “triangles” means to use the `GL_TRIANGLES` primitive.

The “tri-fan” used above means to use the `GL_TRIANGLE_FAN` primitive. This primitive has the triangle primitive type, so this vertex stream will generate triangles. But it will generate them using a different representation.

`GL_TRIANGLES` takes each independent set of 3 vertices as a single triangle. `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 `GL_TRIANGLE_FAN` will generate triangles as follows: (0, 1, 2), (0, 2, 3), (0, 3, 4), …. Visually, a triangle fan looks like this:

Figure 7.4. Triangle Fan



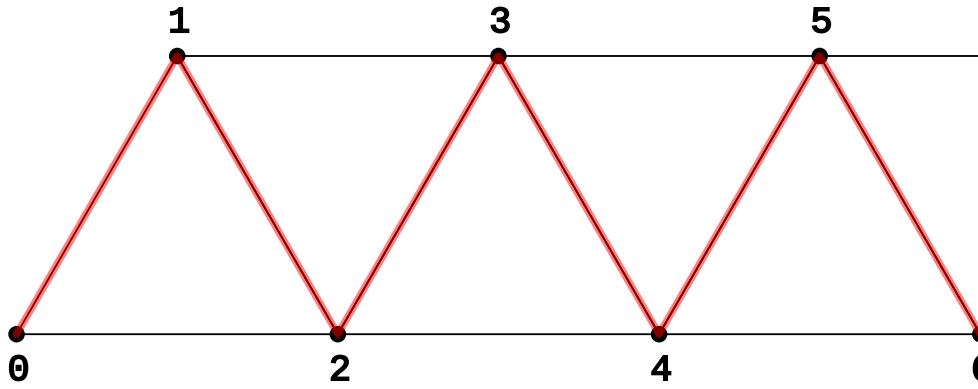
The numbers represent the order that the vertices are in in the vertex stream. The red line shows the triangle edges that are directly specified by the vertex stream. All other edges are generated automatically by the primitive representation.

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 cylinder mesh uses two fans to cap render the end pieces of the cylinder.

The “tri-strip” in the cylinder mesh represents the `GL_TRIANGLE_STRIP` primitive. As the name suggests, it has a triangle primitive type. The 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:

Figure 7.5. Triangle Strip



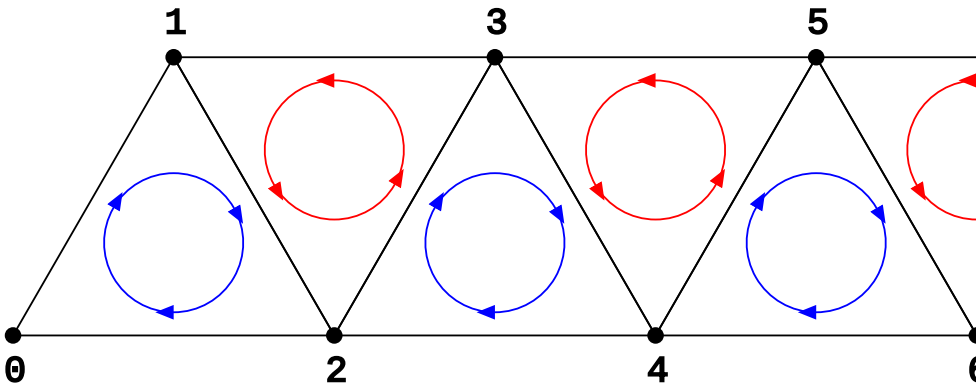
Like with triangle fans, the number of vertices must be 3 or greater, but can be any number otherwise. For N vertices in a stream, triangle strips will generate N-2 triangles.

The cylinder mesh uses a single triangle strip to render the sides of the cylinder.

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:

Figure 7.6. Triangle Strips with Winding Order



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.

Shared Uniforms

The World Space example had a few annoyances in it. Of particular pain was the fact that, whenever the perspective projection matrix or the world-to-camera matrix changed, we had to change uniforms in 3 programs. They all used the same value; it seems strange that we should have to go through so much trouble to change these uniforms.

Also, 3 programs is a relatively simple case. When dealing with real examples, the number of programs can get quite large.

There is a way to share uniforms between programs. To do this, we use a buffer object to store uniform data, and then tell our programs to use this particular buffer object to find its uniform data. A buffer object that stores uniforms is commonly called a *uniform buffer object*.

It is important to understand that there is nothing special about a uniform buffer. Any of the things you could do with a regular buffer object can be done with a uniform buffer object. You can bind it to the `GL_ARRAY_BUFFER` and use it for vertex data, you can use it for indexed rendering with `GL_ELEMENT_ARRAY_BUFFER`, and many other things that buffer objects can be used for. Now granted, that doesn't mean that you should, only that you can.

The example World with UBO uses a uniform buffer object to store the camera and perspective matrices.

Uniform Blocks

This begins with how the vertex shaders are defined.

Example 7.10. UBO-based Vertex Shader

```
#version 330

layout(location = 0) in vec4 position;

layout(std140) uniform GlobalMatrices
{
    mat4 cameraToClipMatrix;
    mat4 worldToCameraMatrix;
};

uniform mat4 modelToWorldMatrix;

void main()
{
    vec4 temp = modelToWorldMatrix * position;
    temp = worldToCameraMatrix * temp;
    gl_Position = cameraToClipMatrix * temp;
}
```

The definition of GlobalMatrices looks like a struct definition, but it is not. It defines a *uniform block*. A uniform block is a series of uniform definitions whose data is not stored in the program object, but instead must come from a uniform buffer.

The name GlobalMatrices is used to identify this particular uniform block. This block has two members, both of the of mat4 type. The order of the components in a uniform block is very important.

Notices that nothing else needs to change in the vertex shader. The `modelToWorldMatrix` is unchanged, and the use of the components of the uniform block do not even need to be scoped with the GlobalMatrices name.

The “layout(std140)” part modifies the definition of the uniform block. Specifically, it specifies the *uniform block layout*.

Buffer objects are unformatted arrays of bytes. Therefore, something must determine how the shader interprets a uniform buffer object's contents. OpenGL itself defines this to a degree, but the layout qualifier modifies the definition.

OpenGL is very clear about how each element within a uniform block is laid out. Floating-point values are just the C++ representation of floats, so you can copy them directly from objects like glm::vec4.

Matrices are slightly trickier due to the column-major vs. row-major issue. The `glUniformMatrix*` functions all had a parameter that defines what order the matrix data given to the function is in. Similarly, a “layout” qualifier can specify “row-major” or “column-major”; these tell OpenGL how the matrices are stored in the buffer object. The default is “column-major,” and since GLM stores its matrices in column-major order, we can use the defaults.

What OpenGL does not directly specify is the spacing *between* elements in the uniform block. This allows different hardware to position elements where it is most efficient for them. Some shader hardware can place 2 vec3's directly adjacent to one another, so that they only take up 6 floats. Other hardware cannot handle that, and must pad each vec3 out to 4 floats.

Normally, this would mean that, in order to set any values into the buffer object, you would have to query the program object for the byte offsets for each element in the uniform block.

However, by using the “std140” layout, this is not necessary. The “std140” layout has an explicit layout specification set down by OpenGL itself. It is basically a kind of lowest-common-denominator among the various different kinds of graphics hardware. The upside is that it allows you to easily know what the layout is without having to query it from OpenGL. The downside is that some space-saving optimizations may not be possible on certain hardware.

One additional feature of “std140” is that the uniform block is sharable. Normally, OpenGL allows the GLSL compiler considerable leeway to make optimizations. In this instance, if a GLSL compiler detects that a uniform is unused in a program, it is allowed to mark it as unused. `glGetUniformLocation` will return -1. It's actually legal to set a value to a location that is -1, but no data will actually be set.

If a uniform block is marked with the “std140” layout, then the ability to disable uniforms in within that block is entirely removed. All uniforms must have storage, even if this particular program does not use them. This means that, as long as you declare the same uniforms in the same order within a block, the storage for that uniform block will have the same layout in *any* program. This means that multiple different programs can use the same uniform buffer.

The other two alternatives to “std140” are “packed” and “shared”. The default, “shared,” prevents the uniform optimization, thus allowing the block's uniform buffer data to be shared among multiple programs. However, the user must still query layout information about where each uniform is stored. “packed” allows uniform optimization, so these blocks cannot be shared between programs at all.

For our needs, “std140” is sufficient. It's also a good first step in any implementation; moving to “packed” or “shared” as needed should generally be done only as an optimization. The rules for the “std140” layout are spelled out explicitly in the OpenGL Specification.

Uniform Block Indices

Uniforms inside a uniform block do not have individual uniform locations. After all, they do not have storage within a program object; their data comes from a buffer object.

So instead of calling glGetUniformLocation, we have a new function.

```
data.globalUniformBlockIndex =
    glGetUniformBlockIndex(data.theProgram, "GlobalMatrices");
```

The function `glGetUniformBlockIndex` takes a program object and the name of a uniform block. It returns a *uniform block index* that is used to refer to this uniform block. This is similar to how a uniform location value is used to refer to a uniform, rather than directly using its string name.

Uniform Buffer Creation

Now that the programs have a uniform block, we need to create a buffer object to store our uniforms in.

Example 7.11. Uniform Buffer Creation

```
glGenBuffers(1, &g_GlobalMatricesUBO);
glBindBuffer(GL_UNIFORM_BUFFER, g_GlobalMatricesUBO);
glBufferData(GL_UNIFORM_BUFFER, sizeof(glm::mat4) * 2, NULL, GL_STREAM_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

For all intents and purposes, this is identical to the way we created other buffer objects. The only difference is the use of the `GL_UNIFORM_BUFFER` binding target.

The `GL_ARRAY_BUFFER` target has a specific meaning. When something is bound to that target, calling `glVertexAttribPointer` will cause the buffer object bound to that target to become the source for that particular attribute, as defined by the function call. The `GL_ELEMENT_ARRAY_BUFFER` target also has a meaning; it specifies where indices come from for indexed rendering. The element array binding is even stored as part of a VAO's data (recall that the array buffer binding is *not* stored in the VAO).

`GL_UNIFORM_BUFFER` does not really have an intrinsic meaning like these other two. Having something bound to this binding means nothing as far as any other function of OpenGL is concerned. Oh, you can call buffer object functions on it, like glBufferData as above. But it does not have any other role to play in rendering. The main reason to use it is to preserve the contents of more useful binding points. It also communicates to someone reading your code that this buffer object is going to be used to store uniform data.

Note

This is not entirely 100% correct. OpenGL is technically allowed to infer something about your intended use of a buffer object based on what target you *first* use to bind it. So by allocating storage for this buffer in `GL_UNIFORM_BUFFER`, we are signaling something to OpenGL, which can change how it allocates storage for the buffer.

However, OpenGL is *not* allowed to make any behavioral changes based on this. It is still legal to use a buffer allocated on the `GL_UNIFORM_BUFFER` target as a `GL_ARRAY_BUFFER` or in any other buffer object usage. It just may not be as fast as you might want.

We know that the size of this buffer needs to be two glm::mat4's in size. The “std140” layout guarantees this. That and the size of glm::mat4, which just so happens to correspond to how large a GLSL mat4 is when stored in a uniform buffer.

The `reshape` function is guaranteed to be called after our `init` function. That's why we can allocate this buffer without filling in a default matrix. The reshape function is as follows:

Example 7.12. UBO-based Perspective Matrix

```
void reshape (int w, int h)
{
    glutil::MatrixStack persMatrix;
    persMatrix.Perspective(45.0f, (w / (float)h), g_fzNear, g_fzFar);

    glBindBuffer(GL_UNIFORM_BUFFER, g_GlobalMatricesUBO);
    glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4), glm::value_ptr(persMatrix.Top()));
    glBindBuffer(GL_UNIFORM_BUFFER, 0);

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

This function just uses `glBufferSubData` to upload the matrix data to the buffer object. Since we defined the perspective matrix as the first matrix in our uniform block, it is uploaded to byte 0.

The `display` function is what uploads the world-to-camera matrix to the buffer object. It is quite similar to what it used to be:

Example 7.13. UBO-based Camera Matrix

```
const glm::vec3 &camPos = ResolveCamPosition();

glutil::MatrixStack camMatrix;
camMatrix.SetMatrix(CalcLookAtMatrix(camPos, g_camTarget, glm::vec3(0.0f, 1.0f, 0.0f)));

glBindBuffer(GL_UNIFORM_BUFFER, g_GlobalMatricesUBO);
glBufferSubData(GL_UNIFORM_BUFFER, sizeof(glm::mat4), sizeof(glm::mat4), glm::value_ptr(camMatrix.T
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

The world-to-camera matrix is the second matrix, so we start the upload at the end of the previous matrix.

Uniform Buffer Binding

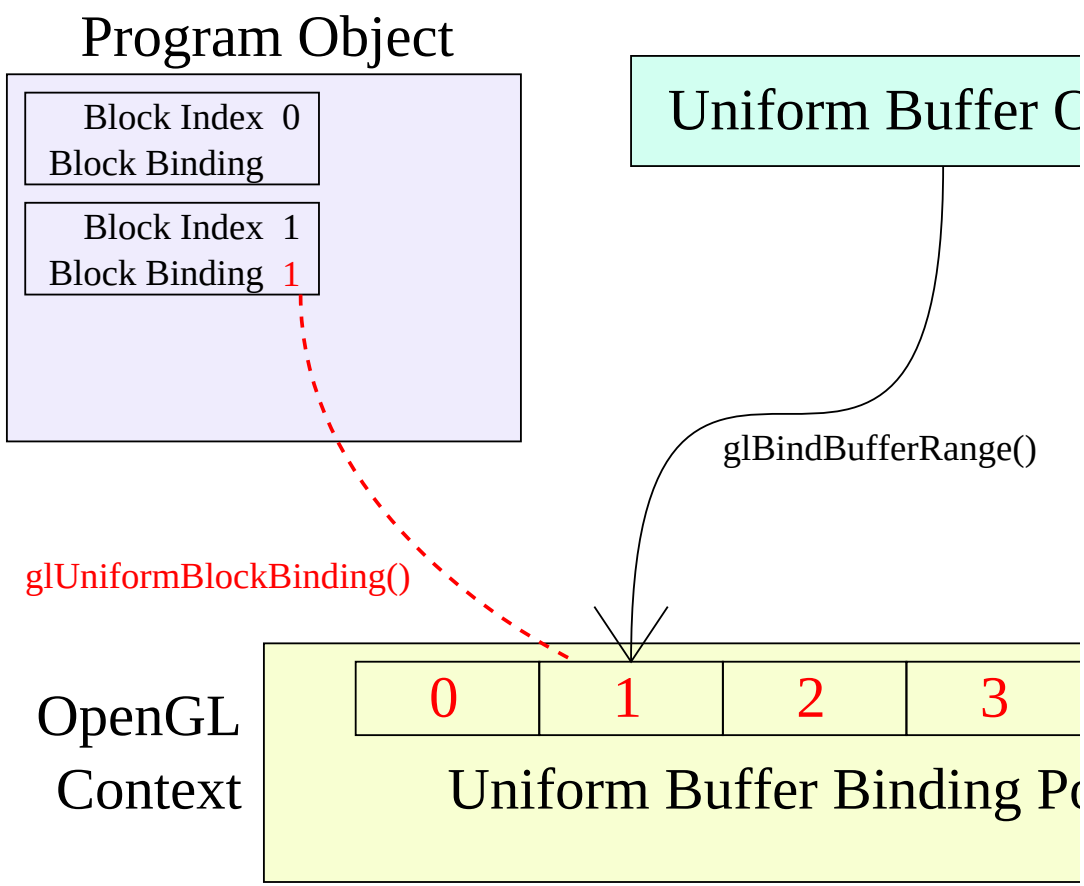
Thus far, we have a uniform buffer object into which we store matrices. And we have a program that has a uniform block that needs a uniform buffer to get its uniforms for. Now, the final step is to create the association between the uniform block in the programs and the uniform buffer object itself.

Your first thought might be that there would be a function like glUniformBuffer that takes a program, a uniform block index, and the uniform buffer to associate that block with. But this is not the case; attaching a uniform buffer to a program's block is more complicated. And this is a good thing if you think about it.

It works like this. The OpenGL context (effectively a giant struct containing each piece of data used to render) has an array of *uniform buffer binding points*. Buffer objects can be bound to each of these binding points. For each uniform block in a program, there is a reference, not to a buffer object, but to one of these uniform buffer binding points. This reference is just a numerical index: 0, 1, 2, etc.

A diagram should make it clearer:

Figure 7.7. Uniform Buffer and Block Binding Points



The program object is given an index that represents one of the slots in the context. The uniform buffer is bound to one of those slots. Therefore, when you render with that program, the uniform buffer that is in the slot specified by the program will be where the program gets its uniform data from.

Therefore, to use a uniform buffer, one first must tell the program object which binding point in the context to find the buffer. This association is made with the `glUniformBlockBinding` function.

```
glUniformBlockBinding(data.theProgram, data.globalUniformBlockIndex,
    g_iGlobalMatricesBindingIndex);
```

The first parameter is the program, the second is the uniform block index queried before. The third is the uniform buffer binding point that this block should use.

After doing this for each program, the uniform buffer must be bound to that binding point. This is done with a new function, `glBindBufferRange`.

```
glBindBufferRange(GL_UNIFORM_BUFFER, g_iGlobalMatricesBindingIndex,
    g_GlobalMatricesUBO, 0, sizeof(glm::mat4) * 2);
```

This functions similarly to `glBindBuffer`; in addition to binding the buffer to the `GL_UNIFORM_BUFFER` target, it also binds the buffer to the given uniform buffer binding point. Lastly, it provides an offset and range, the last two parameters. This allows you to put uniform data in arbitrary places in a buffer object. You could have the uniform data for *several* uniform blocks in several programs all in one buffer object. The range parameters would be how to say where that block's data begins and how big it is.

The reason this is better than directly binding a buffer object to the program object can be seen in exactly where all of this happens. Both of these functions are called as part of initialization code. `glUniformBlockBinding` is called right after creating the program, and similarly `glBindBufferRange` is called immediately after creating the buffer object. Neither one needs to ever be changed. Yes, we change the contents of the buffer object. But where it is bound never changes.

The global constant `g_iGlobalMatricesBindingIndex` is, as the name suggests, global. By convention, all programs get their buffer data from this index. Because of this convention, if we wanted to use a different buffer, we would not have to update every program that needs to use that buffer. Sure, for one or two programs, that would be a simple operation. But real applications can have hundreds of programs. Being able to establish this kind of convention makes using uniform buffer objects much easier than if they were directly associated with programs.

The Viewpoint

In the World Space example, we drew the camera's look-at target directly in camera space, bypassing the world-to-camera matrix. Doing that with uniform buffers would be harder, since we would have to set the uniform buffer value twice in the same draw call. This is not particularly difficult, but it could be a drain on performance.

Instead, we just use the camera's target position to compute a model-to-world matrix that always positions the object at the target point.

Example 7.14. Viewing Point with UBO

```
glDisable(GL_DEPTH_TEST);

glutil::PushStack push(modelMatrix);

modelMatrix.Translate(g_camTarget);
modelMatrix.Scale(1.0f, 1.0f, 1.0f);

glUseProgram(ObjectColor.theProgram);
glUniformMatrix4fv(ObjectColor.modelToWorldMatrixUnif, 1,
    GL_FALSE, glm::value_ptr(modelMatrix.Top()));
g_pCubeColorMesh->Render();
glUseProgram(0);
glEnable(GL_DEPTH_TEST);
```

We do not get the neat effect of having the object always face the camera though. We still shut off the depth test, so that we can always see the object.

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 inches, and you want precision to within 0.25 inches. You want to cover an area with a radius of 1,000 miles, or 63,360,000 inches.

Let us also say that the various pieces of this world all have their own model spaces and are transformed into their appropriate positions via a model-to-world transformation matrix. So the world is assembled out of various parts. This is almost always true to some degree.

Let us also say that, while you do have a large world, you are not concerned about rendering *all* of it at any one time. The part of the world you're interested in is the part within view from the camera. And you're not interested in viewing incredibly distant objects; the far depth plane is going to cull out the world beyond a certain point from the camera.

The problem is that a 32-bit floating-point number can only hold about 7 digits of precision. So towards the edges of the world, at around 63,000,000 inches, you only have a precision out to about ±10 inches at best. This means that vertex positions closer than this will not be distinct from one another. Since your world is modeled down to 0.25 inches of precision, this is a substantial problem. Indeed, even if you go out to 6,000,000 inches, ten-times closer to the middle, you still have only ±1 inch, which is greater than the tolerance you need.

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.

```
#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. And as stated before, the only parts of the world that we are interested in seeing are the parts close to the camera. 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. While you can have depth buffer precision problems, that only happens at far distances from the near plane. Again, since everything is relative to the camera, there is no precision problem.

The only precision problem is with `worldPos`. Or rather, in the `modelToWorldMatrix`.

Think about what `modelToWorldMatrix` and `worldToCameraMatrix` must look like regardless of the precision of the values. 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.

In Review

In this tutorial, you have learned the following:

- World space is an intermediate space between model space and camera space. All objects are transformed into it, and the position/orientation of the camera is specified relative to it.
- OpenGL can processes a sequence of vertex data as triangles in different ways. It can process the vertices as a list of triangles, a triangle fan, or a triangle strip. Each of these has its own way of building triangles from a sequence of vertices.
- Uniform data can be stored in buffer objects, so that multiple programs can share the same uniform. Changing the buffer object data will automatically change the data the programs get.
- It is usually not a good idea to have vertex positions in an explicit world space. Doing so can lead to numerical precision problems if the vertex positions are sufficiently far from 0.

Further Study

Play around with the world space tutorials in the following ways:

- In the World Space tutorial, we use 3 matrices. This requires an extra matrix multiply, which is a wasteful per-vertex cost. Fold the camera matrix into the perspective transformation matrix, so that only two matrices are used. Any time parameters change to one matrix, make sure to recompute both and combine them together before uploading the combined world-to-clip-space matrix.
- Instead of folding the world-to-camera transform into the perspective matrix, fold it into the model-to-world matrix instead. Simply push it onto the same stack as everything else. The function MatrixStack::ApplyMatrix can right-multiply an arbitrary matrix with the current matrix.

OpenGL Functions of Note

glGetUniformBlockIndex	Retrieves the uniform block index for a particular uniform block name from a program.
glUniformBlockBinding	Sets the uniform buffer binding index used by a particular uniform block in a given program.
glBindBufferRange	Binds a buffer object to a particular indexed location, as well as binding it to the given. When used with GL_UNIFORM_BUFFER, it binds the buffer object to a particular uniform buffer binding point. It has range parameters that can be used to effectively bind part of the buffer object to an indexed location.

Glossary

world space	An intermediate space between model space and camera space. Conceptually, all objects are transformed into this space along the transformation chain.
spherical coordinate system, polar coordinates	A three dimensional coordinate system where the three coordinates are not defined by 3 values multiplied by vectors, but by two angles and a radius. One angle specifies rotation around a point in a known plane. The other angle specifies rotation above and below this plane. And the radius specifies the distance from the origin. Spherical coordinates are not a Euclidean geometry.
Euclidean geometry	A specific kind of coordinate system that follows certain axioms. For the sake of brevity, consider it a “regular” coordinate system, one that follows the simple, obvious rules one might expect of a 2D sheet of paper.
vertex stream	An ordered sequence of vertices given to OpenGL by one of the `glDraw*` series of functions.
primitive, rendering mode	The mechanism used by OpenGL for interpreting and rendering a vertex stream. Every `glDraw*` function takes a rendering mode as its first parameter. The primitive mode defines two things: the primitive type and the primitive representation.

primitive type	The kind of object that OpenGL will draw with a vertex stream. OpenGL draws triangles, but it can also draw points or other things.
primitive representation	The way the vertex stream is converted into one or more of the primitive type. Each primitive type consumes a number of vertices; the primitive representation specifies the manor in which the stream of length N is converted into a number M of primitives.
uniform buffer object	A buffer object used to store data for uniform blocks.
uniform block	A named set of uniform definitions. This set of uniforms is not stored in the program object, but instead is taken from a buffer object bound to a buffer object binding point in the OpenGL rendering context. There is a limit on the number of uniform blocks a single program object can use. There is also a per-shader stage limit as well.
uniform block layout	The way a uniform block is layed out by the GLSL compiler. This determines whether uniform blocks can be shared with other programs, and whether the user needs to query the location of each uniform within the block.
uniform block index	A number, queried from a program object, that represents a particular uniform block. This number is used to refer to the uniform block in other functions.
uniform buffer binding points	An array of locations in the OpenGL context where uniform buffers can be bound to. Programs can have their uniform blocks associated with one of the entries in this array. When using such a program, it will use the buffer object bound to that location to find the data for that uniform block.

Chapter 8. Getting Oriented

In this tutorial, we will investigate specific problems with regard to orienting objects.

Gimbal Lock

Remember a few tutorials back, when we said that a rotation matrix is not a rotation matrix at all, that it is an orientation matrix? We also said that forgetting this can come back to bite you. Well, here's likely the most common way.

Normally, when dealing with orienting an object like a plane or spaceship in 3D space, you want to orient it based on 3 rotations about the 3 main axes. The obvious way to do this is with a series of 3 rotations. This means that the program stores 3 angles, and you generate a rotation matrix by creating 3 rotation matrices based on these angles and concatenating them. Indeed, the 3 angles often have special names, based on common flight terminology: yaw, pitch, and roll.

Pitch is the rotation that raises or lowers the front of the object. Yaw is the rotation that turns the object left and right. Roll is the rotation that spins it around the direction of motion. These terms neatly duck the question of what the axes technically are; they are defined in terms of semantic axes (front, left, direction of motion, etc), rather than a specific model space. So in one model space, roll might be a rotation about the X axis, but in another, it might be a rotation about the Z axis.

One of the first problems you will note is that the order you apply these rotations matter. As previously stated, a rotation matrix is an *orientation* transform. 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 the roll first, we have now changed what the axis for the subsequent yaw is.

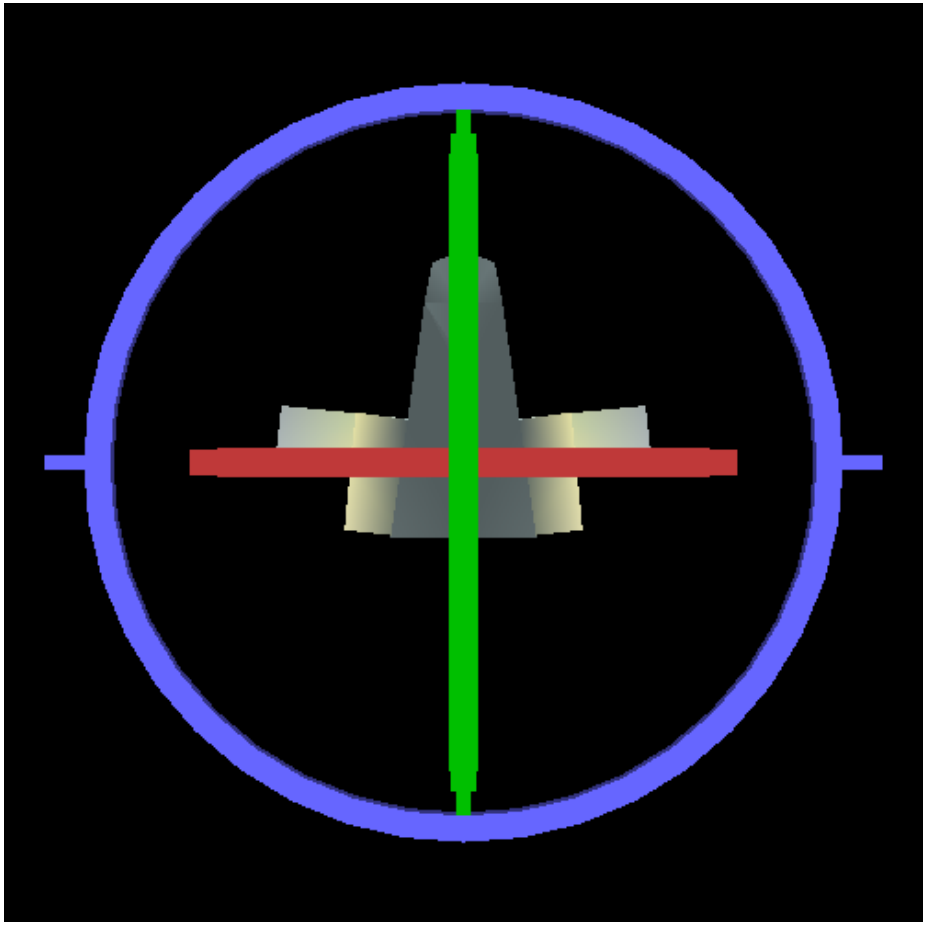
You can use any order that you like, so long as you understand what these angles mean. If you apply the roll first, your yaw and pitch must be in terms of the new roll coordinate system, and not the original coordinate system. That is, a change that is intended to only affect the roll of the final space may need yaw or pitch changes to allow it to have the same apparent orientation (except for the new roll, of course).

But there is a far more insidious problem lurking in here. And this problem happens anytime you compute a final rotation from a series of 3 rotations about axes perpendicular to each other.

The tutorial project Gimbal Lock illustrates this problem. Because the problem was first diagnosed with a physical device called a gimbal [http://en.wikipedia.org/wiki/Gimbal], the problem has become known as *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. The Gimbal Lock project has a set of 3 square gimbals, each with a pivot axis that is perpendicular to the other two. This effectively mimics the common yaw/pitch/roll angle setup.

Figure 8.1. Gimbal Lock Project

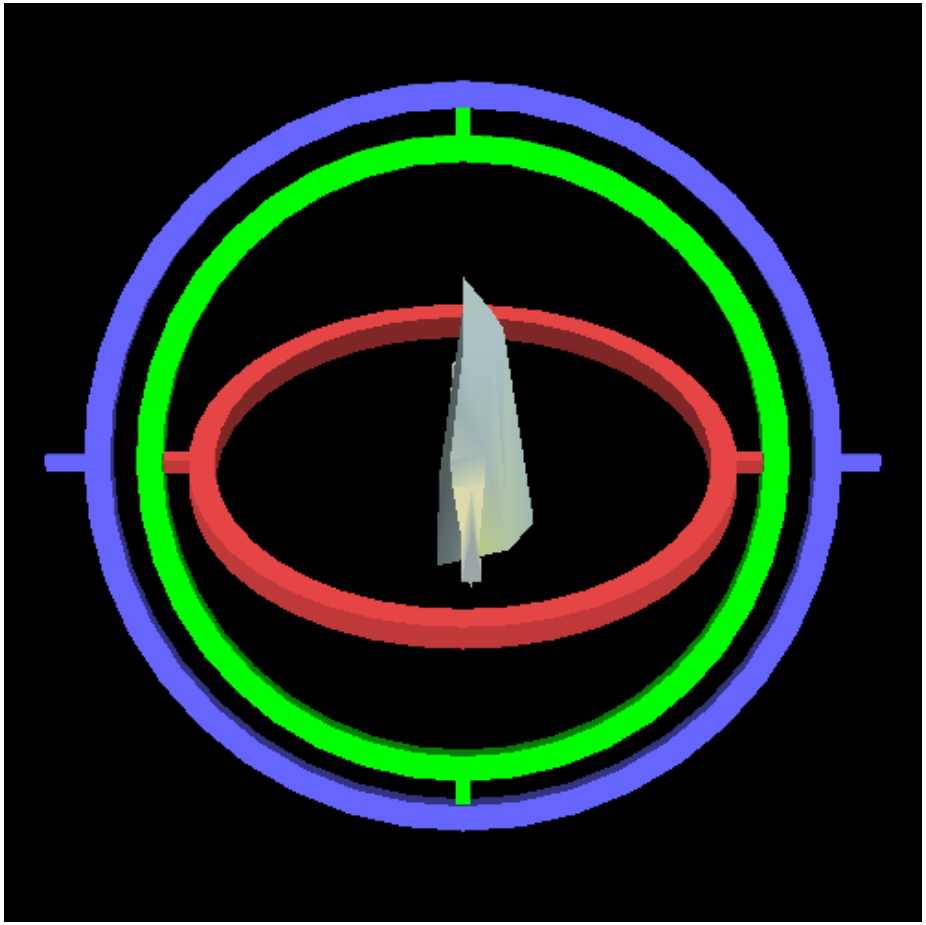


You can control the orientation of each gimbal separately. The **W** and **S** keys control the outer gimbal, the **A** and **D** keys control the middle gimbal, and the **Q** and **E** keys control the inner gimbal. If you just want to see (and affect) the orientation of the ship, press the **SpaceBar** to toggle drawing the gimbal rings.

The first thing you discover when attempting to use the gimbals to orient the ship is that the yaw, pitch, and roll controls of the gimbal change each time you move one of them. That is, when the gimbal arrangement is in the original position, the outer gimbal controls the pitch. But if you move the middle gimbal, it no longer controls *only* the pitch. Orienting the ship is very unintuitive.

The bigger is what happens when two of the gimbals are parallel with one another:

Figure 8.2. Parallel Gimbals



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 yaw, pitch, and roll. However, look at this picture.

Given the controls of these gimbals, can you cause the object to pitch up and down? That is, move its nose up and down from where it is currently? Only slightly; you can use the middle gimbal, which has a bit of pitch rotation. But that is not much.

The reason we do not have as much freedom to orient the object is because the outer and inner gimbals are now rotating about the same axis. This means you really only have two gimbals to manipulate in order to orient the red gimbal. And 3D orientation cannot be fully controlled with only 2 axial rotations, with only 2 gimbals.

When gimbals are in such a position, you have what is known as *gimbal lock*; you have locked one of the gimbals to another, and now both cause the same effect.

Rendering

Before we find a solution to the problem, let's review the code. Most of it is nothing you have not seen elsewhere, so this will be quick.

There is no explicit camera matrix set up for this example; it is too simple for that. The three gimbals are loaded from mesh files as we saw in our last tutorial. They are built to fit into the above array. The ship is also from a mesh file.

The rendering function looks like this:

Example 8.1. Gimbal Lock Display Code

```
void display()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glutil::MatrixStack currMatrix;
    currMatrix.Translate(glm::vec3(0.0f, 0.0f, -200.0f));
    currMatrix.RotateX(g_angles.fAngleX);
    DrawGimbal(currMatrix, GIMBAL_X_AXIS, glm::vec4(0.4f, 0.4f, 1.0f, 1.0f));
    currMatrix.RotateY(g_angles.fAngleY);
    DrawGimbal(currMatrix, GIMBAL_Y_AXIS, glm::vec4(0.0f, 1.0f, 0.0f, 1.0f));
    currMatrix.RotateZ(g_angles.fAngleZ);
    DrawGimbal(currMatrix, GIMBAL_Z_AXIS, glm::vec4(1.0f, 0.3f, 0.3f, 1.0f));

    glUseProgram(theProgram);
    currMatrix.Scale(3.0, 3.0, 3.0);
    currMatrix.RotateX(-90);
    //Set the base color for this object.
    glUniform4f(baseColorUnif, 1.0, 1.0, 1.0, 1.0);
    glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(currMatrix.Top()));

    g_pObject->Render("tint");

    glUseProgram(0);

    glutSwapBuffers();
}
```

The translation done first acts as our camera matrix: positioning the objects far enough away to be comfortably visible. From there, the three gimbals are drawn, with their appropriate rotations. Since each rotation applies to the previous one, the final orientation is given to the last gimbal.

The `DrawGimbal` function does some rotations of its own, but this is just to position the gimbals properly in the array. The gimbals are given a color programmatically, which is the 3rd parameter to `DrawGimbal`.

After building up the rotation matrix, we draw the ship. We use a scale to make it reasonably large, and then rotate it so that it points in the correct direction relative to the final gimbal. In model space, the ship faces the +Z axis, but the gimbal faces the +Y axis. So we needed a change of coordinate system.

Quaternions

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?

Part of the problem is that we are trying to store an orientation as a series of 3 accumulated axial rotations. Orientations are *orientations*, not rotations. And orientations are certainly not a series of rotations. So we need to treat the orientation of the ship as an orientation, as a specific quantity.

The first thought towards this end would be to keep the orientation as a matrix. When the time comes to modify the orientation, we simply apply a transformation to this matrix, storing the result as the new current orientation.

This means that every yaw, pitch, and roll applied to the current orientation will be relative to that current orientation. Which is precisely what we need. If the user applies a positive yaw, you want that yaw to rotate them relative to where they are current pointing, not relative to some fixed coordinate system.

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.*

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:

Equation 8.1. Angle/Axis to Quaternion

$$\text{Axis} = (x, y, z)$$
$$\text{Angle} = \#$$
$$\text{Quaternion} = \begin{bmatrix} x * \sin\left(\frac{\#}{2}\right) \\ y * \sin\left(\frac{\#}{2}\right) \\ z * \sin\left(\frac{\#}{2}\right) \\ \cos\left(\frac{\#}{2}\right) \end{bmatrix}$$

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:

Equation 8.2. Quaternion Multiplication

$$a * b = \begin{bmatrix} a_w * b_x + a_x * b_w + a_y * b_z - a_z * b_y \\ a_w * b_y + a_y * b_w + a_z * b_x - a_x * b_z \\ a_w * b_z + a_z * b_w + a_x * b_y - a_y * b_x \\ a_w * b_w - a_x * b_x - a_y * b_y - a_z * b_z \end{bmatrix}$$

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*b) * c = a * (b*c)), but not commutative (a*b != b*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. While we could convert a unit quaternion back into angle/axis rotations, it's much preferable to do it directly:

Equation 8.3. Quaternion to Matrix

$$\begin{bmatrix} 1 - 2q_yq_y - 2q_zq_z & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y & 0 \\ 2q_xq_y + 2q_wq_z & 1 - 2q_xq_x - 2q_zq_z & 2q_yq_z - 2q_wq_x & 0 \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & 1 - 2q_xq_x - 2q_yq_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This does look suspiciously similar to the formula for generating a matrix from an angle/axis rotation.

Composition Type

So our goal is to compose successive rotations into a final orientation. When we want to increase the pitch, for example, we will take the current orientation and multiply into it a quaternion that represents a pitch rotation 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.

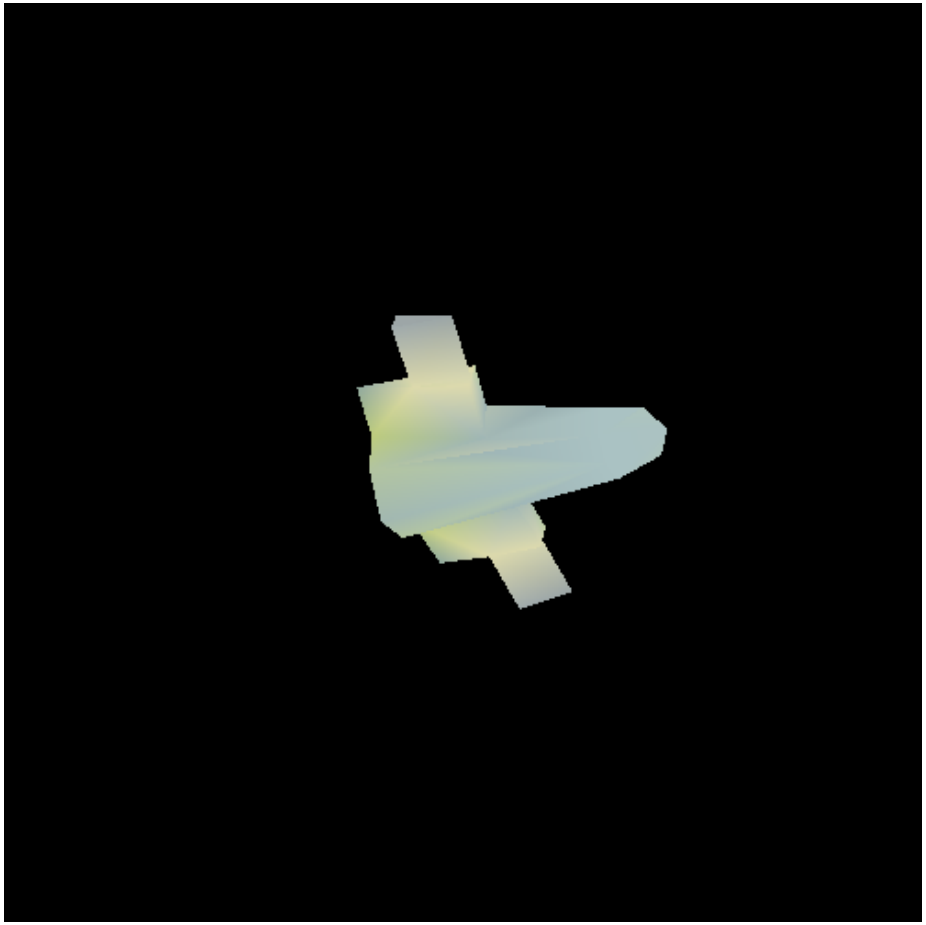
Our positions (p) are in model space. We are transforming them into world space. The current transformation matrix is represented by the orientation O. Thus, to transform points, we use O*p

Now, we want to adjust the orientation O by applying some small pitch change. Well, the pitch of the model is defined by model space. Therefore, the pitch change (R) is a transformation that takes coordinates in model space and transforms them to the pitch space. So our total transformation is O*R*p; the new orientation is O*R.

Yaw Pitch Roll

We implement this in the Quaternion YPR tutorial. This tutorial does not show gimbals, but the same controls exist for yaw, pitch, and roll transformations. Here, pressing the **SpaceBar** will switch between right-multiplying the YPR values to the current orientation and left-multiplying them. Post-multiplication will apply the YPR transforms from world-space.

Figure 8.3. Quaternion YPR Project



The rendering code is pretty straightforward.

Example 8.2. Quaternion YPR Display

```
void display()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glutil::MatrixStack currMatrix;
    currMatrix.Translate(glm::vec3(0.0f, 0.0f, -200.0f));
```

```
    currMatrix.ApplyMatrix(glm::mat4_cast(g_orientation));

    glUseProgram(theProgram);
    currMatrix.Scale(3.0, 3.0, 3.0);
    currMatrix.RotateX(-90);
    //Set the base color for this object.
    glUniform4f(baseColorUnif, 1.0, 1.0, 1.0, 1.0);
    glUniformMatrix4fv(modelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(currMatrix.Top()));

    g_pShip->Render("tint");

    glUseProgram(0);

    glutSwapBuffers();
}
```

Though GLSL does not have quaternion types or quaternion arithmetic, the GLM math library provides both. The `g_orientation` variable is of the type `glm::fquat`, which is a floating-point quaternion. The `glm::mat4_cast` function converts a quaternion into a 4x4 rotation matrix. This stands in place of the series of 3 rotations used in the last tutorial.

In response to keypresses, `g_orientation` is modified, applying a transform to it. This is done with the `OffsetOrientation` function.

Example 8.3. OffsetOrientation Function

```
void OffsetOrientation(const glm::vec3 &_axis, float fAngDeg)
{
    float fAngRad = Framework::DegToRad(fAngDeg);

    glm::vec3 axis = glm::normalize(_axis);

    axis = axis * sinf(fAngRad / 2.0f);
    float scalar = cosf(fAngRad / 2.0f);

    glm::fquat offset(scalar, axis.x, axis.y, axis.z);

    if(g_bRightMultiply)
        g_orientation = g_orientation * offset;
    else
        g_orientation = offset * g_orientation;

    g_orientation = glm::normalize(g_orientation);
}
```

This generates the offset quaternion from an angle and axis. Since the axis is normalized, there is no need to normalize the resulting `offset` quaternion. Then the offset is multiplied into the orientation, and the result is normalized.

In particular, pay attention to the difference between right multiplication and left multiplication. When you right-multiply, the offset orientation is in model space. When you left-multiply, the offset is in *world* space. Both of these can be useful for different purposes.

Camera-Relative Orientation

As useful as model and world space offsetting is, there is one more space that it might be useful to orient from. Camera-space.

This is primarily useful in modelling applications, but it can have other applications as well. In such programs, as the user spins the camera around to different angles, the user may want to transform the object relative to the direction of the view.

In order to understand the solution to doing this, let's frame the problem explicitly. We have positions (p) in model space. These positions will be transformed by a current model-to-world orientation (O), and then by a final camera matrix (C). Thus, our transform equation is C*O*p.

We want to apply an orientation offset (R), which takes points in camera-space. If we wanted to apply this to the camera matrix, it would simply be multiplied by the camera matrix: R*C*O*p. That's nice and all, but we want to apply a transform to O, not to C.

Therefore, we need to use our transforms to generate a new orientation offset (N), which will produce the same effect:

C*N*O = R*C*O

Inversion

In order to solve this, we need to introduce a new concept. Given a matrix M, there may be a matrix N such that MN = I, where I is the identity matrix. If there is such a matrix, the matrix N is called the *inverse matrix* of M. The notation for the inverse matrix of M is M^{-1}. The symbol "$^{-1}$" does not mean to raise the matrix to the -1 power; it means to invert it.

The matrix inverse can be analogized to the scalar multiplicative inverse (ie: reciprocal). The scalar multiplicative inverse of X is the number Y such that XY = 1.

In the case of the scalar inverse, this is very easy to solve for: Y = 1/X. Easy though this may be, there are values of X for which there is no multiplicative inverse. OK, there's *one* such real value of X: 0.

The case of the inverse matrix is much more complicated. Just as with the scalar inverse, there are matrices that have no inverse. Unlike the scalar case, there are a *lot* of matrices with no inverse. Also, computing the inverse matrix is a *lot* more complicated than simply taking the reciprocal of a value.

Most common transformation matrices do have an inverse. And for the basic transformation matrices, the inverse matrix is very easy to compute. For a pure rotation matrix, simply compute a new rotation matrix by negating the angle that the old one was generated with. For a translation matrix, negate the origin value in the matrix. For a scale matrix, take the reciprocal of the scale along each axis.

To take the inverse of a sequence of matrices, you can take the inverse of each of the component matrices. But you have to do the matrix multiplication in *reverse* order. So if we have M = TRS, then $M^{-1} = S^{-1}R^{-1}T^{-1}$.

Quaternions, like matrices, have a multiplicative inverse. The inverse of a pure rotation matrix, which quaternions represent, is a rotation about the same axis with the negative of the angle. For any angle θ, it is the case that $\sin(-\theta) = -\sin(\#)$. It is also the case that $\cos(-\theta) = \cos(\#)$. Since the vector part of a quaternion is built by multiplying the axis by the sine of the half-angle, and the scalar part is the cosine of the half-angle, the inverse of any quaternion is just the negation of the vector part.

You can also infer this to mean that, if you negate the axis of rotation, you are effectively rotating about the old axis but negating the angle. Which is true, since the direction of the axis of rotation defines what direction the rotation angle moves the points in. Negate the axis's direction, and you're rotating in the opposite direction.

In quaternion lingo, the inverse quaternion is more correctly called the *conjugate quaternion*. We use the same inverse notation, "$^{-1}$," to denote conjugate quaternions.

Incidentally, the identity quaternion is a quaternion who's rotation angle is zero. The cosine of 0 is one, and the sine of 0 is zero, so the vector part of the identity quaternion is zero and the scalar part is one.

Solution

Given our new knowledge of inverse matrices, we can solve our problem.

C*N*O = R*C*O

We can right-multiply both sides of this equation by the inverse transform of O.

$(C*N*O)*O^{-1} = (R*C*O)*O^{-1}$

C*N*I = R*C*I

The I is the identity transform. From here, we can left-multiply both sides by the inverse transform of C:

$C^{-1} *(C*N) = C^{-1}*(R*C)$

$N = C^{-1}*(R*C)$

Therefore, given an offset that is in camera space, we can generate the world-space equivalent by multiplying it between the camera and inverse camera transforms.

Transformation Spaces

It turns out that this is a generalized operation. It can be used for much more than just orientation changes.

Consider a scale operation. Scales apply along the main axes of their space. But if you want to scale something along a different axis, how do you do that?

You rotate the object into a coordinate system where the axis you want to scale *is* one of the basis axes, perform your scale, then rotate it back with the inverse of the previous rotation.

Effectively, what we are doing is transforming, not positions, but other transformation matrices into different spaces. A transformation matrix has some input space and defines an output space. If we want to apply that transformation in a different space, we perform this operation.

The general form of this sequence is as follows. Suppose you have a transformation matrix T, which operates on points in a space called F. We have some positions in the space P. What we want is to create a matrix that applies T's transformation operation, except that it needs to operate on points in the space of P. Given a matrix M that transforms from P space to F space, that matrix is $M^{-1}*T*M$.

Final Orientation

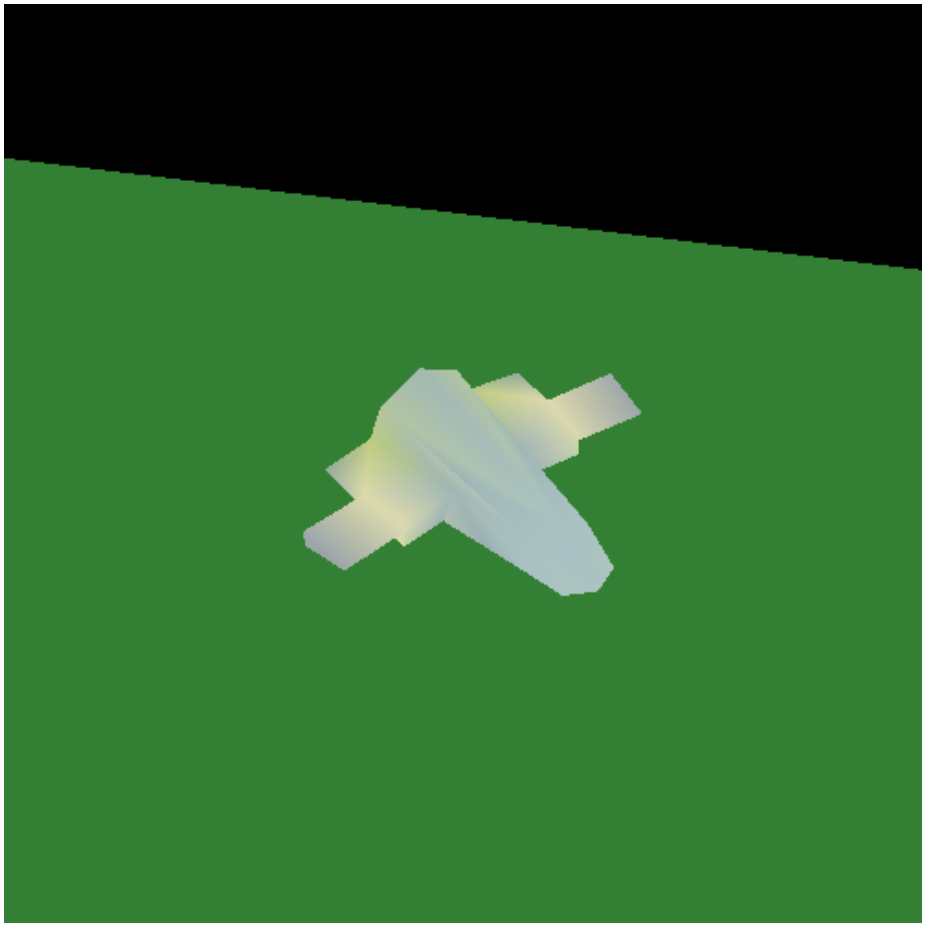
Let's look at how this all works out in code, with the Camera Relative tutorial. This works very similarly to the last tutorial, but with a few differences.

Since we are doing camera-relative rotation, we need to have an actual camera that can move independently of the world. So we incorporate our camera code from our world space into this one. As before, the **I** and **K** keys will move the camera up and down, relative to a center point. The **J** and **K** keys will move the camera left and right around the center point. Holding **Shift** with these keys will move the camera in smaller increments.

The **SpaceBar** will toggle between three transforms: model-relative (yaw/pitch/roll-style), world-relative, and camera-relative.

Our scene also includes a ground plane as a reference.

Figure 8.4. Camera Relative Project



The `display` function only changed where needed to deal with drawing the ground plane and to handle the camera. Either way, it's nothing that has not been seen elsewhere.

The substantive changes were in the `OffsetOrientation` function:

Example 8.4. Camera Relative OffsetOrientation

```
void OffsetOrientation(const glm::vec3 &_axis, float fAngDeg)
{
    float fAngRad = Framework::DegToRad(fAngDeg);

    glm::vec3 axis = glm::normalize(_axis);
```

```
    axis = axis * sinf(fAngRad / 2.0f);
    float scalar = cosf(fAngRad / 2.0f);

    glm::fquat offset(scalar, axis.x, axis.y, axis.z);

    switch(g_iOffset)
    {
    case MODEL_RELATIVE:
        g_orientation = g_orientation * offset;
        break;
    case WORLD_RELATIVE:
        g_orientation = offset * g_orientation;
        break;
    case CAMERA_RELATIVE:
        {
            const glm::vec3 &camPos = ResolveCamPosition();
            const glm::mat4 &camMat = CalcLookAtMatrix(camPos, g_camTarget, glm::vec3(0.0f, 1.0f, 0

            glm::fquat viewQuat = glm::quat_cast(camMat);
            glm::fquat invViewQuat = glm::conjugate(viewQuat);

            const glm::fquat &worldQuat = (invViewQuat * offset * viewQuat);
            g_orientation = worldQuat * g_orientation;
        }
        break;
    }

    g_orientation = glm::normalize(g_orientation);
}
```

The change here is the addition of the camera-relative condition. To do this in quaternion math, we must first convert the world-to-camera matrix into a quaternion representing that orientation. This is done here using `glm::quat_cast`.

The conjugate quaternion is computed with GLM. Then, we simply multiply them with the offset to compute the world-space offset orientation. This gets left-multiplied into the current orientation, and we're done.

Interpolation

A quaternion represents an orientation; it defines a coordinate system relative to another. If we have two orientations, we can consider the orientation of the same object represented in both coordinate systems.

What if we want to generate an orientation that is halfway between them, for some definition of "halfway"? Or even better, consider an arbitrary interpolation between two orientations, so that we can watch an object move from one orientation to another. This would allow us to see an object smoothly moving from one orientation to another.

This is one more trick we can play with quaternions that we cannot 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.

The Interpolation tutorial demonstrates this. The **Q**, **W**, **E**, **R**, **T**, **Y**, and **U** keys cause the ship to interpolate to a new orientation. Each key corresponds to a particular orientation, and the **Q** key is the initial orientation.

We can see that there are some pretty reasonable looking transitions. The transition from **Q** to **W**, for example. However, there are some other transitions that do not look so good; the **Q** to **E** transition. What exactly is going on?

The Long Path

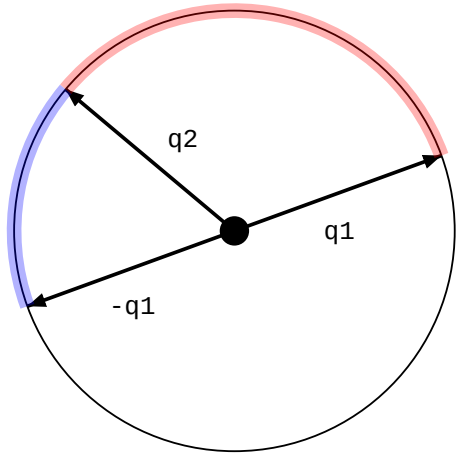
Unit quaternions represent orientations, but they are also vector directions. Specifically, directions in a four-dimensional space. Being unit vectors, they represent points on a 4D sphere of radius one. Therefore, the path between two orientations can be considered to be simply moving from one direction to another on the surface of the 4D sphere.

While unit quaternions do represent orientations, a quaternion is not a *unique* representation of an orientation. That is, there are multiple quaternions that represent the same orientation. Well, there are two.

The conjugate of a quaternion, its inverse orientation, is the negation of the vector part of the quaternion. If you negate all four components however, you get something quite different: the same orientation as before. Negating a quaternion does not affect its orientation.

While the two quaternions represent the same orientation, they are not the same as far as interpolation is concerned. Consider a two-dimensional case:

Figure 8.5. Interpolation Directions



If the angle between the two quaternions is greater than 90°, then the interpolation between them will take the "long path" between the two orientations. Which is what we see in the **Q** to **E** transition. The orientation **R** is the negation of **E**; if you try to interpolate between them, nothing changes. The **Q** to **R** transition looks much better behaved.

This can be detected easily enough. If the 4-vector dot product between the two quaternions is less than zero, then the long path will be taken. If you want to prevent the long path from being used, simply negate one of the quaternions before interpolating if you detect this. Similarly, if you want to force the long path, then ensure that the angle is greater than 90° by negating a quaternion if the dot product is greater than zero.

Interpolation Speed

There is another problem. Notice how fast the **Q** to **E** interpolation is. It starts off slow, then rapidly spins around, then slows down towards the end. Why does this happen?

The linear interpolation code looks like this:

Example 8.5. Quaternion Linear Interpolation

```
glm::fquat Lerp(const glm::fquat &v0, const glm::fquat &v1, float alpha)
{
    glm::vec4 start = Vectorize(v0);
    glm::vec4 end = Vectorize(v1);
    glm::vec4 interp = glm::mix(start, end, alpha);
    interp = glm::normalize(interp);
    return glm::fquat(interp.w, interp.x, interp.y, interp.z);
}
```

Note

GLM's quaternion support does something unusual. The W component is given first to the fquat constructor. Be aware of that when looking through the code.

The `Vectorize` function simply takes a quaternion and returns a vec4; this is necessary because GLM fquat do not support many of the operations that GLM vec4's do. In this case, the glm::mix function, which performs component-wise linear interpolation.

Each component of the vector is interpolated separately from the rest. The quaternion for **Q** is (0.7071f, 0.7071f, 0.0f, 0.0f), while the quaternion for **E** is (-0.4895f, -0.7892f, -0.3700f, -0.02514f). In order for the first componet of Q to get to E's first component, it will have to go through zero.

When the alpha is around 0.5, half-way through the movement, the resultant vector before normalization is very small. But the vector itself is not what provides the orientation; the *direction* of the 4D vector is. Which is why it moves very fast in the middle: the direction is changing rapidly.

In order to get smooth interpolation, we need to interpolate based on the direction of the vectors. That is, we interpolate along the angle between the two vectors. This kind of interpolation is called *spherical linear interpolation* or *slerp*.

To see the difference this makes, press the **SpaceBar**; this toggles between regular linear interpolation and slerp. The slerp version is much smoother.

The code for slerp is rather complex:

Example 8.6. Spherical Linear Interpolation

```
glm::fquat Slerp(const glm::fquat &v0, const glm::fquat &v1, float alpha)
{
    float dot = glm::dot(v0, v1);

    const float DOT_THRESHOLD = 0.9995f;
    if (dot > DOT_THRESHOLD)
        return Lerp(v0, v1, alpha);

    glm::clamp(dot, -1.0f, 1.0f);
    float theta_0 = acosf(dot);
    float theta = theta_0*alpha;

    glm::fquat v2 = v1 - v0*dot;
    v2 = glm::normalize(v2);

    return v0*cos(theta) + v2*sin(theta);
}
```


Slerp and Performance

It's important to know what kind of problems slerp is intended to solve and what kind it is not. Slerp becomes increasingly more important the more disparate the two quaternions being interpolated are. If you know that two quaternions are always quite close to one another, then slerp is not worth the expense.

The `acos` call in the slerp code alone is pretty substantial in terms of performance. Whereas lerp is typically just a vector/scalar multiply followed by a vector/vector addition. Even on the CPU, the performance difference is important, particularly if you're doing thousands of these per frame. As you might be in an animation system.

In Review

In this tutorial, you have learned the following:

- A fixed sequence of successive rotations can prevent other rotations from contributing to the object's orientation. It also makes it difficult to correctly orient the object in an intuitive way, since previous rotations have effects on later ones.
- Quaternions are 4-dimensional vectors that can encode an orientation. They can be used for successively applying small rotations to an orientation. Matrices fail at this because of the difficulty of orthonormalizing them to avoid floating-point error accumulation.
- Quaternions work almost identically to matrices, in so far as they specify orientations. They can be constructed directly from an angle/axis rotation, and they can be composed with one another via quaternion multiplication.
- One can transform a matrix, or a quaternion with another matrix or quaternion, such that the resulting transform is specified in a different space. This is useful for applying rotations to an object's orientation that are in camera-space,, while the object's orientation remains a model-to-world transform.
- Quaternions can be interpolated, either with component-wise linear interpolation or with spherical linear interpolation. If the angle between two quaternion vectors is greater than 90°, then the interpolation between them will move indirectly between the two.

Further Study

Try doing the following with the orientation tutorials.

- Modify the Interpolation tutorial to allow multiple animations to be active simultaneously. The `Orientation` class is already close to being able to allow this. Instead of storing a single `Orientation::Animation` object, it should store a `std::deque` of them. When the external code adds a new one, it gets pushed onto the end of the deque. During update, the front-most entries can end and be popped off, recording its destination index as the new current one in the `Orientation` class. To get the orientation, just call each animation's orientation function, feeding the previous result into the next one.
- Change the Interpolation tutorial to allow one to specify whether the long path or short path between two orientations should be taken. This can work for both linear and spherical interpolation.

Further Research

This discussion has focused on the utility of quaternions in orienting objects, and it has deftly avoided answering the question of exactly what a quaternion *is.* After all, saying that a quaternion is a four-dimensional complex number does not explain why they are useful in graphics. They are a quite fascinating subject for those who like oddball math concepts.

This discussion has also glossed over a few uses of quaternions in graphics, such as how to directly rotate a position or direction by a quaternion. Such information is readily available online.

Glossary

gimbal lock — When applying 3 or more successive rotations about axes that are orthogonal to each other, gimbal lock occurs when a degree of rotational freedom is lost due to two or more axes that cause the same effective rotation.

orthonormal	A transform has the property of being orthonormal if the three basis axes are all orthogonal and the three axes are normal (have a length of 1).
quaternion	A four-dimensional vector that represents an orientation. The first three components of the quaternion, the X, Y and Z, are considered the vector part of the quaternion. The fourth component is the scalar part.
inverse matrix	The inverse matrix of the matrix M is the matrix N for which the following equation is true: MN = I, where I is the identity matrix. The inverse of M is usually denoted as M^{-1}.
conjugate quaternion	Analogous to the inverse matrix. It is computed by negating the vector part of the quatenrion.
spherical linear interpolation, slerp	Interpolation between two unit vectors that is linear based on the angle between them, rather than the vectors themselves.

Part III. Illumination

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.

Properly modeling the interaction between light and a surface is vital in creating a convincing world. Lighting defines how we see and understand shapes to a large degree. The lack of lighting is the reason why the objects we have used thus far look fairly flat. A curved surface appears curved to us because of how the light plays over the surface. The same goes for a flat surface.

Without this visual hinting, surfaces appear flat even when they are modeled with many triangles and yield a seemingly-curved polygonal mesh. A proper lighting model makes objects appear real. A poor or inconsistent lighting model shows the virtual world to be the forgery that it is.

This section of the book will cover lighting, using a variety of light/surface modelling techniques. It will cover dynamic range and linear colorspaces in lighting equations. Also, it will cover techniques to use lighting to produce entirely fake surfaces.

Chapter 9. Lights On

It is always best to start simply. And since lighting is a big topic, we will begin with the simplest possible scenario.

Modelling Lights

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 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. For the most part, in this book, we will focus on approximating photorealism.

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

In the real world, our eyes see by detecting light that hits them. The structure of our iris and lenses use a number of photorecepters (light-sensitive cells) to resolve a pair of images. 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.*

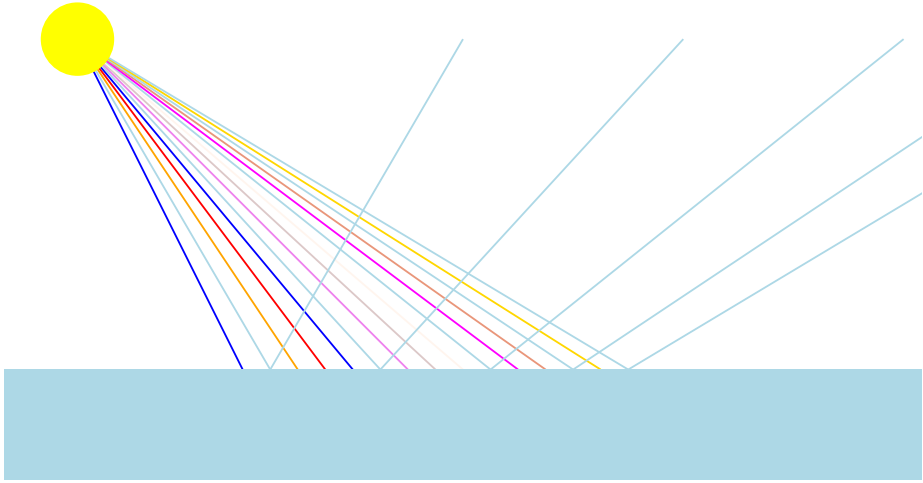
The interaction between a light and a surface is the most important part of a lighting model. It is also the most difficult to get right. The way light interacts with atoms on a surface alone involves complicated quantum mechanical principles that are difficult to understand. And even that does not get into the fact that surfaces are not perfectly smooth or perfectly opaque.

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.

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.

Figure 9.1. Surface Light Absorption



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.

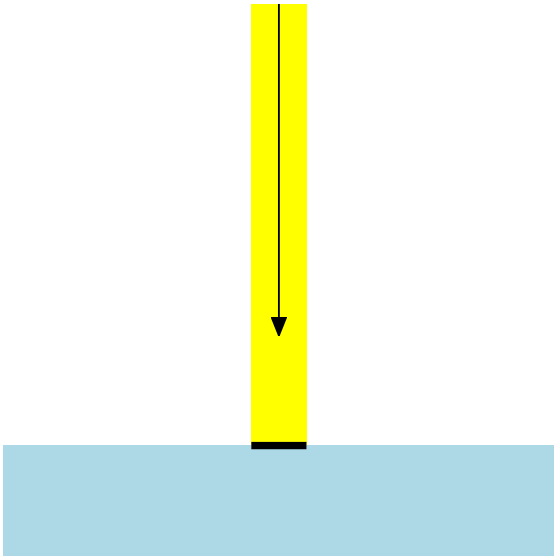
The very first approximation that is made is that not all of these wavelengths matter. Instead of tracking millions of wavelengths in the visible spectrum, we will instead track 3. Red, green, and blue.

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.

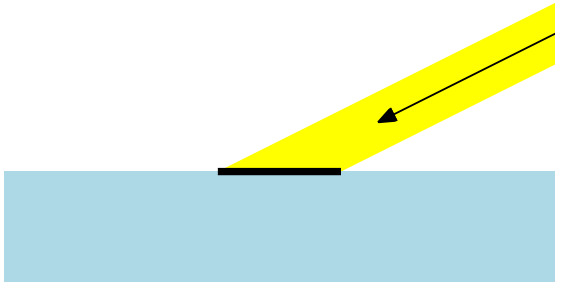
Consider a perfectly flat surface. If you shine a column of light with a known intensity directly onto that surface, the intensity of that light at each point under the surface will be a known value, based on the intensity of the light divided by the area projected on the surface.

Figure 9.2. Perpendicular Light



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.

Figure 9.3. Light at an Angle



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.

A lighting model is a function of all of these parameters. This is far from a comprehensive list of lighting parameters; this list will be expanded considerably in future discussions.

Standard Diffuse Lighting

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.

Figure 9.4. Diffuse Reflectance

An ideal diffuse material will reflect light evenly in all directions, as shown in the picture above. No actual surfaces are ideal diffuse materials, but this is a good starting point and looks pretty decent.

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 equation for this lighting model is quite simple:

Equation 9.1. Diffuse Lighting Equation

R # Reflected Color I # Light Intensity

D # Diffuse Surface Absorption # # Angle of Incidence

Diffuse Lighting Equation $R = D * I * \cos(\#)$

The cosine of the angle of incidence is used because it represents the perfect hemisphere of light that would be reflected. When the angle of incidence is 0°, the cosine of this angle will be 1.0. The lighting will be at its brightest. When the angle of incidence is 90°, the cosine of this angle will be 0.0, so the lighting will be 0. Values less than 0 are clamped to 0.

Surface Orientation

Now that we know what we need to compute, the question becomes how to compute it. Specifically, this means how to compute the angle of incidence for the light, but it also means where to perform the lighting computations.

Since our mesh geometry is made of triangles, each individual triangle is flat. Therefore, much like the plane above, each triangle faces a single direction. This direction is called the *surface normal* or *normal*. It is the direction that the surface is facing at the location of interest.

Every point along the surface of a single triangle has the same geometric surface normal. That's all well and good, for actual triangles. But polygonal models are usually supposed to be approximations of real, curved surfaces. If we use the actual triangle's surface normal for all of the points on a triangle, the object would look very faceted. This would certainly be an accurate representation of the actual triangular mesh, but it

reveals the surface to be exactly what it is: a triangular mesh approximation of a curved surface. If we want to create the illusion that the surface really is curved, we need to do something else.

Instead of using the triangle's normal, we can assign to each vertex the normal that it *would* have had on the surface it is approximating. That is, while the mesh is an approximation, the normal for a vertex is the actual normal for that surface. This actually works out surprisingly well.

This means that we must add to the vertex's information. In past tutorials, we have had a position and sometimes a color. 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.

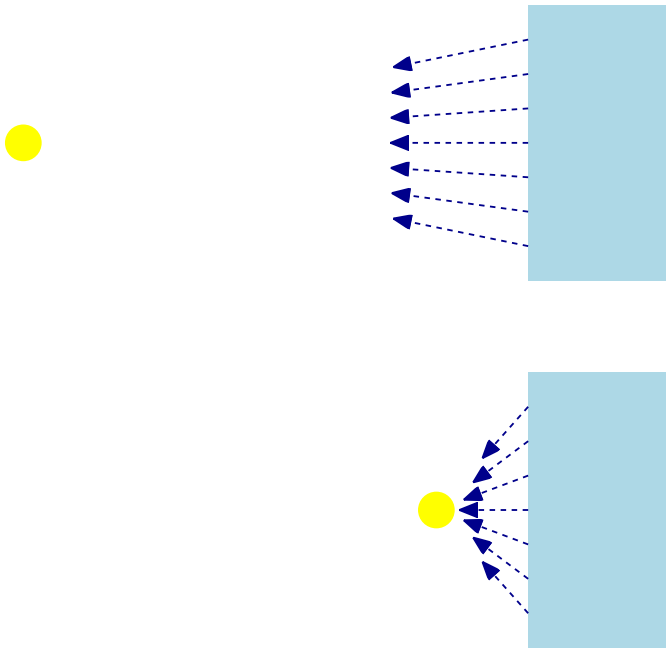
That being said, modern games have essentially abandoned this technique. Part of that is because the per-fragment computation is not as slow and limited as it used to be. And part of it is simply that games tend to not use just diffuse lighting anymore, so the Gouraud approximation is more noticeably inaccurate.

Directional Light Source

The angle of incidence is the angle between the surface normal and the direction towards the light. Computing the direction from the point in question to the light can be done in a couple of ways.

If you have a light source that is very close to an object, then the direction towards the light can change dramatically over the surface of that object. As the light source is moved farther and farther away, the direction towards the light varies less and less over the surface of the object.

Figure 9.5. Near and Far Lights



If the light source is sufficiently distant, relative to the size of the scene being rendered, then the direction towards the light is nearly the same for every point on every object you render. Since the direction is the same everywhere, the light can be represented as just a single direction given to all of the objects. There is no need to compute the direction based on the position of the point being illuminated.

This situation is called a *directional light source*. Light from such a source effectively comes from a particular direction as a wall of intensity, evenly distributed over the scene.

Direction light sources are a good model for lights like the sun relative to a small region of the Earth. It would not be a good model for the sun relative to the rest of the solar system. So scale is important.

Light sources do not have to be physical objects rendered in the scene. All we need to use a directional light is to provide a direction to our lighting model when rendering the surface we want to see. However, having light appear from seemingly nothing hurts verisimilitude; this should be avoided where possible.

Alternatives to directional lights will be discussed a bit later.

Normals and Space

Normals have many properties that positions do. Normals are vector directions, so like position vectors, they exist in a certain coordinate system. It is usually a good idea 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 another space. That other space needs to be the same space that the lighting direction is in; otherwise, the two vectors cannot be compared. One might think that world space is a fine choice. After all, the light direction is already defined in world space.

You certainly could use world space to do lighting. However, for our purposes, we will use camera space. The reason for this is partially illustrative: in later tutorials, we are going to do lighting in some rather unusual spaces. By using camera space, it gets us in the habit of transforming both our light direction and the surface normals into different spaces.

We will talk more in later sections about exactly how we transform the normal. For now, we will just transform it with the regular transformation matrix.

Drawing with Lighting

The full lighting model for computing the diffuse reflectance from directional light sources, using per-vertex normals and Gouraud shading, is as follows. The light will be represented by a direction and a light intensity (color). The light direction passed to our shader is expected to be in camera space already, so the shader is not responsible for this transformation. For each vertex (in addition to the normal position transform), we:

1. Transform the normal from model space to camera space using the model-to-camera transformation matrix.

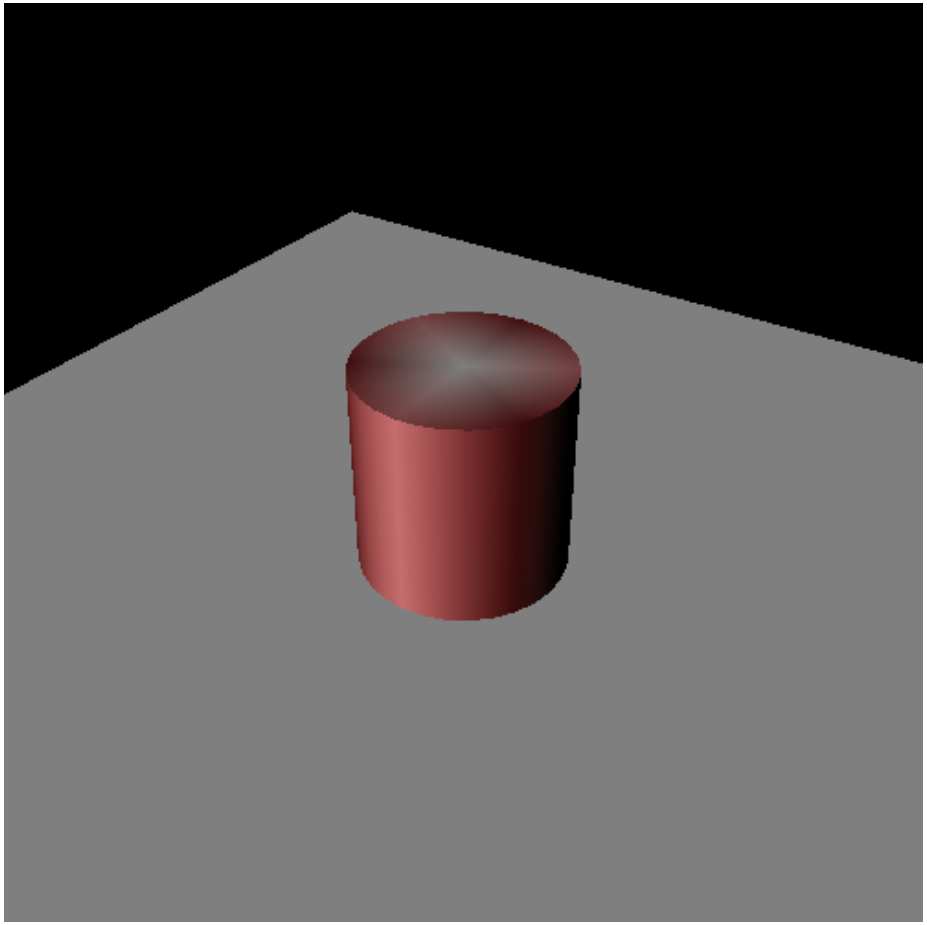
2. Compute the cosine of the angle of incidence.

3. Multiply the light intensity by the cosine of the angle of incidence, and multiply that by the diffuse surface color.

4. Pass this value as a vertex shader output, which will be written to the screen by the fragment shader.

This is what we do in the Basic Lighting tutorial. It renders a cylinder above a flat plane, with a single directional light source illuminating both objects. One of the nice things about a cylinder is that it has both curved and flat surfaces, thus making an adequate demonstration of how light interacts with a surface.

Figure 9.6. Basic Lighting



The light is at a fixed direction; the model and camera both can be rotated.

Mouse Movement

This is the first tutorial that uses mouse movement to orient objects and the camera. These controls will be used throughout the rest of this book.

The camera can be oriented with the left mouse button. Left-clicking and dragging will rotate the camera around the target point. This will rotate both horizontally and vertically. Think of the world as a sphere. Starting to drag means placing your finger on the sphere. Moving your mouse is like moving your finger; the sphere rotates along with your finger's movement. If you hold **Ctrl** when you left-click, you can rotate either horizontally or vertically, depending on the direction you move the mouse. Whichever direction is farthest from the original location clicked will be the axis that is rotated.

The camera's up direction can be changed as well. To do this, left-click while holding **Alt**. Only horizontal movements of the mouse will spin the view. Moving left spins counter-clockwise, while moving right spins clockwise.

The camera can be moved closer to it's target point and farther away. To do this, scroll the mouse wheel up and down. Up scrolls move closer, while down moves farther away.

The object can be controlled by the mouse as well. The object can be oriented with the right-mouse button. Right-clicking and dragging will rotate the object horizontally and vertically, relative to the current camera view. As with camera controls, holding **Ctrl** when you right-click will allow you to rotate horizontally or vertically only.

The object can be spun by right-clicking while holding **Alt**. As with the other object movements, the spin is relative to the current direction of the camera.

The code for these are contained in the Unofficial SDK's GL Util library. Specifically, the objects glutil::ViewPole and glutil::ObjectPole. The source code in them is, outside of how FreeGLUT handles mouse input, nothing that has not been seen previously.

Pressing the **Spacebar** will switch between a cylinder that has a varying diffuse color and one that is pure white. This demonstrates the effect of lighting on a changing diffuse color.

The initialization code does the usual: loads the shaders, gets uniforms from them, and loads a number of meshes. In this case, it loads a mesh for the ground plane and a mesh for the cylinder. Both of these meshes have normals at each vertex; we'll look at the mesh data a bit later.

The display code has gone through a few changes. The vertex shader uses only two matrices: one for model-to-camera, and one for camera-to-clip-space. So our matrix stack will have the camera matrix at the very bottom.

Example 9.1. Display Camera Code

```
glutil::MatrixStack modelMatrix;
modelMatrix.SetMatrix(g_viewPole.CalcMatrix());

glm::vec4 lightDirCameraSpace = modelMatrix.Top() * g_lightDirection;

glUseProgram(g_WhiteDiffuseColor.theProgram);
glUniform3fv(g_WhiteDiffuseColor.dirToLightUnif, 1, glm::value_ptr(lightDirCameraSpace));
glUseProgram(g_VertexDiffuseColor.theProgram);
glUniform3fv(g_VertexDiffuseColor.dirToLightUnif, 1, glm::value_ptr(lightDirCameraSpace));
glUseProgram(0);
```

Since our vertex shader will be doing all of its lighting computations in camera space, we need to move the `g_lightDirection` from world space to camera space. So we multiply it by the camera matrix. Notice that the camera matrix now comes from the MousePole object.

Now, we need to talk a bit about vector transforms with matrices. When transforming positions, the fourth component was 1.0; this was used so that the translation component of the matrix transformation would be added to each position.

Normals represent directions, not absolute positions. And while rotating or scaling a direction is a reasonable operation, translating it is not. Now, we could just adjust the matrix to remove all translations before transforming our light into camera space. But that's highly unnecessary; we can simply put 0.0 in the fourth component of the direction. This will do the same job, only we do not have to mess with the matrix to do so.

This also allows us to use the same transformation matrix for vectors as for positions.

We upload the camera-space light direction to the two programs.

To render the ground plane, we run this code:

Example 9.2. Ground Plane Lighting

```
glutil::PushStack push(modelMatrix);

glUseProgram(g_WhiteDiffuseColor.theProgram);
glUniformMatrix4fv(g_WhiteDiffuseColor.modelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(modelMa
glm::mat3 normMatrix(modelMatrix.Top());
glUniformMatrix3fv(g_WhiteDiffuseColor.normalModelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(n
glUniform4f(g_WhiteDiffuseColor.lightIntensityUnif, 1.0f, 1.0f, 1.0f, 1.0f);
g_pPlaneMesh->Render();
glUseProgram(0);
```

We upload two matrices. One of these is used for normals, and the other is used for positions. The normal matrix is only 3x3 instead of the usual 4x4. This is because normals do not use the translation component. We could have used the trick we used earlier, where we use a 0.0 as the W component of a 4 component normal. But instead, we just extract the top-left 3x3 area of the model-to-camera matrix and send that.

Of course, the matrix is the same as the model-to-camera, except for the lack of translation. The reason for having separate matrices will come into play later.

We also upload the intensity of the light, as a pure-white light at full brightness. Then we render the mesh.

To render the cylinder, we run this code:

Example 9.3. Cylinder Lighting

```
glutil::PushStack push(modelMatrix);

modelMatrix.ApplyMatrix(g_objtPole.CalcMatrix());

if(g_bDrawColoredCyl)
{
    glUseProgram(g_VertexDiffuseColor.theProgram);
    glUniformMatrix4fv(g_VertexDiffuseColor.modelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(mo
    glm::mat3 normMatrix(modelMatrix.Top());
    glUniformMatrix3fv(g_VertexDiffuseColor.normalModelToCameraMatrixUnif, 1, GL_FALSE, glm::value_
    glUniform4f(g_VertexDiffuseColor.lightIntensityUnif, 1.0f, 1.0f, 1.0f, 1.0f);
    g_pCylinderMesh->Render("lit-color");
}
else
{
    glUseProgram(g_WhiteDiffuseColor.theProgram);
    glUniformMatrix4fv(g_WhiteDiffuseColor.modelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(mod
    glm::mat3 normMatrix(modelMatrix.Top());
    glUniformMatrix3fv(g_WhiteDiffuseColor.normalModelToCameraMatrixUnif, 1, GL_FALSE, glm::value_p
    glUniform4f(g_WhiteDiffuseColor.lightIntensityUnif, 1.0f, 1.0f, 1.0f, 1.0f);
    g_pCylinderMesh->Render("lit");
}
glUseProgram(0);
```

The cylinder is not scaled at all. It is one unit from top to bottom, and the diameter of the cylinder is also 1. Translating it up by 0.5 simply moves it to being on top of the ground plane. Then we apply a rotation to it, based on user inputs.

We actually draw two different kinds of cylinders, based on user input. The colored cylinder is tinted red and is the initial cylinder. The white cylinder uses a vertex program that does not use per-vertex colors for the diffuse color; instead, it uses a hard-coded color of full white. These both come from the same mesh file, but have special names to differentiate between them.

What changes is that the “flat” mesh does not pass the color vertex attribute and the “tint” mesh does.

Other than which program is used to render them and what mesh name they use, they are both rendered similarly.

The camera-to-clip matrix is uploaded to the programs in the `reshape` function, as previous tutorials have demonstrated.

Vertex Lighting

There are two vertex shaders used in this tutorial. One of them uses a color vertex attribute as the diffuse color, and the other assumes the diffuse color is (1, 1, 1, 1). Here is the vertex shader that uses the color attribute, `DirVertexLighting_PCN`:

Example 9.4. Lighting Vertex Shader

```
#version 330

layout(location = 0) in vec3 position;
layout(location = 1) in vec4 diffuseColor;
layout(location = 2) in vec3 normal;

smooth out vec4 interpColor;

uniform vec3 dirToLight;
uniform vec4 lightIntensity;

uniform mat4 modelToCameraMatrix;
uniform mat3 normalModelToCameraMatrix;

layout(std140) uniform Projection
{
    mat4 cameraToClipMatrix;
};

void main()
{
    gl_Position = cameraToClipMatrix * (modelToCameraMatrix * vec4(position, 1.0));

 vec3 normCamSpace = normalize(normalModelToCameraMatrix * normal);

    float cosAngIncidence = dot(normCamSpace, dirToLight);
    cosAngIncidence = clamp(cosAngIncidence, 0, 1);

    interpColor = lightIntensity * diffuseColor * cosAngIncidence;
}
```

We define a single output variable, `interpColor`, which will be interpolated across the surface of the triangle. We have a uniform for the camera-space lighting direction `dirToLight`. Notice the name: it is the direction from the surface *towards* the light. It is not the direction *from* the light.

We also have a light intensity uniform value, as well as two matrices for positions and a separate one for normals. Notice that the `cameraToClipMatrix` is in a uniform block. This allows us to update all programs that use the projection matrix just by changing the buffer object.

The first line of `main` simply does the position transforms we need to position our vertices, as we have seen before. We do not need to store the camera-space position, so we can do the entire transformation in a single step.

The next line takes our normal and transforms it by the model-to-camera matrix specifically for normals. As noted earlier, the contents of this matrix are identical to the contents of `modelToCameraMatrix`. The `normalize` function takes the result of the transform and ensures that the normal has a length of one. The need for this will be explained later.

We then compute the cosine of the angle of incidence. We'll explain how this math computes this shortly. Do note that after computing the cosine of the angle of incidence, we then clamp the value to between 0 and 1 using the GLSL built-in function `clamp`.

This is important, because the cosine of the angle of incidence can be negative. This is for values which are pointed directly away from the light, such as the underside of the ground plane, or any part of the cylinder that is facing away from the light. The lighting computations do not make sense with this value being negative, so the clamping is necessary.

After computing that value, we multiply it by the light intensity and diffuse color. This result is then passed to the interpolated output color. The fragment shader is a simple passthrough shader that writes the interpolated color directly.

The version of the vertex shader without the per-vertex color attribute simply omits the multiplication with the `diffuseColor` (as well as the definition of that input variable). This is the same as doing a multiply with a color vector of all 1.0.

Vector Dot Product

We glossed over an important point in looking at the vertex shader. Namely, how the cosine of the angle of incidence is computed.

Given two vectors, one could certainly compute the angle of incidence, then take the cosine of it. But both computing that angle and taking its cosine are quite expensive. Instead, we elect to use a vector math trick: the *vector dot product.*

The vector dot product between two vectors can be mathematically computed as follows:

Equation 9.2. Dot Product

$$\vec{a} \cdot \vec{b} = \#\vec{a}\# * \#\vec{b}\# * \cos(\#)$$

If both vectors have a length of one (ie: they are unit vectors), then the result of a dot product is just the cosine of the angle between the vectors.

This is also part of the reason why the light direction is the direction *towards* the light rather than from the light. Otherwise we would have to negate the vector before performing the dot product.

What makes this faster than taking the cosine of the angle directly is that, while the dot product is geometrically the cosine of the angle between the two unit vectors, computing the dot product via vector math is very simple:

Equation 9.3. Dot Product from Vector Math

$$\vec{a} \cdot \vec{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \cdot \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = a_x * b_x + a_y * b_y + a_z * b_z$$

This does not require any messy cosine transcendental math computations. This does not require using trigonometry to compute the angle between the two vectors. Simple multiplications and additions; most graphics hardware can do billions of these a second.

Obviously, the GLSL function `dot` computes the vector dot product of its arguments.

Normal Transformation

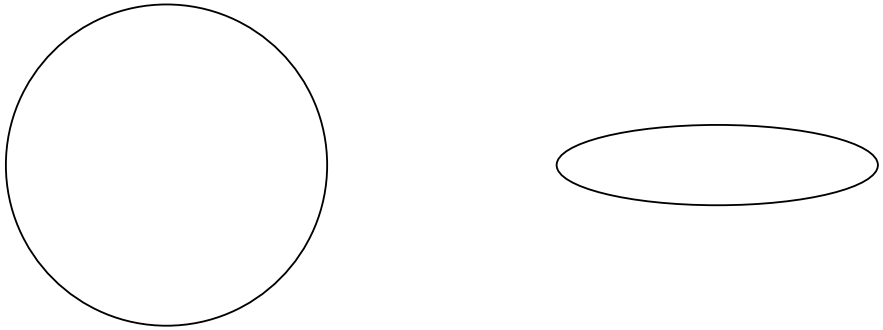
In the last section, we saw that our computation of the cosine of the angle of incidence has certain requirements. Namely, that the two vectors involved, the surface normal and the light direction, are of unit length. The light direction can be assumed to be of unit length, since it is passed directly as a uniform.

The surface normal can also be assumed to be of unit length. *Initially*. However, the normal undergoes a transformation by an arbitrary matrix; there is no guarantee that this transformation will not apply scaling or other transformations to the vector that will result in a non-unit vector.

Of course, it is easy enough to correct this. The GLSL function `normalize` will return a vector that is of unit length without changing the direction of the input vector.

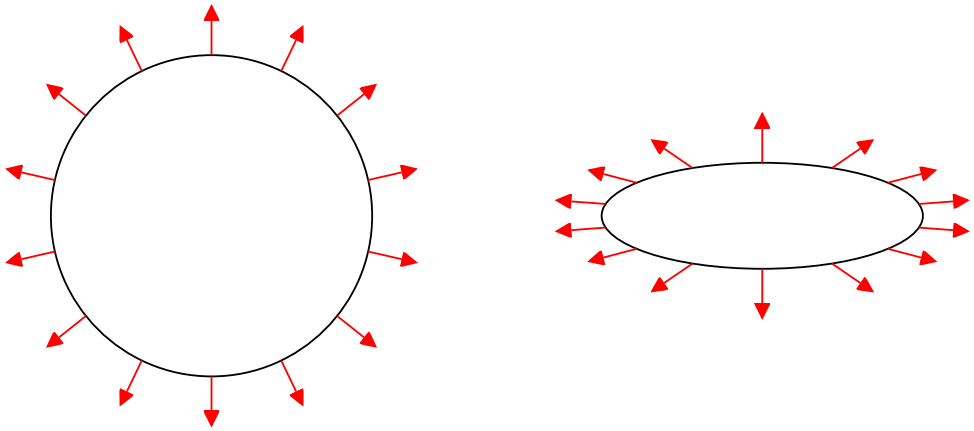
And while mathematically this would function, geometrically, it would be nonsense. For example, consider a 2D circle. We can apply a non-uniform scale (different scales in different axes) to the positions on this circle that will transform it into an ellipse:

Figure 9.7. Circle Scaling



This is all well and good, but consider the normals in this transformation:

Figure 9.8. Circle Scaling with Normals



The ellipse in the middle has the normals that you would expect if you transformed the normals from the circle by the same matrix the circle was transformed by. They may be unit length, but they no longer reflect the *shape* of the ellipse. The ellipse on the right has normals that reflect the actual shape.

It turns out that, what you really want to do is transform the normals with the same rotations as the positions, but invert the scales. That is, a scale of 0.5 along the X axis will shrink positions in that axis by half. For the surface normals, you want to *double* the X value of the normals, then normalize the result.

This is easy if you have a simple matrix. But more complicated matrices, composed from multiple successive rotations, scales, and other operations, are not so easy to compute.

Instead, what we must do is compute something called the *inverse transpose* of the matrix in question. This means we first compute the inverse matrix, then compute the *transpose* of that matrix. The transpose of a matrix is simply the same matrix flipped along the diagonal. The columns of the original matrix are the rows of its transpose. That is:

Equation 9.4. Matrix Transpose

$$M^{\mathrm{T}} = \begin{bmatrix} m11 & m21 & m31 & m41 \\ m12 & m22 & m32 & m42 \\ m13 & m23 & m33 & m43 \\ m14 & m24 & m34 & m44 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix}$$

So how does this inverse transpose help us?

Remember: what we want is to invert the scales of our matrix without affecting the rotational characteristics of our matrix. Given a 3x3 matrix M that is composed of only rotation and scale transformations, we can re-express this matrix as follows:

$$\mathbf{M} = \boldsymbol{R}_1 * \mathbf{S} * \boldsymbol{R}_2$$

That is, the matrix can be expressed as doing a rotation into a space, followed by a single scale transformation, followed by another rotation. We can do this *regardless* of how many scale and rotation matrices were used to build M. That is, M could be the result of twenty rotation and scale matrices, but all of those can be extracted into two rotations with a scale inbetween.[1]

Recall that what we want to do is invert the scales in our transformation. Where we scale by 0.4 in the original, we want to scale by 2.5 in the inverse. The inverse matrix of a pure scale matrix is a matrix with each of the scaling components inverted. Therefore, we can express the matrix that we actually want as this:

$$\mathbf{M}_{\mathbf{want}} = \boldsymbol{R}_1 * \mathbf{S}^{-1} * \boldsymbol{R}_2$$

An interesting fact about pure-rotation matrices: the inverse of any rotation matrix *is* its transpose. Also, taking the inverse of a matrix twice results in the original matrix. Therefore, you can express any pure-rotation matrix as the inverse transpose of itself, without affecting the matrix. Since the inverse is its transpose, and doing a transpose twice on a matrix does not change its value, the inverse-transpose of a rotation matrix is a no-op.

Also, since the values in pure-scale matrices are along the diagonal, a transpose operation on scale matrices does nothing. With these two facts in hand, we can re-express the matrix we want to compute as:

$$\mathbf{M}_{\mathbf{want}} = \left(\boldsymbol{R}_1^{-1}\right)^T * \left(\mathbf{S}^{-1}\right)^T * \left(\boldsymbol{R}_2^{-1}\right)^T$$

Using matrix algebra, we can factor the transposes out, but doing so requires reversing the order of the matrix multiplication:

$$\mathbf{M}_{\mathbf{want}} = \left(\boldsymbol{R}_2^{-1} * \mathbf{S}^{-1} * \boldsymbol{R}_1^{-1}\right)^T$$

[1] We will skip over deriving how exactly this is true. If you are interested, search for "Singular Value Decomposition [http://en.wikipedia.org/wiki/Singular_value_decomposition]". But be warned: it is math-heavy.

Similar, we can factor out the inverse operations, but this requires reversing the order again:

$$\mathbf{M}_{\mathbf{want}} = \left(\left(\boldsymbol{R}_1 * \mathbf{S} * \boldsymbol{R}_2\right)^{-1}\right)^T = \left(\mathbf{M}^{-1}\right)^T$$

Thus, the inverse-transpose solves our problem. And both GLM and GLSL have nice functions that can do these operations for us. Though really, if you can avoid doing an inverse-transpose in GLSL, you are *strongly* advised to do so; this is not a trivial computation.

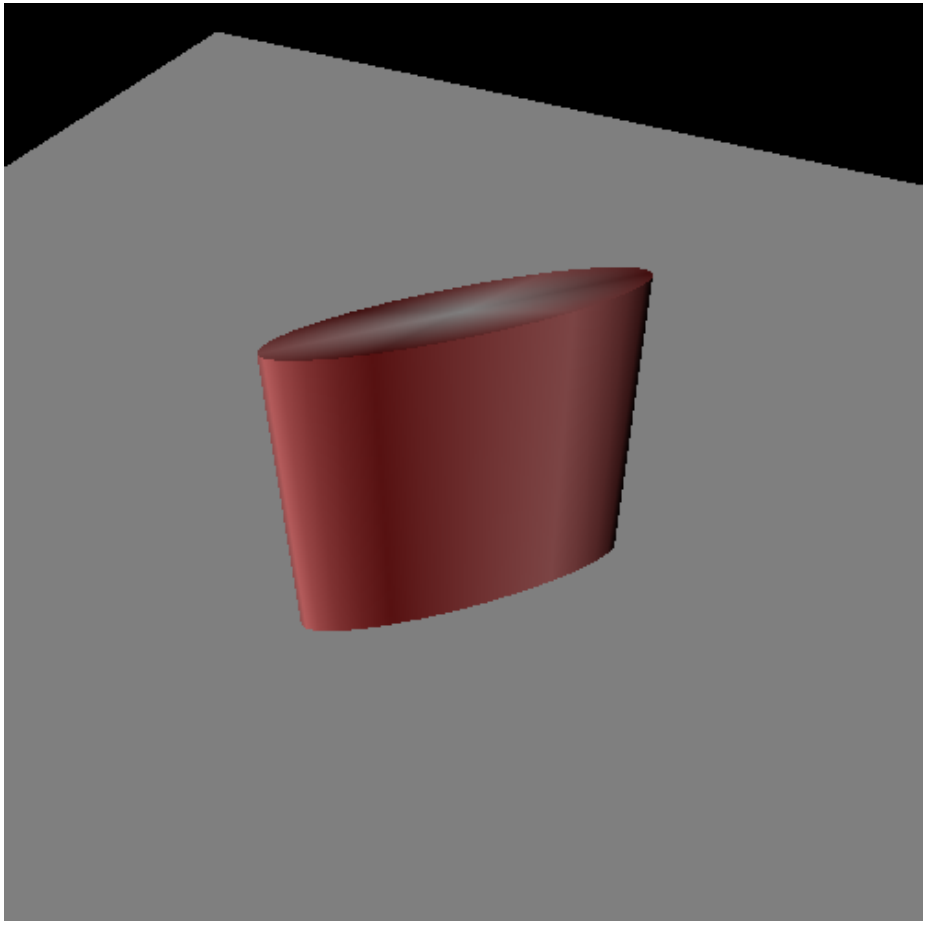
We do this in the Scale and Lighting tutorial. It controls mostly the same as the previous tutorial, with a few exceptions. Pressing the space bar will toggle between a regular cylinder and a scaled one. The "T" key will toggle between properly using the inverse-transpose (the default) and not using the inverse transpose. The rendering code for the cylinder is as follows:

Example 9.5. Lighting with Proper Normal Transform

```
glutil::PushStack push(modelMatrix);

modelMatrix.ApplyMatrix(g_objtPole.CalcMatrix());

if(g_bScaleCyl)
{
    modelMatrix.Scale(1.0f, 1.0f, 0.2f);
}

glUseProgram(g_VertexDiffuseColor.theProgram);
glUniformMatrix4fv(g_VertexDiffuseColor.modelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(modelM
glm::mat3 normMatrix(modelMatrix.Top());
if(g_bDoInvTranspose)
{
    normMatrix = glm::transpose(glm::inverse(normMatrix));
}
glUniformMatrix3fv(g_VertexDiffuseColor.normalModelToCameraMatrixUnif, 1, GL_FALSE, glm::value_ptr(
glUniform4f(g_VertexDiffuseColor.lightIntensityUnif, 1.0f, 1.0f, 1.0f, 1.0f);
g_pCylinderMesh->Render("lit-color");
glUseProgram(0);
```

It's pretty self-explanatory.

Figure 9.9. Lighting and Scale



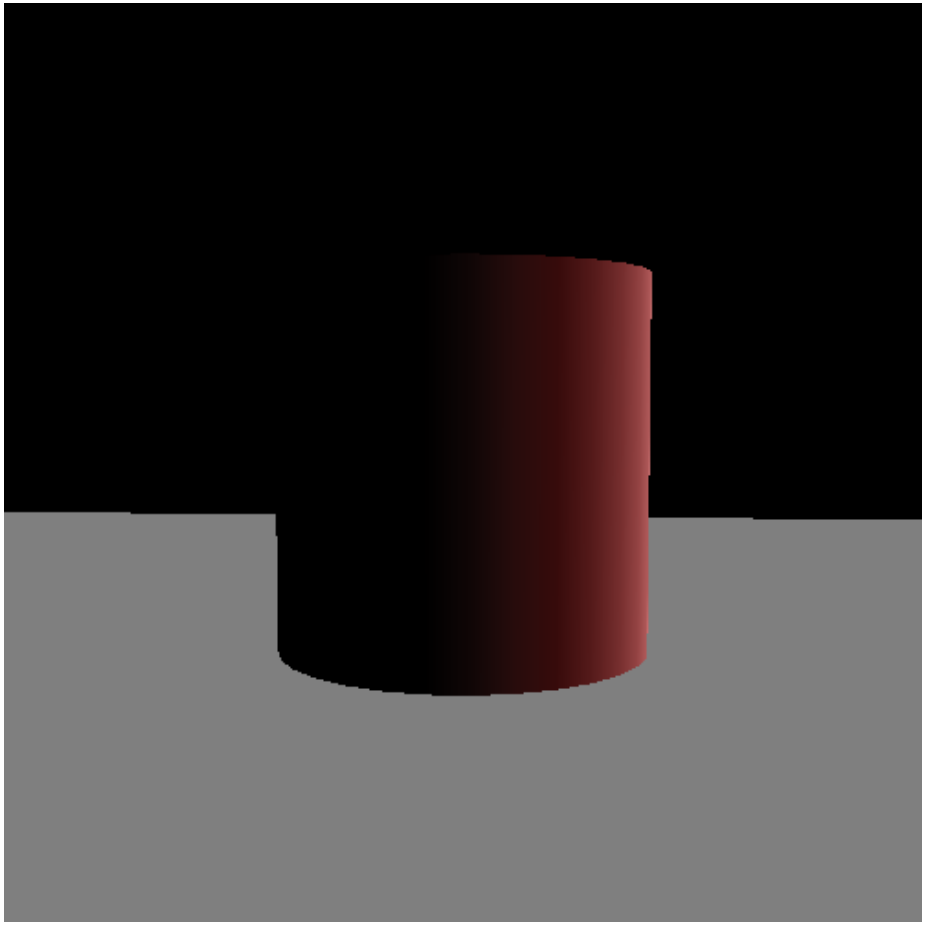
One more thing to note before we move on. Doing the inverse-transpose is only really necessary if you are using a *non-uniform* scale. In practice, it's actually somewhat rare to use this kind of scale factor. We do it in these tutorials, so that it is easier to build models from simple geometric components. But when you have an actual modeller creating objects for a specific purpose, non-uniform scales generally are not used. At least, not in the output mesh. It's better to just get the modeller to adjust the model as needed in their modelling application.

Uniform scales are more commonly used. So you still need to normalize the normal after transforming it with the model-to-camera matrix, even if you are not using the inverse-transpose.

Global Illumination

You may notice something very unrealistic about the results of this tutorial. For example, take this image:

Figure 9.10. Half Lit



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 reason for this is somewhat complicated.

Consider a scene of the outdoors. In normal daylight, there is exactly one light source: the sun. Objects that are in direct sunlight appear to be bright, and objects that have some object between them and the sun are in shadow.

But think about what those shadows look like. They're not 100% black. They're certainly darker than the surrounding area, but they still have some color. And remember: we only see anything because our eyes detect light. In order to see an object in the shadow of a light source, that object must either be emitting light directly or reflecting light that came from somewhere else. Grass is not known for its light-emitting qualities, so where does the light come from?

Think about it. We see because an object reflects light into our eyes. But our eyes are not special; the object does not reflect light *only* into our eyes. It reflects light in all directions. Not necessarily at the same intensity in each direction, but objects that reflect light tend to do so in all directions to some degree. What happens when that light hits another surface?

The same thing that happens when light hits any surface: some of it is absorbed, and some is reflected in some way.

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 contributer 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.

As you might imagine, modelling global illumination is hard. *Very* hard. It is typically a subtle effect, but in many scenes, particularly outdoor scenes, it is almost a necessity to provide at least basic global illumination modelling in order to achieve a decent degree of photorealism. Incidentally, this is a good part of the reason why most games tend to avoid outdoor scenes or light outdoor scenes as though the sky were cloudy or overcast. This neatly avoids needing to do complex global illumination modelling by damping down the brightness of the sun to levels when interreflection would be difficult to notice.

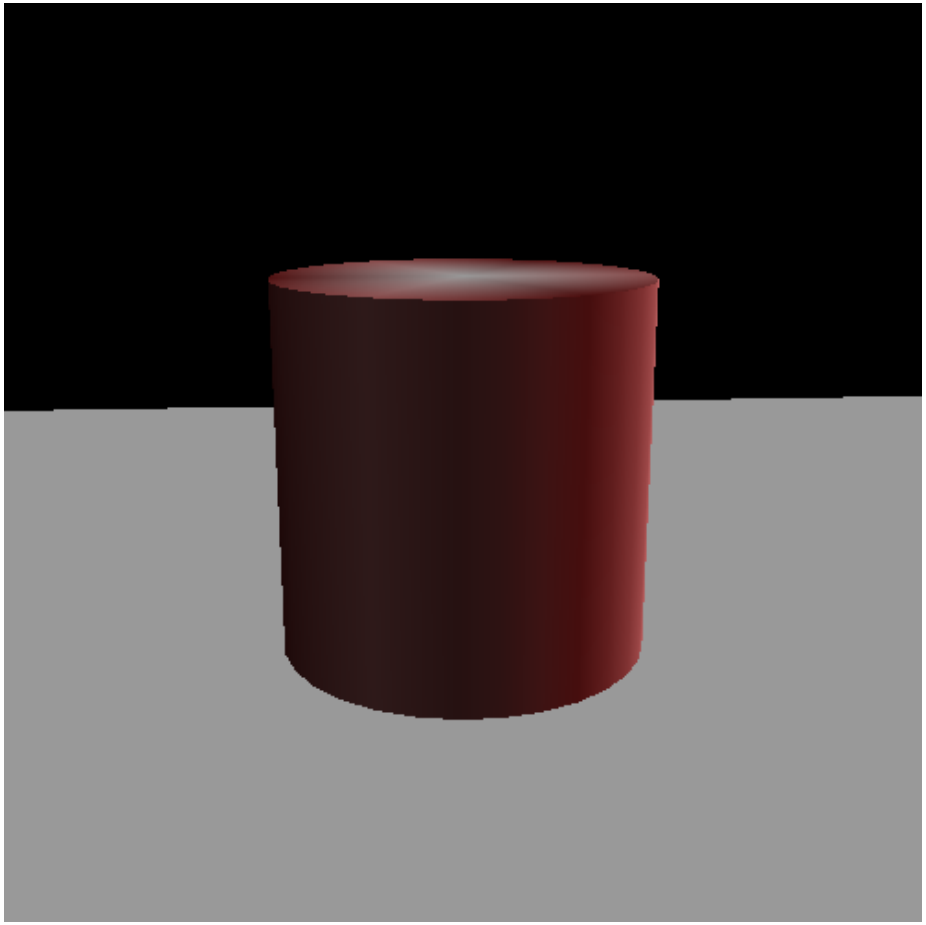
Having this completely black area in our rendering looks incredibly fake. Since doing actual global illumination modelling is hard, we will instead use a time-tested technique: *ambient lighting.*

The ambient lighting "model"[2] 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 is simply the ambient light intensity * the diffuse surface color.

We do this in the Ambient Lighting tutorial. The controls are the same as the last tutorial, except that the space bar swaps between the two cylinders (red and white), and that the **T** key toggles ambient lighting on and off (defaults to off).

[2] I put the word model in quotations because ambient lighting is so divorced from anything in reality that it does not really deserve to be called a model. That being said, just because it does not actually model global illumination in any real way does not mean that it is not *useful*.

Figure 9.11. Ambient Lighting



The detail seen in the dark portion of the cylinder only comes from the diffuse color. And because the ambient is fairly weak, the diffuse color of the surface appears muted in the dark areas.

The rendering code now uses four of vertex shaders instead of two. Two of them are used for non-ambient lighting, and use the same shaders we have seen before, and the other two use ambient lighting.

The ambient vertex shader that uses per-vertex colors is called `DirAmbVertexLighting_PCN.vert` and reads as follows:

Example 9.6. Ambient Vertex Lighting

```
#version 330

layout(location = 0) in vec3 position;
```

```
layout(location = 1) in vec4 diffuseColor;
layout(location = 2) in vec3 normal;

smooth out vec4 interpColor;

uniform vec3 dirToLight;
uniform vec4 lightIntensity;
uniform vec4 ambientIntensity;

uniform mat4 modelToCameraMatrix;
uniform mat3 normalModelToCameraMatrix;

layout(std140) uniform Projection
{
    mat4 cameraToClipMatrix;
};

void main()
{
    gl_Position = cameraToClipMatrix * (modelToCameraMatrix * vec4(position, 1.0));

    vec3 normCamSpace = normalize(normalModelToCameraMatrix * normal);

    float cosAngIncidence = dot(normCamSpace, dirToLight);
    cosAngIncidence = clamp(cosAngIncidence, 0, 1);

    interpColor = (diffuseColor * lightIntensity * cosAngIncidence) +
        (diffuseColor * ambientIntensity);
}
```

It takes two uniforms that specify lighting intensity. One specifies the intensity for the diffuse lighting, and the other for the ambient lighting. The only other change is to the last line in the shader. The usual diffuse lighting result has its value added to the ambient lighting computation. Also, note that the contribution from two lighting models is added together.

Of particular note is the difference between the lighting intensities in the pure-diffuse case and the diffuse+ambient case:

Example 9.7. Lighting Intensity Settings

```
if(g_bShowAmbient)
{
    glUseProgram(whiteDiffuse.theProgram);
    glUniform4f(whiteDiffuse.lightIntensityUnif, 0.8f, 0.8f, 0.8f, 1.0f);
    glUniform4f(whiteDiffuse.ambientIntensityUnif, 0.2f, 0.2f, 0.2f, 1.0f);
    glUseProgram(vertexDiffuse.theProgram);
    glUniform4f(vertexDiffuse.lightIntensityUnif, 0.8f, 0.8f, 0.8f, 1.0f);
    glUniform4f(vertexDiffuse.ambientIntensityUnif, 0.2f, 0.2f, 0.2f, 1.0f);
}
else
{
    glUseProgram(whiteDiffuse.theProgram);
    glUniform4f(whiteDiffuse.lightIntensityUnif, 1.0f, 1.0f, 1.0f, 1.0f);
    glUseProgram(vertexDiffuse.theProgram);
    glUniform4f(vertexDiffuse.lightIntensityUnif, 1.0f, 1.0f, 1.0f, 1.0f);
}
```

In the pure-diffuse case, the light intensity is full white. But in the ambient case, we deliberately set the diffuse intensity to less than full white. This is very intensional.

We will talk more about this issue in the future, but it is very critical that light intensity values not exceed 1.0. This includes *combined* lighting intensity values. OpenGL clamps colors that it writes to the output image to the range [0, 1]. So any light intensity that exceeds 1.0, whether alone or combined with other lights, can cause unpleasant visual effects.

There are ways around this, and those ways will be discussed in the eventual future.

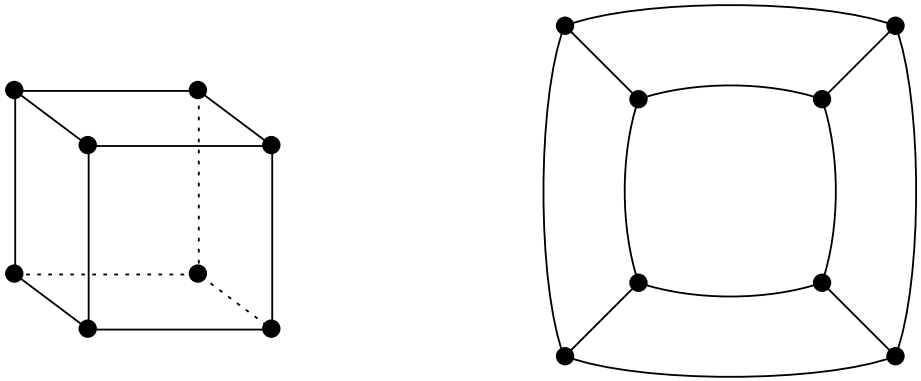
Mesh Topology

Thus far, we have seen three different kinds of vertex attributes. We have used positions, colors, and now normals. Before we can continue, we need to discuss the ramifications of having three independent vertex attributes on our meshes.

A mesh's *topology* defines the connectedness between the vertices. However, each vertex attribute can have its own separate topology. How is this possible?

Consider a simple cube. It has 8 vertex positions. The topology between the positions of the cube is as follows:

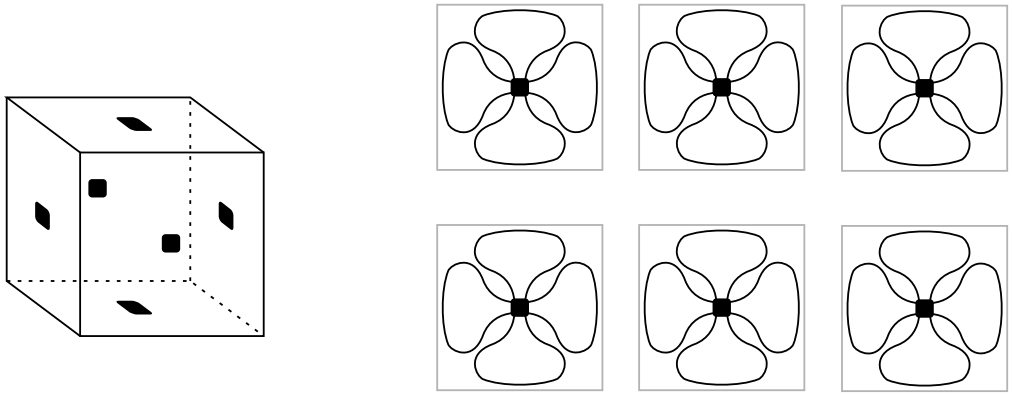
Figure 9.12. Cube Position Topology



The topology diagram has the 8 different positions, with each position connected to three neighbors. The connections represent the edges of the cube faces, and the area bounded by connections represent the faces themselves. So each face has four edges and four positions.

Now consider the topology of the normals of a cube. A cube has 6 faces, each of which has a distinct normal. The topology is a bit unorthodox:

Figure 9.13. Cube Normal Topology



The square points represent distinct normals, not positions in space. The connections between normals all loop back on themselves. That's because each vertex of each face uses the same normal. While the front face and the top face share two vertex positions in common, they share no vertex normals at all. Therefore, there is no topological relation between the front normal and the top normal. Each normal value is connected to itself four times, but it is connected to nothing else.

We have 8 positions and 6 normals. They each have a unique topology over the mesh. How do we turn this into something we can render?

If we knew nothing about OpenGL, this would be simple. We simply use the topologies to build the meshes. Each face is broken down into two triangles. We would have an array of 8 positions and 6 normals. And for each vertex we render, we would have a list of indices that fetch values from these two arrays.

The index list for two faces of the cube might look like this (using a C++-style multidimensional list):

```
{
  {0, 0}, {1, 0}, {2, 0}, {1, 0}, {3, 0}, {2, 0},
  {0, 1}, {4, 1}, {1, 1}, {4, 1}, {6, 1}, {1, 1},
}
```

The first index in each element of the list pulls from the position array, and the second index pulls from the normal array. This list explicitly specifies the relationship between faces and topology: the first face (composed of two triangles) contains 4 positions, but uses the same normal for all vertices. The second face shares two positions with the first, but no normals.

This is complicated in OpenGL because of one simple reason: we only get *one* index list. When we render with an index array, every attribute array uses the same index. Therefore, what we need to do is convert the above index list into a list of unique combinations of vertex attributes. So each pair of indices must refer to a unique index.

Consider the first face. It consists of 4 unique vertices; the index pairs {1, 0} and {2, 0} are repeated, since we must draw triangles. Since these pairs are repeats, we can collapse them; they will refer to the same index. However, the fact that each vertex uses the same normal must be ignored entirely. Therefore, the final index list we use for the first face is:

```
{ 0, 1, 2, 1, 3, 2 }
```

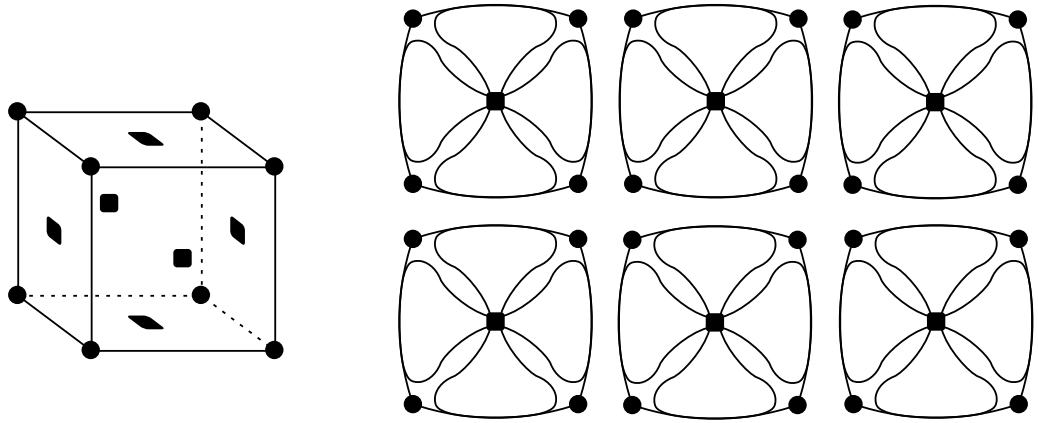
The attribute arrays for this one face contain 4 positions and 4 normals. But while the positions are all different, the normals must all be the same value. Even though they are stored in four separate locations. This seems like a waste of memory.

This gets worse once we move on to the next face. Because we can only collapse index pairs that are identical, absolutely none of the vertices in the second face share indices with the first face. The fact that we reuse two positions in the next face is irrelevant: we must have the following index list:

```
{ 4, 5, 6, 5, 7, 6 }
```

The attribute array for both faces contains 8 positions and 8 normals. Again, the normals for the second face are all duplicates of each other. And there are two positions that are duplicated. Topologically speaking, our cube vertex topology looks like the following:

Figure 9.14. Full Cube Topology



Each face is entirely distinct topologically from the rest.

In the end, this gives us 24 positions and 24 normals in our arrays. There will only be 6 distinct normal values and 8 distinct position values in the array, but there will be 24 of each. So this represents a significant increase in our data size.

Do not be too concerned about the increase in data however. A cube, or any other faceted object, represents the worst-case scenario for this kind of conversion. Most actual meshes of smooth objects have much more interconnected topologies.

Mesh topology is something to be aware of, as is the ability to convert attribute topologies into forms that OpenGL can directly process. Each attribute has its own topology which affects how the index list is built. Different attributes can share the same topology. For example, we could have colors associated with each face of the cube. The color topology would be identical to the normal topology.

Most of the details are usually hidden behind various command-line tools that are used to generate proper meshes from files exported from modelling programs. Many videogame developers have complex asset conditioning pipelines that process exported files into binary formats suitable for loading and direct upload into buffer objects. But it is important to understand how mesh topologies work.

In Review

In this tutorial, you have learned the following:

- Diffuse lighting is a simple lighting model based on the angle between the light source and the surface normal.
- Surface normals are values used, per-vertex, to define the direction of the surface at a particular location. They do not have to mirror the actual normal of the mesh geometry.
- Surface normals must be transformed by the inverse-transpose of the model-to-camera matrix, if that matrix can involve a non-uniform scale operation.

- Light interreflection can be approximated by adding a single light intensity that has no direction.
- Each vertex attribute has its own topology. In order to render these vertices in OpenGL, attribute data must be replicated so that each unique combination of attributes has a topology.

Further Study

Try doing these things with the given programs.

- Modify the ambient lighting tutorial, bumping the diffuse light intensity up to 1.0. See how this effects the results.
- Change the shaders in the ambient lighting tutorial to use the lighting intensity correction mentioned above. Divide the diffuse color by a value, then pass larger lighting intensities to the shader. Notice how this changes the quality of the lighting.

Further Research

Lambertian diffuse reflectance is a rather good model for diffuse reflectance for many surfaces. Particularly rough surfaces however do not behave in a Lambertian manor. If you are interested in modelling such surfaces, investigate the Oren-Nayar reflectance model.

GLSL Functions of Note

```
vec clamp(vec val, vec minVal, vec maxVal);
```

This function does a clamping operation of each component of *`val`*. All of the parameters must scalars or vectors of the same dimensionality. This function will work with any scalar or vector type. It returns a scalar or vector of the same dimensionality as the parameters, where each component of *`val`* will be clamped to the closed range [*`minVal`*, *`maxVal`*]. This is useful for ensuring that values are in a certain range.

All components of *`minVal`* *must* be smaller than the corresponding components of *`maxVal`*.

```
float dot(vec x, vec y);
```

This function performs a vector dot product on *`x`* and *`y`*. This always results in a scalar value. The two parameters must have the same dimensionality and must be vectors.

```
vec normalize(vec x);
```

This function returns a vector in the same direction as *`x`*, but with a length of 1. *`x`* must have a length greater than 0 (that is, it cannot be a vector with all zeros).

Glossary

photorealism	A rendering system has achieved photorealism when it can render a still image that is essentially indistinguishable from a real photograph.
lighting model	A mathematical model that defines how light is absorbed and reflected from a surface. This can attempt to model reality, but it does not have to.
light source	Mathematically, this is something that produces light and adds it to a scene. It does not have to be an actual object shown in the world.
light intensity	The intensity, measured in RGB, of light emitted from a light-casting source.
angle of incidence	The angle between the surface normal and the direction towards the light.
diffuse lighting	A lighting model that assumes light is reflected from a surface in many directions, as opposed to a flat mirror that reflects light in one direction.
Lambertian reflectance	A particular diffuse lighting model that represents the ideal diffuse case: lighting is reflected evenly in all directions.

surface normal, normal	The direction that a particular point on a surface faces.
Gouraud shading	Computing lighting computations at every vertex, and interpolating the results of these computations across the surface of the triangle.
directional light source	A light source that emits light along a particular direction. Every point in the scene to be rendered receives light from the same direction. This models a very distant light source that lights the scene evenly from a single direction.
vector dot product	Computes the length of the projection of one vector onto another. If the two vectors are unit vectors, then the dot product is simply the cosine of the angle between them.
transpose	A matrix operation that flips the matrix along the main diagonal. The columns of the original matrix become the rows of the transpose.
inverse transpose	A matrix operation, where a matrix is inverted and then transposed.
interreflection	Light that reflects off of multiple surfaces before reaching the viewer.
global illumination	A category of lighting models that take into account lighting contributions from interreflection.
local illumination, direct illumination	Lighting computations made only from light sources that cast light directly onto the surface.
ambient lighting	A lighting model that models all contributions from interreflection as a single light intensity that does not originate from any particular direction.
mesh topology	The interconnected nature between different values of a vertex attribute in a mesh. Each attribute has its own separate topology. Rendering in OpenGL requires finding all of the unique combinations of attributes and building a new topology out of it, where each attributes topology is the same. This can require replicating attribute data.

Chapter 10. Plane Lights

Directional lights are useful for representing light sources like the sun and so forth. But most light sources are more likely to be represented as point lights.

A *point light source* is a light source that has a position in the world and shines with equal intensity in all directions. Our simple diffuse lighting equation is a function of these properties:

- The surface normal at that point.
- The direction from the point on the surface to the light.

The direction to the light source from the point is a constant when dealing with directional light. It is a parameter for lighting, but it is a constant value for all points in the scene. The difference between directional lighting and point lights is only that this direction must be computed for each position in the scene.

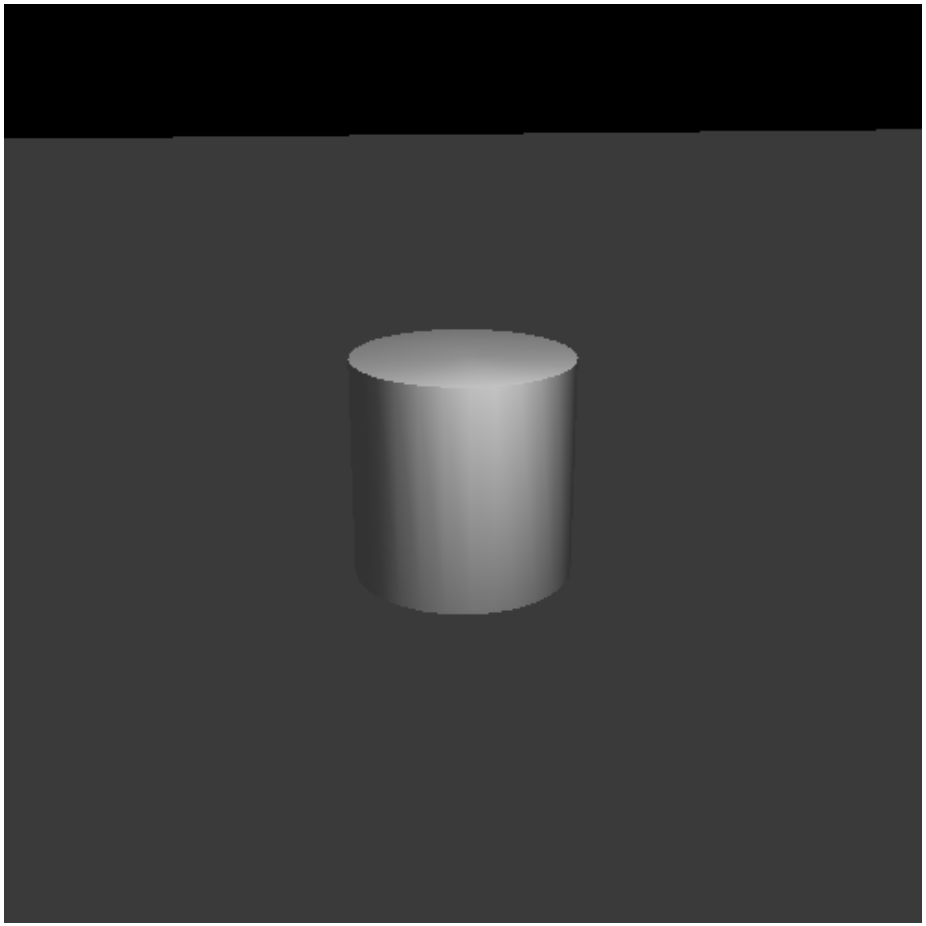
Computing this is quite simple. At the point of interest, we take the difference between the point on the surface and the light's position. We normalize the result to produce a unit vector direction to the light. Then we use the light direction as we did before. The surface point, light position, and surface normal must all be in the same space for this equation to make sense.

Vertex Point Lighting

Thus far, we have computed the lighting equation at each vertex and interpolated the results across the surface of the triangle. We will continue to do so for point lights. For the moment, at least.

We implement point lights per-vertex in the Vertex Point Lighting tutorial. This tutorial has a moving point light that circles around the cylinder.

Figure 10.1. Vertex Point Lighting



To toggle an indicator of the light's position, press the **Y** key. The **B** key will toggle rotation of the light. The **I** and **K** keys move the light up and down respectively, while the **J** and **L** keys will decrease and increase the light's radius. Holding shift with these keys will move in smaller increments.

Most of the code is nothing we have not seen elsewhere. The main changes are at the top of the rendering function.

Example 10.1. Per-Vertex Point Light Rendering

```
glutil::MatrixStack modelMatrix;
modelMatrix.SetMatrix(g_viewPole.CalcMatrix());

const glm::vec4 &worldLightPos = CalcLightPosition();
```

```
glm::vec4 lightPosCameraSpace = modelMatrix.Top() * worldLightPos;

glUseProgram(g_WhiteDiffuseColor.theProgram);
glUniform3fv(g_WhiteDiffuseColor.lightPosUnif, 1, glm::value_ptr(lightPosCameraSpace));
glUseProgram(g_VertexDiffuseColor.theProgram);
glUniform3fv(g_VertexDiffuseColor.lightPosUnif, 1, glm::value_ptr(lightPosCameraSpace));
```

The light is computed initially in world space, then transformed into camera space. The camera-space light position is given to both of the shaders. Rendering proceeds normally from there.

Our vertex shader, `PosVertexLighting_PCN.vert` has had a few changes:

Example 10.2. Per-Vertex Point Light Vertex Shader

```
#version 330

layout(location = 0) in vec3 position;
layout(location = 1) in vec4 diffuseColor;
layout(location = 2) in vec3 normal;

smooth out vec4 interpColor;

uniform vec3 lightPos;
uniform vec4 lightIntensity;
uniform vec4 ambientIntensity;

uniform mat4 modelToCameraMatrix;
uniform mat3 normalModelToCameraMatrix;

uniform Projection
{
    mat4 cameraToClipMatrix;
};

void main()
{
    vec4 cameraPosition = (modelToCameraMatrix * vec4(position, 1.0));
    gl_Position = cameraToClipMatrix * cameraPosition;

    vec3 normCamSpace = normalize(normalModelToCameraMatrix * normal);

    vec3 dirToLight = normalize(lightPos - vec3(cameraPosition));

    float cosAngIncidence = dot(normCamSpace, dirToLight);
    cosAngIncidence = clamp(cosAngIncidence, 0, 1);

    interpColor = (diffuseColor * lightIntensity * cosAngIncidence) +
        (diffuseColor * ambientIntensity);
}
```

The vertex shader takes a camera-space light position instead of a camera-space light direction. It also stores the camera-space vertex position in a temporary in the first line of `main`. This is used to compute the direction to the light. From there, the computation proceeds normally.

Note the order of operations in computing `dirToLight`. The `lightPos` is on the left and the `cameraPosition` is on the right. Geometrically, this is correct. If you have two points, and you want to find the direction from point A to point B, you compute B - A. The `normalize` call is just to convert it into a unit vector.

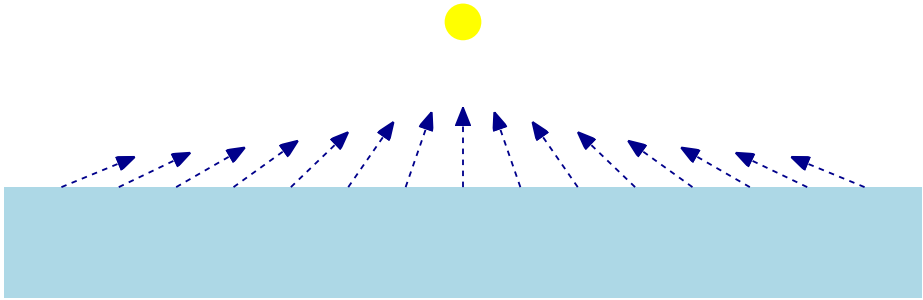
Interpolation

As you can see, doing point lighting is quite simple. Unfortunately, the visual results are not.

For example, use the controls to display the position of the point light source, then position it near the ground plane. See anything wrong?

If everything were working correctly, one would expect to see a bright area directly under the light. After all, geometrically, this situation looks like this:

Figure 10.2. Light Near Surface



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.

You can see this is evident by putting the light position next to the cylinder. If the light is at the top or bottom of the cylinder, then the area near the light will be bright. But if you move the light to the middle of the cylinder, far the top or bottom vertices, then the illumination will be much dimmer.

This is not the only problem with doing per-vertex lighting. For example, run the tutorial again and do not move the light. Just watch how the light behaves on the cylinder's surface as it animates around. Unlike with directional lighting, you can very easily see the triangles on the cylinder's surface. Though the per-vertex computations are not helping matters, the main problem here has to do with interpolating the values.

If you move the light source farther away, you can see that the triangles smooth out and become indistinct from one another. But this is simply because, if the light source is far enough away, the results are indistinguishable from a directional light. Each vertex's direction to the light is almost the same as each other vertex's direction to the light.

Per-vertex lighting was reasonable when dealing with directional lights. But it simply is not a good idea for point lighting. The question arises: why was per-vertex lighting good with directional lights to begin with?

Remember that our diffuse lighting equation has two parameters: the direction to the light and the surface normal. In directional lighting, the direction to the light is always the same. Therefore, the only value that changes over a triangle's surface is the surface normal.

Linear interpolation of vectors looks like this:

$$V_a\# + V_b(1 - \#)$$

The α in the equation is the factor of interpolation between the two values. When α is one, we get V_a, and when it is zero, we get V_b. The two values, V_a and V_b can be scalars or vectors.

Our diffuse lighting equation is this:

$$D * I * \left(\hat{N} \cdot \hat{L}\right)$$

If the surface normal N is being interpolated, then at any particular point on the surface, we get this equation for a directional light (the light direction L does not change):

$$D * I * \left(\hat{L} \cdot \left(\hat{N}_a\# + \hat{N}_b(1 - \#)\right)\right)$$

The dot product is distributive, like scalar multiplication. So we can distribute the L to both sides of the dot product term:

$$D * I * \left(\left(\hat{L} \cdot \left(\hat{N}_a\#\right)\right) + \left(\hat{L} \cdot \left(\hat{N}_b(1 - \#)\right)\right)\right)$$

We can extract the linear terms from the dot product. Remember that the dot product is the cosine of the angle between two vectors, times the length of those vectors. The two scaling terms directly modify the length of the vectors. So they can be pulled out to give us:

$$D * I * \left(\#\left(\hat{L} \cdot \hat{N}_a\right) + (1 - \#)\left(\hat{L} \cdot \hat{N}_b\right)\right)$$

Recall that vector/scalar multiplication is distributive. We can distribute the multiplication by the diffuse color and light intensity to both terms. This gives us:

$$\left(D * I * \#\left(\hat{L} \cdot \hat{N}_a\right)\right) + \left(D * I * (1 - \#)\left(\hat{L} \cdot \hat{N}_b\right)\right)$$
$$\left(D * I * \left(\hat{L} \cdot \hat{N}_a\right)\right)\# + \left(D * I * \left(\hat{L} \cdot \hat{N}_b\right)\right)(1 - \#)$$

This means that if L is constant, linearly interpolating N is exactly equivalent to linearly interpolating the results of the lighting equation. And the addition of the ambient term does not change this, since it is a constant and would not be affected by linear interpolation.

When doing point lighting, you would have to interpolate both N and L. And that does not yield the same results as linearly interpolating the two colors you get from the lighting equation. This is a big part of the reason why the cylinder does not look correct.

The more physically correct method of lighting is to perform lighting at every rendered pixel. To do that, we would have to interpolate the lighting parameters across the triangle, and perform the lighting computation in the fragment shader.

Fragment Lighting

So, in order to deal with interpolation artifacts, we need to interpolate the actual light direction and normal, instead of just the results of the lighting equation. This is called per-fragment lighting or just *fragment lighting.*

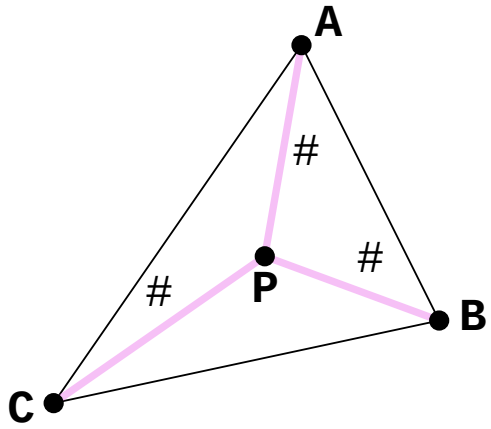
This is pretty simple, conceptually. We simply need to do the lighting computations in the fragment shader. So the fragment shader needs the position of the fragment, the light's position (or the direction to the light from that position), and the surface normal of the fragment at that position. And all of these values need to be in the same coordinate space.

There is a problem that needs to be dealt with first. Normals do not interpolate well. Or rather, wildly different normals do not interpolate well. And light directions can be very different if the light source is close to the triangle relative to that triangle's size.

Consider the large plane we have. The direction toward the light will be very different at each vertex, so long as our light remains in relatively close proximity to the plane.

Part of the problem is with interpolating values along the diagonal of our triangle. Interpolation within a triangle works like this. For any position within the area of the triangle, that position can be expressed as a weighted sum of the positions of the three vertices.

Figure 10.3. Triangle Interpolation

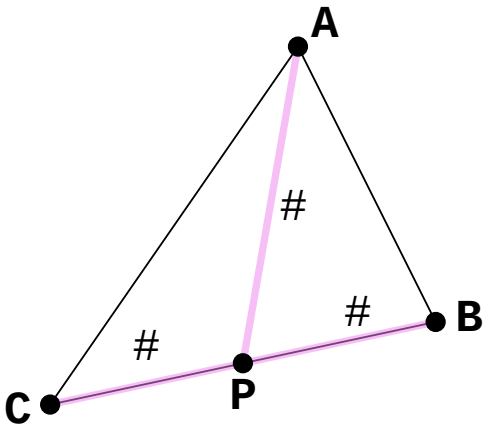


P = #A + #B + #C, where $\alpha + \beta + \gamma = 1.0$

The α, β, and γ values are not the distances from their respective points to the point of interest. In the above case, the point P is in the exact center of the triangle. Thus, the three values are each #.

If the point of interest is along an edge of the triangle, then the contribution of the vertex not sharing that edge is zero.

Figure 10.4. Triangle Edge Interpolation

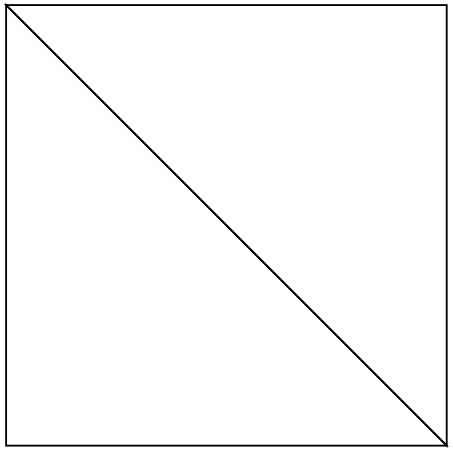


Here, point P is exactly halfway between points C and B. Therefore, β, and γ are both 0.5, but α is 0.0. If point P is anywhere along the edge of a triangle, it gets none of its final interpolated value from the third vertex. So along a triangle's edge, it acts like the kind of linear interpolation we have seen before.

This is how OpenGL interpolates the vertex shader outputs. It takes the α, β, and γ coordinates for the fragment's position and combines them with the vertex output value for the three vertices in the same way it does for the fragment's position. There is slightly more to it than that, but we will discuss that later.

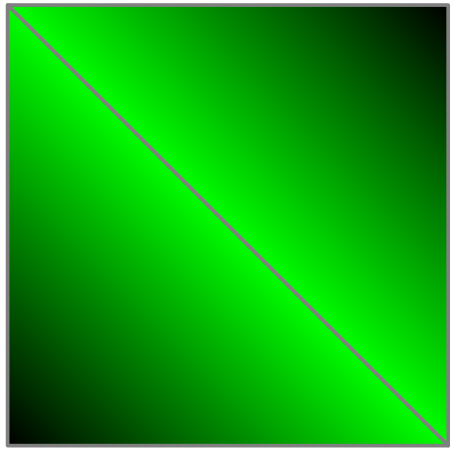
The ground plane in our example is made of two large triangles. They look like this:

Figure 10.5. Two Triangle Quadrilateral



What happens if we put the color black on the top-right and bottom-left points, and put the color green on the top-left and bottom-right points? If you interpolate these across the surface, you would get this:

Figure 10.6. Two Triangle Interpolation



The color is pure green along the diagonal. That is because along a triangle's edge, the value interpolated will only be the color of the two vertices along that edge. The value is interpolated based only on each triangle individually, not on extra data from another neighboring triangle.

In our case, this means that for points along the main diagonal, the light direction will only be composed of the direction values from the two vertices on that diagonal. This is not good. This would not be much of a problem if the light direction did not change much along the surface, but with large triangles (relative to how close the light is to them), that is simply not the case.

Since we cannot interpolate the light direction very well, we need to interpolate something else. Something that does exhibit the characteristics we need when interpolated.

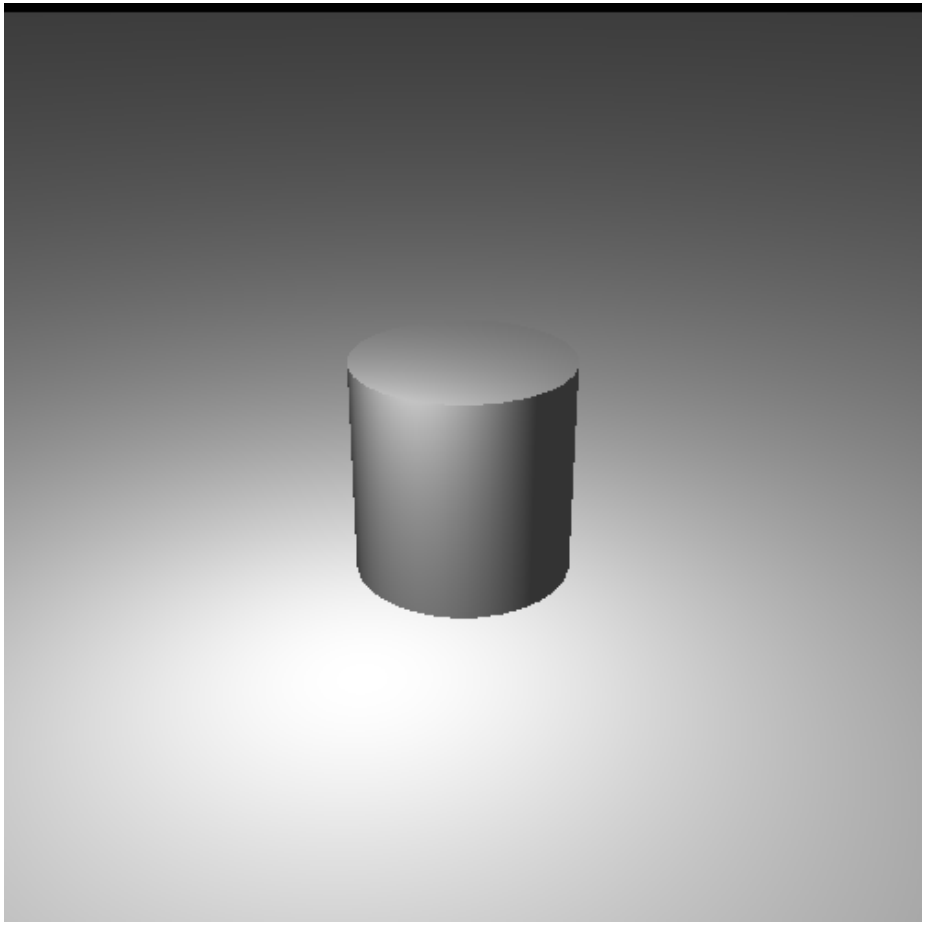
Positions interpolate quite well. Interpolating the top-left position and bottom-right positions gets an accurate position value along the diagonal. So instead of interpolating the light direction, we interpolate the components of the light direction. Namely, the two positions. The light position is a constant, so we only need to interpolate the vertex position.

Now, we could do this in any space. But for illustrative purposes, we will be doing this in model space. That is, both the light position and vertex position will be in model space.

One of the advantages of doing things in model space is that it gets rid of that pesky matrix inverse/transpose we had to do to transform normals correctly. Indeed, normals are not transformed at all. One of the disadvantages is that it requires computing an inverse matrix for our light position, so that we can go from world space to model space.

The Fragment Point Lighting tutorial shows off how fragment lighting works.

Figure 10.7. Fragment Point Lighting



Much better.

This tutorial is controlled as before, with a few exceptions. Pressing the **t** key will toggle a scale factor onto to be applied to the cylinder, and pressing the **h** key will toggle between per-fragment lighting and per-vertex lighting.

The rendering code has changed somewhat, considering the use of model space for lighting instead of camera space. The start of the rendering looks as follows:

Example 10.3. Initial Per-Fragment Rendering

```
glutil::MatrixStack modelMatrix;
modelMatrix.SetMatrix(g_viewPole.CalcMatrix());
```

```
const glm::vec4 &worldLightPos = CalcLightPosition();

glm::vec4 lightPosCameraSpace = modelMatrix.Top() * worldLightPos;
```

The new code is the last line, where we transform the world-space light into camera space. This is done to make the math much easier. Since our matrix stack is building up the transform from model to camera space, the inverse of this matrix would be a transform from camera space to model space. So we need to put our light position into camera space before we transform it by the inverse.

After doing that, it uses a variable to switch between per-vertex and per-fragment lighting. This just selects which shaders to use; both sets of shaders take the same uniform values, even though they use them in different program stages.

The ground plane is rendered with this code:

Example 10.4. Ground Plane Per-Fragment Rendering

```
glutil::PushStack push(modelMatrix);

glUseProgram(pWhiteProgram->theProgram);
glUniformMatrix4fv(pWhiteProgram->modelToCameraMatrixUnif, 1, GL_FALSE,
glm::value_ptr(modelMatrix.Top()));

glm::mat4 invTransform = glm::inverse(modelMatrix.Top());
glm::vec4 lightPosModelSpace = invTransform * lightPosCameraSpace;
glUniform3fv(pWhiteProgram->modelSpaceLightPosUnif, 1, glm::value_ptr(lightPosModelSpace));

g_pPlaneMesh->Render();
glUseProgram(0);
```

We compute the inverse matrix using `glm::inverse` and store it. Then we use that to compute the model space light position and pass that to the shader. Then the plane is rendered.

The cylinder is rendered using similar code. It simply does a few transformations to the model matrix before computing the inverse and rendering.

The shaders are where the real action is. As with previous lighting tutorials, there are two sets of shaders: one that take a per-vertex color, and one that uses a constant white color. The vertex shaders that do per-vertex lighting computations should be familiar:

Example 10.5. Model Space Per-Vertex Lighting Vertex Shader

```
#version 330

layout(location = 0) in vec3 position;
layout(location = 1) in vec4 inDiffuseColor;
layout(location = 2) in vec3 normal;

out vec4 interpColor;

uniform vec3 modelSpaceLightPos;
uniform vec4 lightIntensity;
uniform vec4 ambientIntensity;

uniform mat4 modelToCameraMatrix;
uniform mat3 normalModelToCameraMatrix;

uniform Projection
{
    mat4 cameraToClipMatrix;
};
```



```
void main()
{
    gl_Position = cameraToClipMatrix * (modelToCameraMatrix * vec4(position, 1.0));

    vec3 dirToLight = normalize(modelSpaceLightPos - position);

    float cosAngIncidence = dot( normal, dirToLight);
    cosAngIncidence = clamp(cosAngIncidence, 0, 1);

    interpColor = (lightIntensity * cosAngIncidence * inDiffuseColor) +
        (ambientIntensity * inDiffuseColor);
}
```

The main differences between this version and the previous version are simply what one would expect from the change from camera-space lighting to model space lighting. The per-vertex inputs are used directly, rather than being transformed into camera space. There is a second version that omits the `inDiffuseColor` input.

With per-vertex lighting, we have two vertex shaders: `ModelPosVertexLighting_PCN.vert` and `ModelPosVertexLighting_PN.vert`. With per-fragment lighting, we also have two shaders: `FragmentLighting_PCN.vert` and `FragmentLighting_PN.vert`. They are disappointingly simple:

Example 10.6. Model Space Per-Fragment Lighting Vertex Shader

```
#version 330

layout(location = 0) in vec3 position;
layout(location = 1) in vec4 inDiffuseColor;
layout(location = 2) in vec3 normal;

out vec4 diffuseColor;
out vec3 vertexNormal;
out vec3 modelSpacePosition;

uniform mat4 modelToCameraMatrix;

uniform Projection
{
    mat4 cameraToClipMatrix;
};

void main()
{
    gl_Position = cameraToClipMatrix * (modelToCameraMatrix * vec4(position, 1.0));

    modelSpacePosition = position;
    vertexNormal = normal;
    diffuseColor = inDiffuseColor;
}
```

Since our lighting is done in the fragment shader, there is not much to do except pass variables through and set the output clip-space position. The version that takes no diffuse color just passes a vec4 containing just 1.0.

The fragment shader is much more interesting:

Example 10.7. Per-Fragment Lighting Fragment Shader

```
#version 330
```

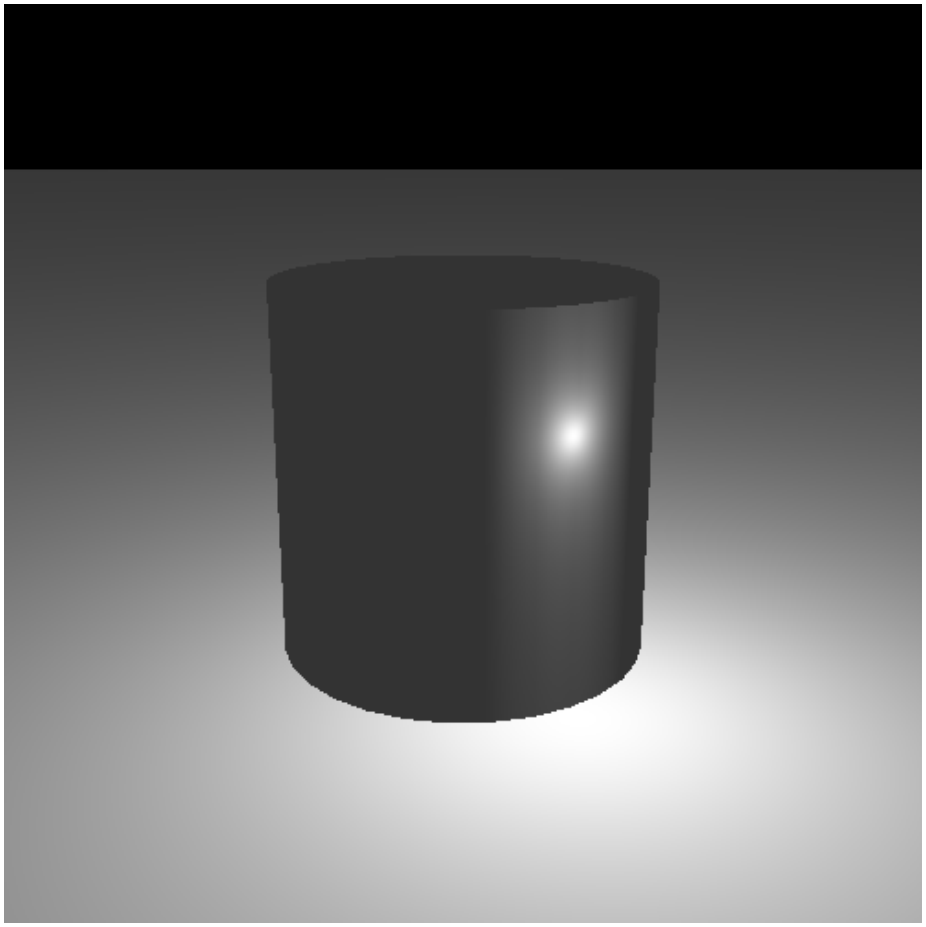
```
in vec4 diffuseColor;
in vec3 vertexNormal;
in vec3 modelSpacePosition;

out vec4 outputColor;

uniform vec3 modelSpaceLightPos;

uniform vec4 lightIntensity;
uniform vec4 ambientIntensity;

void main()
{
    vec3 lightDir = normalize(modelSpaceLightPos - modelSpacePosition);

    float cosAngIncidence = dot(normalize(vertexNormal), lightDir);
    cosAngIncidence = clamp(cosAngIncidence, 0, 1);

    outputColor = (diffuseColor * lightIntensity * cosAngIncidence) +
        (diffuseColor * ambientIntensity);
}
```

The math is essentially identical between the per-vertex and per-fragment case. The main difference is the normalization of `vertexNormal`. This is necessary because interpolating between two unit vectors does not mean you will get a unit vector after interpolation. Indeed, interpolating the 3 components guarantees that you will not get a unit vector.

Gradient Matters

While this may look perfect, there is still one problem. Use the **Shift**+**J** key to move the light really close to the cylinder, but without putting the light inside the cylinder. You should see something like this:

Figure 10.8. Close Lit Cylinder



Notice the vertical bands on the cylinder. This are reminiscent of the same interpolation problem we had before. Was not doing lighting at the fragment level supposed to fix this?

It is similar to the original problem, but technically different. Per-vertex lighting caused lines because of color interpolation artifacts. This is caused by an optical illusion created by adjacent linear gradients.

The normal is being interpolated linearly across the surface. This also means that the lighting is changing somewhat linearly across the surface. While the lighting isn't a linear change, it can be approximated as one over a small area of the surface.

The edge between two triangles changes how the light interacts. On one side, the nearly-linear gradient has one slope, and on the other side, it has a different one. That is, the rate at which the gradients change abruptly changes.

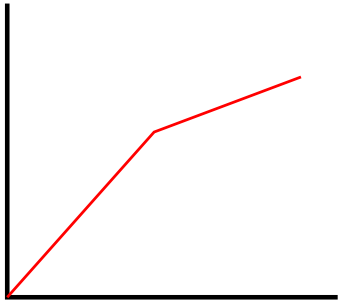
Here is a simple demonstration of this:

Figure 10.9. Adjacent Gradient

These are two adjacent linear gradients, from the bottom left corner to the top right. The color value increases in intensity as it goes from the bottom left to the top right. They meet along the diagonal in the middle. Both gradients have the same color value in the middle, yet it appears that there is a line down the center that is brighter than the colors on both sides. But it is not; the color on the right side of the diagonal is actually brighter than the diagonal itself.

That is the optical illusion. Here is a diagram that shows the color intensity as it moves across the above gradient:

Figure 10.10. Gradient Intensity Plot



The color curve is continuous; there are no breaks or sudden jumps. But it is not a smooth curve; there is a sharp edge.

It turns out that human vision really wants to find sharp edges in smooth gradients. Anytime we see a sharp edge, our brains try to turn that into some kind of shape. And if there is a shape to the gradient intersection, such as a line, we tend to see that intersection “pop” out at us.

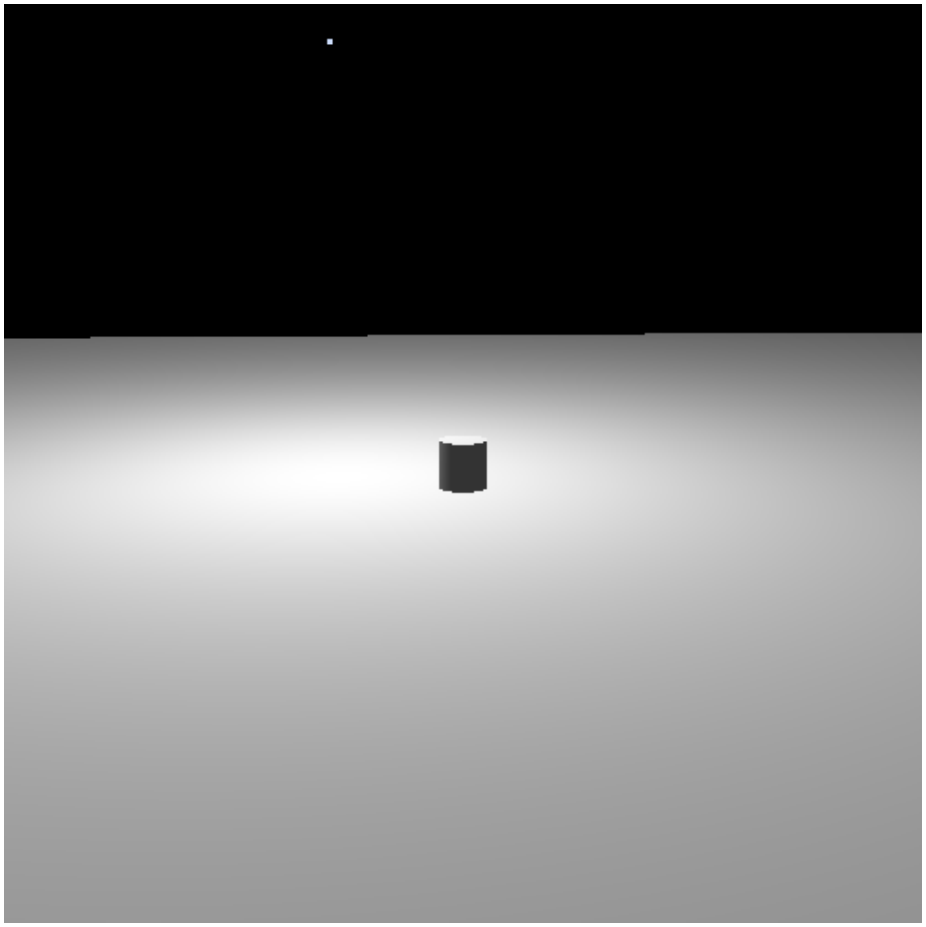
The solution to this problem is not yet available to us. One of the reasons we can see this so clearly is that the surface has a very regular diffuse reflectance (ie: color). If the surface color was irregular, if it changed at most every fragment, then the effect would be virtually impossible to notice.

But the real source of the problem is that the normal is being linearly interpolated. While this is certainly much better than interpolating the per-vertex lighting output, it does not produce a normal that matches with the normal of a perfect cylinder. The correct solution, which we will get to eventually, is to provide a way to encode the normal for a surface at many points, rather than simply interpolating vertex normals.

Distant Points of Light

There is another issue with our current example. Use the **i** key to raise the light up really high. Notice how bright all of the upwardly-facing surfaces get:

Figure 10.11. High Light



You probably have no experience with this in real life. Holding a light farther from the surface in reality does not make the light brighter. So obviously something is happening in reality that our simple lighting model is not accounting for.

In reality, lights emit a certain quantity of light per unit time. For a point-like light such as a light bulb, it emits this light radially, in all directions. The farther from the light source one gets, the more area that this must ultimately cover.

Light is essentially a wave. The farther away from the source of the wave, the less intense the wave is. For light, this is called *light attenuation*.

Our model does not include light attenuation, so let's fix that.

Attenuation is a well-understood physical phenomenon. In the absence of other factors (atmospheric light scattering, etc), the light intensity varies with the inverse of the square of the distance. An object 2 units away from the light feels the light with one-fourth the intensity. So our equation for light attenuation is as follows:

Equation 10.1. Physical Light Attenuation

$$\text{Attenuated Light} = \frac{I}{(1.0 + k * r^2)}$$

There is a constant in the equation, which is used for unit correction. Of course, we can (and will) use it as a fudge factor to make things look right.

The constant can take on a physical meaning. The constant can mean the distance at which half of the light intensity is lost. To compute such a constant, for a half-light distance of r_λ, use this equation:

$$k = \frac{1}{r_\#^2}$$

This equation computes physically realistic light attenuation for point-lights. But it often does not look very good. The equation tends to create a sharper intensity falloff than one would expect.

There is a reason for this, but it is not one we are ready to get into quite yet. What is often done is to simply use the inverse rather than the inverse-square of the distance:

Equation 10.2. Light Attenuation Inverse

$$\text{Attenuated Light} = \frac{I}{(1.0 + k * r)}$$

It looks brighter at greater distances than the physically correct model. This is fine for simple examples, but as we get more advanced, it will not be acceptable. This solution is really just a stop-gap; the real solution is one that we will discuss in a few tutorials.

Reverse of the Transform

However, there is a problem. We previously did per-fragment lighting in model space. And while this is a perfectly useful space to do lighting in, model space is not world space.

We want to specify the attenuation constant factor in terms of world space distances. But we are not dealing in world space; we are in model space. And model space distances are, naturally, in model space, which may well be scaled relative to world space. Here, any kind of scale in the model-to-world transform is a problem, not just non-uniform scales. Although if there was a uniform scale, we could apply theoretically apply the scale to the attenuation constant.

So now we cannot use model space. Fortunately, camera space is a space that has the same scale as world space, just with a rotation/translation applied to it. So we can do our lighting in that space.

Doing it in camera space requires computing a camera space position and passing it to the fragment shader to be interpolated. And while we could do this, that's not clever enough. Is not there some way to get around that?

Yes, there is. Recall `gl_FragCoord`, an intrinsic value given to every fragment shader. It represents the location of the fragment in window space. So instead of transforming from model space to camera space, we will transform from window space to camera space.

Note

The use of this reverse-transformation technique here should not be taken as a suggestion to use it in all, or even most cases like this. In all likelihood, it will be much slower than just passing the camera space position to the fragment shader. It is here primarily for demonstration purposes, and because it will be useful in the future.

The sequence of transformations that take a position from camera space to window space is as follows:

Table 10.1. Transform Legend

Field Name	Meaning
M	The camera-to-clip transformation matrix.
P_{camera}	The camera-space vertex position.
C	The clip-space vertex position.
N	The normalized device coordinate position.
V_{xy}	The X and Y values passed to `glViewport`.
V_{wh}	The width and height passed to `glViewport`.
D_{nf}	The depth near and far values passed to `glDepthRange`.

Equation 10.3. Camera to Window Transforms

$$\begin{aligned}
\vec{C} &= \mathbf{M}\vec{P}_{camera} \\
\vec{N} &= \frac{\vec{C}}{C_w} \\
\text{gl_FragCoord.x} &= \frac{V_w}{2}N_x + V_x + \frac{V_w}{2} \\
\text{gl_FragCoord.y} &= \frac{V_h}{2}N_y + V_y + \frac{V_h}{2} \\
\text{gl_FragCoord.z} &= \frac{D_f - D_n}{2}N_z + \frac{D_f + D_n}{2} \\
\text{gl_FragCoord.w} &= \frac{1}{C_w}
\end{aligned}$$

Therefore, given `gl_FragCoord`, we will need to perform the reverse of these:

Equation 10.4. Window to Camera Transforms

$$\begin{aligned}
N_x &= \frac{2 * \text{gl_FragCoord.x}}{V_w} - \frac{2V_x}{V_w} - 1 \\
N_y &= \frac{2 * \text{gl_FragCoord.y}}{V_h} - \frac{2V_y}{V_h} - 1 \\
N_y &= \frac{2 * \text{gl_FragCoord.z} - D_f - D_n}{D_f - D_n} \\
\vec{C}_{xyz} &= \frac{\vec{N}}{\text{gl_FragCoord.w}} \\
C_w &= \frac{1}{\text{gl_FragCoord.w}} \\
\vec{P}_{camera} &= \mathbf{M}^{-1}\vec{C}
\end{aligned}$$

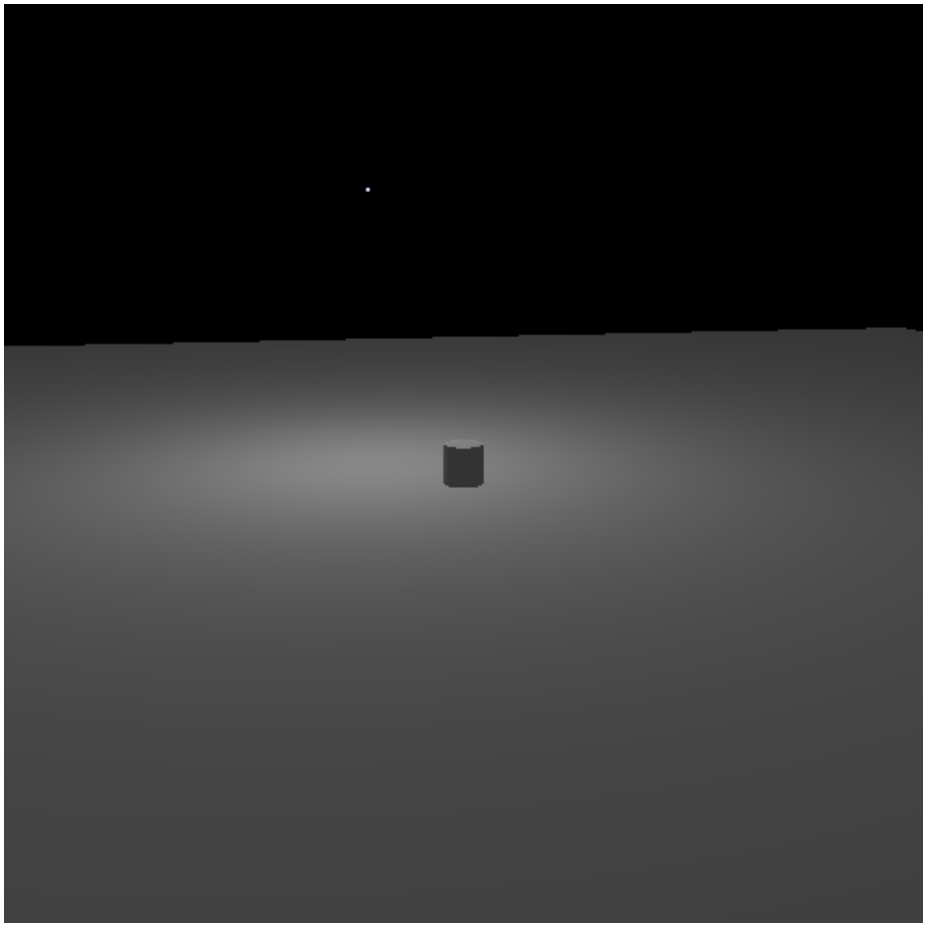
In order for our fragment shader to perform this transformation, it must be given the following values:

- The inverse projection matrix.
- The viewport width/height.
- The depth range.

Applied Attenuation

The Fragment Attenuation tutorial performs per-fragment attenuation, both with linear and quadratic attenuation.

Figure 10.12. Fragment Attenuation



This tutorial controls as before, with the following exceptions. The **O** and **U** keys increase and decrease the attenuation constant. However, remember that decreasing the constant makes the attenuation less, which makes the light appear *brighter* at a particular distance. Using the shift key in combination with them will increase/decrease the attenuation by smaller increments. The **H** key swaps between the linear and quadratic interpolation functions.

The drawing code is mostly the same as we saw in the per-vertex point light tutorial, since both this and that one perform lighting in camera space. The vertex shader is also nothing new; passes the vertex normal and color to the fragment shader. The vertex normal is multiplied by the normal matrix, which allows us to use non-uniform scaling.

New Uniform Types

The more interesting part is the fragment shader. The definitions are not much changed from the last one, but there have been some additions:

Example 10.8. Light Attenuation Fragment Shader Definitions

```
uniform float lightAttenuation;
uniform bool bUseRSquare;

uniform UnProjection
{
    mat4 clipToCameraMatrix;
    ivec2 windowSize;
};
```

The `lightAttenuation` uniform is just a float, but `bUseRSquare` uses a new type: boolean.

GLSL has the bool type just like C++ does. The `true` and `false` values work just like C++'s equivalents. Where they differ is that GLSL also has vectors of bools, called bvec#, where the # can be 2, 3, or 4. We do not use that here, but it is important to note.

OpenGL's API, however, is still a C API. And C (at least, pre-C99) has no bool type. Uploading a boolean value to a shader looks like this:

```
glUniform1i(g_FragWhiteDiffuseColor.bUseRSquareUnif, g_bUseRSquare ? 1 : 0);
```

The integer form of uniform uploading is used, but the floating-point form could be allowed as well. The number 0 represents false, and any other number is true.

The `UnProjection` uniform block contains data that only changes when the window changes. This uniform block is updated along with the vertex shader's `Projection` block. This data is used to perform the previously-discussed reverse-transformation operation, so that we can turn `gl_FragCoord` into a camera-space position.

Notice that the `windowSize` uses a new type: ivec2. This is a 2-dimensional vector of integers.

Functions in GLSL

For the first time, we have a shader complex enough that splitting it into different functions makes sense. So we do that. The first function is one that computes the camera-space position:

Example 10.9. Window to Camera Space Function

```
vec3 CalcCameraSpacePosition()
{
    vec4 ndcPos;
    ndcPos.xy = ((gl_FragCoord.xy / windowSize.xy) * 2.0) - 1.0;
    ndcPos.z = (2.0 * gl_FragCoord.z - gl_DepthRange.near - gl_DepthRange.far) /
        (gl_DepthRange.far - gl_DepthRange.near);
    ndcPos.w = 1.0;

    vec4 clipPos = ndcPos / gl_FragCoord.w;

    return vec3(clipToCameraMatrix * clipPos);
}
```

Not unsurprisingly, GLSL functions are defined much like C and C++ functions.

The first three lines compute the position in normalized device coordinates. Notice that the computation of the X and Y coordinates is simplified from the original function. This is because our viewport always sets the lower-left position of the viewport to (0, 0). This is what you get when you plug zeros into that equation.

The `gl_DepthRange` variable is a special uniform defined by GLSL for fragment shaders. As the name suggests, it properly mirrors the values passed to `glDepthRange`; this way, we do not have to put it in our uniform block.

After the transformation to NDC space, we compute the clip-space position as previously shown. Then the result is multiplied through the clip-to-camera matrix, and that vector is returned to the caller.

This is a simple function that uses only uniforms to compute a value. It takes no arguments. The second function is not quite as simple.

Example 10.10. Light Intensity Application Function

```
vec4 ApplyLightIntensity(in vec3 cameraSpacePosition, out vec3 lightDirection)
{
    vec3 lightDifference =  cameraSpaceLightPos - cameraSpacePosition;
    float lightDistanceSqr = dot(lightDifference, lightDifference);
    lightDirection = lightDifference * inversesqrt(lightDistanceSqr);

    float distFactor = bUseRSquare ? lightDistanceSqr : sqrt(lightDistanceSqr);

    return lightIntensity * (1 / ( 1.0 + lightAttenuation * distFactor));
}
```

The function header looks rather different from the standard C/C++ function definition syntax. Parameters to GLSL functions are designated as being inputs, outputs, or inputs and outputs.

Parameters designated with `in` are input parameters. Functions can change these values, but they will have no effect on the variable or expression used in the function call. This is much like the default in C/C++, where parameter changes are local. Naturally, this is the default with GLSL parameters if you do not specify a qualifier.

Parameters designated with `out` can be written to, and its value will be returned to the calling function. These are similar to non-const reference parameter types in C++. And just as with reference parameters, the caller of a function must call it with a real variable (called an "l-value"). And this variable must be a variable that can be *changed*, so you cannot pass a uniform or shader stage input value as this parameter.

However, the initial value of parameters declared as outputs is *not* initialized from the calling function. This means that the initial value is uninitialized and therefore undefined (ie: it could be anything). Because of this, you can pass shader stage outputs as `out` parameters. Recall that shader stage output variables can be written to, but *never* read from.

Parameters designated as `inout` will have its value initialized by the caller and have the final value returned to the caller. These are exactly like non-const reference parameters in C++. The main difference is that the value is initialized with the one that the user passed in, which forbids the passing of shader stage outputs as `inout` parameters.

This particular function is semi-complex, as an optimization. Previously, our functions simply normalized the difference between the vertex position and the light position. In computing the attenuation, we need the distance between the two. And the process of normalization computes the distance. So instead of calling the GLSL function to normalize the direction, we do it ourselves, so that the distance is not computed twice (once in the GLSL function and once for us).

The second line performs a dot product with the same vector. Remember that the dot product between two vectors is the cosine of the angle between them, multiplied by each of the lengths of the vectors. Well, the angle between a vector and itself is zero, and the cosine of zero is always one. So what you get is just the length of the two vectors times one another. And since the vectors are the same, the lengths are the same. Thus, the dot product of a vector with itself is the square of its length.

To normalize a vector, we must divide the vector by it's length. And the length of `lightDifference` is the square root of `lightDistanceSqr`. The `inversesqrt` computes 1 / the square root of the given value, so all we need to do is multiply this with the `lightDifference` to get the light direction as a normalized vector. This value is written to our output variable.

The next line computes our lighting term. Notice the use of the ?: operator. This works just like in C/C++. If we are using the square of the distance, that's what we store. Otherwise we get the square-root and store that.

Note

The assumption in using ?: here is that only one or the other of the two expressions will be evaluated. That's why the expensive call to `sqrt` is done here. However, this may not be the case. It is entirely possible (and quite likely) that the shader will always evaluate *both* expressions and simply store one value or the other as needed. So do not rely on such conditional logic to save performance.

After that, things proceed as expected.

Making these separate functions makes the main function look almost identical to prior versions:

Example 10.11. Main Light Attenuation

```
void main()
{
    vec3 cameraSpacePosition = CalcCameraSpacePosition();

    vec3 lightDir = vec3(0.0);
    vec4 attenIntensity = ApplyLightIntensity(cameraSpacePosition, lightDir);

    float cosAngIncidence = dot(normalize(vertexNormal), lightDir);
    cosAngIncidence = clamp(cosAngIncidence, 0, 1);

    outputColor = (diffuseColor * attenIntensity * cosAngIncidence) +
        (diffuseColor * ambientIntensity);
}
```

Function calls appear very similar to C/C++, with the exceptions about parameters noted before. The camera-space position is determined. Then the light intensity, modified by attenuation, is computed. From there, things proceed as before.

Alternative Attenuation

As nice as these somewhat-realistic attenuation schemes are, it is often useful to model light attenuation in a very different way. This is in no way physically accurate, but it can look reasonably good.

We simply do linear interpolation based on the distance. When the distance is 0, the light has full intensity. When the distance is beyond a given distance, the maximum light range (which varies per-light), the intensity is 0.

Note that "reasonably good" depends on your needs. The closer you get in other ways to providing physically accurate lighting, the closer you get to photorealism, the less you can rely on less accurate phenomena. It does no good to implement a complicated sub-surface scattering lighting model that includes Fresnel factors and so forth, while simultaneously using a simple interpolation lighting attenuation model.

In Review

In this tutorial, you have learned the following:

- Point lights are lights that have a position within the world, radiating light equally in all directions. The light direction at a particular point on the surface must be computed using the position at that point and the position of the light.
- Attempting to perform per-vertex lighting computations with point lights leads to artifacts.
- Lighting can be computed per-fragment by passing the fragment's position in an appropriate space.
- Lighting can be computed in model space.
- Point lights have a falloff with distance, called attenuation. Not performing this can cause odd effects, where a light appears to be brighter when it moves farther from a surface. Light attenuation varies with the inverse of the square of the distance, but other attenuation models can be used.
- Fragment shaders can compute the camera space position of the fragment in question by using `gl_FragCoord` and a few uniform variables holding information about the camera to window space transform.
- GLSL can have integer vectors, boolean values, and functions.

Further Study

Try doing these things with the given programs.

- When we used model space-based lighting computations, we had to perform an inverse on our matrix from the matrix stack to transform the light position from camera space to model space. However, it would be entirely possible to simply build an inverse matrix at the same time we build a regular matrix on our matrix stack. The inverse of a rotation matrix is just the rotation matrix with a negated angle; the inverse of a scale is just the multiplicative inverse of the scales, and the inverse of the translation is the negation of the translation vector.

  To do this, you will need to modify the `MatrixStack` class in a number of ways. It must store a second matrix representing the accumulated inverse matrix. When a transformation command is given to the stack, it must also generate the inverse matrix for this transform and *left multiply* this into the accumulated inverse. The push/pop will have to push/pop the inverse matrix as well. It can use the same stack, so long as the pop function puts the two matrices in the proper places.

- Implement the alternative attenuation described at the end of the section on attenuation.

GLSL Features of Note

gl_DepthRange — A built-in OpenGL uniform defined for fragment shaders only. This uniform stores the parameters passed to `glDepthRange.` When those parameters change, all programs are automatically updated.

```
vec inversesqrt(vec x);
```

This function computes 1 / the square root of `x`. This is a component-wise computation, so vectors may be used. The return value will have the same type as `x`.

```
vec sqrt(vec x);
```

This function computes the square root of `x`. This is a component-wise computation, so vectors may be used. The return value will have the same type as `x`.

Glossary

point light source — A light source that emits light from a particular location in the world. The light is emitted in all directions evenly.

fragment lighting — Evaluating the lighting equation at every fragment.

This is also called Phong shading, in contrast with Goroud shading, but this name has fallen out of favor due to similarities with names for other lighting models.

light attenuation — The decrease of the intensity of light with distance from the source of that light.

Chapter 11. 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.

One way to show that an object is shiny is to model *specular highlights.* A specular highlight is a bright highlight on an object caused by direct illumination from a light source. The position of the highlight changes with the view direction as well as the light direction.

Modelling true specular reflection would require reflecting all light from objects in the scene, whether direct or indirect. However for many objects, like shiny plastics and the like, indirect specular reflections are very weak. Thus, by modeling direct specular reflections, we can make an object appear shiny without having to do too much work.

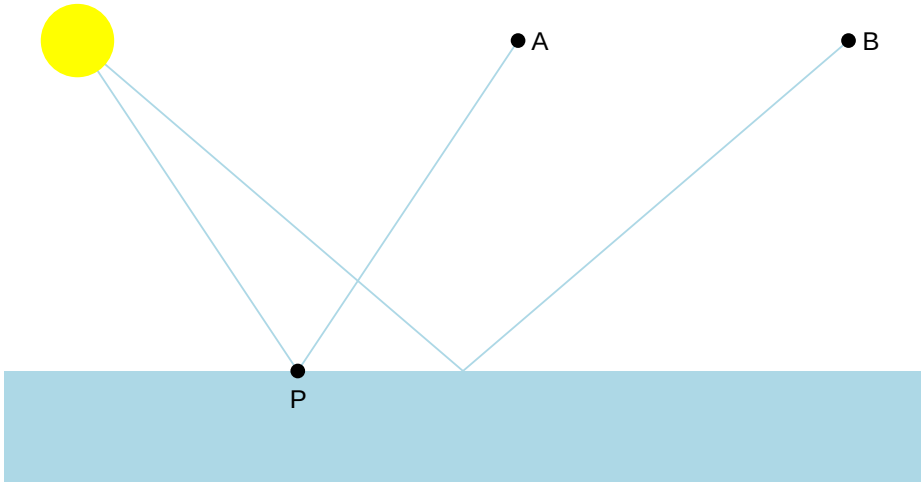
We will look at several models for specular highlights and reflection. The Lambertian diffuse reflectance model was reasonably good for modelling diffuse lighting, but there are several models for specular reflection that should be considered. They vary in quality and performance.

Note that these models do not throw away diffuse lighting. They all act as supplements, adding their contribution into the overall result for the lighting equation.

Microfacets

All of these specular reflection models work based on an assumption about the characteristics of the surface. If a surface was perfectly smooth, then the specular highlight from a point light would be infinitely small (since point lights themselves are infinitely small).

Figure 11.1. Perfect Specular Reflection



Notice that the intensity of the reflected light depends not only on the angle of incidence but also on the direction to the viewer. This is called the *angle of view* or *viewing angle*. Viewing position A detects the light specularly reflected from the surface at the point P, but the viewing position B does not.

Surfaces however are rarely perfect specular reflectors (mirrors are the most common perfect reflectors). Surfaces that seem smooth from far away can be rough on closer examination. This is true at the microscopic level as well, even for surfaces that appear quite smooth. This roughness can be modelled by assuming that a surface is composed of a number of *microfacets*.

A microfacet is a flat plane that is oriented in a single direction. Each microfacet reflects light perfectly in that direction. Surfaces with microfacets would look like this:

Figure 11.2. Smooth and Rough Microfacets

It is part of the microfacet model's assumption that many microfacets on a surface will contribute to the light returned under a single pixel of the final image. So each pixel in the rendered image is the result of an aggregate of the microfacets that lie under the area of that pixel on the surface.

The average normal of the microfacets is the surface normal at that point. The relative smoothness of a surface can therefore be modeled as a statistical distribution of the orientation of microfacets on the surface. A smooth surface has a distribution close to the average, while a rough surface has a broader distribution.

Thus, a model of specular reflections includes a term that defines the overall smoothness of the source. This is a surface characteristic, representing the distribution of microfacets using whatever statistical distribution the particular specular model is using. One of the main differences between specular models is the kind of statistical distribution that they use.

Specular highlights are formed because, even though the surface normal may not be oriented to directly reflect light from the light source to the viewer, some microfacets may still be oriented to reflect a portion of that light. A microfacet distribution model determines the proportion of microfacets that happen to be oriented to reflect light towards the viewer.

Smooth surfaces, those who's microfacets do not deviate much from the surface normal, will have a small, bright highlight. Rough surfaces, who's microfacets are oriented in wildly divergent directions, will have a much dimmer, but larger specular highlight. These highlights will have positions and shapes based on the angle of incidence and the angle of view.

Note that specular reflectance models do not become diffuse reflectance models when taken to the extreme case of maximum roughness. Specular reflection represents a different mode of light/surface interaction from diffuse reflection.

Phong Model

The simplest model of specular illumination is the *Phong model*. The distribution of microfacets is not determined by a real statistical distribution. Instead it is determined by... making things up.

On Phong and Nomenclature

The term “Phong shading” was once commonly used to refer to what we now know as per-fragment (or per-pixel) lighting. That is, evaluating the lighting equation at every fragment over a surface. This term should not be confused with the Phong specular lighting model. Because of this, the term “Phong shading” has fallen out of common usage.

The Phong model is not really based on anything real. It does not deal in microfacet distributions at all. What the Phong model is is something that looks decent enough and is cheap to compute. It approximates a statistical distribution of microfacets, but it is not really based on anything real.

The Phong model states that the light reflected in the direction of the viewer varies based on the angle between difference between the view direction and the direction of perfect reflection. Mathematically, the Phong model looks like this:

Equation 11.1. Phong Specular Term

$$\text{Phong term} = (\hat{V} \cdot \hat{R})^s$$

The Phong term is multiplied by the light intensity in the lighting equation.

The brightness of the specular highlight for a particular viewing direction is based on raising the cosine of the angle between the view direction and the reflection direction to a power. As previously stated, this model is not based on anything real. It simply creates a bright somewhat-circular area on the surface. This area gets dimmer as the viewer is farther from the direction of perfect reflection.

The `s` term in the equation represents the roughness of the surface. A smooth surface, which should have a smaller highlight, has a large `s`. Since the cosine of the angle is a number on [0, 1], taking it to a power greater than 1.0 will make the number smaller. Therefore, a large `s` exponent will make for a small highlight.

The specular exponent can range from $(0, \infty)$. A small exponent makes for a rougher appearance, while a large exponent suggests a shiny surface.

Specular Absorption

The Phong term computed above is then multiplied with the light intensity. This represents the maximum light reflected along the view direction.

However, just as with diffuse lighting, surfaces can absorb some quantity of the light that would be specularly reflected. We could use the diffuse color here, multiplying it by the specular term. But this would not be physically correct for many kinds of objects.

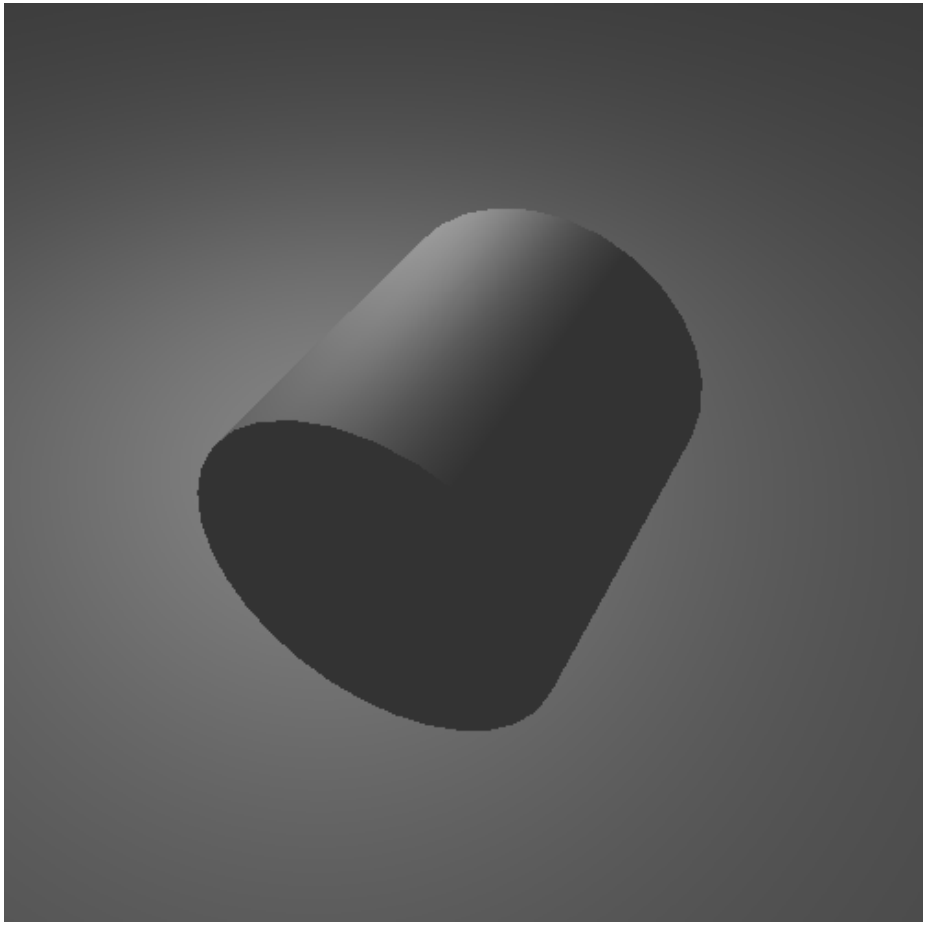
Many surfaces, particularly certain man-made pigments and plastics, have multiple layers to them. The top layer will specularly reflect some portion of the light. However, it will also let some portion of that light reach lower layers. These layers have stronger diffuse reflectance. So the specular absorption on the surface has different characteristics than the diffuse absorption in the lower layers. Usually, the specular layer reflects equally on all wavelenths, so the specular highlight tends to be the color of the light itself.

Notably, metals do not do this. Their diffuse absorption tends to be the same as their specular absorption. So while blue plastic under white light has a white specular highlight, gold metal under white light has a gold highlight.

Drawing Phong

The Phong Lighting tutorial demonstrates the Phong specular model.

Figure 11.3. Phong Lighting



The tutorial is controlled similarly to previous lighting tutorials. Clicking and dragging with the right mouse button controls the orientation of the cylinder. Pressing the **T** key will swap between the scaled and unscaled cylinder. The **Y** key toggles the drawing of the light source. The **B** key will toggle the light's rotation on/off. Pressing the **Space Bar** toggles between drawing the uncolored cylinder and the colored one.

The light's position is mostly controlled as before, with the **I**,**J**, **K**, and **L** keys. The specular value is controlled by the **U** and **O** keys. They raise and low the specular exponent. Using **Shift** in combination with them will raise/lower the exponent by smaller amounts.

The **G** key toggles between a diffuse color of (1, 1, 1) and a darker diffuse color of (0.2, 0.2, 0.2). This is useful for seeing what the specular would look like on a darker surface color.

The **H** key selects between specular and diffuse, just specular and just diffuse. The ambient term is always used. Pressing **Shift**+**H** will toggle between diffuse only and diffuse+specular.

The rendering code is nothing you have not seen in earlier tutorials. It loads 6 programs, and uses the various controls to select which to use to render.

There are two vertex shaders in use. One that takes the position and normal attributes, and one that takes them plus a per-vertex color. Both of them output the camera-space vertex normal (computed with a normal matrix), the camera-space vertex position, and the diffuse color. In the case of the shader that does not take a per-vertex color, the diffuse color output is taken from a uniform set by the code.

The fragment shaders are more interesting. They do lighting in camera space, so there is no need for the reverse-transform trick we used previously. The shaders also use light attenuation, but it only varies with the inverse of the distance, rather than the inverse squared.

The main portion of the specular+diffuse fragment shader is as follows:

Example 11.1. Phong Lighting Shader

```
vec3 lightDir = vec3(0.0);
float atten = CalcAttenuation(cameraSpacePosition, lightDir);
vec4 attenIntensity = atten * lightIntensity;

vec3 surfaceNormal = normalize(vertexNormal);
float cosAngIncidence = dot(surfaceNormal, lightDir);
cosAngIncidence = clamp(cosAngIncidence, 0, 1);

vec3 viewDirection = normalize(-cameraSpacePosition);
vec3 reflectDir = reflect(-lightDir, surfaceNormal);
float phongTerm = dot(viewDirection, reflectDir);
phongTerm = clamp(phongTerm, 0, 1);
phongTerm = cosAngIncidence != 0.0 ? phongTerm : 0.0;
phongTerm = pow(phongTerm, shininessFactor);

outputColor = (diffuseColor * attenIntensity * cosAngIncidence) +
    (specularColor * attenIntensity * phongTerm) +
    (diffuseColor * ambientIntensity);
```

The initial section of code should be familiar. The Phong specular computations start with computing the direction to the camera. Since we are working in camera space, we know that the camera is at the origin (0, 0, 0). The direction from point A to point B is the normalization of B - A. Since the destination point is at the origin, that becomes simply -A, normalized.

The next line computes the direction of perfect reflection, given the light direction and the surface normal. The function here, `reflect`, is a standard GLSL function used for precisely this purpose. Notice that the function in question requires the that the light direction is the direction *from* the light. Our light direction is the direction to the light. This is why it is negated.

This function is useful, but it is important to know how to compute the reflection direction on your own. Here is the formula:

Equation 11.2. Vector Reflection

$$\vec{I} = -\vec{L}$$
$$\vec{R} = \vec{I} - 2(\vec{N} \cdot \vec{I}) * \vec{N}$$

The L vector is the direction to the light, so negating it produces the vector *from* the light.

From here, the Phong term is computed by taking the dot product of the reflection direction and the view direction, clamping it to 0, 1. The next line, where we use the angle of incidence, is very important. What this line does is prevent us from having a specular term when the surface normal is oriented away from the light. If this line were not here, it would be possible to have specular highlights appear to shine *through* a surface. Which is not particularly realistic.

The GLSL standard function `pow` is next used to raise the Phong term to the power. This function seems generally useful, but it has a large number of limitations. The `pow` function computes X^Y, where X is the first parameter and Y is the second.

This function only works for values of X that are greater than or equal to 0; it returns undefined values (ie: anything) otherwise. Clamping the Phong term ensures this. Also, if X is exactly 0.0, then Y must be strictly greater than zero; undefined values are returned otherwise. These limitations exist to make computing the power function much faster.

And for Phong specular computations, the limitations almost never come into play. The cosine of the angle is clamped, and the specular exponent is not allowed to be zero.

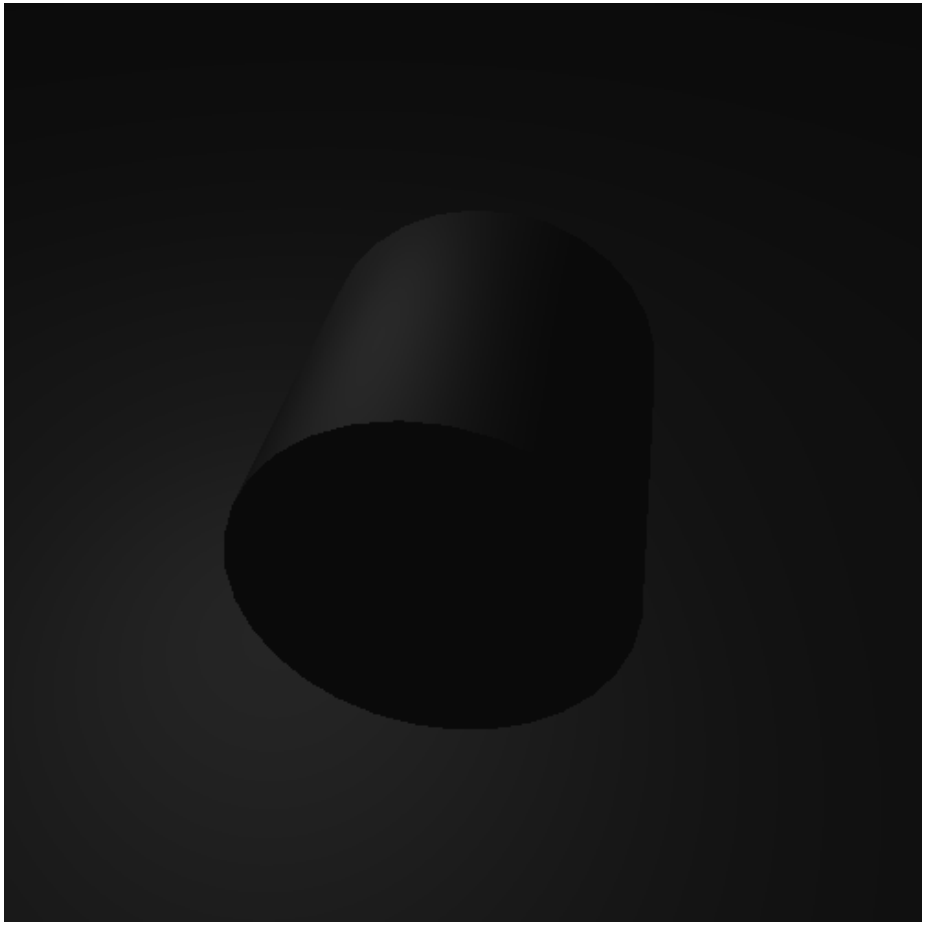
Notice that the specular term is added into the diffuse and ambient terms. This has meaning for the intensity issue we have seen before. If the diffuse and specular colors are too large, and the light attenuation is quite small, then the resulting values from the lighting equations can be larger than 1.0 in magnitude. Since OpenGL automatically clamps the colors to 1.0, this can cause unpleasant effects, where there appears to be a very bright, white area on a surface.

Visual Specular

Having even a weak specular term can make a significant, if subtle, difference. In our case, the specular color of the material is a fairly weak (0.25, 0.25, 0.25). But even with a rough specular highlight, the surface looks more physically reasonable.

In particular, it is interesting to note what happens when you use a very dark diffuse color. You can activate this by pressing the **G** key.

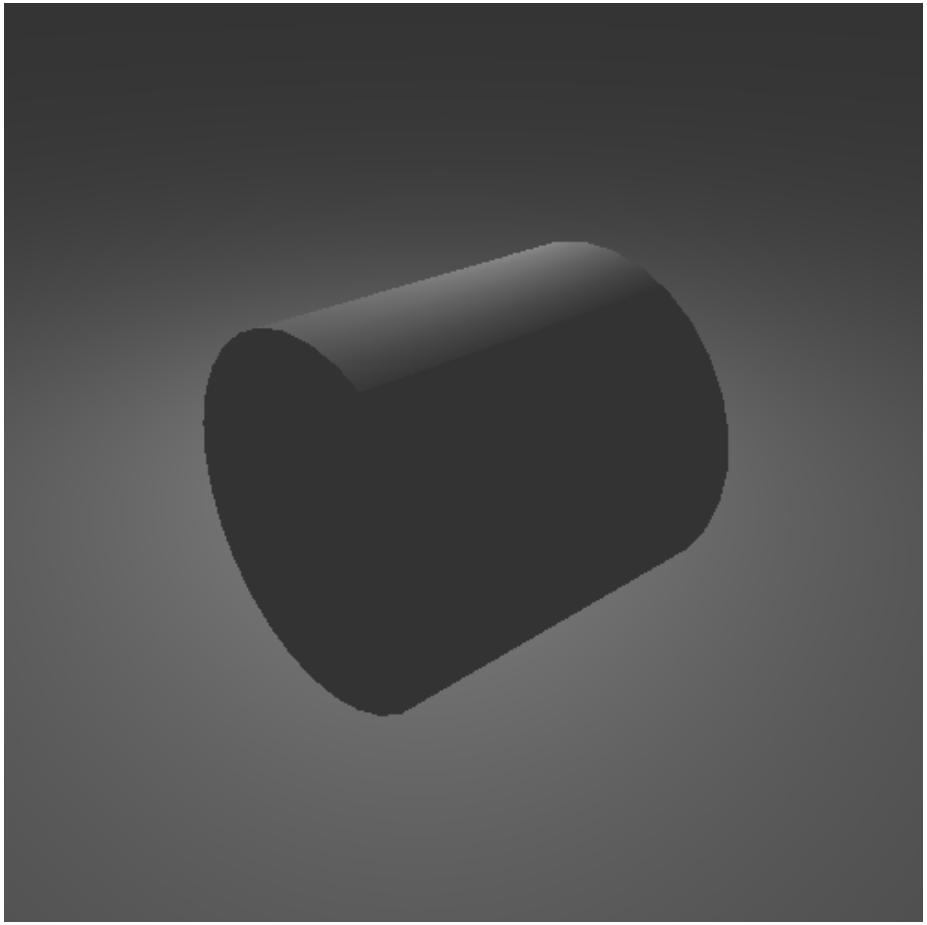
Figure 11.4. Phong with Dark Diffuse



If there was no specular term at all, you would see very little.. The specular highlight, even with the fairly weak specular reflection of 0.25, is strong enough to give some definition to the object when seen from various angles. This more accurately shows what a black plastic object might look like.

One thing you may notice is that, if you bring the light close to the surface, the specular area tends to have very sharp edges.

Figure 11.5. Phong Clipping

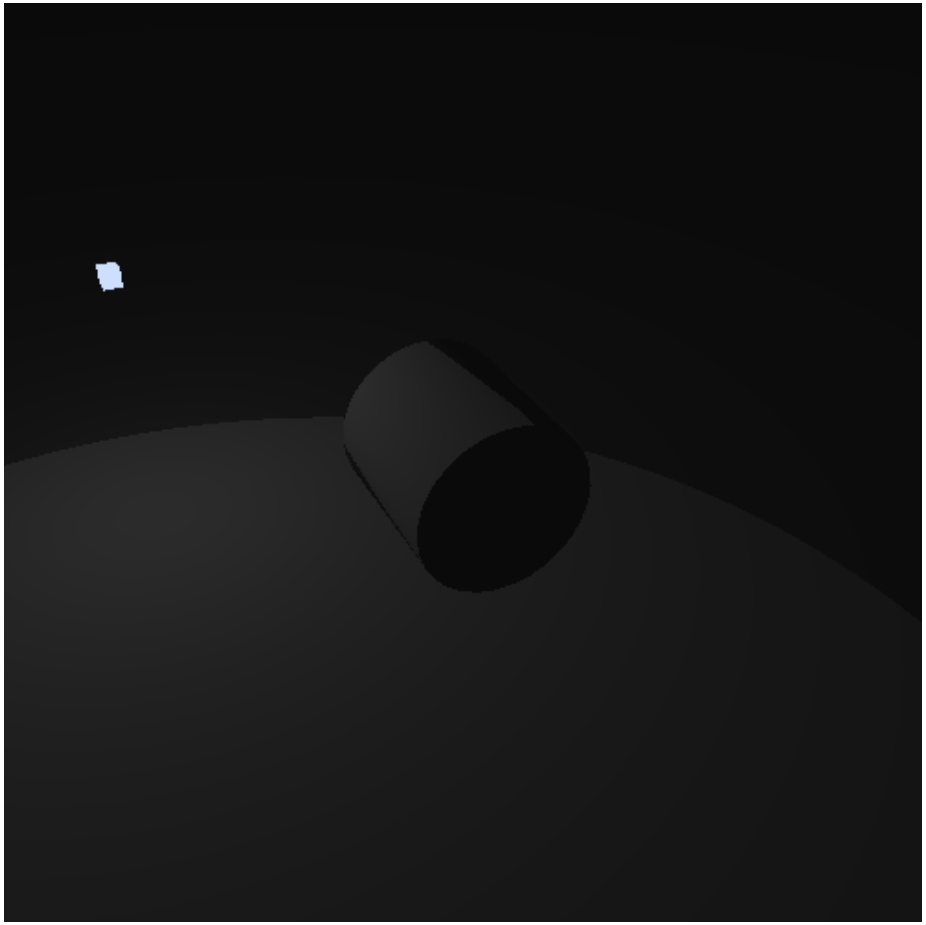


This is part of the nature of specular reflections. If the light is almost perpendicular to the surface, the specular reflection will shine brightest when the light is almost eclipsed by the surface. This creates a strong discontinuity at the point where the light is no longer in view.

You generally see this most with rough surfaces (small exponents). With smoother surfaces, this is rarely seen. But this is not the only visual oddity with Phong and having small exponents.

If you drop the exponent down to the minimum value the code will allow, you will see something like this:

Figure 11.6. Phong Distortion



This ring area shows one of the main limitations of the Phong model. When trying to depict a surface that is rough but still has specular highlights, the Phong model starts to break down. It will not allow any specular contribution from areas outside of a certain region.

This region comes from the angle between the reflection direction and the view direction. This area is the region where the reflection direction and view direction are more than 90 degrees apart.

Under the microfacet model, there is still some chance that some microfacets are oriented towards the camera, even if reflection direction is pointed sharply away. Thus, there should be at least some specular contribution from those areas. The Phong model cannot allow this, due to how it is computed.

What all this tells us is that Phong works best with larger exponents. Small exponents show its problems and limitations.

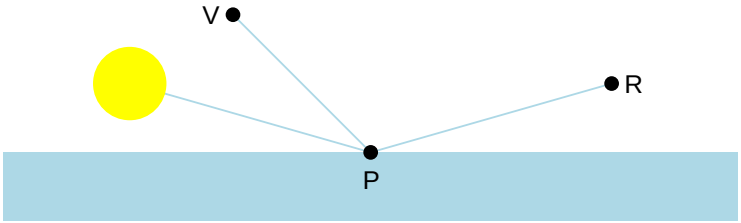
Blinn-Phong Model

The problem with Phong, with regard to the reflection and view directions being greater than 90 degrees, can be solved by changing the computation. This modified model is called the *Blinn-Phong specular model* or just the *Blinn specular model.*

It is no more physically correct than the Phong model. But it does tend to account for more than Phong.

The main problem with Phong is that the angle between the view direction and the reflection direction has to be less than 90 degrees in order for the specular term to be non-zero.

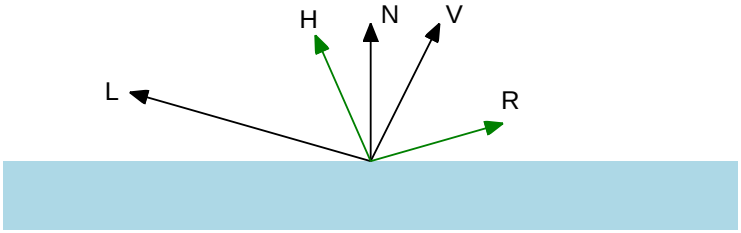
Figure 11.7. Large View and Reflect Angle



The angle between V and R is greater than 90 degrees. Cases like this are not modeled correctly by Phong. There could be microfacets at the point which are oriented towards the camera, but Phong cannot properly model this. The problem is that the dot product between the view direction and reflection direction can be negative, which does not lead to a reasonable result when passed through the rest of the equation.

The Blinn model uses a different set of vectors for its computations, one that are less than 90 degrees in all valid cases. The Blinn model requires computing the *half-angle vector*. The half-angle vector is the direction halfway between the view direction and the light position.

Figure 11.8. Geometric Half-Angle Vector

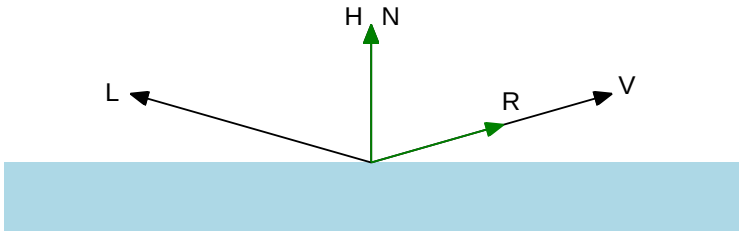


Equation 11.3. Half-Angle Vector

$$\hat{H} = \frac{\hat{L} + \hat{V}}{\left\| \hat{L} + \hat{V} \right\|}$$

When the view direction is perfectly aligned with the reflected direction, the half-angle vector is perfectly aligned with the surface normal. Or to put it another way, the half-angle is the direction the surface normal would need to be facing in order for the viewer to see a specular reflection from the light source.

Figure 11.9. Perfect Reflection Half-Angle Vector



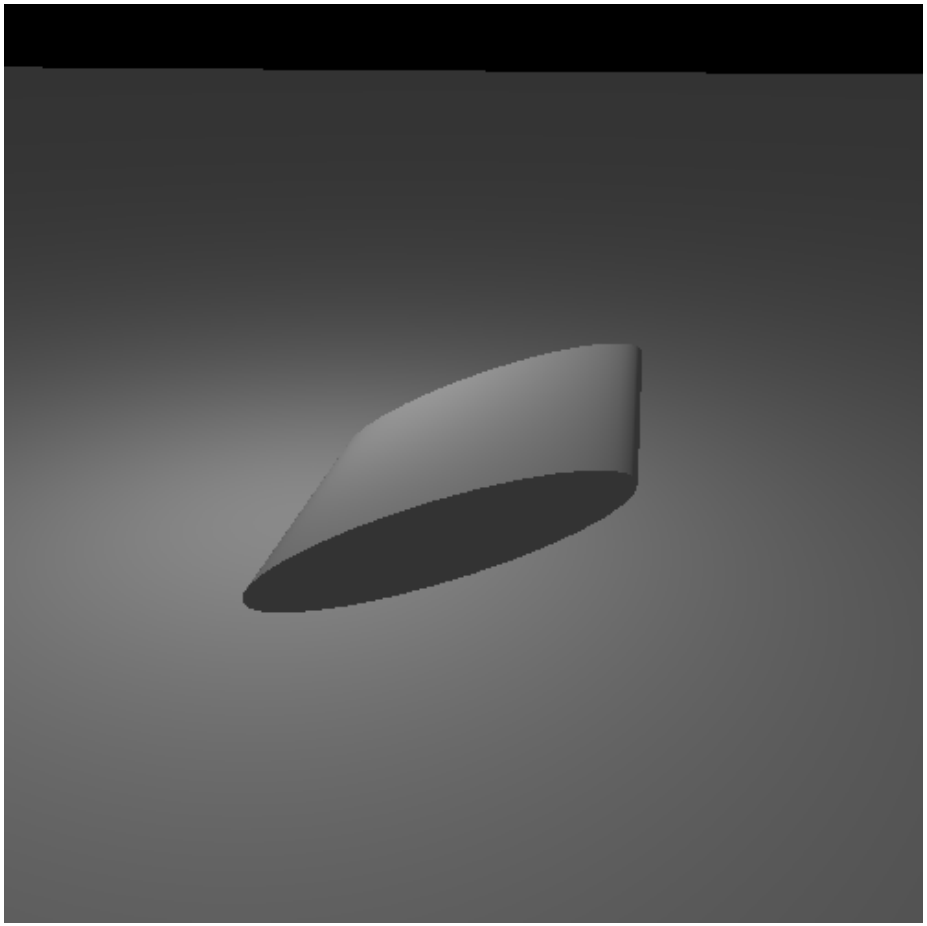
So instead of comparing the reflection vector to the view direction, the Blinn model compares the half-angle vector to the surface normal. It then raises this value to a power representing the shininess of the surface.

Equation 11.4. Blinn Specular Term

$$\text{Blinn term} = \left(\hat{H} \cdot \hat{N}\right)^{s}$$

The angle between the half-angle vector and the normal is always less than 90 degrees. So the Blinn specular model produces similar results to the Phong model, but without some of Phong's problems. This is demonstrated in the Blinn vs Phong Lighting tutorial.

Figure 11.10. Blinn Lighting



The controls are similar to the last tutorial. Pressing the **H** key will switch between Blinn and Phong specular. Pressing **Shift**+**H** will switch between diffuse+specular and specular only. Because the specular exponents have different meanings between the two lighting models, each model has a separate exponent. The keys for changing the exponent values will only change the value for the lighting model currently being viewed.

The real work here is, as before, in the shader computations. Here is the main code for computing the diffuse + Blinn illumination.

Example 11.2. Blinn-Phong Lighting Shader

```
vec3 lightDir = vec3(0.0);
float atten = CalcAttenuation(cameraSpacePosition, lightDir);
vec4 attenIntensity = atten * lightIntensity;
```



```
vec3 surfaceNormal = normalize(vertexNormal);
float cosAngIncidence = dot(surfaceNormal, lightDir);
cosAngIncidence = clamp(cosAngIncidence, 0, 1);

vec3 viewDirection = normalize(-cameraSpacePosition);

vec3 halfAngle = normalize(lightDir + viewDirection);
float blinnTerm = dot(surfaceNormal, halfAngle);
blinnTerm = clamp(blinnTerm, 0, 1);
blinnTerm = cosAngIncidence != 0.0 ? blinnTerm : 0.0;
blinnTerm = pow(blinnTerm, shininessFactor);

outputColor = (diffuseColor * attenIntensity * cosAngIncidence) +
    (specularColor * attenIntensity * blinnTerm) +
    (diffuseColor * ambientIntensity);
```

The half-angle vector is computed by normalizing the sum of the light direction and view direction vectors. As before, we take the dot product between that and the surface normal, clamp, then raise the result to a power.

Blinn specular solves the Phong problem with the reflection direction.

Figure 11.11. Blinn vs. Phong Lighting

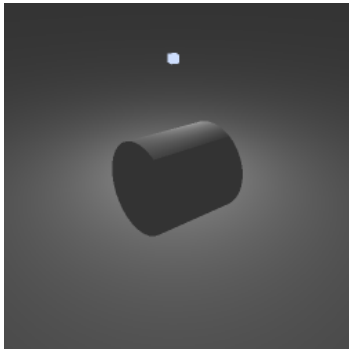
The Blinn version is on the left, with the Phong version on the right.

The Blinn specular exponent does not mean quite the same thing as the Phong exponent. In general, to produce a highlight the same size as a Phong one, you will need a larger Blinn exponent. Play around with the different exponents, to get a feel for what Blinn and Phong can and cannot achieve.

Hard Specular Edge

There are still a few artifacts in the rendering. For example, if you arrange the light, object, and camera as follows, you can see this:

Figure 11.12. Light Edge



The cylinder looks like it has a very sharp corner. What causes this? It is caused by this line in the shader:

```
blinnTerm = cosAngIncidence != 0.0 ? blinnTerm : 0.0;
```

If the angle between the normal and the light direction is greater than 90 degrees, then we force the specular term to zero. The reason behind this is very simple: we assume our surface is a closed object. Given that assumption, if the normal at a location on the surface is facing away from the light, then this could only happen if there is some other part of the surface between itself and the light. Therefore, the surface cannot be directly illuminated by that light.

That is a reasonable assumption, and it certainly makes sense in reality. But real-life objects don't have these kinds of hard specular lines. So what are we missing in our model?

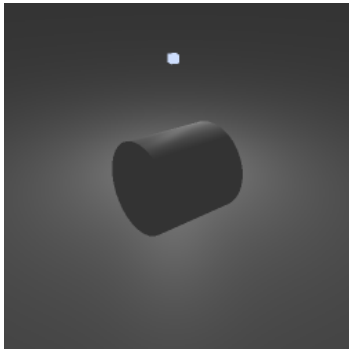
What we are missing is that point lights don't exist in the real world. Light illumination does not come from a single, infinitely small location in space. Even the size of the Sun relative to Earth has a significant area. So what this means is that, for a given point on a surface, it could be in partial view of the light source. Imagine Earth at sunset for an example: part of the sun is below the horizon and part of it is not.

Since only part of the light is visible from that point on the surface, then only part of the light contributes to the overall illumination. So at these places where you might get hard specular boundaries, under more real lighting conditions, you still get a semi-gentle fall-off.

That's all well and good, but modeling true area lights is difficult even for simple cases. A much simpler way to resolve this is to not use such a low specular exponent. This specular exponent is relatively small, leading to a very broad specular highlight. If we restrict our use of a specular term to surfaces who's specular exponent is reasonably large, we can prevent this artifact from appearing.

Here is the same scene, but with a larger exponent:

Figure 11.13. Improved Light Edge



We could also adjust the specular reflectance, so that surfaces with a low specular exponent also have a small specular reflectance.

Gaussian

Phong and Blinn are nice toy heuristics that take relatively little computational power. But if you're truly serious about computing specular highlights, you need a model that actually *models* microfacets.

Real microfacet models are primarily based on the answer to the question “What proportion of the microfacets of this surface are oriented in such a way as to specularly reflect light towards the viewer?” The greater the proportion of properly oriented microfacets, the stronger the reflected light. This question is ultimately one of statistics.

Thus it makes sense to model this as a probability distribution. We know that the average microfacet orientation is the surface normal. So it's just a matter of developing a probability distribution function that says what portion of the surface's microfacets are oriented to provide specular reflections given the light direction and the view direction.

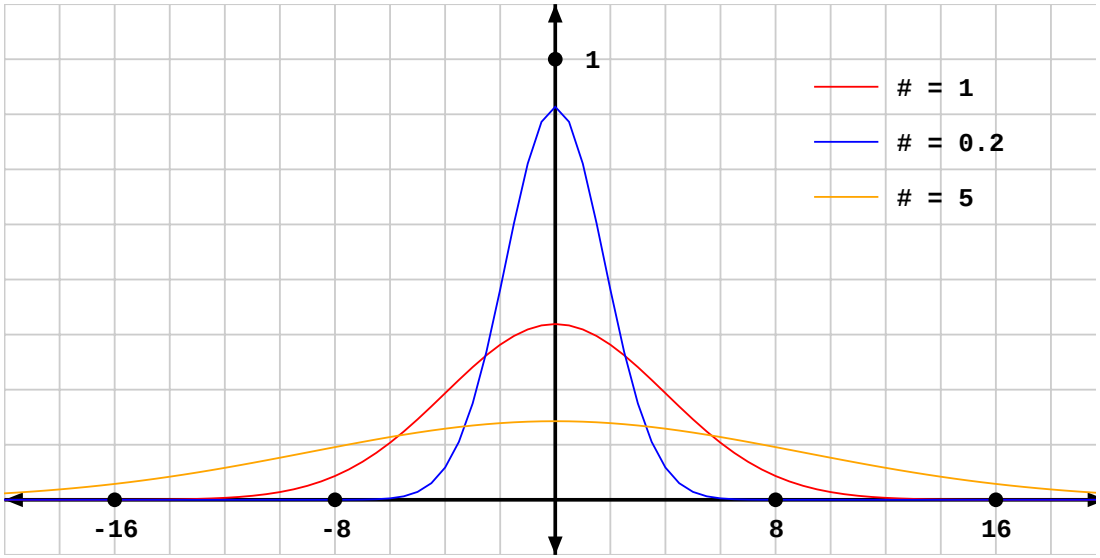
In statistics, the very first place you go for modelling anything with a probability distribution is to the *normal distribtuion* or *Gaussian distribution.* It may not be the correct distribution that physically models what the microfacet distribution of a surface looks like, but it's usually a good starting point.

The Gaussian distribution is the classic “bell-shaped curve” distribution. The mathematical function for computing the probability density of the Gaussian distribution at a particular point X is:

Equation 11.5. Gaussian Distribution Function

$$f(x) = \frac{1}{\sqrt{2\#\#}} e^{-\frac{(x-\mu)^2}{2\#^2}}$$

Figure 11.14. Gaussian Probability Distribution Curves



This represents the percentage of the items in the distribution that satisfy the property that the X in the distribution is trying to model. The `e` in this equation is a common mathematical constant, equivalent to ~2.718. The value of μ is the average. So the absolute value of X is not important; what matters is how far X is from the average.

The value $\#^2$ is the variance of the Gaussian distribution. Without getting too technical, the larger this value becomes, the flatter and wider the distribution is. The variance specifies how far from the average you can get to achieve a certain probability density. The area of the distribution that is positive and negative # away from the average takes up ~68% of the possible values. The area that is 2# away represents ~95% of the possible values.

We know what the average is for us: the surface normal. We can incorporate what we learned from Blinn, by measuring the distance from perfect reflection by comparing the surface normal to the half-angle vector. Thus, the X values represents the angle between the surface normal and half-angle vector. The value μ, the average, is zero.

The equation we will be using for modelling the microfacet distribution with a Gaussian distribution is a slightly simplified form of the Gaussian distribution equation.

Equation 11.6. Gaussian Specular Term

: Angle between H and N

$$\text{Gaussian Term} = e^{-(\frac{\#}{m})^2}$$

This replaces our Phong and Blinn terms in our specular lighting equation and gives us the *Gaussian specular model*. The value `m` ranges from (0, 1], with larger values representing an increasingly rougher surface. Technically, you can use values larger than 1, but the results begin looking increasingly less useful. A value of 1 is plenty rough enough for specular reflection; properly modelling extremely rough surfaces requires additional computations besides determining the distribution of microfacets.

The Gaussian Specular Lighting tutorial shows an implementation of Gaussian specular. It allows a comparison between Phong, Blinn, and Gaussian. It controls the same as the previous tutorial, with the **H** key switching between the three specular computations, and the **Shift**+**H** switching between diffuse+specular and specular only.

Here is the fragment shader for doing Gaussian lighting.

Example 11.3. Gaussian Lighting Shader

```
vec3 lightDir = vec3(0.0);
float atten = CalcAttenuation(cameraSpacePosition, lightDir);
vec4 attenIntensity = atten * lightIntensity;

vec3 surfaceNormal = normalize(vertexNormal);
float cosAngIncidence = dot(surfaceNormal, lightDir);
cosAngIncidence = clamp(cosAngIncidence, 0, 1);

vec3 viewDirection = normalize(-cameraSpacePosition);

vec3 halfAngle = normalize(lightDir + viewDirection);
float angleNormalHalf = acos(dot(halfAngle, surfaceNormal));
float exponent = angleNormalHalf / shininessFactor;
exponent = -(exponent * exponent);
float gaussianTerm = exp(exponent);

gaussianTerm = cosAngIncidence != 0.0 ? gaussianTerm : 0.0;

outputColor = (diffuseColor * attenIntensity * cosAngIncidence) +
    (specularColor * attenIntensity * gaussianTerm) +
    (diffuseColor * ambientIntensity);
```

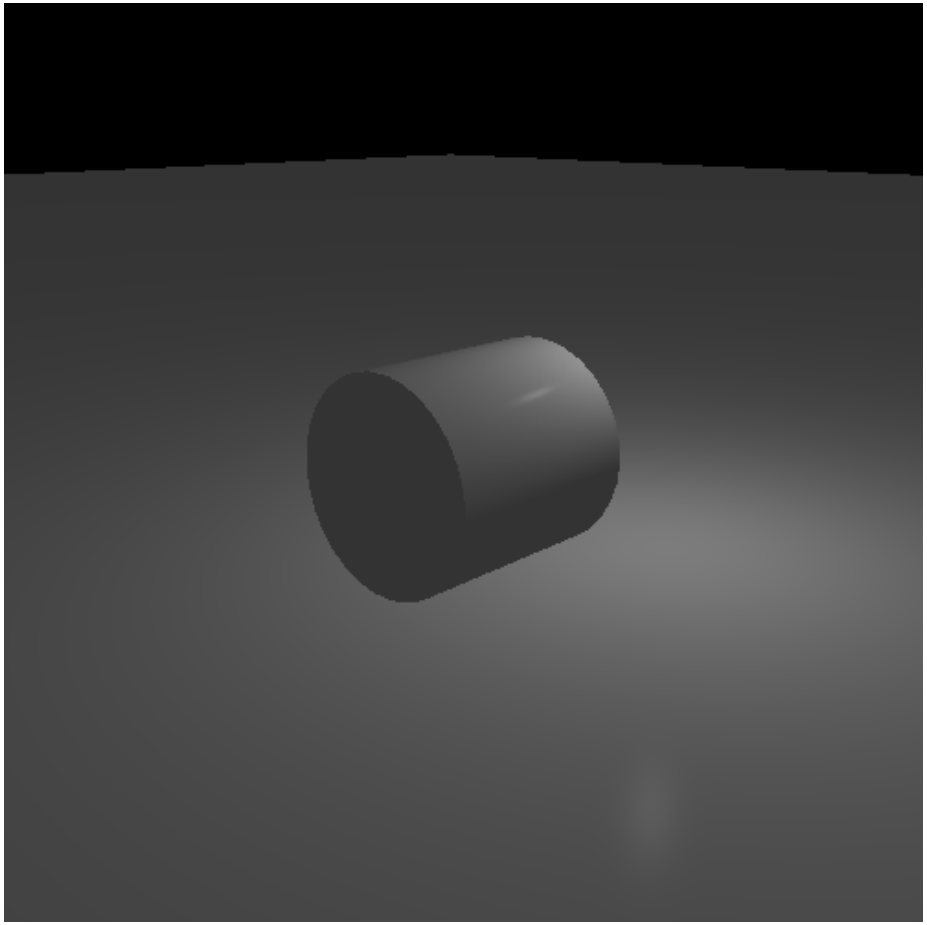
Computing the angle between the half-angle vector and the surface normal requires the use of the `acos` function. We use the dot-product to compute the cosine of the angle, so we need a function to undo the cosine operation. The *arc cosine* or *inverse cosine* function takes the result of a cosine and returns the angle that produced that value.

To do the exponentiation, we use the `exp` function. This function raises the constant `e` to the power of the argument. Everything else proceeds as expected.

What Gaussian Offers

If you play around with the controls, you can see how much the Gaussian distribution offers over Phong and Blinn. For example, set the Gaussian smoothness value to 0.05.

Figure 11.15. Gaussian with Sharp Highlight



It requires very large exponents, well in excess of 100, to match the small size and focus of that specular highlight with Phong or Blinn. It takes even larger exponents to match the Gaussian value of 0.02.

Otherwise the differences between Gaussian and Blinn are fairly subtle. For rough surfaces, there is little substantive difference. But Gaussian tends to have a sharper, more distinct highlight for shiny surfaces.

On Performance

The three specular highlight models seen here are obviously more computationally expensive than diffuse lighting. But which is ultimately more expensive than the others?

The difference between Phong and Blinn is that Phong must compute the reflection vector, while Blinn computes the half-angle vector. The equation for computing the reflection vector is:

$$\hat{I} = -\hat{L}$$
$$\hat{R} = \hat{I} - 2(\hat{N} \cdot \hat{I}) * \hat{N}$$

This involves a vector dot product, a scalar multiply, a vector-scalar multiply, and a vector addition (subtraction). Computing the half-angle vector requires doing a vector addition and performing a normalize operation. Normalizing a vector requires a vector dot product (dotting the vector with itself), taking the square-root of that value, and then a vector-scalar divide by that value.

Time once was that it was easy to know what was faster. The presence of a square-root operation alone would have condemned Blinn as the slower method. On modern 3D graphics hardware however, taking the reciprocal square-root (1 / #X) is generally about as fast as doing a vector multiply. This puts Blinn as approximately equal in performance to Phong; on some hardware, it may even be faster. In general, the performance difference between the two will be negligible.

Gaussian is a different story. It would be reasonable to expect the `pow` function, taking x^y for arbitrary values, to be slower than executing `exp`, e^x. They might have the same performance, but if one is going to be faster, it is more likely to be `exp` than `pow`. However, Gaussian also uses the inverse cosine to compute the angle between the normal and half-angle vector; that pretty much negates any possibility of performance parity. The inverse cosine computation is certainly not built into the hardware, and thus must be computed using the shader logic. And while this is likely true of exponentiation and power functions, Gaussian has to do *two* of these operations, compared to just one for Phong or Blinn.

One might consider using Gouraud shading with specular reflections as a method of optimization. That is, doing per-vertex specular reflections. Since there are fewer vertices than fragments, this might sound like a good idea. However, this is not for the best. Specular highlights do not interpolate linearly at all, so unless the mesh is finely divided, it will generally look awful.

In Review

In this tutorial, you have learned the following:

- Specular lighting represents direct, mirror-like reflections from a surface. Specular highlights are mirror-like reflections directly from a light source. Adding weak specular highlights to even rough surfaces can increase visual realism.
- The microfacet model of specular reflection means that, for a given surface area, there are many mirror-like surfaces. Each microfacet reflects perfectly in its direction. The average of the microfacets
- The Phong and Blinn models of specular reflection use a power function based on how close the viewer is to perfect reflection to approximate a microfacet distribution.
- A Gaussian statistical distribution can be used to more accurately model the distributions of microfacets on a surface.

Further Study

Try doing these things with the given programs.

- Change the shaders to use the diffuse color as the specular color. You may need to drop the specular color somewhat to keep from over-brightening the scene. How this all looks will be particularly evident with the colored cylinder.

Further Research

As you might guess, this is far from the end on specular reflections and specular highlights. Accurately modelling specular reflection is very difficult; doing so while maintaining high performance is even moreso.

If you are interested in more accurate models of specular highlights, there is the Beckmann distribution. This is a particular statistical distribution of microfacets that is more physically based than a Gaussian distribution. It may or may not be a bit more computationally expensive than Gaussian; Beckmann lacks the inverse cosine, but has more other math to it. The two do have a roughness factor that has the same range, (0, 1], and the roughness has the same general meaning in both distributions.

If you want to go even farther, investigate the Cook-Torrance model of specular reflection. It incorporates several terms. It uses a statistical distribution to determine the number of microfacets oriented in a direction. This distribution can be Gaussian, Beckmann, or some other distribution. It modifies this result based on a geometric component that models microfacet self-shadowing and the possibility for multiple interreflections among a microfaceted surface. And it adds a term to compensate for the Fresnel effect: an effect where specular reflection from a surface is more intense when viewed edge-on than directly top-down.

GLSL Functions of Note

```
vec reflect(vec I, vec N);
```

Computes the vector that would be reflected across the normal `N` from an incident vector `I`. The vector result will be normalized if the input vectors are normalized. Note that `I` vector is the vector *towards* the surface.

```
vec pow(vec X, vec Y);
```

Raises `X` to the power of `Y`, component-wise. If a component of `X` is less than 0, then the resulting value is undefined. If `X` is exactly zero, and `Y` is less than or equal to 0, then the resulting value is undefined.

```
vec acos(vec X);
```

Returns the inverse cosine of `X`, component-wise. This returns the angle in radians, which is on the range $[0, \pi]$. If any component of `X` is outside of the [-1, 1] range, then that component of the result will be undefined. This is because the cosine of a value is always on [-1, 1], so the inverse-cosine function cannot take values outside of this range.

```
vec exp(vec exponent);
```

Returns the value of $e^{exponent}$, component-wise.

Glossary

specular reflection
: A mirror-like reflection of light from a surface. Specular reflections reflect light; thus, the color the viewer sees is strongly based on the view angle relative to the light. Specular reflections often do not affect the color of the incoming light.

specular highlights
: Mirror-like reflections directly from light sources. Since light sources are brighter than light reflected by other objects, modelling only specular highlights can provide useful realism without having to model reflections from light produced by other objects in the scene.

angle of view, viewing angle
: The angle between the surface normal and the direction to the viewer/camera.

microfacet model
: Describes a surface as a number of flat planes called microfacets. Each microfacet reflects light using a simple lighting model. The light from a portion of the surface is simply the aggregate of the light from all of the microfacets of the surface. The statistical distribution of microfacet directions on a surface becomes an integral part of the lighting equation. The normal of a surface at a point is the average normal of the microfacets of that part of the surface.

  The microfacet model can be used to model the reflectance characteristics of rough surfaces.

Phong specular model
: A simple model for creating specular highlights. It uses a power function to determine the distribution of microfacets of the surface. The base of the power function is the cosine of the angle between the view direction and the direction of perfect reflection along the surface normal. The exponent is an arbitrary value on the range $(0, \infty)$; large values describe increasingly shiny surfaces, while small values are for rough surfaces.

half-angle vector
: The vector halfway between the direction towards the light and the view direction. When the half-angle vector is oriented exactly with the surface normal, then the view direction is oriented along the reflection direction. For a given light and view direction, it is the direction that the surface normal would need to be facing for a direct light reflection to go from the light source to the viewer.

Blinn-Phong specular model
: A simple model for creating specular highlights. Like standard Phong, it uses a power function to model the distribution of microfacets. The base of the power function is the cosine of the angle between the half-angle vector and the surface normal. The exponent is an arbitrary value on the range $(0, \infty)$; large values describe increasingly shiny surfaces, while small values are for rough surfaces.

Gaussian distribution, normal distribution
: A common statistical distribution. It defines the familiar “bell-shaped curve,” with the average value at the highest point of the distribution.

Gaussian specular model
: A model for creating specular highlights. It uses the Gaussian distribution to model the distribution of microfacets on a surface. It uses a value to control the distribution; this value ranges on (0, 1], where small numbers are smooth surfaces and large numbers are rough surfaces.

inverse cosine, arc cosine
: Performs the opposite of the cosine function. The cosine function takes angles and returns a value on the range [-1, 1]. The inverse cosine takes values on the range [-1, 1] and returns an angle in radians.

Chapter 12. Dynamic Range

Thus far, our lighting examples have been fairly prosaic. A single light source illuminating a simple object hovering above flat terrain. This tutorial will demonstrate how to use multiple lights among a larger piece of terrain in a dynamic lighting environment. We will demonstrate how to properly light a scene. This is less about the artistic qualities of how a scene should look and more about how to make a scene look a certain way if that is how you desire it to look.

Setting the Scene

The intent for this scene is to be dynamic. The terrain will be large and hilly, unlike the flat plain we've seen in previous tutorials. It will use vertex colors where appropriate to give it terrain-like qualities. There will also be a variety of objects on the terrain, each with its own set of reflective characteristics. This will help show off the dynamic nature of the scene.

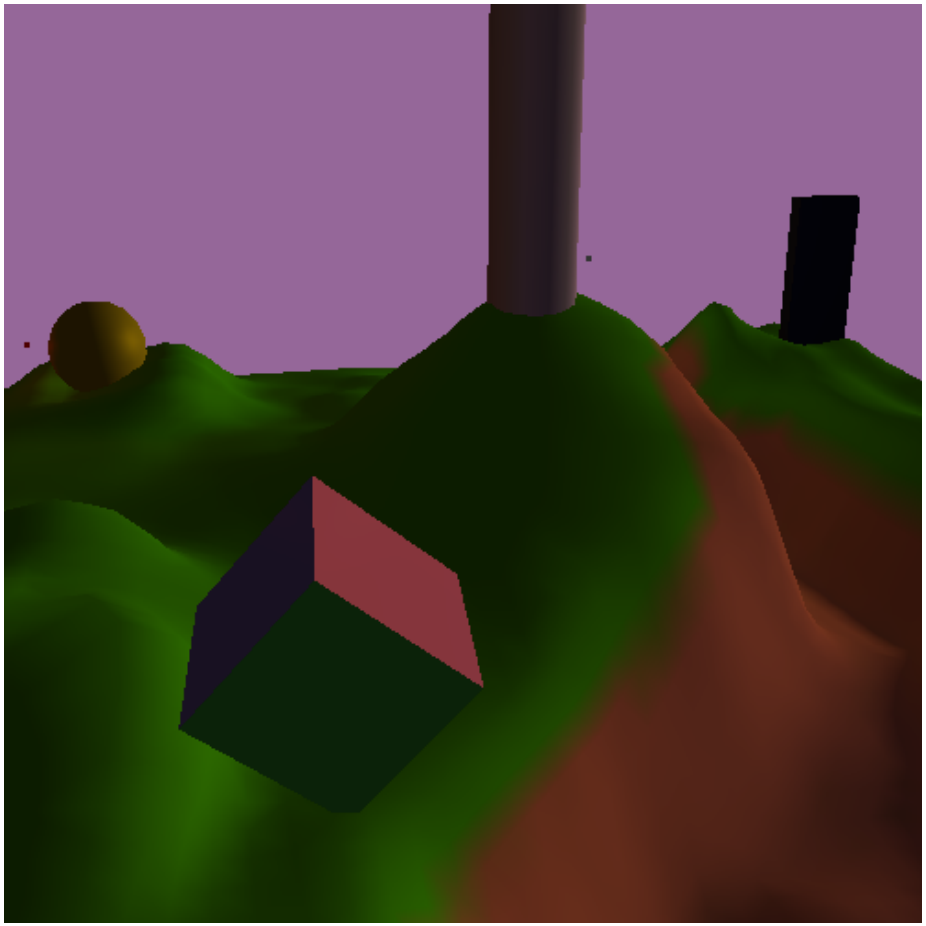
The very first step in lighting a scene is to explicitly detail what you want; without that, you're probably not going to find your way there. In this case, the scene is intended to be outdoors, so there will be a single massive directional light shining down. There will also be a number of smaller, weaker lights. All of these lights will have animated movement.

The biggest thing here is that we want the scene to dynamically change lighting levels. Specifically, we want a full day/night cycle. The sun will sink, gradually losing intensity until it has none. There, it will remain until the dawn of the next day, where it will gain strength and rise again. The other lights should be much weaker in overall intensity than the sun.

One thing that this requires is a dynamic ambient lighting range. Remember that the ambient light is an attempt to resolve the global illumination problem: that light bounces around in a scene and can therefore come from many sources. When the sun is at full intensity, the ambient lighting of the scene should be bright as well. This will mean that surfaces facing away from the sunlight will still be relatively bright, which is the case we see outside. When it is night, the ambient light should be virtually nil. Only surfaces directly facing one of the lights should be illuminated.

The Scene Lighting tutorial demonstrates the first version of attempting to replicate this scene.

Figure 12.1. Scene Lighting



The camera is rotated and zoomed as in prior tutorials. Where this one differs is that the camera's target point can be moved. The **W**, **A**, **S**, and **D** keys move the cameras forward/backwards and left/right, relative to the camera's current orientation. The **Q** and **E** keys raise and lower the camera, again relative to its current orientation. Holding **Shift** with these keys will move in smaller increments. You can toggle viewing of the current target point by pressing **T**.

Because the lighting in this tutorial is very time based, there are specialized controls for playing with time. There are two sets of timers: one that controls the sun's position (as well as attributes associated with this, like the sun's intensity, ambient intensity, etc), and another set of timers that control the positions of other lights in the scene. Commands that affect timers can affect the sun only, the other lights only, or both at the same time.

To have timer commands affect only the sun, press **2**. To have timer commands affect only the other lights, press **3**. To have timer commands affect both, press **1**.

To rewind time by one second (of real-time), press the **-** key. To jump forward one second, press the **=** key. To toggle pausing, press the **p** key. These commands only affect the currently selected timers. Also, pressing the **SpaceBar** will print out the current sun-based time, in 24-hour notation.

Materials and UBOs

The source code for this tutorial is much more complicated than prior ones. Due to this complexity, it is spread over several files. All of the tutorial projects for this tutorial share the `Scene.h/cpp` and `Lights.h/cpp` files. The Scene files set up the objects in the scene and render them. This file contains the surface properties of the objects.

A lighting function requires two specific sets of parameters: values that represent the light, and values that represent the surface. Surface properties are often called *material properties*. Each object has its own material properties, as defined in `Scene.cpp`.

The scene has 6 objects: the terrain, a tall cylinder in the middle, a multicolored cube, a sphere, a spinning tetrahedron, and a mysterious black obelisk. Each object has its own material properties defined by the `GetMaterials` function.

These properties are all stored in a uniform buffer object. We have seen these before for data that is shared among several programs; here, we use it to quickly change sets of values. These material properties do not change with time; we set them once and do not change them ever again. This is primarily for demonstration purposes, but it could have a practical effect.

Each object's material data is defined as the following struct:

Example 12.1. Material Uniform Block

```
//GLSL
layout(std140) uniform;

uniform Material
{
    vec4 diffuseColor;
    vec4 specularColor;
    float specularShininess;
} Mtl;

//C++
struct MaterialBlock
{
    glm::vec4 diffuseColor;
    glm::vec4 specularColor;
    float specularShininess;
    float padding[3];
};
```

The `padding` variable in the C++ definition represents the fact that the GLSL definition of this uniform block will be padded out to a size of 12 floats. This is due to the nature of “std140” layout (feel free to read the appropriate section in the OpenGL specification to see why). Note the global definition of “std140” layout; this sets all uniform blocks to use “std140” layout unless they specifically override it. That way, we do not have to write “layout(std140)” for each of the three uniform blocks we use in each shader file.

Also, note the use of `Mtl` at the foot of the uniform block definition. This is called the *instance name* of an interface block. When no instance name is specified, then the names in the uniform block are global. If an instance name is specified, this name must be used to qualify access to the names within that block. This allows us to have the `in vec4 diffuseColor` be separate from the material definition's `Mtl.diffuseColor`.

What we want to do is put 6 material blocks in a single uniform buffer. One might naively think that one could simply allocate a buffer object 6 times the `sizeof(MaterialBlock)`, and simply store the data as a C++ array. Sadly, this will not work due to a UBO limitation.

When you use `glBindBufferRange` to bind a UBO, OpenGL requires that the offset parameter, the parameter that tells where the beginning of the uniform block's data is within the buffer object, be aligned to a specific value. That is, the begining of a uniform block within a uniform buffer must be a multiple of a specific value. 0 works, of course, but since we have more than one block within a uniform buffer, they cannot all start at the buffer's beginning.

What is this value, you may ask? Welcome to the world of implementation-dependent values. This means that it can (and most certainly will) change depending on what platform you're running on. This code was tested on two different hardware platforms; one has a minimum alignment of 64, the other an alignment of 256.

To retrieve the implementation-dependent value, we must use a previously-unseen function: `glGetIntegerv`. This is a function that does one simple thing: gets integer values from OpenGL. However, the meaning of the value retrieved depends on the enumerator passed as the first parameter. Basically, it's a way to have state retrieval functions that can easily be extended by adding new enumerators rather than new functions.

The code that builds the material uniform buffer is as follows:

Example 12.2. Material UBO Construction

```
int uniformBufferAlignSize = 0;
glGetIntegerv(GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT, &uniformBufferAlignSize);

m_sizeMaterialBlock = sizeof(MaterialBlock);
m_sizeMaterialBlock += uniformBufferAlignSize -
 (m_sizeMaterialBlock % uniformBufferAlignSize);

int sizeMaterialUniformBuffer = m_sizeMaterialBlock * MATERIAL_COUNT;

std::vector<MaterialBlock> materials;
GetMaterials(materials);
assert(materials.size() == MATERIAL_COUNT);

std::vector<GLubyte> mtlBuffer;
mtlBuffer.resize(sizeMaterialUniformBuffer, 0);

GLubyte *bufferPtr = &mtlBuffer[0];

for(size_t mtl = 0; mtl < materials.size(); ++mtl)
 memcpy(bufferPtr + (mtl * m_sizeMaterialBlock), &materials[mtl], sizeof(MaterialBlock));

glGenBuffers(1, &m_materialUniformBuffer);
glBindBuffer(GL_UNIFORM_BUFFER, m_materialUniformBuffer);
glBufferData(GL_UNIFORM_BUFFER, sizeMaterialUniformBuffer, bufferPtr, GL_STATIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

We use `glGetIntegerv` to retrieve the alignment requirement. Then we compute the size of a material block, plus enough padding to satisfy the alignment requirements. From there, it's fairly straightforward. The `mtlBuffer` is just a clever way to allocate a block of memory without having to directly use new/delete. And yes, that is perfectly valid and legal C++.

When the scene is rendered, it uses `glBindBufferRange` to bind the proper region within the buffer object for rendering.

Lighting

The code for lighting is rather more complicated. It uses two aspects of the framework library to do its job: interpolators and timers. `Framework::Timer` is a generally useful class that can keep track of a looped range of time, converting it into a [0, 1) range. The interpolators are used to convert a [0, 1) range to a particular value based on a series of possible values. Exactly how they work is beyond the scope of this discussion, but some basic information will be presented.

The `LightManager` class controls all timers. It has all of the fast-forwarding, rewinding, and so forth controls built into it. It's basic functionality is to compute all of the lighting values for a particular time. It does this based on information given to it by the main tutorial source file, `SceneLighting.cpp`. The important values are sent in the `SetupDaytimeLighting` function.

Example 12.3. Daytime Lighting

```
SunlightValue values[] =
```

```
{
    { 0.0f/24.0f, /*...*/},
    { 4.5f/24.0f, /*...*/},
    { 6.5f/24.0f, /*...*/},
    { 8.0f/24.0f, /*...*/},
    {18.0f/24.0f, /*...*/},
    {19.5f/24.0f, /*...*/},
    {20.5f/24.0f, /*...*/},
};

g_lights.SetSunlightValues(values, 7);

g_lights.SetPointLightIntensity(0, glm::vec4(0.2f, 0.2f, 0.2f, 1.0f));
g_lights.SetPointLightIntensity(1, glm::vec4(0.0f, 0.0f, 0.3f, 1.0f));
g_lights.SetPointLightIntensity(2, glm::vec4(0.3f, 0.0f, 0.0f, 1.0f));
```

For the sake of clarity, the actual lighting parameters were removed from the main table. The `SunlightValue` struct defines the parameters that vary based on the sun's position. Namely, the ambient intensity, the sun's light intensity, and the background color. The first parameter of the struct is the time, on the [0, 1) range, when the parameters should have this value. A time of 0 represents noon, and a time of 0.5 represents midnight. For clarity's sake, I used 24-hour notation (where 0 is noon rather than midnight).

We will discuss the actual lighting values later.

The main purpose of the `LightManager` is to retrieve the light parameters. This is done by the function `GetLightInformation`, which takes a matrix (to transform the light positions and directions into camera space) and returns a `LightBlock` object. This is an object that represents a uniform block defined by the shaders:

Example 12.4. Light Uniform Block

```
struct PerLight
{
    vec4 cameraSpaceLightPos;
    vec4 lightIntensity;
};

const int numberOfLights = 4;

uniform Light
{
    vec4 ambientIntensity;
    float lightAttenuation;
    PerLight lights[numberOfLights];
} Lgt;
```

```
struct PerLight
{
    glm::vec4 cameraSpaceLightPos;
    glm::vec4 lightIntensity;
};

const int NUMBER_OF_LIGHTS = 4;

struct LightBlock
{
    glm::vec4 ambientIntensity;
    float lightAttenuation;
    float padding[3];
    PerLight lights[NUMBER_OF_LIGHTS];
```



```
};
```

Again, there is the need for a bit of padding in the C++ version of the struct. Also, you might notice that we have both arrays and structs in GLSL for the first time. They work pretty much like C/C++ structs and arrays (outside of pointer logic, since GLSL does not have pointers), though arrays have certain caveats.

Many Lights Shader

In this tutorial, we use 4 shaders. Two of these take their diffuse color from values passed by the vertex shader. The other two use the material's diffuse color. The other difference is that two do specular reflection computations, and the others do not. This represents the variety of our materials.

Overall, the code is nothing you have not seen before. We use Gaussian specular and an inverse-squared attenuation, in order to be as physically correct as we currently can be. One of the big differences is in the `main` function.

Example 12.5. Many Lights Main Function

```
void main()
{
    vec4 accumLighting = diffuseColor * Lgt.ambientIntensity;
    for(int light = 0; light < numberOfLights; light++)
    {
        accumLighting += ComputeLighting(Lgt.lights[light]);
    }

    outputColor = accumLighting;
}
```

Here, we compute the lighting due to the ambient correction. Then we loop over each light and compute the lighting for it, adding it into our accumulated value. Loops and arrays are generally fine.

The other trick is how we deal with positional and directional lights. The `PerLight` structure does not explicitly say whether a light is positional or directional. However, the W component of the `cameraSpaceLightPos` is what we use to differentiate them; this is a time-honored technique. If the W component is 0.0, then it is a directional light; otherwise, it is a point light.

The only difference between directional and point lights in the lighting function are attenuation (directional lights do not use attenuation) and how the light direction is computed. So we simply compute these based on the W component:

```
vec3 lightDir;
vec4 lightIntensity;
if(lightData.cameraSpaceLightPos.w == 0.0)
{
    lightDir = vec3(lightData.cameraSpaceLightPos);
    lightIntensity = lightData.lightIntensity;
}
else
{
    float atten = CalcAttenuation(cameraSpacePosition,
        lightData.cameraSpaceLightPos.xyz, lightDir);
    lightIntensity = atten * lightData.lightIntensity;
}
```

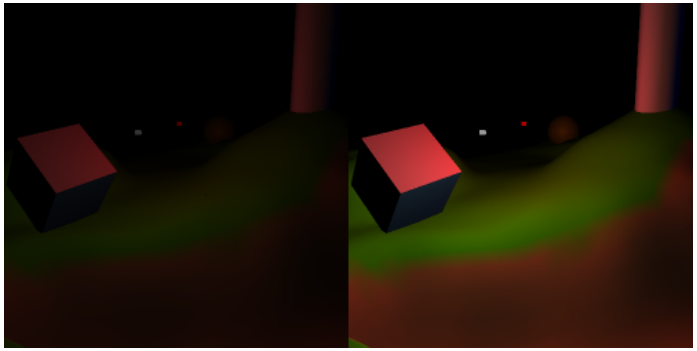
Lighting Problems

There are a few problems with our current lighting setup. It looks (mostly) fine in daylight. The moving point lights have a small visual effect, but mostly they're not very visible. And this is what one would expect in broad daylight; flashlights do not make much impact in the day.

But at night, everything is exceedingly dark. The point lights, the only active source of illumination, are all too dim to be very visible. The terrain almost completely blends into the black background.

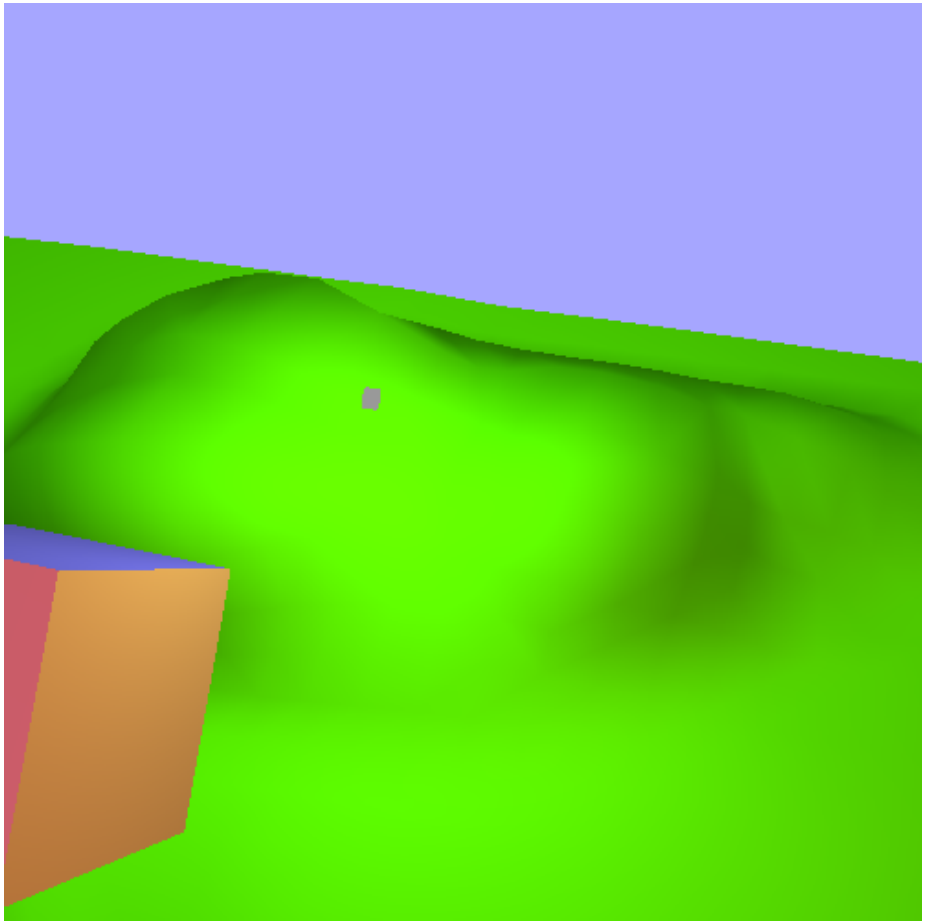
There is an alternative set of light parameters that corrects this problem. Press **Shift**+**L**; that switches to a night-time optimized version (press **L** to switch back to day-optimized lighting). Here, the point lights provide reasonable lighting at night. The ground is still dark when facing away from the lights, but we can reasonably see things.

Figure 12.2. Darkness, Day vs. Night



The problem is that, in daylight, the night-optimized point lights are too powerful. They are very visible and have very strong effects on the scene. Also, they cause some light problems when one of the point lights is in the right position. At around 12:50, find the floating white light near the cube:

Figure 12.3. Light Clipping



Notice the patch of iridescent green. This is *light clipping* or light clamping, and it is usually a very undesirable outcome. It happens when the computed light intensity falls outside of the [0, 1] range, usually in the positive direction (like in this case). The object cannot be shown to be brighter, so it becomes a solid color that loses all detail.

The obvious solution to our lighting problem is to simply change the point light intensity based on the time of day. However, this is not realistic; flashlights do not actually get brighter at night. So if we have to do something that antithetical to reality, then there's probably some aspect of reality that we are not properly modelling.

High Dynamic Range

In order to answer this question, we must first determine why flashlights appear brighter at night than in the daytime. Much of the answer has to do with our eyes.

The pupil is the hole in our eyes that allows light to pass through it; cameras call this hole the aperture. The hole is small, relative to the world, which helps with resolving an image. However, the quantity of light that passes through the hole depends on how large it is. Our iris's can expand and contract to control the size of the pupil. When the pupil is large, more light is allowed to enter the eye; when the pupil is small, less light can enter.

The iris contracts automatically in the presence of bright light, since too much like can damage the retina (the surface of the eye that detects light). However, in the absence of light, the iris slowly relaxes, expanding the pupil. This has the effect of allowing more light to enter the eye, which adds to the apparent brightness of the scene.

So what we need is not to change the overall light levels, but instead apply the equivalent of an iris to the final lighting computations of a scene. That is, we determine the overall illumination at a point, but we then filter out some of this light based on a global setting. In dark scenes, we filter less light, and in bright scenes, we filter more light.

This overall process is called *high dynamic range lighting* (HDR). It is fairly simple and requires very few additional math computations compared to our current model.

Note

You may have heard this term in conjunction with pictures where bright objects seem to glow. While HDR is typically associated with that glowing effect, that is a different effect called bloom. It is a woefully over-used effect, and we will discuss how to implement it later. HDR and bloom do interact, but you can use one without the other.

The first step is to end the myth that lights themselves have a global maximum brightness. In all previous examples, our light intensity values lived with the range [0, 1]. Clamping light intensity to this range simply does not mirror reality. In reality, the sun is many orders of magnitude brighter than a flashlight. We must allow for this in our lighting equation.

This also means that our accumulated lighting intensity, the value we originally wrote to the fragment shader output, is no longer on the [0, 1] range. And that poses a problem. We can perform lighting computations with a high dynamic range, but monitors can only display colors on the [0, 1] range. We therefore must map from the HDR to the low dynamic range (LDR).

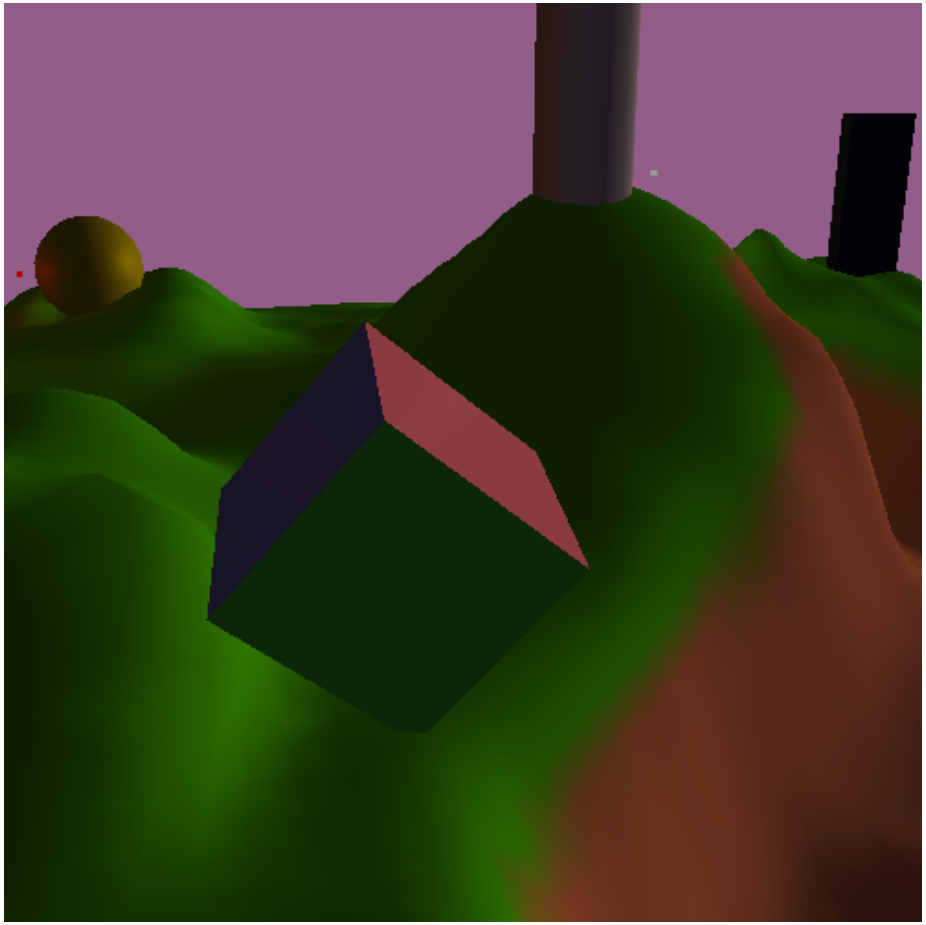
This part of HDR rendering is called *tone mapping*. There are many possible tone mapping functions, but we will use one that simulates a flexible aperture. It's quite a simple function, really. First, we pick a maximum intensity value for the scene; intensity values above this will be clamped. Then, we just divide the HDR value by the maximum intensity to get the LDR value.

It is the maximum intensity that we will vary. As the sun goes down, the intensity will go down with it. This will allow the sun to be much brighter in the day than the lights, thus overwhelming their contributions to the scene's illumination. But at night, when the maximum intensity is much lower, the other lights will have an apparently higher brightness.

This is implemented in the HDR Lighting tutorial.

This tutorial controls as the previous one, except that the **K** key will activate HDR lighting. Pressing **L** or **Shift**+**L** will go back to day or night-time LDR lighting from the last tutorial, respectively.

Figure 12.4. HDR Lighting



The code is quite straightforward. We add a floating-point field to the `Light` uniform block and the `LightBlock` struct in C++. Technically, we just steal one of the padding floats, so the size remains the same:

Example 12.6. HDR LightBlock

```
struct LightBlockHDR
{
    glm::vec4 ambientIntensity;
    float lightAttenuation;
    float maxIntensity;
    float padding[2];
    PerLight lights[NUMBER_OF_LIGHTS];
```

```
};
```

We also add a new field to `SunlightValue`: the maximum light intensity. There is also a new function in the `LightManager` that computes the HDR-version of the light block: `GetLightInformationHDR`. Technically, all of this code was already in `Light.h/cpp`, since these files are shared among all of the tutorials here.

Scene Lighting in HDR

Lighting a scene in HDR is a different process from LDR. Having a varying maximum intensity value, as well as the ability to use light intensities greater than 1.0 change much about how you set up a scene.

In this case, everything in the lighting was designed to match up to the daytime version of LDR in the day, and the nighttime version of LDR at night. Once the division by the maximum intensity was taken into account.

Table 12.1. Scene Lighting Values

	HDR				LDR Day-optimized			LDR Night-optimized		
Noon Sun Intensity	1.8	1.8	1.8	(3.0)	0.6	0.6	0.6	0.6	0.6	0.6
Noon Ambient Intensity	0.6	0.6	0.6		0.2	0.2	0.2	0.2	0.2	0.2
Evening Sun Intensity	0.45	0.15	0.15	(1.5)	0.3	0.1	0.1	0.3	0.1	0.1
Evening Ambient Intensity	0.225	0.075	0.075		0.15	0.05	0.05	0.15	0.05	0.05
Circular Light Intensity	0.6	0.6	0.6		0.2	0.2	0.2	0.6	0.6	0.6
Red Light Intensity	0.7	0.0	0.0		0.3	0.0	0.0	0.7	0.0	0.0
Blue Light Intensity	0.0	0.0	0.7		0.0	0.0	0.3	0.0	0.0	0.7

The numbers in parenthesis represents the max intensity at that time.

In order to keep the daytime lighting the same, we simply multiplied the LDR day's sun and ambient intensities by the ratio between the sun intensity and the intensity of one of the lights. This ratio is 3:1, so the sun and ambient intensity is increased by a magnitude of 3.

The maximum intensity was derived similarly. In the LDR case, the difference between the max intensity (1.0) and the sum of the sun and ambient intensities is 0.2 (1.0 - (0.6 + 0.2)). To maintain this, we set the max intensity to the sum of the ambient and sun intensities, plus 3 times the original ratio.

This effectively means that the light, as far as the sun and ambient are concerned, works the same way in HDR daytime as in the LDR day-optimized settings. To get the other lights to work at night, we simply kept their values the same as the LDR night-optimized case.

Linearity and Gamma

There is one major issue left, and it is one that has been glossed over since the beginning of our look at lighting: your screen.

The fundamental assumption underlying all of our lighting equations since the very beginning is the idea that the surface colors and light intensities are all in a linear *colorspace*. A colorspace defines how we translate from a set of numerical values to actual, real colors that you can see. A colorspace is a *linear colorspace* if doubling any value in that colorspace results in a color that is twice as bright. The linearity refers to the relationship between values and overall brightness of the resulting color.

This assumption can be taken as a given for our data thus far. All of our diffuse and specular color values were given by us, so we can know that they represent values in a linear RGB colorspace. The light intensities are likewise in a linear colorspace. When we multiplied the sun and ambient intensities by 3 in the last section, we were increasing the brightness by 3x. Multiplying the maximum intensity by 3 had the effect of reducing the overall brightness by 3x.

There's just one problem: your screen does not work that way. Time for a short history of television/monitors.

The original televisions used an electron gun fired at a phosphor surface to generate light and images; this is called a CRT display (cathode ray tube). The strength of the electron beam determined the brightness of that part of the image. However, the strength of the beam did not vary linearly with the brightness of the image.

The easiest way to deal with that in the earliest days of TV was to simply modify the incoming image at the source. TV broadcasts sent image data that was non-linear in the opposite direction of the CRT's normal non-linearity. Thus, the final output was displayed linearly, as it was originally captured by the camera.

The term for this process, de-linearizing an image to compensate for a non-linear display, is called *gamma correction*.

You may be wondering why this matters. After all, odds are, you do not have a CRT-based monitor; you probably have some form of LCD, plasma, LED, or similar technology. So what does the vagaries of CRT monitors matter to you?

Because gamma correction is everywhere. It's in DVDs, video-tapes, and Blu-Ray discs. Every digital camera does it. And this is how it has been for a long time. Because of that, you could not sell an LCD monitor that tried to do linear color reproduction; nobody would buy it because all media for it (including your OS) was designed and written expecting CRT-style non-linear displays.

This means that every non-CRT display must mimic the CRT's non-linearity; this is built into the basic video processing logic of every display device.

So for twelve tutorials now, we have been outputting linear RGB values to a display device that expects gamma-corrected non-linear RGB values. But before we started doing lighting, we were just picking nice-looking colors, so it did not matter. Now that we're doing something vaguely realistic, we need to perform gamma-correction. This will let us see what we've *actually* been rendering, instead of what our monitor's gamma-correction circuitry has been mangling.

Gamma Functions

A gamma function is the function that maps linear RGB space to non-linear RGB space. The gamma function for CRT displays was fairly standard, and all non-CRT displays mimic this standard. It is ultimately based on a math function of CRT displays. The strength of the electron beam is controlled by the voltage passed through it. This correlates with the light intensity as follows:

Equation 12.1. Display Gamma Function

$$LinearRGB \# \ Voltage^{\#}$$

This is called a gamma function due to the Greek letter γ (gamma). The input signal directly controls the voltage, so the input signal needed to be corrected for the power of gamma.

Modern displays usually have gamma adjustments that allow the user to set the display's gamma. The default is usually a gamma of around 2.2; this is a useful compromise value and an excellent default for our gamma-correction code.

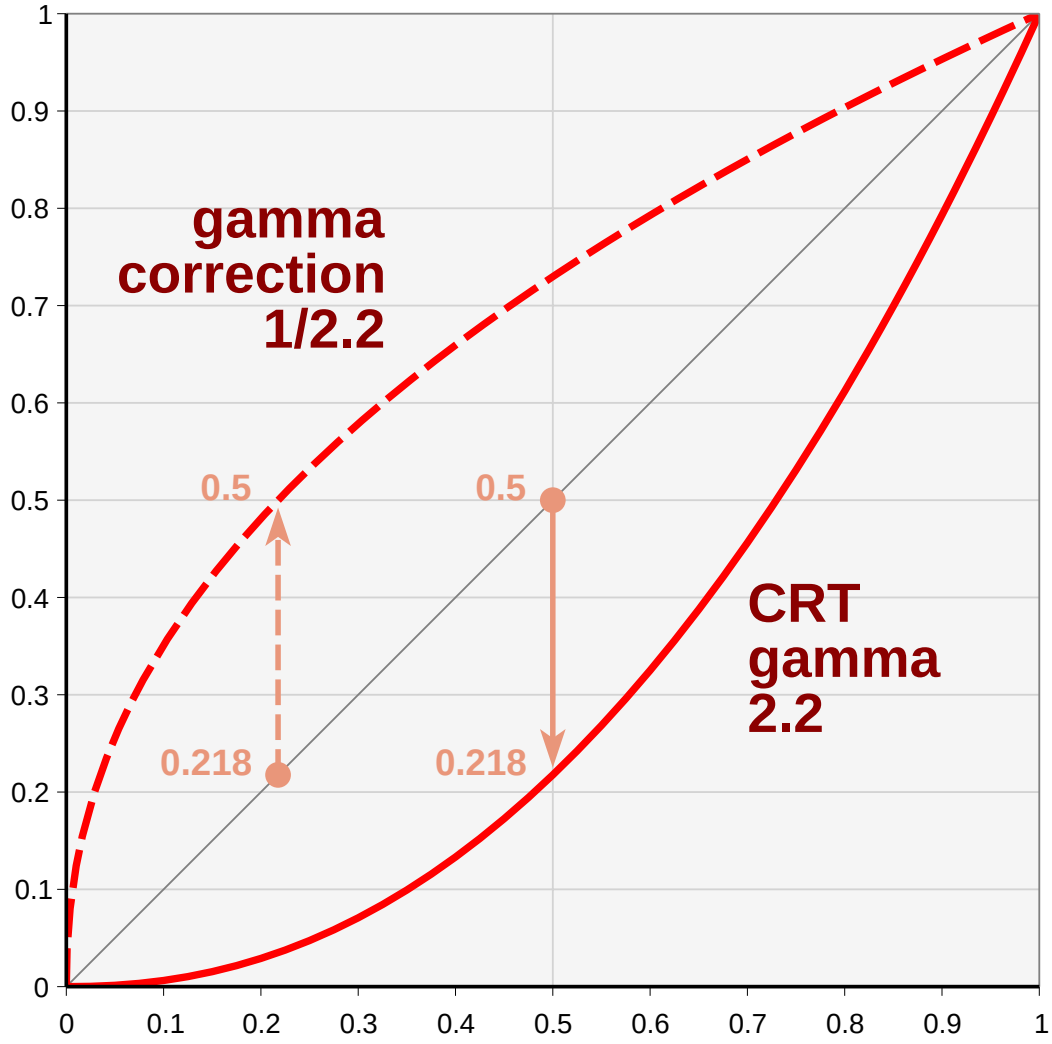
So, given the gamma function above, we need to output values from our shader that will result in our original linear values after the gamma function is applied. This is gamma correction, and the function for that is straightforward.

Equation 12.2. Gamma Correction Function

$$GammaRGB = LinearRGB^{\frac{1}{\#}}$$

It would be interesting to see a graph of these functions, to speculate about what we will see in our gamma-correct images.

Figure 12.5. Gamma Function Graph



Without gamma correction, our linearRGB colors (the diagonal line in the graph) would become the CRT gamma curve at the bottom. This means that what we have been seeing is a *severely* darkened version of our colors. A linearRGB value of 0.5 drops to an intensity of 0.218; that's more than half of the brightness gone.

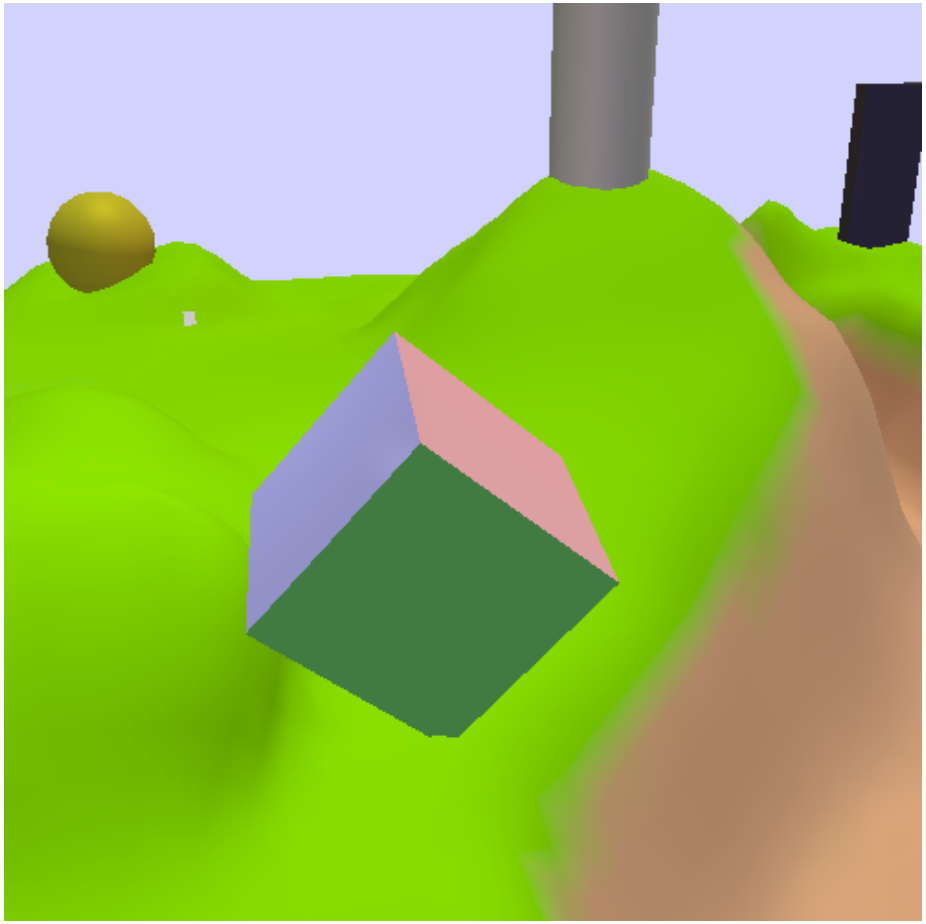
With proper gamma correction, we can expect to see our scene become much brighter.

Gamma in Action

Gamma correction is implemented in the Gamma Correction tutorial.

The **K** key toggles gamma correction. The default gamma value is 2.2, but it can be raised and lowered with the **Y** and **H** keys respectively.

Figure 12.6. Gamma Correction



That is very bright; it uses the same HDR-based lighting environment from the previous tutorials. Let's look at some code.

The gamma value is an odd kind of value. Conceptually, it has nothing to do with lighting, per-se. It is a global value across many shaders, so it should be in a UBO somewhere. But it is not a material parameter; it does not change from object to object. In this tutorial, we stick it in the `Light` uniform block and the `LightBlockGamma` struct. Again, we steal a float from the padding:

Example 12.7. Gamma LightBlock

```
struct LightBlockGamma
{
    glm::vec4 ambientIntensity;
    float lightAttenuation;
    float maxIntensity;
    float gamma;
    float padding;
    PerLight lights[NUMBER_OF_LIGHTS];
};
```

For the sake of clarity in this tutorial, we send the actual gamma value. For performance's sake, we ought send 1/gamma, so that we do not have to needlessly do a division in every fragment.

The gamma is applied in the fragment shader as follows:

Example 12.8. Fragment Gamma Correction

```
accumLighting = accumLighting / Lgt.maxIntensity;
vec4 gamma = vec4(1.0 / Lgt.gamma);
gamma.w = 1.0;
outputColor = pow(accumLighting, gamma);
```

Otherwise, the code is mostly unchanged from the HDR tutorial. Speaking of which, gamma correction does not require HDR per se, nor does HDR require gamma correction. However, the combination of the two has the power to create substantial improvements in overall visual quality.

One final change in the code is for light values that are written directly, without any lighting computations. The background color is simply the clear color for the framebuffer. Even so, it needs gamma correction too; this is done on the CPU by gamma correcting the color before drawing it. If you have any other colors that are drawn directly, *do not* forget to do this.

Gamma Correct Lighting

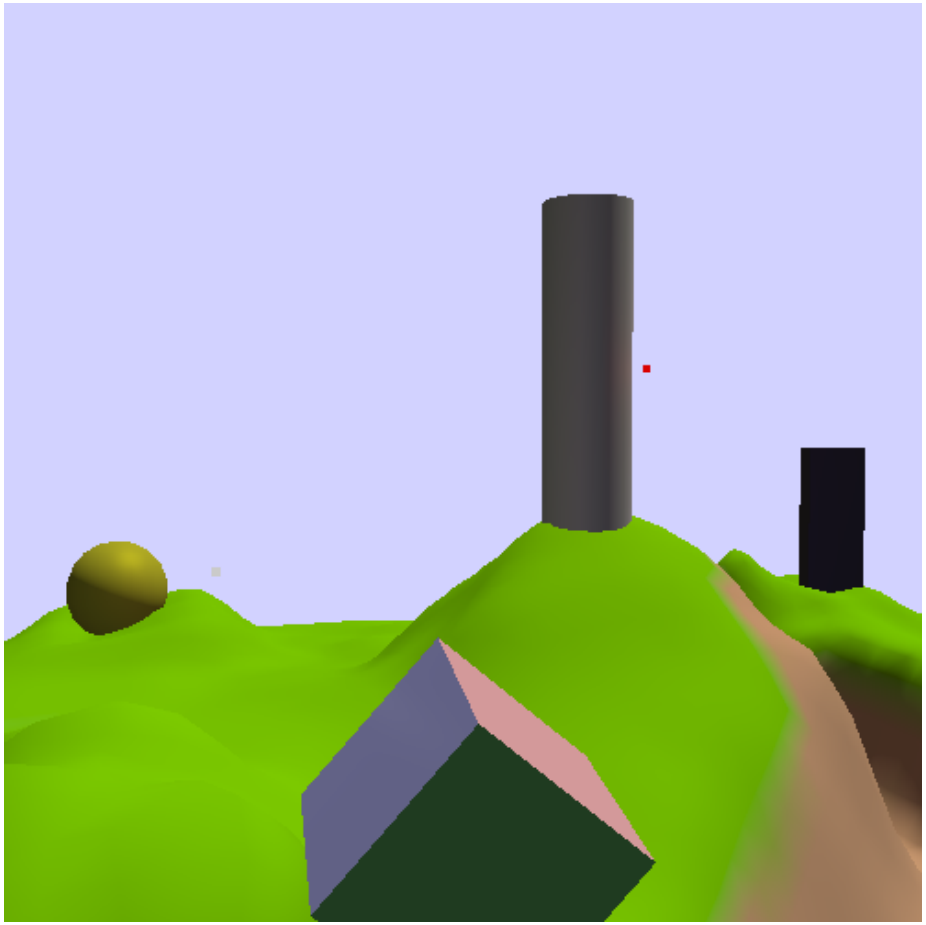
What we have seen is what happens when you apply HDR lighting to a scene who's light properties were defined *without* gamma correction. Look at the scene at night; the point lights are extremely bright, and their lighting effects seem to go much farther than before. This last point bears investigating.

When we first talked about light attenuation, we said that the correct attenuation function for a point light was an inverse-square relationship with respect to the distance to the light. We also said that this usually looked wrong, so people often used a plain inverse attenuation function.

Gamma is the reason for this. Or rather, lack of gamma correction is the reason. Without correcting for the display's gamma function, the attenuation of $1/r^2$ effectively becomes $(1/r^2)^{2.2}$, which is $1/r^{4.4}$. The lack of proper gamma correction magnifies the effective attenuation of lights. A simple 1/r relationship looks better without gamma correction because the display's gamma function turns it into something that is much closer to being in a linear colorspace: $1/r^{2.2}$.

Since this lighting environment was not designed while looking at gamma correct results, let's look at some scene lighting that was developed with proper gamma in mind. Turn on gamma correction and set the gamma value to 2.2 (the default if you did not change it). The press **Shift**+**L**:

Figure 12.7. Gamma Lighting

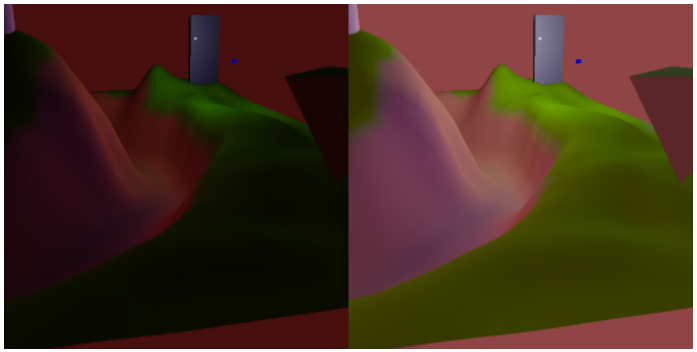


This is more like it.

If there is one point you should learn from this exercise, it is this: make sure that you implement gamma correction and HDR *before* trying to light your scenes. If you do not, then you may have to adjust all of the lighting parameters again, and you may need to change materials as well. In this case, it was not even possible to use simple corrective math on the lighting environment to make it work right. This lighting environment was developed essentially from scratch.

One thing we can notice when looking at the gamma correct lighting is that proper gamma correction improves shadow details substantially:

Figure 12.8. Gamma Shadow Details



These two images use the HDR lighting; the one on the left does not have gamma correction, and the one on the right does. Notice how easy it is to make out the details in the hills near the triangle on the right.

Looking at the gamma function, this makes sense. Without proper gamma correction, fully half of the linearRGB range is shoved into the bottom one-fifth of the available light intensity. That does not leave much room for areas that are dark but not too dark to see anything.

As such, gamma correction is a key process for producing color-accurate rendered images. It allows details within darkness to show up much more easily.

In Review

In this tutorial, you have learned the following:

- How to build and light a scene containing multiple objects and multiple light sources.
- High dynamic range lighting means using a maximum illumination that can vary from frame to frame, rather than a single, fixed value.
- Color values have a space, just like positions or normals. Lighting equations work in a linear colorspace, where twice the brightness of a value is achieved by multiplying its value times two. It is vital for proper imaging results to make sure that the final result of lighting is in the colorspace that the output display expects. This process is called gamma correction.

Further Study

Try doing these things with the given programs.

- Add a fifth light, a directional light representing the moon, to the gamma-correct scene. This will require creating another set of interpolators and expanding the SunlightValues structure to hold the lighting intensity of the moon. It also means expanding the number of lights the shaders use from 4 to 5 (or removing one of the point lights). The moon should be much less bright than the sun, but it should still have a noticeable effect on brightness.
- Play with the ambient lighting intensity in the gamma-correct scene, particularly in the daytime. A little ambient, even with a maximum intensity as high as 10, really goes a long way to bringing up the level of brightness in a scene.

Further Research

HDR is a pretty large field. This tutorial covered perhaps the simplest form of tone mapping, but there are many equations one can use. There are tone mapping functions that map the full $[0, \infty)$ range to [0, 1]. This would not be useful for a scene that needs a dynamic aperture size, but if the aperture is static, it does allow the use of a large range of lighting values.

When doing tone mapping with some form of variable aperture setting, computing the proper maximum intensity value can be difficult. Having a hard-coded value, even one that varies algorithmically, works well enough for some scenes. But for scenes where the user can control where the camera faces, it can be inappropriate. In many modern games, they actually read portions of the rendered image back to the CPU and do some computational analysis to determine what the maximum intensity should be for the next frame. This is delayed, of course, but it allows for an aperture that varies based on what the player is currently looking at, rather than hard-coded values.

Just remember: pick your HDR and tone mapping algorithms *before* you start putting the scene together. If you try to change them mid-stream, you will have to redo a lot of work.

OpenGL Functions of Note

glGetIntegerv	Retrieves implementation-dependent integer values and a number of context state integer values. There are also `glGetFloatv`, `glGetBooleanv`, and various other typed `glGet*` functions. The number of values this retrieves depends on the enumerator passed to it.

Glossary

material properties	The set of inputs to the lighting equation that represent the characteristics of a surface. This includes the surface's normal, diffuse reflectance, specular reflectance, any specular power values, and so forth. The source of these values can come from many sources: uniform values for an object, fragment-shader inputs, or potentially other sources.
instance name	For uniform blocks (or other kinds of interface blocks), this name is used within a shader to name-qualifier the members of the block. These are optional, unless there is a naming conflict, or unless an array needs to be specified.
light clipping	Light values drawn to the screen are clamped to the range [0, 1]. When lighting produces values outside of this range, the light is said to be clipped by the range. This produces a very bright, flat section that loses all detail and distinction in the image. It is something best avoided.
high dynamic range lighting	Lighting that uses values outside of the [0, 1] range. This allows for the use of a full range of lighting intensities.
tone mapping	The process of mapping HDR values to a [0, 1] range. This may or may not be a linear mapping.
colorspace	The set of reference colors that define a way of representing a color in computer graphics, and the function mapping between those reference colors and the actual colors. All colors are defined relative to a particular colorspace.
linear colorspace	A colorspace where the brightness of a color varies linearly with its values. Doubling the value of a color doubles its brightness.
gamma correction	The process of converting from a linear colorspace to a non-linear colorspace that a display device expects, usually through the use of a power function. This process ensures that the display produces an image that is linear.

Chapter 13. Lies and Impostors

Lighting in these tutorials has ultimately been a form of deception. An increasingly accurate one, but it is deception all the same. We are not rendering round objects; we simply use lighting and interpolation of surface characteristics to make an object appear round. Sometimes we have artifacts or optical illusions that show the lie for what it is. Even when the lie is near-perfect, the geometry of a model still does not correspond to what the lighting makes the geometry appear to be.

In this tutorial, we will be looking at the ultimate expression of this lie. We will use lighting computations to make an object appear to be something entirely different from its geometry.

Simple Sham

We want to render a sphere. We could do this as we have done in previous tutorials. That is, generate a mesh of a sphere and render it. But this will never be a mathematically perfect sphere. It is easy to generate a sphere with an arbitrary number of triangles, and thus improve the approximation. But it will always be an approximation.

Spheres are very simple, mathematically speaking. They are simply the set of points in a space that are a certain distance from a specific point. This sounds like something we might be able to compute in a shader.

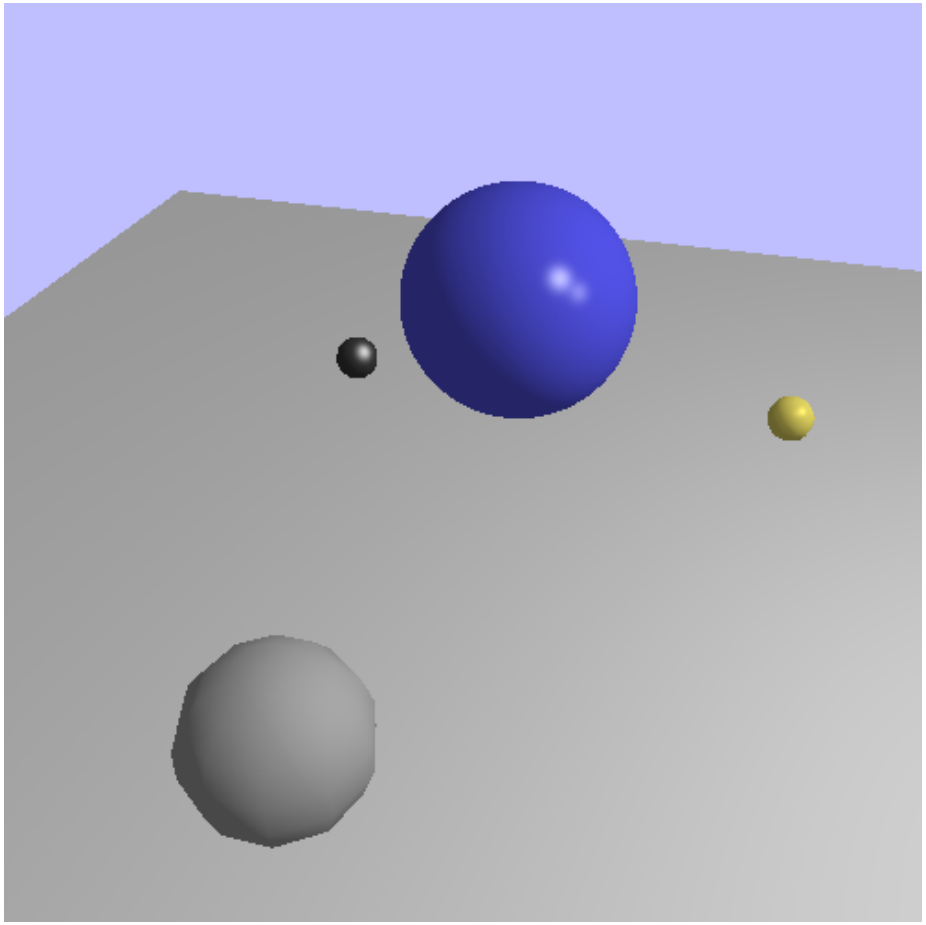
Our first attempt to render a sphere will be quite simple. We will use the vertex shader to compute the vertex positions of a *square* in clip-space. This square will be in the same position and width/height as the actual circle would be, and it will always face the camera. In the fragment shader, we will compute the position and normal of each point along the sphere's surface. By doing this, we can map each point on the square to a point on the sphere we are trying to render. This square is commonly called a *flat card* or *billboard*.

For those points on the square that do not map to a sphere point (ie: the corners), we have to do something special. Fragment shaders are required to write a value to the output image. But they also have the ability to abort processing and write neither color information nor depth to the color and depth buffers. We will employ this to draw our square-spheres.

This technique is commonly called *impostors*. The idea is that we're actually drawing a square, but we use the fragment shaders to make it look like something else. The geometric shape is just a placeholder, a way to invoke the fragment shader over a certain region of the screen. The fragment shader is where the real magic happens.

The tutorial project Basic Impostor demonstrates this technique. It shows a scene with several spheres, a directional light, and a moving point light source.

Figure 13.1. Basic Impostor



The camera movement is controlled in the same way as previous tutorials. The **T** key will toggle a display showing the look-at point. The **-** and **=** keys will rewind and fast-forward the time, and the **P** key will toggle pausing of the time advancement.

The tutorial starts showing mesh spheres, to allow you to switch back and forth between actual meshes and impostor spheres. Each sphere is independently controlled:

Table 13.1. Sphere Impostor Control Key Map

Key	Sphere
1	The central blue sphere.
2	The orbiting grey sphere.

Key	Sphere
3	The black marble on the left.
4	The gold sphere on the right.

This tutorial uses a rendering setup similar to the last one. The shaders use uniform blocks to control most of the uniforms. There is a shared global lighting uniform block, as well as one for the projection matrix.

Grifting Geometry

The way this program actually renders the geometry for the impostors is interesting. The vertex shader looks like this:

Example 13.1. Basic Impostor Vertex Shader

```
#version 330

layout(std140) uniform;

out vec2 mapping;

uniform Projection
{
 mat4 cameraToClipMatrix;
};

uniform float sphereRadius;
uniform vec3 cameraSpherePos;

void main()
{
    vec2 offset;
    switch(gl_VertexID)
    {
    case 0:
        //Bottom-left
        mapping = vec2(-1.0, -1.0);
        offset = vec2(-sphereRadius, -sphereRadius);
        break;
    case 1:
        //Top-left
        mapping = vec2(-1.0, 1.0);
        offset = vec2(-sphereRadius, sphereRadius);
        break;
    case 2:
        //Bottom-right
        mapping = vec2(1.0, -1.0);
        offset = vec2(sphereRadius, -sphereRadius);
        break;
    case 3:
        //Top-right
        mapping = vec2(1.0, 1.0);
        offset = vec2(sphereRadius, sphereRadius);
        break;
    }

    vec4 cameraCornerPos = vec4(cameraSpherePos, 1.0);
```



```
    cameraCornerPos.xy += offset;

    gl_Position = cameraToClipMatrix * cameraCornerPos;
}
```

Notice anything missing? There are no input variables declared anywhere in this vertex shader.

It does still use an input variable: `gl_VertexID`. This is a built-in input variable; it contains the current index of this particular vertex. When using array rendering, it's just the count of the vertex we are in. When using indexed rendering, it is the index of this vertex.

When we render this mesh, we render 4 vertices as a `GL_TRIANGLE_STRIP`. This is rendered in array rendering mode, so the `gl_VertexID` will vary from 0 to 3. Our switch/case statement determines which vertex we are rendering. Since we're trying to render a square with a triangle strip, the order of the vertices needs to be appropriate for this.

After computing which vertex we are trying to render, we use the radius-based offset as a bias to the camera-space sphere position. The Z value of the sphere position is left alone, since it will always be correct for our square. After that, we transform the camera-space position to clip-space as normal.

The output `mapping` is a value that is used by the fragment shader, as we will see below.

Since this vertex shader takes no inputs, our vertex array object does not need to contain anything either. That is, we never call `glEnableVertexAttribArray` on the VAO. Since no attribute arrays are enabled, we also have no need for a buffer object to store vertex array data. So we never call `glVertexAttribPointer`. We simply generate an empty VAO with `glGenVertexArrays` and use it without modification.

Racketeering Rasterization

Our lighting equations in the past needed only a position and normal in camera-space (as well as other material and lighting parameters) in order to work. So the job of the fragment shader is to provide them. Even though they do not correspond to those of the actual triangles in any way.

Here are the salient new parts of the fragment shader for impostors:

Example 13.2. Basic Impostor Fragment Shader

```
in vec2 mapping;

void Impostor(out vec3 cameraPos, out vec3 cameraNormal)
{
    float lensqr = dot(mapping, mapping);
    if(lensqr > 1.0)
        discard;

    cameraNormal = vec3(mapping, sqrt(1.0 - lensqr));
    cameraPos = (cameraNormal * sphereRadius) + cameraSpherePos;
}

void main()
{
    vec3 cameraPos;
    vec3 cameraNormal;

    Impostor(cameraPos, cameraNormal);

    vec4 accumLighting = Mtl.diffuseColor * Lgt.ambientIntensity;
    for(int light = 0; light < numberOfLights; light++)
    {
```

```
        accumLighting += ComputeLighting(Lgt.lights[light],
            cameraPos, cameraNormal);
    }

    outputColor = sqrt(accumLighting); //2.0 gamma correction
}
```

In order to compute the position and normal, we first need to find the point on the sphere that corresponds with the point on the square that we are currently on. And to do that, we need a way to tell where on the square we are.

Using `gl_FragCoord` will not help, as it is relative to the entire screen. We need a value that is relative only to the impostor square. That is the purpose of the `mapping` variable. When this variable is at (0, 0), we are in the center of the square, which is the center of the sphere. When it is at (-1, -1), we are at the bottom left corner of the square.

Given this, we can now compute the sphere point directly "above" the point on the square, which is the job of the `Impostor` function.

Before we can compute the sphere point however, we must make sure that we are actually on a point that has the sphere above it. This requires only a simple distance check. Since the size of the square is equal to the radius of the sphere, if the distance of the `mapping` variable from its (0, 0) point is greater than 1, then we know that this point is off of the sphere.

Here, we use a clever way of computing the length; we do not. Instead, we compute the square of the length. We know that if $X^2 > Y^2$ is true, then $X > Y$ must also be true for all positive real numbers X and Y. So we just do the comparison as squares, rather than taking a square-root to find the true length.

If the point is not under the sphere, we execute something new: `discard`. The `discard` keyword is unique to fragment shaders. It tells OpenGL that the fragment is invalid and its data should not be written to the image or depth buffers. This allows us to carve out a shape in our flat square, turning it into a circle.

A Word on Discard

Using `discard` sounds a lot like throwing an exception. Since the fragment's outputs will be ignored and discarded, you might expect that executing this instruction will cause the fragment shader to stop executing. This is not necessarily the case.

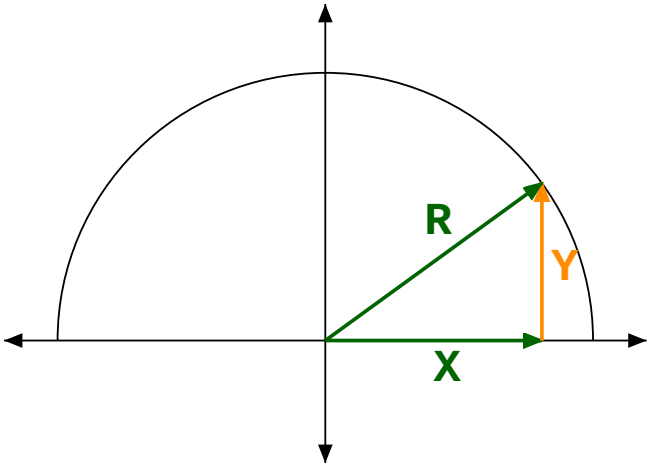
Due to the way that shaders tend to work, multiple executions of the same shader are often operating at the same time. All of them are running in lock-step with one another; they all execute instructions at the same time, just on different datasets. If one of them does a discard, it still has to keep doing what it was doing, because the other three may not have discarded, since the discard was based on data that may be different between each shader. This is also why branches in shaders will often execute both sides rather than actually branching; it keeps the shader logic simpler.

However, that does not mean `discard` is without use for stopping unwanted processing. If all of the shaders that are running together hit a `discard`, then they can all be aborted with no problems. And hardware often does this where possible. So if there is a great deal of spatial coherency with discard, this is useful.

The computation of the normal is based on simple trigonometry. The normal of a sphere does not change based on the sphere's radius. Therefore, we can compute the normal in the space of the mapping, which uses a normalized sphere radius of 1. The normal of a sphere at a point is in the same direction as the direction from the sphere's center to that point on the surface.

Let's look at the 2D case. To have a 2D vector direction, we need an X and Y coordinate. If we only have the X, but we know that the vector has a certain length, then we can compute the Y component of the vector based on the Pythagorean theorem:

Figure 13.2. Circle Point Computation



$$X^2 + Y^2 = R^2$$
$$Y = \pm\sqrt{R^2 - X^2}$$

We simply use the 3D version of this. We have X and Y from `mapping`, and we know the length is 1.0. So we compute the Z value easily enough. And since we are only interested in the front-side of the sphere, we know that the Z value must be positive.

Computing the position is also easy. The position of a point on the surface of a sphere is the normal at that position scaled by the radius and offset by the center point of the sphere.

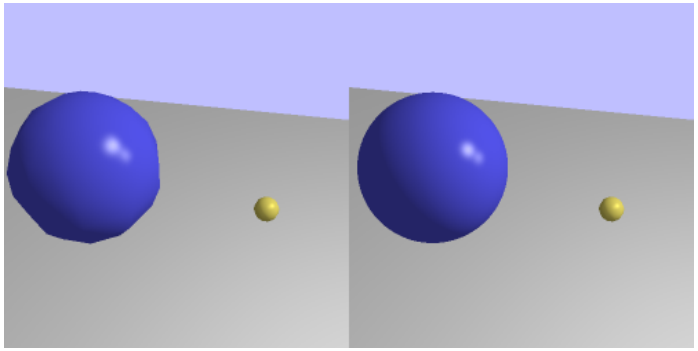
One final thing. Notice the square-root at the end, being applied to our accumulated lighting. This effectively simulates a gamma of 2.0, but without the expensive `pow` function call. A `sqrt` call is much less expensive and far more likely to be directly built into the shader hardware. Yes, this is not entirely accurate, since most displays simulate the 2.2 gamma of CRT displays. But it's a lot less inaccurate than applying no correction at all. We'll discuss a much cheaper way to apply proper gamma correction in future tutorials.

Correct Chicanery

Our perfect sphere looks pretty nice. It has no polygonal outlines and you can zoom in on it forever. However, it is unfortunately very wrong.

To see how, toggle back to rendering the mesh on sphere **1** (the central blue one). Then move the camera so that the sphere is at the left edge of the screen. Then toggle back to impostor rendering.

Figure 13.3. Bad Impostor



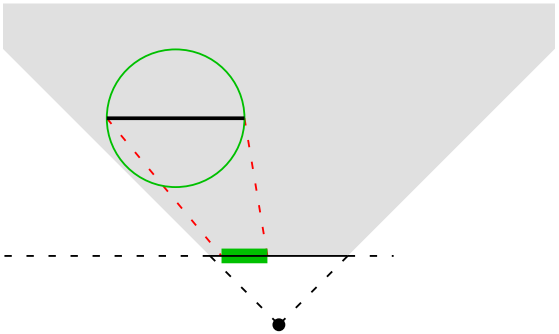
What's going on here? The mesh sphere seems to be wider than the impostor sphere. This must mean that the mesh sphere is doing something our impostor is not. Does this have to do with the inaccuracy of the mesh sphere?

Quite the opposite, in fact. The mesh sphere is correct. The problem is that our impostor is too simple.

Look back at how we did our computations. We map a sphere down to a flat surface. The problem is that “down” in this case is in the camera-space Z direction. The mapping between the surface and the sphere is static; it does not change based on the viewing angle.

Consider this 2D case:

Figure 13.4. Circle Projection



The dark line through the circle represents the square we drew. When viewing the sphere off to the side like this, we should not be able to see the left-edge of the sphere facing perpendicular to the camera. And we should see some of the sphere on the right that is behind the plane.

So how do we solve this?

Use better math. Our last algorithm is a decent approximation if the spheres are somewhat small. But if the spheres are reasonably large (which also can mean close to the camera), then our approximation is shown to be very fake. Our new algorithm needs to take this into account.

This algorithm is based on a term you may have heard before: *ray tracing*. We will not be implementing a full ray tracing algorithm here; instead, we will use it solely to get the position and normal of a sphere at a certain point.

A ray is a direction and a position; it represents a line extending from the position along that direction. The points on the ray can be expressed as the following equation:

Equation 13.1. Ray Equation

$\hat{D}$ = Ray Direction

$\vec{O}$ = Ray Origin

$\vec{P}(t) = \hat{D}t + \vec{O}$

The `t` value can be positive or negative, but for our needs, we'll stick with positive values.

For each fragment, we want to create a ray from the camera position in the direction towards that point on the impostor square. Then we want to detect the point on the sphere that it hits, if any. If the ray intersects the sphere, then we use that point and normal for our lighting equation.

The math for this is fairly simple. The equation for the points on a sphere is this:

Equation 13.2. Sphere Equation

R = Sphere Radius

$\vec{S}$ = Sphere Center

$\| \vec{P} - \vec{S} \| = R$

For any point P, if this equation is true, if the length between that point and the sphere's center equals the radius, then P is on the sphere. So we can substitute our ray equation for P:

$\| \hat{D}t + \vec{O} - \vec{S} \| = R$

Our ray goes from the camera into the scene. Since we're in camera space, the camera is at the origin. So O can be eliminated from the equation. To solve for `t`, we need to get rid of that length. One way to do it is to re-express the sphere equation as the length squared. So then we get:

$\| \vec{P} - \vec{S} \|^2 = R^2$

$\| \hat{D}t - \vec{S} \|^2 = R^2$

The square of the length of a vector is the same as that vector dot-producted with itself. So let's do that:

$(\hat{D}t - \vec{S}) \cdot (\hat{D}t - \vec{S}) = R^2$

The dot product is distributive. Indeed, it follows most of the rules of scalar multiplication. This gives us:

$(\hat{D} \cdot \hat{D})t^2 - 2(\hat{D} \cdot \vec{S})t + (\vec{S} \cdot \vec{S}) = R^2$

While this equation has a lot of vector elements in it, as far as t is concerned, it is a scalar equation. Indeed, it is a quadratic equation, with respect to t. Ah, good old algebra.

$Ax^2 + Bx + C = 0$

$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$

$A = \hat{D} \cdot \hat{D} = 1$

$B = -2(\hat{D} \cdot \vec{S})$

$C = (\vec{S} \cdot \vec{S}) - R^2$

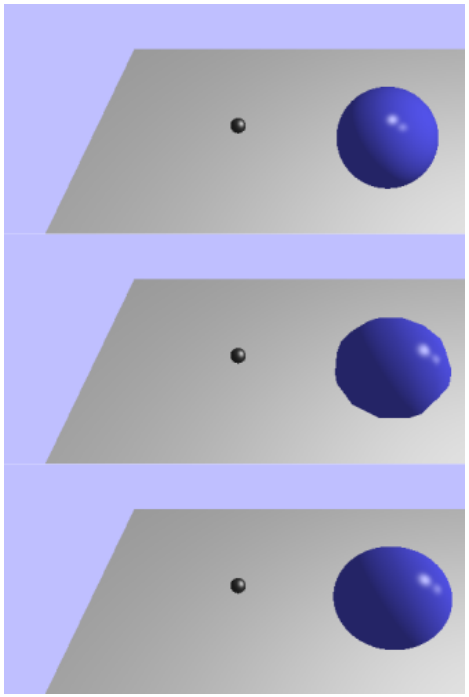
In case you've forgotten, the part under the square root in the quadratic formula is called the discriminant. If this value is negative, then the equation has no solution. In terms of our ray test, this means the ray misses the sphere.

As you may recall, the square root can be either positive or negative. This gives us two t values. Which makes sense; the ray hits the sphere in two places: once going in, and once coming out. The correct t value that we're interested in is the smallest one. Once we have that, we can use the ray equation to compute the point. With the point and the center of the sphere, we can compute the normal. And we're back in business.

Extorting and Expanding

To see this done, open up the last tutorial project. Since they use the exact same source, and since they use the same uniforms and other interfaces for their shaders, there was no need to make another code project for it. To see the ray-traced version, press the **J** key; all impostors will use the perspective version. To go back to the flat version, press **L**.

Figure 13.5. Bad vs. Good



The top is the original impostor, the middle is the actual mesh, and the bottom is our new ray traced impostor.

The `Impostor` function in the new fragment shader implements our ray tracing algorithm. More important than this are the changes to the vertex shader's computation of the impostor square:

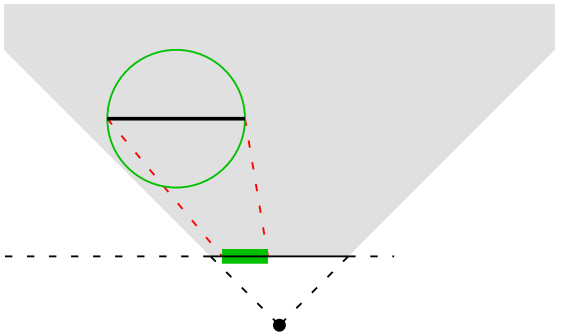
Example 13.3. Ray Traced Impostor Square

```
const float g_boxCorrection = 1.5;

void main()
```



```
{
    vec2 offset;
    switch(gl_VertexID)
    {
    case 0:
        //Bottom-left
        mapping = vec2(-1.0, -1.0) * g_boxCorrection;
        offset = vec2(-sphereRadius, -sphereRadius);
        break;
    case 1:
        //Top-left
        mapping = vec2(-1.0, 1.0) * g_boxCorrection;
        offset = vec2(-sphereRadius, sphereRadius);
        break;
    case 2:
        //Bottom-right
        mapping = vec2(1.0, -1.0) * g_boxCorrection;
        offset = vec2(sphereRadius, -sphereRadius);
        break;
    case 3:
        //Top-right
        mapping = vec2(1.0, 1.0) * g_boxCorrection;
        offset = vec2(sphereRadius, sphereRadius);
        break;
    }

    vec4 cameraCornerPos = vec4(cameraSpherePos, 1.0);
    cameraCornerPos.xy += offset * g_boxCorrection;

    gl_Position = cameraToClipMatrix * cameraCornerPos;
}
```

We have expanded the size of the square by 50%. What is the purpose of this? Well, let's look at our 2D image again.



The black line represents the square we used originally. There is a portion to the left of the projection that we should be able to see. However, with proper ray tracing, it would not fit onto the area of the radius-sized square.

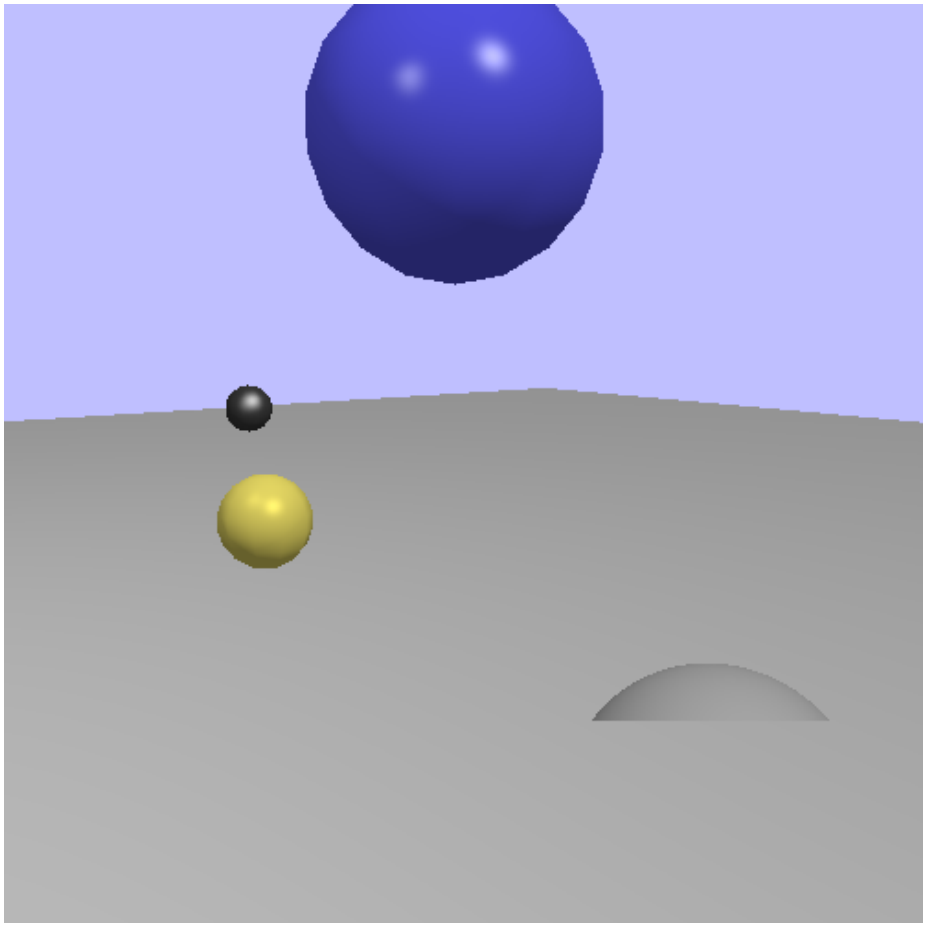
This means that we need to expand the size of the square. Rather than finding a clever way to compute the exact extent of the sphere's area projected onto a square, it's much easier to just make the square bigger. This is even moreso considering that such math would have to take into

account things like the viewport and the perspective matrix. Sure, we will end up running the rasterizer rather more than strictly necessary. But it's overall much simpler.

Deceit in Depth

While the perspective version looks great, there remains one problem. Move the time around until the rotating grey sphere ducks underneath the ground.

Figure 13.6. Bad Intersection



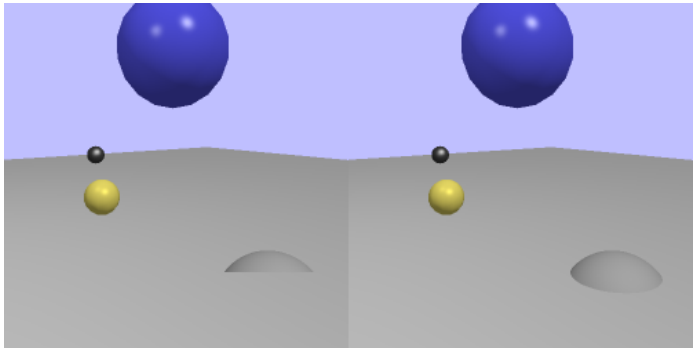
Hmm. Even though we've made it look like a mathematically perfect sphere, it does not act like one to the depth buffer. As far as it is concerned, it's just a circle (remember: `discard` prevents depth writes and tests as well).

Is that the end for our impostors? Hardly.

Part of the fragment shader's output is a depth value. If you do not write one, then OpenGL will happily use `gl_FragCoord.z` as the depth output from the fragment shader. This value will be depth tested against the current depth value and, if the test passes, written to the depth buffer.

But we do have the ability to write a depth value ourselves. To see how this is done, load up the tutorial (using the same code again) and press the **H** key. This will cause all impostors to use depth-correct shaders.

Figure 13.7. Depth Correct Impostor



This shader is identical to the ray traced version, except for these lines in the fragment shader:

Example 13.4. Depth Correct Fragment Shader

```
Impostor(cameraPos, cameraNormal);

//Set the depth based on the new cameraPos.
vec4 clipPos = cameraToClipMatrix * vec4(cameraPos, 1.0);
float ndcDepth = clipPos.z / clipPos.w;
gl_FragDepth = ((gl_DepthRange.diff * ndcDepth) +
    gl_DepthRange.near + gl_DepthRange.far) / 2.0;
```

Basically, we go through the process OpenGL normally goes through to compute the depth. We just do it on the camera-space position we computed with the ray tracing function. The position is transformed to clip space. The perspective division happens, transforming to normalized device coordinate (NDC) space. The depth range function is applied, forcing the [-1, 1] range in the fragment shader to the range that the user provided with `glDepthRange.`

We write the final depth to a built-in output variable `gl_FragDepth.`

Fragments and Depth

The default behavior of OpenGL is, if a fragment shader does not write to the output depth, then simply take the `gl_FragCoord.z` depth as the depth of the fragment. Oh, you could do this manually. One could add the following statement to any fragment shader that uses the default depth value:

```
gl_FragDepth = gl_FragCoord.z
```

This is, in terms of behavior a noop; it does nothing OpenGL would not have done itself. However, in terms of *performance*, this is a drastic change.

The reason fragment shaders are not required to have this line in all of them is to allow for certain optimizations. If the OpenGL driver can see that you do not set `gl_FragDepth` anywhere in the fragment shader, then it can dramatically improve performance in certain cases.

If the driver knows that the output fragment depth is the same as the generated one, it can do the whole depth test *before* executing the fragment shader. This is called *early depth test* or *early-z*. This means that it can discard fragments *before* wasting precious time executing potentially complex fragment shaders. Indeed, most hardware nowadays has complicated early z culling hardware that can discard multiple fragments with a single test.

The moment your fragment shader writes anything to `gl_FragDepth`, all of those optimizations have to go away. So generally, you should only write a depth value yourself if you *really* need to do it.

Also, if your shader writes `gl_FragDepth` anywhere, it must ensure that it is *always* written to, no matter what conditional branches your shader uses. The value is not initialized to a default; you either always write to it or never mention "`gl_FragDepth`" in your fragment shader at all. Obviously, you do not always have to write the same value; you can conditionally write different values. But you cannot write something in one path and not write something in another. Initialize it explicitly with `gl_FragCoord.z` if you want to do something like that.

Purloined Primitives

Our method of rendering impostor spheres is very similar to our method of rendering mesh spheres. In both cases, we set uniforms that define the sphere's position and radius. We bind a material uniform buffer, then bind a VAO and execute a draw command. We do this for each sphere.

However, this seems rather wasteful for impostors. Our per-vertex data for the impostor is really the position and the radius. If we could somehow send this data 4 times, once for each square, then we could simply put all of our position and radius values in a buffer object and render every sphere in one draw call. Of course, we would also need to find a way to tell it which material to use.

We accomplish this task in the Geometry Impostor tutorial project. It looks exactly the same as before; it always draws impostors, using the depth-accurate shader.

Impostor Interleaving

To see how this works, we will start from the front of the rendering pipeline and follow the data. This begins with the buffer object and vertex array object we use to render.

Example 13.5. Impostor Geometry Creation

```
glBindBuffer(GL_ARRAY_BUFFER, g_imposterVBO);
glBufferData(GL_ARRAY_BUFFER, NUMBER_OF_SPHERES * 4 * sizeof(float), NULL, GL_STREAM_DRAW);

glGenVertexArrays(1, &g_imposterVAO);
glBindVertexArray(g_imposterVAO);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(0));
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(12));
```



```
glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

This code introduces us to a new feature of `glVertexAttribPointer`. In all prior cases the fifth parameter was 0. Now it is `4 * sizeof(float)`. What does this parameter mean?

This parameter is the array's `stride`. It is the number of bytes from one value for this attribute to the next in the buffer. When this parameter is 0, that means that the actual stride is the size of the base type (`GL_FLOAT` in our case) times the number of components. When the stride is non-zero, it must be larger than that value.

What this means for our vertex data is that the first 3 floats represent attribute 0, and the next float represents attribute 1. The next 3 floats is attribute 0 of the next vertex, and the float after that is attribute 1 of that vertex. And so on.

Arranging attributes of the same vertex alongside one another is called *interleaving*. It is a very useful technique; indeed, for performance reasons, data should generally be interleaved where possible. One thing that it allows us to do is build our vertex data based on a struct:

```
struct VertexData
{
    glm::vec3 cameraPosition;
    float sphereRadius;
};
```

Our vertex array object perfectly describes the arrangement of data in an array of `VertexData` objects. So when we upload our positions and radii to the buffer object, we simply create an array of these structs, fill in the values, and upload them with `glBufferData`.

Misnamed and Maligned

So, our vertex data now consists of a position and a radius. But we need to draw four vertices, not one. How do we do that?

We could replicate each vertex data 4 times and use some simple `gl_VertexID` math in the vertex shader to figure out which corner we're using. Or we could get complicated and learn something new. That new thing is an entirely new programmatic shader stage: *geometry shaders*.

Our initial pipeline discussion ignored this shader stage, because it is an entirely optional part of the pipeline. If a program object does not contain a geometry shader, then OpenGL just does its normal stuff.

The most confusing thing about geometry shaders is that they do not shade geometry. Vertex shaders take a vertex as input and write a vertex as output. Fragment shader take a fragment as input and potentially writes a fragment as output. Geometry shaders take a *primitive* as input and write zero or more primitives as output. By all rights, they should be called "primitive shaders."

In any case, geometry shaders are invoked just after the hardware that collects vertex shader outputs into a primitive, but before any clipping, transforming or rasterization happens. Geometry shaders get the values output from multiple vertex shaders, performs arbitrary computations on them, and outputs one or more sets of values to new primitives.

In our case, the logic begins with our drawing call:

```
glBindVertexArray(g_imposterVAO);
glDrawArrays(GL_POINTS, 0, NUMBER_OF_SPHERES);
glBindVertexArray(0);
```

This introduces a completely new primitive and primitive type: `GL_POINTS`. Recall that multiple primitives can have the same base type. `GL_TRIANGLE_STRIP` and `GL_TRIANGLES` are both separate primitives, but both generate triangles. `GL_POINTS` does not generate triangle primitives; it generates point primitives.

`GL_POINTS` interprets each individual vertex as a separate point primitive. There are no other forms of point primitives, because points only contain a single vertex worth of information.

The vertex shader is quite simple, but it does have some new things to show us:

Example 13.6. Vertex Shader for Points

```
#version 330
```

```
layout(location = 0) in vec3 cameraSpherePos;
layout(location = 1) in float sphereRadius;

out VertexData
{
    vec3 cameraSpherePos;
    float sphereRadius
} outData;

void main()
{
 outData.cameraSpherePos = cameraSpherePos;
    outData.sphereRadius = sphereRadius;
}
```

`VertexData` is not a struct definition, though it does look like one. It is an *interface block* definition. Uniform blocks are a kind of interface block, but inputs and outputs can also have interface blocks.

An interface block used for inputs and outputs is a way of collecting them into groups. One of the main uses for these is to separate namespaces of inputs and outputs using the interface name (`outData`, in this case). This allows us to use the same names for inputs as we do for their corresponding outputs. They do have other virtues, as we will soon see.

Do note that this vertex shader does not write to `gl_Position.` That is not necessary when a vertex shader is paired with a geometry shader.

Speaking of which, let's look at the global definitions of our geometry shader.

Example 13.7. Geometry Shader Definitions

```
#version 330
#extension GL_EXT_gpu_shader4 : enable

layout(std140) uniform;
layout(points) in;
layout(triangle_strip, max_vertices=4) out;

uniform Projection
{
    mat4 cameraToClipMatrix;
};

in VertexData
{
    vec3 cameraSpherePos;
    float sphereRadius;
} vert[];

out FragData
{
    flat vec3 cameraSpherePos;
    flat float sphereRadius;
    smooth vec2 mapping;
};
```

Note

The `#extension` line exists to fix a compiler bug for NVIDIA's OpenGL. It should not be necessary.

We see some new uses of the `layout` directive. The `layout(points) in` command is geometry shader-specific. It tells OpenGL that this geometry shader is intended to take point primitives. This is required; also, OpenGL will fail to render if you try to draw something other than `GL_POINTS` through this geometry shader.

Similarly, the output layout definition states that this geometry shader outputs triangle strips. The `max_vertices` directive states that we will write at most 4 vertices. There are implementation defined limits on how large `max_vertices` can be. Both of these declarations are required for geometry shaders.

Below the `Projection` uniform block, we have two interface blocks. The first one matches the definition from the vertex shader, with two exceptions. It has a different interface name. But that interface name also has an array qualifier on it.

Geometry shaders take a primitive. And a primitive is defined as some number of vertices in a particular order. The input interface blocks define what the input vertex data is, but there is more than one set of vertex data. Therefore, the interface blocks must be defined as arrays. Granted, in our case, it is an array of length 1, since point primitives have only one vertex. But this is still necessary even in that case.

We also have another output fragment block. This one matches the definition from the fragment shader, as we will see a bit later. It does not have an instance name. Also, note that several of the values use the `flat` qualifier. We could have just used `smooth`, since we're passing the same values for all of the triangles. However, it's more descriptive to use the `flat` qualifier for values that are not supposed to be interpolated. It might even save performance.

Here is the geometry shader code for computing one of the vertices of the output triangle strip:

Example 13.8. Geometry Shader Vertex Computation

```
//Bottom-left
mapping = vec2(-1.0, -1.0) * g_boxCorrection;
cameraSpherePos = vec3(vert[0].cameraSpherePos);
sphereRadius = vert[0].sphereRadius;
cameraCornerPos = vec4(vert[0].cameraSpherePos, 1.0);
cameraCornerPos.xy += vec2(-vert[0].sphereRadius, -vert[0].sphereRadius) * g_boxCorrection;
gl_Position = cameraToClipMatrix * cameraCornerPos;
gl_PrimitiveID = gl_PrimitiveIDIn;
EmitVertex();
```

This code is followed by three more of these, using different mapping and offset values for the different corners of the square. The `cameraCornerPos` is a local variable that is re-used as temporary storage.

To output a vertex, write to each of the output variables. In this case, we have the three from the output interface block, as well as the built-in variables `gl_Position` and `gl_PrimitiveID` (which we will discuss more in a bit). Then, call `EmitVertex()`; this causes all of the values in the output variables to be transformed into a vertex that is sent to the output primitive type. After calling this function, the contents of those outputs are undefined. So if you want to use the same value for multiple vertices, you have to store the value in a different variable or recompute it.

Note that clipping, face-culling, and all of that stuff happens after the geometry shader. This means that we must ensure that the order of our output positions will be correct given the current winding order.

`gl_PrimitiveIDIn` is a special input value. Much like `gl_VertexID` from the vertex shader, `gl_PrimitiveIDIn` represents the current primitive being processed by the geometry shader (once more reason for calling it a primitive shader). We write this to the built-in output `gl_PrimitiveID`, so that the fragment shader can use it to select which material to use.

And speaking of the fragment shader, it's time to have a look at that.

Example 13.9. Fragment Shader Changes

```
in FragData
{
    flat vec3 cameraSpherePos;
    flat float sphereRadius;
    smooth vec2 mapping;
```

```
};

out vec4 outputColor;

layout(std140) uniform;

struct MaterialEntry
{
    vec4 diffuseColor;
    vec4 specularColor;
    vec4 specularShininess;        //ATI Array Bug fix. Not really a vec4.
};

const int NUMBER_OF_SPHERES = 4;

uniform Material
{
    MaterialEntry material[NUMBER_OF_SPHERES];
} Mtl;
```

The input interface is just the mirror of the output from the geometry shader. What's more interesting is what happened to our material blocks.

In our original code, we had an array of uniform blocks stored in a single uniform buffer in C++. We bound specific portions of this material block when we wanted to render with a particular material. That will not work now that we are trying to render multiple spheres in a single draw call.

So, instead of having an array of uniform blocks, we have a uniform block that *contains* an array. We bind all of the materials to the shader, and let the shader pick which one it wants as needed. The source code to do this is pretty straightforward.

Note

Notice that the material `specularShininess` became a vec4 instead of a simple float. This is due to an unfortunate bug in ATI's OpenGL implementation.

As for how the material selection happens, that's simple. In our case, we use the primitive identifier. The `gl_PrimitiveID` value written from the vertex shader is used to index into the `Mtl.material[]` array.

Do note that uniform blocks have a maximum size that is hardware-dependent. If we wanted to have a large palette of materials, on the order of several thousand, then we may exceed this limit. At that point, we would need an entirely new way to handle this data. Once that we have not learned about yet.

Or we could just split it up into multiple draw calls instead of one.

In Review

In this tutorial, you have learned the following:

- Impostors are objects who's geometric representation has little or no resemblance to what the viewer sees. These typically generate an object procedurally by cutting fragments out to form a shape, and then use normals to do lighting computations on the cut-out.
- Fragments can be discarded from within a fragment shader. This prevents the outputs from the shader from being written to the final image.
- Ray tracing can be employed by a fragment shader to determine the position and normal for a point. Those values can be fed into the lighting equation to produce a color value.
- Fragment shaders can change the depth value that is used for the depth test and is written to the framebuffer.
- Geometry shaders are a shader stage between the vertex shader and the rasterizer. They take a primitive as input and return one or more primitives as output.

Further Study

Try doing these things with the given programs.

- The first version of our impostors was a sphere approximation. It was not useful for relatively large spheres, but it could be useful for small ones. However, that approximation did not compute the depth of the fragment correctly. Make a version of it that does.
- Change the geometry impostor tutorial to take another vertex input: the material to use. The vertex shader should pass it along to the geometry shader, and the geometry shader should hand it to the fragment shader. You can still use `gl_PrimitiveID` as the way to tell the fragment shader. Regardless of how you send it, you will need to convert the value to an integer at some point. That can be done with this constructor-like syntax: `int(value_to_convert)`.

Further Research

This is an introduction to the concept of impostors. Indeed, the kind of ray tracing that we did has often been used to render more complex shapes like cylinders or quadratic surfaces. But impostors are capable of much, much more.

In effect, impostors allow you to use the fragment shader to just draw stuff to an area of the screen. They can be used to rasterize perfect circles, rather than drawing line-based approximations. Some have even used them to rasterize Bézier curves perfectly.

There are other impostor-based solutions. Most particle systems (a large and vibrant topic that you should investigate) use flat-cards to draw pictures that move through space. These images can animate, changing from one image to another based on time, and large groups of these particles can be used to simulate various phenomena like smoke, fire, and the like.

All of these subjects are worthy of your time. Of course, moving pictures through space requires being able to draw pictures. That means textures, which coincidentally is the topic for our next section.

GLSL Features of Note

discard	This fragment shader-only directive will cause the outputs of the fragment to be ignored. The fragment outputs, including the implicit depth, will not be written to the framebuffer.
gl_VertexID	An input to the vertex shader of type int. This is the index of the vertex being processed.
gl_FragDepth	An output from the fragment shader of type float. This value represents the depth of the fragment. If the fragment shader does not use this value in any way, then the depth will be written automatically, using `gl_FragCoord.z`. If the fragment shader writes to it somewhere, then it must ensure that *all* codepaths write to it.
gl_PrimitiveID	A geometry shader output and the corresponding fragment shader input of type int. If there is no geometry shader, then this value will be the current count of primitives that was previously rendered in this draw call. If there is a geometry shader, but it does not write to this value, then the value will be undefined.
gl_PrimitiveIDin	A geometry shader input. It is the current count of primitives previously processed in this draw call.

`void` **`EmitVertex`**`();`

Available on in the geometry shader, when this function is called, all output variables previously set by the geometry shader are consumed and transformed into a vertex. The value of those variables becomes undefined after calling this function.

Glossary

billboard, flat card	Terms used to describe the actual geometry used for impostors that are based on rendering camera-aligned shapes.
impostor	Any object who's geometry does not even superficially resemble the final rendered product. In these cases, the mesh geometry is usually just a way to designate an area of the screen to draw to, while the fragment shader does the real work.

ray tracing	For the purposes of this book, ray tracing is a technique whereby a mathematical object is tested against a ray (direction + position), to see if the ray intersects the object. At the point of intersection, one can generate a normal. With a position and normal in hand, one can use lighting equations to produce an image.
early depth test, early-z	An optimization in the depth test, where the incoming fragment's depth value is tested *before* the fragment shader executes. If the fragment shader is long, this can save a great deal of time. If the fragment shader exercises the option to modify or replace the fragment's depth, then the early depth test optimization will not be active.
interleaving	A way of storing vertex attributes in a buffer object. This involves entwining the attribute data, so that most or all of each vertex's attributes are spatially adjacent in the buffer object. This is as opposed to giving each vertex attribute its own array. Interleaving can lead to higher rendering performance in vertex transfer limited cases.
geometry shaders	A programmable stage between the vertex shader and the clipping/rasterization state. Geometry shaders take a primitive of a certain type as input, and returns zero or more primitives of a possibly different type as output. The vertex data taken as input does not have to match the vertex data taken as output, but the geometry shader's output interface must match that of the fragment shader's input interface.
interface block	A ordered grouping of uniforms, shader inputs or shader outputs. When used with uniforms, these are called uniform blocks. These are useful for name scoping, so that inputs and outputs can use the name that is most convenient and descriptive.

Part IV. Texturing

If you are at all familiar with 3D graphics, you have probably heard the term “texture” before. And if you look at virtually any instruction material on 3D graphics, they will introduce textures in the earliest parts of the work. Typically, this happens well before lighting is introduced.

This book is approximately halfway over and only now do we introduce textures. There is a good reason for this.

Consider everything you have learned up until now. You have learned how to transfer arbitrary data to vertex shaders, how to pass them to fragment shaders, and how to compute colors from them. You have learned how to transform positions of triangles and use this ability to provide a perspective projection of a world as well as to position objects and have a mobile camera. You have learned how lighting works and how to generate a lighting model. In the very last tutorial, we were able to convincingly render a mathematically perfect representation of a sphere simply by rendering two triangles.

All of this has been done without textures. Thus, the first lesson this book has to teach you about textures is that they are not *that* important. What you have learned is how to think about solving graphics problems without textures.

Many graphics texts overemphasize the importance of textures. This is mostly a legacy of the past. In the older days, before the availability of real programmable hardware, you needed textures to do anything of real importance in graphics rendering. Textures were used to simulate lighting and various other effects. If you wanted to do anything like per-fragment lighting, you had to use textures to do it.

Yes, textures are important for creating details in rendered images. They are important for being able to vary material parameters over a polygonal surface. And they have value in other areas as well. But there is so much more to rendering than textures, and this is especially true with programmable hardware.

A texture is a look-up table; an array. There is a lot of minutiae 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 important lesson you could learn is that textures are tools. Use them where appropriate, but do not let them become your primary solution to any rendering problem.

Chapter 14. Textures are not Pictures

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, let us define a problem that textures can solve without having to be pictures.

We have seen that the Gaussian specular function is a pretty useful specular function. Its shininess value has a nice range (0, 1], and it produces pretty good results visually. It has fewer artifacts than the less complicated Blinn-Phong function. But there is one significant problem: Gaussian is much more expensive to compute. Blinn-Phong requires a single power-function; Gaussian requires not only exponentiation, but also an inverse-cosine function. This is in addition to other operations like squaring the exponent.

Let us say that we have determined that the Gaussian specular function is good but too expensive for our needs.[1] So we want to find a way to get the equivalent quality of Gaussian specular but with more performance. What are our options?

A common tactic in optimizing math functions is a *look-up table*. These are arrays of some dimensionality that represents a function. For any function F(x), where x is valid over some range [a, b], you can define a table that stores the results of the function at various points along the valid range of x. Obviously if x has an infinite range, there is a problem. But if x has a finite range, one can decide to take some number of values on that range and store them in a table.

The obvious downside of this approach is that the quality you get depends on how large this table is. That is, how many times the function is evaluated and stored in the table.

The Gaussian specular function takes three parameters: the surface normal, the half-angle vector, and the specular shininess of the surface. However, if we redefine the specular function in terms of the dot-product of the surface normal and the half-angle vector, we can reduce the number of parameters to two. Also, the specular shininess is a constant value across a mesh. So, for any given mesh, the specular function is a function of one parameter: the dot-product between the half-angle vector and the surface normal.

Equation 14.1. Gaussian as Function of One Variable

$$F(d) = e^{-\left(\frac{d}{m}\right)^2}$$

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.

The First Texture

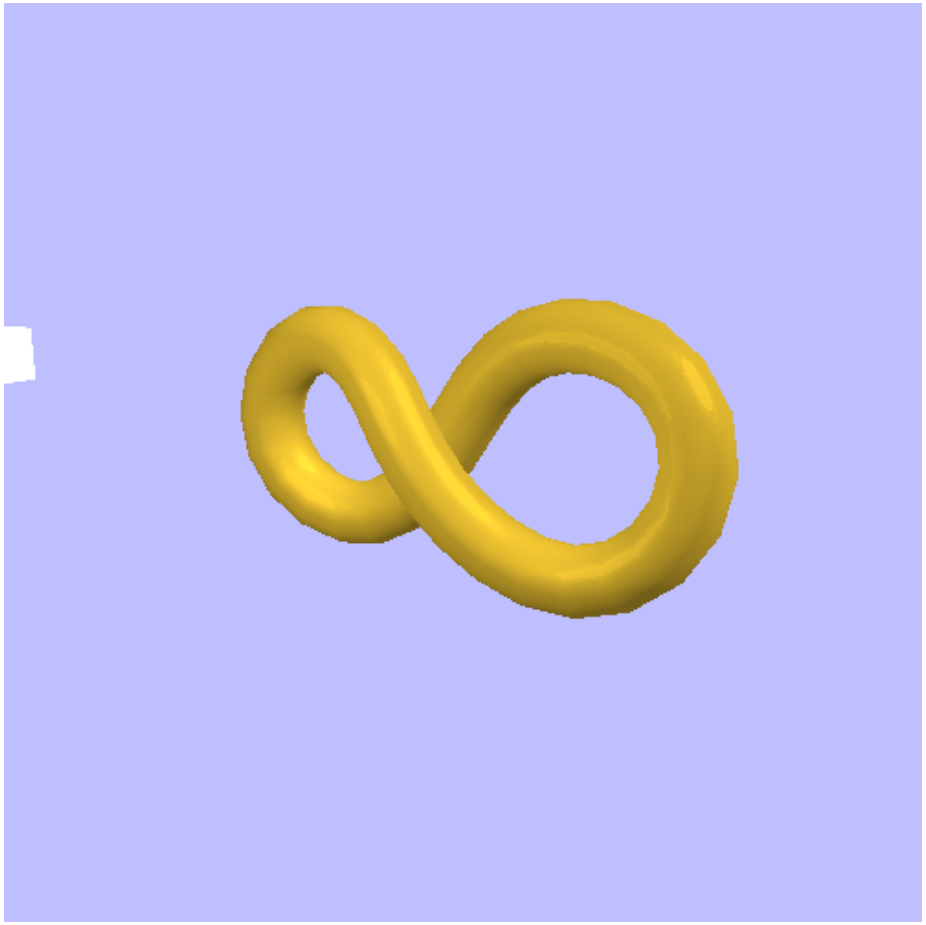
A *texture* is an object that contains one or more arrays of data, with all of the arrays having some dimensionality. The storage for a texture is owned by OpenGL and the GPU, much like they own the storage for buffer objects. Textures can be accessed in a shader, which fetches data from the texture 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.

Our first use of textures is in the Basic Texture tutorial. This tutorial shows a scene containing a golden infinity symbol, with a directional light and a second moving point light source.

[1] This is for demonstration purposes only. You should not undertake this process in the real world unless you have determined with proper profiling that the specular function is a performance problem that you should work to alleviate.

Figure 14.1. Basic Texture



The camera and the object can be rotated using the left and right mouse buttons respectively. Pressing the **Spacebar** toggles between shader-based Gaussian specular and texture-based specular. The **1** through **4** keys switch to progressively larger textures, so that you can see the effects that higher resolution look-up tables has on the visual result.

Normalized Integers

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. The function `BuildGaussianData` generates the data that we want to put into our OpenGL texture.

Example 14.1. BuildGaussianData function

```
void BuildGaussianData(std::vector<GLubyte> &textureData,
```

```
                        int cosAngleResolution)
{
    textureData.resize(cosAngleResolution);

    std::vector<GLubyte>::iterator currIt = textureData.begin();
    for(int iCosAng = 0; iCosAng < cosAngleResolution; iCosAng++)
    {
        float cosAng = iCosAng / (float)(cosAngleResolution - 1);
        float angle = acosf(cosAng);
        float exponent = angle / g_specularShininess;
        exponent = -(exponent * exponent);
        float gaussianTerm = glm::exp(exponent);

        *currIt++ = (GLubyte)(gaussianTerm * 255.0f);
    }
}
```

This function fills a `std::vector` with bytes that represents our lookup table. It's a pretty simple function. The parameter `cosAngleResolution` specifies the number of entries in the table. As we iterate over the range, we convert them into cosine values and then perform the Gaussian specular computations.

However, the result of this computation is a float, 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 `gaussianTerm` 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.

Texture Objects

The function `CreateGaussianTexture` calls `BuildGaussianData` to generate the array of normalized integers. The rest of that function uses the array to build the OpenGL texture object:

Example 14.2. CreateGaussianTexture function

```
GLuint CreateGaussianTexture(int cosAngleResolution)
{
    std::vector<GLubyte> textureData;
    BuildGaussianData(textureData, cosAngleResolution);

    GLuint gaussTexture;
    glGenTextures(1, &gaussTexture);
    glBindTexture(GL_TEXTURE_1D, gaussTexture);
    glTexImage1D(GL_TEXTURE_1D, 0, GL_R8, cosAngleResolution, 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. Note that once you have bound a texture to the context with a certain type, it must *always* be bound with that same 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. The first specifies the type of the currently bound texture. As with buffer objects, multiple textures can be bound to different texture type locations. So you could have a texture bound to `GL_TEXTURE_1D` and another boudn to `GL_TEXTURE_2D`. But it's really bad form to try to exploit this. It is best to just have one target bound at a time.

The second parameter is something we will talk about in the next tutorial. The third parameter is the format that OpenGL will use to store the texture's data. The fourth parameter 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 last three parameters of all functions of the form `glTexImage*` are special. They tell OpenGL how to read the texture data in our array. This seems redundant, since we already told OpenGL what the format of the data was with the third parameter. This bears further examination.

Textures and buffer objects have many similarities. They both represent memory owned by OpenGL. The user can modify this memory with various functions. Besides the fact that a texture object can contain multiple images, the major difference is the arrangement of data as it is stored by the GPU.

Buffer objects are linear arrays of memory. The data stored by OpenGL must be binary-identical to the data that the user specifies with `glBuffer(Sub)Data` calls. The format of the data stored in a buffer object is defined externally to the buffer object itself. Buffer objects used for vertex attributes have their formats defined by `glVertexAttribPointer`. The format for buffer objects that store uniform data is defined by the arrangement of types in a GLSL uniform block.

There are other ways that use buffer objects that allow OpenGL calls to fill them with data. But in all cases, the binary format of the data to be stored is very strictly controlled by the user. It is the *user's* responsibility to make sure that the data stored there uses the format that OpenGL was told to expect. Even when OpenGL itself is generating the data being stored in it.

Textures do not work this way. The format of an image stored in a texture is controlled by OpenGL itself. The user tells it what format to use, but the specific arrangements of bytes is up to OpenGL. This allows different hardware to store textures in whatever way is most optimal for accessing them.

Because of this, there is an intermediary between the data the user provides and the data that is actually stored in the texture. The data the user provides must be transformed into the format that OpenGL uses internally for the texture's data. Therefore, `glTexImage*` functions must specify both the expected internal format and a description of how the texture data is stored in the user's array.

Pixel Transfer and Formats. This process, the conversion between an image's internal format and a user-provided array, is called a *pixel transfer* operation. These are somewhat complex, but not too difficult to understand.

Each pixel in a texture is more properly referred to as a *texel*. Since texture data is accessed in OpenGL by the texel, we want our array of normalized unsigned integers to each be stored in a single texel. So our input data has only one value per texel, that value is 8-bits in size, and it represents an normalized unisgned integer.

The last three parameters describe this to OpenGL. The parameter `GL_RED` says that we are uploading a single component to the texture, namely the red component. Components of texels are named after color components. Because this parameter does not end in "_INTEGER", OpenGL knows that the data we are uploading is either a floating-point value or a normalized integer value (which converts to a float when accessed by the shader).

The parameter `GL_UNSIGNED_BYTE` says that each component that we are uploading is stored in an 8-bit unsigned byte. This, plus the pointer to the data, is all OpenGL needs to read our data.

That describes the data format as we are providing it. The format parameter, the third parameter to the `glTexImage*` functions, describes the format of the texture's internal storage. The texture's format defines the properties of the texels stored in that texture:

- The components stored in the texel. Multiple components can be used, but only certain combinations of components are allowed. The components include the RGBA of colors, and certain more exotic values we will discuss later.

- The number of bits that each component takes up when stored by OpenGL. Different components within a texel can have different bitdepths.

- The data type of the components. Certain exotic formats can give different components different types, but most of them give them each the same data type. Data types include normalized unsigned integers, floats, non-normalized signed integers, and so forth.

The parameter `GL_R8` defines all of these. The "R" represents the components that are stored. 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`.

Note that this perfectly matches the texture data that we generated. 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.

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 the next tutorial.

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. The restrictions on variables of sampler types are:

- Samplers can only declared at the global scope as `uniform` or in function parameter lists with the `in` qualifier. They cannot even be declared as local variables.

- Samplers cannot be members of structs or uniform blocks.

- Samplers can be used in arrays, but the index for sampler arrays must be a compile-time constant.

- Samplers do not have values. No mathematical expressions can use sampler variables.

- The only use of variables of sampler type is as parameters to functions. User-defined functions can take them as parameters, and there are a number of built-in functions that take samplers.

In the shader `TextureGaussian.frag`, we have an example of creating a sampler:

```
uniform sampler1D gaussianTexture;
```

This creates a sampler for a 1D texture type; the user cannot use any other type of texture with this sampler.

Texture Sampling. The process of fetching data from a texture, at a particular location, is called *sampling.* This is done in the shader as part of the lighting computation:

Example 14.3. Shader Texture Access

```
vec3 halfAngle = normalize(lightDir + viewDirection);
float texCoord = dot(halfAngle, surfaceNormal);
float gaussianTerm = texture(gaussianTexture, texCoord).r;
```



```
gaussianTerm = cosAngIncidence != 0.0 ? gaussianTerm : 0.0;
```

The third line is 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, 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.

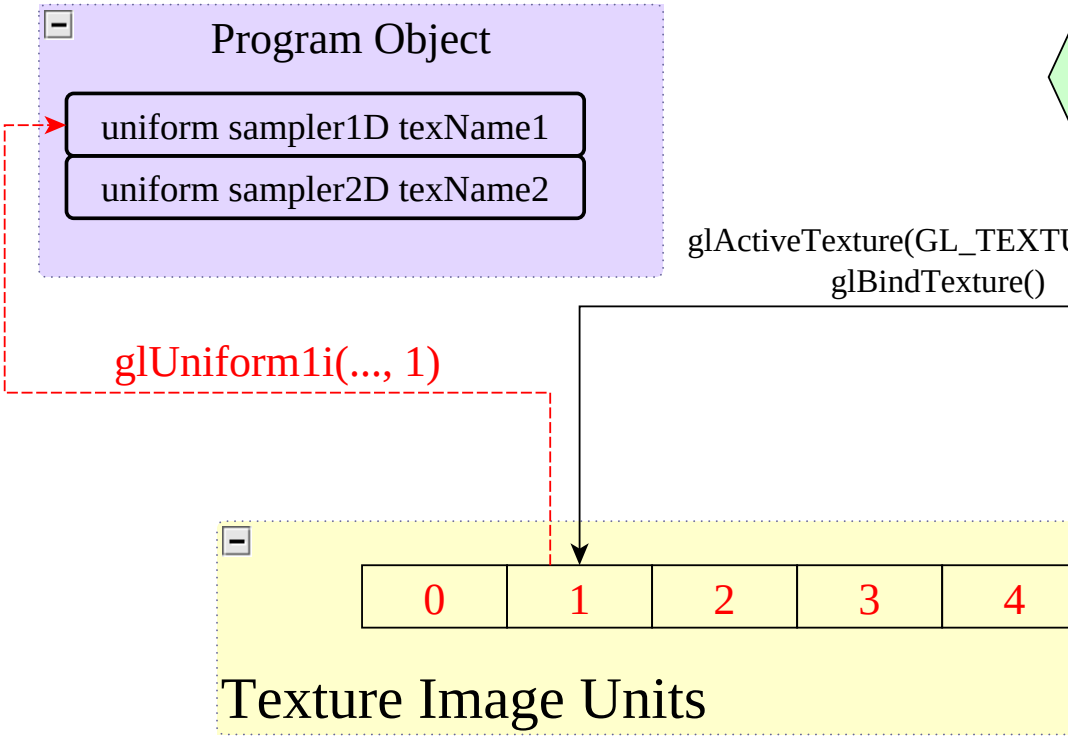
Texture Binding

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?

Although the API is slightly more obfuscated due to legacy issues, this association is made essentially the same way as with uniform buffer objects.

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.

Figure 14.2. Texture Binding and Context



Though the idea is essentially the same, there are many API differences between the UBO mechanism and the texture mechanism. We will start with setting the sampler uniform to an image unit.

With UBOs, this used a different API from regular uniforms. Because samplers are actual uniforms, the sampler API is just the uniform API:

```
GLuint gaussianTextureUnif = glGetUniformLocation(data.theProgram, "gaussianTexture");
glUseProgram(data.theProgram);
glUniform1i(gaussianTextureUnif, g_gaussTexUnit);
```

Sampler uniforms are considered 1-dimesional (scalar) integer values from the OpenGL side of the API. Do not forget that, in the GLSL side, samplers have no value at all.

When it comes time to bind the texture object to that image unit, OpenGL again overloads existing API rather than making a new one the way UBOs did:

```
glActiveTexture(GL_TEXTURE0 + g_gaussTexUnit);
glBindTexture(GL_TEXTURE_1D, g_gaussTextures[g_currTexture]);
```

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. To put it another way, with UBOs, it was possible

to bind a buffer object to `GL_UNIFORM_BUFFER` without overwriting any of the uniform buffer binding points. This is possible because there are two functions for buffer object binding: `glBindBuffer` which binds only to the target, and `glBindBufferRange` which binds to the target and an indexed location.

Texture units do not have this. There is one binding function, `glBindTexture`. And it always binds to whatever texture unit happens to be current. Namely, the one set by the last call to `glActiveTexture`.

What this means is that if you want to modify a texture, you must overwrite a texture unit that may already be bound. This is usually not a huge problem, because you rarely modify textures in the same area of code used to render. But you should be aware of this API oddity.

Also note the peculiar `glActiveTexture` syntax for specifying the image unit: `GL_TEXTURE0 + g_gaussTexUnit`. 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.

If you look at the rendering function, you will find that the texture will always be bound, even when not rendering with the texture. This is perfectly harmless; the contents of a texture image unit is ignored unless a program has a sampler uniform that is associated with that image unit.

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 in the `CreateGaussianTextures` function.

Example 14.4. Sampler Object Creation

```
glGenSamplers(1, &g_gaussSampler);
glSamplerParameteri(g_gaussSampler, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glSamplerParameteri(g_gaussSampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glSamplerParameteri(g_gaussSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

As with most OpenGL objects, we create a sampler object with `glGenSamplers`. However, notice something unusual with the next series of functions. We do not bind a sampler to the context to set parameters in it, nor does `glSamplerParameter` take a context target. We simply pass an object directly to the function.

In this above code, we set three parameters. The first two parameters are things we will discuss in the next tutorial. 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.

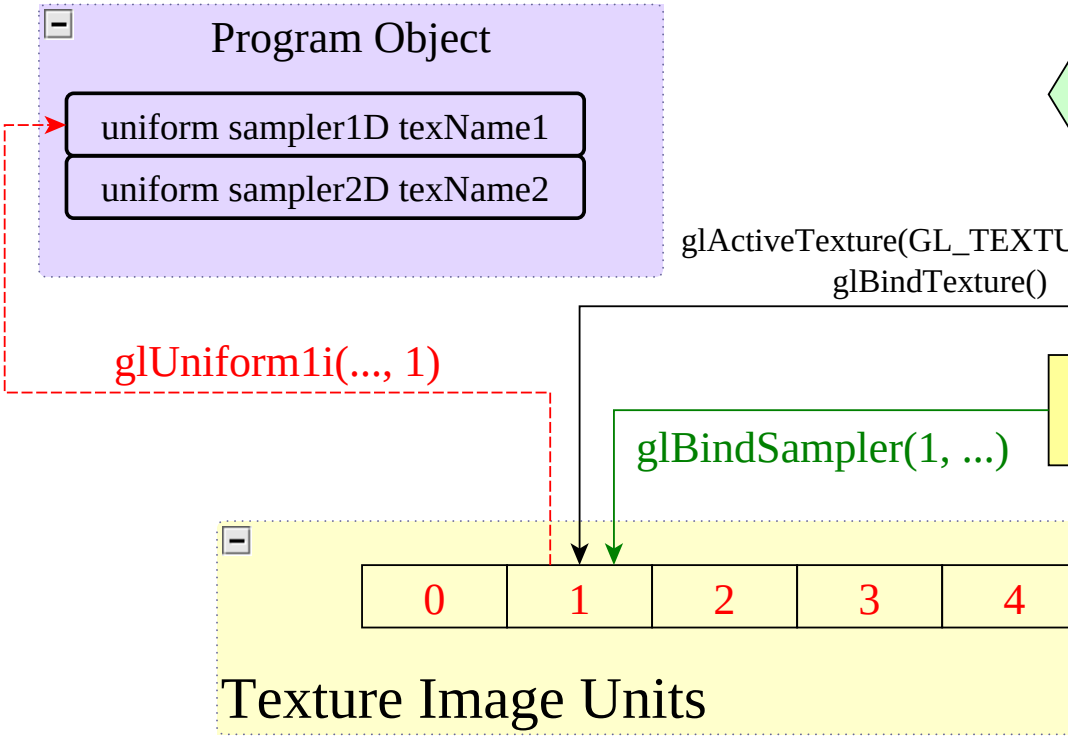
The `GL_TEXTURE_WRAP_S` parameter defines how the “s” component of the texture coordinate will be adjusted if it falls outside of the [0, 1] range. Setting this to `GL_CLAMP_TO_EDGE` clamps this component of the texture coordinate to the edge of the texture. Each component of the texture coordinate can 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:

```
glBindSampler(g_gaussTexUnit, g_gaussSampler);
```

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.

Figure 14.3. Sampler Binding and Context



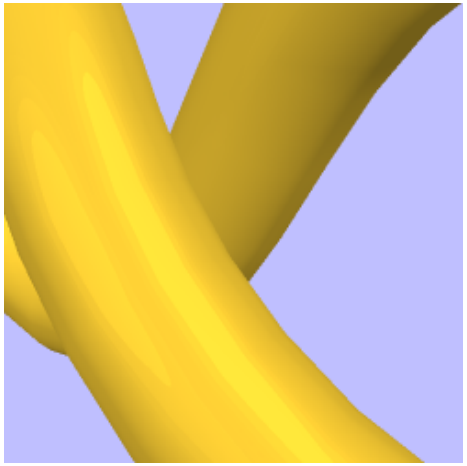
Note

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.

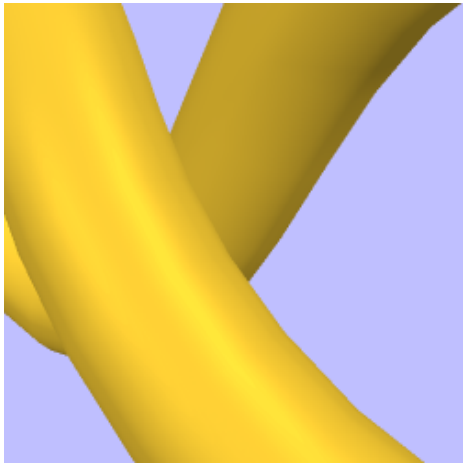
Texture Resolution

This tutorial creates multiple textures at a variety of resolutions. The resolution corresponding with the **1** is the lowest resolution, while the one corresponding with **4** is the highest.

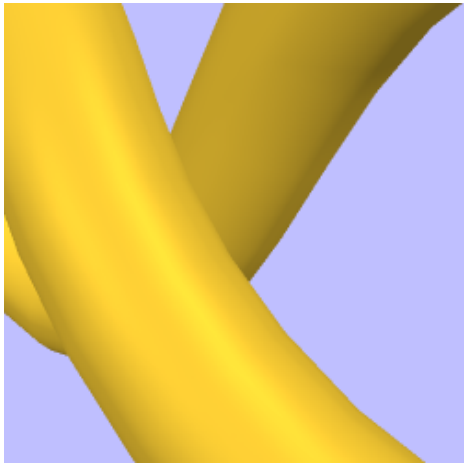
If we use resolution **1**, we can see that it is a pretty rough approximation. We can very clearly see the distinction between the different texels in our lookup table. It is a 64-texel lookup table.



Switching to the level **3** resolution shows more gradations, and looks much more like the shader calculation. This one is 256 texels across.



The largest resolution, **4**, is 512 texels, and it looks nearly identical to the pure shader version for this object.



Interpolation Redux

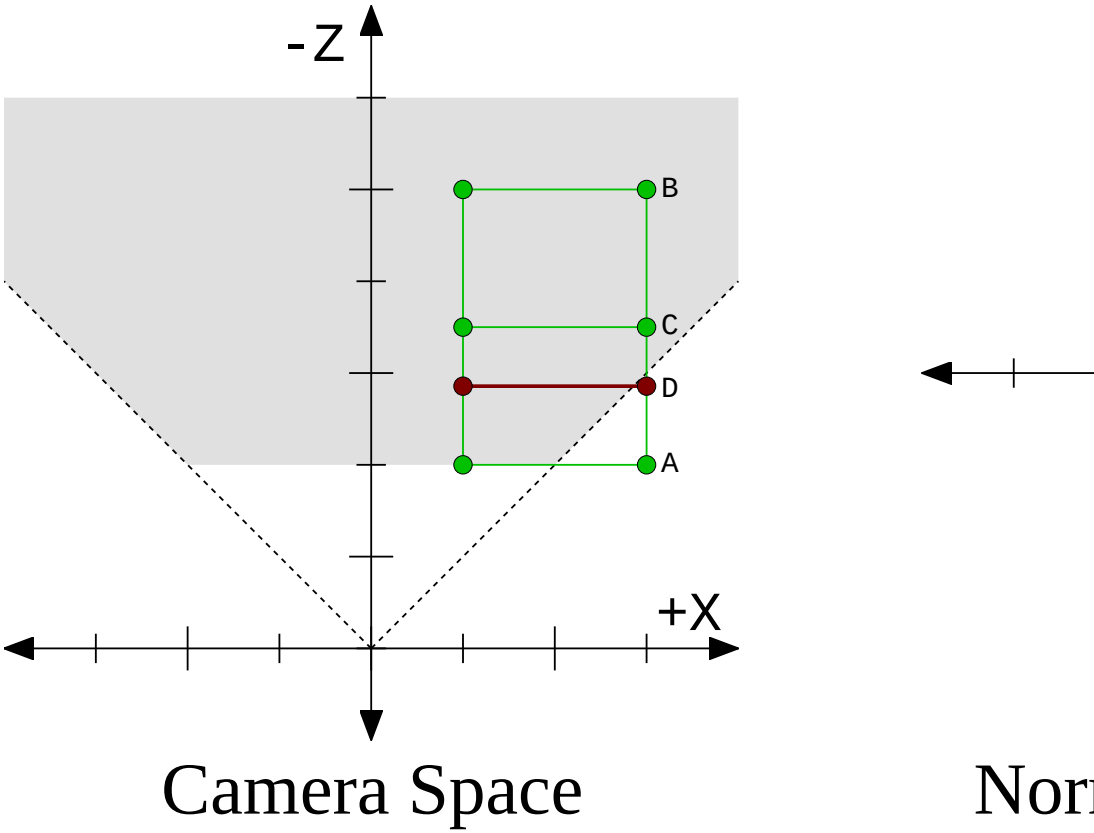
The next step when working with textures is to associate a texture with locations on the surface of an object. But before we can do that, we need to have a discussion about what it means to interpolate a value across a triangle.

Thus far, we have more or less glossed over the details of interpolation. We expanded on this earlier when we explained why per-vertex lighting would not work for certain kinds of functions, as well as when explaining why normals do not interpolate well. But now that we want to associate vertices of a triangle with locations on a texture, we need to fully explain what interpolation means.

The main topic is linearity. In the earlier discussions, it was stressed that interpolation was linear. The question that was danced around is both simple and obscure: linear in what space?

The perspective projection is a non-linear transform; that's why a matrix multiplication is insufficient to express it. Matrices can only handle linear transformations, and the perspective projection needs a division, which is non-linear. We have seen the effect of this non-linear transformation before:

Figure 14.4. Projection and Interpolation



The transformation from normalized device coordinate space to window space is fully linear. So the problem is the transformation from camera space to NDC space, the perspective projection.

From this diagram we see that lines which are parallel in camera space are not necessarily parallel in NDC space; this is one of the features of non-linear transforms. But most important of all is the fact that the distance between objects has changed non-linearly. In camera-space, the lines parallel to the Z axis are all equally spaced. In NDC space, they are not.

Look at the lines A and B. Imagine that these are the only two vertices in the object. In camera-space, the point halfway between them is C. However, in NDC space, the point halfway between them is D. The points C and D are not that close to one another in either space.

So, what space has OpenGL been doing our interpolation in? It might seem obvious to say window space, since window space is the space that the rasterizer (the hardware that does the interpolating) sees and uses. But if it had, we would have had a great many interpolation problems.

Consider interpolating camera space positions. This only works if the interpolation happens in camera-space (or some linear transform thereof). Look at the diagram again; the camera-space position C would be computed for the NDC location D. That would be very wrong.

So our interpolation has somehow been happening in camera space, even though the rasterizer only sees window space. What mechanism causes this?

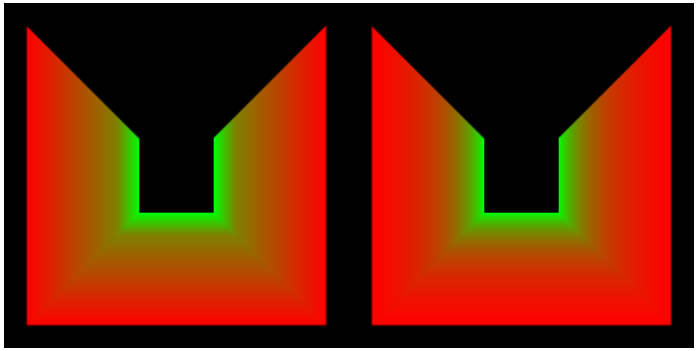
The ability to linearly interpolate values in pre-projection space is called *perspective-correct interpolation*. And we now get to the final reason why our vertex shader provides values in clip-space rather than having the shader perform the perspective divide. The W term of clip-space is vital for performing perspective-correct interpolation.

This makes sense; the clip-space W is after all what makes our transformation non-linear. Perspective-correction simply uses the clip-space W to adjust the interpolation so that it happens in a space that is linear with respect to clip-space. And since clip-space is a linear transform of camera space (using 4D homogeneous coordinates), everything works out. Technically, perspective-correct interpolation does not cause interpolation in camera space, but it interpolates in a space that is a linear transform from camera space.

To see the effects of perspective-correction most dramatically, fire up the Perspective Interpolation project.

There are no camera controls in this demo; the camera is fixed so as to allow the illusion presented to work. Pressing the **P** key switches between perspective-correct interpolation and window-space linear interpolation.

Figure 14.5. Perspective Correct Interpolation



Left: Linear interpolation. Right: Perspective-correct interpolation

The interesting bit is as follows. Switch to the perspective-correct version (a message will appear in the console window) and press the **S** key. Now, the **P** key no longer seems to have any effect; we seem to be trapped in linear-interpolation.

What happens is that the **S** key switches meshes. The “fake” mesh is not really a hallway; it is perfectly flat. It is more or less a mesh who's vertex positions are in NDC-space, after multiplying the original hallway by the perspective matrix. The difference is that there is no W coordinate; it's just a flat object, an optical illusion. There is no perspective information for the perspective-correction logic to key on, so it looks just like window-space linear interpolation.

The switch used to turn on or off perspective-correct interpolation is the interpolation qualifier. Previously, we said that there were three qualifiers: `flat`, `smooth`, and `noperspective`. The third one was previously left undefined before; you can probably guess what it does now.

We are not going to use `noperspective` in the immediate future. Indeed, doing window space interpolation with a perspective projection is exceedingly rare, far more rare than `flat`. The important thing to understand from this section is that interpolation style matters. And `smooth` will be our default interpolation; fortunately, it is OpenGL's default too.

Texture Mapping

One of the most important uses of textures is to vary material parameters across a surface. Previously, the finest granularity that we could get for material parameters is per-vertex values. Textures allow us to get a granularity down to the texel. While we could target the most common material parameter controlled by textures (aka: the diffuse color), we will instead look at something less common. We will vary the specular shininess factor.

To achieve this variation of specular shininess, 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.

Note

Some people refer to textures themselves as “texture maps.” This is sadly widespread terminology, but is incorrect. This text will not refer to them as such, and you are strongly advised not to do the same.

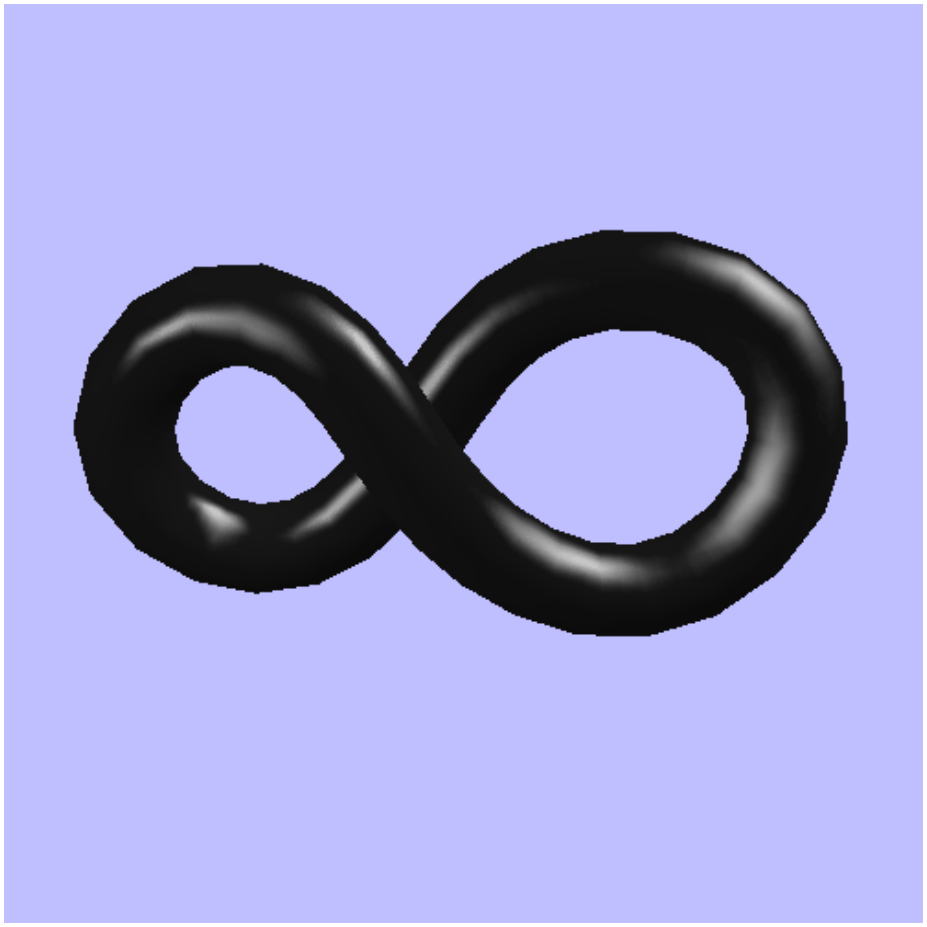
In the last example, the texture coordinate was a value computed based on lighting parameters. The texture coordinate for accessing our shininess texture will instead come from interpolated per-vertex parameters. Hence the prior discussion of the specifics of interpolation.

For simple cases, we could generate the texture coordinate from vertex positions. And in some later tutorials, we will. In the vast majority of cases however, texture coordinates for texture mapping will be part of the per-vertex attribute data.

Since the texture map's coordinates come from per-vertex attributes, this will affect our mesh topography. It adds yet another channel with its own topology, which must be massaged into the overall topology of the mesh.

To see texture mapping in action, load up the Material Texture tutorial. This tutorial uses the same scene as before, but the infinity symbol can use a texture to define the specular shininess of the object.

Figure 14.6. Material Texture



The **Spacebar** switches between one of three rendering modes: fixed shininess with a Gaussian lookup-table, a texture-based shininess with a Gaussian lookup-table, and a texture-based shininess with a shader-computed Gaussian term. The **Y** key switches between the infinity symbol and a flat plane; this helps make it more obvious what the shininess looks like. The **9** key switches to a material with a dark diffuse color and bright specular color; this makes the effects of the shininess texture more noticeable. Press the **8** key to return to the gold material.

Texture 2D

The **1** through **4** keys still switch to different resolutions of Gaussian textures. Speaking of which, that works rather differently now.

Previously, we assumed that the specular shininess was a fixed value for the entire surface. Now that our shininess values can come from a texture, this is not the case. With the fixed shininess, we had a function that took one parameter: the dot-product of the half-angle vector with the normal. But with a variable shininess, we have a function of two parameters. Functions of two variables are often called “two dimensional.”

It is therefore not surprising that we model such a function with a two-dimensional texture. The S texture coordinate represents the dot-product, while the T texture coordinate is the shininess value. Both range from [0, 1], so they fit within the expected range of texture coordinates.

Our new function for building the data for the Gaussian term is as follows:

Example 14.5. BuildGaussianData in 2D

```
void BuildGaussianData(std::vector<GLubyte> &textureData,
                       int cosAngleResolution,
                       int shininessResolution)
{
    textureData.resize(shininessResolution * cosAngleResolution);

    std::vector<unsigned char>::iterator currIt = textureData.begin();
    for(int iShin = 1; iShin <= shininessResolution; iShin++)
    {
        float shininess = iShin / (float)(shininessResolution);
        for(int iCosAng = 0; iCosAng < cosAngleResolution; iCosAng++)
        {
            float cosAng = iCosAng / (float)(cosAngleResolution - 1);
            float angle = acosf(cosAng);
            float exponent = angle / shininess;
            exponent = -(exponent * exponent);
            float gaussianTerm = glm::exp(exponent);

            *currIt = (unsigned char)(gaussianTerm * 255.0f);
            ++currIt;
        }
    }
}
```

This function writes into a 1D array of data. It writes a full set of values for a particular shininess, then writes the next values for that shininess, and so on. This is the most standard way that image data is stored in virtually every image format. Naturally, this is also how OpenGL takes its data.

However, notice that the texture data expects a lower-left origin: the first row, which corresponds to the smallest shininess value (a T value of 0), is the *first* row. Sadly, this not how most image formats store rows of pixel data; they tend to use a top-left orientation, so the first row in most image formats is the top row.

This brings us to how we present this data to OpenGL. The function is similar to what we saw before, only with a couple of changes.

Example 14.6. CreateGaussianTexture in 2D

```
GLuint CreateGaussianTexture(int cosAngleResolution, int shininessResolution)
{
    std::vector<unsigned char> textureData;
    BuildGaussianData(textureData, cosAngleResolution, shininessResolution);

    GLuint gaussTexture;
    glGenTextures(1, &gaussTexture);
    glBindTexture(GL_TEXTURE_2D, gaussTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_R8, cosAngleResolution, shininessResolution, 0,
        GL_RED, GL_UNSIGNED_BYTE, &textureData[0]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
    glBindTexture(GL_TEXTURE_2D, 0);

    return gaussTexture;
```

```
}
```

Here, we can see that we use the `GL_TEXTURE_2D` target instead of the 1D version. We also 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.

Image From a File

Our Gaussian texture comes from data we compute, but the specular shininess texture is defined by a file. For this, we use the GL Image library that is part of the OpenGL SDK. While the GL Image library has functions that will directly create textures for us, it is instructive to see a more manual process.

Example 14.7. CreateShininessTexture function

```
void CreateShininessTexture()
{
    std::auto_ptr<glimg::ImageSet> pImageSet;

    try
    {
        pImageSet.reset(glimg::loaders::dds::LoadFromFile("data\\main.dds"));
        std::auto_ptr<glimg::Image> pImage(pImageSet->GetImage(0, 0, 0));

        glimg::Dimensions dims = pImage->GetDimensions();

        glGenTextures(1, &g_shineTexture);
        glBindTexture(GL_TEXTURE_2D, g_shineTexture);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_R8, dims.width, dims.height, 0,
            GL_RED, GL_UNSIGNED_BYTE, pImage->GetImageData());
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
        glBindTexture(GL_TEXTURE_2D, 0);
    }
    catch(glimg::ImageCreationException &e)
    {
        printf(e.what());
        throw;
    }
}
```

The GL Image library has a number of loaders for different image formats; the one we use in the first line of the try-block is the DDS loader. DDS stands for “Direct Draw Surface,” but it really has nothing to do with Direct3D or DirectX. It is unique among image file formats

The `glimg::ImageSet` object also supports all of the unique features of textures; an `ImageSet` represents all of the images for a particular texture. To get at the image data, we first select an image with the `GetImage` function. We will discuss later what exactly these parameters represent, but (0, 0, 0) represents the single image that the DDS file contains.

Images in textures can have different sizes, so each `glimg::Image` object has its own dimensions, which we retrieve. After this, we use the usual methods to upload the texture. The `GetImageData` object returns a pointer to the data for that image as loaded from the DDS file.

Shaders Textures in 2D

Since we are using texture objects of `GL_TEXTURE_2D` type, we must use sampler2D samplers in our shader.

```
uniform sampler2D gaussianTexture;
uniform sampler2D shininessTexture;
```

We have two textures. The shininess texture determines our specular shininess value. This is accessed in the fragment shader's main function, before looping over the lights:

Example 14.8. Shininess Texture Access

```
void main()
{
    float specularShininess = texture(shininessTexture, shinTexCoord).r;

    vec4 accumLighting = Mtl.diffuseColor * Lgt.ambientIntensity;
    for(int light = 0; light < numberOfLights; light++)
    {
        accumLighting += ComputeLighting(Lgt.lights[light],
            cameraSpacePosition, vertexNormal, specularShininess);
    }

    outputColor = sqrt(accumLighting); //2.0 gamma correction
}
```

The `ComputeLighting` function now takes the specular term as a parameter. It uses this as part of its access to the Gaussian texture:

Example 14.9. Gaussian Texture with Specular

```
vec3 halfAngle = normalize(lightDir + viewDirection);
vec2 texCoord;
texCoord.s = dot(halfAngle, surfaceNormal);
texCoord.t = specularShininess;
float gaussianTerm = texture(gaussianTexture, texCoord).r;

gaussianTerm = cosAngIncidence != 0.0 ? gaussianTerm : 0.0;
```

The use of the S and T components matches how we generated the lookup texture. The shader that computes the Gaussian term uses the specular passed in, and is little different otherwise from the usual Gaussian computations.

Rendering with Shininess

We have two textures in this example, but we do not have two sampler objects (remember: sampler objects are not the same as sampler types in GLSL). We can use the same sampler object for accessing both of our textures.

Because they are 2D textures, they are accessed with two texture coordinates: S and T. So we need to clamp both S and T in our sampler object:

```
glGenSamplers(1, &g_textureSampler);
glSamplerParameteri(g_textureSampler, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glSamplerParameteri(g_textureSampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glSamplerParameteri(g_textureSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glSamplerParameteri(g_textureSampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

When the time comes to render, the sampler is bound to both texture image units:

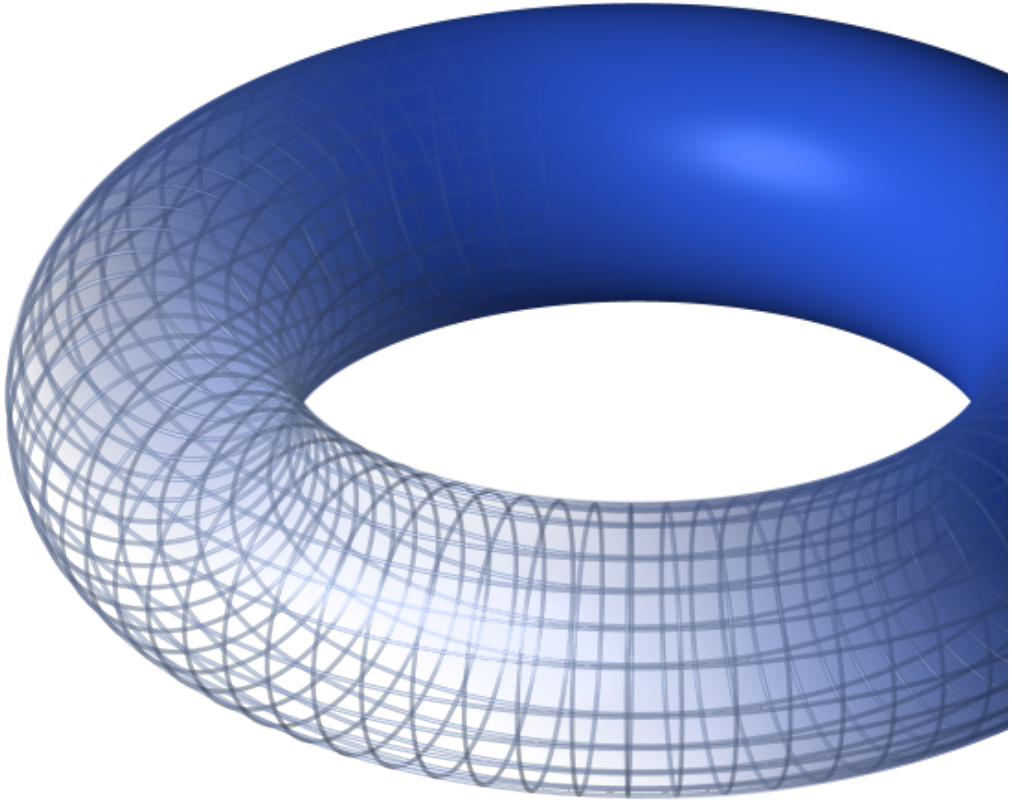
```
glActiveTexture(GL_TEXTURE0 + g_gaussTexUnit);
glBindTexture(GL_TEXTURE_2D, g_gaussTextures[g_currTexture]);
glBindSampler(g_gaussTexUnit, g_textureSampler);

glActiveTexture(GL_TEXTURE0 + g_shineTexUnit);
glBindTexture(GL_TEXTURE_2D, g_shineTexture);
glBindSampler(g_shineTexUnit, g_textureSampler);
```

It is perfectly valid to bind the same sampler to more than one texture unit. Indeed, while many programs may have hundreds of individual textures, they may have less than 10 distinct samplers. It is also perfectly valid to bind the same texture to different units that have different samplers attached to them.

The Way of the Map

We use two objects in this tutorial: a flat plane and an infinity symbol. The mapping of the plane is fairly obvious, but the infinity symbol's map is more interesting. Topologically, the infinity symbol is no different from that of a torus.

Figure 14.7. A Torus



That is, the infinity symbol and a torus have the same connectivity between vertices; those vertices are just in different positions.

Mapping an object onto a 2D plane generally means finding a way to slice the object into pieces that fit onto that plane. However, a torus is, topologically speaking, equivalent to a plane. This plane is rolled into a tube, and bent around, so that each side connects to its opposing side directly. Therefore, mapping a texture onto this means reversing the process. The tube is cut at one end, creating a cylinder. Then, it is cut lengthwise, much like a car tire, and flattened out into a plane.

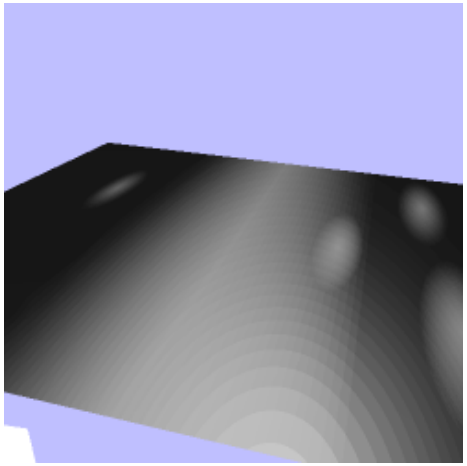
Exactly where those cuts need to be made is arbitrary. And because the specular texture mirrors perfectly in the S and T directions, it is not possible to tell exactly where the seams in the topology are. But they do need to be there.

What this does mean is that the vertices along the same have duplicate positions and normals. Because they have different texture coordinates, their shared positions and normals must be duplicated to match what OpenGL needs.

Smudges on Glass

The best way to understand how the shininess texture affects the rendered result is to switch to the dark material with the **9** key. The plane also shows this a bit easier than the curved infinity symbol.

Figure 14.8. Surface smudges



The areas with lower shininess, the bright areas, look like smudge marks. While the bright marks in the highly shiny areas only reflect light when the light source is very close to perfectly reflecting, the lower shininess areas will reflect light from much larger angles.

One interesting thing to note is how our look-up table works with the flat surface. Even at the highest resolution, 512 individual values, the lookup table is pretty poor; a lot of concentric rings are plainly visible. It looked more reasonable on the infinity symbol because it was heavily curved, and therefore the specular highlights were much smaller. On this flat surface, the visual artifacts become much more obvious. The **Spacebar** can be used to switch to a shader-based computation to see the correct version.

If our intent was to show a smudged piece of metal or highly reflective black surface, we could enhance the effect by also applying a texture that changed the specular reflectance. Smudged areas don't tend to reflect as strongly as the shiny ones. We could use the same texture mapping (ie: the same texture coordinates) and the specular texture would not even have to be the same size as our shininess texture.

There is one more thing to note about the shininess texture. The size of the texture is 1024x256 in size. The reason for that is that the texture is intended to be used on the infinity symbol. This object is longer in model space than it is around. By making the texture map 4x longer in the axis that is mapped to the S coordinate, we are able to more closely maintain the aspect ratio of the objects on the texture than the flat plane we see here. All of those oval smudge marks you see are in fact round in the texture. They are still somewhat ovoid and distorted on the infinity symbol though.

It is generally the job of the artist creating the texture mapping to ensure that the aspect ratio and stretching of the mapped object remains reasonable for the texture. In the best possible case, every texel in the texture maps to the same physical size on the object's surface. Fortunately for a graphics programmer, doing that isn't your job.

Unless of course your job is writing the tool that the artists use to help them in this process.

In Review

In this tutorial, you have learned the following:

- Textures are objects that store one or more arrays of data of some dimensionality. They can be created and filled with data from OpenGL. Shaders can reference them with sampler types, and they can access them using texturing functions. The values in a texture have a specific meaning; never forget what the texture and its stored data represent.
- The data in textures can represent arbitrary information. They can be used to vary a material parameter across a surface, replace a complex function with a look-up table, or anything else you might need a multi-dimensional array of values for.
- Vertex or geometry shader outputs interpolated across polygons can be interpolated linearly in window space or linearly in pre-projection space. The GLSL interpolation qualifiers control which kind of interpolation happens.
- Textures can be associated with points on a surface by giving those vertex attributes texture coordinates. The texture coordinate is interpolated across the triangle's surface and then used to fetch values from a texture. This is but a part of the utility of textures.

Further Study

Try doing these things with the given programs.

- If you were to look at the look-up table for our specular function, you will see that much of it is very dark, if not actually at 0.0. Even when the dot product is close to 1.0, it does not take very far before the specular value becomes negligible. One way to improve our look-up table without having to use larger textures is to change how we index the texture. If we index the texture by the square-root of the dot-product, then there will be more room in the table for the values close to 1.0, and less for the values close to 0.0. This is similar to how gamma correction works. Implement this by storing the values in the table based on the square-root of the dot-product, and then take the square-root of the dot-product in the shader before accessing the texture.
- Animate the texture coordinates in the texture mapping tutorial. Do this by sending an offset to the fragment shader which is applied to the texture coordinates. You can generate the offset based on the Framework::Timer `g_lightTimer`. Make sure to use the `mod` function on the texture coordinates with a value of 1.0, so that the texture coordinate will always stay on the range [0, 1].

OpenGL Functions of Note

glGenTextures, glBindTexture, glActiveTexture	These functions create texture objects and bind them to a specific texture target in the OpenGL context. `glActiveTexture` selects which texture unit the texture all texture object commands refer to, including `glBindTexture`. The first time a texture is bound to a target, that texture object takes on the texture type associated with that target. It then becomes illegal to bind that texture to a different target. So if you bind a texture to `GL_TEXTURE_2D` the first time, you cannot bind it to any other target ever again.
glTexImage1D, glTexImage2D	Allocates storage for an image in the currently bound texture of the currently active texture unit. If the last parameter is not NULL, then these functions will also upload data to that image. Otherwise, the content of this image is undefined.
glTexParameter	Sets a parameter in the currently bound texture of the currently active texture unit.
glGenSamplers, glBindSampler	These functions create sampler objects and bind them to the context for use.
glSamplerParameter	Sets a parameter to the given sampler object. Unlike most OpenGL functions that operate on objects, this function takes a sampler object as a parameter; it does not require that the sampler object be bound to the context.

GLSL Functions of Note

```
vec4 texture(sampler texSampler, vec texCoord);
```


Accesses the texture associated with *`texSampler`*, at the location given by *`texCoord`*. The sampler type can be any of the sampler types. The number of components of vec depends on the type of sampler used; a sampler1D takes a single float, while a sampler2D takes a vec2.

Glossary

look-up table	A table that is used to represent an expensive function computation. The function is sampled at discrete intervals. To access the function, the input values for the function are transformed into a discrete location in the table and that value is returned.
texture	An object that contains one or more images of a particular dimensionality. The data in the images can be fetched by the user in a shader. Textures have a type, which represents the nature of that particular texture.
image	An array of data of a particular dimensionality. Images can be 1D, 2D, or 3D in size. The points of data in an image are 4-vector values, which can be floating point or integers.
texture type	Represents the basic nature of the texture. The texture type defines the dimensionality of the images it stores. It defines the size of the texture coordinate that the texture takes, the number of images it can contain, and various other information about the texture.
normalized integers	An integer that represents a floating-point value on the range [0, 1] for unsigned integers and [-1, 1] for signed integers. Normalized integers use their entire bitrange to represent a floating point value. The maximum value for the integer's bitdepth represents the maximum floating point value, and the minimum value for the integer's bitdepth represents the minimum floating point value.
pixel transfer	The act of sending pixel data to an image in OpenGL, or receiving pixel data from OpenGL.
texel	A pixel within a texture image. Used to distinguish between a pixel in a destination image and pixels in texture images.
GLSL sampler	A number of types in GLSL that represents a texture bound to a texture image unit of the OpenGL context. For every texture type in OpenGL, there is a matching sampler type. There are a number of restrictions on the use of samplers in GLSL. They can only be declared globally as uniforms and as input parameters to functions. They can only be used as the value passed to a function, whether user-defined or built-in.
sampling	The process of accessing data from one or more of the images of the texture, using a specific texture coordinate.
texture coordinate	A value that is used to access locations within a texture. Each texture type defines what dimensionality of texture coordinate it takes (note that the texture type may define a different texture coordinate dimensionality from the image dimensionality). Texture coordinates are often normalized on the range [0, 1]. This allows texture coordinates to ignore the size of the specific texture they are used with. Texture coordinates are comprised by the S, T, R, and Q components, much like regular vectors are composed of X, Y, Z, and W components. In GLSL, the R component is called “P” instead.
texture image unit	An array of locations in the OpenGL context where texture objects are bound to. Programs can have their GLSL sampler uniforms associated with one of the entries in this array. When using such a program, it will use the texture object bound to that location to find the texture for that GLSL sampler.
sampler object	An OpenGL object that defines how a texture is accessed in the shader. The parameters that are set on a sampler object can also be set on a texture object, but if a sampler is bound to the same image unit as a texture, then the sampler takes precidence.
perspective-correct interpolation	A scheme for interpolating values across the surface of a triangle in pre-projection space. This is necessary when working with perspective projections. This is the default interpolation scheme in OpenGL; it can be selectively disabled with the `noperspective` GLSL qualifier.
texture mapping	The association between one or more textures and positions on the surface. This association is made by putting texture coordinates in the per-vertex attribute data. Therefore, each triangle vertex has a texture coordinate.

Chapter 15. Many Images

In the last tutorial, we looked at textures that were not pictures. Now, we will look at textures that are pictures. However, unlike the last tutorial, where the textures represented some parameter in the light equation, here, we will just be directly outputting the values read from the texture.

Graphics Fudging

Before we begin however, there is something you may need to do. When you installed your graphics drivers, installed along with it was an application that allows you to provide settings for your graphics driver. This affects how graphics applications render and so forth.

Thus far, most of those settings have been irrelevant to us because everything we have done has been entirely in our control. The OpenGL specification defined almost exactly what could and could not happen, and outside of actual driver bugs, the results we produced are reproducible and nearly identical across hardware.

That is no longer the case, as of this tutorial.

Texturing has long been a place where graphics drivers have been given room to play and fudge results. The OpenGL specification plays fast-and-loose with certain aspects of texturing. And with the driving need for graphics card makers to have high performance and high image quality, graphics driver writers can, at the behest of the user, simply ignore the OpenGL spec with regard to certain aspects of texturing.

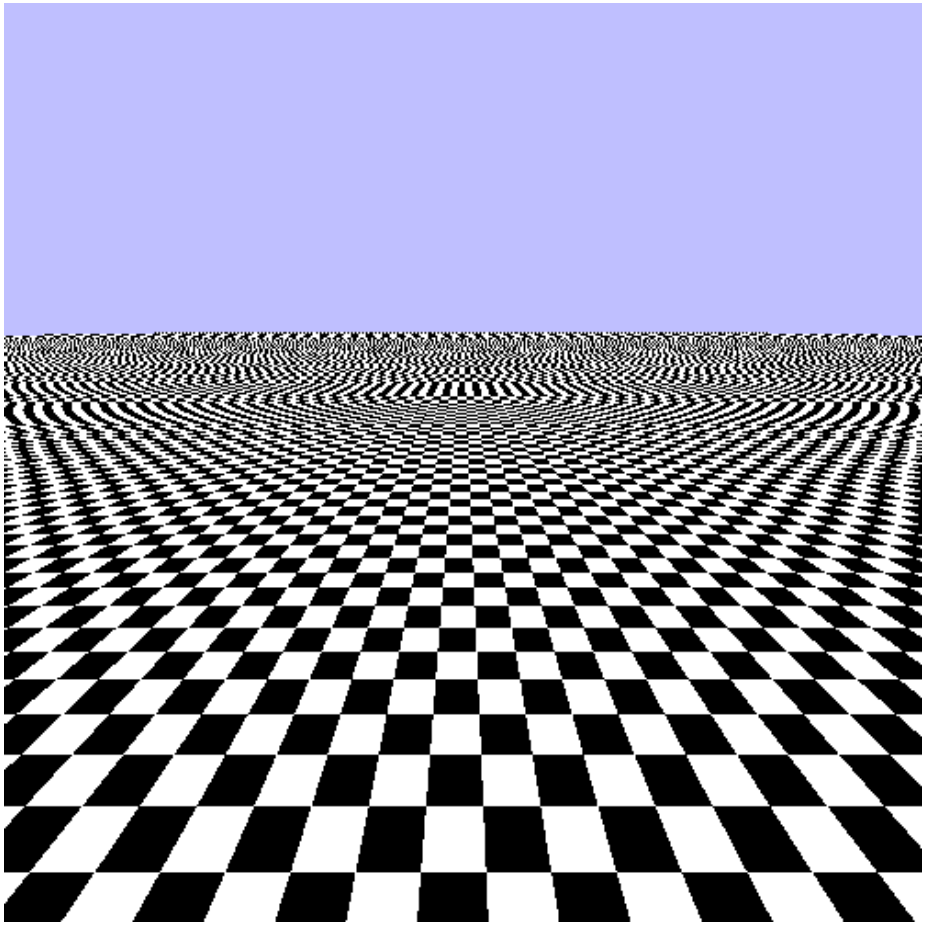
The image quality settings in your graphics driver provide control over this. They are ways for you to tell graphics drivers to ignore whatever the application thinks it should do and instead do things their way. That is fine for a game, but right now, we are learning how things work. If the driver starts pretending that we set some parameter that we clearly did not, it will taint our results and make it difficult to know what parameters cause what effects.

Therefore, you will need to go into your graphics driver application and change all of those setting to the value that means to do what the application says. Otherwise, the visual results you get for the following code may be very different from the given images. This includes settings for antialiasing.

Playing Checkers

We will start by drawing a single large, flat plane. The plane will have a texture of a checkerboard drawn on it. The camera will hover above the plane, looking out at the horizon as if the plane were the ground. This is implemented in the Many Images tutorial project.

Figure 15.1. Basic Checkerboard Plane



The camera is automatically controlled, though it's motion can be paused with the **P** key. The other functions of the tutorial will be explained as we get to them.

If you look at the `BigPlane.xml` file, you will find that the texture coordinates are well outside of the [0, 1] range we are used to. They span from [-64, 64] now, but the texture itself is only valid within the [0, 1] range.

Recall from the last tutorial that the sampler object has a parameter that controls what texture coordinates outside of the [0, 1] range mean. This tutorial uses many samplers, but all of our samplers use the same S and T wrap modes:

```
glSamplerParameteri(g_samplers[samplerIx], GL_TEXTURE_WRAP_S, GL_REPEAT);
glSamplerParameteri(g_samplers[samplerIx], GL_TEXTURE_WRAP_T, 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. Well, usually; this does not work for certain texture types, but only because they take texture coordinates with special meanings. For them, the wrap modes are ignored entirely.

You may toggle between two meshes with the **Y** key. The alternative mesh is a long, square corridor.

The shaders used here are very simple. The vertex shader takes positions and texture coordinates as inputs and outputs the texture coordinate directly. The fragment shader takes the texture coordinate, fetches a texture with it, and writes that color value as output. Not even gamma correction is used.

The texture in question is 128x128 in size, with 4 alternating black and white squares on each side. Each of the black or white squares is 32 pixels across.

Linear Filtering

While this example certainly draws a checkerboard, you can see that there are some visual issues. We will start finding solutions to this with the least obvious glitches first.

Take a look at one of the squares at the very bottom of the screen. Notice how the line looks jagged as it moves to the left and right. You can see the pixels of it sort of crawl up and down as it shifts around on the plane.

Figure 15.2. Jagged Texture Edge

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.

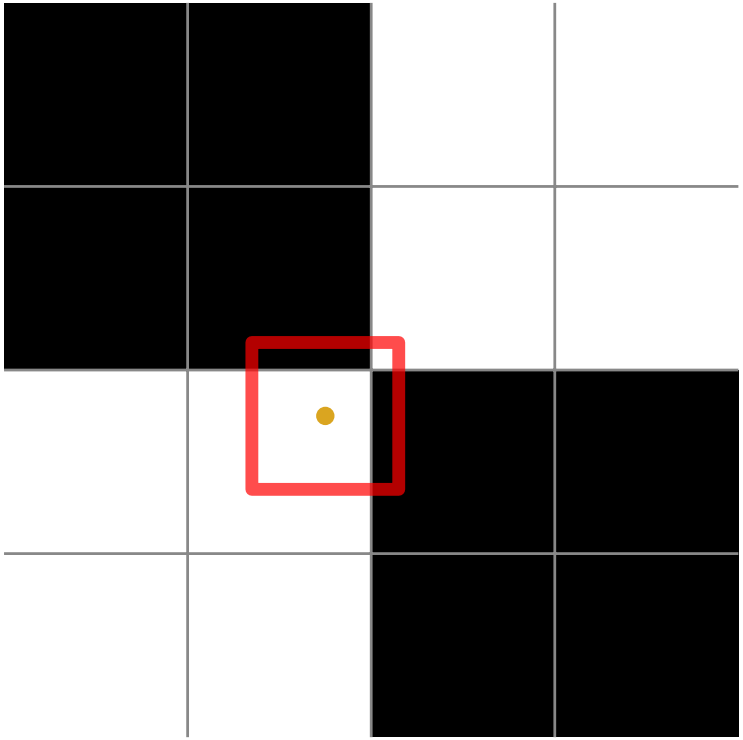
In OpenGL, magnification and minification filtering are each set independently. That is what the `GL_TEXTURE_MAG_FILTER` and `GL_TEXTURE_MIN_FILTER` sampler parameters control. We are currently using `GL_NEAREST` for both; this is called *nearest filtering*. This mode means that each texture coordinate 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 half-way 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.

Each fragment being rendered takes up a certain area of space on the screen: the area of the destination pixel for that fragment. The texture mapping of the rendered surface to the texture gives a texture coordinate for each point on the surface. But a pixel is not a single, infinitely small point on the surface; it represents some finite area of the surface.

Therefore, we can use the texture mapping in reverse. We can take the four corners of a pixel area and find the texture coordinates from them. The area of this 4-sided figure, in the space of the texture, is the area of the texture that is being mapped to that location on the screen. With a perfect texture accessing system, the value we get from the GLSL `texture` function would be the average value of the colors in that area.

Figure 15.3. Nearest Sampling



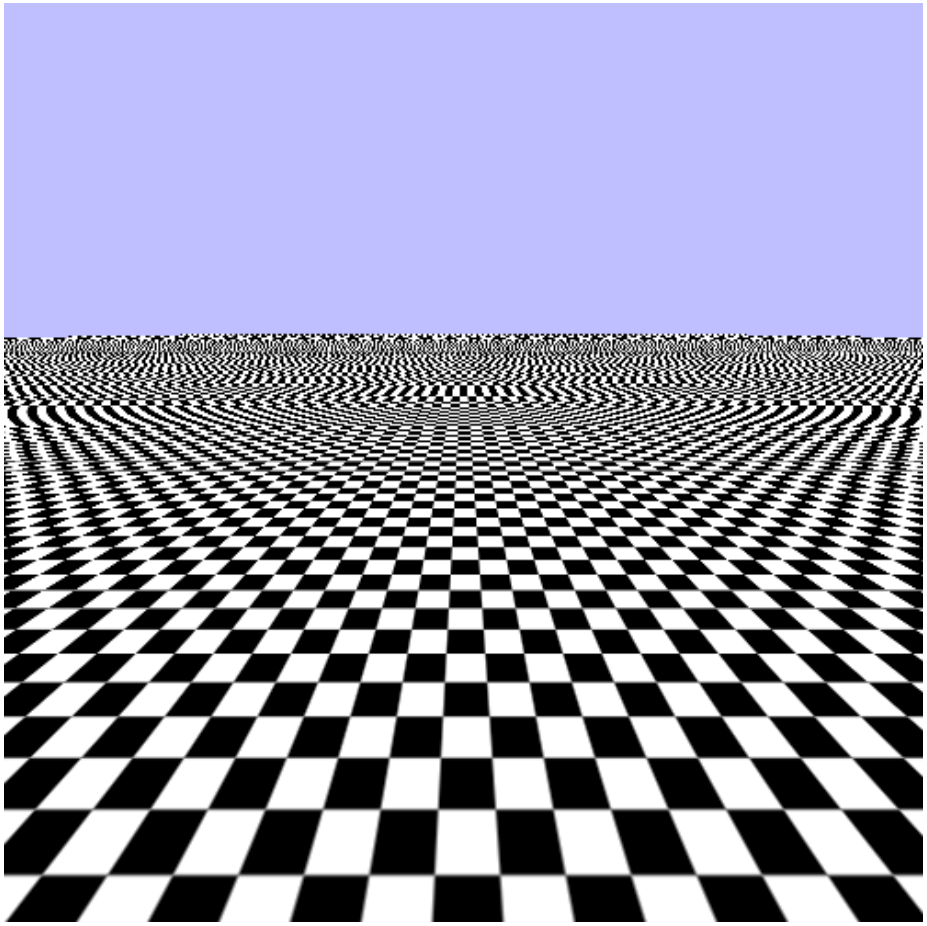
The dot represents the texture coordinate's location on the texture. The box is the area that the fragment covers. The problem happens because a fragment area mapped into the texture's space may cover some white area and some black area. Since nearest only picks a single texel, which is either black or white, it does not accurately represent the mapped area of the fragment.

One obvious way to smooth out the differences is to do exactly that. Instead of picking a single sample for each texture coordinate, pick the nearest 4 samples and then interpolate the values based on how close they each are to the texture coordinate. To do this, we set the magnification and minification filters to `GL_LINEAR`.

```
glSamplerParameteri(g_samplers[1], GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(g_samplers[1], GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

This is called, surprisingly enough, *linear filtering*. In our tutorial, press the **2** key to see what linear filtering looks like; press **1** to go back to nearest sampling.

Figure 15.4. Linear Filtering



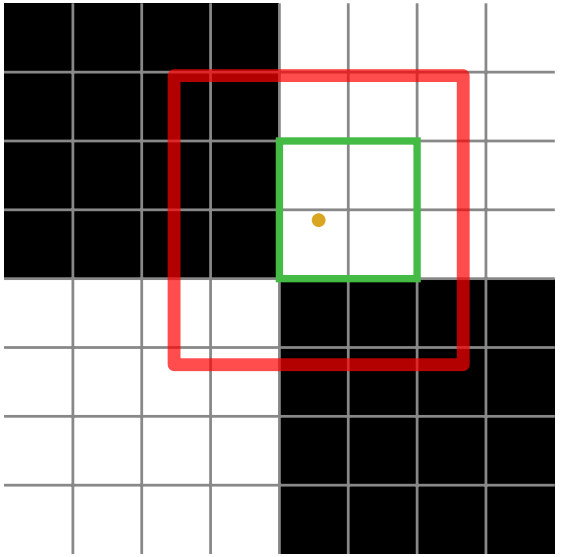
That looks much better for the squares close to the camera. It creates a bit of fuzziness, but this is generally a lot easier for the viewer to tolerate than pixel crawl. Human vision tends to be attracted to movement, and false movement like dot crawl can be distracting.

Needs More Pictures

Speaking of distracting, let's talk about what is going on in the distance. When the camera moves, 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.

Figure 15.5. Large Minification Sampling

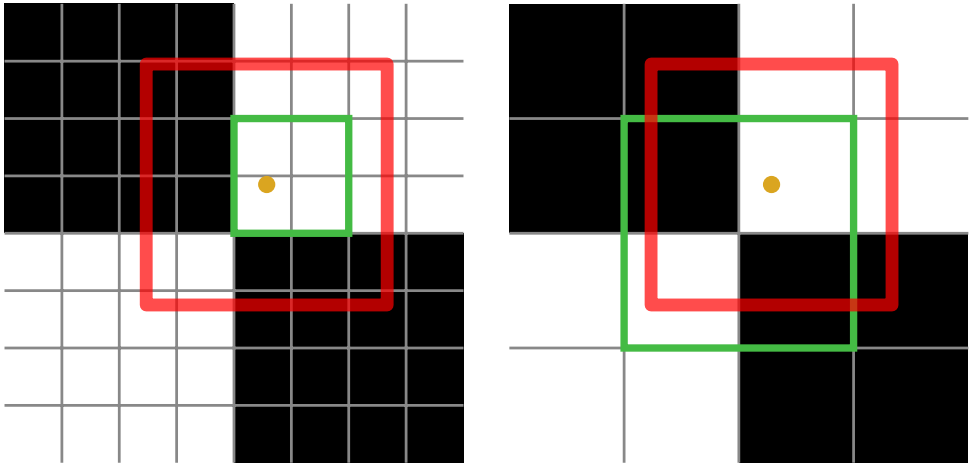


The inner box represents the nearest texels, while the outer box represents the entire fragment mapped area. We can see that the value we get with nearest sampling will be pure white, since the four nearest values are white. But the value we should get based on the covered area is some shade of gray.

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.

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.

Figure 15.6. Mipmapped Minification Sampling



These smaller versions of an image are called *mipmaps*; they are also sometimes called mipmap levels. Previously, it was said that textures can store multiple images. The additional images, for many texture types, are mipmaps. 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.

The DDS image format is one of the few image formats that actually supports storing all of the mipmaps for a texture in the same file. Most image formats only allow one image in a single file. The texture loading code for our 128x128 texture with mipmaps is as follows:

Example 15.1. DDS Texture Loading with Mipmaps

```
std::string filename(LOCAL_FILE_DIR);
filename += "checker.dds";

std::auto_ptr<glimg::ImageSet> pImageSet(glimg::loaders::dds::LoadFromFile(filename.c_str()));

glGenTextures(1, &g_checkerTexture);
glBindTexture(GL_TEXTURE_2D, g_checkerTexture);
```



```
for(int mipmapLevel = 0; mipmapLevel < pImageSet->GetMipmapCount(); mipmapLevel++)
{
    glimg::SingleImage image = pImageSet->GetImage(mipmapLevel, 0, 0);
    glimg::Dimensions dims = pImage->GetDimensions();

    glTexImage2D(GL_TEXTURE_2D, mipmapLevel, GL_RGB8, dims.width, dims.height, 0,
        GL_BGRA, GL_UNSIGNED_INT_8_8_8_8_REV, image.GetImageData());
}

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, pImageSet->GetMipmapCount() - 1);
glBindTexture(GL_TEXTURE_2D, 0);
```

Because the file contains multiple mipmaps, we must load each one in turn. The GL Image library considers each mipmap to be its own image. The `GetDimensions` member of `glimg::SingleImage` returns the size of the particular mipmap.

The `glTexImage2D` function takes the mipmap level to load as the second parameter. The width and height parameters represent the size of the mipmap in question, not the size of the base level.

Notice that the last statements have changed. The `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL` parameters tell OpenGL what mipmaps in our texture can be used. This represents a closed range. Since a 128x128 texture has 8 mipmaps, we use the range [0, 7]. 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*. This tutorial does not load two checkerboard textures; it only ever uses one checkerboard. The reason mipmaps have not been used until now is because mipmap filtering was not activated. 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.

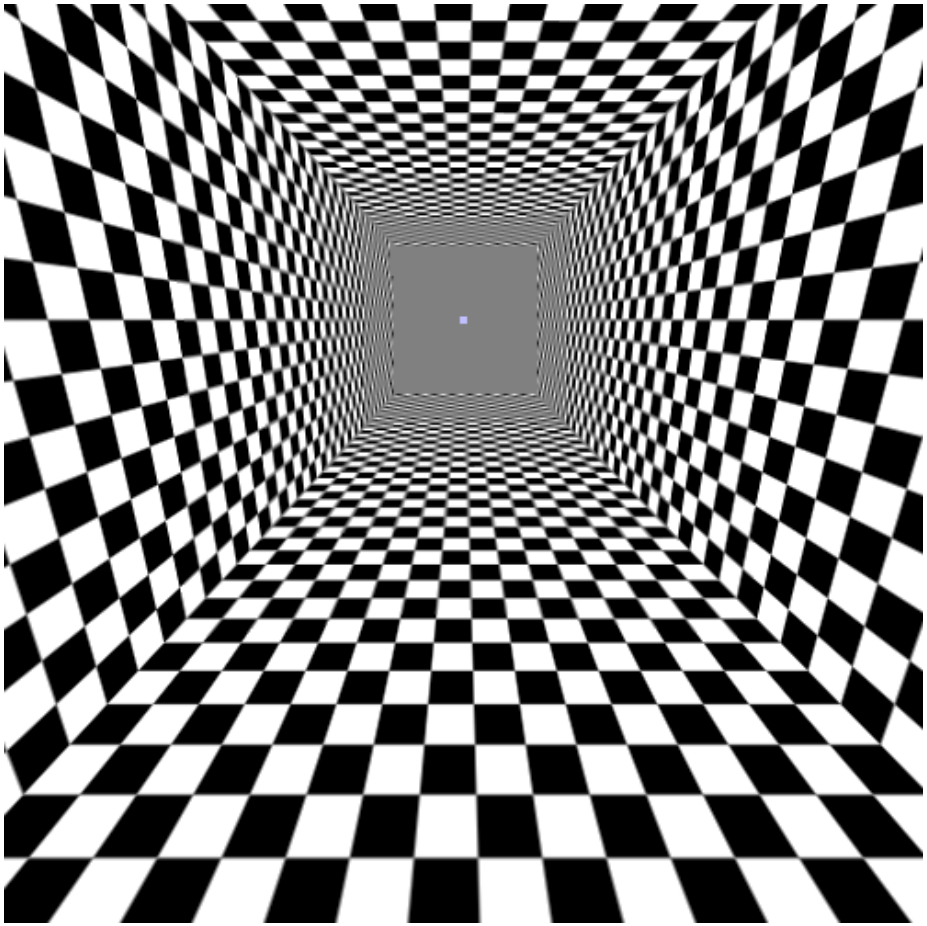
```
glSamplerParameteri(g_samplers[2], GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(g_samplers[2], GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
```

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[1]. Then, when it samples from that mipmap, it will use linear filtering of the four nearest samples within that one mipmap.

If you press the **3** key in the tutorial, you can see the effects of this filtering mode.

[1]This is a simplification; a more thorough discussion is forthcoming.

Figure 15.7. Hallway with Mipmapping

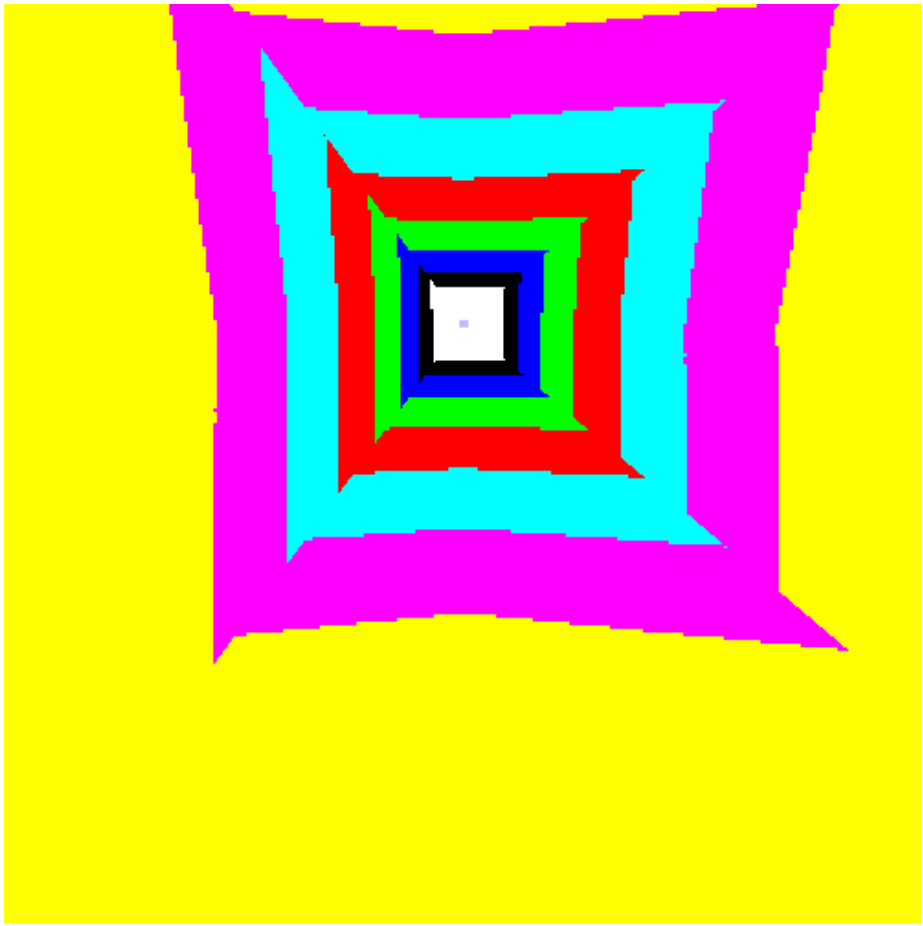


That's a lot more reasonable. It isn't perfect, but it is much better than the random motion in the distance that we have previously seen.

It can be difficult to truly understand the effects of mipmap filtering when using normal textures and mipmaps. Therefore, if you press the **Spacebar**, the tutorial will switch to a special texture. It is not loaded from a file; it is instead constructed at runtime.

Normally, mipmaps are simply smaller versions of larger images, using linear filtering or various other algorithms to compute a reasonable scaled down result. This special texture's mipmaps are all flat colors, but each mipmap has a different color. This makes it much more obvious where each mipmap is.

Figure 15.8. Hallway with Special Texture



Now we can really see where the different mipmaps are. They don't quite line up on the corners. But remember: this just shows the mipmap boundaries, not the texture coordinates themselves.

Special Texture Generation

The special mipmap viewing texture is interesting, as it demonstrates an issue you may need to work with when uploading certain textures: alignment.

The checkerboard texture, though it only stores black and white values, actually has all three color channels, plus a fourth value. Since each channel is stored as 8-bit unsigned normalized integers, each pixel takes up 4 * 8 or 32 bits, which is 4 bytes.

OpenGL image uploading and downloading is based on horizontal rows of image data. Each row is expected to have a certain byte alignment. The OpenGL default is 4 bytes; since our pixels are 4 bytes in length, every mipmap will have a line size in bytes that is a multiple of 4 bytes. Even the 1x1 mipmap level is 4 bytes in size.

Note that the internal format we provide is `GL_RGB8`, even though the components we are transferring are `GL_BGRA` (the A being the fourth component). This means that OpenGL will more or less discard the fourth component we upload. That is fine.

The issue with the special texture's pixel data is that it is not 4 bytes in length. The function used to generate a mipmap level of the special texture is as follows:

Example 15.2. Special Texture Data

```
void FillWithColor(std::vector<GLubyte> &buffer,
                   GLubyte red, GLubyte green, GLubyte blue,
                   int width, int height)
{
    int numTexels = width * height;
    buffer.resize(numTexels * 3);

    std::vector<GLubyte>::iterator it = buffer.begin();
    while(it != buffer.end())
    {
        *it++ = red;
        *it++ = green;
        *it++ = blue;
    }
}
```

This creates a texture that has 24-bit pixels; each pixel contains 3 bytes.

That is fine for any width value that is a multiple of 4. However, if the width is 2, then each row of pixel data will be 6 bytes long. That is not a multiple of 4 and therefore breaks alignment.

Therefore, we must change the pixel alignment that OpenGL uses. The `LoadMipmapTexture` function is what generates the special texture. One of the first lines is this:

```
GLint oldAlign = 0;
glGetIntegerv(GL_UNPACK_ALIGNMENT, &oldAlign);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

The first two lines gets the old alignment, so that we can reset it once we are finished. The last line uses `glPixelStorei`

Note that the GL Image library does provide an alignment value; it is part of the `Dimensions` structure of an image. We have simply not used it yet. In the last tutorial, our row widths were aligned to 4 bytes, so there was no chance of a problem. In this tutorial, our image data is 4-bytes in pixel size, so it is always intrinsically aligned to 4 bytes.

That being said, you should always keep row alignment in mind, particularly when dealing with mipmaps.

Filtering Between Mipmaps

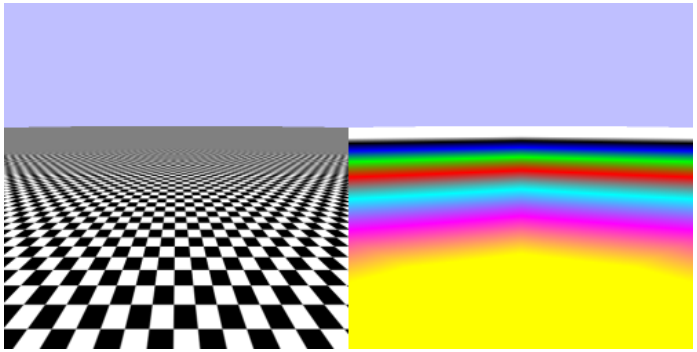
Our mipmap filtering has been a dramatic improvement over previous efforts. However, it does create artifacts. One of particular concern is the change between mipmap levels. It is abrupt and somewhat easy to notice for a moving scene. Perhaps there is a way to smooth that out.

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.

To see this in action, press the **4** key.

Figure 15.9. Linear Mipmap Linear Comparison



That is an improvement. There are still issues to work out, but it is much harder to see where one mipmap ends and another begins.

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.

Filtering Nomenclature

If you are familiar with texture filtering from other sources, you may have heard the terms “bilinear filtering” and “trilinear filtering” before. Indeed, you may know that linear filtering between mipmap levels is commonly called trilinear filtering.

This book does not use that terminology. And for good reason: “trilinear filtering” is a misnomer.

To understand the problem, it is important to understand what “bilinear filtering” means. The “bi” in bilinear comes from doing linear filtering along the two axes of a 2D texture. So there is linear filtering in the S and T directions (remember: standard OpenGL nomenclature calls the 2D texture coordinate axes S and T); since that is two directions, it is called “bilinear filtering”. Thus “trilinear” comes from adding a third direction of linear filtering: between mipmap levels.

Therefore, one could consider using `GL_LINEAR` mag and min filtering to be bilinear, and using `GL_LINEAR_MIPMAP_LINEAR` to be trilinear.

That's all well and good... for 2D textures. But what about for 1D textures? Since 1D textures are one dimensional, `GL_LINEAR` mag and min filtering only filters in one direction: S. Therefore, it would be reasonable to call 1D `GL_LINEAR` filtering simply “linear filtering.” Indeed, filtering between mipmap levels of 1D textures (yes, 1D textures can have mipmaps) would have to be called “bilinear filtering.”

And then there are 3D textures. `GL_LINEAR` mag and min filtering filters in all 3 directions: S, T, and R. Therefore, that would have to be called “trilinear filtering.” And if you add linear mipmap filtering on top of that (yes, 3D textures can have mipmaps), it would be “quadrilinear filtering.”

Therefore, the term “trilinear filtering” means absolutely nothing without knowing what the texture's type is. Whereas `GL_LINEAR_MIPMAP_LINEAR` always has a well-defined meaning regardless of the texture's type.

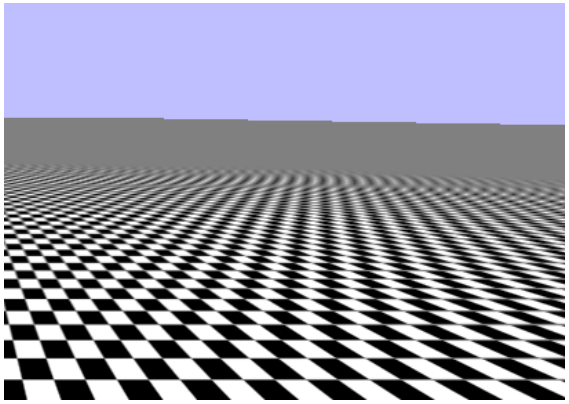
Unlike geometry shaders, which ought to have been called primitive shaders, OpenGL does not enshrine this common misnomer into its API. There is no `GL_TRILINEAR` enum. Therefore, in this book, we can and will use the proper terms for these.

Anisotropy

Linear mipmap filtering is good; it eliminates most of the fluttering and oddities in the distance. The problem is that it replaces a lot of that fluttering with... grey. Mipmap-based filtering works reasonably well, but it tends to over-compensate.

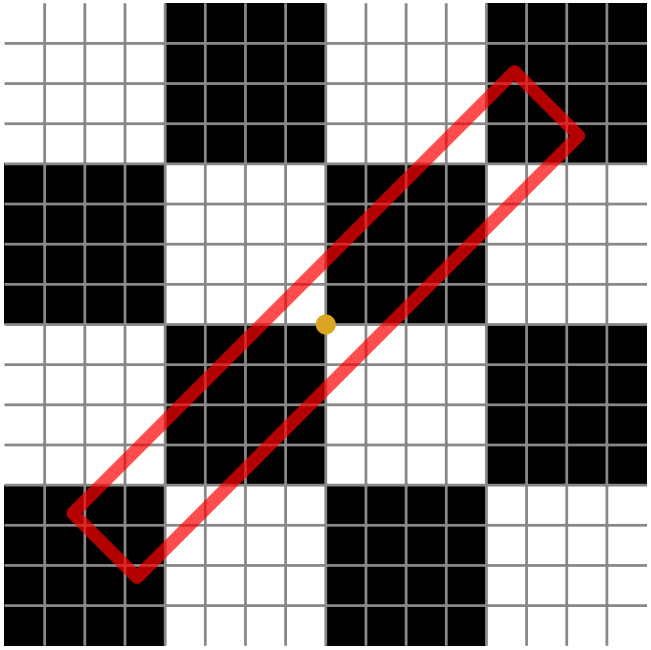
For example, take the diagonal chain of squares at the left or right of the screen. Expand the window horizontally if you need to.

Figure 15.10. Main Diagonal



Pixels that are along this diagonal should be mostly black. As they get farther and farther away, the fragment area becomes more and more distorted length-wise, relative to the texel area:

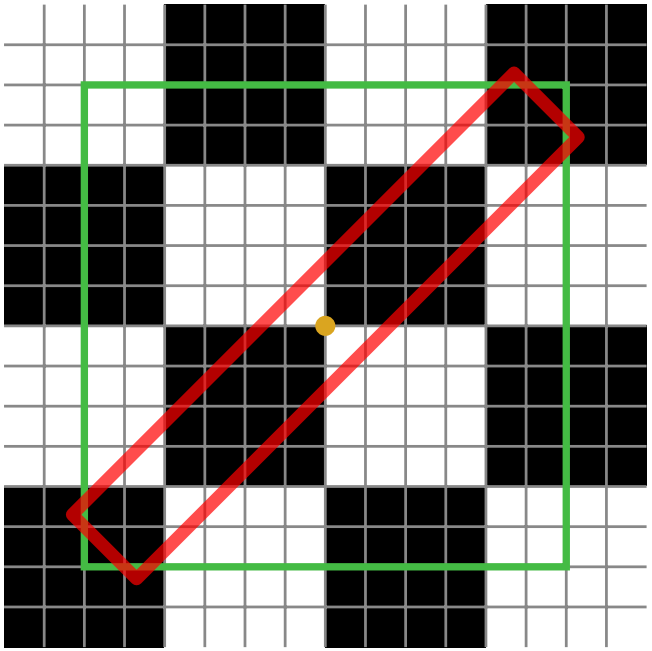
Figure 15.11. Long Fragment Area



With perfect filtering, we should get a value that is mostly black. But instead, we get a much lighter shade of grey. The reason has to do with the specifics of mipmapping and mipmap selection.

Mipmaps are pre-filtered versions of the main texture. The problem is that they are filtered in both directions equally. This is fine if the fragment area is square, but for oblong shapes, mipmap selection becomes more problematic. The particular algorithm used is very conservative. It selects the smallest mipmap level possible for the fragment area. So long, thin areas, in terms of the values fetched by the texture function, will be no different from a square area.

Figure 15.12. Long Fragment with Sample Area



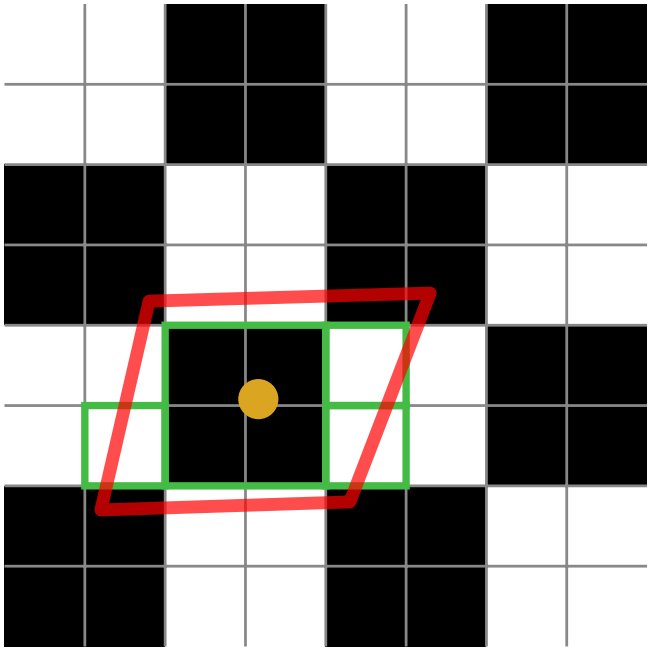
The large square represents the effective filtering box, while the diagonal area is the one that we are actually sampling from. Mipmap filtering can often combine texel values from outside of the sample area, and in this particularly degenerate case, it pulls in texel values from very far outside of the sample area.

This happens when the filter box is not a square. A square filter box is said to be isotropic: uniform in all directions. Therefore, a non-square filter box is anisotropic. Filtering that takes into account the anisotropic nature of a particular filter box is naturally called *anisotropic filtering*.

The OpenGL specification is usually very particular about most things. It explains the details of which mipmap is selected as well as how closeness is defined for linear interpolation between mipmaps. But for anisotropic filtering, the specification is very loose as to exactly how it works.

The general idea is this. The implementation will take some number of samples that approximates the shape of the filter box in the texture. It will select from mipmaps, but only when those mipmaps represent a closer filtered version of area being sampled. Here is an example:

Figure 15.13. Parallelogram Sample Area

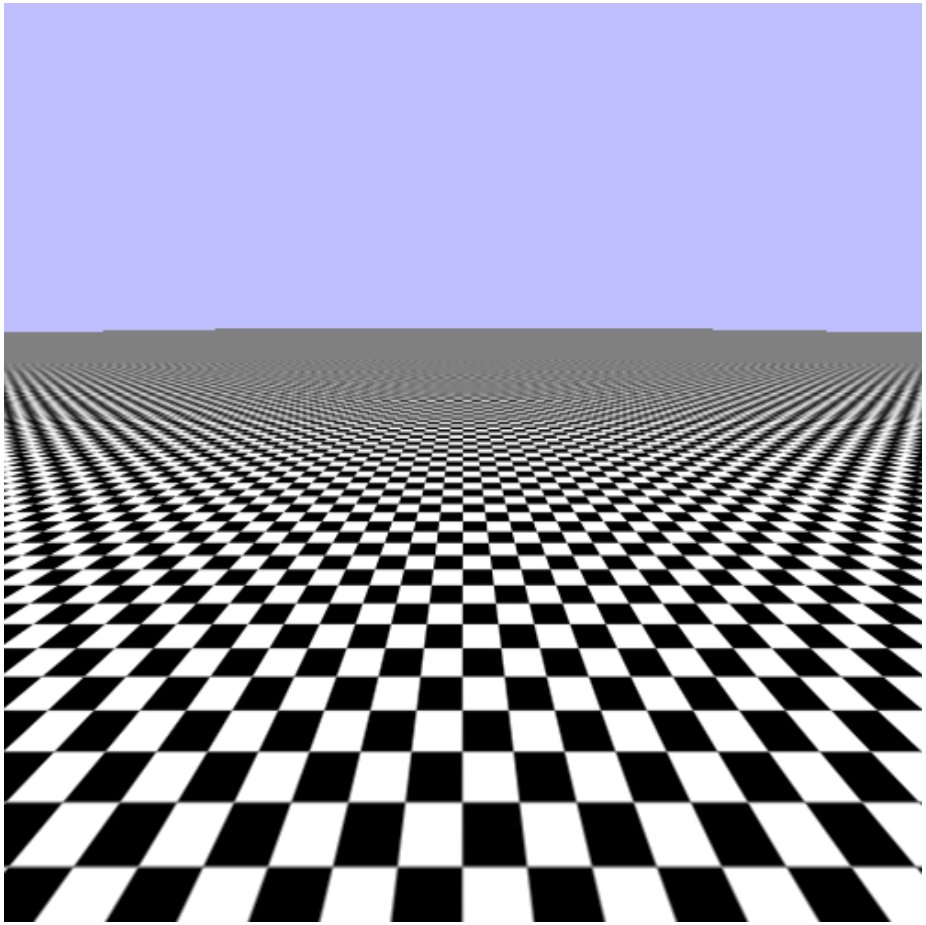


Some of the samples that are entirely within the sample area can use smaller mipmaps to reduce the number of samples actually taken. The above image only needs four samples to approximate the sample area: the three small boxes, and the larger box in the center.

All of the sample values will be averaged together based on a weighting algorithm that best represents that sample's contribution to the filter box. Again, this is all very generally; the specific algorithms are implementation dependent.

Run the tutorial again. The **5** key turns activates a form of anisotropic filtering.

Figure 15.14. Anisotropic Filtering



That's an improvement.

Sample Control

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:

```
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.

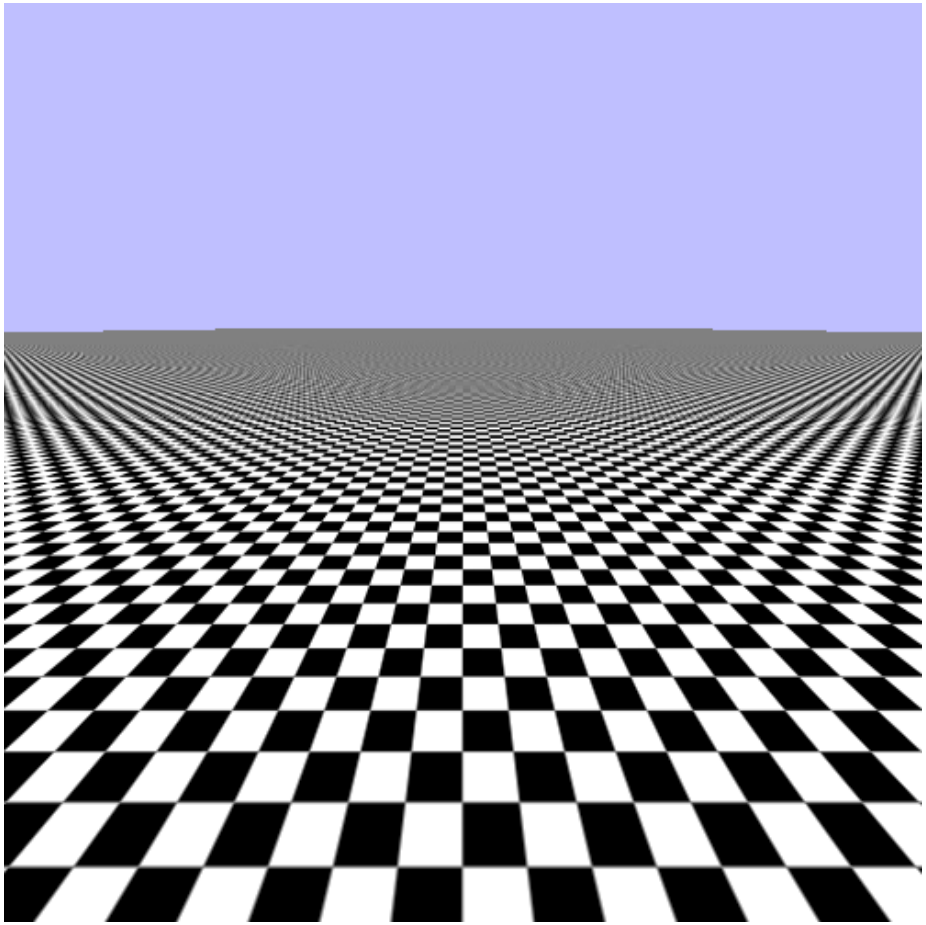
The max anisotropy is a floating point value, in part because the specific nature of anisotropic filtering is left up to the hardware. But in general, you can treat it like an integer value.

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.

```
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);
```

To see the results of this, press the **6** key.

Figure 15.15. Max Anisotropic Filtering



That looks pretty good now. There are still some issues out in the distance. Remember that your image may not look exactly like this one, since the details of anisotropic filtering are implementation specific.

You may be concerned that none of the filtering techniques produces perfect results, even the max anisotropic one. In the distance, the texture still becomes a featureless grey even along the diagonal. The reason is because rendering large checkerboard is perhaps one of the most difficult problems from a texture filtering perspective. This becomes even worse when it is viewed edge on, as we do here.

Indeed, the repeating checkerboard texture was chosen specifically because it highlights the issues in a very obvious way. A more traditional diffuse color texture typically looks much better with reasonable filtering applied. Also, there is one issue that we are currently missing that will be applied in the next tutorial.

A Matter of EXT

You may have noticed the “EXT” suffix on `GL_TEXTURE_MAX_ANISOTROPY_EXT`. This suffix means that this enumerator comes from an *OpenGL extension*. First and foremost, this means that this enumerator is not part of the OpenGL Specification.

An OpenGL extension is a modification of OpenGL exposed by a particular implementation. Extensions have published documents that explain how they change the standard GL specification; this allows users to be able to use them correctly. Because different implementations of OpenGL will implement different sets of extensions, there is a mechanism for querying whether an extension is implemented. This allows user code to detect the availability of certain hardware features and use them or not as needed.

There are several kinds of extensions. There are proprietary extensions; these are created by a particular vendor and are rarely if ever implemented by another vendor. In some cases, they are based on intellectual property owned by that vendor and thus cannot be implemented without explicit permission. The enums and functions for these extensions end with a suffix based on the proprietor of the extension. An NVIDIA-only extension would end in “NV,” for example.

ARB extensions are a special class of extension that is blessed by the OpenGL ARB (who governs the OpenGL specification). These are typically created as a collaboration between multiple members of the ARB. Historically, they have represented functionality that implementations were highly recommended to implement.

EXT extensions are a class between the two. They are not proprietary extensions, and in many cases were created through collaboration among ARB members. Yet at the same time, they are not “blessed” by the ARB. Historically, EXT extensions have been used as test beds for functionality and APIs, to ensure that the API is reasonable before promoting the feature to OpenGL core or to an ARB extension.

The `GL_TEXTURE_MAX_ANISOTROPY_EXT` enumerator is part of the EXT_texture_filter_anisotropic extension. Since it is an extension rather than core functionality, it is usually necessary for the user to detect if the extension is available and only use it if it was. If you look through the tutorial code, you will find no code that does this test.

The reason for that is simply a lack of need. The extension itself dates back to the GeForce 256 (not the GeForce 250GT; the original GeForce), way back in 1999. Virtually all GPUs since then have implemented anisotropic filtering and exposed it through this extension. That is why the tutorial does not bother to check for the presence of this extension; if your hardware can run these tutorials, then it exposes the extension.

If it is so ubiquitous, why has the ARB not adopted the functionality into core OpenGL? Why must anisotropic filtering be an extension that is de facto guaranteed but not technically part of OpenGL? This is because OpenGL must be Open.

The “Open” in OpenGL refers to the availability of the specification, but also to the ability for anyone to implement it. As it turns out, anisotropic filtering has intellectual property issues associated with it. If it were adopted into the core, then core OpenGL would not be able to be implemented without licensing the technology from the holder of the IP. It is not a proprietary extension because none of the ARB members have the IP; it is held by a third party.

Therefore, you may assume that anisotropic filtering is available through OpenGL. But it is technically an extension.

How Mipmap Selection Works

Previously, we discussed mipmap selection and interpolation in terms related to the geometry of the object. That is true, but only when we are dealing with simple texture mapping schemes, such as when the texture coordinates are attached directly to vertex positions. But as we saw in our first tutorial on texturing, texture coordinates can be entirely arbitrary. So how does mipmap selection and anisotropic filtering work then?

Very carefully.

Imagine a 2x2 pixel area of the screen. Now imagine that four fragment shaders, all from the same triangle, are executing for that screen area. Since the fragment shaders from the same triangle are all guaranteed to have the same uniforms and the same code, the only thing that is different among them is the fragment inputs. And because they are executing the same code, you can conceive of them executing in lockstep. That is, each of them executes the same instruction, on their individual dataset, at the same time.

Under that assumption, for any particular value in a fragment shader, you can pick the corresponding 3 other values in the other fragment shaders executing alongside it. If that value is based solely on uniform or constant data, then each shader will have the same value. But if it is based on input values (in part or in whole), then each shader may have a different value, based on how it was computed and what those inputs were.

So, let's look at the texture coordinate value; the particular value used to access the texture. Each shader has one. If that value is associated with the triangle's vertices, via perspective-correct interpolation and so forth, then the *difference* between the shaders' values will represent the window space geometry of the triangle. There are two dimensions for a difference, and therefore there are two differences: the difference in the window space X axis, and the window space Y axis.

These two differences, sometimes called gradients or derivatives, are how mipmapping actually works. If the texture coordinate used is just an interpolated input value, which itself is directly associated with a position, then the gradients represent the geometry of the triangle in window space. If the texture coordinate is computed in more unconventional ways, it still works, as the gradients represent how the texture coordinates are changing across the surface of the triangle.

Having two gradients allows for the detection of anisotropy. And therefore, it provides enough information to reasonably apply anisotropic filtering algorithms.

Now, you may notice that this process is very conditional. Specifically, it requires that you have 4 fragment shaders all running in lock-step. There are two circumstances where that might not happen.

The most obvious is on the edge of a triangle, where a 2x2 block of neighboring fragments is not possible without being outside of the triangle area. This case is actually trivially covered by GPUs. No matter what, the GPU will rasterize each triangle in 2x2 blocks. Even if some of those blocks are not actually part of the triangle of interest, they will still get fragment shader time. This may seem inefficient, but it's reasonable enough in cases where triangles are not incredibly tiny or thin, which is quite often. The results produced by fragment shaders outside of the triangle are simply discarded.

The other circumstance is through deliberate user intervention. Each fragment shader running in lockstep has the same uniforms but different inputs. Since they have different inputs, it is possible for them to execute a conditional branch based on these inputs (an if-statement or other conditional). This could cause, for example, the left-half of the 2x2 quad to execute certain code, while the other half executes different code. The 4 fragment shaders are no longer in lock-step. How does the GPU handle it?

Well... it doesn't. Dealing with this requires manual user intervention, and it is a topic we will discuss later. Suffice it to say, it makes everything complicated.

Performance

Mipmapping has some unexpected performance characteristics. A texture with a full mipmap pyramid will take up ~33% more space than just the base level. So there is some memory overhead. The unexpected part is that this is actually a memory vs. speed tradeoff, as mipmapping usually improves performance.

If a texture is going to be minified significantly, providing mipmaps is a performance benefit. The reason is this: for a highly minified texture, the texture accesses for adjacent fragment shaders will be very far apart. Texture sampling units like texture access patterns where there is a high degree of locality, where adjacent fragment shaders access texels that are very near one another. The farther apart they are, the less useful the optimizations in the texture samplers are. Indeed, if they are far enough apart, those optimizations start becoming performance penalties.

Textures that are used as lookup tables should generally not use mipmaps. But other kinds of textures, like those that provide surface details, can and should where reasonable.

While mipmapping is free, linear mipmap filtering, `GL_LINEAR_MIPMAP_LINEAR`, is generally not free. But the cost of it is rather small these days. For those textures where mipmap interpolation makes sense, it should be used.

Anisotropic filtering is even more costly, as one might expect. After all, it means taking more texture samples to cover a particular texture area. However, anisotropic filtering is almost always implemented adaptively. This means that it will only take extra samples for fragments where it detects that this is necessary. And it will only take enough samples to fill out the area, up to the maximum the user provides of course. Therefore, turning on anisotropic filtering, even just 2x or 4x, only hurts for the fragments that need it.

In Review

In this tutorial, you have learned the following:

- Visual artifacts can appear on objects that have textures mapped to them due to the discrete nature of textures. These artifacts are most pronounced when the texture's mapped size is larger or smaller than its actual size.

- Filtering techniques can reduce these artifacts, transforming visual popping into something more visually palatable. This is most easily done for texture magnification.
- Mipmaps are reduced size versions of images. The purpose behind them is to act as pre-filtered versions of images, so that texture sampling hardware can effectively sample and filter lots of texels all at once. The downside is that it can appear to over-filter textures, causing them to blend down to lower mipmaps in areas where detail could be retained.
- Filtering can be applied between mipmap levels. Mipmap filtering can produce quite reasonable results with a relatively negligible performance penalty.
- Anisotropic filtering attempts to rectify the over-filtering problems with mipmapping by filtering based on the coverage area of the texture access. Anisotropic filtering is controlled with a maximum value, which represents the maximum number of additional samples the texture access will use to compose the final color.

Further Study

Try doing these things with the given programs.

- Use non-mipmap filtering with anisotropic filtering and compare the results with the mipmap-based anisotropic version.
- Change the `GL_TEXTURE_MAX_LEVEL` of the checkerboard texture. Subtract 3 from the computed max level. This will prevent OpenGL from accessing the bottom 3 mipmaps: 1x1, 2x2, and 4x4. See what happens. Notice how there is less grey in the distance, but some of the shimmering from our non-mipmapped version has returned.
- Go back to Basic Texture in the previous tutorial and modify the sampler to use linear mag and min filtering on the 1D texture. See if the linear filtering makes some of the lower resolution versions of the table more palatable. If you were to try this with the 2D lookup texture in Material Texture tutorial, it would cause filtering in both the S and T coordinates. This would mean that it would filter across the shininess of the table as well. Try this and see how this affects the results. Also try using linear filtering on the shininess texture.

Glossary

texture filtering
: The process of fetching the value of a texture at a particular texture coordinate, potentially involving combining multiple texel values together.

  Filtering can happen in two directions: magnification and minification. Magnification happens when the fragment area projected into a texture is smaller than the texel itself. Minification happens when the fragment area projection is larger than a texel.

nearest filtering
: Texture filtering where the texel closest to the texture coordinate is the value returned.

linear filtering
: Texture filtering where the closest texel values in each dimension of the texture are access and linearly interpolated, based on how close the texture coordinate was to those values. For 1D textures, this picks two values and interpolates. For 2D textures, it picks four; for 3D textures, it selects 8.

mipmap, mipmap level
: Subimages of a texture. Each subsequence mipmap of a texture is half the size, rounded down, of the previous image. The largest mipmap is the base level. Many texture types can have mipmaps, but some cannot.

mipmap filtering
: Texture filtering that uses mipmaps. The mipmap choosen when mipmap filtering is used is based on the angle of the texture coordinate, relative to the screen.

  Mipmap filtering can be nearest or linear. Nearest mipmap filtering picks a single mipmap and returns the value pulled from that mipmap. Linear mipmap filtering pics samples from the two nearest mipmaps and linearly interpolates between them. The sample returned in either case can have linear or nearest filtering applied within that mipmap.

anisotropic filtering
: Texture filtering that takes into account the anisotropy of the texture access. This requires taking multiple samples from a surface that covers an irregular area of the surface. This works better with mipmap filtering.

OpenGL extension
: Functionality that is not part of OpenGL proper, but can be conditionally exposed by different implementations of OpenGL.

Chapter 16. Gamma and Textures

In the last tutorial, we had our first picture texture. That was a simple, flat scene; now, we are going to introduce lighting. But before we can do that, we need to have a discussion about what is actually stored in the texture.

The sRGB Colorspace

One of the most important things you should keep in mind with textures is the answer to the question, “what does the data in this texture mean?” In the first texturing tutorial, we had many textures with various meanings. We had:

- A 1D texture that represented the Gaussian model of specular reflections for a specific shininess value.
- A 2D texture that represented the Gaussian model of specular reflections, where the S coordinate represented the angle between the normal and the half-angle vector. The T coordinate is the shininess of the surface.
- A 2D texture that assigned a specular shininess to each position on the surface.

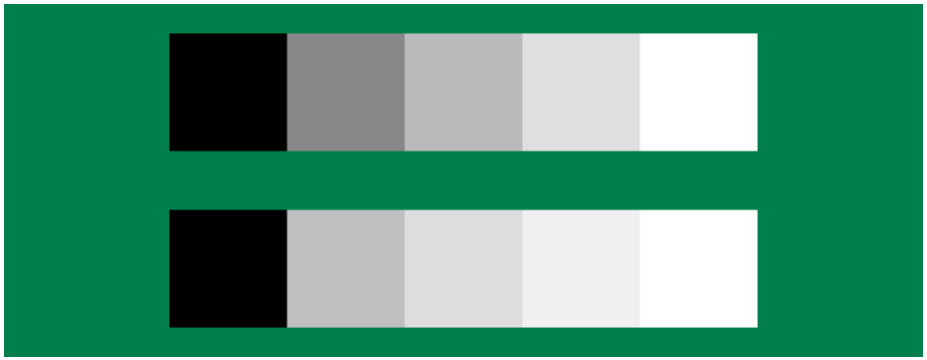
It is vital to know what data a texture stores and what its texture coordinates mean. Without this knowledge, one could not effectively use those textures.

Earlier, we discussed how important colors in a linear colorspace was to getting accurate color reproduction in lighting and rendering. Gamma correction was applied to the output color, to map the linear RGB values to the gamma-correct RGB values the display expects.

At the time, we said that our lighting computations all assume that the colors of the vertices were linear RGB values. Which means that it was important that the creator of the model, the one who put the colors in the mesh, ensure that the colors being added were in fact linear RGB colors. If the modeller failed to do this, if the modeller's colors were in a non-linear RGB colorspace, then the mesh would come out with colors that were substantially different from what he expected.

The same goes for textures, only much moreso. And that is for one very important reason. Load up the Gamma Ramp tutorial.

Figure 16.1. Gamma Ramp



These are just two rectangles with a texture mapped to them. The top one is rendered without the shader's gamma correction, and the bottom one is rendered with gamma correction. These textures are 320x64 in size, and they are rendered at exactly this size.

The texture contains five greyscale color blocks. Each block increases in brightness from the one to its left, in 25% increments. So the second block to the left is 25% of maximum brightness, the middle block is 50% and so on. This means that the second block to the left should appear half as bright as the middle, and the middle should appear half as bright as the far right block.

Gamma correction exists to make linear values appear properly linear on a non-linear display. It corrects for the display's non-linearity. Given everything we know, the bottom rectangle, the one with gamma correction which takes linear values and converts them for proper display, should appear correct. The top rectangle should appear wrong.

And yet, we see the exact opposite. The relative brightness of the various blocks is off in the bottom block, but not the top. Why does this happen?

Because, while the apparent brightness of the texture values increases in 25% increments, the color values that are used by that texture do not. This texture was not created by simply putting 0.0 in the first block, 0.25 in the second, and so forth. It was created by an image editing program. The colors were selected by their *apparent* relative brightness, not by simply adding 0.25 to the values.

This means that the color values have *already been* gamma corrected. They cannot be in a linear colorspace, because the person creating the image selected colors based the colors on their appearance. Since the appearance of a color is affected by the non-linearity of the display, the texture artist was effectively selected post-gamma corrected color values. To put it simply, the colors in the texture are already in a non-linear color space.

Since the top rectangle does not use gamma correction, it is simply passing the pre-gamma corrected color values to the display. It simply works itself out. The bottom rectangle effectively performs gamma correction twice.

This is all well and good, when we are drawing a texture directly to the screen. But if the colors in that texture were intended to represent the diffuse reflectance of a surface as part of the lighting equation, then there is a major problem. The color values retrieved from the texture are non-linear, and all of our lighting equations *need* the input values to be linear.

We could un-gamma correct the texture values manually, either at load time or in the shader. But that is entirely unnecessary and wasteful. Instead, we can just tell OpenGL the truth: that the texture is not in a linear colorspace.

Virtually every image editing program you will ever encounter, from the almighty Photoshop to the humble Paint, displays colors in a non-linear colorspace. But they do not use just any non-linear colorspace; they have settled on a specific colorspace called the *sRGB colorspace.* So when an artist selects a shade of green for example, they are selecting it from the sRGB colorspace, which is non-linear.

How commonly used is the sRGB colorspace? It's built into every JPEG. It's used by virtually every video compression format and tool. It is assumed by virtual every image editing program. In general, if you get an image from an unknown source, it would be perfectly reasonable to assume the RGB values are in sRGB unless you have specific reason to believe otherwise.

The sRGB colorspace is an approximation of a gamma of 2.2. It is not exactly 2.2, but it is close enough that you can display an sRGB image to the screen without gamma correction. Which is exactly what we did with the top rectangle.

Because of the ubiquity of the sRGB colorspace, sRGB decoding logic is built directly into GPUs these days. And naturally OpenGL supports it. This is done via special image formats.

Example 16.1. sRGB Image Format

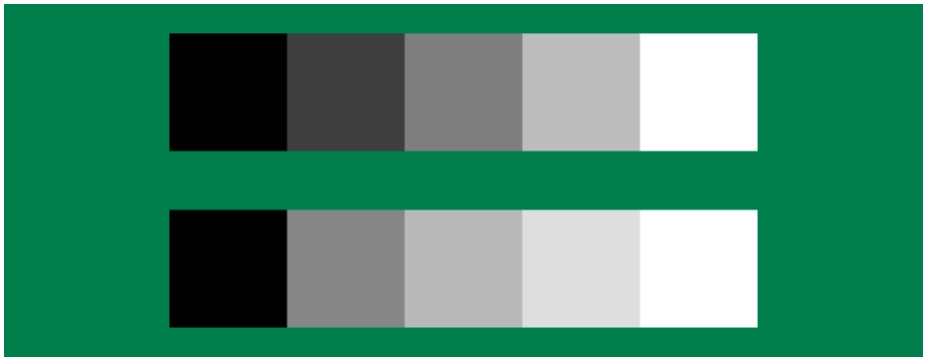
```
std::auto_ptr<glimg::ImageSet> pImageSet(glimg::loaders::stb::LoadFromFile(filename.c_str()));

glimg::SingleImage image = pImageSet->GetImage(0, 0, 0);
glimg::Dimensions dims = image.GetDimensions();

glimg::OpenGLPixelTransferParams pxTrans = glimg::GetUploadFormatType(pImageSet->GetFormat(), 0);

glBindTexture(GL_TEXTURE_2D, g_textures[0]);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, dims.width, dims.height, 0,
    pxTrans.format, pxTrans.type, image.GetImageData());
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, pImageSet->GetMipmapCount() - 1);

glBindTexture(GL_TEXTURE_2D, g_textures[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB8, dims.width, dims.height, 0,
    pxTrans.format, pxTrans.type, image.GetImageData());
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, pImageSet->GetMipmapCount() - 1);

glBindTexture(GL_TEXTURE_2D, 0);
```

This code loads the same texture data twice, but with a different texture format. The first one uses the `GL_RGB8` format, while the second one uses `GL_SRGB8`. The latter identifies the texture's color data as being in the sRGB colorspace.

To see what kind of effect this has on our rendering, you can switch between which texture is used. The **1** key switches the top texture between linear RGB (which from now on will be called lRGB) and sRGB, while **2** does the same for the bottom.

Figure 16.2. Gamma Ramp with sRGB Images



When using the sRGB version for both the top and the bottom, we can see that the gamma correct bottom one is right.

When a texture uses one of the sRGB formats, texture access functions to those textures do things slightly differently. When they fetch a texel, OpenGL automatically linearizes the color from the sRGB colorspace. This is exactly what we want. And the best part is that the linearisation cost is negligible. So there is no need to play with the data or otherwise manually linearize it. OpenGL does it for us.

Note that the shader does not change. It still uses a regular sampler2D, accesses it with a 2D texture coordinate and the `texture` function, etc. The shader does not have to know or care whether the image data is in the sRGB colorspace or a linear one. It simply calls the `texture` function and expects it to return lRGB color values.

Pixel Positioning

There is an interesting thing to note about the rendering in this tutorial. Not only does it use an orthographic projection (unlike most of our tutorials since Tutorial 4), it does something special with its orthographic projection. In the pre-perspective tutorials, the orthographic projection was used essentially by default. We were drawing vertices directly in clip-space. And since the W of those vertices was 1, clip-space is identical to NDC space, and we therefore had an orthographic projection.

It is often useful to want to draw certain objects using window-space pixel coordinates. This is commonly used for drawing text, but it can also be used for displaying images exactly as they appear in a texture, as we do here. Since a vertex shader must output clip-space values, the key is to develop a matrix that transforms window-space coordinates into clip-space. OpenGL will handle the conversion back to window-space internally.

This is done via the `reshape` function, as with most of our projection matrix functions. The computation is actually quite simple.

Example 16.2. Window to Clip Matrix Computation

```
glutil::MatrixStack persMatrix;
persMatrix.Translate(-1.0f, 1.0f, 0.0f);
```



```
persMatrix.Scale(2.0f / w, -2.0f / h, 1.0f);
```

The goal is to transform window-space coordinates into clip-space, which is identical to NDC space since the W component remains 1.0. Window-space coordinates have an X range of [0, w) and Y range of [0, h). NDC space has X and Y ranges of [-1, 1].

The first step is to scale our two X and Y ranges from [0, w/h) to [0, 2]. The next step is to apply a simply offset to shift it over to the [-1, 1] range. Don't forget that the transforms are applied in the reverse order from how they are applied to the matrix stack.

There is one thing to note however. NDC space has +X going right and +Y going up. OpenGL's window-space agrees with this; the origin of window-space is at the lower-left corner. That is nice and all, but many people are used to a top-left origin, with +Y going down.

In this tutorial, we use a top-left origin window-space. That is why the Y scale is negated and why the Y offset is positive (for a lower-left origin, we would want a negative offset).

Note

By negating the Y scale, we flip the winding order of objects rendered. This is normally not a concern; most of the time you are working in window-space, you aren't relying on face culling to strip out certain triangles. In this tutorial, we do not even enable face culling. And oftentimes, when you are rendering with pixel-accurate coordinates, face culling is irrelevant and should be disabled.

Vertex Formats

In all of the previous tutorials, our vertex data has been arrays of floating-point values. For the first time, that is not the case. Since we are working in pixel coordinates, we want to specify vertex positions with integer pixel coordinates. This is what the vertex data for the two rectangles look like:

```
const GLushort vertexData[] = {
     90, 80, 0,     0,
     90, 16, 0,     65535,
    410, 80, 65535, 0,
    410, 16, 65535, 65535,

     90, 176,   0,     0,
     90, 112,   0,     65535,
    410, 176,   65535, 0,
    410, 112,   65535, 65535,
};
```

Our vertex data has two attributes: position and texture coordinates. Our positions are 2D, as are our texture coordinates. These attributes are interleaved, with the position coming first. So the first two columns above are the positions and the second two columns are the texture coordinates.

Instead of floats, our data is composed of GLushorts, which are 2-byte integers. How OpenGL interprets them is specified by the parameters to `glVertexAttribPointer`. It can interpret them in two ways (technically 3, but we don't use that here):

Example 16.3. Vertex Format

```
glBindVertexArray(g_vao);
glBindBuffer(GL_ARRAY_BUFFER, g_dataBufferObject);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_UNSIGNED_SHORT, GL_FALSE, 8, (void*)0);
glEnableVertexAttribArray(5);
glVertexAttribPointer(5, 2, GL_UNSIGNED_SHORT, GL_TRUE, 8, (void*)4);

glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Attribute 0 is our position. We see that the type is not `GL_FLOAT` but `GL_UNSIGNED_SHORT`. This matches the C++ type we use. But the attribute taken by the GLSL shader is a floating point vec2, not an integer 2D vector (which would be ivec2 in GLSL). How does OpenGL reconcile this?

It depends on the fourth parameter, which defines whether the integer value is normalized. If it is set to `GL_FALSE`, then it is not normalized. Therefore, it is converted into a float as though by standard C/C++ casting. An integer value of 90 is cast into a floating-point value of 90.0f. And this is exactly what we want.

Well, that is what we want to for the position; the texture coordinate is a different matter. Normalized texture coordinates should range from [0, 1] (unless we want to employ wrapping of some form). To accomplish this, integer texture coordinates are often, well, normalized. By passing `GL_TRUE` to the fourth parameter (which only works if the third parameter is an integer type), we tell OpenGL to normalize the integer value when converting it to a float.

This normalization works exactly as it does for texel value normalization. Since the maximum value of a GLushort is 65535, that value is mapped to 1.0f, while the minimum value 0 is mapped to 0.0f. So this is just a slightly fancy way of setting the texture coordinates to 1 and 0.

Note that all of this conversion is *free*, in terms of performance. Indeed, it is often a useful performance optimization to compact vertex attributes as small as is reasonable. It is better in terms of both memory and rendering performance, since reading less data from memory takes less time.

OpenGL is just fine with using normalized shorts alongside 32-bit floats, normalized unsigned bytes (useful for colors), etc, all in the same vertex data (though not within the same *attribute*). The above array could have use `GLubyte` for the texture coordinate, but it would have been difficult to write that directly into the code as a C-style array. In a real application, one would generally not get meshes from C-style arrays, but from files.

sRGB and Mipmaps

The principle reason lighting functions require lRGB values is because they perform linear operations. They therefore produce inaccurate results on non-linear colors. This is not limited to lighting functions; *all* linear operations on colors require a lRGB value to produce a reasonable result.

One important linear operation performed on texel values is filtering. Whether magnification or minification, non-nearest filtering does some kind of linear arithmetic. Since this is all handled by OpenGL, the question is this: if a texture is in an sRGB format, does OpenGL's texture filtering occur *before* converting the texel values to lRGB or after?

The answer is quite simple: filtering comes after linearizing. So it does the right thing.

Note

It's not quite that simple. The OpenGL specification technically leaves it undefined. However, if your hardware can run these tutorials without modifications (ie: your hardware is OpenGL 3.3 capable), then odds are it will do the right thing. It is only on pre-3.0 hardware where this is a problem.

A bigger question is this: do you generate the mipmaps correctly for your textures? Mipmap generation was somewhat glossed over in the last tutorial, as tools generally do this for you. In general, mipmap generation involves some form of linear operation on the colors. For this process to produce correct results for sRGB textures, it needs to linearize the sRGB color values, perform its filtering on them, then convert them back to sRGB for storage.

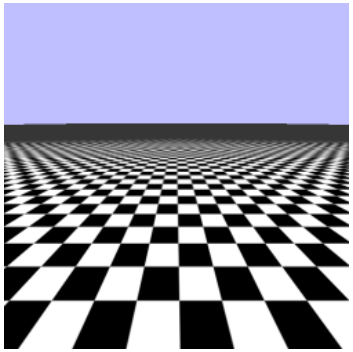
Unless you are writing texture processing tools, this question is answered by asking your texture tools themselves. Most freely available texture tools are completely unaware of non-linear colorspaces. You can tell which ones are aware based on the options you are given at mipmap creation time. If you can specify a gamma for your texture, or if there is some setting to specify that the texture's colors are sRGB, then the tool can do the right thing. If no such option exists, then it cannot. For sRGB textures, you should use a gamma of 2.2, which is what sRGB approximates.

Note

The DDS plugin for GIMP is a good, free tool that is aware of linear colorspaces. NVIDIA's command-line texture tools, also free, are as well.

To see how this can affect rendering, load up the Gamma Checkers project.

Figure 16.3. Gamma Checkers



This works like the filtering tutorials. The **1** and **2** keys respectively select linear mipmap filtering and anisotropic filtering (using the maximum possible anisotropy).

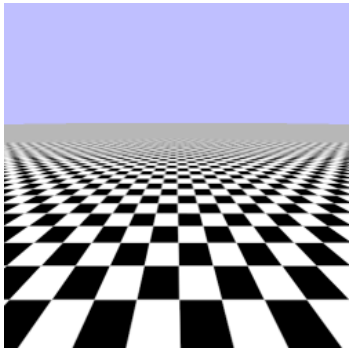
We can see that this looks a bit different from the last time we saw it. The distant grey field is much darker than it was. This is because we are using sRGB colorspace textures. While the white and black are the same in sRGB (1.0 and 0.0 respectively), a 50% blend of them (0.5) is not. The sRGB texture assumes the 0.5 color is the sRGB 0.5, so it becomes darker than we would expect.

Initially, we render with no gamma correction. To toggle gamma correction, press the **a** key. This restores the view to what we saw previously.

However, the texture we are using is actually wrong. 0.5, as previously stated, is not the sRGB color for a 50% blend of black and white. In the sRGB colorspace, that color would be ~0.73. The texture is wrong because its mipmaps were not generated in the correct colorspace.

To switch to a texture who's mipmaps were properly generated, press the **g** key.

Figure 16.4. Gamma Correct with Gamma Mipmaps



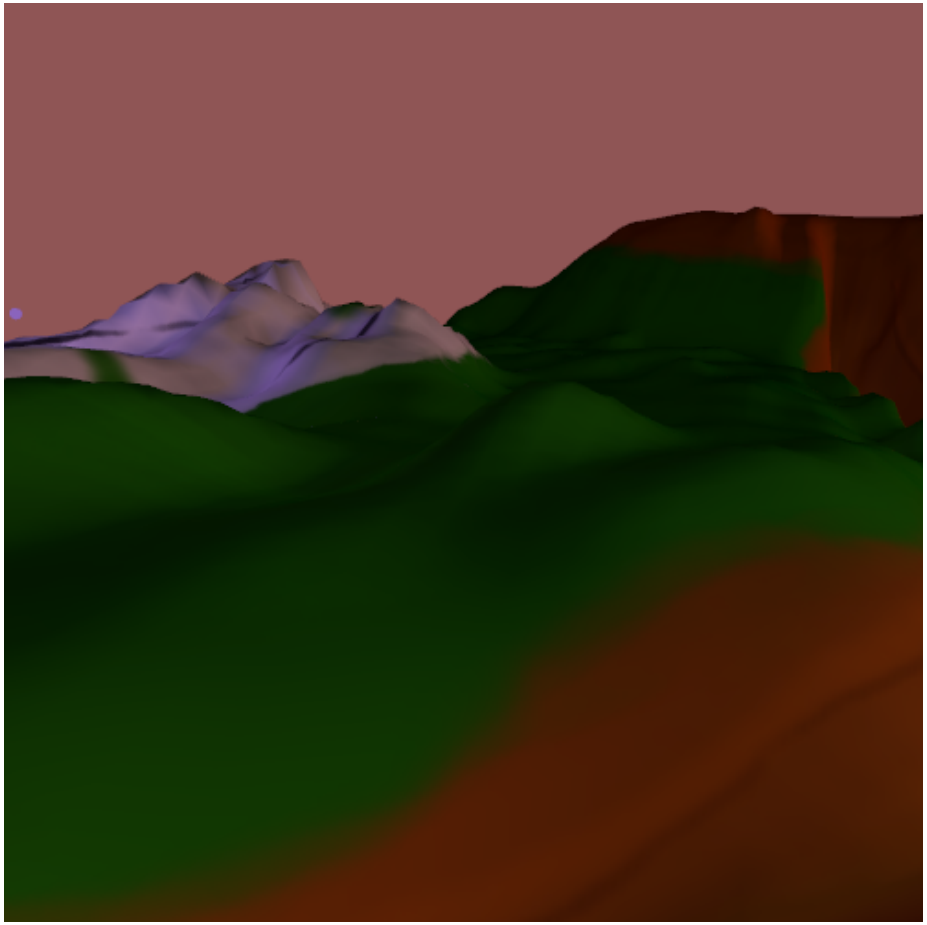
This still looks different from the last tutorial. Which naturally tells us that not rendering with gamma correction before was actually a problem, as this version looks much better. The grey blends much better with the checkerboard, as the grey is now correctly halfway between white and black. The take-home point here is that ensuring linearity in all stages of the pipeline is always important. This includes mipmap generation.

sRGB and the Screen

Thus far, we have seen how to use sRGB textures to store gamma-corrected images, such that they are automatically linearized upon being fetched from a shader. Since the sRGB colorspace closely approximates a gamma of 2.2, if we could use an sRGB image as the image we render to, we would automatically get gamma correction without having to put it into our shaders. But this would require two things: the ability to specify that the screen image is sRGB, and the ability to state that we are outputting linear values and want them converted to the sRGB colorspace when stored.

Naturally, OpenGL provides both of these. To see how they work, load up the last project, Gamma Landscape. This shows off some textured terrain with a day/night cycle and a few lights running around.

Figure 16.5. Gamma Landscape



It uses the standard mouse-based controls to move around. As before, the **1** and **2** keys respectively select linear mipmap filtering and anisotropic filtering. The main feature is the non-shader-based gamma correction. This is enabled by default and can be toggled by pressing the **SpaceBar**.

sRGB Screen Image. The process for setting this up is a bit confusing, but is ultimately quite simple for our tutorials. The OpenGL specification specifies how to use the OpenGL rendering system, but it does not specify how to *create* the OpenGL rendering system. That is relegated to platform-specific APIs. Therefore, while code that uses OpenGL is platform-neutral, that code is ultimately dependent on platform-specific initialization code to create the OpenGL context.

These tutorials rely on FreeGLUT for setting up the OpenGL context and managing the platform-specific APIs. The `framework.cpp` file is responsible for doing the initialization setup work, telling FreeGLUT exactly how we want our screen image set up. In order to allow different tutorials to adjust how we set up our FreeGLUT screen image, the framework calls the `defaults` function.

Example 16.4. Gamma Landscape defaults Function

```
unsigned int defaults(unsigned int displayMode, int &width, int &height)
{
    return displayMode | GLUT_SRGB;
}
```

The `displayMode` argument is a bitfield that contains the standard FreeGLUT display mode flags set up by the framework. This function must return that bitfield, and all of our prior tutorials have returned it unchanged. Here, we change it to include the `GLUT_SRGB` flag. That flag tells FreeGLUT that we want the screen image to be in the sRGB colorspace.

Linear to sRGB Conversion. This alone is insufficient. We must also tell OpenGL that our shaders will be writing linear colorspace values and that these values should be converted to sRGB before being written to the screen. This is done with a simple `glEnable` command:

Example 16.5. Enable sRGB Conversion

```
if(g_useGammaDisplay)
    glEnable(GL_FRAMEBUFFER_SRGB);
else
    glDisable(GL_FRAMEBUFFER_SRGB);
```

The need for this is not entirely obvious, especially since we cannot manually turn off sRGB-to-linear conversion when reading from textures. The ability to disable linear-to-sRGB conversion for screen rendering is useful when we are drawing something directly in the sRGB colorspace. For example, it is often useful to have parts of the interface drawn directly in the sRGB colorspace, while the actual scene being rendered uses color conversion.

Note that the color conversion is just as free in terms of performance as it is for texture reads. So you should not fear using this as much and as often as reasonable.

Having this automatic gamma correction is better than manual gamma correction because it covers everything that is written to the screen. In prior tutorials, we had to manually gamma correct the clear color and certain other colors used to render solid objects. Here, we simply enable the conversion and everything is affected.

The process of ensuring a linear pipeline from texture creation, through lighting, through to the screen is commonly called *gamma-correct texturing*. The name is a bit of a misnomer, as “texturing” is not a requirement; we have been gamma-correct since Tutorial 12's introduction of that concept (except for Tutorial 15, where we looked at filtering). However, textures are the primary source of potential failures to maintain a linear pipeline, as many image formats on disc have no way of saying if the image data is in sRGB or linear. So the name still makes some sense.

In Review

In this tutorial, you have learned the following:

- At all times, it is important to remember what the meaning of the data stored in a texture is.
- Most of the time, when a texture represents actual colors, those colors are in the sRGB colorspace. An appropriate image format must be selected.
- Linear operations like filtering must be performed on linear values. All of OpenGL's operations on sRGB textures do this.
- Similarly, the generation of mipmaps, a linear operation, must perform conversion from sRGB to lRGB, do the filtering, and then convert back. Since OpenGL does not (usually) generate mipmaps, it is incumbent upon the creator of the image to ensure that the mipmaps were generated properly.

- Lighting operations need linear values.
- The framebuffer can also be in the sRGB colorspace. OpenGL also requires a special enable when doing so, thus allowing for some parts of the rendering to be sRGB encoded and other parts not.

OpenGL Functions of Note

glEnable/ glDisable(GL_FRAMEBUFFER_SRGB)
: Enables/disables the conversion from linear to sRGB. When this is enabled, colors written by the fragment shader to an sRGB image are assumed to be linear. They are therefore converted into the sRGB colorspace. When this is disabled, the colors written by the fragment shader are assumed to already be in the sRGB colorspace; they are written exactly as given.

Glossary

sRGB colorspace
: A non-linear RGB colorspace, which approximates a gamma of 2.2. The *vast* majority of image editing programs and operating systems work in the sRGB colorspace by default. Therefore, most images you will encounter will be in the sRGB colorspace.

  OpenGL has the ability to work with sRGB textures and screen images directly. Accesses to sRGB textures will return lRGB values, and writes to sRGB screen images can be converted from linear to sRGB values, so long as a proper enable is used.

gamma-correct texturing
: The process of ensuring that all textures, images, and other sources and destinations of colors (such as vertex attributes) are either already in lRGB or are converted to/from the linear colorspace as needed. Textures in the sRGB format are part of that, but so is rendering to an sRGB screen image (or manually doing gamma correction). These provide automatic correction. Manual correction may need to be applied to vertex color attributes, and for proper interpolation, this correction needs to be applied before interpolation.

Chapter 17. Spotlight on Textures

Previously, we have seen textures used to vary surface parameters. But we can use textures to vary something else: light intensity. In this way, we can simulate light sources who's intensity changes with something more than just distance from the light.

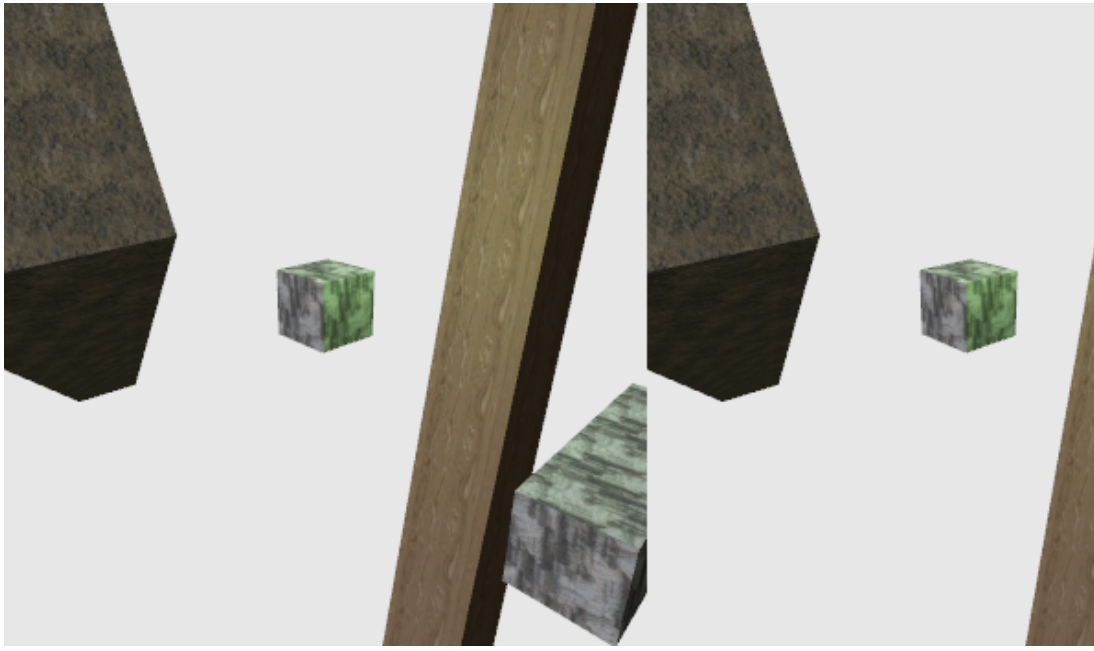
Our first effort in varying light intensity with textures will be to build an incandescent flashlight. The light beam from a flashlight is not a single solid intensity, due to the way the mirrors focus the light. A texture is the simplest way to define this pattern of light intensity.

Post-Projection Space

Before we can look at how to use a texture to make a flashlight, we need to make a short digression. Perspective projection will be an important part of how we make a texture into a flashlight, so we need to revisit perspective projection. Specifically, we need to look at what happens when transforming after a perspective projection operation.

Open up the project called Double Projection. It renders four objects, using various textures, in a scene with a single directional light and a green point light.

Figure 17.1. Double Projection



This tutorial displays two images of the same scene. The image on the left is the view of the scene from one camera, while the image on the right is the view of the scene from another camera. The difference between the two cameras is mainly in where the camera transformation matrices are applied.

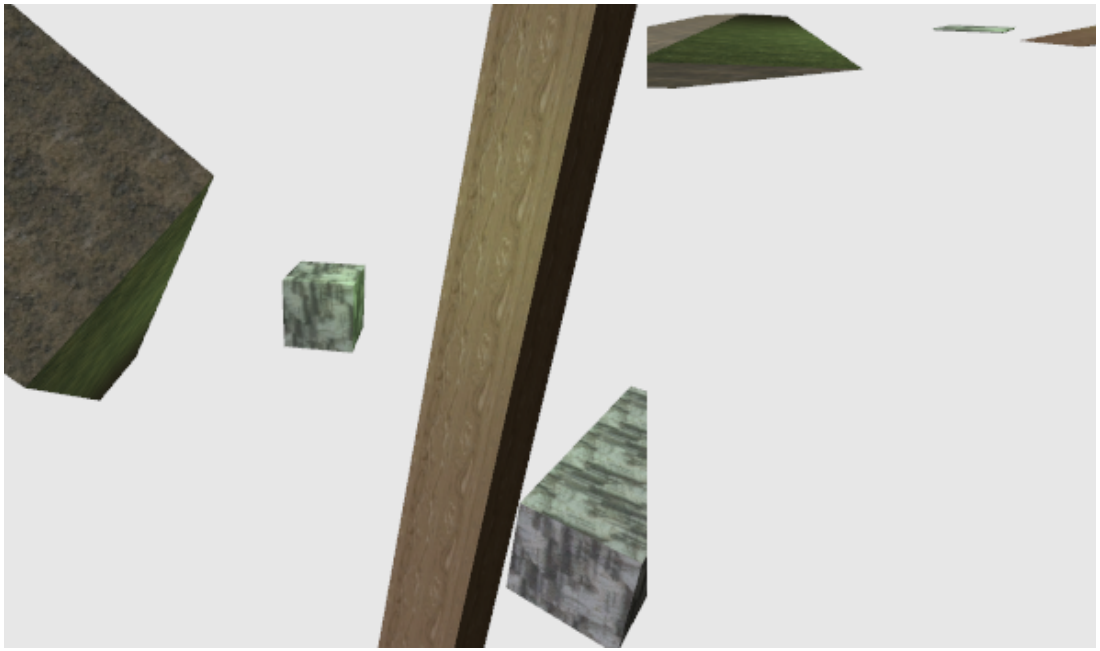
The left camera works normally. It is controlled by the left mouse button, the mouse wheel, and the WASD keys, as normal. The right camera however provides the view direction that is applied after the perspective projection matrix. The sequence of transforms thus looks like this: Model -> Left Camera -> Projection -> Right Camera. The right camera is controlled by the right mouse button; only orientation controls work on it.

The idea is to be able to look at the shape of objects in normalized device coordinate (NDC) space after a perspective projection. NDC space is a [-1, 1] box centered at the origin; by rotating objects in NDC space, you will be able to see what those objects look like from a different perspective. Pressing the **SpaceBar** will reset the right camera back to a neutral view.

Note that post-perspective projection space objects are very distorted, particularly in the Z direction. Also, recall one of the fundamental tricks of the perspective projection: it rescales objects based on their Z-distance from the camera. Thus, objects that are farther away are physically smaller in NDC space than closer ones. Thus, rotating NDC space around will produce results that are not intuitively what you might expect and may be very disorienting at first.

For example, if we rotate the right camera to an above view, relative to whatever the left camera is, we see that all of the objects seems to shrink down into a very small width.

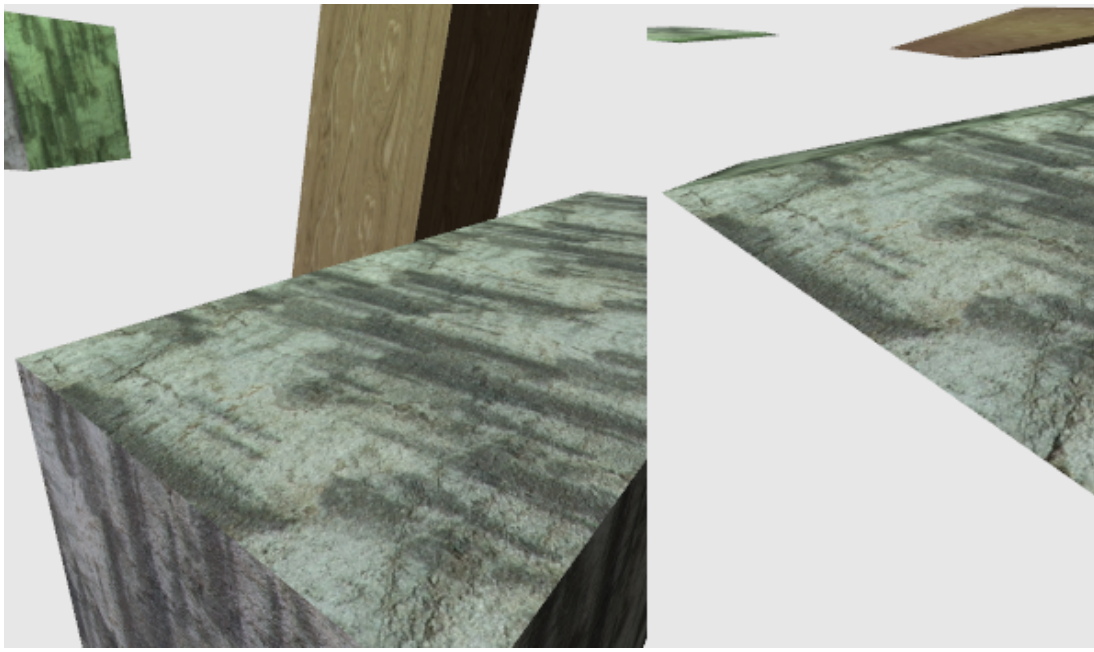
Figure 17.2. Top View Projection



This is due to the particulars of the perspective projection's work on the Z coordinate. The Z coordinate in NDC space is the result of the clip-space Z divided by the negative of the camera-space Z. This forces it into the [-1, 1] range, but the clip-space Z also is affected by the zNear and zFar of the perspective matrix. The wider these are, the more narrowly the Z is compressed. Objects farther from the camera are compressed into smaller ranges of Z; we saw this in our look at the effect of the camera Z-range on precision. Close objects use more of the [-1, 1] range than those farther away.

This can be seen by moving the left camera close to an object. The right camera, from a top-down view, has a much thicker view of that object in the Z direction.

Figure 17.3. Near View Projection



Pressing the **Y** key will toggle depth clamping in the right camera. This can explain some of the unusual things that will be seen there. Sometimes the wrong objects will appear on top of each other; when this happens, it is almost always due to a clamped depth.

The reason why depth clamping matters so much in the right screen is obvious if you think about it. NDC space is a [-1, 1] square. But that is not the NDC space we actually render to. We are rendering to a rotated portion of this space. So the actual [-1, 1] space that gets clipped or clamped is different from the one we see. We are effectively rotating a square and cutting off any parts of it that happen to be outside of the square viewing area. This is easy to see in the X and Y directions, but in Z, it results in some unusual views.

Scene Graphs

This is the first code in the tutorial to use the scene graph part of the framework. The term *scene graph* refers to a piece of software that manages a collection of objects, typically in some kind of object hierarchy. In this case, the `Scene.h` part of the framework contains a class that loads an XML description of a scene. This description includes meshes, shaders, and textures to load. These assets are then associated with named objects within the scene. So a mesh combined with a shader can be rendered with one or more textures.

The purpose of this system is to remove a *lot* of the boilerplate code from the tutorial files. The setup work for the scene graph is far less complicated than the setup work seen in previous tutorials.

As an example, here is the scene graph to load and link a particular shader:

Example 17.1. Scene Graph Shader Definition

```
<prog
    xml:id="p_unlit"
```

```
    vert="Unlit.vert"
    frag="Unlit.frag"
    model-to-camera="modelToCameraMatrix">
    <block name="Projection" binding="0"/>
</prog>
```

The `xml:id` gives it a name; this is used by objects in the scene to refer to this program. It also provides a way for other code to talk to it. Most of the rest is self-explanatory. `model-to-camera` deserves some explanation.

Rendering the scene graph is done by calling the scene graph's render function with a camera matrix. Since the objects in the scene graph store their own transformations, the scene graph combines each object's local transform with the given camera matrix. But it still needs to know how to provide that matrix to the shader. Thus, `model-to-camera` specifies the name of the mat4 uniform that receives the model-to-camera transformation matrix. There is a similar matrix for normals that is given the inverse-transpose of the model-to-camera matrix.

The `block` element is the way we associate a uniform block in the program with a uniform block binding point. There is a similar element for `sampler` that specifies which texture unit that a particular GLSL sampler is bound to.

Objects in scene graph systems are traditionally called “nodes,” and this scene graph is no exception.

Example 17.2. Scene Graph Node Definition

```
<node
    name="spinBar"
    mesh="m_longBar"
    prog="p_lit"
    pos="-7 0 8"
    orient="-0.148446 0.554035 0.212003 0.791242"
    scale="4">
    <texture name="t_stone_pillar" unit="0" sampler="anisotropic"/>
</node>
```

Nodes have a number of properties. They have a name, so that other code can reference them. They have a mesh that they render and a program they use to render that mesh. They have a position, orientation, and scale transform. The orientation is specified as a quaternion, with the W component specified last (this is different from how glm::fquat specifies it. The W there comes first). The order of these transforms is scale, then orientation, then translation.

This node also has a texture bound to it. `t_stone_pillar` was a texture that was loaded in a `texture` command. The `unit` property specifies the texture unit to use. And the `sampler` property defines which of the predefined samplers to use. In this case, it uses a sampler with ansiotropic filtering to the maximum degree allowed by the hardware. The texture wrapping modes of this sampler are to wrap the S and T coordinates.

This is what the C++ setup code looks like for the entire scene:

Example 17.3. Double Projection LoadAndSetupScene

```
std::auto_ptr<Framework::Scene> pScene(new Framework::Scene("dp_scene.xml"));

std::vector<Framework::NodeRef> nodes;
nodes.push_back(pScene->FindNode("cube"));
nodes.push_back(pScene->FindNode("rightBar"));
nodes.push_back(pScene->FindNode("leaningBar"));
nodes.push_back(pScene->FindNode("spinBar"));

AssociateUniformWithNodes(nodes, g_lightNumBinder, "numberOfLights");
SetStateBinderWithNodes(nodes, g_lightNumBinder);

GLuint unlit = pScene->FindProgram("p_unlit");
Framework::Mesh *pSphereMesh = pScene->FindMesh("m_sphere");
```



```
//No more things that can throw.
g_spinBarOrient = nodes[3].NodeGetOrient();
g_unlitProg = unlit;
g_unlitModelToCameraMatrixUnif = glGetUniformLocation(unlit, "modelToCameraMatrix");
g_unlitObjectColorUnif = glGetUniformLocation(unlit, "objectColor");

std::swap(nodes, g_nodes);
nodes.clear(); //If something was there already, delete it.

std::swap(pSphereMesh, g_pSphereMesh);

Framework::Scene *pOldScene = g_pScene;
g_pScene = pScene.release();
pScene.reset(pOldScene); //If something was there already, delete it.
```

This code does some fairly simple things. The scene graph system is good, but we still need to be able to control uniforms not in blocks manually from external code. Specifically in this case, the number of lights is a uniform, not a uniform block. To do this, we need to use a uniform state binder, `g_lightNumBinder`, and set it into all of the nodes in the scene. This binder allows us to set the uniform for all of the objects (regardless of which program they use).

The `p_unlit` shader is never actually used in the scene graph; we just use the scene graph as a convenient way to load the shader. Similarly, the `m_sphere` mesh is not used in a scene graph node. We pull references to both of these out of the graph and use them ourselves where needed. We extract some uniform locations from the unlit shader, so that we can draw unlit objects with colors.

Note

The code as written here is designed to be exception safe. Most of the functions that find nodes by name will throw if the name is not found. What this exception safety means is that it is easy to make the scene reloadable. It only replaces the old values in the global variables after executing all of the code that could throw an exception. This way, the entire scene, along with all meshes, textures, and shaders, can be reloaded by pressing **Enter**. If something goes wrong, the new scene will not be loaded and an error message is displayed.

Two of the objects in the scene rotate. This is easily handled using our list of objects. In the `display` method, we access certain nodes and change their transforms them:

```
g_nodes[0].NodeSetOrient(glm::rotate(glm::fquat(),
    360.0f * g_timer.GetAlpha(), glm::vec3(0.0f, 1.0f, 0.0f)));

g_nodes[3].NodeSetOrient(g_spinBarOrient * glm::rotate(glm::fquat(),
    360.0f * g_timer.GetAlpha(), glm::vec3(0.0f, 0.0f, 1.0f)));
```

We simply set the orientation based on a timer. For the second one, we previously stored the object's orientation after loading it, and use that as the reference. This allows us to rotate about its local Z axis.

Multiple Scenes

The split-screen trick used here is actually quite simple to pull off. It's also one of the advantages of the scene graph: the ability to easily re-render the same scene multiple times.

The first thing that must change is that the projection matrix cannot be set in the old `reshape` function. That function now only sets the new width and height of the screen into global variables. This is important because we will be using two projection matrices.

The projection matrix used for the left scene is set up like this:

Example 17.4. Left Projection Matrix

```
glm::ivec2 displaySize(g_displayWidth / 2, g_displayHeight);
```

```
{
    glutil::MatrixStack persMatrix;
    persMatrix.Perspective(60.0f, (displaySize.x / (float)displaySize.y), g_fzNear, g_fzFar);

    ProjectionBlock projData;
    projData.cameraToClipMatrix = persMatrix.Top();

    glBindBuffer(GL_UNIFORM_BUFFER, g_projectionUniformBuffer);
    glBufferData(GL_UNIFORM_BUFFER, sizeof(ProjectionBlock), &projData, GL_STREAM_DRAW);
    glBindBuffer(GL_UNIFORM_BUFFER, 0);
}

glViewport(0, 0, (GLsizei)displaySize.x, (GLsizei)displaySize.y);
g_pScene->Render(modelMatrix.Top());
```

Notice that `displaySize` uses only half of the width. And this half width is passed into the `glViewport` call. It is also used to generate the aspect ratio for the perspective projection matrix. It is the `glViewport` function that causes our window to be split into two halves.

What is more interesting is the right projection matrix computation:

Example 17.5. Right Projection Matrix

```
{
    glutil::MatrixStack persMatrix;
    persMatrix.ApplyMatrix(glm::mat4(glm::mat3(g_persViewPole.CalcMatrix())));
    persMatrix.Perspective(60.0f, (displaySize.x / (float)displaySize.y), g_fzNear, g_fzFar);

    ProjectionBlock projData;
    projData.cameraToClipMatrix = persMatrix.Top();

    glBindBuffer(GL_UNIFORM_BUFFER, g_projectionUniformBuffer);
    glBufferData(GL_UNIFORM_BUFFER, sizeof(ProjectionBlock), &projData, GL_STREAM_DRAW);
    glBindBuffer(GL_UNIFORM_BUFFER, 0);
}

if(!g_bDepthClampProj)
    glDisable(GL_DEPTH_CLAMP);
glViewport(displaySize.x + (g_displayWidth % 2), 0,
    (GLsizei)displaySize.x, (GLsizei)displaySize.y);
g_pScene->Render(modelMatrix.Top());
glEnable(GL_DEPTH_CLAMP);
```

Notice that we first take the camera matrix from the perspective view and apply it to the matrix stack before the perspective projection itself. Remember that transforms applied to the stack happen in *reverse* order. This means that vertices are projected into 4D homogeneous clip-space coordinates, then are transformed by a matrix. Only the rotation portion of the right camera matrix is used. The translation is removed by the conversion to a mat3 (which takes only the top-left 3x3 part of the matrix, which if you recall contains the rotation), then turns it back into a mat4.

Notice also that the viewport's X location is biased based on whether the display's width is odd or not (`g_displayWidth % 2` is 0 if it is even, 1 if it is odd). This means that if the width is stretched to an odd number, there will be a one pixel gap between the two views of the scene.

Intermediate Projection

One question may occur to you: how is it possible for our right camera to provide a rotation in NDC space, if it is being applied to the end of the projection matrix? After all, the projection matrix goes from camera space to *clip*-space. The clip-space to NDC space transform is done by OpenGL after our vertex shader has done this matrix multiply. Do we not need the shader to divide the clip-space values by W, then do the rotation?

Obviously not, since this code works. But just because code happens to work doesn't mean that it should. So let's see if we can prove that it does. To do this, we must prove this:

$$\mathbf{T} * \frac{\vec{V}}{w} = = \frac{\mathbf{T} * \vec{V}}{w'}$$

This might look like a simple proof by inspection due to the associative nature of these, but it is not. The reason is quite simple: w and w' may not be the same. The value of w is the fourth component of v; w' is the fourth component of what results from T*v. If T changes w, then the equation is not true. But at the same time, if T doesn't change w, if w == w', then the equation is true.

Well, that makes things quite simple. We simply need to ensure that our T does not alter w. Matrix multiplication tells us that w' is the dot product of V and the bottom row of T.

$$w' = T_{03} * \text{v.x} + T_{13} * \text{v.y} + T_{23} * \text{v.z} + T_{33} * \text{v.w}$$

Therefore, if the bottom row of T is (0, 0, 0, 1), then w == w'. And therefore, we can use T before the division. Fortunately, the only matrix we have that has a different bottom row is the projection matrix, and T is the rotation matrix we apply after projection.

So this works, as long as we use the right matrices. We can rotate, translate, and scale post-projective clip-space exactly as we would post-projective NDC space. Which is good, because we get to preserve the w component for perspective-correct interpolation.

The take-home lesson here is very simple: projections are not that special as far as transforms are concerned. Post-projective space is mostly just another space. It may be a 4-dimensional homogeneous coordinate system, and that may be an odd thing to fully understand. But that does not mean that you can't apply a regular matrix to objects in this space.

Projective Texture

In order to create our flashlight effect, we need to do something called *projective texturing*. Projective texturing is a special form of texture mapping. It is a way of generating texture coordinates for a texture, such that it appears that the texture is being projected onto a scene, in much the same way that a film projector projects light. Therefore, we need to do two things: implement projective texturing, and then use the value we sample from the projected texture as the light intensity.

The key to understanding projected texturing is to think backwards, compared to the visual effect we are trying to achieve. We want to take a 2D texture and make it look like it is projected onto the scene. To do this, we therefore do the opposite: we project the *scene* onto the 2D texture. We want to take the vertex positions of every object in the scene and project them into the space of the texture.

Since this is a perspective projection operation, and it involves transforming vertex positions, naturally we need a matrix. This is math we already know: we have vertex positions in model space. We transform them to a camera space, one that is different from the one we use to view the scene. Then we use a perspective projection matrix to transform them to clip-space; both the matrix and this clip-space are again different spaces from what we use to render the scene. Once perspective divide later, and we're done.

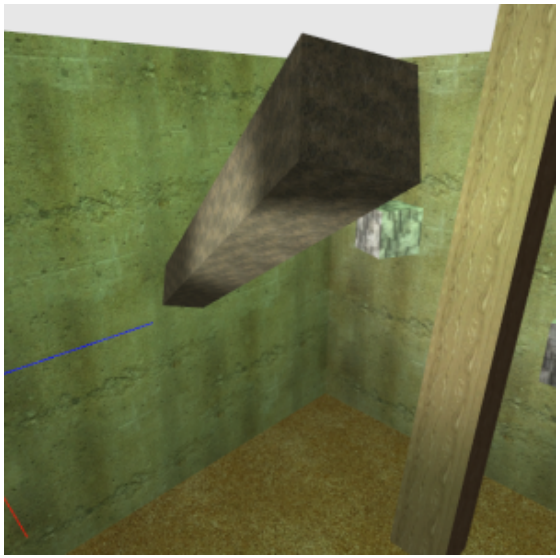
That last part is the small stumbling block. See, after the perspective divide, the visible world, the part of the world that is projected onto the texture, lives in a [-1, 1] sized cube. That is the size of NDC space, though it is again a different NDC space from the one we use to render. The problem is that the range of the texture coordinates, the space of the 2D texture itself, is [0, 1].

This is why we needed the prior discussion of post-projective transforms. Because we need to do a post-projective transform here: we have to transform the XY coordinates of the projected position from [-1, 1] to [0, 1] space. And again, we do not want to have to perform the perspective divide ourselves; OpenGL has special functions for texture accesses with a divide. Therefore, we encode the translation and scale as a post-projective transformation. As previously demonstrated, this is mathematically identical to doing the transform after the division.

This entire process represents a new kind of light. We have seen directional lights, which are represented by a light intensity coming from a single direction. And we have seen point lights, which are represented by a position in the world which casts light in all directions. What we are defining now is typically called a *spotlight*: a light that has a position, direction, and oftentimes a few other fields that limit the size and nature of the spot effect. Spotlights cast light on a cone-shaped area.

We implement spotlights via projected textures in the Projected Light project. This tutorial uses a similar scene to the one before, though with slightly different numbers for lighting. The main difference, scene wise, is the addition of a textured background box.

Figure 17.4. Projected Light



The camera controls work the same way as before. The projected flashlight, represented by the red, green, and blue axes, is moved with the IJKL keyboard keys, with O and U moving up and down, respectively. The right mouse button rotates the flashlight around; the blue line points in the direction of the light. The flashlight's position and orientation are built around the camera controls, so it rotates around a point in front of the flashlight. It translates relative to its current facing as well. As usual, holding down the **Shift** key will cause the flashlight to move more slowly.

Pressing the **G** key will toggle all of the regular lighting on and off. This makes it easier to see just the light from our projected texture.

Flashing the Light

Let us first look at how we achieve the projected texture effect. We want to take the model space positions of the vertices and project them onto the texture. However, there is one minor problem: the scene graph system provides a transform from model space into the visible camera space. We need a transform to our special projected texture camera space, which has a different position and orientation.

We resolve this by being clever. We already have positions in the viewing camera space. So we simply start there and construct a matrix from view camera space into our texture camera space.

Example 17.6. View Camera to Projected Texture Transform

```
glutil::MatrixStack lightProjStack;
//Texture-space transform
lightProjStack.Translate(0.5f, 0.5f, 0.0f);
lightProjStack.Scale(0.5f, 0.5f, 1.0f);
//Project. Z-range is irrelevant.
lightProjStack.Perspective(g_lightFOVs[g_currFOVIndex], 1.0f, 1.0f, 100.0f);
//Transform from main camera space to light camera space.
lightProjStack.ApplyMatrix(lightView);
lightProjStack.ApplyMatrix(glm::inverse(cameraMatrix));

g_lightProjMatBinder.SetValue(lightProjStack.Top());
```


Reading the modifications to `lightProjStack` in bottom-to-top order, we begin by using the inverse of the view camera matrix. This transforms all of our vertex positions back to world space, since the view camera matrix is a world-to-camera matrix. We then apply the world-to-texture-camera matrix. This is followed by a projection matrix, which uses an aspect ratio of 1.0. The last two transforms move us from [-1, 1] NDC space to the [0, 1] texture space.

The zNear and zFar for the projection matrix are almost entirely irrelevant. They need to be within the allowed ranges (strictly greater than 0, and zFar must be larger than zNear), but the values themselves are meaningless. We will discard the Z coordinate entirely later on.

We use a matrix uniform binder to associate that transformation matrix with all of the objects in the scene. This is all we need to do to set up the projection, as far as the matrix math is concerned.

Our vertex shader (`projLight.vert`) takes care of things in the obvious way:

```
lightProjPosition = cameraToLightProjMatrix * vec4(cameraSpacePosition, 1.0);
```

Note that this line is part of the vertex shader; `lightProjPosition` is passed to the fragment shader. One might think that the projection would work best in the fragment shader, but doing it per-vertex is actually just fine. The only time one would need to do the projection per-fragment would be if one was using imposters or was otherwise modifying the depth of the fragment. Indeed, because it works per-vertex, projected textures were a preferred way of doing cheap lighting in many situations.

In the fragment shader, `projLight.frag`, we want to use the projected texture as a light. We have the `ComputeLighting` function in this shader from prior tutorials. All we need to do is make our projected light appear to be a regular light.

```
PerLight currLight;
currLight.cameraSpaceLightPos = vec4(cameraSpaceProjLightPos, 1.0);
currLight.lightIntensity =
    textureProj(lightProjTex, lightProjPosition.xyw) * 4.0;

currLight.lightIntensity = lightProjPosition.w > 0 ?
 currLight.lightIntensity : vec4(0.0);
```

We create a simple structure that we fill in. Later, we pass this structure to `ComputeLighting`, and it does the usual thing.

The view camera space position of the projected light is passed in as a uniform. It is necessary for our flashlight to properly obey attenuation, as well as to find the direction towards the light.

The next line is where we do the actual texture projection. The `textureProj` is a texture accessing function that does projective texturing. Even though `lightProjTex` is a sampler2D (for 2D textures), the texture coordinate has three dimensions. All forms of `textureProj` take one extra texture coordinate compared to the regular `texture` function. This extra texture coordinate is divided into the previous one before being used to access the texture. Thus, it performs the perspective divide for us.

Note

Mathematically, there is virtually no difference between using `textureProj` and doing the divide ourselves and calling `texture` with the results. While there may not be a mathematical difference, there very well may be a performance difference. There may be specialized hardware that does the division much faster than the general-purpose opcodes in the shader. Then again, there may not. However, using `textureProj` will certainly be no slower than `texture` in the general case, so it's still a good idea.

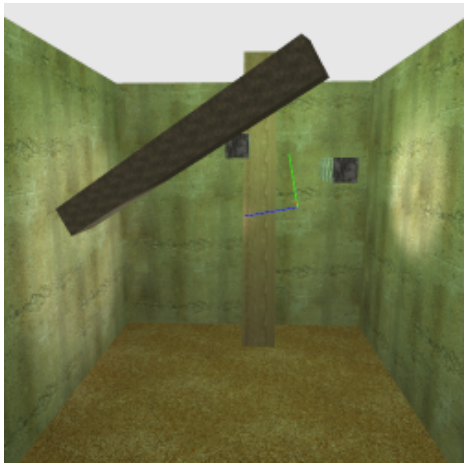
Notice that the value pulled from the texture is scaled by 4.0. This is done because the color values stored in the texture are clamped to the [0, 1] range. To bring it up to our high dynamic range, we need to scale the intensity appropriately.

The texture being projected is bound to a known texture unit globally; the scene graph already associates the projective shader with that texture unit. So there is no need to do any special work in the scene graph to make objects use the texture.

The last statement is special. It compares the W component of the interpolated position against zero, and sets the light intensity to zero if the W component is less than or equal to 0. What is the purpose of this?

It stops this from happening:

Figure 17.5. Back Projected Light



The projection math doesn't care what side of the center of projection an object is on; it will work either way. And since we do not actually do clipping on our texture projection, we need some way to prevent this from happening. We effectively need to do some form of clipping.

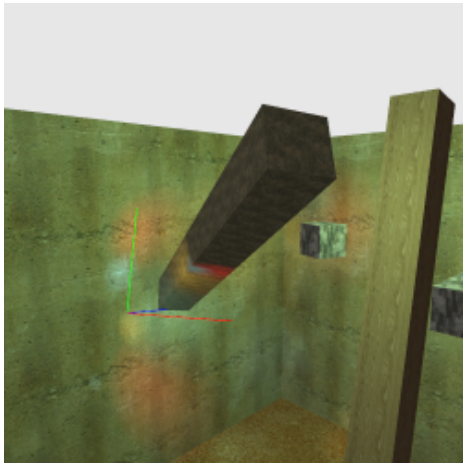
Recall that, given the standard projection transform, the W component is the negation of the camera-space Z. Since the camera in our camera space is looking down the negative Z axis, all positions that are in front of the camera must have a W > 0. Therefore, if W is less than or equal to 0, then the position is behind the camera.

Spotlight Tricks

The size of the flashlight can be changed simply by modifying the field of view in the texture projection matrix. Pressing the **Y** key will increase the FOV, and pressing the **N** key will decrease it. An increase to the FOV means that the light is projected over a greater area. At a large FOV, we effectively have an entire hemisphere of light.

Another interesting trick we can play is to have multi-colored lights. Press the **2**; this will change to a texture that contains spots of various different colors.

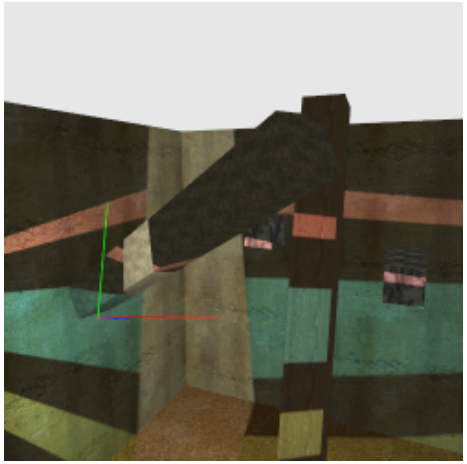
Figure 17.6. Colored Spotlight



This kind of complex light emitter would not be possible without using a texture. Well it could be possible without textures, but it would require a lot more processing power than a few matrix multiplies, a division in the fragment shader, and a texture access. Press the **1** key to go back to the flashlight texture.

There is one final issue that can and will crop up with projected textures: what happens when the texture coordinates are outside of the [0, 1] boundary. With previous textures, we used either `GL_CLAMP_TO_EDGE` or `GL_REPEAT` for the S and T texture coordinate wrap modes. Repeat is obviously not a good idea here; thus far, our sampler objects have been clamping to the texture's edge. That worked fine because our edge texels have all been zero. To see what happens when they are not, press the **3** key.

Figure 17.7. Edge Clamped Light



That rather ruins the effect. Fortunately, OpenGL does provide a way to resolve this. It gives us a way to say that texels fetched outside of the [0, 1] range should return a particular color. As before, this is set up with the sampler object:

Example 17.7. Border Clamp Sampler Objects

```
glSamplerParameteri(g_samplers[1], GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glSamplerParameteri(g_samplers[1], GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);

float color[4] = {0.0f, 0.0f, 0.0f, 1.0f};
glSamplerParameterfv(g_samplers[1], GL_TEXTURE_BORDER_COLOR, color);
```

The S and T wrap modes are set to `GL_CLAMP_TO_BORDER`. Then the border's color is set to zero. To toggle between the edge clamping sampler and the border clamping one, press the **H** key.

Figure 17.8. Border Clamped Light

That's much better now.

Line Drawing

You may have noticed that the position and orientation of the light was shown by three lines forming the three directions of an axis. These are a new primitive type: lines.

Lines have a uniform width no matter how close or far away they are from the camera. Point primitives are defined by one vertex, triangle primitives by 3. So it makes sense that lines are defined by two vertices.

Just as triangles can come in strips and fans, lines have their own variations. `GL_LINES` are like `GL_TRIANGLES`: a list of independent lines, with each line coming from individual pairs of vertices. `GL_LINE_STRIP` represents a sequence of lines attached head to tail; every vertex has a line to the previous vertex and the next in the list. `GL_LINE_LOOP` is like a strip, except the last and first vertices are also connected by a line.

This is all encapsulated in the Framework's `Mesh` class. The axis used here (and later on in the tutorials) is a simple

Pointing Projections

Spotlights represent a light that has position, direction, and perhaps an FOV and some kind of aspect ratio. Through projective texturing, we can make spotlights that have arbitrary light intensities, rather than relying on uniform values or shader functions to compute light intensity. That is all well and good for spotlights, but there are other forms of light that might want varying intensities.

It doesn't really make sense to vary the light intensity from a directional light. After all, the while point of directional lights is that they are infinitely far away, so all of the light from them is uniform, in both intensity and direction.

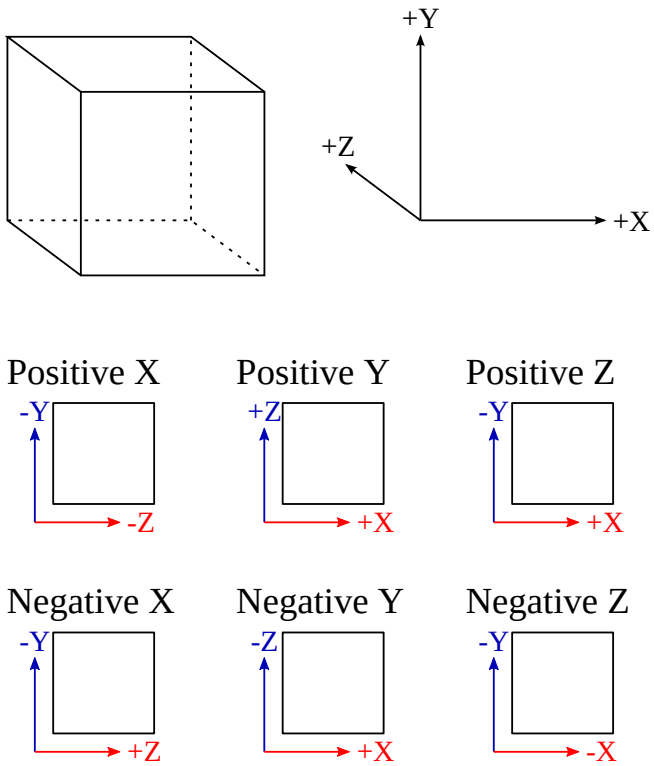
Varying the intensity of a point light is a more reasonable possibility. We can vary the point light's intensity based on one of two possible parameters: the position of the light and the direction from the light towards a point in the scene. The latter seems far more useful; it represents a light that may cast more or less brightly in different directions.

To do this, what we need is a texture that we can effectively access via a direction. While there are ways to convert a 3D vector direction into a 2D texture coordinate, we will not use any of them. We will instead use a special texture type creates specifically for exactly this sort of thing.

The common term for this kind of texture is *cube map*, even though it is a texture rather than a mapping of a texture. A cube map texture is a texture where every mipmap level is 6 2D images, not merely one. Each of the 6 images represents one of the 6 faces of a cube. The texture coordinates for a cube map are a 3D vector direction; the texture sampling hardware selects which face to sample from and which texel to pick based on the direction.

It is important to know how the 6 faces of the cube map fit together. OpenGL defines the 6 faces based on the X, Y, and Z axes, in the positive and negative directions. This diagram explains the orientation of the S and T coordinate axes of each of the faces, relative to the direction of the faces in the cube.

Figure 17.9. Cube Map Face Orientation



This information is vital for knowing how to construct the various faces of a cube map.

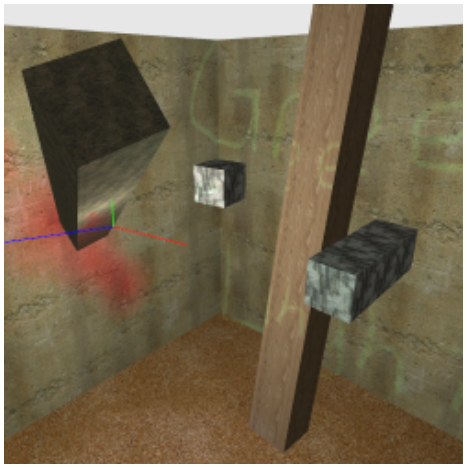
To use a cube map to specify the light intensity changes for a point light, we simply need to do the following. First, we get the direction from the light to the surface point of interest. Then we use that direction to sample from the cube map. From there, everything is normal.

The issue is getting the direction from the light to the surface point. Before, a point light had no orientation, and this made sense. It cast light uniformly in all directions, so even if it had an orientation, you would never be able to tell it was there. Now that our light intensity can vary, the point light now needs to be able to orient the cube map.

The easiest way to handle this is a simple transformation trick. The position and orientation of the light represents a space. If we transform the position of objects into that space, then the direction from the light can easily be obtained. The light's position relative to itself is zero, after all. So we need to transform positions from some space into the light's space. We will see exactly how this is done momentarily.

Cube map point lights are implemented in the Cube Point Light project. This puts a fixed point light using a cube map in the middle of the scene. The orientation of the light can be changed with the right mouse button.

Figure 17.10. Cube Point Light



This cube texture has various different light arrangements on the different sides. One side even has green text on it. As before, you can use the **G** key to toggle the non-cube map lights off.

Pressing the **2** key switches to a texture that looks somewhat resembles a planetarium show. Pressing **1** switches back to the first texture.

Cube Texture Loading

We have seen how 2D textures get loaded over the course of 3 tutorials now, so we use GL Image's functions for creating a texture directly from ImageSet. Cube map textures require special handling, so let's look at this now.

Example 17.8. Cube Texture Loading

```
std::string filename(Framework::FindFileOrThrow(g_texDefs[tex].filename));
std::auto_ptr<glimg::ImageSet> pImageSet(glimg::loaders::dds::LoadFromFile(filename.c_str()));

glBindTexture(GL_TEXTURE_CUBE_MAP, g_lightTextures[tex]);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAX_LEVEL, 0);

glimg::Dimensions dims = pImageSet->GetDimensions();
GLenum imageFormat = (GLenum)glimg::GetInternalFormat(pImageSet->GetFormat(), 0);

for(int face = 0; face < 6; ++face)
{
    glimg::SingleImage img = pImageSet->GetImage(0, 0, face);
    glCompressedTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + face,
        0, imageFormat, dims.width, dims.height, 0,
        img.GetImageByteSize(), img.GetImageData());
}

glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
```

The DDS format is one of the few image file formats that can actually store all of the faces of a cube map. Similarly, the glimg::ImageSet class can store cube map faces.

The first step after loading the cube map faces is to bind the texture to the `GL_TEXTURE_CUBE_MAP` texture binding target. Since this cube map is not mipmapped (yes, cube maps can have mipmaps), we set the base and max mipmap levels to zero. The call to `glimg::GetInternalFormat` is used to allow GL Image to tell us the OpenGL image format that corresponds to the format of the loaded texture data.

From there, we loop over the 6 faces of the texture, get the SingleImage for that face, and load each face into the OpenGL texture. For the moment, pretend the call to `glCompressedTexImage2D` is a call to `glTexImage2D`; they do similar things, but the final few parameters are different. It may seem odd to call a TexImage2D function when we are uploading to a cube map texture. After all, a cube map texture is a completely different texture type from 2D textures.

However, the "TexImage" family of functions specify the dimensionality of the image data they are allocating an uploading, not the specific texture type. Since a cube map is simply 6 sets of 2D image images, it uses the "TexImage2D" functions to allocate the faces and mipmaps. Which face is specified by the first parameter.

OpenGL has six enumerators of the form `GL_TEXTURE_CUBE_MAP_POSITIVE/NEGATIVE_X/Y/Z`. These enumerators are ordered, starting with positive X, so we can loop through all of them by adding the numbers [0, 5] to the positive X enumerator. That is what we do above. The order of these enumerators is:

1. POSITIVE_X
2. NEGATIVE_X
3. POSITIVE_Y
4. NEGATIVE_Y
5. POSITIVE_Z
6. NEGATIVE_Z

This mirrors the order that the ImageSet stores them in (and DDS files, for that matter).

The samplers for cube map textures also needs some adjustment:

```
glSamplerParameteri(g_samplers[0], GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glSamplerParameteri(g_samplers[0], GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glSamplerParameteri(g_samplers[0], GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Cube maps take 3D texture coordinates, so wrap modes must be specified for each of the three dimensions of texture coordinates. Since this cube map has no mipmaps, the filtering is simply set to `GL_LINEAR`.

Texture Compression

Now we will take a look at why we are using `glCompressedTexImage2D`. And that requires a discussion of image formats and sizes.

Images take up a lot of memory. And while disk space and even main memory are fairly generous these days, GPU memory is always at a premium. Especially if you have lots of textures and those textures are quite large. The smaller that texture data can be, the more and larger textures you can have in a complex scene.

The first stop for making this data smaller is to use a smaller image format. For example, the standard RGB color format stores each channel as an 8-bit unsigned integer. This is usually padded out to make it 4-byte aligned, or a fourth component (alpha) is added, making for an RGBA color. That's 32-bits per texel, which is what `GL_RGBA8` specifies. A first pass for making this data smaller is to store it with fewer bits. OpenGL provides `GL_RGB565` for those who do not need the fourth component, or `GL_RGBA4` for those who do. Both of these use 16-bits per texel.

They both also can produce unpleasant visual artifacts for the textures. Plus, OpenGL does not allow such textures to be in the sRGB colorspace; there is no `GL_SRGB565` format.

For files, this is a solved problem. There are a number of traditional compressed image formats: PNG, JPEG, GIF, etc. Some are lossless, meaning that the exact input image can be reconstructed. Others are lossy, which means that only an approximation of the image can be returned. Either way, these all formats have their benefits and downsides. But they are all better, in terms of visual quality and space storage, than using 16-bit per texel image formats.

They also have one other thing in common: they are absolutely terrible for *textures*, in terms of GPU hardware. These formats are designed to be decompressed all at once; you decompress the entire image when you want to see it. GPUs don't want to do that. GPUs generally access textures in pieces; they access certain sections of a mipmap level, then access other sections, etc. GPUs gain their performance by being incredibly parallel: multiple different invocations of fragment shaders can be running simultaneously. All of them can be accessing different textures and so forth.

Stopping that processes to decompress a 50KB PNG would pretty much destroy rendering performance entirely. These formats may be fine for storing files on disk. But they are simply not good formats for being stored compressed in graphics memory.

Instead, there are special formats designed specifically for compressing textures. These *texture compression* formats are designed specifically to be friendly for texture accesses. It is easy to find the exact piece of memory that stores the data for a specific texel. It takes no more than 64 bits of data to decompress any one texel. And so forth. These all combine to make texture compression formats useful for saving graphics card memory, while maintaining reasonable image quality.

The regular `glTexImage2D` function is not capable of directly uploading compressed texture data. The pixel transfer information, the last three parameters of `glTexImage2D`, is simply not appropriate for dealing with compressed texture data. Therefore, OpenGL uses a different function for uploading texture data that is already compressed.

```
glCompressedTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + face,
    0, imageFormat, dims.width, dims.height, 0,
    img.GetImageByteSize(), img.GetImageData());
```

Instead of taking OpenGL enums that define what the format of the compressed data is, `glCompressedTexImage2D`'s last two parameters are very simple. They specify how big the compressed image data is in bytes and provide a pointer to that image data. That is because `glCompressedTexImage2D` does not allow for format conversion; the format of the pixel data passed to it must exactly match what the image format says it is. This also means that the `GL_UNPACK_ALIGNMENT` has no effect on compressed texture uploads.

Cube Texture Space

Creating the cube map texture was just the first step. The next step is to do the necessary transformations. Recall that the goal is to transform the vertex positions into the space of the texture, defined relative to world space by a position and orientation. However, we ran into a problem previously, because the scene graph only provides a model-to-camera transformation matrix.

This problem still exists, and we will solve it in exactly the same way. We will generate a matrix that goes from camera space to our cube map light's space.

Example 17.9. View Camera to Light Cube Texture

```
glutil::MatrixStack lightProjStack;
lightProjStack.ApplyMatrix(glm::inverse(lightView));
lightProjStack.ApplyMatrix(glm::inverse(cameraMatrix));

g_lightProjMatBinder.SetValue(lightProjStack.Top());

glm::vec4 worldLightPos = lightView[3];
glm::vec3 lightPos = glm::vec3(cameraMatrix * worldLightPos);

g_camLightPosBinder.SetValue(lightPos);
```

This code is rather simpler than the prior time. Again reading bottom up, we transform by the inverse of the world-to-camera matrix, then we transform by the inverse of the light matrix. The `lightView` matrix is inverted because the matrix is ordinarily designed to go from light space to world space. So we invert it to get the world-to-light transform. The light's position in world space is taken similarly.

The vertex shader (cubeLight.vert) is about what you would expect:

```
lightSpacePosition = (cameraToLightProjMatrix * vec4(cameraSpacePosition, 1.0)).xyz;
```

The `lightSpacePosition` is output from the vertex shader and interpolated. Again we find that this interpolates just fine, so there is no need to do this transformation per-fragment.

The fragment shader code (`cubeLight.frag`) is pretty simple. First, we have to define our GLSL samplers:

```
uniform sampler2D diffuseColorTex;
uniform samplerCube lightCubeTex;
```

Because cube maps are a different texture type, they have a different GLSL sampler type as well. Attempting to use texture with the one type on a sampler that uses a different type results in unpleasantness. It's usually easy enough to keep these things straight, but it can be a source of errors or non-rendering.

The code that fetches from the cube texture is as follows:

```
PerLight currLight;
currLight.cameraSpaceLightPos = vec4(cameraSpaceProjLightPos, 1.0);

vec3 dirFromLight = normalize(lightSpacePosition);
currLight.lightIntensity =
    texture(lightCubeTex, dirFromLight) * 6.0f;
```

We simply normalize the light-space position, since the cube map's space has the light position at the origin. We then use the `texture` to access the cubemap, the same one we used for 2D textures. This is possible because GLSL overloads the `texture` based on the type of sampler. So when `texture` is passed a samplerCube, it expects a vec3 texture coordinate.

In Review

In this tutorial, you have learned the following:

- Vertex positions can be further manipulated after a perspective projection. Thus the perspective transform is not special. The shape of objects in post-projective space can be unusual and unexpected.
- Textures can be projected onto meshes. This is done by transforming those meshes into the space of the texture, which is equivalent to transforming the texture into the space of the meshes. The transform is governed by its own camera matrix, as well as a projection matrix and a post-projective transform that transforms it into the [0, 1] range of the texture.
- Cube maps are textures that have 6 face images for every mipmap level. The 6 faces are arranged in a cube. Texture coordinates are effectively directions of a vector centered within the cube. Thus a cube map can provide a varying value based on a direction in space.

Further Study

Try doing these things with the given programs.

- In the spotlight project, change the projection texture coordinate from a full 4D coordinate to a 2D. Do this by performing the divide-by-W step directly in the vertex shader, and simply pass the ST coordinates to the fragment shader. Just use `texture` instead of `textureProj` in the fragment shader. See how that affects things. Also, try doing the perspective divide in the fragment shader and see how this differs from doing it in the vertex shader.
- In the spotlight project, change the interpolation style from `smooth` to `noperspective`. See how non-perspective-correct interpolation changes the projection.
- Instead of using a projective texture, build a lighting system for spot lights entirely within the shader. It should have a maximum angle; the larger the angle, the wider the spotlight. It should also have an inner angle that is smaller than the maximum angle. This the the point where the light starts falling off. At the maximum angle, the light intensity goes to zero; at the minimum angle, the light intensity is full. The key here is remembering that the dot product between the spotlight's direction and the direction from the surface to the light is the cosine of the angle between the two vectors. The `acos` function can be used to compute the angle (in radians) from the cosine.

Further Research

Cube maps are fairly old technology. The version used in GPUs today derive from the Renderman standard and earlier works. However, before hardware that allowed cubemaps became widely available, there were alternative techniques that were used to achieve similar effects.

The basic idea behind all of these is to transform a 3D vector direction into a 2D texture coordinate. Note that converting a 3D direction into a 2D plane is a problem that was encountered long before computer graphics. It is effectively the global mapping problem: how you create a 2D map of a 3D spherical surface. All of these techniques introduce some distance distortion into the 2D map. Some distortion is more acceptable in certain circumstances than others.

One of the more common pre-cube map techniques was sphere mapping. This required a very heavily distorted 2D texture, so the results left something to be desired. But the 3D-to-2D computations were simple enough to be encoded into early graphics hardware, or performed quickly on the CPU, so it was acceptable as a stop-gap. Other techniques, such as dual paraboloid mapping, were also used. The latter used a pair of textures, so they ate up more resources. But they required less heavy distortions of the texture, so in some cases, they were a better tradeoff.

OpenGL Functions of Note

glCompressedTexImage2D	Allocates a 2D image of the given size and mipmap for the current texture, using the given compressed image format, and uploads compressed pixel data. The pixel data must exactly match the format of the data defined by the compressed image format.

GLSL Functions of Note

`vec4 textureProj(sampler texSampler, vec texCoord);`

Accesses the texture associated with *`texSampler`*, using post-projective texture coordinates specified by *`texCoord`*. The sampler type can be many of the sampler types, but not samplerCube, among a few others. The texture coordinates are in homogeneous space, so they have one more components than the number of dimensions of the texture. Thus, the number of components in *`texCoord`* for a sampler of type sampler1D is vec2. For sampler2D, it is vec3.

Glossary

scene graph	The general term for a data structure that holds the objects within a particular scene. Objects in a scene graph often have parent-child relationships for their transforms, as well as references to the shaders, meshes, and textures needed to render them.
projective texturing	A texture mapping technique that generates texture coordinates to make a 2D texture appear to have been projected onto a surface. This is done by transforming the vertex positions of objects into the scene through a projective series of transformations into the space of the texture itself.
spotlight source	A light source that emits from a position in the world in a generally conical shape along a particular direction. Some spot lights have a full orientation, while others only need a direction. Spotlights can be implemented in shader code, or more generally via projective texturing techniques.
cube map texture	A type of texture that uses 6 2D images to represent faces of a cube. It takes 3D texture coordinates that represent a direction from the center of a cube onto one of these faces. Thus, each texel on each of the 6 faces comes from a unique direction. Cube maps allow data based on directions to vary based on stored texture data.
texture compression	A set of image formats that stores texel data in a small format that is optimized for texture access. These formats are not as small as specialized image file formats, but they are designed for fast GPU texture fetch access, while still saving significant graphics memory.

Part V. Framebuffer

Render targets and framebuffer blending are key components to many advanced effects. These tutorials will cover many per-framebuffer operations, from blending to render targets.

Part VI. Advanced Lighting

Simple diffuse lighting and directional shadows are useful, but better, more effective lighting models and patterns exist. These tutorials will explore those, from Phong lighting to reflections to HDR and blooming.

Appendix A. Further Study

G

Topics of Interest

This book should provide a firm foundation for understanding graphics development. However, there are many subjects that it does not cover which are also important in rendering. Here is a list of topics that you should investigate, with a quick introduction to the basic concepts.

This list is not intended to be a comprehensive tour of all interesting graphical effects. It is simply an introduction to a few concepts that you should spend some time investigating. There may be others not on this list that are worthy of your time.

Vertex Weighting. All of our meshes have had fairly simple linear transformations applied to them (outside of the perspective projection). However, the mesh for a human or human-like character needs to be able to deform based on animations. The transformation for the upper arm and the lower arm are different, but they both affect vertices at the elbow in some way.

The system for dealing with this is called vertex weighting or skinning (note: “skinning”, as a term, has also been applied to mapping a texture on an object. So be aware of that when doing searches). A character is made of a hierarchy of transformations; each transform is called a bone. Vertices are weighted to particular bones. Where it gets interesting is that vertices can have weights to multiple bones. This means that the vertex's final position is determined by a weighted combination of two (or more) transforms.

Vertex shaders generally do this by taking an array of matrices as a uniform block. Each matrix is a bone. Each vertex contains a vec4 which contains up to 4 indices in the bone matrix array, and another vec4 that contains the weight to use with the corresponding bone. The vertex is multiplied by each of the four matrices, and the results are averaged together.

This process is made more complicated by normals and the tangent-space basis necessary for bump mapping. And it is complicated even further by a technique called dual quaternion skinning. This is done primarily to avoid issues with certain bones rotating relative to one another. It prevents vertices from pinching inwards when the wrist bone is rotated 180 degrees from the forearm.

BRDFs. The term Bidirectional Reflectance Distribution Function (BRDF) refers to a special kind of function. It is a function of two directions: the direction towards the incident light and the direction towards the viewer, both of which are specified relative to the surface normal. This last part makes the BRDF independent of the surface normal, as it is an implicit parameter in the equation. The output of the BRDF is the percentage of light from the light source that is reflected along the view direction. Thus, the output of the BRDF is multiples into the incident light intensity to produce the output light intensity.

By all rights, this sounds like a lighting equation. And it is. Indeed, every lighting equation in this book can be expressed in the form of a BRDF. One of the things that make BRDFs as a class of equations interesting is that you can actually take a physical object into a lab, perform a series of tests on it, and produce a BRDF table out of them. This BRDF table, typically expressed as a texture, can then be directly used by a shader to show how a surface in the real world actually behaves under lighting conditions. This can provide much more accurate results than using models as we have done.

Scalable Alpha Testing. We have seen how alpha-test works via `discard`: a fragment is culled if its alpha is beneath a certain threshold. However, when magnifying a texture providing that alpha, it can create an unfortunate stair-step effect along the border between the culled and unculled part. It is possible to avoid these artifacts, if one preprocesses the texture correctly.

Valve software's Chris Green wrote a paper entitled *Improved Alpha-Tested Magnification for Vector Textures and Special Effects*. This paper describes a way to take a high-resolution version of the alpha and convert it into a distance field. Since distances interpolate much better in a spatial domain like images, using distance-based culling instead of edge-based culling produces a much smoother result even with a heavily magnified image.

The depth field can also be used to do other effects, like draw outlines around objects or drop shadows. And the best part is that it is a very inexpensive technique to implement. It requires some up-front preprocessing, but what you get in the end is quite powerful and very performance-friendly.

Screen-Space Ambient Occlusion. One of the many difficult processes when doing rasterization-based rendering is dealing with interreflection. That is, light reflected from one object that reflects off of another. We covered this by providing a single ambient light as something of a hack. A useful one, but a hack nonetheless.

Screen-space ambient occlusion (SSAO) is the term given to a hacky modification of this already hacky concept. The idea works like this. If two objects form an interior corner, then the amount of interreflected light for the pixels around that interior corner will be less than the general level of interreflection. This is a generally true statement. What SSAO does is find all of those corners, in screen-space, and decreases the ambient light intensity for them proportionately.

Doing this in screen space requires access to the screen space depth for each pixel. So it combines very nicely with deferred rendering techniques. Indeed, it can simply be folded into the ambient lighting pass of deferred rendering, though getting it to perform reasonably fast is the biggest challenge. But the results can look good enough to be worth the effort.

Light Scattering. When light passes through the atmosphere, it can be absorbed and reflected by the atmosphere itself. After all, this is why the sky is blue: because it absorbs some of the light coming from the sun, tinting the sunlight blue. Clouds are also a form of this: light that hits the water vapor that comprises clouds is reflected around and scattered. Thin clouds appear white because much of the light still makes it through. Thick clouds appear dark because they scatter and absorb so much light that not much passes through them.

All of these are light scattering effects. The most common in real-time scenarios is fog, which meteorologically speaking, is simply a low-lying cloud. Ground fog is commonly approximated in graphics by applying a change to the intensity of the light reflected from a surface towards the viewer. The farther the light travels, the more of it is absorbed and reflected, converting it into the fog's color. So objects that are extremely distant from the viewer would be indistinguishable from the fog itself. The thickness of the fog is based on the distance light has to travel before it becomes just more fog.

Fog can also be volumetric, localized in a specific region in space. This is often done to create the effect of a room full of steam, smoke, or other particulate aerosols. Volumetric fog is much more complex to implement than distance-based fog. This is complicated even more by objects that have to move through the fog region.

Fog system deal with the light reflected from a surface to the viewer. Generalized light scattering systems deal with light from a light source that is scattered through fog. Think about car headlights in a fog: you can see the beam reflecting off of the fog itself. That is an entirely different can of worms and a general implementation is very difficult to pull off. Specific implementations, sometimes called “God rays” for the effect of strong sunlight on dust particles in a dark room, can provide some form of this. But they generally have to be special cased for every occurrence, rather than a generalized technique that can be applied.

Non-Photorealistic Rendering. Talking about non-photorealistic rendering (NPR) as one thing is like talking about non-Elephant biology as one thing. Photorealism may have the majority of the research effort in it, but the depth of non-photorealistic possibilities with modern hardware is extensive.

These techniques often extend beyond mere rendering, from how textures are created and what they store, to exaggerated models, to various other things. Once you leave the comfort of approximately realistic lighting models, all bets are off.

In terms of just the rendering part, the most well-known NPR technique is probably cartoon rendering, also known as cel shading. The idea with realistic lighting is to light a curved object so that it appears curved. With cel shading, the idea is often to light a curved object so that it appears *flat*. Or at least, so that it approximates one of the many different styles of cel animation, some of which are more flat than others. This generally means that light has only a few intensities: on, perhaps a slightly less on, and off. This creates a sharp highlight edge in the model, which can give the appearance of curvature without a full gradient of intensity.

Coupled with cartoon rendering is some form of outline rendering. This is a bit more difficult to pull off in an aesthetically pleasing way. When an artist is drawing cel animation, they have the ability to fudge things in arbitrary ways to achieve the best result. Computers have to use an algorithm, which is more likely to be a compromise than a perfect solution for every case. What looks good for outlines in one case may not work in another. So testing the various outlining techniques is vital for pulling off a convincing effect.

Other NPR techniques include drawing objects that look like pencil sketches, which require more texture work than rendering system work. Some find ways to make what could have been a photorealistic rendering look like an oil painting of some form, or in some cases, the glossy colors of a comic book. And so on. NPR has as its limits the user's imagination. And the cleverness of the programmer to find a way to make it work, of course.

Appendix B. History of PC Graphics Hardware

A Programmer's View

For those of you had the good fortune of not being graphics programmers during the formative years of the development of consumer graphics hardware, what follows is a brief history. Hopefully, it will give you some perspective on what has changed in the last 15 years or so, as well as an idea of how grateful you should be that you never had to suffer through the early days.

Voodoo Magic

In the years 1995 and 1996, a number of graphics cards were released. Graphics processing via specialized hardware on PC platforms was nothing new. What was new about these cards was their ability to do 3D rasterization.

The most popular of these for that era was the Voodoo Graphics card from 3Dfx Interactive. It was fast, powerful for its day, and provided high quality rendering (again, for its day).

The functionality of this card was quite bare-bones from a modern perspective. Obviously there was no concept of shaders of any kind. Indeed, it did not even have vertex transformation; the Voodoo Graphics pipeline began with clip-space values. This required the CPU to do vertex transformations. This hardware was effectively just a triangle rasterizer.

That being said, it was quite good for its day. As inputs to its rasterization pipeline, it took vertex inputs of a 4-dimensional clip-space position (though the actual space was not necessarily the same as OpenGL's clip-space), a single RGBA color, and a single three-dimensional texture coordinate. The hardware did not support 3D textures; the extra component was in case the user wanted to do projective texturing.

The texture coordinate was used to map into a single texture. The texture coordinate and color interpolation was perspective-correct; in those days, that was a significant selling point. The venerable Playstation 1 could not do perspective-correct interpolation.

The value fetched from the texture could be combined with the interpolated color using one of three math functions: additions, multiplication, or linear interpolation based on the texture's alpha value. The alpha of the output was controlled with a separate math function, thus allowing the user to generate the alpha with different math than the RGB portion of the output color. This was the sum total of its fragment processing.

It had framebuffer blending support. Its framebuffer could even support a destination alpha value, though you had to give up having a depth buffer to get it. Probably not a good tradeoff. Outside of that issue, its blending support was superior even to OpenGL 1.1. It could use different source and destination factors for the alpha component than the RGB component; the old GL 1.1 forced the RGB and A to be blended with the same factors.

The blending was even performed with full 24-bit color precision and then downsampled to the 16-bit precision of the output upon writing.

From a modern perspective, spoiled with our full programmability, this all looks incredibly primitive. And, to some degree, it is. But compared to the pure CPU solutions to 3D rendering of the day, the Voodoo Graphics card was a monster.

It's interesting to note that the simplicity of the fragment processing stage owes as much to the lack of inputs as anything else. When the only values you have to work with are the color from a texture lookup and the per-vertex interpolated color, there really is not all that much you could do with them. Indeed, as we will see in the next phases of hardware, increases in the complexity of the fragment processor was a reaction to increasing the number of inputs *to* the fragment processor. When you have more data to work with, you need more complex operations to make that data useful.

Dynamite Combiners

The next phase of hardware came, not from 3Dfx, but from a new company, NVIDIA. While 3Dfx's Voodoo II was much more popular than NVIDIA's product, the NVIDIA Riva TNT (released in 1998) was more interesting in terms of what it brought to the table for programmers. Voodoo II was purely a performance improvement; TNT was the next step in the evolution of graphics hardware.

Like other graphics cards of the day, the TNT hardware had no vertex processing. Vertex data was in clip-space, as normal, so the CPU had to do all of the transformation and lighting. Where the TNT shone was in its fragment processing. The power of the TNT is in it's name; TNT stands for TwiN Texel. It could access from two textures at once. And while the Voodoo II could do that as well, the TNT had much more flexibility to its fragment processing pipeline.

In order to accomidate two textures, the vertex input was expanded. Two textures meant two texture coordinates, since each texture coordinate was directly bound to a particular texture. While they were allowing two of things, NVIDIA also allowed for two per-vertex colors. The idea here has to do with lighting equations.

For regular diffuse lighting, the CPU-computed color would simply be dot(N, L), possibly with attenuation applied. Indeed, it could be any complicated diffuse lighting function, since it was all on the CPU. This diffuse light intensity would be multiplied by the texture, which represented the diffuse absorption of the surface at that point.

This becomes less useful if you want to add a specular term. The specular absorption and diffuse absorption are not necessarily the same, after all. And while you may not need to have a specular texture, you do not want to add the specular component to the diffuse component *before* you multiply by their respective colors. You want to do the addition afterwards.

This is simply not possible if you have only one per-vertex color. But it becomes possible if you have two. One color is the diffuse lighting value. The other color is the specular component. We multiply the first color by the diffuse color from the texture, then add the second color as the specular reflectance.

Which brings us nicely to fragment processing. The TNT's fragment processor had 5 inputs: 2 colors sampled from textures, 2 colors interpolated from vertices, and a single “constant” color. The latter, in modern parlance, is the equivalent of a shader uniform value.

That's a lot of potential inputs. The solution NVIDIA came up with to produce a final color was a bit of fixed functionality that we will call the texture environment. It is directly analogous to the OpenGL 1.1 fixed-function pipeline, but with extensions for multiple textures and some TNT-specific features.

The idea is that each texture has an environment. The environment is a specific math function, such as addition, subtraction, multiplication, and linear interpolation. The operands to this function could be taken from any of the fragment inputs, as well as a constant zero color value.

It can also use the result from the previous environment as one of its arguments. Textures and environments are numbered, from zero to one (two textures, two environments). The first one executes, followed by the second.

If you look at it from a hardware perspective, what you have is a two-opcode assembly language. The available registers for the language are two vertex colors, a single uniform color, two texture colors, and a zero register. There is also a single temporary register to hold the output from the first opcode.

Graphics programmers, by this point, had gotten used to multipass-based algorithms. After all, until TNT, that was the only way to apply multiple textures to a single surface. And even with TNT, it had a pretty confining limit of two textures and two opcodes.

This was powerful, but quite limited. Two opcodes really was not enough.

The TNT cards also provided something else: 32-bit framebuffers and depth buffers. While the Voodoo cards used high-precision math internally, they still wrote to 16-bit framebuffers, using a technique called dithering to make them look like higher precision. But dithering was nothing compared to actual high precision framebuffers. And it did nothing for the depth buffer artifacts that a 16-bit depth buffer gave you.

While the original TNT could do 32-bit, it lacked the memory and overall performance to really show it off. That had to wait for the TNT2. Combined with product delays and some poor strategic moves by 3Dfx, NVIDIA became one of the dominant players in the consumer PC graphics card market. And that was cemented by their next card, which had real power behind it.

Tile-Based Rendering

While all of this was going on, a small company called PowerVR released its Series 2 graphics chip. PowerVR's approach to rendering was fundamentally different from the standard rendering pipeline.

They used what they called a “deferred, tile-based renderer.” The idea is that they store all of the clip-space triangles in a buffer. Then, they sort this buffer based on which triangles cover which areas of the screen. The output screen is divided into a number of tiles of a fixed size. Say, 8x8 in size.

For each tile, the hardware finds the triangles that are within that tile's area. Then it does all the usual scan conversion tricks and so forth. It even automatically does per-pixel depth sorting for blending, which remains something of a selling point (no more having to manually sort blended objects). After rendering that tile, it moves on to the next. These operations can of course be executed in parallel; you can have multiple tiles being rasterized at the same time.

The idea behind this is to avoid having large image buffers. You only need a few 8x8 depth buffers, so you can use very fast, on-chip memory for it. Rather than having to deal with caches, DRAM, and large bandwidth memory channels, you just have a small block of memory where you do all of your logic. You still need memory for textures and the output image, but your bandwidth needs can be devoted solely to textures.

For a time, these cards were competitive with the other graphics chip makers. However, the tile-based approach simply did not scale well with resolution or geometry complexity. Also, they missed the geometry processing bandwagon, which really hurt their standing. They fell farther and farther behind the other major players, until they stopped making desktop parts altogether.

However, they may ultimately have the last laugh; unlike 3Dfx and so many others, PowerVR still exists. They provided the GPU for the Sega Dreamcast console. And while that console was a market failure, it did show where PowerVR's true strength lay: embedded platforms.

Embedded platforms tend to play to their tile-based renderer's strengths. Memory, particularly high-bandwidth memory, eats up power; having less memory means longer-lasting mobile devices. Embedded devices tend to use smaller resolutions, which their platform excels at. And with low resolutions, you are not trying to push nearly as much geometry.

Thanks to these facts, PowerVR graphics chips power the vast majority of mobile platforms that have any 3D rendering in them. Just about every iPhone, Droid, iPad, or similar device is running PowerVR technology. And that's a growth market these days.

Vertices and Registers

The next stage in the evolution of graphics hardware again came from NVIDIA. While 3Dfx released competing cards, they were again behind the curve. The NVIDIA GeForce 256 (not to be confused with the GeForce GT250, a much more modern card), released in 1999, provided something truly new: a vertex processing pipeline.

The OpenGL API has always defined a vertex processing pipeline (it was fixed-function in those days rather than shader-based). And NVIDIA implemented it in their TNT-era drivers on the CPU. But only with the GeForce 256 was this actually implemented in hardware. And NVIDIA essentially built the entire OpenGL fixed-function vertex processing pipeline directly into the GeForce hardware.

This was primarily a performance win. While it was important for the progress of hardware, a less-well-known improvement of the early GeForce hardware was more important to its future.

In the fragment processing pipeline, the texture environment stages were removed. In their place was a more powerful mechanism, what NVIDIA called “register combiners.”

The GeForce 256 provided 2 regular combiner stages. Each of these stages represented up to four independent opcodes that operated over the register set. The opcodes could result in multiple outputs, which could be written to two temporary registers.

What is interesting is that the register values are no longer limited to color values. Instead, they are signed values, on the range [-1, 1]; they have 9 bits of precision or so. While the initial color or texture values are on [0, 1], the actual opcodes themselves can perform operations that generate negative values. Opcodes can even scale/bias their inputs, which allow them to turn unsigned colors into signed values.

Because of this, the GeForce 256 was the first hardware to be able to do functional bump mapping, without hacks or tricks. A single register combiner stage could do 2 3-vector dot-products at a time. Textures could store normals by compressing them to a [0, 1] range. The light direction could either be a constant or interpolated per-vertex in texture space.

Now granted, this still was a primitive form of bump mapping. There was no way to correct for texture-space values with binormals and tangents. But this was at least something. And it really was the first step towards programmability; it showed that textures could truly represent values other than colors.

There was also a single final combiner stage. This was a much more limited stage than the regular combiner stages. It could do a linear interpolation operation and an addition; this was designed specifically to implement OpenGL's fixed-function fog and specular computations.

The register file consisted of two temporary registers, two per-vertex colors, two texture colors, two uniform values, the zero register, and a few other values used for OpenGL fixed-function fog operations. The color and texture registers were even writeable, if you needed more temporaries.

There were a few other sundry additions to the hardware. Cube textures first came onto the scene. Combined with the right texture coordinate computations (now in hardware), you could have reflective surfaces much more easily. Anisotropic filtering and multisampling also appeared at this time. The limits were relatively small; anisotropic filtering was limited to 4x, while the maximum number of samples was restricted to two. Compressed texture formats also appeared on the scene.

What we see thus far as we take steps towards true programmability is that increased complexity in fragment processing starts pushing for other needs. The addition of a dot product allows lighting computations to take place per-fragment. But you cannot have full texture-space bump mapping because of the lack of a normal/binormal/tangent matrix to transform vectors to texture space. Cubemaps allow you to do arbitrary reflections, but computing reflection directions per-vertex requires interpolating reflection normals, which does not work very well over large polygons.

This also saw the introduction of something called a rectangle texture. This texture type is something of an odd duck that still remains in current day. It was a way of creating a texture of arbitrary size; until then, textures were limited to powers of two in size (though the sizes did not have to be the same). The texture coordinates for rectangle textures are not normalized; they were in texture space values.

The GPU Divide

When NVIDIA released the GeForce 256, they coined the term “Geometry Processing Unit” or GPU. Until this point, graphics chips were called exactly that: graphics chips. The term GPU was intended by NVIDIA to differentiate the GeForce from all of its competition, including the final cards from 3Dfx.

Because the term was so reminiscent to CPUs, the term took over. Every graphics chip is a GPU now, even ones released before the term came to exist.

In truth, the term GPU never really made much sense until the next stage, where the first cards with actual programmability came onto the scene.

Programming at Last

How do you define a demarcation between non-programmable graphics chips and programmable ones? We have seen that, even in the humble TNT days, there were a couple of user-defined opcodes with several possible input values.

One way is to consider what programming is. Programming is not simply a mathematical operation; programming needs conditional logic. Therefore, it is not unreasonable to say that something is not truly programmable until there is the possibility of some form of conditional logic.

And it is at this point where that first truly appears. It appears first in the *vertex* pipeline rather than the fragment pipeline. This seems odd until one realizes how crucial fragment operations are to overall performance. It therefore makes sense to introduce heavy programmability in the less performance-critical areas of hardware first.

The GeForce 3, released in 2001 (a mere 3 years after the TNT), was the first hardware to provide this level of programmability. While GeForce 3 hardware did indeed have the fixed-function vertex pipeline, it also had very flexible programmable pipeline. The retaining of the fixed-function

code was a performance need; the vertex shader was not as fast as the fixed-function one. It should be noted that the original X-Box's GPU, designed in tandem with the GeForce 3, eschewed the fixed-functionality altogether in favor of having multiple vertex shaders that could compute several vertices at a time. This was eventually adopted for later GeForces.

Vertex shaders were pretty powerful, even in their first incarnation. While there was no conditional branching, there was conditional logic, the equivalent of the ?: operator. These vertex shaders exposed up to 128 vec4 uniforms, up to 16 vec4 inputs (still the modern limit), and could output 6 vec4 outputs. Two of the outputs, intended for colors, were lower precisions than the others. There was a hard limit of 128 opcodes. These vertex shaders brought full swizzling support and a plethora of math operations.

The GeForce 3 also added up to two more textures, for a total of four textures per triangle. They were hooked directly into certain per-vertex outputs, because the per-fragment pipeline did not have real programmability yet.

At this point, the holy grail of programmability at the fragment level was dependent texture access. That is, being able to access a texture, do some arbitrary computations on it, and then access another texture with the result. The GeForce 3 had some facilities for that, but they were not very good ones.

The GeForce 3 used 8 register combiner stages instead of the 2 that the earlier cards used. Their register files were extended to support two extra texture colors and a few more tricks. But the main change was something that, in OpenGL terminology, would be called “texture shaders.”

What texture shaders did was allow the user to, instead of accessing a texture, perform a computation on that texture's texture unit. This was much like the old texture environment functionality, except only for texture coordinates. The textures were arranged in a sequence. And instead of accessing a texture, you could perform a computation between that texture unit's coordinate and possibly the coordinate from the previous texture shader operation, if there was one.

It was not very flexible functionality. It did allow for full texture-space bump mapping, though. While the 8 register combiners were enough to do a full matrix multiply, they were not powerful enough to normalize the resulting vector. However, you could normalize a vector by accessing a special cubemap. The values of this cubemap represented a normalized vector in the direction of the cubemap's given texture coordinate.

But using that required spending a total of 3 texture shader stages. Which meant you get a bump map and a normalization cubemap only; there was no room for a diffuse map in that pass. It also did not perform very well; the texture shader functions were quite expensive.

True programmability came to the fragment shader from ATI, with the Radeon 8500, released in late 2001.

The 8500's fragment shader architecture was pretty straightforward, and in terms of programming, it is not too dissimilar to modern shader systems. Texture coordinates would come in. They could either be used to fetch from a texture or be given directly as inputs to the processing stage. Up to 6 textures could be used at once. Then, up to 8 opcodes, including a conditional operation, could be used. After that, the hardware would repeat the process using registers written by the opcodes. Those registers could feed texture accesses from the same group of textures used in the first pass. And then another 8 opcodes would generate the output color.

It also had strong, but not full, swizzling support in the fragment shader. Register combiners had very little support for swizzling.

This era of hardware was also the first to allow 3D textures. Though that was as much a memory concern as anything else, since 3D textures take up lots of memory which was not available on earlier cards. Depth comparison texturing was also made available.

While the 8500 was a technological marvel, it was a flop in the market compared to the GeForce 3 & 4. Indeed, this is a recurring theme of these eras: the card with the more programmable hardware often tends to lose in its first iteration.

API Hell

This era is notable in what it did to graphics APIs. Consider the hardware differences between the 8500 and the GeForce 3/4 in terms of fragment processing.

On the Direct3D front, things were not the best. Direct3D 8 promised a unified shader development pipeline. That is, you could write a shader according to their specifications and it would work on any D3D 8 hardware. And this was effectively true. For vertex shaders, at least.

However, the D3D 8.0 pixel shader pipeline was nothing more than NVIDIA's register combiners and texture shaders. There was no real abstraction of capabilities; the D3D 8.0 pixel shaders simply took NVIDIA's hardware and made a shader language out of it.

To provide support for the 8500's expanded fragment processing feature-set, there was D3D 8.1. This version altered the pixel shader pipeline to match the capabilities of the Radeon 8500. Fortunately, the 8500 would accept 8.0 shaders just fine, since it was capable of doing everything the GeForce 3 could do. But no one would mistake either shader specification for any kind of real abstraction.

Things were much worse on the OpenGL front. At least in D3D, you used the same basic C++ API to provide shaders; the shaders themselves may have been different, but the base API was the same. Not so in OpenGL land.

NVIDIA and ATI released entirely separate proprietary extensions for specifying fragment shaders. NVIDIA's extensions built on the register combiner extension they released with the GeForce 256. They were completely incompatible. And worse, they were not even string-based.

Imagine having to call a C++ function to write every opcode of a shader. Now imagine having to call *three* functions to write each opcode. That's what using those APIs was like.

Things were better on vertex shaders. NVIDIA initially released a vertex shader extension, as did ATI. NVIDIA's was string-based, but ATI's version was like their fragment shader. Fortunately, this state of affairs did not last long; the OpenGL ARB came along with their own vertex shader extension. This was not GLSL, but an assembly like language based on NVIDIA's extension.

It would take much longer for the fragment shader disparity to be worked out.

Dependency

The Radeon 9700 was the 8500's successor. It improved on the 8500 somewhat. The vertex shader gained real conditional branching logic. Some of the limits were also relaxed; the number of available outputs and uniforms increased. The fragment shader's architecture remained effectively the same; the 9700 simply increased the limits. There were 8 textures available and 16 opcodes, and it could perform 4 passes over this set.

The GeForce FX, released in 2003, was a substantial improvement, both over the GeForce 3/4 and over the 9700 in terms of fragment processing. NVIDIA took a different approach to their fragment shaders; their fragment processor worked not entirely unlike modern shader processors do.

It read an instruction, which could be a math operation, conditional branch (they had actual branches in fragment shading), or texture lookup instruction. It then executed that instruction. The texture lookup could be from a set of 8 textures. And then it repeated this process on the next instruction. It was doing math computations in a way not entirely unlike a traditional CPU.

There was no real concept of a dependent texture access for the GeForce FX. The inputs to the fragment pipeline were simply the texture coordinates and colors from the vertex stage. If you used a texture coordinate to access a texture, it was fine with that. If you did some computations with them and then accessed a texture, it was just as fine with that. It was completely generic.

It also failed in the marketplace. This was due primarily to its lateness and its poor performance in high-precision computation operations. The FX was optimized for doing 16-bit math computations in its fragment shader; while it *could* do 32-bit math, it was half as fast when doing this. But Direct3D 9's shaders did not allow the user to specify the precision of computations; the specification required at least 24-bits of precision. To match this, NVIDIA had no choice but to force 32-bit math on all D3D 9 applications, making them run much slower than their ATI counterparts (the 9700 always used 24-bit precision math).

Things were no better in OpenGL land. The two competing unified fragment processing APIs, GLSL and an assembly-like fragment shader, did not have precision specifications either. Only NVIDIA's proprietary extension for fragment shaders provided that, and developers were less likely to use it. Especially with the head start that the 9700 gained in the market by the FX being released late.

It performs so poorly in the market that NVIDIA dropped the FX name for the next hardware revision. The GeForce 6 improved its 32-bit performance to the point where it was competitive with the ATI equivalents.

This level of hardware saw the gaining of a number of different features. sRGB textures and framebuffers appeared, as did floating-point textures. Blending support for floating-point framebuffers was somewhat spotty; some hardware could do it only for 16-bit floating-point, some could not do it at all. The restrictions of power-of-two texture sizes was also lifted, to varying degrees. None of ATI's hardware of this era fully supported this when used with mipmapping, but NVIDIA's hardware from the GeForce 6 and above did.

The ability to access textures from vertex shaders was also introduced in this series of hardware. Vertex texture accesses uses a separate list of textures from those bound for fragment shaders. Only four textures could be accessed from a vertex shader, while 8 textures was normal for fragment shaders.

Render to texture also became generally available at this time, though this was more of an API issue (neither OpenGL nor Direct3D allowed textures to be used as render targets before this point) than hardware functionality. That is not to say that hardware had no role to play. Textures are often not stored as linear arrays of memory the way they are loaded with `glTexImage`. They are usually stored in a swizzled format, where 2D or 3D blocks of texture data are stored sequentially. Thus, rendering to a texture required either the ability to render directly to swizzled formats or the ability to read textures that are stored in unswizzled formats.

More than just render to texture was introduced. What was also introduced was the ability to render to multiple textures or buffers at one time. The number of renderable buffers was generally limited to 4 across all hardware platforms.

Rise of the Compilers

Microsoft put their foot down after the fiasco with D3D 8's fragment shaders. They wanted a single standard that all hardware makers would support. While this lead to the FX's performance failings, it also meant that compilers were becoming very important to shader performance.

In order to have a real abstraction, you need compilers that are able to take the abstract language and map it to very different kinds of hardware. With Direct3D and OpenGL providing standards for shading languages, compiler quality started to become vital for performance.

OpenGL moved whole-heartedly, and perhaps incautiously, into the realm of compilers when the OpenGL ARB embraced GLSL, a C-style language. They developed this language to the exclusion of all others.

In Direct3D land, Microsoft developed the High-Level Shading Language, HLSL. But the base shading languages used by Direct3D 9 were still the assembly-like shading languages. HLSL was compiled by a Microsoft-developed compiler into the assembly languages, which were fed to Direct3D.

With compilers and semi-real languages with actual logic constructs, a new field started to arise: General Programming GPU or GPGPU. The idea was to use a GPU to do non-rendering tasks. It started around this era, but the applications were limited due to the nature of hardware. Only fairly recently, with the advent of special languages and APIs (OpenCL, for example) that are designed for GPGPU tasks, has GPGPU started to really move into its own. Indeed, in the most recent hardware era, hardware makers have added features to GPUs that have somewhat... dubious uses in the field of graphics, but substantial uses in GPGPU tasks.

Modern Unification

Welcome to the modern era. All of the examples in this book are designed on and for this era of hardware, though some of them could run on older ones with some alteration. The release of the Radeon HD 2000 and GeForce 8000 series cards in 2006 represented unification in more ways than one.

With the prior generations, fragment hardware had certain platform-specific peculiarities. While the API kinks were mostly ironed out with the development of proper shading languages, there were still differences in the behavior of hardware. While 4 dependent texture accesses were sufficient for most applications, naive use of shading languages could get you in trouble on ATI hardware.

With this generation, neither side really offered any real functionality difference. There are still differences between the hardware lines, and certainly in terms of performance. But the functionality differences have never been more blurred than they were with this revision.

Another form of unification was that both NVIDIA and ATI moved to a unified shader architecture. In all prior generations, fragment shaders and vertex shaders were fundamentally different hardware. Even when they started doing the same kinds of things, such as accessing textures, they were both using different physical hardware to do so. This led to some inefficiencies.

Deferred rendering probably gives the most explicit illustration of the problem. The first pass, the creation of the g-buffers, is a very vertex-shader-intensive activity. While the fragment shader can be somewhat complex, doing several texture fetches to compute various material parameters, the vertex shader is where much of the real work is done. Lots of vertices come through the shader, and if there are any complex transformations, they will happen here.

The second pass is a *very* fragment shader intensive pass. Each light layer is comprised of exactly 4 vertices. Vertices that can be provided directly in clip-space. From then on, the fragment shader is what is being worked. It performs all of the complex lighting calculations necessary for the various rendering techniques. Four vertices generate literally millions of fragments, depending on the rendering resolution.

In prior hardware generations, in the first pass, there would be fragment shaders going to waste, as they would process fragments faster than the vertex shaders could deliver triangles. In the second pass, the reverse happens, only even moreso. Four vertex shader executions, and then all of those vertex shaders would be completely useless. All of those parallel computational units would go to waste.

Both NVIDIA and ATI devised hardware such that the computational elements were separated from their particular kind of computations. All shader hardware could be used for vertices, fragments, or geometry shaders (new in this generation). This would be changed on demand, based on the resource load. This makes deferred rendering in particular much more efficient; the second pass is able to use almost all of the available shader resources for lighting operations.

This unified shader approach also means that every shader stage has essentially the same capabilities. The standard for the maximum texture count is 16, which is plenty enough for doing just about anything. This is applied equally to all shader types, so vertex shaders have the same number of textures available as fragment shaders.

This smoothed out a great many things. Shaders gained quite a few new features. Uniform buffers became available. Shaders could perform computations directly on integer values. Unlike every generation before, all of these features were parceled out to all types of shaders equally.

Along with unified shaders came a long list of various and sundry improvements to non-shader hardware. These include, but are not limited to:

- Floating-point blending was worked out fully. Hardware of this era supports full 32-bit floating point blending, though for performance reasons you're still advised to use the lowest precision you can get away with.
- Arbitrary texture swizzling as a direct part of texture sampling parameters, rather than in the shader itself.
- Integer texture formats, to compliment the shader's ability to use integer values.
- Array textures.

Various other limitations were expanded as well.

Post-Modern

This was not the end of hardware evolution; there has been hardware released in recent years The Radeon HD 5000 and GeForce GT 400 series and above have increased rendering features. They're just not as big of a difference compared to what came before.

One of the biggest new feature in this hardware is tessellation, the ability to take triangles output from a vertex shader and split them into new triangles based on arbitrary (mostly) shader logic. This sounds like what geometry shaders can do, but it is different.

Tessellation is actually something that ATI toyed around with for years. The Radeon 9700 had tessellation support with something they called PN triangles. This was very automated and not particularly useful. The entire Radeon HD 2000-4000 cards included tessellation features as well. These were pre-vertex shader, while the current version comes post-vertex shader.

In the older form, the vertex shader would serve double duty. An incoming triangle would be broken down into many triangles. The vertex shader would then have to compute the per-vertex attributes for each of the new triangles, based on the old attributes and which vertex in the new series of vertices is being computed. Then it would do its normal transformation and other operations on those attributes.

The current form introduces two new shader stages. The first, immediately after the vertex shader, controls how much tessellation happens on a particular primitive. The tessellation happens, splitting the single primitive into multiple primitives. The next stage determines how to compute the new positions, normals, etc of the primitive, based on the values of the primitive being tessellated. The geometry shader still exists; it is executed after the final tessellation shader stage.

Another feature is the ability to have a shader arbitrarily read *and* write to images in textures. This is not merely sampling from a texture; it uses a different interface (no filtering), and it means very different things. This form of image data access breaks many of the rules around OpenGL, and it is very easy to use the feature wrongly.

These are not covered in this book for a few reasons. First, there is not as much hardware out there that supports it (though this is increasing daily). Sticking to OpenGL 3.3 meant casting a wider net; requiring OpenGL 4.2 would have meant fewer people could run those tutorials.

Second, these features are quite complicated to use. Any discussion of tessellation would require discussing tessellation algorithms, which are all quite complicated. Any discussion of image reading/writing would require talking about shader hardware at a level of depth that is well beyond the beginner level. These are useful features, to be sure, but they are also very complex features.

Appendix C. Getting Started with OpenGL

Now that you understand at least the beginnings of graphics programming, it would be useful to discuss how to get started using OpenGL in your own projects. This discussion will assume that you know how to set up a build project in your build system of choice.

The easiest way is to just use the Unofficial OpenGL SDK [http://glsdk.sourceforge.net/docs/html/index.html]. It is a well-documented collection of tools for writing simple OpenGL applications. It has functions for compiling shaders, the mouse-based controls we have used, image loading tools, and various other utilities that are vital to making OpenGL applications. Details for how to use it are found on the SDK's website.

Manual Usage

If you choose not to use the SDK, then you will have to build a set of useful tools yourself.

In order to use OpenGL, you will need to do two things. You must create a window and attach an OpenGL context to it, and you must load the OpenGL functions for that context. There are a number of other tools you may want (vector math, model loading, image loading, etc), but these tools are ones you need if you aren't going to do them manually.

Window and OpenGL Creation

Windows, displayable surfaces in GUI operating environments, are very platform-specific. Most Linux distributions rely on X11 for the lowest-level of window creation, and X11 has hooks, called GLX functions, for attaching OpenGL to those windows. The Win32 API is used on Windows to create windows. OpenGL can be attached to these windows using the WGL API.

Because window creation and OpenGL attachment are platform-specific, there are a number of cross-platform tools that make it possible to write platform-neutral OpenGL code. FreeGLUT is the tool that these tutorials use, and it is included as part of the SDK in the source distribution. There are a number of other tools available. We will discuss many of the options here.

FreeGLUT. FreeGLUT is based on the original GLUT, the OpenGL Utility Toolkit. You should never use GLUT; it is old and has not been updated in a decade. But FreeGLUT is 100% backwards compatible with it; any application that used GLUT can use FreeGLUT with no source code changes.

FreeGLUT creates and manages the window for the application. It provides callbacks so that the user can respond to various events. In these tutorials, the framework has initialized FreeGLUT and registered several standard callbacks, which each tutorial implements. There is a callback for when the display needs updating, when the user has resized the window, when the user has pressed a key, and when mouse input happens.

FreeGLUT can create windows or full-screen displays. It also has some limited support for menus. This does not allow you to create a menu bar, but it does allow you to create a right-click context menu.

FreeGLUT is a good tool for rapidly prototyping an effect. However, building a real application in it is problematic, particularly if you have specific timing needs. FreeGLUT owns the message processing loop; this limits your options for dealing with strict timing and so forth. FreeGLUT is good for demo programs and prototyping, but serious applications should avoid it.

FreeGLUT does have some text rendering functions, but these do not work when using shader-based rendering. If you are using a core OpenGL context, these functions will fail. They will also fail in compatibility contexts if you have a program object bound to the context.

FreeGLUT uses the X-Consortium license.

GLFW. GLFW is an alternative to FreeGLUT. Like FreeGLUT, GLFW is fairly bare-bones. It provides a way to create windows or full-screen displays. It provides ways to get keyboard and mouse input.

The biggest difference between them is that, while FreeGLUT owns the message processing loop, GLFW does not. GLFW requires that the user poll it to process messages. This allows the user to maintain reasonably strict timings for rendering. While this makes GLFW programs a bit more complicated than FreeGLUT ones (which is why these tutorials use FreeGLUT), it does mean that GLFW would be more useful in serious applications.

GLFW also provides more robust input support as well as

GLFW uses the zLib license.

Multimedia Libraries. A multimedia library is a library that handles, in a cross-platform way, graphics, sound, input, and other things. The impetus for all of these was DirectX, a Microsoft library that handles graphics, sound, input and a few other things for the purpose of improving the life of game developers. The purpose of DirectX was to be hardware-independent; code could be written against all graphics or sound hardware, and DirectX would sort out the details.

Cross-platform multimedia libraries do this as well, but they take things cross-platform. They generally support the big 3 operating systems (Windows, Linux, MacOSX), possibly also supporting BSD or various other shades of UNIX. Unlike DirectX, the multimedia libraries did not create their own 3D rendering system; they instead simply provide a way to use OpenGL.

The two biggest multimedia libraries are SDL (Simple Directmedia Layer) [http://www.libsdl.org/] and SFML (Simple and Fast Multimedia Library) [http://www.sfml-dev.org/]. SDL is the older, but it is still receiving updates. It is a C library, so if you are allergic to C-isms, you may wish to avoid it. Work is being done on SDL 1.3, which will apparently have support for mobile platforms. SDL uses the zLib license.

SFML is a newer library, which has a C++ API. While SDL is one big library (and requires dependencies like DirectX on Windows), SFML is more of a choose-your-own package. The base package contains just input and the ability to create a window, while there are other packages that build upon this. Also SFML makes it possible to integrate SFML windows with other GUI toolkits (see below); this makes it easier to build non-gaming applications that use actual GUI tools. SFML uses the zLib license.

Allegro [http://alleg.sourceforge.net/] is a game-centric multimedia library. Version 5 has native support for OpenGL in its graphics subsystem. It uses a C interface. It also has a number of advanced 2D rendering functions. Allegro uses the “giftware license,” which is rather like the MIT license.

GUI Toolkits. There are a number of cross-platform GUI libraries that deal with detailed window management, provide controls, and generally act like a full-fledged windowing system. Most of them have support for using OpenGL in one or more of their windows.

Which window creation tools you use are entirely up to you; the possible needs that you might have are well beyond the scope of this book.

Function Loading

Once an OpenGL context has been created, one must then load OpenGL's functions. In a normal library, this would not actually be a step; those functions would have a header that you include and a library of some sort that you link to, either statically or dynamically. Due to various complexities around OpenGL, it cannot work that way on most platforms.

Therefore, the user must query the context itself for the functions to load. And OpenGL has many functions. So the common way to handle this is to use a library to do it for you.

GL Load. This comes with the Unofficial OpenGL SDK. And while most of the SDK is not intended for high-performance use, GL Load is still perfectly serviceable in that capacity. After all, this is generally a one-time initialization step, so performance is ultimately irrelevant.

GL Load works with core and compatibility contexts equally well. GL Load uses the MIT License.

GLEW. GLEW (the OpenGL Extension Wrangler) is perhaps the most widely used alternative. While it is technically intended for loading extension functions, it works just as well on the OpenGL core functions.

The principle downside of GLEW is that it does not work well with non-compatibility contexts. There is some experimental support to make this work, but it is experimental. GLEW uses the MIT license.

GL3W. GL3W is unique in that it is not technically a library. It is a Python script that downloads and parses a header file, which it uses to create a library. This allows it to be as up-to-date as the source file that it downloads. The downside is that the format of this source file may change, and if it does, it would break this tool.

GL3W is intended specifically for core contexts, and it does not work with compatibility or older OpenGL versions. Being a Python script, it requires a Python 2.6 environment; it is unknown if it works with later Python versions. GL3W is public domain software.

Initialization

Once you have selected your tools of choice, the next step is to make them work. After downloading and compiling them, you will need to register them with your C/C++ build tool of choice. On Linux-platforms, you typically have some global registry for these things, but on Windows, things go where you put them.

How to use them depends on which tools you use. But the general idea for OpenGL context creation tools is that there will be some series of commands to supply parameters to the underlying system for context creation. And then a function is called to use those parameters to create the context. It is only *after* this process has successfully completed that one should call whatever initialization function the OpenGL function loading system uses. Your tools' documentation should explain this.

After doing both of these, you should be able to make OpenGL calls. The specifics on how to swap framebuffers (which is not part of the OpenGL API) is again left up to the window management tool.