

Table of Contents

1. [Introduction](#)
2. [Introduction](#)
 - i. [OpenGL](#)
 - ii. [OpenTK](#)
3. [Getting a window](#)
 - i. [Setting up the window](#)
 - ii. [Using callbacks](#)
 - iii. [Going fullscreen](#)
 - iv. [Your code](#)
 - v. [Input](#)
4. [OpenGL States and Primitives](#)
 - i. [State functions](#)
 - ii. [Errors and Hints](#)
 - iii. [Handling Primitives](#)
 - iv. [Vertex](#)
 - v. [Points](#)
 - vi. [Lines](#)
 - vii. [Polygons](#)
 - viii. [Triangles](#)
 - ix. [Quadrilaterals and Polygons](#)
 - x. [Attributes](#)
5. [Transformations And Matrices](#)
 - i. [Understanding Coordinate Transformations](#)
 - ii. [Transformation Deep Dive](#)
 - iii. [OpenGL and Matrices](#)
 - iv. [ModelView](#)
 - i. [Translation](#)
 - ii. [Rotation](#)
 - iii. [Scale](#)
 - iv. [View](#)
 - v. [Function Order](#)
 - i. [Solution](#)
 - vi. [Matrix Stacks](#)
 - vii. [Projections](#)
 - viii. [Viewport](#)
 - ix. [Mr.Roboto](#)
 - i. [Parent Child](#)
 - x. [CustomMatrices](#)
 - i. [Projection & LookAt](#)
 - ii. [Make your own stack](#)
 - xi. [Solar System](#)
 - xii. [What goes where](#)
6. [Colors and Lights](#)
 - i. [Using Colors](#)
 - ii. [Shading](#)
 - i. [On Your Own](#)
 - iii. [Lighting](#)

- iv. [Light Sources](#)
 - i. [Position and direction](#)
 - ii. [Light Color](#)
 - iii. [Attenuation](#)
 - iv. [Spotlights](#)
 - v. [Moving and Rotating](#)
 - v. [Programming Exercises](#)
 - i. [Directional Light](#)
 - ii. [Point Light](#)
 - iii. [Spot Light](#)
 - vi. [Materials](#)
 - vii. [Defining Materials](#)
 - i. [Test Scene](#)
 - ii. [Ambient and Diffuse](#)
 - iii. [Specular](#)
 - iv. [Color Tracking](#)
 - viii. [Normals](#)
 - ix. [The Lighting Model](#)
 - x. [Lighting Review](#)
7. [Blending and Fog](#)
- i. [Blending](#)
 - i. [Challenge](#)
 - ii. [Solution](#)
 - iii. [Issues](#)
 - ii. [Fog](#)
8. [Texture Mapping](#)
- i. [Overview](#)
 - ii. [Enable Texturing](#)
 - iii. [Texture Objects](#)
 - iv. [Texture Binding](#)
 - v. [Specifying Textures](#)
 - vi. [Min & Mag Filters](#)
 - vii. [Deleting Textures](#)
 - viii. [Texture Coordinates](#)
 - ix. [Putting it all together](#)
 - i. [Loading Help](#)
 - x. [Mip Mapping](#)
 - xi. [Texture Parameters](#)
 - xii. [Textures for UI](#)
 - i. [On your own](#)
 - xiii. [Sorting Objects](#)
9. [Particles](#)
- i. [Design](#)
 - i. [Particles](#)
 - ii. [Particle Systems](#)
 - ii. [Implementation](#)
 - i. [Particle](#)
 - ii. [Particle System](#)
 - iii. [Snow Example](#)
10. [Optimization](#)
- i. [Vertex Arrays](#)

- i. [Array Based Data](#)
 - ii. [Enabling Vertex Arrays](#)
 - iii. [Working With Arrays](#)
 - iv. [Pinning and Pointers](#)
 - v. [DrawArrays](#)
 - vi. [DrawElements](#)
 - vii. [Example](#)
 - ii. [Vertex Buffers](#)
 - i. [Generate and Bind](#)
 - ii. [Fill data](#)
 - iii. [Render](#)
 - iv. [Cleanup](#)
 - v. [Review](#)
 - vi. [On Your Own](#)
 - iii. [Loading Meshes](#)
 - i. [Implementation](#)
 - ii. [Test it](#)
 - iv. [Resource Management](#)
 - i. [Design](#)
 - ii. [Implementation](#)
 - i. [Helper Functions](#)
 - ii. [Public API](#)
 - iii. [On Your Own](#)
 - iv. [Sample Usage](#)
 - v. [Implementing a camera](#)
 - i. [Halfspace Culling](#)
 - ii. [Frustum Culling](#)
11. [Advanced Topics](#)
- i. [Multi-Texturing](#)
 - ii. [Sub Textures](#)
 - iii. [Billboarding](#)
 - iv. [Normal Mapping](#)
 - v. [Specular Mapping](#)
 - vi. [Ambient Occlusion](#)
 - vii. [Outlines](#)
 - viii. [Planar Shadows](#)
 - ix. [Stencyl Shadows](#)
 - x. [Shadow Mapping](#)
 - xi. [Reflections](#)
 - xii. [Cube Mapped Reflections](#)
 - xiii. [Particles](#)
 - xiv. [Motion Blur](#)
 - xv. [Terrains](#)

Introduction

This tutorial is about learning OpenGL through C# with the OpenTK wrapper. The book mostly follows along with [Beginning OpenGL Game Programming](#), first edition. Some parts more closely resemble [OpenGL Distilled](#) and there is code in here taken from [The red book](#), second edition.

License

This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or distribute this software, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

In jurisdictions that recognize copyright laws, the author or authors of this software dedicate any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to <http://unlicense.org>

OpenGL

OpenGL was originally developed by Silicon Graphics, Inc. (SGI) as a multi-purpose platform-independent graphics API. Since 1992, the development of OpenGL has been overseen by the OpenGL Architecture Review Board (ARB), which is made up of graphics vendors and other industry leaders such as ATI, Dell, HP, IBM, Intel, Microsoft, NVIDIA, etc... The role of the ARB is to establish and maintain the OpenGL specification which dictate which features must be included when one is developing an OpenGL distribution.

OpenGL History

The current OpenGL version is 4.5. OpenGL 4.5 will continue to be developed but the API is branching off into a more modern API called Vulkan. The two will both be maintained for some time, until the old OpenGL api is phased out.

We are going to cover OpenGL 1.5. It is an older version of the API, but it's the perfect introduction into 3D graphics. It's easy to use and understand. We will delve into more modern versions of the API later.

The designers of OpenGL knew that hardware vendors would want to add features that may not be exposed by default to the OpenGL interface. To address this they included a method for extending OpenGL. Hardware vendors can create OpenGL extensions. When other vendors start implementing the functionality as well and the extension gains enough support it is submitted for review to the ARB. It is up to the ARB to promote the extension into the core specification of OpenGL.

By default the framework we will use to access OpenGL, OpenTK exposes extensions to us. We do not need to worry about having to manually load them.

Software V Hardware

Initially OpenGL 1.0 was a software renderer. It would crunch numbers, do math and "render" a 3D scene all on the CPU. Right around 1998 graphics cards started getting more and more popular. Graphics cards have specialized chips on them that can perform simple matrix math MUCH faster than the CPU can.

OpenGL 1.0 was able to use the GPU to render using Extensions. Hardware transform & lighting was promoted to core in OpenGL 1.1. By OpenGL 1.5 all graphics calculations happen on the GPU

OpenGL Architecture

OpenGL is a collection of several hundred functions providing access to all the features offered by your graphics card. Internally OpenGL functions as a state machine, a collection of states that tell the OpenGL what to do and are changed in a very well-defined manner. Using the OpenGL API you can set various aspects of the state machine, including things such as the current color, lighting and blending.

When rendering, everything drawn is drawn according to the rules set inside the OpenGL statemachine. It's important to be aware of what the various states are and how they effect what gets rendered. It is not uncommon at all to get a black screen with nothing being rendered simply because you forgot to set a single state. Graphics Programming is pretty finicky, and you kind of need to know what you are doing.

OpenTK

What is OpenTK? The Open Toolkit is an advanced, low-level C# library that wraps OpenGL, OpenCL and OpenAL. It is suitable for games, scientific applications and any other project that requires 3d graphics, audio or compute functionality.

What this means is OpenTK takes several low level C libraries (OpenGL, OpenCL and OpenAL) and exposes them to as a higher level C# Interface.

Other options

OpenTK is not the only C# binding of OpenGL. There are a lot of others to choose from. The most notable one is the TAO framework.

With other options, why use OpenTK? First, it's the most actively supported binding. More importantly, OpenTK attempts to leverage the language design of C# to solve issues that OpenGL has inherited from C. This is something no other binding is doing.

Problem with C

OpenGL is natively implemented as a C-API. C does not have Objects, only functions and memory. As such the API can be hard to navigate. For example, every state switch is one big enumeration. This means the following states (GL_TRIANGLES, GL_QUADS, GL_COLOR_BIT) are all really integers. So given the following function:

```
void glBegin(int primitiveState);
```

Which of the above states is valid here? GL_TRIANGLES and GL_QUADS are. The only way to know that is to go through the docs. What happens if you pass GL_COLOR_BIT to the function? Nothing renders!

As you can imagine intellisense is of no help either. The second you type GL_ about 500 options pop up. There is no indication what is appropriate and what is not.

Solution with C-Sharp

To solve this problem OpenTK defines its own enumerations. Many of them, that map to the huge list of C enumerations. For instance, in OpenTK has the following enum:

```
enum PrimitiveType { GL_TRIANGLES, GL_QUADS, GL_FAN };
```

Now when you look at this function

```
void glBegin(PrimitiveType primitiveState);
```

Visual studios intellisense will tell you that the argument is of type `PrimitiveType` and when you type in `PrimitiveType.` intellisense will pop up only the relevant items in the correct enumeration

Other improvements

OpenTK is filled with other similar improvements that make writing OpenGL programs in C# much faster and less error prone than writing them in C. The OpenTK framework is available in Visual Studio's package manager NuGet.

Take some time and check out the [official website](#) which has some cool guides and nifty code samples. Definately take a peek at [the OpenTK docs](#) which are written in a book-like fashion.

Getting a window

One of the most important things we need to do is have a window where we can actually see what gets rendered. This section will walk you through the details of an OpenTK window.

Some of it might look familiar, the 2DOpenTKFramework projects use this same code.

Getting a window

Most of this is going to look familiar. We used OpenTK to manage windows in the OpenTK2DFramework project. This chapter just provides more detailed information about what is happening.

The `openTK` namespace contains the `GameWindow` class. OpenTK can either manage the game window for you, or be embedded in a windows forms application. For the sake of speed, we will let OpenTK create and manage the main window for us. This is the minimum code to get a window up and running:

```
using System;
using OpenTK;
using System.Drawing;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        // global reference
        public static OpenTK.GameWindow Window = null;

        [STAThread]
        public static void Main() {
            //create static (global) window instance
            // take note, we set make a new MainGameWindow object, which inherits from OpenTK.GameWindow
            Window = new MainGameWindow();

            //run game at 60fps. will not return until window is closed
            Window.Run(60.0f);

            // We are finished with the window, clean up memory
            Window.Dispose();
        }
    }
}
```

What happened?

We created a static variable called `window` of type `OpenTK.GameWindow`. This is a public static variable to allow global access. If you don't need to access the window from outside your class you could have that variable be local to the `Main` function. It doesn't have to be static.

The `Main` function has to be marked as `STAThread` as with other non-form C# applications.

The first line of `Main`, `Window = new MainGameWindow();` creates a new window object. When created, the window object allocates memory for the new window and initializes everything the window needs.

Why do we create a `MainGameWindow` object instead of a raw `OpenTK.GameWindow` object? Because the `MainGameWindow` extends the `OpenTK.GameWindow` class this will work. For now you could replace that with a `new OpenTK.GameWindow()` call and everything would still work. But later we will need to override methods in the `OpenTK.GameWindow` class.

The next line `Window.Run(60.0f);` actually shows the window. After the window is shown this method will enter the update loop. The window will try to run at 60FPS if it can. The actual update speed is up to you.

The `Run` method of the window only returns when the window is closed. For the duration of the application this method will be executing.

Once the window is closed and `Run` returns we have to clean up any memory it allocated and all resources it's eating up. That's what the `Dispose` method does.

Callbacks

In OpenTK you don't have access to the windows message loop. Instead OpenTK provides a number of very useful callbacks that you can use to do things during the lifecycle of the window. Set all of the callbacks before calling the `Run` method of the window.

Initialize

The initialize function gets called right after the window is created, before the windows first update / render cycle.

The function takes a sender `object` and a `EventArgs` event, it returns void. This is how you can hook up the `Initialize` callback:

```
using System;
using OpenTK;
using System.Drawing;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        //reference to OpenTK window
        public static OpenTK.GameWindow Window = null;

        public static void Initialize(object sender, EventArgs e) {
            // Do initialization code here
        }

        [STAThread]
        public static void Main() {
            //create static(global) window instance
            Window = new MainGameWindow();

            //hook up the initialize callback
            Window.Load += new EventHandler<EventArgs>(Initialize);

            //set window title and size
            Window.Title = "Game Name";
            Window.ClientSize = new Size(800, 600);

            // turn on vsynch, prevents screen tearing when swapping buffers
            // This is on by default, but take no changes!
            Window.VSync = VSyncMode.On;

            //run game at 60fps. will not return until window is closed
            Window.Run(60.0f);
            Window.Dispose();
        }
    }
}
```

Update

The Update function gets called 1 time every frame. The function takes a sender `object` and a `FrameEventArgs` event, it returns void. You need to hook up to the `UpdateFrame` method of the OpenTK window.

You can get the delta time of the update loop trough the `FrameEventArgs` argument of the function.

```

// ...

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        // ...

        public static void Update(object sender, FrameEventArgs e) {
            float deltaTime = (float)e.Time;
        }

        [STAThread]
        public static void Main() {
            // ...

            Window.UpdateFrame += new EventHandler<FrameEventArgs>(Update);

            // ...

            Window.Run(60.0f);
            Window.Dispose();
        }
    }
}

```

Render

The Render function is similar to the Update function. The function gets hooked up to the windows `RenderWindow` callback. Just like Update the Render function takes a `sender object` and a `FrameEventArgs event`, it returns `void`.

Because the second argument is a `FrameEventArgs event` you can access delta time the same way you did for Update if you need it, tough generally in a Render function you wont.

Inside this function we need to do the following

- Tell OpenGL what color to clear the screen to
- Clear the screen
- Render our game
- Swap the back and front display buffers
 - These are managed by OpenTK.
 - OpenTK is by default double buffered.

This code contains the first two lines of OpenGL we are going to write. Don't worry about what they mean yet, for now just copy them in.

```

// ...

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        public static OpenTK.GameWindow Window = null;

        // ...

        public static void Render(object sender, FrameEventArgs e) {
            // Tell OpenGL what color to clear the screen to
            GL.ClearColor(Color.CadetBlue);
            // Tell OpenGL to clear the screen.
            GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

            // TODO: Render your game here

            Window.SwapBuffers();
        }
    }
}

```

```

    }

    // ...

    [STAThread]
    public static void Main() {
        // ...

        Window.RenderFrame += new EventHandler<FrameEventArgs>(Render);

        // ...

        Window.Run(60.0f);
        Window.Dispose();
    }
}

```

Resize

The resize callback is a bit special. Not in the good way. This callback is the whole reason that our `MainGameWindow` class has to extend the `OpenTK.GameWindow` class.

Resize does not have a callback event, instead the `OpenTK.GameWindow` class implements it as a virtual function that we must override. This function will take a `EventArgs` event for an argument and return `void`.

Getting the width and height of the window on resize is also a bit dumb. That information is not passed in with the event argument (practically nothing is!). Instead you must check a variable called `ClientRectangle` that is inherited from `OpenTK.Window`. `ClientRectangle` is a [System.Drawing.Rectangle](#) object.

```

// ...

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {

        // ...

        protected override void OnResize(EventArgs e) {
            // You must call this!
            base.OnResize(e);

            // ClientRectangle is inherited from OpenTK.GameWindow
            Rectangle drawingArea = ClientRectangle;

            // Do resize window stuff here
        }

        // ...

        [STAThread]
        public static void Main() {

            // ...

        }
    }
}

```

Shutdown

The shutdown callback is similar to the initialize callback. The function takes a sender `object` and a `EventArgs`

event, it returns void. You hook shutdown up the the windows `Unload` callback

```
// ...

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {

        // ...

        public static void Shutdown(object sender, EventArgs e) {

        }

        [STAThread]
        public static void Main() {

            // ...

            Window.Unload += new EventHandler<EventArgs>(Shutdown);

            // ...

            Window.Run(60.0f);
            Window.Dispose();
        }
    }
}
```

All together

The code below puts all of the above callbacks in place. It also sets the window Title and Size, through public getters and setters exposed to the `GameWindow` class. This should be everything we need for the main window.

If you're curious, a complete list of the callbacks and properties of the `GameWindow` class can be found on the [OpenTK Doxygen](#) documentation page.

```
using System;
using OpenTK;
using System.Drawing;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        //reference to OpenTK window
        public static OpenTK.GameWindow Window = null;

        public static void Initialize(object sender, EventArgs e) {

        }

        public static void Update(object sender, FrameEventArgs e) {
            float deltaTime = (float)e.Time;
        }

        public static void Render(object sender, FrameEventArgs e) {
            GL.ClearColor(Color.CadetBlue);
            GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

            // TODO: Render your game here

            Window.SwapBuffers();
        }

        protected override void OnResize(EventArgs e) {
            // You must call this!
        }
    }
}
```

```
base.OnResize(e);

// ClientRectangle is inherited from OpenTK.GameWindow
Rectangle drawingArea = ClientRectangle;
}

public static void Shutdown(object sender, EventArgs e) {

}

[STAThread]
public static void Main() {
    //create static(global) window instance
    Window = new MainGameWindow();

    //hook up the initialize callback
    Window.Load += new EventHandler<EventArgs>(Initialize);
    //hook up the update callback
    Window.UpdateFrame += new EventHandler<FrameEventArgs>(Update);
    //hook up render callback
    Window.RenderFrame += new EventHandler<FrameEventArgs>(Render);
    //hook up shutdown callback
    Window.Unload += new EventHandler<EventArgs>(Shutdown);

    //set window title and size
    Window.Title = "Game Name";
    Window.ClientSize = new Size(800, 600);

    // turn on vsynch, prevents screen tearing when swapping buffers
    // This is on by default, but take no changes!
    Window.VSync = VSyncMode.On;

    //run game at 60fps. will not return until window is closed
    Window.Run(60.0f);

    Window.Dispose();
}
}
}
```

Full Screen

Most games have the ability to switch between full screen mode and windowed mode. A lot of times this is implemented as a menu option. It used to be standard to toggle fullscreen mod with the `Alt+Enter` shortcut , however that trend has been slowly dying out.

How you want to implement your fullscreen toggle logic is up to you, i'm just going to show you how to actually toggle. If you end up accidentally going into fullscreen mode press `Alt+Tab` to bring up the program selection toolbox. Navigate to Visual studio (Keep alt pressed down while tapping the tab button to move between programs) and use the red square to terminate the application.

API

We can switch our game to fullscreen mode by setting a few variables inherited from `OpenTK.GameWindow` .

First, we need to set the `WindowBorder` variable. It is an enum of type `WindowBorder` . The values that interest us are `WindowBorder.Resizable` for a regular window and `WindowBorder.Hidden` for a fullscreen window.

Next, we want to set the `WindowState` variable. It is an enum of type `WindowState` . The values that we care about are `WindowState.Normal` for windowed mode and `WindowState.Fullscreen` for fullscreen mode.

One last thing, when going from fullscreen to windowed mode you need to reset the client size the same way you did when we first created the window:

```
ClientSize = new Size(800, 600);
```

Failure to do so will result in undefined behaviour. On my machine the window shrinks by 1 pixel every frame until it has a size of 0

Implementation

The first thing we need to do is add a new boolean to the `MainGameWindow` class to keep track of the current window mode (windowed or full screen).

```
using System;
using OpenTK;
using System.Drawing;
using OpenTK.Graphics.OpenGL;
using GameFramework;

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        // reference to OpenTK window
        public static OpenTK.GameWindow Window = null;

        // keep track of window mode
        public static bool IsFullscreen = false;
```

Next we implement the actual toggle function

```
public static void ToggleFullscreen() {
    if (IsFullscreen) {
        Window.WindowBorder = WindowBorder.Resizable;
        Window.WindowState = WindowState.Normal;
        Window.ClientSize = new Size(800, 600);
    }
    else {
        Window.WindowBorder = WindowBorder.Hidden;
        Window.WindowState = WindowState.Fullscreen;
    }
    IsFullscreen = !IsFullscreen;
}

// ... Rest of MainGameWindow class
```

Keep in mind, when exiting fullscreen mode you MUST reset the window to its original size, otherwise the behaviour is undefined.

That's it. You can now call the `ToggleFullscreen` function from a menu, or hook it up to `Alt+Enter` your call.

Running your code

Now that you have a window set up, running with a blue clear screen (run the game, make sure you do!) the question is where should you write game code?

Turns out that question has a LOT of answers. You need to be able to choose the fastest solution, the one you are most comfortable with and build your project on that.

1) Inline

This is the easiest way to do things. I would only suggest writing inline code for small demo programs that you mainly use to test things out for yourself. If you expect the program to be large, don't do this.

Inline code is kind of what it sounds. The `MainWindow` class already has static Initialize, Update, Render and Shutdown functions. Just write your code inside of these functions and call it a day.

2) Static Reference

This is my personal favorite. You don't really need anything special to make it work. Let's assume you have a `Game` class.

```
class Game {
    public void Initialize() {

    }

    public void Update(float dt) {

    }

    public void Render() {

    }

    public void Shutdown() {

    }
}
```

You simply make a static reference variable to the Game class inside the `MainWindow` class, just like you have a `Window` variable referencing an OpenTK window.

```
using System;
using OpenTK;
using System.Drawing;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        //reference to OpenTK window
        public static OpenTK.GameWindow Window = null;

        // reference to game
        public static Game TheGame = null;
    }
}
```

```

public static void Initialize(object sender, EventArgs e) {
    TheGame.Initialize();
}

public static void Update(object sender, FrameEventArgs e) {
    float deltaTime = (float)e.Time;
    TheGame.Update(deltaTime);
}

public static void Render(object sender, FrameEventArgs e) {
    GL.ClearColor(Color.CadetBlue);
    GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
    TheGame.Render();
    Window.SwapBuffers();
}

public static void Shutdown(object sender, EventArgs e) {
    TheGame.Shutdown();
}

[STAThread]
public static void Main() {
    //create static (global) window instance
    Window = new MainGameWindow();

    // create static (global) game instance
    TheGame = new Game();

    // ...
}

```

3) Singleton

This method should be familiar by now, it's the one we used for both Mario and the DungeonCrawler project. It's very similar to approach #2, except instead of storing a static reference to the `Game` class in the `MainGameWindow` class you store a reference to it in the `Game` class through a singleton.

A singleton is just a static way to access a SINGLE instance of a class. We ensure a class can only have a single instance by making its constructor private.

```

class Game() {
    // Private constructor.
    // Only the Game class can make a new Game instance.
    private Game() {

    }

    // Global reference to the single instance of this class
    private static Game instance = null;

    // Public access (read only) to the game instance
    public static Game Instance {
        get {
            // Lazy initialization
            if (instance == null) {
                // Remember, only this class can make a new instance ;
                instance = new Game();
            }
            return instance;
        }
    }
}

```

Input

For the sake of completeness i'm going to show you how to read raw input here. While this is usefull, the `InputManager` from the `GameFramework` still works just fine (Assuming you set it up correctly). I highly suggest using the `InputManager` or rolling your own version of it.

After reading this, go ahead and browse trough the `InputManager` class to get a better understanding of what it's doing and how it's doing it. The `InputManager` is just a convenient abstraction that provides buffered input instead of raw input.

OpenTK.Input

All input related function reside within the `openTK.Input` namespace. This namespace has several enumerations and classes we care about:

- `OpenTK.Input.Key`
 - Enumeration of keyboard keys
- `OpenTK.Input.MouseButton`
 - Enumeration of mouse buttons
- `OpenTK.Input.KeyboardState`
 - Structure containing keyboard information
- `OpenTK.Input.MouseState`
 - Structure containing mouse information

For a compleate overview of the namespace, check out [the doxygen](#) pages for OpenTK.

Keyborad Input

In order to read the keyboard, all you need is access to a `KeyboardState` instance. Read [this article](#) before proceeding.

There above article get a `Keyboardstate` reference using this code:

```
KeyboardState state = OpenTK.Input.Keyboard.GetState();
```

However this functionality is deprecated. It might get compleatly removed from future versions of OpenTK. Instead, to get a `KeyboardState` instance, you need access to an instance of `openTK.GameWindow` class.

Lucky with our windowing code:

```
using System;
using OpenTK;
using System.Drawing;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        // reference to OpenTK window
        public static OpenTK.GameWindow Window = null;
```

The static variable `MainGameWindow.Window` is accessible from anywhere. So, get a reference to the `KeyboardState` structure using the `Keyboard` getter of the `OpenTK.GameWindow` class. Like so:

```
// Call somewhere in the update loop
void CheckInput() {
    KeyboardState keboard = MainGameWindow.Window.Keyboard;
}
```

There is only one property of `KeyboardState` that we care about, and that is the fact that the `[]` accessor is implemented to return a `bool`. This override takes a `OpenTK.Input.Key` enum value as an argument and returns true if the button is being held down, false if it is up.

This is how you could check if the button A is down

```
// Call somewhere in the update loop
void CheckInput() {
    KeyboardState keboard = MainGameWindow.Window.Keyboard;

    if (keyboard[OpenTK.Input.Key.A]) {
        Console.WriteLine("A is down");
    }
}
```

We can take advantage of the fact that each enum entry is essentially an integer and print out any key that is down:

```
// Call somewhere in the update loop
void CheckInput() {
    KeyboardState keboard = MainGameWindow.Window.Keyboard;

    // LastKey is the number of keys on the keyboard + 1
    int numKeys = (int)OpenTK.Input.Key.LastKey;

    for (int i = 0; i < numKeys; ++i) {
        OpenTK.Input.Key iAsKey = (OpenTK.Input.Key)i;

        if (keyboard[iAsKey]) {
            Console.WriteLine(iAsKey.ToString() + " is down");
        }
    }
}
```

Mouse Input

Mouse input is similar to the keyboard input. Just like with the keyboard there are some [outdated docs](#) on the OpenTK site about working with a mouse.

Again, the first thing you need is an instance of the `OpenTK.Input.MouseState` class. You can get this through the `Mouse` getter of the `OpenTK.GameWindow` class.

```
// Call somewhere in the update loop
void CheckInput() {
    MouseState mouse = MainGameWindow.Window.Mouse;
}
```

Just like the `KeyboardState`, `MouseState` also overrides the `[]` accessor. It takes a `OpenTK.Input.MouseButton` enum value as an argument. It returns true if the mouse button is down, false if it is up. You can check for a click like this:

```
// Call somewhere in the update loop
void CheckInput() {
    MouseState mouse = MainGameWindow.Window.Mouse;

    if (mouse[OpenTK.Input.MouseButton.Left]) {
        Console.WriteLine("Left Mouse Button Is Down!");
    }
}
```

Getting the X and Y positions of the mouse is stupid simple. The `MouseState` has public `x` and `y` accessors. They are both of type int. You can use them like this:

```
// Call somewhere in the update loop
void CheckInput() {
    MouseState mouse = MainGameWindow.Window.Mouse;
    int x = mouse.X;
    int y = mouse.Y;

    if (mouse[OpenTK.Input.MouseButton.Left]) {
        Console.WriteLine("Left Mouse Button Is Down!");
    }

    if (x > 20 && x < 50 && y > 20 && y < 50) {
        Console.WriteLine("Mouse is inside Rectangle(20, 20, 50, 50)");
    }
}
```

Gamepad Input

Gamepad support in OpenTK is in a very sad state. It's super duper broken. I'm not even going to cover it here. If you are interested, you can check out the `InputManager` code behind joystick support, but it's a verbose ugly hack!

Buffered Input

Using the `InputManager` we can check more states of buttons. For example we don't just have access to a bool indicating if a key is down or up, we can also check if the key was pressed this frame, or released this frame. That's called buffered input, and it's the main high level concept that the `InputManager` provides.

If you want to implement your own buffered input, it's not difficult. The same principles that apply to double buffered rendering apply to buffered input:

- Make a front-buffer.
 - Array of bool, size is the number of available keys
 - Will hold the key states of this frame
- Make a back buffer
 - Array of bool, size is the number of available keys
 - Will hold the key states of the last frame
- In update
 - Copy the contents of the front-buffer INTO the back-buffer
 - Copy the current keyboard state INTO the front-buffer
- In update, after the input buffers have been updated

- A key is down if:
 - it is true in the front buffer
- A key is up if:
 - it is false in the front buffer
- A key was pressed this frame if:
 - it is true in the front-buffer
 - it is false in the back-buffer
- A key was released this frame if:
 - it is false in the front-buffer
 - it is true in the back-buffer

The code for this might look something like this:

```
using System;
using OpenTK;
using System.Drawing;
using OpenTK.Graphics.OpenGL;
using OpenTK.Input;

namespace GameApplication {
    class MainGameWindow : OpenTK.GameWindow {
        public static OpenTK.GameWindow Window = null;

        private static bool[] KeyboardFront = null;
        private static bool[] KeyboardBack = null;

        public static void Initialize(object sender, EventArgs e) {
            int numKeys = (int)OpenTK.Input.Key.LastKey;

            KeyboardFront = new bool[numKeys];
            KeyboardBack = new bool[numKeys];

            for (int i = 0; i < numKeys) {
                KeyboardFront[i] = false;
                KeyboardBack[i] = false;
            }
        }

        private static void UpdateInput() {
            int numKeys = (int)OpenTK.Input.Key.LastKey;

            for (int i = 0; i < numKeys; ++i) {
                KeyboardBack[i] = KeyboardFront[i];
            }
            for (int i = 0; i < numKeys; ++i) {
                OpenTK.Input.Key iAsKey = (OpenTK.Input.Key)i;
                KeyboardFront[i] = Window.Keyboard[iAsKey];
            }
        }

        private static bool KeyDown(OpenTK.Input.Key key) {
            return KeyboardFront[key];
        }

        private static bool KeyUp(OpenTK.Input.Key key) {
            return !KeyboardFront[key];
        }

        private static bool KeyPressed(OpenTK.Input.Key key) {
            return KeyboardFront[key] && !KeyboardBack[key];
        }

        private static bool KeyReleased(OpenTK.Input.Key key) {
            return !KeyboardFront[key] && KeyboardBack[key];
        }

        public static void Update(object sender, FrameEventArgs e) {
            float deltaTime = (float)e.Time;
```

```
UpdateInput();
bool altDown = KeyDown(OpenTK.Input.Key.AltLeft) || KeyDown(OpenTK.Input.Key.AltRight);

if (altDown && KeyPressed(OpenTK.Input.Key.Enter)) {
    ToggleFullscreen();
}
}

// ... Rest of the MainGameWindow class
```

That's all it takes to implement buffered input. In this example, we use the keyboard `Alt+Enter` to toggle in and out of full screen. Of course in general you don't want to clutter up your window with this kind of code, unless it makes sense to you.

OpenGL States and Primitives

Now it's time to finally get into the meat of OpenGL! To begin to unlock the power of OpenGL, you need to start with the basics and that means understanding primitives. Before we start, we need to discuss something that is going to come up during our discussion of primitives and pretty much everything else from this point on. The OpenGL state machine.

The OpenGL state machine consists of hundreds of settings that affect various aspects of rendering. Because the state machine will play a role in everything you do, it's important to understand what the default settings are, how you can get information about the current settings and how to change those settings. Several generic functions are used to control the state machine, so we will look at those here.

GL

Within OpenTK, there is a class called GL. All OpenGL functions are implemented as static function of the GL class. For example, to begin a primitive you would:

```
GL.Begin(PrimitiveType.Points);
```

State functions

OpenGL provides a number of multipurpose functions that allow you to query the OpenGL state machine. Most of these functions begin with `GL.Get...`. The most generic of these functions will be covered here, the rest will be covered as we use them in later chapters.

Querying numeric states

There are four general purpose functions that allow you to retrieve numeric (or Boolean) values in the OpenGL state machine:

- `bool GL.GetBoolean(GetPName pname)`
- `double GL.GetDouble(GetPName pname)`
- `float GL.GetFloat(GetPName pname)`
- `int GL.GetInteger(GetPName pname)`

`GetPName` is an enumeration containing all of the states we may want to query. In each of the function, we query the state represented by its name.

Of course determining the current state machine settings is interesting, but not nearly as interesting as being able to change those settings. Contrary to what is expected, there is no `GL.set...` or similar function for setting state machine values. Instead there is a variety of more specific functions, which we will discuss as they become relevant.

Enabling and disabling states

We know how to find out the states in the OpenGL state machine, so how do we turn states on and off? Enter the `GL.Enable` and `GL.Disable` functions.

```
void GL.Enable(EnableCap enableCap);
void GL.Disable(EnableCap enableCap);
```

The `cap` parameter represents the OpenGL capability you want to turn on or off. `GL.Enable` turns it on, conversely `GL.Disable` turns it off. Some of the more common caps are `EnableCap.Blend` (for texture blending), `EnableCap.Texture2D` (for texturing) and `EnableCap.Lighting` for lighting.

GL.IsEnabled

Often times you just want to find out whether a particular OpenGL capacity is on or off. Although this can be done with `GL.GetBoolean` it's easier to do with `GL.IsEnabled`.

```
bool GL.IsEnabled(EnableCap enableCap);
```

It's easier to use this function because it takes an `EnableCap` enum as its argument, instead of the `GetPName` enum that `GL.GetBoolean` takes.

Querying string values

You can find out details of the OpenGL implementation being used at runtime with the following function

```
string GL.GetString(StringName name)
```

The `StringName` enum only has the following five properties:

- **Vendor**: The returned string indicates the name of the company whose OpenGL implementation you are using. For example, the vendor string for API drivers is *API Technologies Inc.*
- **Renderer** The string contains information that usually reflects the hardware being used. For example *Radeon 9800 Pro X86/MMX/3DNow!/SSE*.
- **Version** This string contains the version number formatted either as: `major.minor` OR `major.minor.patch`, possibly followed by additional information.
- **Extensions** The string returned contains a space-delimited list of all of the OpenGL extensions your driver supports. This is not very useful for OpenTK.
- **ShadingLanguageVersion** this is the version of GLSL your graphics card supports. We will not be dealing with shaders for a while, for now this is safe to ignore.

It's good practice to print out the **Vendor**, **Renderer** and **Version** strings on separate lines in your Initialize function. This will give you a feel for what hardware you are running on.

Finding errors

When you make an error OpenGL will usually not throw an exception, instead nothing will render. Finding errors in OpenGL is a hard task.

Passing incorrect values to OpenGL functions causes an error flag to be set. When this happens the functions will return without actually doing anything. To track down the problem you have to query the state machine error flag, you can do so with this function:

```
ErrorCode GL.GetError();
```

The return values of the error are documented on OpenTK's doxygen page.

Giving OpenGL a Hint

Some operations in OpenGL will vary slightly from implementation to implementation (or driver to driver). An attempt was made to allow developers some form of control over these variations.

Not all implementations follow the command, the `GL.Hint` function allows you to specify your desired level of trade off between image quality and speed for several different behaviours.

```
void GL.Hint(HintTarget target, HintMode mode)
```

The target parameter specifies the behaviour you want to control. The hint parameter is one of three options

- `HintMode.Fastest` is used to hint that the fastest and most efficient implementation should be used, possibly sacrificing image quality.
- `HintMode.Nicest` is used to indicate that the highest quality implementation should be used, even at the cost of execution speed / performance.
- `HintMode.DontCare` indicates that you don't have a preference. Let your OpenGL implementation decide what to use.

Lets take drawing a line for example. If you are drawing a few lines and you want them to be as smooth as possible, not pixelated at all you could set the following hint:

```
GL.Hint(HintTarget.LineSmoothHint, HintMode.Nicest);
```

If you are rendering millions of lines for whatever reason, anti-aliasing them all will give you a huge drop in performance. You could tell OpenGL to not anti-alias them and just draw the lines jagged / pixelated with:

```
GL.Hint(HintTarget.LineSmoothHint, HintMode.Fastest);
```

Handling Primitives

So, what are primitives? Simply put primitives are basic geometric entities such as points, lines and triangles.

You will be using thousands and thousands of these primitives to make your games, so it is important to know how they work. Before we get into specific primitive types, we need to talk about a few OpenGL functions that you will be using often for working with primitives. The first is `GL.Begin`. The function has the following prototype:

```
void GL.Begin(PrimitiveType mode);
```

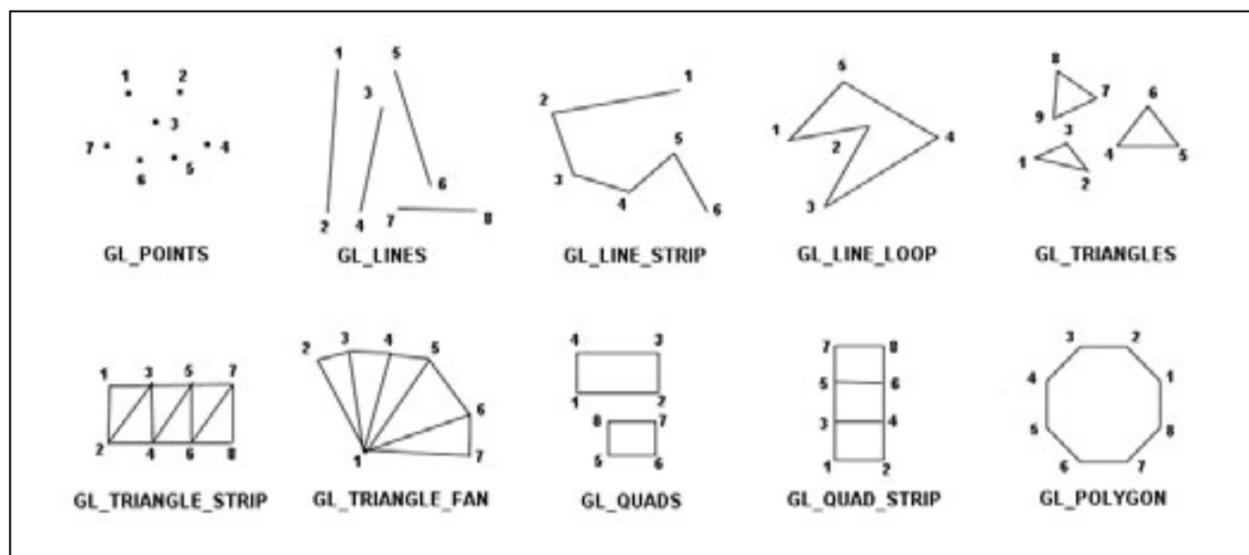
You use `GL.Begin` to tell OpenGL two things:

- That you are ready to start drawing
- The primitive type you want to draw

You specify the primitive type with the `PrimitiveType` enum. It's values are:

- **PrimitiveType.Points** Individual points
- **PrimitiveType.Lines** Line segments composed of pairs of vertices
- **PrimitiveType.LineStrip** A series of connected lines
- **PrimitiveType.LineLoop** A closed loop of connected lines. The last segment is automatically created between the first and last vertices.
- **PrimitiveType.Triangles** Single triangles as a triplet of vertices. This is what you will use most often!
- **PrimitiveType.TriangleStrip** Series of connected triangles
- **PrimitiveType.TriangleFan** Triangle around a common, central vertex
- **PrimitiveType.Quads** A quadraletiral (Polygon with 4 vertices)
- **PrimitiveType.QuadStrip** A series of connected quadralatirals
- **PrimitiveType.Polygon** A convex polygon with an arbitrary number of vertices.

Here is an example of what each primitive would draw like:



Each call to `GL.Begin` needs to be accompanied by a call to `GL.End`, the signature of this function looks like:

```
void GL.End();
```

`GL.End` tells OpenGL that you are done drawing the primitive you specified with `GL.Begin`.

`GL.Begin()` and `GL.End()` blocks may **not** be nested.

Not all OpenGL functions can be used inside a Begin / End block. Calling an invalid function will generate a `InvalidOperation` error and render nothing.

This is a list of valid functions that can be used between Begin and End:

- `GL.Vertex*()`
- `GL.Color*()`
- `GL.SecondaryColor*()`
- `GL.Index*()`
- `GL.Normal*()`
- `GL.TexCoord*()`
- `GL.MultiTexCoord*()`
- `GL.FogCoord*()`
- `GL.ArrayElement()`
- `GL_EvalCoord*()`
- `GL.EvalPoint*()`
- `GL.Material*()`
- `GL.EdgeFlag*()`
- `GL.CallList*()`
- `GL.CallLists*()`

Why do most of those functions end in an asterisk? Because they have multiple variations. Each variation has the asterisk replaced by a number, that number signifies how many arguments the function takes.

For example, `GL.Vertex` has the following variations:

```
void GL.Vertex2(float, float);
void GL.Vertex3(float, float, float);
void GL.Vertex4(float, float, float, float)
```

GL.Vertex

Lets talk about one particularly important function, the `GL.Vertex` function. This function needs to be called between `GL.Begin` and `GL.End` function calls. It is used to specify a point in space (A vertex), which is then interpreted appropriateley depending on the argument of `GL.Begin`.

The `GL.Vertex` function is so important because every object you draw with OpenGL is ultimateley described as an ordered set of vertices.

The version of this function you will use most often is `GL.Vertex3` which takes 3 floating point arguments (x, y, z coordinates) you need to get used to this multidimensional notation.

Lets see an example:

```
// Tell OpenGL we want to draw a line
GL.Begin(PrimitiveType.Lines);

// Tell OpenGL where the first point of the line is
GL.Vertex3(2.0f, 1.0f, 3.0f);

// Tell OpenGL where the second point of the line is
GL.Vertex3(6.0f, -1.0f, 8.0f);

// Tell OpenGL we are done drawing the line
GL.End();
```

Every time you specify a vertex other data becomes associated with it based on the current state of the OpenGL state machine. Data like the color of the vertex, texture and fog. We will cover all of this later, so no need to worry about it right now. The important thing to understand is that none of these things matter without a vertex to use them!

Drawing a Point with OpenGL

Rendering a point

It doesn't get any more primitive than a point, so that's what we should look at first. Drawing a point on screen is actually a really powerful tool, if you can draw a single pixel on screen you can draw anything! With that said, you can draw a point on screen by putting this code in your render function:

```
GL.Begin(PrimitiveType.Points);
    GL.Vertex3(0.0f, 0.0f, 0.0f);
GL.End();
```

Run this code (Seriously, when I say run this code I mean type it in and try it for yourself) and you will see a tiny white pixel in the middle of your screen.

The first line tells OpenGL that we are about to draw points, by passing the Points primitive type to `GL.Begin`. The next line tells OpenGL to draw a point at the origin (0, 0, 0). The last line tells OpenGL that we are done drawing.

Note, the indentation on the second line is optional. I indent my code that sits between Begin / End calls to make it easier to read.

What if you want to draw a second point at (0, 1 0)? Well you could type out:

```
GL.Begin(PrimitiveType.Points);
    GL.Vertex3(0.0f, 0.0f, 0.0f);
GL.End();
GL.Begin(PrimitiveType.Points);
    GL.Vertex3(0.0f, 1.0f, 0.0f);
GL.End();
```

However this is very inefficient! Every time you see a Begin / End block that is 1 draw call. The above code uses two draw calls to render two points.

A draw call is when the CPU has to upload data to the GPU. It is an expensive operation, you should aim to have as few draw calls as possible.

Take note that the primitive type passed to Begin is Points, plural. This suggests that between a single begin / end call you can render multiple points, and that is exactly the case.

In the real world, you would render two points like so:

```
GL.Begin(PrimitiveType.Points);
    GL.Vertex3(0.0f, 0.0f, 0.0f);
    GL.Vertex3(0.0f, 1.0f, 0.0f);
GL.End();
```

Modifying Point Size

OpenGL gives you a great deal of control when rendering through its state machine. There are many aspects of a single point you can change. Let's take a look at size. To modify the size of a point you use

```
void GL.PointSize(float size);
```

This results in a square whose width and height are both represented by the size argument. The default size is 1.0 If point antialiasing is disabled (which it is by default) the point size will be rounded to the nearest integer (with a minimum of 1).

If you want to get the current point size, you can do so with `GL.GetFloat` by passing `GetPName.PointSize` as its argument.

Here is an example of how this could be used:

```
float oldSize = GL.GetFloat(GetPName.PointSize);
// if a point was small, make it big. Otherwise make it 1!
if (oldSize < 1.0f) {
    GL.PointSize(5.0f);
}
else {
    GL.PointSize(1.0f);
}
```

Antialiasing points

Although you can specify primitives with almost infinite precision, there are a finite number of pixels on screen. This causes the edges of those primitives to look jagged. Anti-Aliasing is a method to smooth those edges, giving the polygon a more natural look.

You can enable anti-aliasing by passing `EnableCap.PointSmooth` to `GL.Enable`. You can disable it by passing the same parameter to `GL.Disable`. If you are unsure if point smoothing is currently enabled in the state machine, use `GL.IsEnabled` with the same argument.

```
// If anti-aliasing is disabled, enable it
if (!GL.IsEnabled(EnableCap.PointSmooth)) {
    GL.Enable(EnableCap.PointSmooth);
}
```

When anti-aliasing is enabled, not all pixel sizes may benefit. The specs say only a point size of 1.0 is guaranteed to get anti-aliased. Other sizes depend on your graphics card and OpenGL driver.

When anti aliasing is on, the current point size is used as the diameter of a circle, centered around the vertex.

Effect of distance

Normally points always occupy the same amount of space onscreen, regardless of how far away they are from the viewer. For some applications (particles, stars) points need to be larger if they are closer, and smaller if they are farther. You can do this with the `GL.PointParamater` function. The signature looks like this

```
void GL.PointParameter(PointParameterName param, int value);
void GL.PointParameter(PointParameterName param, int[] value);
void GL.PointParameter(PointParameterName param, float value);
void GL.PointParameter(PointParameterName param, float[] value);
```

The following are valid arguments:

- **PointSizeMin** Sets the lower bounds OpenGL will scale a point to, second argument is a float.
- **PointSizeMax** Sets the upper bounds OpenGL will scale a point to, second argument is a float.
- **PointDistanceAttenuation** Takes an Array of 3 floats that correspond to a, b, c of the attenuation algorithm $1 / (a + b * d + c * (d * d))$
- **PointFadeThreshold** Takes a single float, points smaller than this will start to fade out

Example

Follow along with this example, see what the results look like on your screen when you add this code to your render block:

```
float pointSize = 0.5f;
for (float pointPosition = -1.0f; pointPosition < 1.0f; pointPosition += 0.25f) {
    GL.PointSize(pointSize);

    GL.Begin(PrimitiveType.Points);
        GL.Vertex3(pointPosition, 0.0f, 0.0f);
    GL.End();

    pointSize += 1.0f;
}
```

Space

This is very important

Take note of where the points on screen are rendered. In the middle. This is because we start X at -1.

In OpenGL, **Point(0, 0)** is the middle of the screen. The left side is **X: -1** the right side is **X: 1**. Similarly the top is **Y: 1** and the bottom is **Y: -1**. Z also ranges from -1 to 1.

Another way to think about this, OpenGL draws inside of a cube. The far left corner of the cube is at (-1, -1, -1), the far right corner is at (1, 1, 1). These coordinates are called **Normalized Device Coordinates** or **NDC** for short.

Drawing lines in 3D

A line is just a connection between two points. As such, rendering a line is not too different from rendering two points. And because you already know how to do that, lets dive right in:

```
GL.Begin(PrimitiveType.Lines);
    GL.Vertex3(-2.0f, -1.0f, 0.0f);
    GL.Vertex3(2.0f, 1.0f, 0.0f);
GL.End();
```

This time you start off by passing `Lines` as the primitive type to `Begin` / `End` so that OpenGL knows that for every two vertices it needs to draw one line.

Just like with points, you can draw multiple lines within the `Begin` / `End` calls. If you wanted to render two lines, you would need to supply four vertices.

As with points, OpenGL allows you to change several parameters of the state machine that affect how your line is drawn. In addition to setting the line width and setting anti-aliasing you can also specify a stipple pattern.

Line Width

The default line width is 1.0. To find out the currently selected line with pass `GetPName.LineWidth` to `GL.GetFloat`. To change the line width, call the `GL.LineWidth` function:

```
void GL.LineWidth(float);
```

Anti aliasing lines

Anti Aliasing points works almost the same as it does for lines. You can toggle it by passing `EnableCap.LineSmooth` to `GL.Enable` and `GL.Disable`. You can check if it's turned on by passing the same enum value to `GL.IsEnabled`.

Again, as with points, when using anti-aliasing OpenGL implementations are only required to support it for line widths of 1.0f.

Stipple pattern

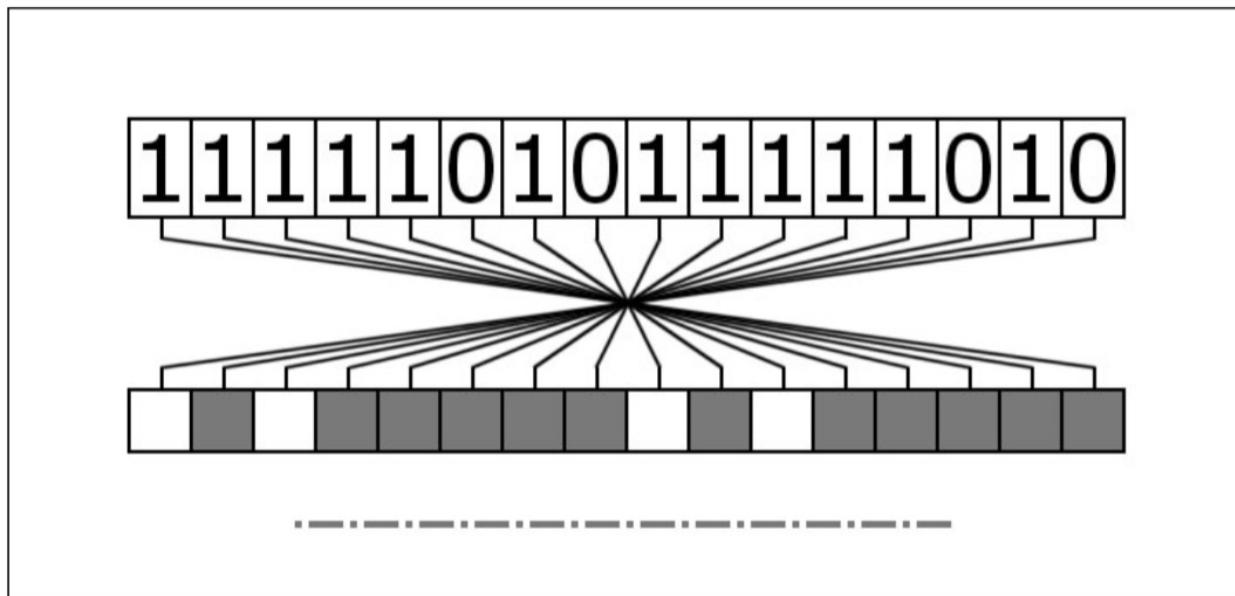
You can specify a stipple pattern with which to draw lines. The stipple pattern specifies a mask that will determine which portions of the line get drawn, therefore it can be used to create dashed lines. Before specifying a pattern, you must enable stipple-ing with `EnableCaps.LineStipple`. Then you set the stipple with the following function:

```
void GL.Stipple(int factor, short pattern);
```

The factor parameter defaults to 1. It is capped to be between 1 and 256. It's used to specify how many times each bit in the pattern is repeated before moving on to the next bit.

The pattern parameter is used to specify a 16-bit pattern. Any bits that are set will result in the corresponding pixels getting drawn. If a bit is not set, its pixel is not drawn. Something to be aware of, these bits are applied in reverse, so the low-order bit effects the left most pixel.

Here is an example of a bitmask, and how it correlates to rendering:



The following code enables line stippling and then specifies a pattern of alternating dashes and dots:

```
GL.Enable(EnableCaps.LineStipple);
short pattern = Convert.ToInt16("FAFA", 16);
// Convert.ToInt16 converts a hex string into a short.
// We are using the below hex:
// 0xFAFA OR 1111 1010 1111 1010

// Draw the pattern as 0101 1111 0101 1111
// because remember, it's reversed, low bit first.
GL.LineStipple(2, pattern);
```

You can determine the currently selected stipple pattern with:

```
int factor = GL.GetInteger(LineStippleRepeat);
short pattern = (short)GL.GetInteger(GetPName.LineStipplePattern)
```

Example

Follow along with this example

```
// Draw a series of lines that increase in width
float lineWidth = 0.5f;
for (float lineY = 1.0f; lineY > -1.0f; lineY -= 0.25f) {
    GL.LineWidth(lineWidth);

    GL.Begin(PrimitiveType.Lines);
    GL.Vertex3(-0.9f, lineY, 0.0f);
    GL.Vertex3(-0.1f, lineY, 0.0f);
    GL.End();
```

```

        lineWidth += 1.0f;
    }

    // Draw a series of lines with stipple
    lineWidth = 0.5f;
    GL.Enable(EnableCap.LineStipple);
    short pattern = Convert.ToInt16("AAAA", 16);
    GL.LineStipple(2, pattern);

    for (float lineY = 1.0f; lineY > -1.0f; lineY -= 0.25f) {
        GL.LineWidth(lineWidth);

        GL.Begin(PrimitiveType.Lines);
        GL.Vertex3(0.1f, lineY, 0.0f);
        GL.Vertex3(0.9f, lineY, 0.0f);
        GL.End();

        lineWidth += 1.0f;
    }

    // Try taking this out, see what happens!
    GL.Disable(EnableCap.LineStipple);

```

In the end, what happens if you take out that `GL.Disable()`? The short of it is, nothing automatically resets the OpenGL state machine at the end of the frame. **If you enable something, it will stay enabled** (across frames & draw calls) **until YOU disable it!**

The bolded sentence above is very important. More often than not when you have a bug it's because you set the state machine into some state, and didn't set it back to default afterwards

Drawing polygons in 3D

Although you can (and will) do some interesting things with points and lines, polygons will give you the most power to create immersive 3D worlds. With that being said, the rest of Chapter 3 will be spent discussing polygons. Before we get into the specific polygon types supported by OpenGL (Triangles, Quadralatirals, Generic Poligons) we need to discuss a few things that pertain to all polygon types.

You draw all polygons by specifying several points in 3D space. These points specify a region that is then filled with color. At least, that's the default behaviour. However as you probably expect by now, the state machine controls the way in which a polygon is drawn. Through the state machine you can change the drawing behaviour. Use the following function to change the way polygons are drawn:

```
void GL.PolygonMode(MaterialFace, PolygonMode);
```

OpenGL handles the front and back faces of polygons separately. A front face is any face of a polygon that is facing the camera. A back face is any face of a polygon that is facing away from the camera.

As a result, when you call `GL.PolygonMode` you need to change the face to which the change will apply. You do this by passing either `MaterialFace.Front`, `MaterialFace.Back` OR `MaterialFace.FrontAndBack` as the first parameter.

The following values are valid for the second parameter:

- **PolygonMode.Point** Each vertex is rendered as a single point. This basically produces the same effect as calling `GL.Begin` with `Lines` as the argument.
- **PolygonMode.Line** This will draw the edges of the polygon as a set of lines. This is *similar* to calling `GL.Begin` with `LineLoop`.
- **PolygonMode.Fill** This is the default state, which renders the polygon with the interior filled. This is the only state in which polygon smoothing takes effect.

If you want to see the back facing polygons of a model, you could render the front facing polygons as lines, and the back facing polygons solid. You can achieve this like so:

```
GL.PolygonMode(MaterialFace.Front, PolygonMode.Line);
GL.PolygonMode(MaterialFace.Back, PolygonMode.Fill);
```

If you have not changed the polygon mode for back facing poly's, the second line is redundant. This is because the default fill mode is `Fill`. But better safe than sorry.

On your own

Try to draw 5 squares (NOT CUBES) on screen. To draw a square, call `GL.Begin` with `PrimitiveType.Polygon` as its argument. Then put 4 vertices down.

Remember, the view goes from -1 to +1. Your vertices all have to be within that range. Keep their Z coordinates at 0.

Once all 5 are rendering, set the following polygon modes (each square will have a different mode)

- `GL.PolygonMode(MaterialFace.Front, PolygonMode.Line);`

- `GL.PolygonMode(MaterialFace.Back, PolygonMode.Point);`
- `GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);`
- `GL.PolygonMode(MaterialFace.Back, PolygonMode.Line);`
- `GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);`

See how the rendering changed.

Face Culling

Altough polygons are infinitley thin they still have two sides. They can be seen from either side. Sometimes, it makes sense to have each side displayed differently, and this is why some of the function presented require you to specify whether you're modifying the front face, back face or both. The takeaway is that rendering states for both sides of a polygon are stored seperately.

Most often the player will only be able to see one side of a polygon. It is possible to tell OpenGL to not render polygons that the player can't see! For example, if you are looking at a ball, only half of the polygons are visible. This is where front and back facing polygons get interesting. If your geometry has no holes in it, that is if your geometry is a solid hull (as most objects in the real world are then back facing polygons will not be seen).

So, why waste time and CPU power transforming and shading them? Well, the answer is dont. OpenGL can automatically not render faces for you trough a process called *culling*. To use polygon culling, you must first enable it by calling:

```
GL.Enable(EnableCap.CullFace);
```

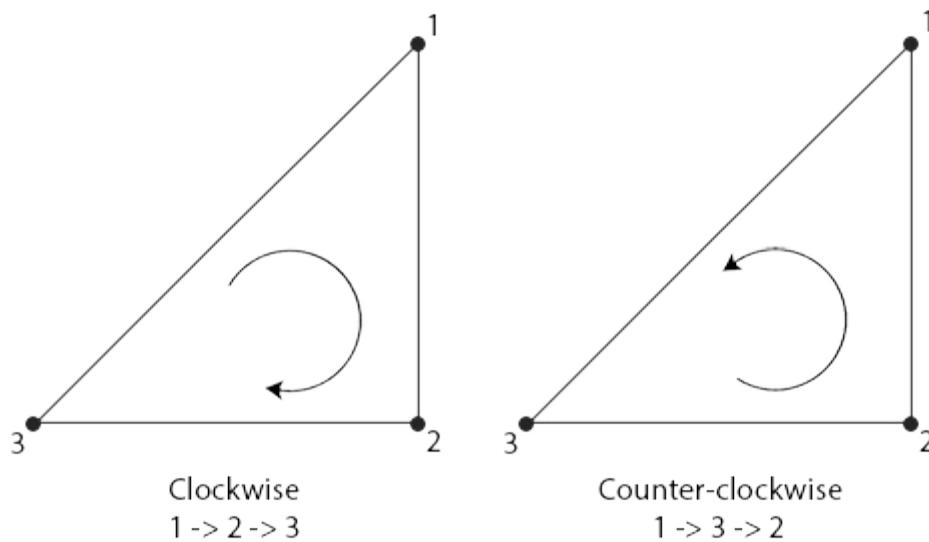
Then you need to specify which faces you want culled, front or back, by calling:

```
void GL.CullFace(CullFaceMode);
```

`CullFaceMode` can be `CullFaceMode.Front`, `CullFaceMode.Back` OR `CullFaceMode.FrontAndBack`. Calling `CullFaceMode.FrontAndBack` will cause polygons to not render at all, this isn't particularly useful. The default value is `Back`.

After you've set your cullface you have to tell openGL what your **Polygon Winding** is. What is winding? Winding is the order in which you define the vertices of your polygons. Yes, the order you define them in matters!

Take this image for example:



For simplicity, let's assume we have three vertices whose X Y and Z coordinates are all 0 1 and 0 respectively. These are the vertices:

```
(0, 0, 0)
(1, 1, 1)
(2, 2, 2)
```

To render them clockwise (**CW**):

```
GL.Begin(PrimitiveType.Polygon);
    GL.Vertex3(0,0,0);
    GL.Vertex3(1,1,1);
    GL.Vertex3(2,2,2);
GL.End();
```

To render them counter-clockwise (**CCW**):

```
GL.Begin(PrimitiveType.Polygon);
    GL.Vertex3(0,0,0);
    GL.Vertex3(2,2,2);
    GL.Vertex3(1,1,1);
GL.End();
```

Winding **directly** affects polygon facing! If a polygon is front facing when wound CW, it will be back facing when wound CCW. OpenGL uses winding information to determine which polygons are front facing and which polygons are back facing.

By default OpenGL treats polygons with counter-clockwise (CCW) ordering as front-facing polygons. This can be changed with the following function:

```
void GL.FrontFace(FrontFaceDirection);
```

Try it! Define a polygon with 3 points in a counter-clockwise fashion. Enable culling. This will produce a triangle. Run your program and it will show up on screen. Now add this line of code before running your program:

```
GL.FrontFace(FrontFaceDirection.Cw);
```

And the triangle disappears! This is because you now set front facing polygons to be CW, and back facing ones to be CCW, the opposite of the default behaviour.

By default culling is disabled. In games this is a bad thing as rendering gets expensive. One of the first things most games do in `Initialize` is to enable culling and set it to cull out back faces. While the default CCW front face is great, we often also explicitly set it to CCW so that future programmers know how the system works by reading the code.

Anti-Aliasing polygons

As with points and lines you can choose to anti-alias polygons. You control polygon antialiasing by passing `EnableCap.PolygonSmooth` to `GL.Enable` or `GL.Disable`. You can check if anti aliasing is enabled by passing `EnableCap.PolygonSmooth` to `GL.IsEnabled`. As you might expect it is disabled by default. Here is an example of how to enable it:

```
if (!GL.IsEnabled(EnableCap.PolygonSmooth)) {
    GL.Enable(EnableCap.PolygonSmooth);
}
```

On Your Own

On your own try to:

- Disable polygon anti-aliasing
- Draw a triangle on the left side of the screen
- Enable polygon anti-aliasing
- Draw a triangle on the right side of the screen

Send me a screenshot of the resulting window on skype (don't upload it anywhere, i don't want compression artifacts in this). I want to see what the difference looks like on your computer.

Triangles

Triangles are generally the preferred polygon to be used in video games. There are several reasons for this:

- Triangles are always coplanar.
 - Because 3 points define a plane
- A triangle is always convex
 - That means it does not fold in on its-self
 - Look up what a convex & concave polygon looks like if this confuses you
- A triangle will never cross over its-self
- A triangle will never have a hole in it

If you try to render a polygon that violates any of the above bullet points, the results will be unpredictable. Because any polygon can be broken down into a set of triangles (a process called triangulation), and rendering triangles is guaranteed to not have undefined behaviour, everything rendered in a video game is a triangle.



Most 3D artists prefer to model with quads. Because of this every 3D modelling program has an option to triangulate models and export only triangles. I wasn't kidding, every 3D model that gets rendered on screen is broken down into triangles!

Drawing a 3D triangle isn't much more difficult than drawing a point, or a line. You just need to pass `PrimitiveType.Triangles` to `GL.Begin` and provide 3 vertices. Every 3 vertices will make one triangle!

```
// Draw a full screen triangle
GL.Begin(PrimitiveType.Triangles);
    GL.Vertex3(-1, -1, 0);
    GL.Vertex3(1, -1, 0);
    GL.Vertex3(0, 1, 0);
GL.End();
```

Just like with points and lines, you can draw multiple triangles at one time. OpenGL treats every vertex triple as a separate triangle. If the number of vertices isn't a multiple of 3, the extra vertices are discarded.

Degenerates

A triangle with an area of 0 is essentially a point or a line, for example:

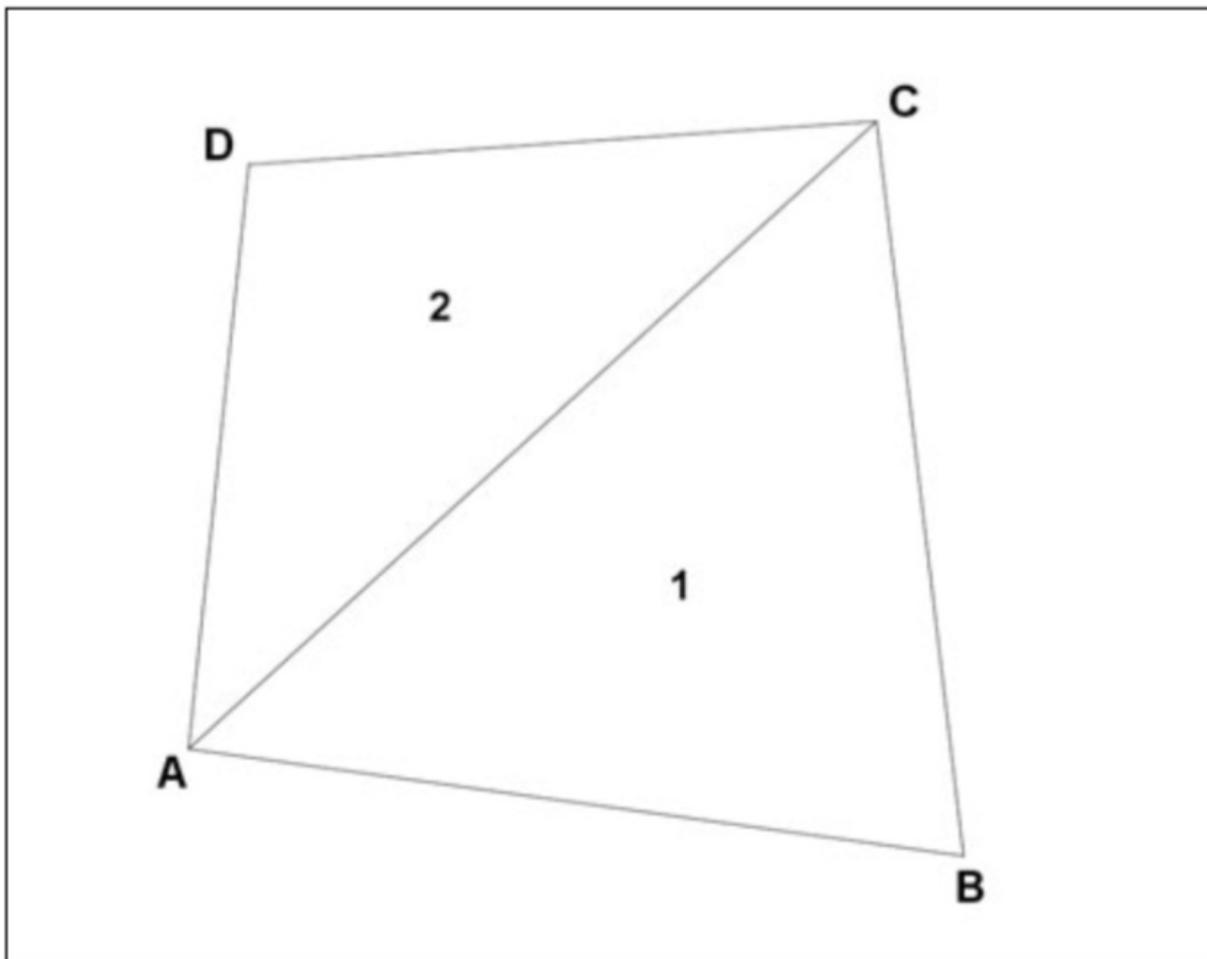
```
(1, 0, 0)
(1, 0, 0)
(0, 1, 0)
```

The above triangle is a line, it has no height! This **WILL NOT RENDER** as a line. Degenerate triangles are rejected and never rendered. (This behaviour is often exploited when rendering large terrains or mountains)

Triangle Variations

OpenGL also supports a couple of primitives related to triangles that might save you some keystrokes. In the olden days they used to save performance too, but on modern cards (Graphics cards created after 2003) there is no performance difference.

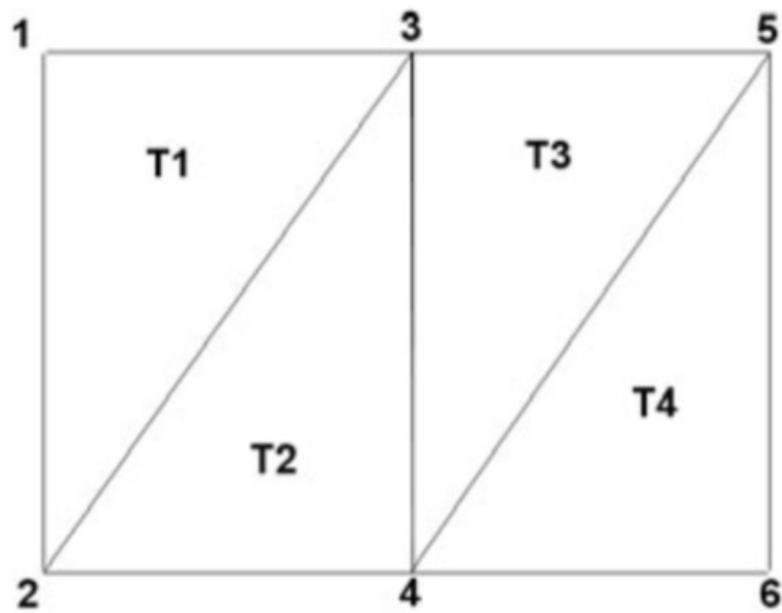
Lets assume you use two triangles to render a distorted quad:



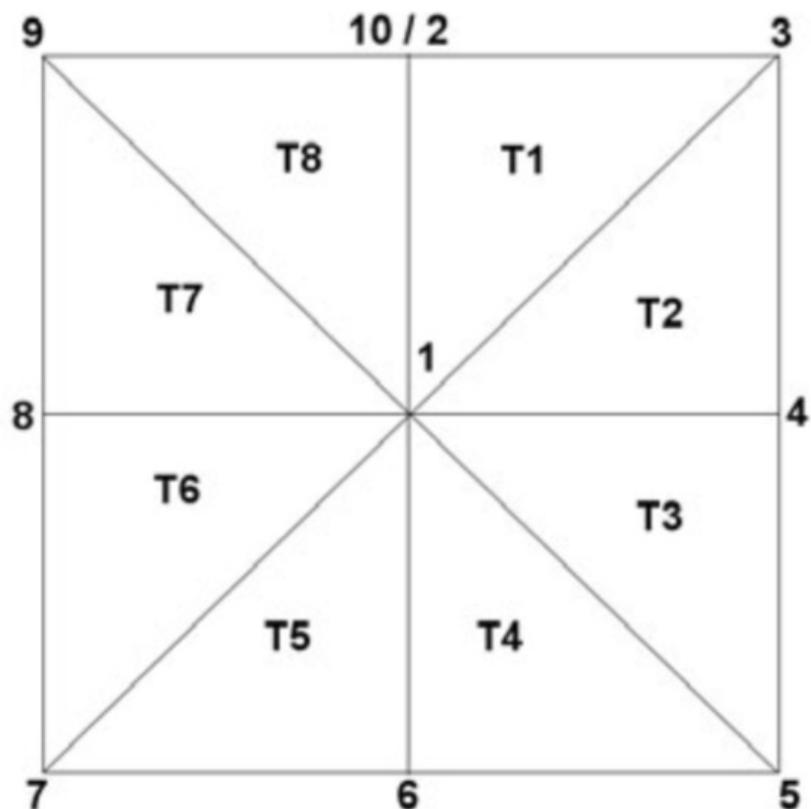
Here you have two connected triangles with vertices **A** and **C** in common. If you render them with `PrimitiveType.Triangles` you will need to define a total of 6 vertices. This means you will send vertices **A** and **C** through the rendering pipeline twice!

One way to avoid this is to use a **triangle strip**. When calling `GL.Begin` pass `PrimitiveType.TriangleStrip` for its argument. OpenGL will **draw the first 3 vertices as a triangle**, after that it will take **every vertex specified and combine it with the previous two vertices** to form another triangle. This means that after the first triangle, every additional triangle will only need one vertex for its data.

Here is a visual example of a triangle strip:



Another variation on triangles is the **triangle fan**. You can visualize them as a series of triangles around a central vertex. You draw fans by passing `PrimitiveType.TriangleFan` to `GL.Begin`. The first vertex specified is the center vertex. Every following two vertices make a triangle with the center vertex. Here is what that looks like:



Fans don't offer as much savings as strips, but they still offer some savings.

A history lesson

From about 2000 to 2006 triangle strips provided a LOT of savings in BOTH memory and performance. We simply didn't have enough ram and horsepower on the GPU to support large models made up of too many triangles.

Many programmers capitalized on this, and wrote software that would take a triangulated model and create a model of triangle strips from it. These software were collectively called strippers, because they made strips.

As you can imagine strippers were not very efficient with large complex models. As graphics cards grew more powerful, these software became more and more outdated.

On a fairly new graphics card (2006+) it is faster to draw a million regular triangles than it is to draw the same number of triangles using a strip. This is because graphics cards manufacturers caught on that rendering triangles fast is important, so they built special circuitry to handle the situation.

And just like that modern games grew up. We have stopped stripping and are instead just rendering large sets of triangles.

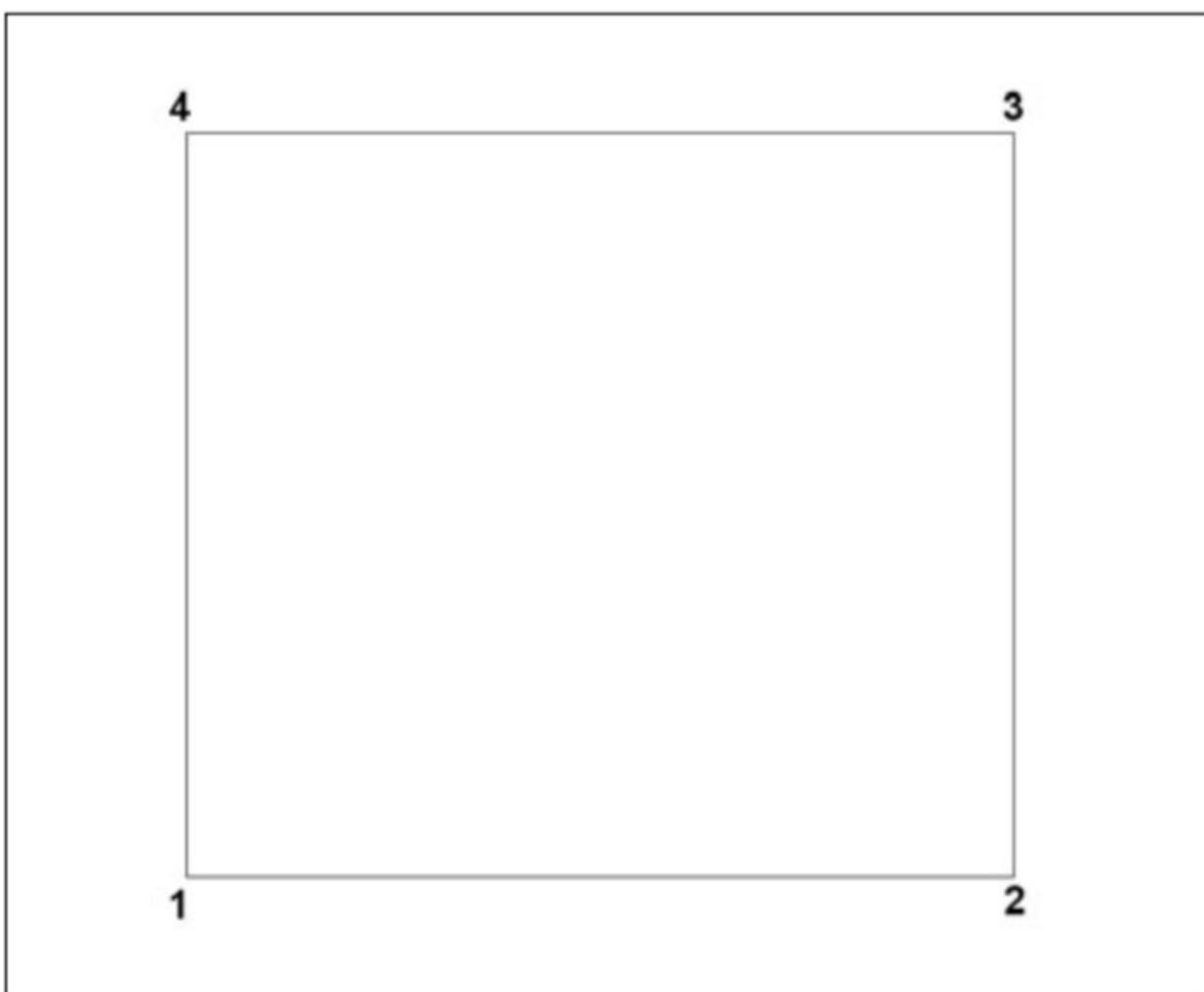
Quadrilaterals

Quadrilaterals, or quads, are four-sided polygons that can be convenient when you want to draw a square or a rectangle. You create them by passing `PrimitiveType.Quads` to `GL.Begin`, then supplying 4 vertices. Like triangles you can draw multiple quads within the same `GL.Begin / GL.End` block.

OpenGL provides Quad strips as a means of improving the rendering of multiple larger quads, but you will use this feature less than you will use triangle strips. They are kind of useless.

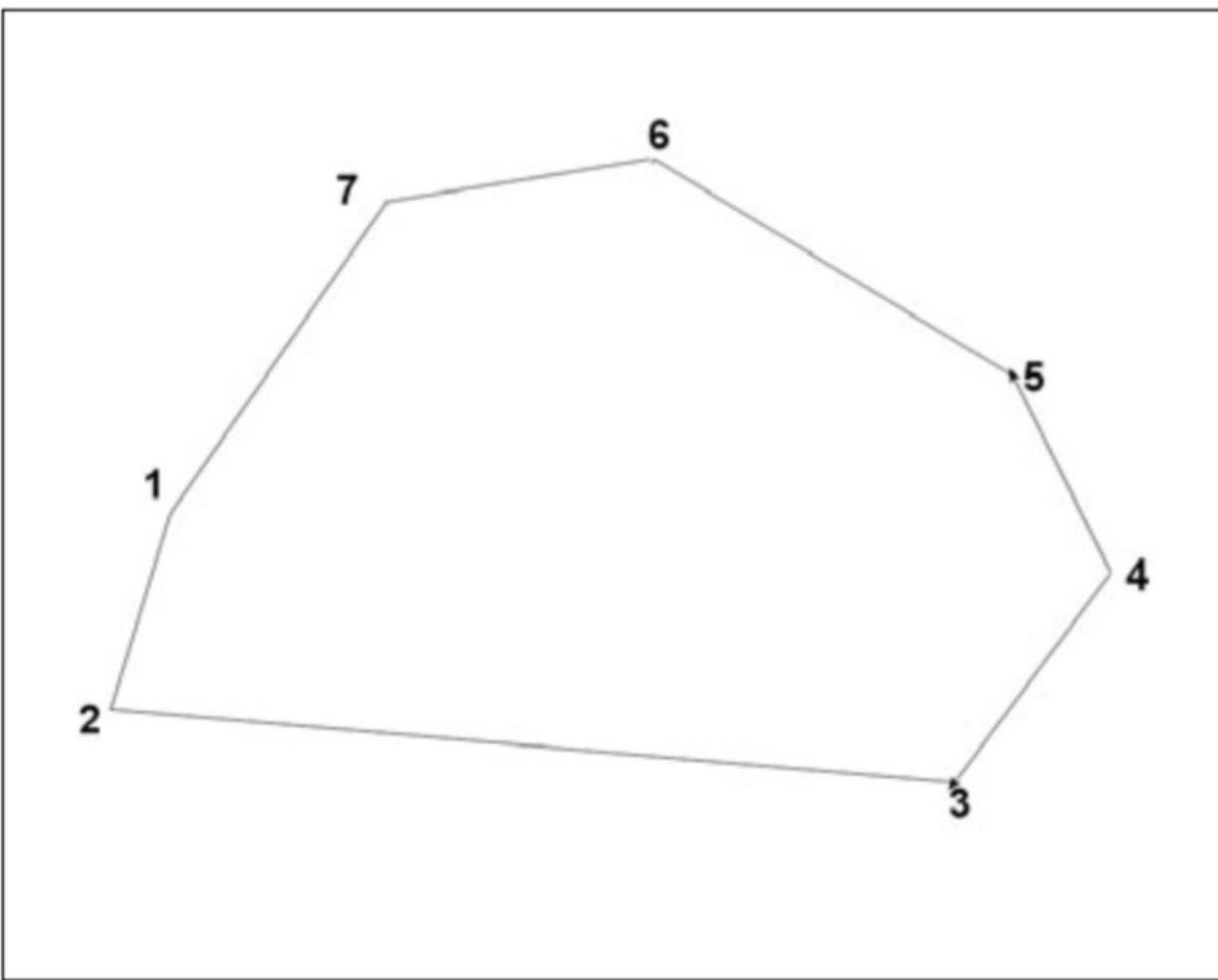
When rendering a quad strip, the first 4 vertices specify the first quad, after that every two vertices will be used with the previous two vertices to create new quads.

This is the order you should define a quad in:



Polygons

As discussed earlier OpenGL also supports rendering polygons with an arbitrary number of sides. In such cases, only one polygon can be drawn inside a `GL.Begin / GL.End` block. To render a polygon pass `PrimitiveType.Polygon` (Note, it's singular) to `GL.Begin`. Once `GL.End` is reached, the last vertex is connected to the first. If less than 3 vertices are provided, the polygon is considered degenerate and nothing will render. Here is an example of what an arbitrary polygon can look like:



Sample

Lets put everything we've learned into one tidy little sample. Follow along with the following code, it goes in your render function:

```
// 0.125 is because 4 * 0.125 = 0.5.
// The width of the screen is 2 (from -1 to +1)
// 1/4 of 2 is 0.5, the same as 4 * 0.125
float scale = 0.125f;

// Draw points
float xOffset = -0.9f;
float yOffset = 0.9f;
GLPointSize(4.0f);
GL.Begin(PrimitiveType.Points);
for (int x = 0; x < 4; ++x) {
    for (int y = 0; y < 4; ++y) {
        // We use 0.9 instead of 1 to go close to the edge, not all the way
        // Why are we multiplying 4 by 0.125 (scale)?
        // Because X and Y both go from 0 to 3, a total of 4 units
        GL.Vertex3(xOffset + x * scale, yOffset - y * scale, 0);
    }
}
GL.End();

// Draw Triangles (note these are back facing!)
xOffset = -0.25f;
yOffset = 0.9f;
GL.Begin(PrimitiveType.Triangles);
for (int x = 0; x < 4; ++x) {
```

```

        for (int y = 0; y < 4; ++y) {
            GL.Vertex3(xOffset + x * scale,      yOffset - y * scale,      0);
            GL.Vertex3(xOffset + (x + 1) * scale, yOffset - y * scale,      0);
            GL.Vertex3(xOffset + x * scale,      yOffset - (y + 1.0f) * scale, 0);
        }
    }
    GL.End();

    // Set polygons to render as lines
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);

    // Draw quads
    xOffset = 0.9f;
    yOffset = 0.9f;
    GL.Begin(PrimitiveType.Quads);
    for (int x = 0; x < 4; ++x) {
        for (int y = 0; y < 4; ++y) {
            GL.Vertex3(xOffset - x * scale,      yOffset - y * scale,      0);
            GL.Vertex3(xOffset - (x + 1) * scale, yOffset - y * scale,      0);
            GL.Vertex3(xOffset - (x + 1) * scale, yOffset - (y + 1) * scale, 0);
            GL.Vertex3(xOffset - x * scale,      yOffset - (y + 1) * scale, 0);
        }
    }
    GL.End();

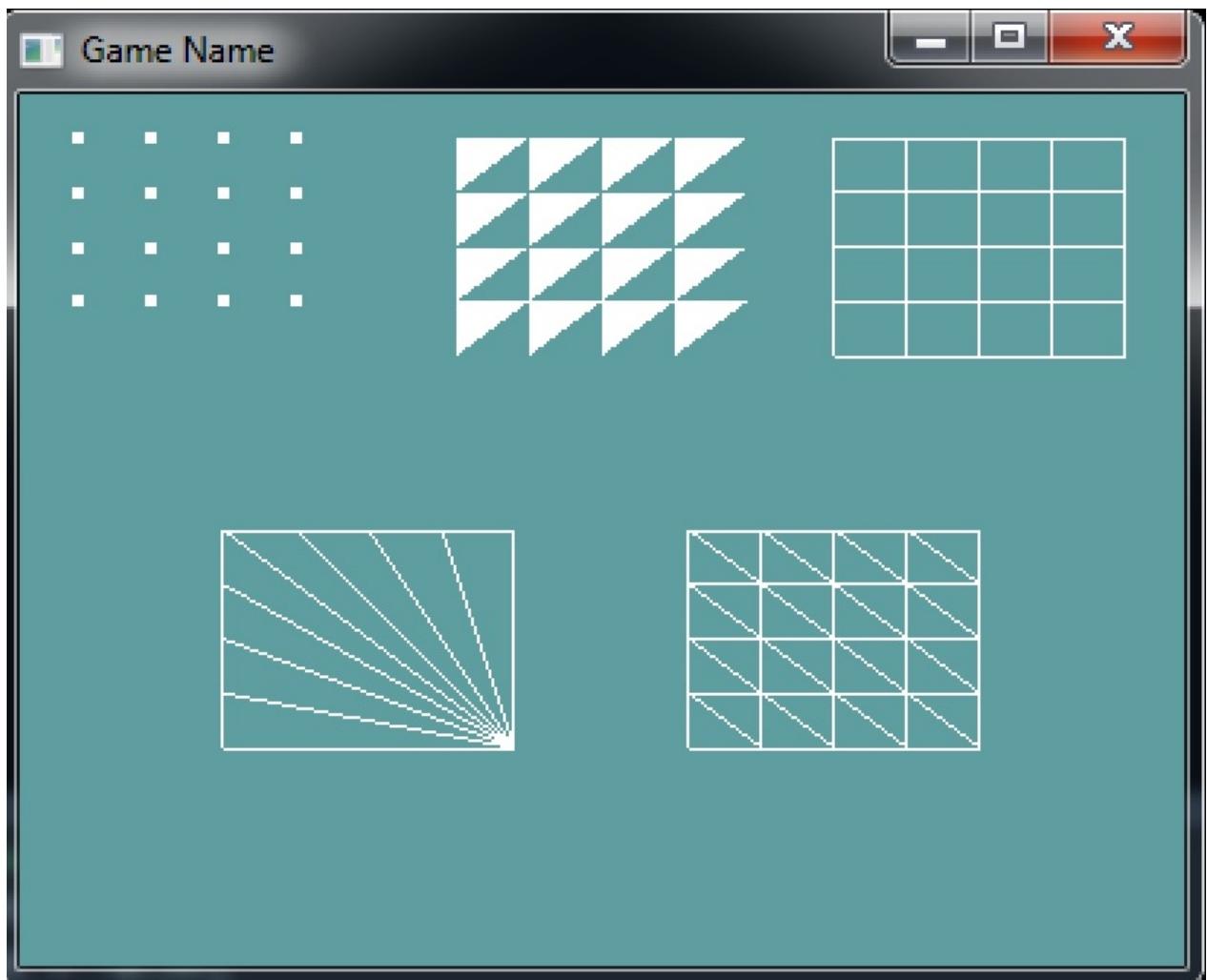
    // Draw triangle strip
    xOffset = 0.15f;
    yOffset = -0.5f;
    for (int x = 0; x < 4; ++x) {
        GL.Begin(PrimitiveType.TriangleStrip);
        for (int y = 0; y < 4; ++y) {
            GL.Vertex3(xOffset + x * scale,      yOffset + y * scale,      0);
            GL.Vertex3(xOffset + (x + 1) * scale, yOffset + y * scale,      0);
            GL.Vertex3(xOffset + x * scale,      yOffset + (y + 1) * scale, 0);
            GL.Vertex3(xOffset + (x + 1) * scale, yOffset + (y + 1) * scale, 0);
        }
    }
    GL.End();
}

// Draw triangle fan (note this is back facing!)
xOffset = -0.15f;
yOffset = -0.5f;
GL.Begin(PrimitiveType.TriangleFan);
// Center of thefan
GL.Vertex3(xOffset - 0.0f, yOffset + 0.0f, 0.0f);
// Bottom side
for (int x = 5; x > 0; --x) {
    GL.Vertex3(xOffset - (x - 1) * scale, yOffset + 4 * scale, 0);
}
// Right side
for (int y = 5; y > 0; --y) {
    GL.Vertex3(xOffset - 4 * scale, yOffset + (y - 1) * scale, 0);
}
GL.End();

// Reset polygon modes for next render
GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);

```

After it's all in there, the final screen should look like this:



Attributes

Earlier in this chapter you saw how to set up and query individual states from OpenGL. Now lets take a look at how to save and restore the values of a set of related state variables with a single command.

An *attribute group* is a set of related state variables that OpenGL classifies into a group. For example, the line group consists of all the line drawing attributes, such as width, stipple pattern and anti-aliasing. The polygon group consists of the same set of attributes, just for polygons. You can save and restore these values by using the `GL.PushAttrib` and `GL.PopAttrib` functions. These are the signatures

```
void GL.PushAttrib(AttribMask);
void GL.PopAttrib();
```

`PushAttrib` saves all of the attributes for an attribute group specified by its argument (an `AttribMask` enum). To be more specific, it pushes all the attributes onto a stack known as the *attribute stack*. `PopAttrib` restores the previous attributes by popping the top of the *attribute stack*.

The argument to `PushAttrib` is a bit-mask. The arguments you pass in can be combined with a bitwise **OR** (`|`) The following `AttribMask` values are valid:

- **AllAttribBits** All opengl state variables in all attribute groups
- **EnableBit** Enable state variables
- **FogBit** Fog state variables
- **LightingBit** Lighting state variables
- **LineBit** Line state variables
- **PointBit** Point state variables
- **PolygonBit** Polygon state variables
- **TextureBit** Texturing state variables

It's common to set a default state for everything in your initialize function. Then, in your render function you push and pop attrs in order to set custom states and restore the defaults. For example, something like this

```
void RenderModel() {
    // Push bits as we might go into wireframe mode, or into untextured mode.
    // We don't know for sure, it's configured with a variable.
    // Some models might be wireframe, others might not!
    GL.PushAttrib(AttribMask.PolygonBit | AttribMask.TextureBit);

    if (renderWireframe) {
        GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);
    }

    // This function might change texturing parameters!
    DoTexturing();

    GL.Begin(PrimitiveType.Triangles);
    foreach(Triangle t in triangles) {
        GL.Vertex3(t.v1.x, t.v1.y, t.v1.z);
        GL.Vertex3(t.v2.x, t.v2.y, t.v2.z);
        GL.Vertex3(t.v3.x, t.v3.y, t.v3.z);
    }
    GL.End();

    // Restore polygon and texture render modes to default!
    GL.PopAttrib();
}
```


Transformations And Matrices

Now it's time to take a short break from learning how to *create* 3D Objects in the world and focus on learning hot to *move* the objects around in the world. This is vital to generating realistic 3D worlds! Without this the 3D scenes you create would be static and non-interactive.

OpenGL makes it easy for the programmer to move objects around trough the use of various *coordinate transformations*, discussed in this chapter. You will also take a look at how to use your own matrices with OpenGL, which provides you with the power to manipulate objects in many different ways.

In this chapter you will learn about

- The basics of coordinate transformations
- The camera and viewing transformations
- OpenGL matrices and matrix stacks
- Projections
- Using your own matrices with OpenGL

If you need a refresher on the matrix math implementation details, [this](#) video appears to be useful.

I tired to follow the book, but it was AWFUL, it confused even me. So this chapter is a take from another book, "OpenGL Distilled" along with articles videos and my insights.

Understanding Coordinate Transformations

The OpenGL transformation pipeline transforms application vertices into window coordinates, where they can be rasterized. Like all 3D graphics systems, OpenGL uses linear algebra; it treats vertices as vectors and transforms them by using vector-matrix multiplication. The transformation process is called a **pipeline** because geometry passes through several coordinate systems on the way to window space. Each coordinate system serves a purpose for one or more OpenGL features.

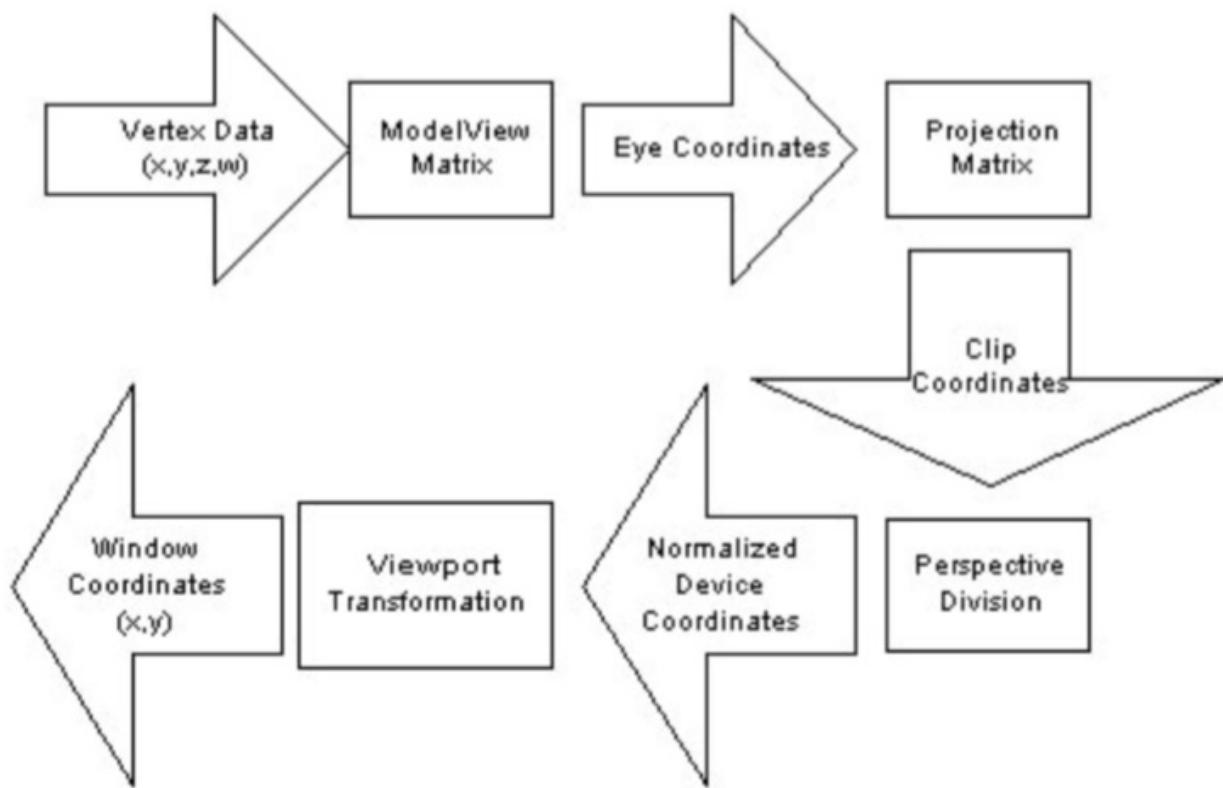
For each stage of the transformation pipeline, this section describes the characteristics of that coordinate system, what OpenGL operations are performed there, and how to construct and control transformations to the next coordinate system in the pipeline.

It's important to understand that when we render an object, we don't move it. We put a vertex through a transformation pipeline, and a new, transformed vertex comes out. We render this transformed vertex, then throw it away. The original vertices of the models will never be changed.

When rendering 3D scenes, vertices pass through 4 types of transformations before they are rendered onto the screen:

- **Modeling Transform** The modeling transformation moves objects around the scene and moves objects from local coordinates into world coordinates
- **Viewing Transform** The viewing transformation specifies the location of the camera and moves objects from world coordinates into *eye coordinates* (camera coordinates)
- **Projection Transform** The projection transformation defines the viewing volume and clipping planes, it maps objects from eye coordinates to clip coordinates (-1 to +1 on all axis, clip coordinates are also called NDC)
- **Viewport transform** The viewport transformation maps the clip coordinates (NDC) into the two-dimensional view port (the window on your screen)

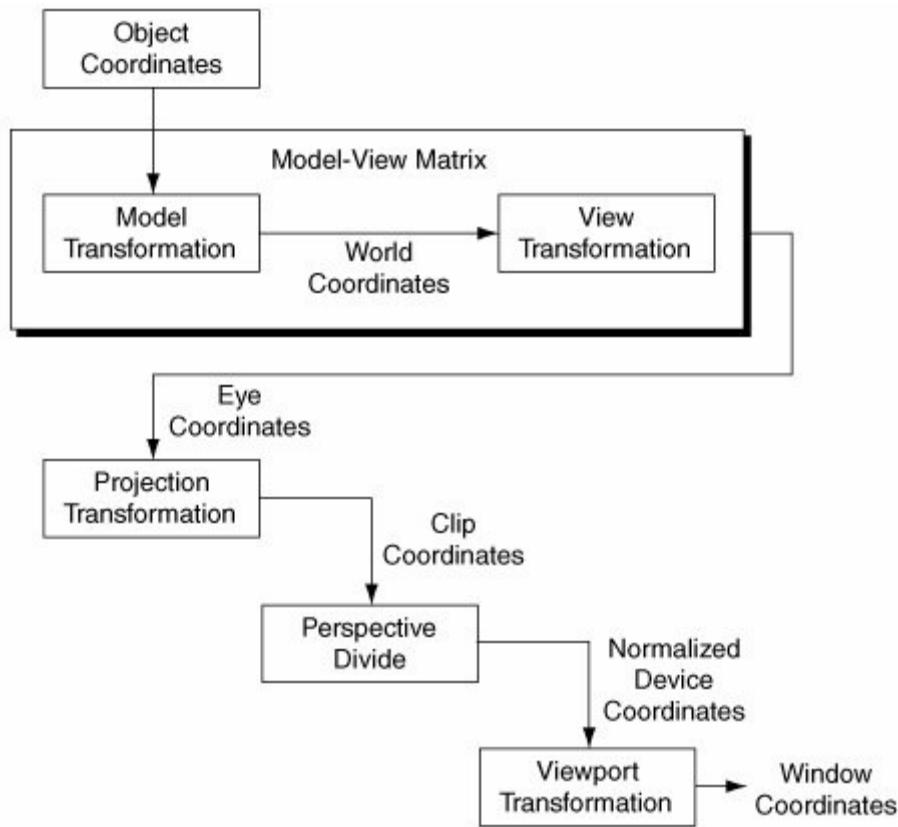
A summary of the transformation pipeline:



While these four transformations are standard in 3D graphics, OpenGL combines the model and view transforms into a single **modelview** transformation. The viewport transform (also known as w divide) is done automatically by OpenGL.

If you look at the above diagram, there are only two matrices in the pipeline. These are the matrices that the state machine lets you specify, the **ModelView** matrix and the **Projection** matrix.

A closer look at the pipeline



This is the entire pipeline in a bit easier to follow fashion. Let's take a closer look at what each section does.

Object Coordinates

Applications specify vertices in object coordinates. The definition of object coordinates is entirely up to the application. Some applications render scenes that are composed of many models, each created and specified in its own individual coordinate system.

What this means is that each model is centered around its own origin. When you export a 3D model from blender, that model was created around (0, 0, 0) and is said to be in model space.

When you load in an object, it is in Object space. It will be in the middle of your scene.

The Model Transformation

The model transformation transforms geometry from object to world coordinates. Applications store both the model and view transforms in a single matrix called the **modelView matrix**.

Once you apply the model transform, the object is now in world space. That is, X, Y, Z is (potentially) no longer located at the origin of the world.

The modeling transformation allows you to position and orient a model by moving, rotating and scaling it. You can perform these operations one at a time or as a combination of events.

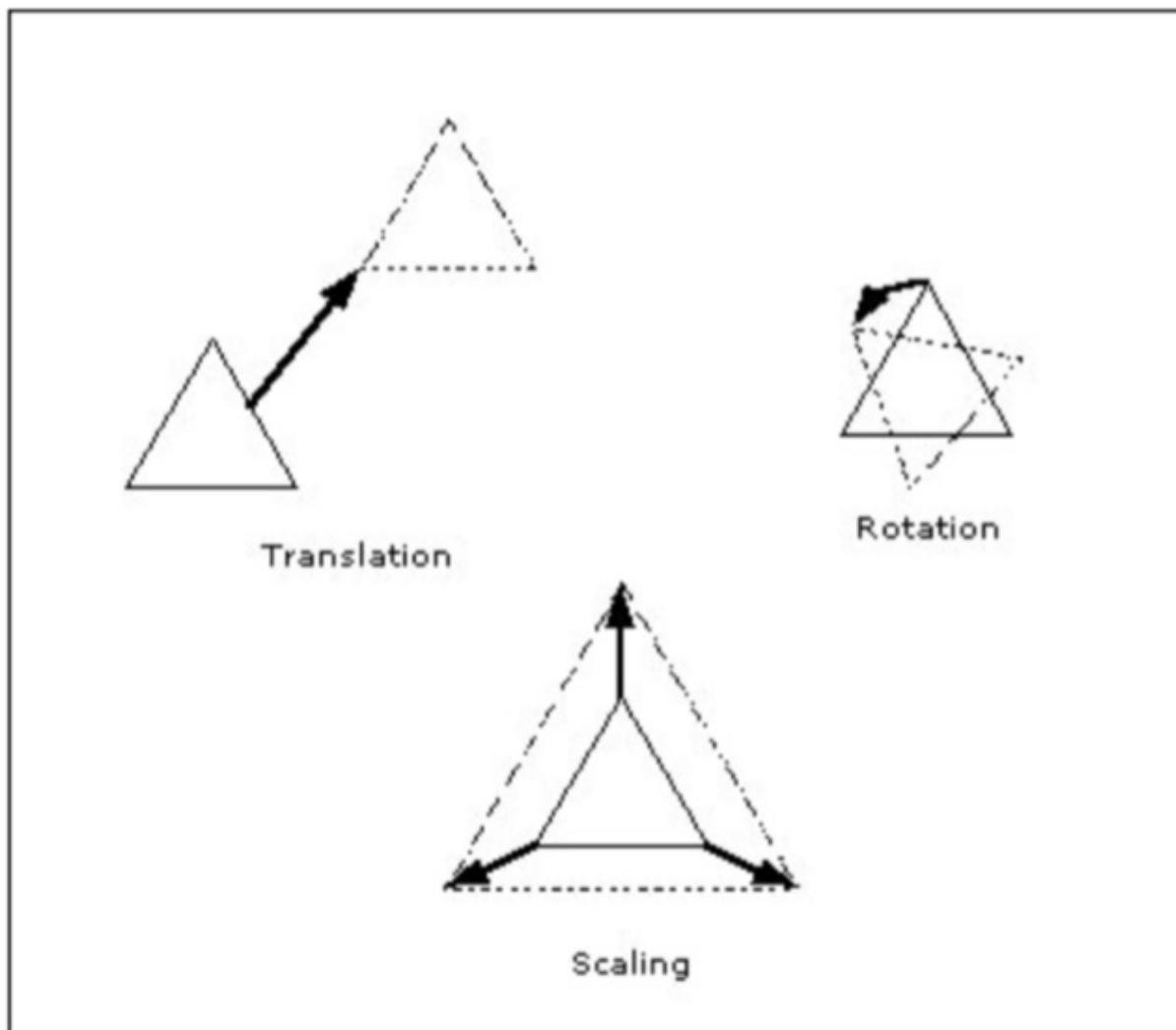
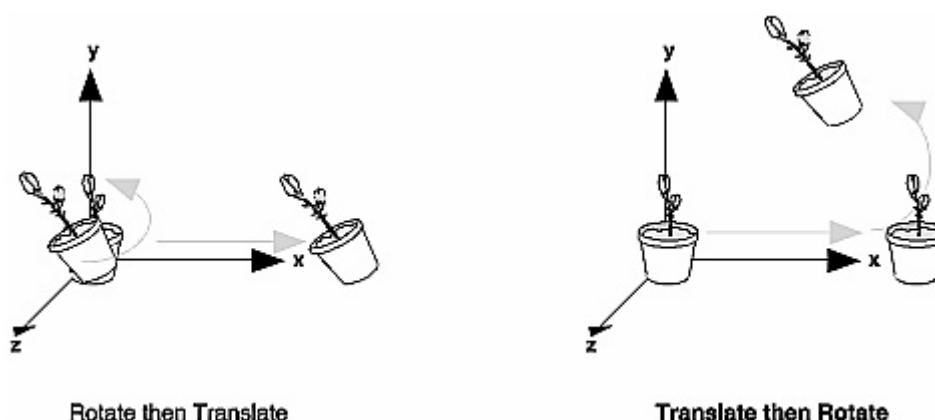


Figure 4.3 The three modeling transformations.

The order in which you specify modeling transformations is very important to the final rendering of your scene.
(remember, order of matrix multiplication matters)



Rotating an object first, then translating it will give a very different result than translating an object and then rotating it. So, what's the proper order to transform your primitives? Well that depends on the result you want to get. But in general, multiplication order is accepted as:

- First Scale
- Second Rotate
- Lastly Translate

World Coordinates

The world-coordinate system is an application construct that exists between the object and eye coordinate systems. If we split the **ModelView** matrix into two matrices, multiplying object space coordinates by the **Model** matrix would get them to world space.

The reason that the **Model** and **View** matrices have been collapsed into a single matrix is because OpenGL has no internal concept of "World Space". There are no rendering calculations to be done in world space, so the state machine has no idea what that actually is.

The View Transform

In precise terms, the **view transform** is the inverse transformation of the cameras world space coordinates.

This may sound a bit odd. Everything goes through the render pipeline, even the camera. So, what happens when you multiply the cameras world coordinates by the view matrix (The inverse of the cameras world coordinates)? You get an identity matrix!

This means that the camera will be positioned at the origin of the world, looking down the negative Z axis. Every object will be transformed relative to this orientation. Do you know what this implies?

This implies that when we render a 3D world, as far as the graphics pipeline is concerned the camera does not move. It stays stationary at origin, instead the world moves around the camera!

Eye Coordinates

OpenGL eye coordinates are defined as follows:

- The viewpoint is at the origin
- The view direction is down the negative Z axis
- Positive Y is up
- Positive X is to the right

For geometry to be visible in a standard perspective projection it must lie somewhere along the negative Z axis.

When you multiply an object by the **ModelView** matrix it ends up in eye coordinates, which meets all 4 of the above requirements. This is why we don't move the camera, rather the world around the camera.

Think of eye coordinates as "what the camera sees". This is what the eye coordinate space looks like, it is the camera multiplied by the view matrix.

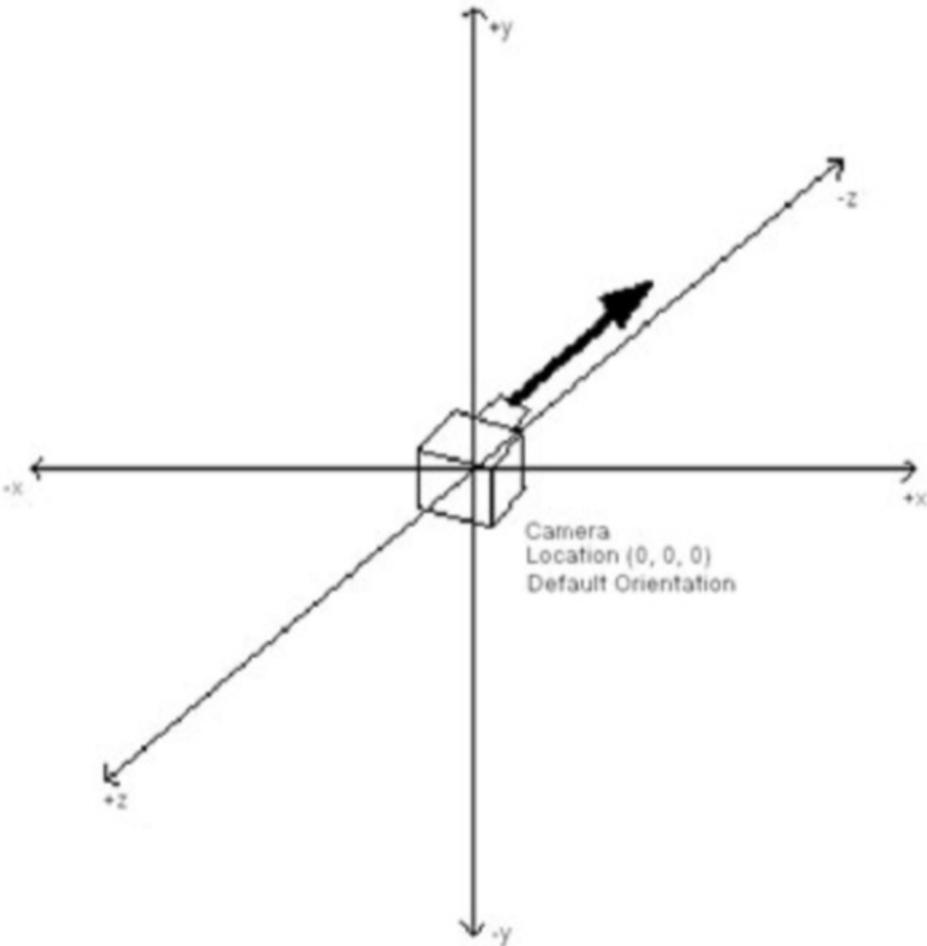


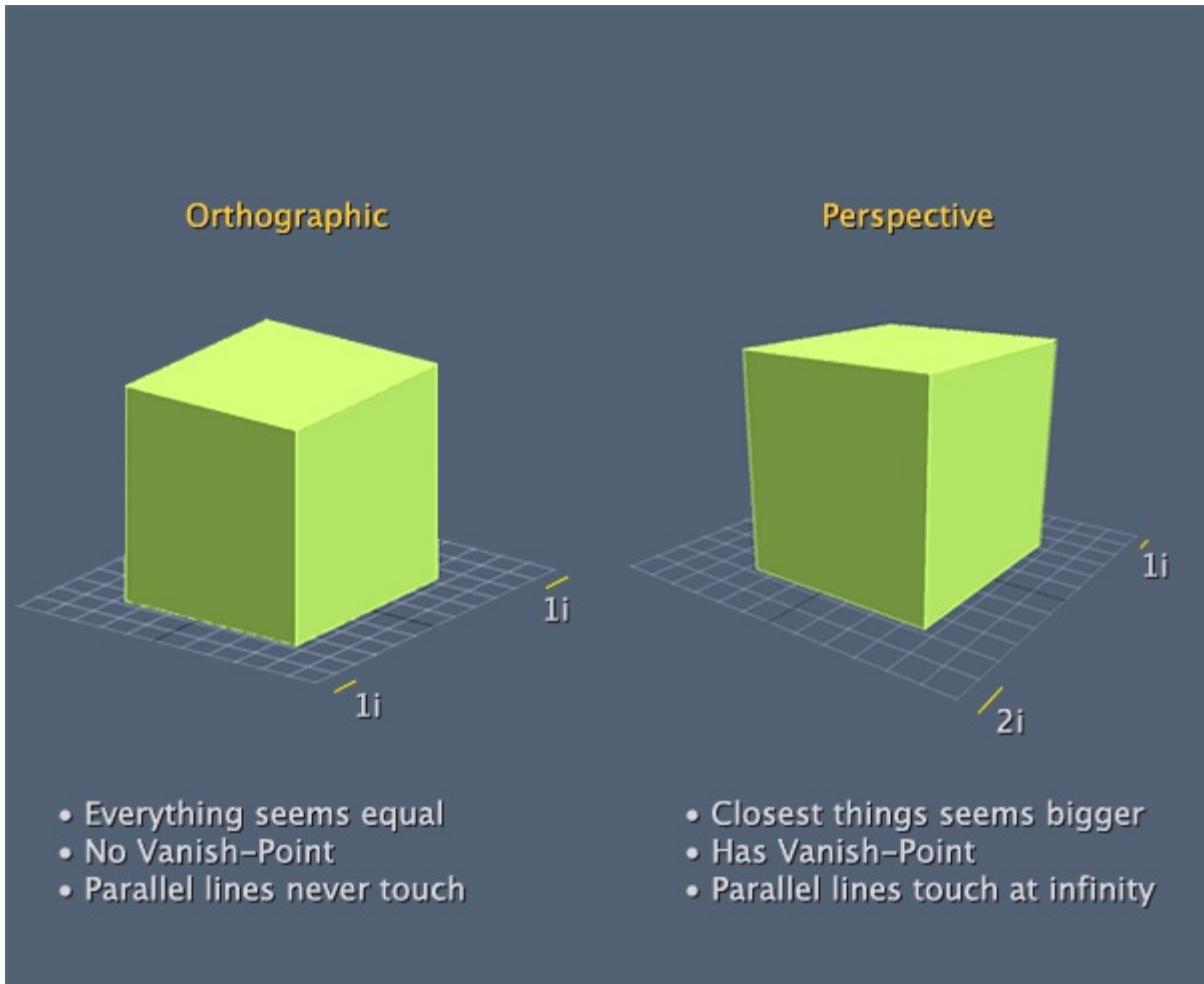
Figure 4.2 The default viewing matrix in OpenGL looks down the negative z axis.

The Projection Transformation

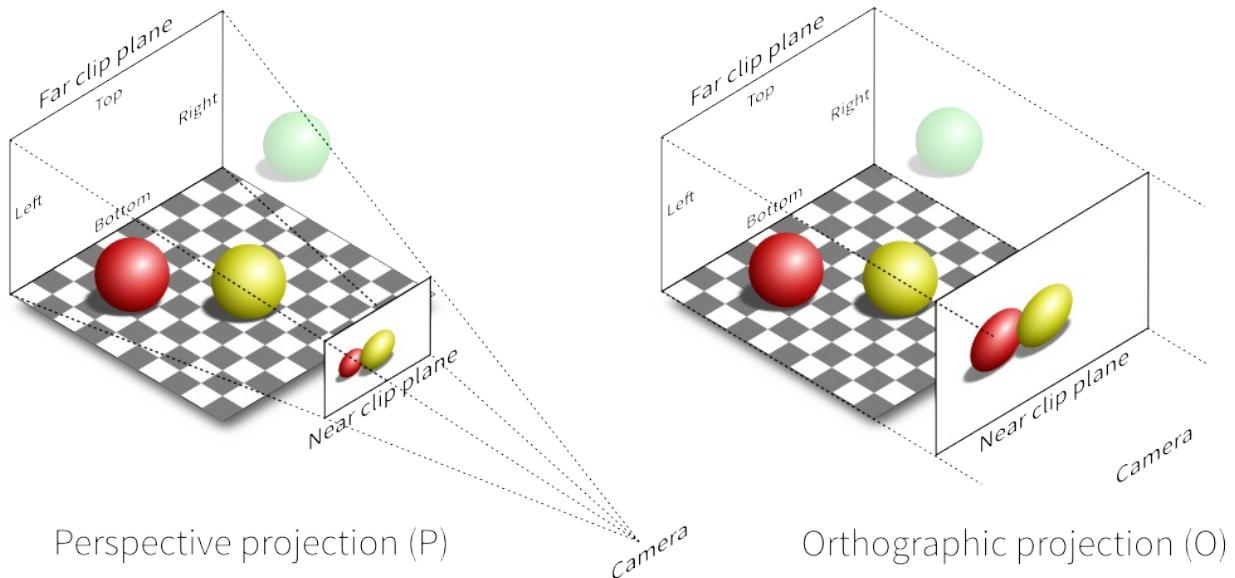
OpenGL multiplies eye coordinates by the projection matrix to produce clip coordinates. The projection matrix defines the size and shape of the view volume, and therefore determines the portion of eye coordinates that will be visible or invisible.

Typically applications create a **perspective projection** or a parallel projection. The proper name a parallel projection is **orthographic projection**

Here is an example of looking at two objects from the exact same position with the exact same orientation. The only difference is one is looking with perspective, the other is looking orthographic:



This is how the respective cameras see the world:



Clip Coordinates

OpenGL clips primitives that lie outside the view volume in clip coordinate space. A clip coordinate vertex is within the view volume if its x , y and z values are all within the $-w$ to w space.

Wow that makes no sense! Don't worry, this is one of those things OpenGL does automatically for you! You will

never have to concern yourself with the implementation of this step.

You will configure clip coordinates using the `GL.Ortho` and `GL.Frustum` methods.

Basically, OpenGL ignores any geometry outside the view volume, (see the green sphere in the above picture)

Perspective Division

OpenGL divides clip coordinate (x, y and z) values by the clip-coordinate w value to produce normalized device coordinates (**NDC**). As with clipping, OpenGL will do this automatically for you (provided you have a projection matrix in place). If you have an orthographics projection matrix, nothing really happens here.

For perspective projections, the perspective division step effectively shrinks distant geometry and expands near geometry.

Remember, **NDC** space goes from -1 to +1 on all axis.

Normalized Device Coordinates

In normalized device coordinate space all vertex values lie within the -1 to +1 range. In previous coordinate systems, they could have been anywhere!

OpenGL performs no calculations in NDC space, it's simply a coordinate system that exists between the perspective division and the viewport transformation to window coordinates.

The Viewport Transform

The viewport transformation is the final stage of the **transformation pipeline**. It's a scale and translation that maps the -1 to +1 NDC cube into X / Y window coordinates. Technically this stage also modifies the Z value to be within an acceptable range of the Z-Buffer, which we will talk about later.

Again, this happens automatically, you have some control using the `GL.Viewport` function, which we will discuss later.

Window Coordinates

Window coordinates have their x, y *origin* in the bottom-left corner of the window. Window z values extend the Z-Buffer.

Programmers often make the mistake of assuming that window coordinates are integers (because hey, a window is rendered as pixels). This is not the case. OpenGL window coordinates are actually floating point values, vertices can exist in sub-pixel locations. This is essential for correct rasterization and effective anti-aliasing.

This is the part of the pipeline where a picture is finally rendered to your window, we are now done with the pipeline.

Transformation Pipeline Deep Dive

The problem with most books is that they glaze over the transformation pipeline. This is bad! The Transformation Pipeline is the heart of 3D rendering! I really, really need you to understand how it works!

[This article](#) does a much better job of explaining how the transformation pipeline works than I can. Bookmark it, read it and take it to heart; print it if you have to. If you are still confused at the end of this chapter, reread that article and talk to me!

More Visualization

To kind of sum up what the above article is saying, this is what's going on:

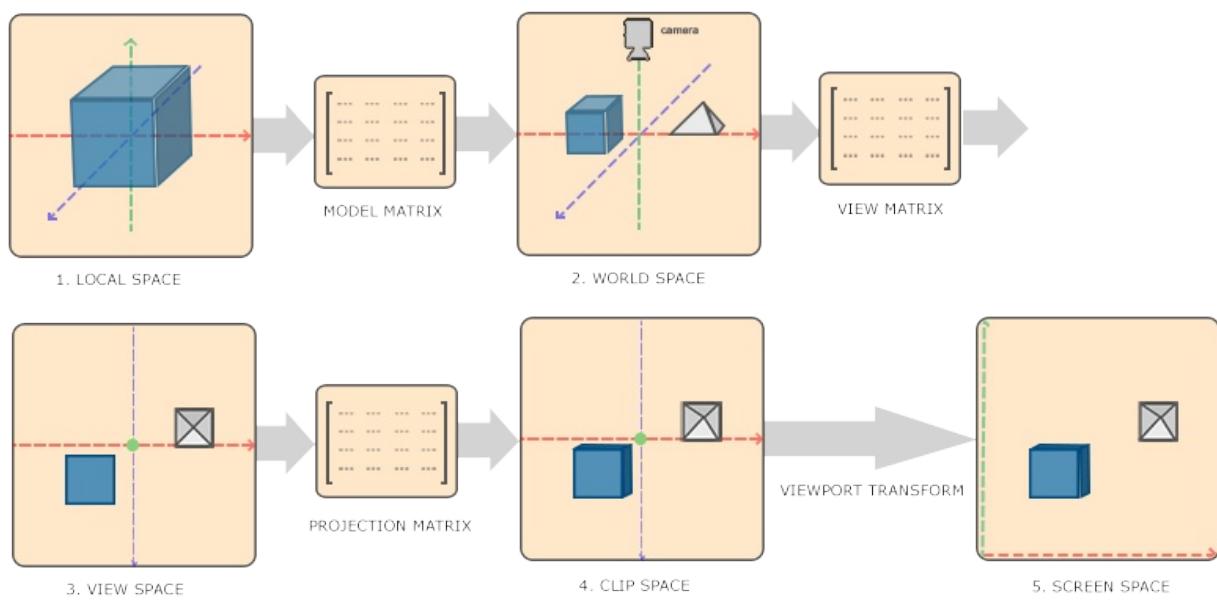


Image was taken from [this article](#), you don't have to read it, but it's there if you are curious

- Local coordinates are the coordinates of your object relative to its local origin; they're the coordinates your object begins in.
- The next step is to transform the local coordinates to world-space coordinates which are coordinates in respect of a larger world. These coordinates are relative to a global origin of the world, together with many other objects also placed relative to the world's origin.
- Next we transform the world coordinates to view-space coordinates in such a way that each coordinate is as seen from the camera or viewer's point of view.
- After the coordinates are in view space we want to project them to clip coordinates. Clip coordinates are processed to the -1.0 and 1.0 range and determine which vertices will end up on the screen.
- And lastly we transform the clip coordinates to screen coordinates in a process we call viewport transform that transforms the coordinates from -1.0 and 1.0 to the coordinate range defined by `glViewport`. The resulting coordinates are then sent to the rasterizer to turn them into fragments.

What moves

The view matrix is interesting, it basically frames the world. The view matrix is what the camera sees. So, what is the

view matrix? It's the INVERSE (this is why inverting matrices is so important!) of the cameras world position.

If you remember, multiplying anything by it's inverse will result in the identity vector. Which means, if we multiply the camera world position by the view matrix it puts the cameras world position at (0, 0, 0) or origin.

The model-view matrix is simply combining an objects world transform with the view matrix. Effectivley moving the object into a space where the camera is at origin looking down the z axis.

That's right, when you move trough a 3D world, as far as rendering is concerned the camera is not moving trough the world, the world is moving around the camera.

OpenGL and Matrices

Now that you've learned about the various transformation involved in OpenGL, lets take a look at how you actually use them. Transformations in OpenGL rely on the **matrix** for all mathematical computations. As you will soon see, OpenGL has what is called the **matrix stack**, which is useful for constructing complicated models composed of many simple objects.

Visualize Origin

Let's start with a simple scene. We're going to draw a small grid on the X-Z plane, then we are going to render the 3 basis vectors of world space. At first this isn't going to show much on screen, because without a projection matrix it will be in the middle of NDC, so we will only see two lines.

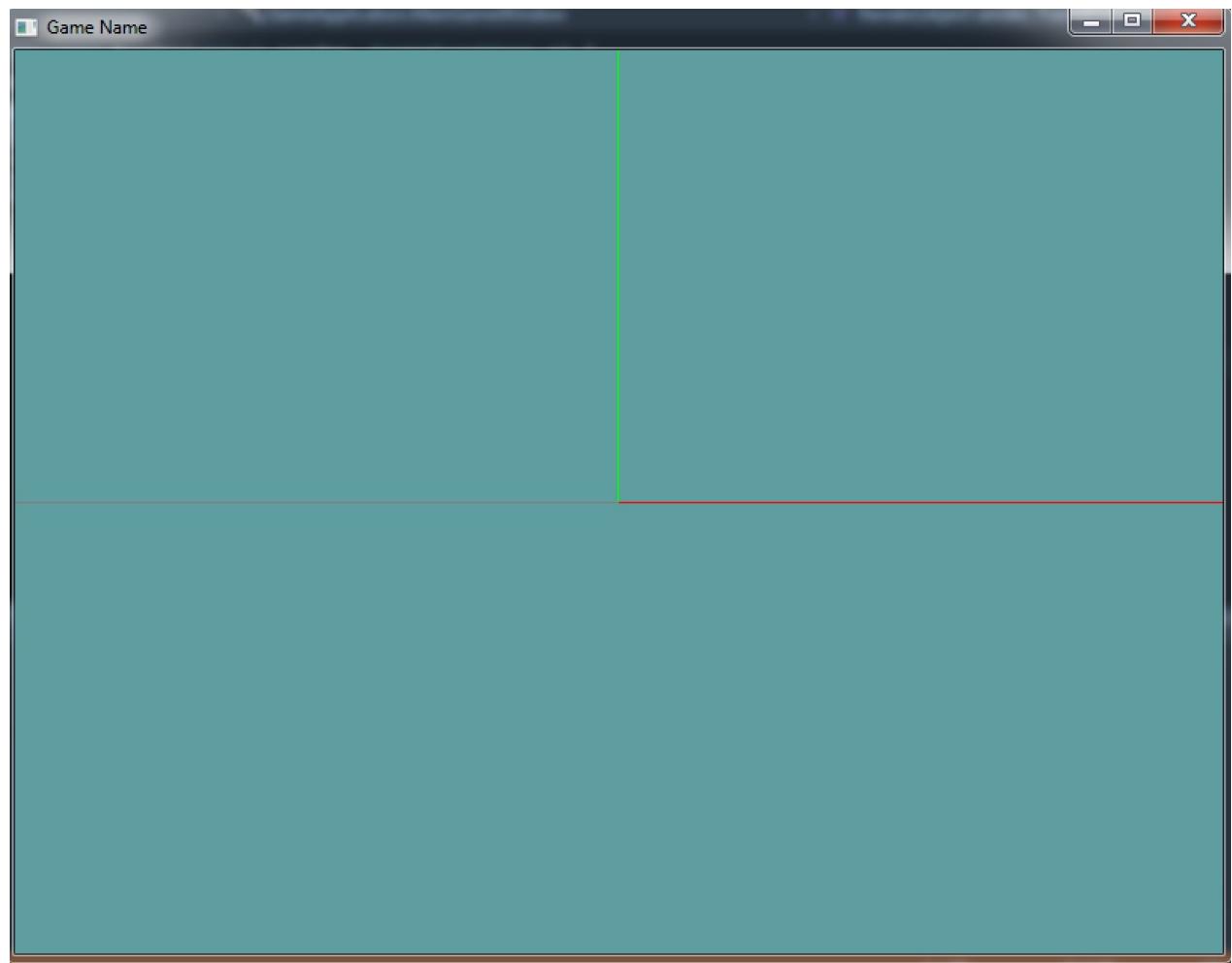
You may want to add this render code to the `MainGameWindow` class, before the scene is rendered. This is going to be a visual baseline so we can see what is going on, and it should appear under every sample we will make in this chapter.

Make a new class, lets call it `Grid`. In it, have a `Render` function. This should be the contents of that function:

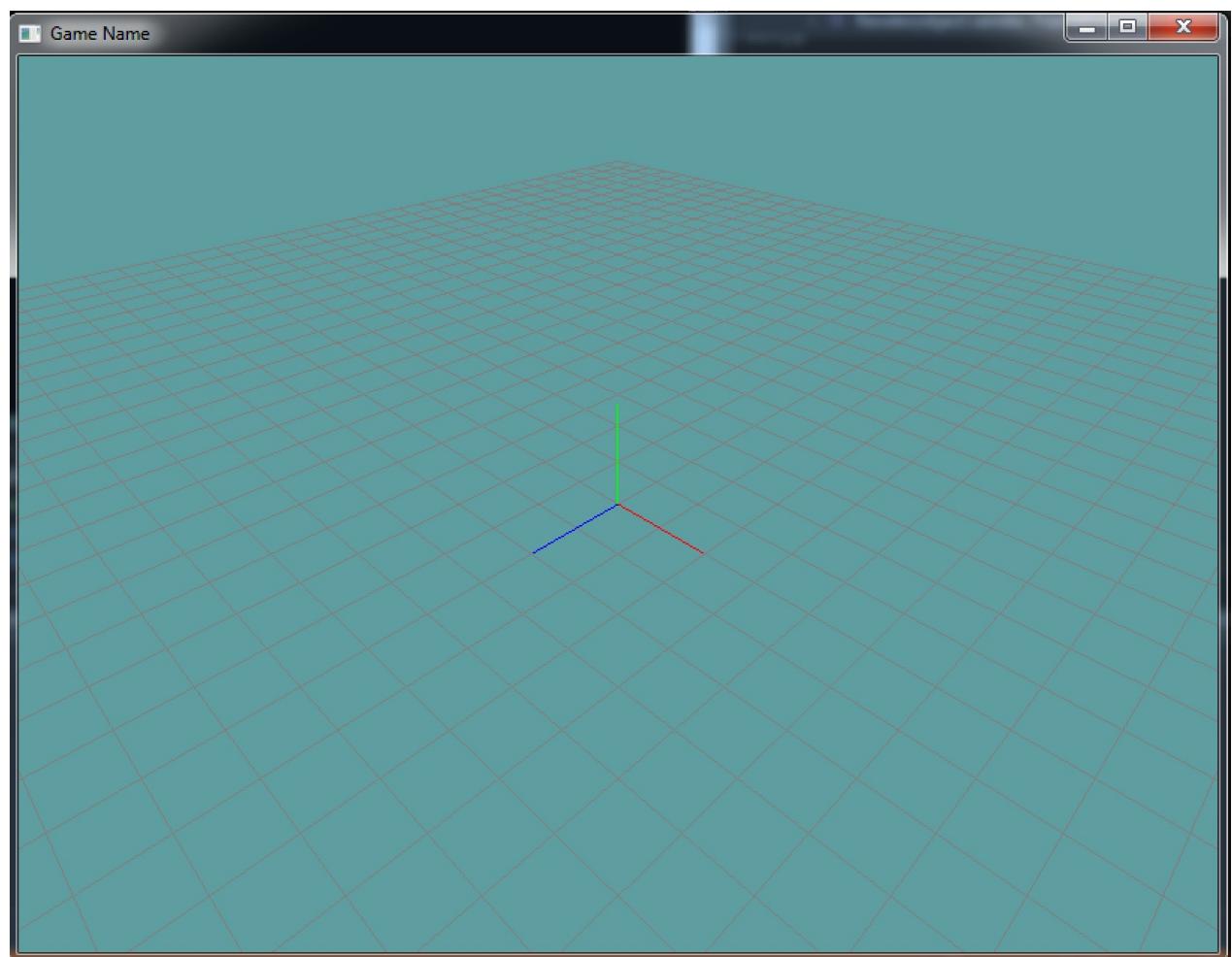
```
// Draw grid
// Set render color to gray
GL.Color3(0.5f, 0.5f, 0.5f);
// Draw a 40x40 grid at 0.5 intervals
// The grid will go from -10 to +10
GL.Begin(PrimitiveType.Lines);
// Draw horizontal lines
for (int x = 0; x < 40; ++x) {
    float xPos = (float)x * 0.5f - 10.0f;
    GL.Vertex3(xPos, 0, -10);
    GL.Vertex3(xPos, 0, 10);
}
// Draw vertical lines
for (int z = 0; z < 40; ++z) {
    float zPos = (float)z * 0.5f - 10.0f;
    GL.Vertex3(-10, 0, zPos);
    GL.Vertex3(10, 0, zPos);
}
GL.End();

// Draw basis vectors
GL.Begin(PrimitiveType.Lines);
// Set the color to R & draw X axis
GL.Color3(1.0f, 0.0f, 0.0f);
GL.Vertex3(0.0f, 0.0f, 0.0f);
GL.Vertex3(1.0f, 0.0f, 0.0f);
// Set the color to G & draw the Y axis
GL.Color3(0.0f, 1.0f, 0.0f);
GL.Vertex3(0.0f, 0.0f, 0.0f);
GL.Vertex3(0.0f, 1.0f, 0.0f);
// Set the color to B & draw the Z axis
GL.Color3(0.0f, 0.0f, 1.0f);
GL.Vertex3(0.0f, 0.0f, 0.0f);
GL.Vertex3(0.0f, 0.0f, 1.0f);
GL.End();
```

Keep in mind, right now we're strictly looking at the center of NDC space. The above code will not look too impressive. Running your application, your screen should look like this:



Once we are able to position the camera and add perspective you will see that what we did really looks like this:



The ModelView Matrix

The **modelView** matrix defines the coordinate system that is used to place and orient objects. This 4x4 matrix can either transform vertices, or it can be transformed its-self by other matrices.

Before you can do anything, you must specify if you are going to be working with the **modelView** or **projection** matrix. You can do this with the `GL.MatrixMode` function. This is the signature:

```
void GL.MatrixMode(MatrixMode);
```

The `MatrixMode` enum has two values we use, but the enum its-self has 5 values

- `MatrixMode.Modelview`
- `MatrixMode.Projection`
- `MatrixMode.Color` - we don't use this
- `MatrixMode.Texture` - we don't use this
- `MatrixMode.ModelviewOExt`

Usually (99% of the time) you want to set your modelView matrix to identity. So that you start around origin. you can achieve this by calling the `GL.LoadIdentity` function. This function loads the identity matrix into the OpenGL state machines active matrix. (You specified the active matrix with `GL.MatrixMode`)

This snippet resets the modelview matrix:

```
GL.MatrixMode(MatrixMode.Modelview);
GL.LoadIdentity(); // Reset modelview matrix
// Do other transformation
```

Multiplying OpenGL matrices

We're about to talk about 3 functions:

- `GL.Translate`
- `GL.Rotate`
- `GL.Scale`

Before we discuss how these functions work, i want you to be aware of what they actually do. When you call `GL.MatrixMode` with `MatrixMode.Modelview` as an argument, you select the **modelView** matrix of the OpenGL state machine as the active matrix.

Once you have a matrix set as the active matrix, all further matrix operations will be performed to that matrix. `GL.LoadIdentity` is one such operation. It takes the current matrix and sets it to identity.

The three functions above will generate an appropriate matrix and multiply the current matrix! `GL.Rotate` for instance will make a rotation matrix and multiply the current matrix by it.

The following OpenGL snippet

```
// Set the currently active matrix
GL.MatrixMode(MatrixMode.Modelview);

// Reset modelview matrix
GL.LoadIdentity();

// Create a rotation matrix
// then multiply the active matrix by it
GL.Rotate(90.0f, 0.0f, 0.0f, 1.0f);
```

Is the equivalent of:

```
void Render(ref Matrix4 modelView, ref Matrix4 projection) {
    // Set the currently active matrix
    Matrix4 currentMatrix = modelView;

    // Reset modelview matrix
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            currentMatrix[i, j] = (i == j)? 1.0f : 0.0f;
        }
    }

    // Create a rotation matrix
    Matrix4 rotation = Matrix4.AngleAxis(90.0f, 0.0f, 0.0f, 1.0f);

    // then multiply the active matrix by it
    currentMatrix = rotation * currentMatrix;
}
```

OpenGL just wraps all the math up for you. But the following function create and multiply matrices for you!

Translation

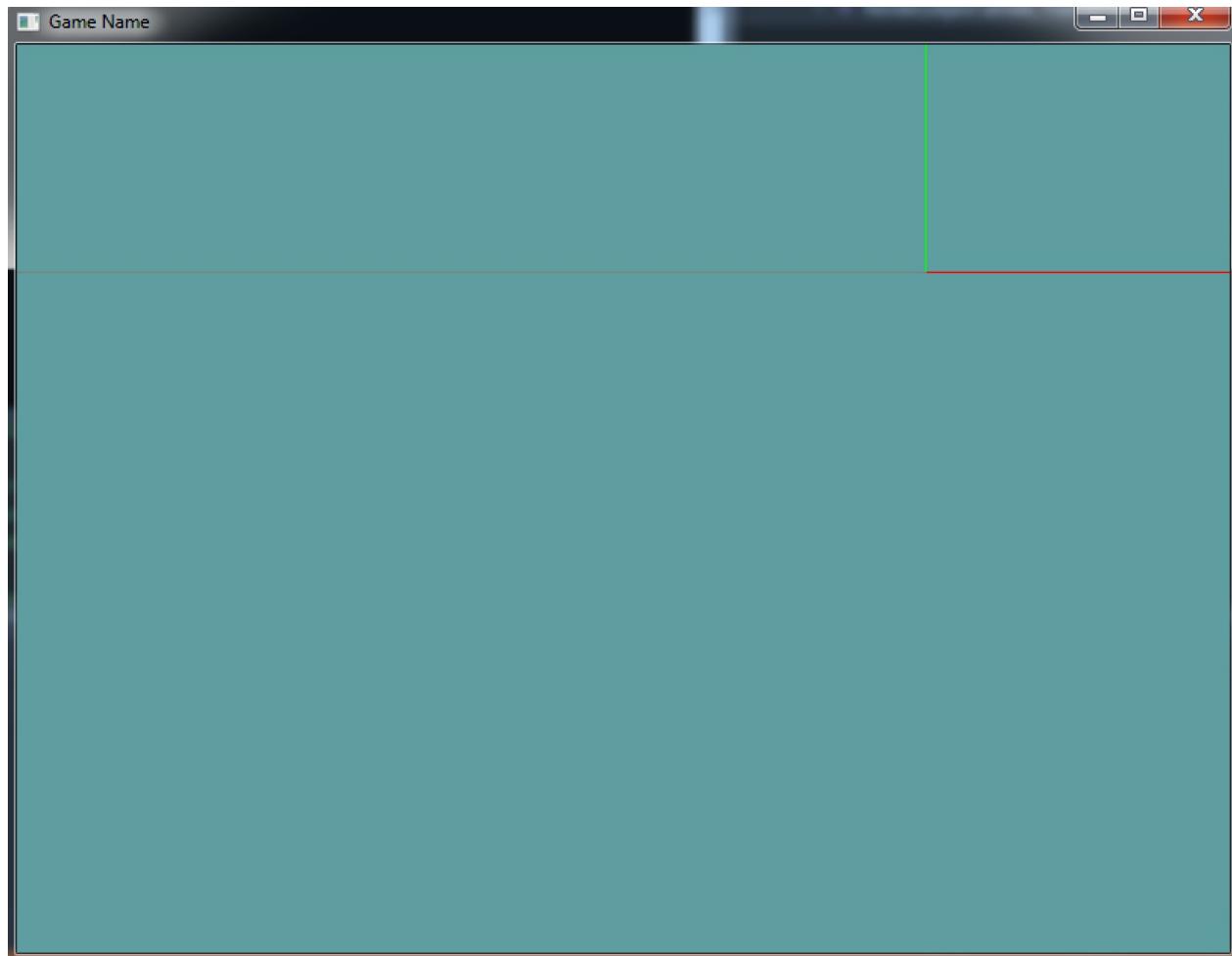
Translation allows you to move an object from one position in the world to another position in the world. The OpenGL function `GL.Translate` performs this functionality, it is defined as follows:

```
void GL.Translate(float x, float y, float z);
```

Suppose you want to move an object from origin to position (0.5, 0.5, 0.5) you would run this bit of code:

```
GL.MatrixMode(MatrixMode.Modelview);
GL.LoadIdentity(); // Reset modelview matrix
GL.Translate(0.5f, 0.5f, 0.5f);
```

If you add that code BEFORE the render code we did above, the screen will look like this:



Because matrix multiplication is associative you can even chain these together!

Try this code:

```
GL.MatrixMode(MatrixMode.Modelview);
GL.LoadIdentity(); // Reset modelview matrix
GL.Translate(0.5f, 0.5f, 0.5f); // At (5, 5, 5)
```

```
GL.Translate(-0.5f, -0.5f, -0.5f); // At (0, 0, 0)
GL.Translate(-0.5f, -0.5f, -0.5f); // At(-5, -5, -5)
```

Demo

```
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class TranslateSample : Game {
        Grid grid = null;

        public override void Initialize() {
            grid = new Grid();
        }
        public override void Update(float dTime) {

        }
        public override void Render() {
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity(); // Reset modelview matrix
            GL.Translate(0.5f, 0.5f, 0.5f);

            grid.Render();
        }
        public override void Shutdown() {

        }
    }
}
```

Rotation

Rotation in OpenGL is accomplished through the `GL.Rotate` function. The code for it will look familiar, that is because `GL.Rotate` creates a rotation matrix from an angle and an axis. Just like the `Matrix4.AngleAxis` function you wrote. It is defined as

```
void GL.Rotate(float angle, float xAxis, float yAxis, float zAxis);
```

The function rotates around the axis specified by `xAxis`, `yAxis` and `zAxis`. It rotates by `angle` degrees. The axis must be normalized. The function takes euler angles! The rotation is counter-clockwise!

For example, if you wanted to rotate by 90 degrees on the X axis, the call would be:

```
GL.Rotate(90.0f, 1.0f, 0.0f, 0.0f);
```

If you want to rotate clock-wise pass in a negative angle.

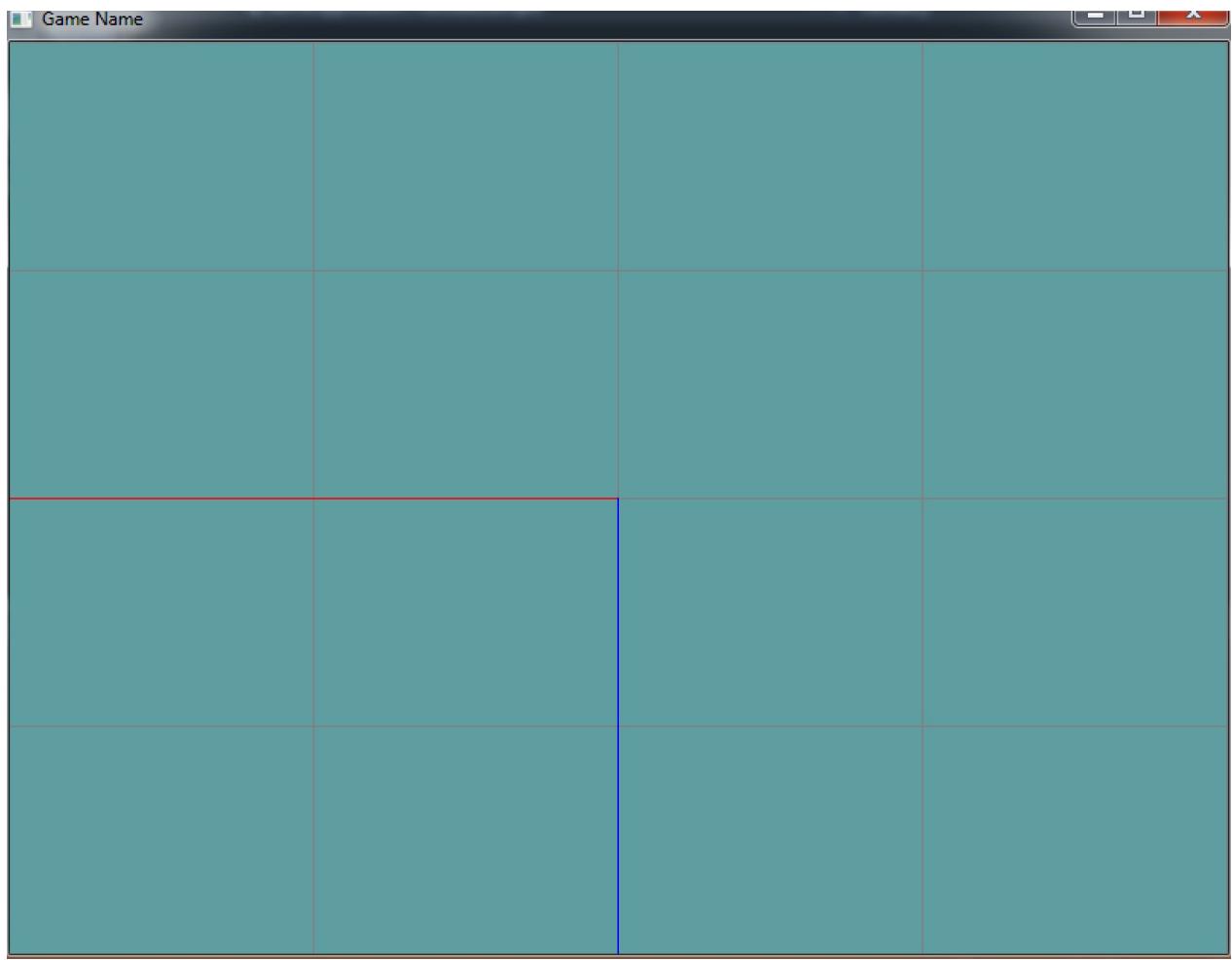
Let's say you want to rotate 45 degrees on the X axis and 7 degrees on the Y axis. Figuring out the arbitrary axis to rotate around and the new rotation angle would require a LOT of math. Luckily matrix multiplication is associative, so we can take advantage of that!

```
// First, rotate 45 degrees on X axis
GL.Rotate(45, 1.0f, 0.0f, 0.0f);
// Then, rotate 7 degrees on Y axis
GL.Rotate(7.0f, 0.0f, 1.0f, 0.0f);
```

On your own

Just like I provided a new class for the translate sample, make a new class for the rotate sample.

Try to rotate your world by 90 degrees on the X axis and then 180 degrees on the Z axis. The resulting window will look like this:



Scaling

Scaling in its most simple definition increases or decreases the size of an object or coordinate system. Scaling is pretty self explanatory, i'm not going to spend too much time on it.

The function looks like this

```
void GL.Scale(float x, float y, float z);
```

Scaling an object large is just using numbers bigger than 1 into the function. To shrink an object, use numbers smaller than 1 but greater than 0 (decimals). Using negative numbers will flip the object on the specific axis.

Go ahead and make a sample like we did with Translation and rotation. Try scaling big (2.0, 2.0, 2.0), also try scaling small (0.25, 0.25, 0.25)

The view matrix

Combining translation, rotation and scale creates a transformation matrix. Sometimes also called the model matrix, or model to world matrix. But we're editing the mode-view matrix! So what is the view matrix?

As discussed earlier, the view matrix is the inverse of the cameras world position! There are two things you need to know about your camera in world space, where it is (it's position) and which what it's looking (it's orientation).

Setting the view matrix with the fixed function pipeline is challenging at best, this is mainly because there is no fixed function equivalent of inserting a matrix!

This means if your camera is located at $(90, 20, 30)$ and rotated to $(7, 1, 0, 0)$ you would have to do the following commands:

```
// Negate the translation of the camera  
GL.Translate(-90, -20, -30);  
// Negate the rotation of the camera  
GL.Rotate(-7, 1, 0, 0);
```

Well, that wasn't so bad... Was it? No, but your world position for the camera is NEVER going to be that simple. But there is an easy way to create the world matrix of the camera, we just won't cover it until later (we cover it in the custom matrices section).

Until then, copy/paste and use [this](#) function. It is defined as follows:

```
void LookAt(float eyeX, float eyeY, float eyeZ, float targetX, float targetY, float targetZ, float upX, float upY, float upZ)
```

`LookAt` is actually a pretty standard OpenGL function. It's not included in the core of OpenGL, but it is so useful every programmer knows how it works!

The functions first three arguments are the position of the camera. So, if the camera is at $(5, 3, 1)$ then the first 3 arguments would be 5, 2, 1.

The second 3 arguments are the target that the camera is looking at. So if the camera is looking at world coordinates (0, 0, 0) then those would be the second three arguments.

The last three arguments are up. Which direction is up? Since the camera is in world space, and in world space positive Y is up, the last 3 arguments are almost always going to be 0, 1, 0.

Example

```
using System;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class LookAtExample : Game {
        Grid grid = null;

        protected static void LookAt(float eyeX, float eyeY, float eyeZ, float targetX, float targetY, float targetZ, float
            float len = (float)Math.Sqrt(upX * upX + upY * upY + upZ * upZ);
```

```

upX /= len; upY /= len; upZ /= len;

float[] f = { targetX - eyeX, targetY - eyeY, targetZ - eyeZ };
len = (float)Math.Sqrt(f[0] * f[0] + f[1] * f[1] + f[2] * f[2]);
f[0] /= len; f[1] /= len; f[2] /= len;

float[] s = { 0f, 0f, 0f };
s[0] = f[1] * upZ - f[2] * upY;
s[1] = f[2] * upX - f[0] * upZ;
s[2] = f[0] * upY - f[1] * upX;
len = (float)Math.Sqrt(s[0] * s[0] + s[1] * s[1] + s[2] * s[2]);
s[0] /= len; s[1] /= len; s[2] /= len;

float[] u = { 0f, 0f, 0f };
u[0] = s[1] * f[2] - s[2] * f[1];
u[1] = s[2] * f[0] - s[0] * f[2];
u[2] = s[0] * f[1] - s[1] * f[0];
len = (float)Math.Sqrt(u[0] * u[0] + u[1] * u[1] + u[2] * u[2]);
u[0] /= len; u[1] /= len; u[2] /= len;

float[] result = {
    s[0], u[0], -f[0], 0.0f,
    s[1], u[1], -f[1], 0.0f,
    s[2], u[2], -f[2], 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f
};

GL.MultMatrix(result);
GL.Translate(-eyeX, -eyeY, -eyeZ);
}

public override void Initialize() {
    grid = new Grid();
}

public override void Update(float dTime) {

}

public override void Render() {
    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity(); // Reset modelview matrix
    LookAt(0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

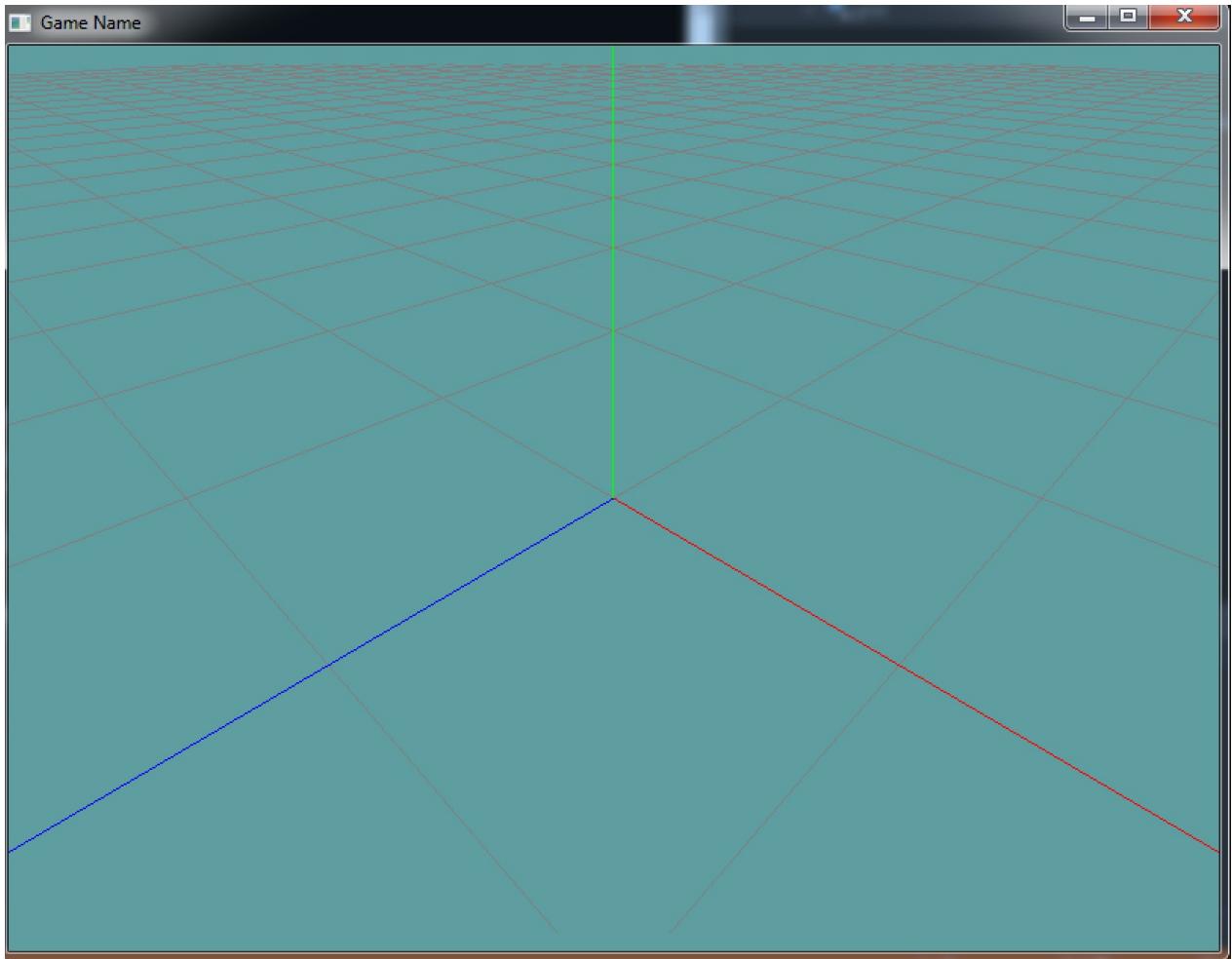
    grid.Render();
}

public override void Shutdown() {

}
}

```

Running the example produces this screen:



Cool, we can now see the grid with some perspective! Why are we so close? Well, we didn't set a projection matrix yet (we don't really know how to), so anything further than 1 will simply not render as our world projection is still in NDC. But hey, at least we can look at things with angles now!

Function order

Boy, we covered a LOT in the last section! We saw how we could translate, rotate and scale an object to create the world matrix. We also saw how we can use lookAt to create a view matrix.

Now, I always say matrix multiplication is associative, but not cumulative. This means:

- $A * B * C = A * (B * C)$
- $A * B \neq B * A$

So, what's the right order to multiply matrices in? Well, with the **modelView** matrix you would expect to multiply the model by the view matrix, but that is NOT the case. That's right, we're about to make matrices a LOT more confusing. But first, let's take a look at the full multiplication that's happening.

Remember, OpenGL is column major, so vectors are column matrices (1×4). Because matrix multiplication inner products must match, this means the vector goes on the right side. The entire transform pipeline looks like this:

```
In theory  
translated vertex = vertex * model * view * projection  
How OpenGL really does it  
renderVertex = vertex * modelView * projection
```

We want to first transform into model space, then view space, then projection space. Column major matrices have a slight gotcha to them, the matrices take effect from left to right, not right to left.

This means that the above code, would first move into projection space, then view space then model space. The exact opposite of what we want!

This is confusing, but in order to do this:

```
translated vertex = vertex -> model -> view -> projection
```

You have to multiply in reverse order, like so:

```
translated vertex = projection * view * model * vertex
```

This is not the case for row major matrices! Only column major ones.

Scaling, Translating and Rotating

The other order we need to worry about is scale-translate-rotate order.

What order should you scale/translate/rotate in?

- Scale First
- Rotate second
- Translate last

So the transformation pipe looks like this:

```
transformed = vertex -> scale -> rotate -> translate
```

Remember, multiply right to leftThis means your multiplication order will look like this:

```
Result = Translate * Rotate * Scale * Vertex
```

Fun times.

OpenGL functions

Hopefully that all made sense, so how does that apply to OpenGL functions? Well if you recall, openGL functions just multiply matrices, so we have to call them in the right order, like so

```
void Render() {
    // Pipeline: translated vertex = vertex -> model -> view -> projection
    // Model matrix: transformed = vertex -> scale -> rotate -> translate
    // Remember, read right to left when implementing multiplication!

    // TODO: Set up projection here
    // Multiply projection

    // Set the modelView matrix to identity
    GL.MatrixMode(MatrixMode.ModelView);
    GL.LoadIdentity();

    // Multiply view
    LookAt(0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

    // Draw an object
    // Multiply model (scale, rotate, translate)
    // Multiply translate
    GL.Translate(-3.0f, 0.0f, 1.0f);
    // Multiply rotate
    GL.Rotate(90.0f, 1.0f, 0.0f, 0.0f);
    // Multiply scale
    GL.Scale(1.0f, 1.0f, 1.0f); // No scale

    // The actual model has the input vertices
    DrawCube();
}
```

If this doesn't make sense right now, go ahead and give me a call on skype. I know it's a bit of a confusing topic.

Box Demo

Let's make a demo that draws a cube in the scene. This cube will have some rotation, scale and translation for you to play around with.

First, make a new demo scene and follow along

```
using System;
using OpenTK.Graphics.OpenGL;
```

```

namespace GameApplication {
    class BoxDemo : Game {
        Grid grid = null;

        // Add Look at function, it's in this gist
        // https://gist.github.com/gszauer/91038dbb010755d719de

        public override void Initialize() {
            grid = new Grid();
        }

        public override void Update(float dTime) {

        }
        public override void Render() {
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity(); // Reset modelview matrix
            LookAt(
                0.5f, 0.5f, 0.5f, // Position
                0.0f, 0.0f, 0.0f, // Target
                0.0f, 1.0f, 0.0f // Up
            );
            // Render grid at the origin of the world
            grid.Render();
        }
        public override void Shutdown() {
        }
    }
}

```

This puts us exactly where we left off in the last section. With the grid being visible at an angle. Remember, your machine will not look 100% like mine. We should have the same image, but yours might be culled out on the side.

Next, let's add a function that draws a cube. How do you draw a cube? A lot of triangles. I went ahead and drew one on paper, then counted all the verts out into CCW triangles. The result is [this static function](#).

Go ahead and add the above function to your class. In render, after drawing the grid set the render color to red, and draw the cube. Like so:

```

public override void Render() {
    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity(); // Reset modelview matrix
    LookAt(
        0.5f, 0.5f, 0.5f, // Position
        0.0f, 0.0f, 0.0f, // Target
        0.0f, 1.0f, 0.0f // Up
    );
    // Render grid at the origin of the world
    grid.Render();

    // Set render color
    GL.Color3(1.0f, 0.0f, 0.0f);
    // Draw cube
    DrawCube();
}

```

When you run the game one of two things is going to happen. Either the entire screen is going to be red, or you will see the scene as before, with no changes. This is because the size of the cube is -1 to +1. The same size as NDC. The cube will either fill NDC and render the whole screen red, or get clipped and not render at all. Depends on your graphics card's default Z-Test setting.

On your own

We now have a rudimentary 3D scene! I want you to apply the following transformation to the 3D cube on your own:

- Scale to 0.05f on all axis
- Rotate 73 degrees on the Y axis
- Rotate 45 degrees on the X axis
- Translate to 0.25f on the X axis
- Translate to -0.25f on the Z axis

Keep in mind, if your pipeline is

```
scale -> rotate -> translate
```

Then the multiplication order is

```
translate * scale * rotate
```

I'll give you a bit of a hint as to where to do this. The full transform pipeline is this:

```
render vertex = projection -> model -> view -> scale -> rotate -> translate -> original vertex
```

We don't have projection yet. No action there

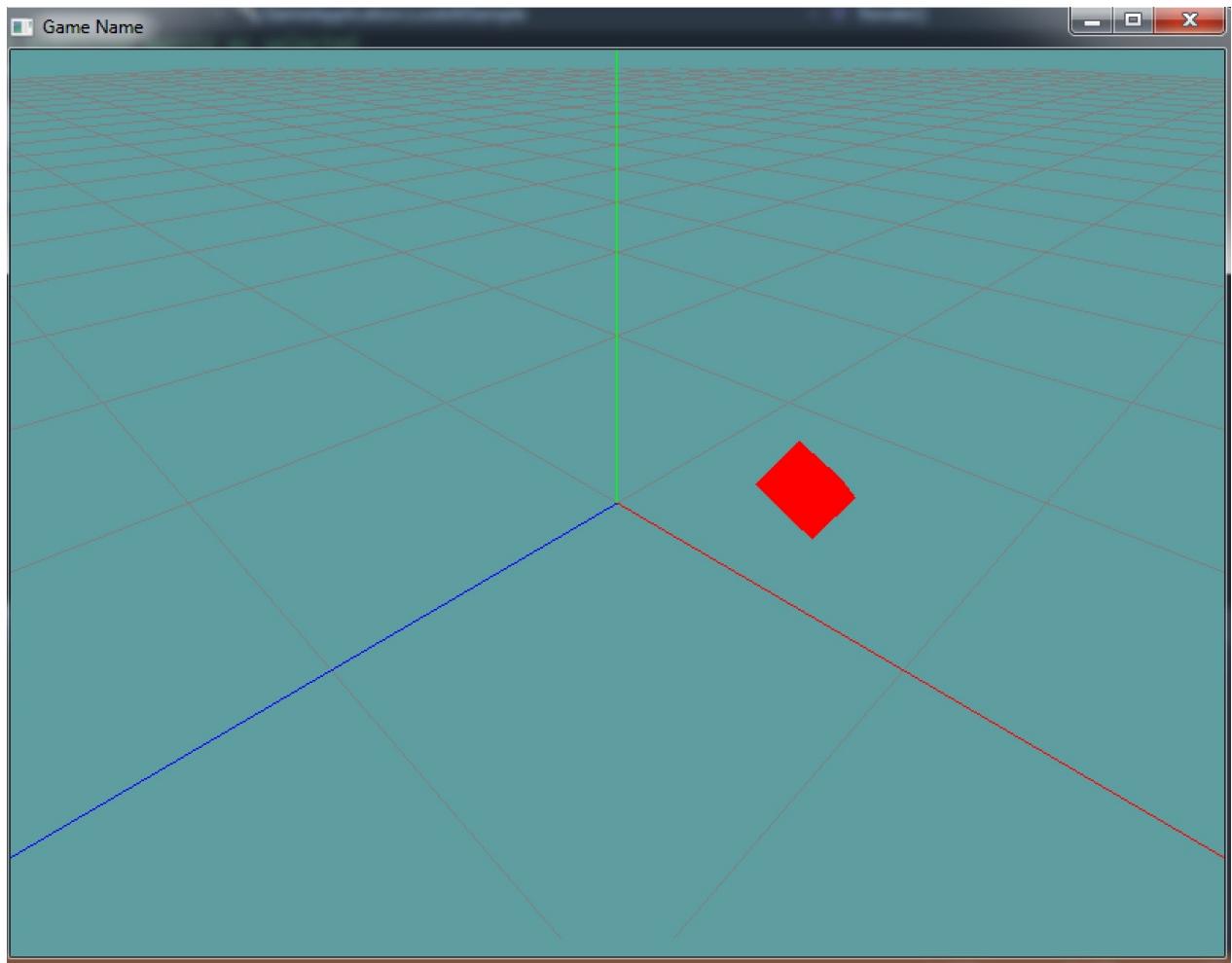
So you want to set the modelview matrix with the lookAt function (this was already done in the previous step, it does not change)

Next you want to render the grid at origin. We don't want to apply any transformations to the grid. You could say, the grid is only transformed by the view matrix, it has no model matrix.

Next you want to do your Scaling, Rotating and Translating. This is going to be the "model" part of the *modelView* matrix. It will position the square in the scene.

Last, you simply set the color to red (already there from previous section) and render by calling DrawCube (already there from previous section).

The resulting screen will look kind of like this:



If you get stuck, this page has a subsection with the solution. If i'm on skype, message me before you look at the solution, we can work trough the problem.

Solution

This is the cube drawing code i used:

```
public override void Render() {
    // Mark modelview matrix as selected
    GL.MatrixMode(MatrixMode.Modelview);

    // Reset modelview matrix
    GL.LoadIdentity();

    // Apply View Matrix
    LookAt(
        0.5f, 0.5f, 0.5f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );

    // Render grid, it has no model matrix
    grid.Render();

    // Construct a model matrix for the cube
    // Construct this matrix from rotation, translation and scale

    // Translate last
    GL.Translate(0.20f, 0.0f, -0.25f);
    // Rotate second
    GL.Rotate(45.0f, 1.0f, 0.0f, 0.0f);
    GL.Rotate(73.0f, 0.0f, 1.0f, 0.0f);
    // Scale first
    GL.Scale(0.05f, 0.05f, 0.05f);

    // Now that the modelview matrix is correct, render cube
    GL.Color3(1.0f, 0.0f, 0.0f);
    DrawCube();
}
```

Matrix Stacks

Rendering the cube scene is actually a pretty big deal! We now have a scene with a cube in it! I would call this the start of a game. What if we want to add another cube to our scene to make the game a bit more interesting?

After we transform the initial cube, we have to reset the modelview matrix to how it was before we rendered the first cube, otherwise the second cube will not render relative to the world origin, but rather relative to the first cube.

So this is what you would have to do:

- Select modelview matrix
- Load identity
- Apply view matrix (LookAt)
- Draw grid
- Apply cube 1 model matrix
- Draw cube 1
- Apply inverse of cube 1 model matrix
- Apply cube 2 model matrix
- draw cube 2

This is what the code for the above steps might look like

```
public override void Render() {
    // Mark modelview matrix as selected
    GL.MatrixMode(MatrixMode.Modelview);

    // Reset modelview matrix
    GL.LoadIdentity();

    // Apply View Matrix
    LookAt(
        0.5f, 0.5f, 0.5f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );

    // Render grid, it has no model matrix
    grid.Render();

    // Construct the model matrix for first cube
    GL.Translate(0.20f, 0.0f, -0.25f);
    GL.Rotate(45.0f, 1.0f, 0.0f, 0.0f);
    GL.Rotate(73.0f, 0.0f, 1.0f, 0.0f);
    GL.Scale(0.05f, 0.05f, 0.05f);

    // Render first cube
    GL.Color3(1.0f, 0.0f, 0.0f);
    DrawCube();

    // UNDO model matrix for first cube
    GL.Translate(-0.20f, -0.0f, 0.25f);
    GL.Rotate(-45.0f, 1.0f, 0.0f, 0.0f);
    GL.Rotate(-73.0f, 0.0f, 1.0f, 0.0f);
    GL.Scale(20.0f, 20.0f, 20.0f);

    // Construct model matrix for second cube

    // Now that the modelview matrix is correct, render cube
    GL.Translate(-2.5f, 0.0f, 0.25f);
    GL.Rotate(33.0f, 0.0f, 0.0f, 1.0f);
    GL.Rotate(97.0, 0.0f, 1.0f, 0.0f);
```

```

    GL.Scale(0.05f, 0.05f, 0.05f);
    GL.Color3(0.0f, 1.0f, 0.0f);
    DrawCube();
}

```

Study the code, if you run it you will see two cubes, one red and one green. This code does work!

You might be thinking to yourself, i see that applying the inverse of cube 1's model matrix just sets the *modelview* matrix back to the view. Instead of those 4 lines, why don't i just call LoadIdentity and LookAt again?

That logic is sound. Doing that would work, and you'd make your program a bit more readable. But that's not a maintainable solution! As soon as you have nested objects that approach breaks.

We can't stay with the first approach either! It's verbose, it's messy and it has the potential to introduce a lot of floating point error.

The stack

OpenGL fixes this issue of matrix-crazyness with matrix stacks! A matrix stack is exactly what it sounds like, a stack of matrices. There are two functions that you can use to control this stack:

```

void GL.PushMatrix();
void GL.PopMatrix();

```

- **GL.PushMatrix** will add a new matrix to the stack. This matrix is a copy of whatever the current top of the stack is
- **GL.PopMatrix** will take one matrix off of the stack

All matrix functions (LoadIdentity, LookAt, Translate, Rotate, Scale, etc...) only effect the top of the stack! This matrix stack acts as a history of matrices, allowing you to undo actions.

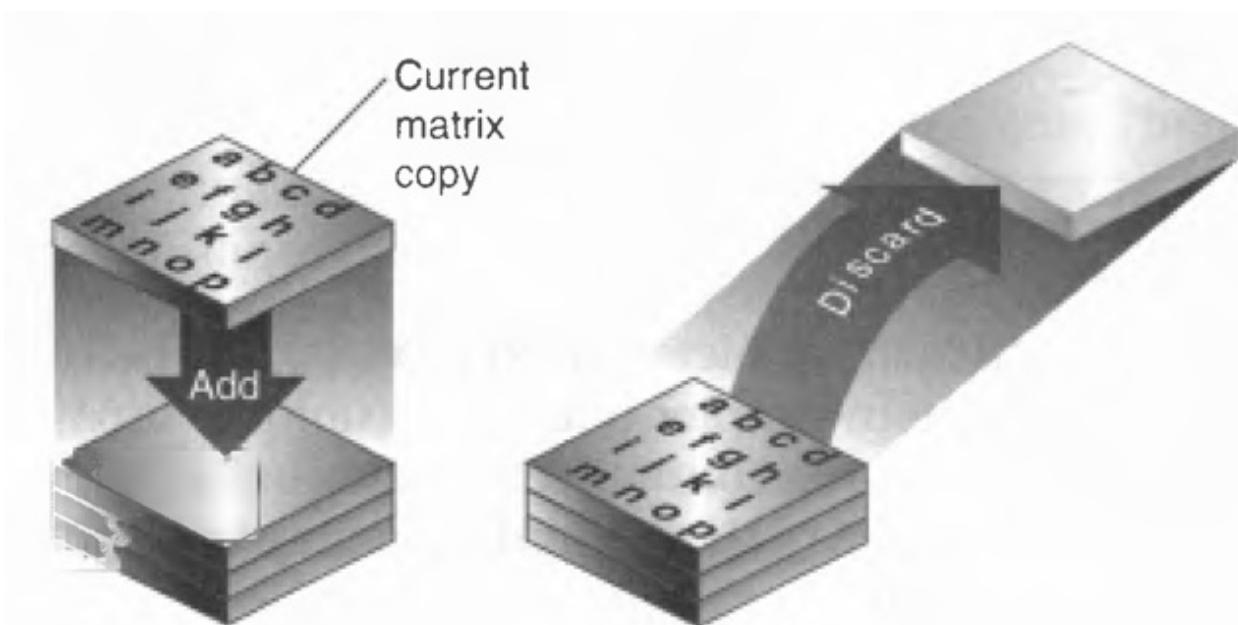
There is always at least one matrix on the stack, by default. This is what the state machine starts working on. For every PushMatrix call you make you MUST provide a matching PopMatrix.

The **modelView** matrix stack is guaranteed to be at least 32 elements deep. the **projection** stack is only guaranteed to be 2 elements deep. Better graphics cards *might* provide deeper stacks.

The same list of steps we took at the top of this page to render two cubes becomes this simple:

- Select modelview matrix
- Load identity
- Apply view matrix (LookAt)
- Draw grid
- **Push Matrix (save modelView)**
- Apply cube 1 model matrix
- Draw cube 1
- **Pop Matrix (restore modelView)**
- **Push Matrix (save modelView)**
- Apply cube 2 model matrix
- draw cube 2
- **PopMatrix(restore modelView)**

Basically you push before making changes to the matrix, then pop to undo those changes. Visually, here is an example of the matrix stacks:



For example:

```
// Stack height: 1
GL.MatrixMode(MatrixMode.Modelview);
GL.LoadIdentity();
// The top of the stack is now identity

GL.PushMatrix(); // Stack height: 2
// The top of the stack is still identity

LookAt(
    0.5f, 0.5f, 0.5f,
    0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f
);
// The top of the stack is now whatever the camera sees

GL.PushMatrix(); // Stack height 3
// The top of the stack is still whatever the camera sees

GL.Translate(3.0f, 0.0f, 0.0f);
// The top of the stack is now translated to 3.0f world space & multiplied by the view matrix

DrawModel();

GL.PopMatrix(); // Stack height 2
// The top of the stack is still whatever the camera sees

GL.PushMatrix(); // Stack height 3
// The top of the stack is still whatever the camera sees

GL.Rotate(45.0f, 1.0f, 0.0f);
// The top of the stack is now rotated at origin & multiplied by the view matrix
DrawModel();

GL.PopMatrix(); // Stack Height 2
// The top of the stack is still whatever the camera sees

GL.PopMatrix(); // Stack Height 1
// The top of the stack is now identity
```

On your own

Make a new demo. Render 3 cubes, one red one blue and one green. Use Push and Pop to save and restore matrix states between rendering them. Render all 3 at different locations on screen. When you are done, update git and let me know so i can review the code.

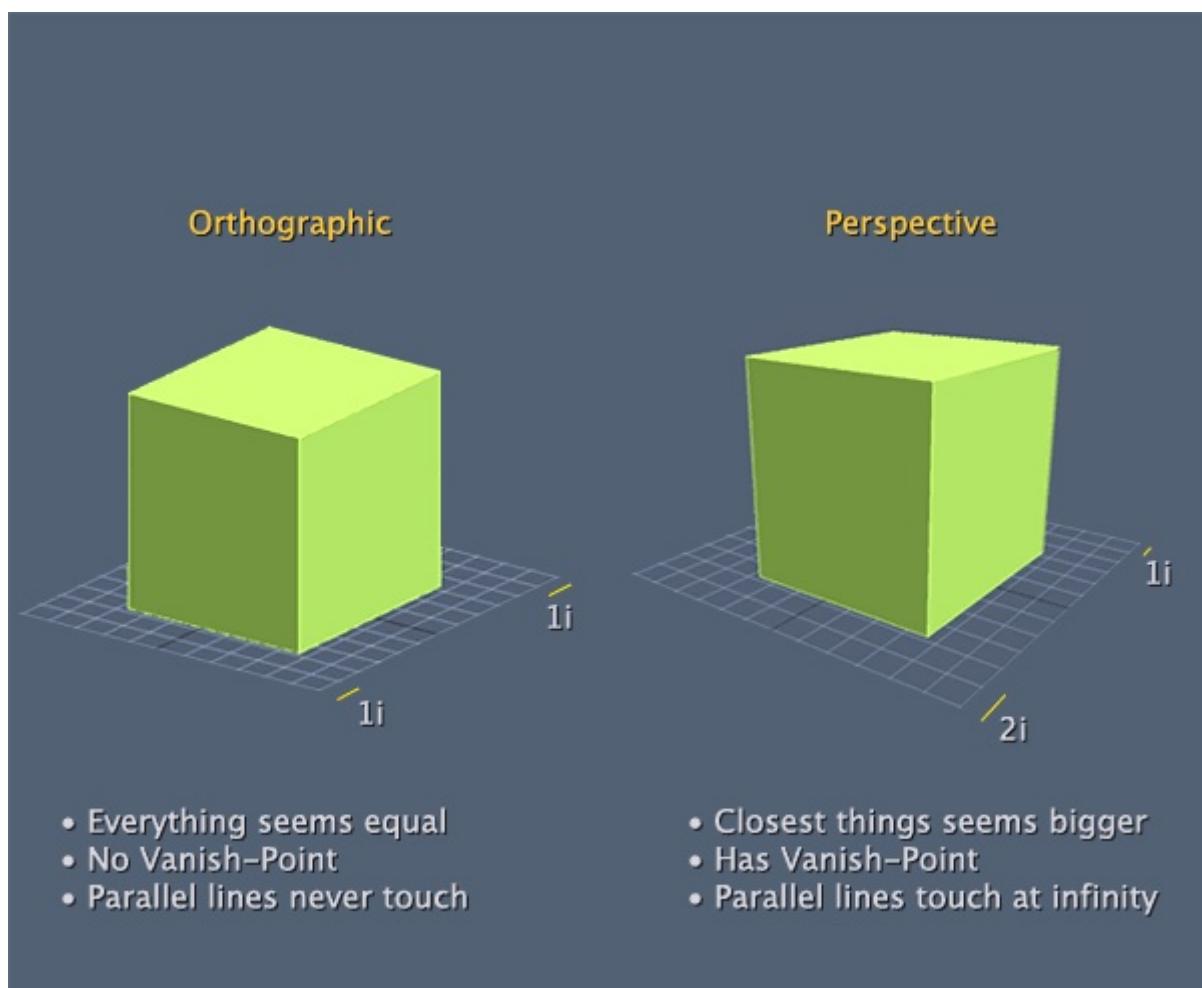
Hint: Keep the scale small, 0.05f or smaller. And don't translate mode than 1.0 units. I'd try to keep under 0.5f.
Remember, everything has to fit into NDC space!

Projections

We've mentioned projection transformations several times now, and even used them in code. It's about time we discuss how they work. As we've pointed out there are two general classes of projections available in OpenGL: **orthographic** (or parallel) and **perspective**.

By setting a projection transform, you are, in effect creating a viewing volume, which serves two purposes. The first is that the viewing volume defines a number of clipping planes, which determine the portion of your 3D world that is visible at any given time. Objects outside the volume are not rendered.

The second purpose of the viewing volume is to determine how objects are drawn. This depends on the shape of the viewing volume, which is the primary difference between perspective and orthographics render modes.



Before specifying any kind of projection transformation you need to make sure that the projection matrix is the selected matrix stack. As with the modelView matrix, this is done through the `GL.MatrixMode` function:

```
GL.MatrixMode(MatrixMode.Projection);
```

In most cases you want to follow this up with a call to `GL.LoadIdentity` to clear out anything that might be stored in this matrix. Then you set the actual matrix.

Unlike the **modelView** matrix, it's not likely that the projection matrix will change much. Usually the only time this

changes is when you resize the window.

Lets take a look at how to set the actual matrix

Orthographic

Orthographics projections are those that involve no perspective correction. In other words, no adjustment is made for the distance of the camera. Objects appear the same size on screen weather they are close up or far away.

OpenGL provides the `GL.Ortho` function to set an orthographic projection:

```
void GL.Ortho(float left, float right, float bottom, float top, float near, float far);
```

Left and right specify the X-coordinate clipping planes. Bottom and top specify the Y, and near and far specify the Z. This function essentially builds a box to render things in.

The default values of the orthographic projection are NDC space, or:

```
GL.Ortho(-1, 1, -1, 1, -1, 1);
```

This is cool, but it gives us a few issues. For one, all our geometry has to fit into the $-1 +1$ range!

Follow along in a new sample project. Let's use our scene that draws one cube as a starting point:

```
public override void Render() {
    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();

    LookAt(
        0.5f, 0.5f, 0.5f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );
    grid.Render();

    GL.Translate(0.20f, 0.0f, -0.25f);
    GL.Rotate(45.0f, 1.0f, 0.0f, 0.0f);
    GL.Rotate(73.0f, 0.0f, 1.0f, 0.0f);
    GL.Scale(0.05f, 0.05f, 0.05f);

    GL.Color3(1.0f, 0.0f, 0.0f);
    DrawCube();
}
```

Now, the projection is the last step in the transform pipeline, because of that, it's the first thing we should multiply! Let's set a projection that's 5 times bigger than the default NDC space!

```
public override void Render() {
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    GL.Ortho(-5, 5, -5, 5, -5, 5);

    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
```

```

        LookAt(
            0.5f, 0.5f, 0.5f,
            0.0f, 0.0f, 0.0f,
            0.0f, 1.0f, 0.0f
        );
        grid.Render();

        GL.Translate(0.20f, 0.0f, -0.25f);
        GL.Rotate(45.0f, 1.0f, 0.0f, 0.0f);
        GL.Rotate(73.0f, 0.0f, 1.0f, 0.0f);
        GL.Scale(0.05f, 0.05f, 0.05f);

        GL.Color3(1.0f, 0.0f, 0.0f);
        DrawCube();
    }
}

```

Run that code, all of a sudden you can see a lot more of the scene! Play around with the numbers, try 10, maybe even 15. See what a larger orthographic projection does.

As your projection matrix gets bigger, you can move your camera to positions other than 0.5f, as they will still hold the scene in View! Experiment a bit, see if you can predict what things will look like. Ortho projection can be a bit of a trip!

Before we leave off, I just want to mention, ortho projections don't have to be cubes. The following is perfectly valid:

```
GL.Ortho(-1, 1, -1, 1, -100, 100);
```

Fun fact, the graphics manager for our 2D framework uses orthographic projection to render the sprites. Maybe take a peek at what it does.

Shape

When you set your orthographic projection to be a square and resize your window to not be a square you will notice that the actual image on screen stretches or squashes. This is because we do not take aspect ratio into consideration!

Aspect ratio effects the WIDTH of your projection, with respect to its height. Therefore the aspect is width / height. Let's say you have the following projection:

```
GL.Ortho(-10f, 10f, -10f, 10f, -100f, 100f);
```

But you want the rendered image to look the same no matter what size the window. All you have to do is find the aspect ratio of the window, and multiply the left and right arguments by it. Like so:

```
float aspect = (float)MainGameManager.Window.Width / (float)MainGameManager.Window.Height;
GL.Ortho(-10f * aspect, 10f * aspect, -10f, 10f, -100f, 100f);
```

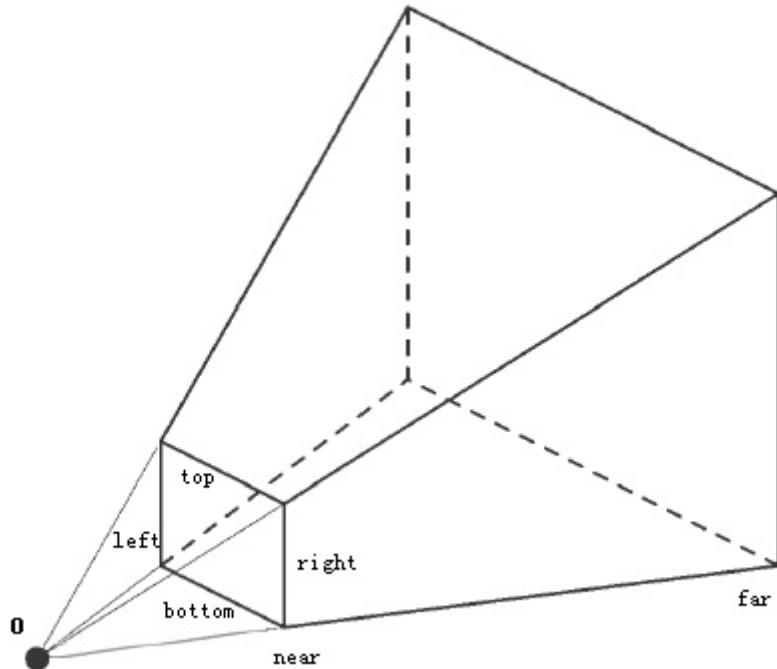
Test this by drawing a cube and resizing the window to be much wider than it is tall. Try it both with accounting for aspect ratio and without. See the difference between how the two look.

Perspective

Although orthographic projections can be interesting (not always in a good way) you need to create perspective projections to make realistic looking images.

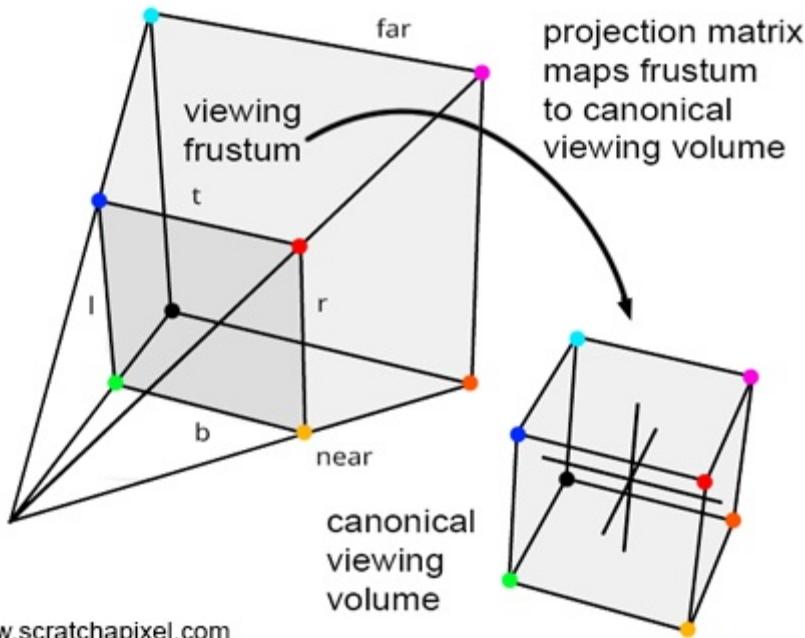
In a perspective projection, the further an object gets from the screen the smaller it becomes. Just like in the real world. This is commonly referred to as **foreshortening** (that term will probably never come up again)

The viewing volume of a perspective projection is called a **frustum** (that term will come up all the time), which looks like a pyramid with the top cut off.



The projection matrix is actually a matrix that maps the Frustum into a cube that occupies NDC space! So, if you have a cube which ranges from -1 to +1 and multiply it by the inverse of the projection matrix you will get a visual for the frustum!

When the projection matrix maps the frustum into a NDC cube, objects closer become smaller and objects further become larger.



© www.scratchapixel.com

There are a couple of ways to set up a view frustum. The canonical way OpenGL provides is the `GL.Frustum` function.

```
void GL.Frustum(float left, float right, float bottom, float top, float near, float far);
```

In the above function `left`, `right` and `bottom` specify the X and Y coordinates of the near clipping plane. `near` and `far` specify the distance from the viewer to the near and far clipping planes.

This means that the top left corner of the near plane is at (`left, top, -near`) and the bottom right corner is at (`right, bottom, -near`). The corners of the far plane are determined by casting a ray from the viewer passing through the near corners by a length for the far value.

Using `GL.Frustum` will allow you to set asymmetrical frustums, which can be cool but are not very useful. At all. In fact, I never use `GL.Frustum` in my code, because it is a VERY hard way to picture a view frustum. Instead i use this helper function (add it to your code!)

```
public static void Perspective(float fov, float aspectRatio, float zNear, float zFar) {
    float ymax = zNear * (float)Math.Tan(fov * Math.PI / 360.0f);
    float xmax = ymax * aspectRatio;

    GL.Frustum(-xmax, xmax, -ymax, ymax, zNear, zFar);
}
```

We will discuss the math behind this function later, in the custom matrices section. The arguments are much simpler, what is the field of view of the player. Human vision is about 90 degrees FOV, most games use a FOV of 60 because it looks good. The aspect ratio is simply the windows width divided by its height. Don't forget to cast these to floats! And the near and far plane are just the length of the frustum.

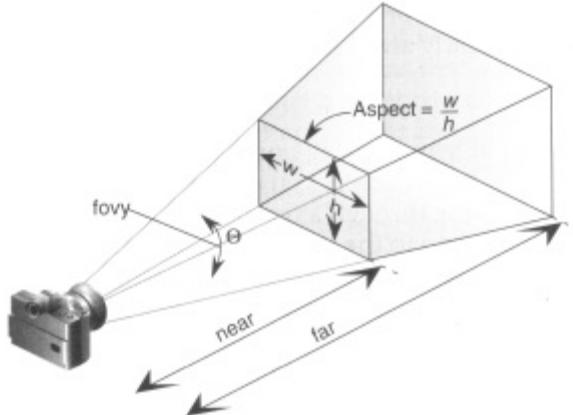


Figure 3-14 Perspective Viewing Volume Specified by gluPerspective()

This is again such a common function, almost every game programmer will recognize its signature!

Let's try to use it in our scene:

```

public override void Render() {
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    float aspect = (float)MainGameWindow.Window.Width / (float)MainGameWindow.Window.Height;
    Perspective(60.0f, aspect, 0.1f, 100.0f);

    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
    LookAt(
        10.0f, 10.0f, 10.0f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );
    grid.Render();

    GL.Translate(0.20f, 0.0f, -0.25f);
    GL.Rotate(45.0f, 1.0f, 0.0f, 0.0f);
    GL.Rotate(73.0f, 0.0f, 1.0f, 0.0f);
    GL.Scale(0.05f, 0.05f, 0.05f);

    GL.Color3(1.0f, 0.0f, 0.0f);
    DrawCube();
}

```

Take note, i actually moved my camera to 10,10,10 to give a better look at the scene. Again, play around in here! Try moving the camera. Play with the arguments of Perspective, see how this works!

If you're feeling brave you can even try adding a second cube ;)

One question you may have is why 0.1 and 100 for the near and far planes. Well, i figured that's a pretty large visible area. There is really no easy formula to figure these values out, it's just whatever looks good for your game. I tend to start out with 0.01 and 1000.0, and adjust from there.

Viewport

Some projection functions we just discussed are closely related to the viewport size. Take Perspective for example, we had to divide the viewport width and height. We know that the viewport transformation happens after the projection transformation, so now is as good a time as any to discuss it.

In essence the viewport specifies the dimensions and orientation of the 2D window on which you will be rendering. You can set it with:

```
void GL.Viewport(int x, int y, int width, int height);
```

x and y specify the coordinates of the **lower left** corner of the viewprot. Width and height specify window size in pixels.

The reason our windows didn't look the same up until this point is because we have had different default viewport values set. Once you are setting your own values for this our windows will look the same.

Let's take the scene we rendered in the Projections section, and manually specify a viewport for it

```
public override void Render() {
    // THIS IS THE ONLY NEW LINE!
    GL.Viewport(0, 0, MainGameWindow.Window.Width, MainGameWindow.Window.Height);

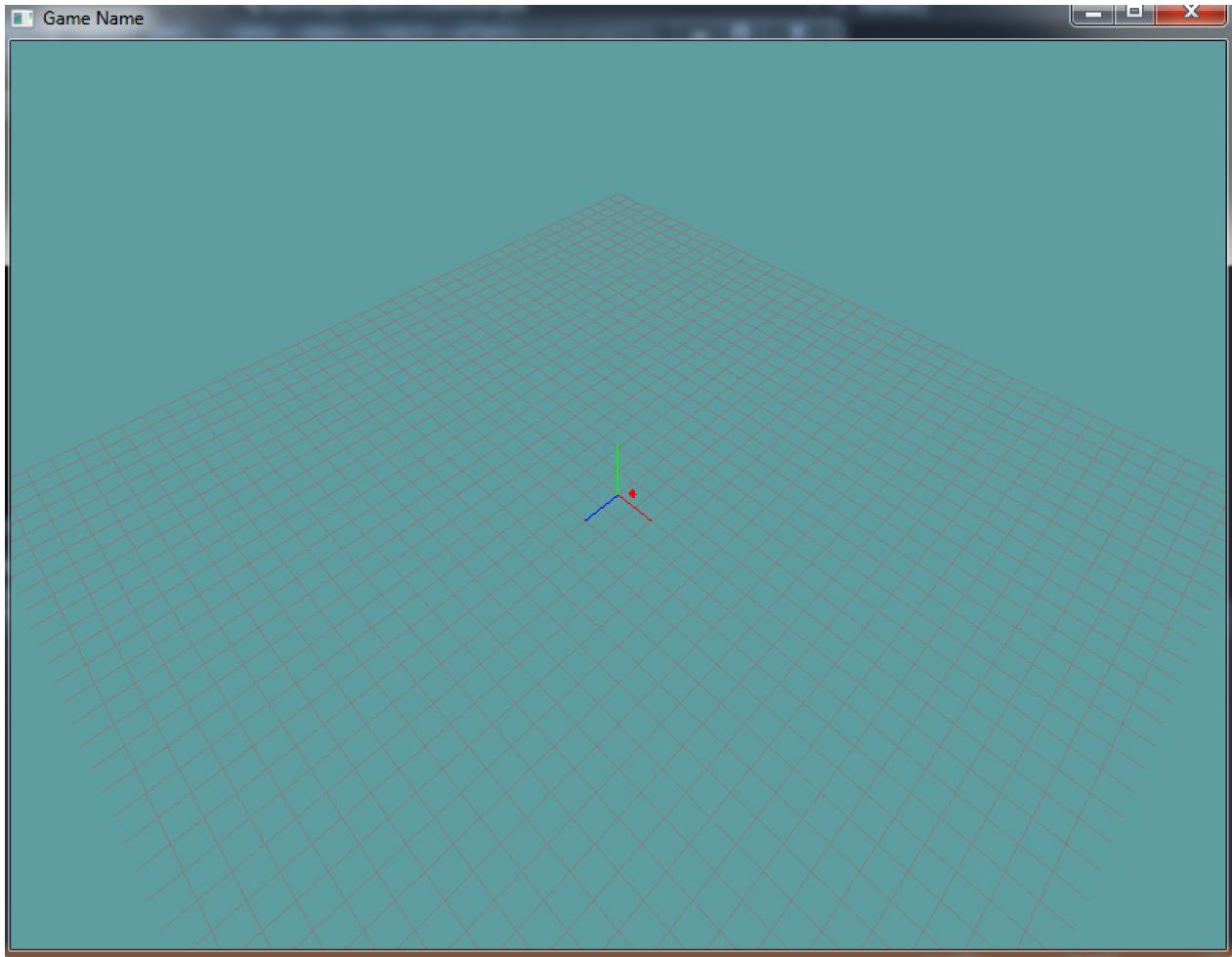
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    float aspect = (float)MainGameWindow.Window.Width / (float)MainGameWindow.Window.Height;
    Perspective(60.0f, aspect, 0.1f, 100.0f);

    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
    LookAt(
        10.0f, 10.0f, 10.0f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );
    grid.Render();

    GL.Translate(0.20f, 0.0f, -0.25f);
    GL.Rotate(45.0f, 1.0f, 0.0f, 0.0f);
    GL.Rotate(73.0f, 0.0f, 1.0f, 0.0f);
    GL.Scale(0.05f, 0.05f, 0.05f);

    GL.Color3(1.0f, 0.0f, 0.0f);
    DrawCube();
}
```

Now your screen should look exactly like mine:



Split-Screen

Ever wonder how split-screen games are done?

- Specify viewport to be half of the screen
 - `GL.Viewport(0, 0, screen.Width / 2, screen.Height)`
- Render the game from player 1's camera.
 - The aspect ratio will be the aspect ratio of the viewport, not the window
- Specify viewport to be second half of screen
 - `GL.Viewport(screen.Width / 2, 0, screen.Width / 2, screen.Height)`
- Render game from player2's camera

Pixel perfect

If you are ever doing any 2D rendering and want it to be pixel perfect, set your viewport to whatever the screen size is

```
GL.Viewport(0, 0, ScreenWidth, ScreenHeight);
```

Then set your orthographic matrix to be that exact size

```
GL.Ortho(0, ScreenWidth, ScreenHeight, 0, -1, 1);
```

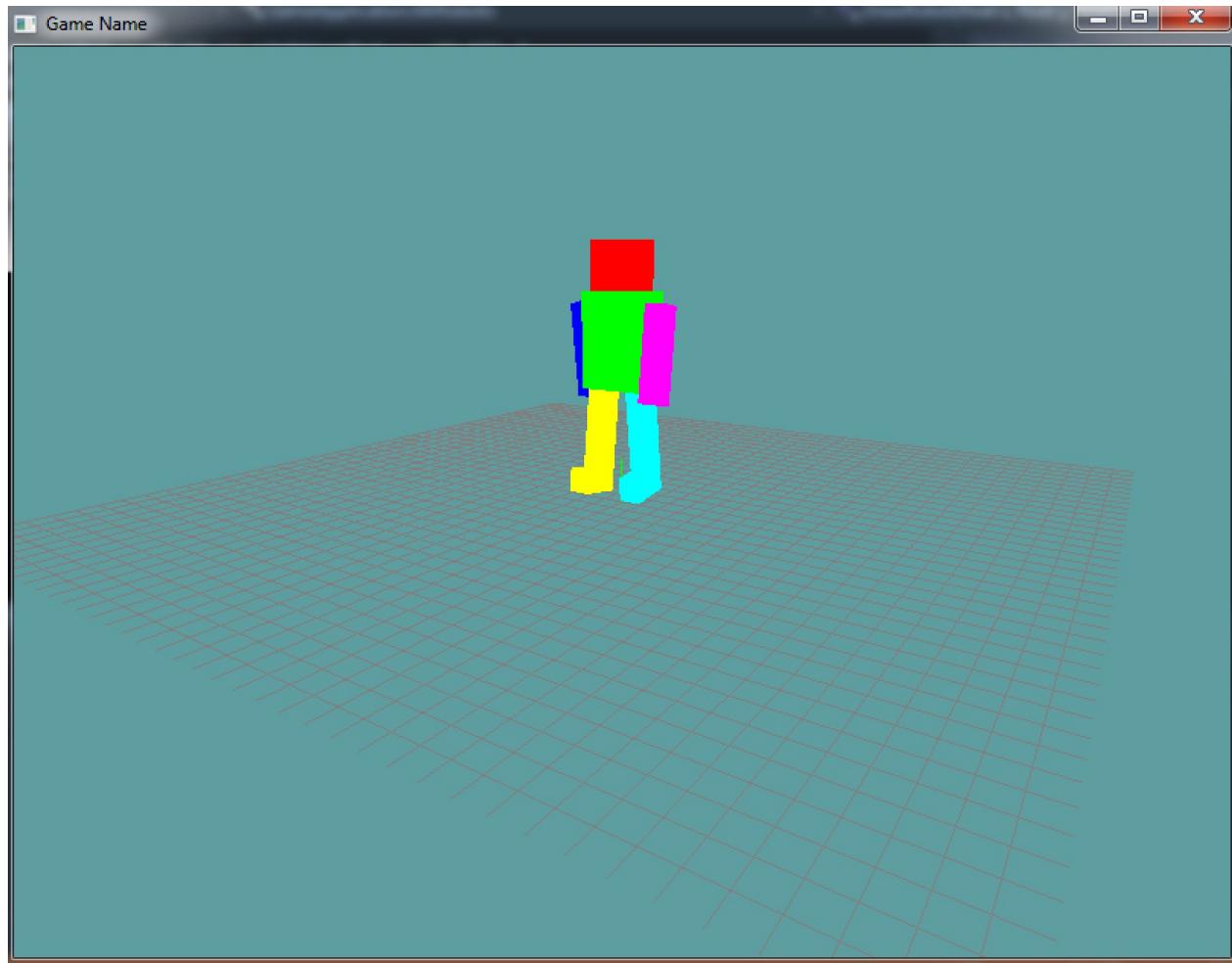
Take note how we map the left side of ortho to 0 and the right side to width.

You now have a pixel perfect render context. So, if you want to draw a line from pixels 4 to 7 you just draw from 4 to 7. This is how our 2D framework used to work.

Mr.Roboto

Now that we can set the perspective, model-view matrix and can position different elements using push and pop matrix lets make a complex scene. Instead of just drawing a few boxes, we're going to draw a box robot. At that, one that moves!

The final project will look something like this:



New scene

So, lets get started! Make a new demo scene that extends the `Game` class. Add the `DrawCube`, `LookAt` and `Perspective` helper functions to this class. Also, add a grid.

We're ging to add 8 floats to this class. The first 4 are the angles for the arms and legs of the robot. The last 4 are the direction in which the arms / legs are moving!

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
```

```

class MrRoboto : Game {
    Grid grid = null;

    float leftArmRot = -15.0f;
    float rightArmRot = 15.0f;
    float leftLegRot = 15.0f;
    float rightLegRot = -15.0f;

    float leftArmDir = 1.0f;
    float rightArmDir = -1.0f;
    float leftLegDir = -1.0f;
    float rightLegDir = 1.0f;

    public override void Initialize() {
        grid = new Grid();
        // THIS CALL IS NEEDED
        // We will discuss what it does later!
        GL.Enable(EnableCap.DepthTest);
    }

    protected static void LookAt(float eyeX, float eyeY, float eyeZ, float targetX, float targetY, float targetZ, float
        // copy / paste
    }

    public static void Perspective(float fov, float aspectRatio, float zNear, float zFar) {
        // copy / paste
    }

    public static void DrawCube() {
        // copy / paste
    }

    public override void Update(float dTime) {
        // todo
    }

    public override void Render() {
        // todo
    }
}

```

The only thing to note in the above code is the `GL.Enable(EnableCap.DepthTest)` call. We have not discussed what that does yet, we will in a later chapter. For now, just turn it on, it's needed for the game to run.

Update

Next up, lets animate those rotation variables! In the update we're going to animate the rotation of each limb by 50 degrees / second. If a limb's rotation is > +20 or < -20 we will reverse it's direction

```

public override void Update(float dTime) {
    // Update rotations
    leftArmRot += 50.0f * dTime * leftArmDir;
    rightArmRot += 50.0f * dTime * rightArmDir;
    leftLegRot += 50.0f * dTime * leftLegDir;
    rightLegRot += 50.0f * dTime * rightLegDir;

    // Clamp & change direction at edge
    if (leftArmRot > 20.0f || leftArmRot < -20.0f) {
        // Clamp to -15 or +15, depending on if the number
        // is negative or positive
        leftArmRot = (leftArmRot < 0.0f) ? -20.0f : 20.0f;
        // Change direction
        leftArmDir *= -1.0f;
    }
}

```

```

        if (rightArmRot > 20.0f || rightArmRot < -20.0f) {
            rightArmRot = (rightArmRot < 0.0f) ? -20.0f : 20.0f;
            rightArmDir *= -1.0f;
        }

        if (leftLegRot > 20.0f || leftLegRot < -20.0f) {
            leftLegRot = (leftLegRot < 0.0f) ? -20.0f : 20.0f;
            leftLegDir *= -1.0f;
        }

        if (rightLegRot > 20.0f || rightLegRot < -20.0f) {
            rightLegRot = (rightLegRot < 0.0f) ? -20.0f : 20.0f;
            rightLegDir *= -1.0f;
        }
    }
}

```

Render

Let's start out by simply rendering our grid, up to this point, the render function should look rather familiar. I'm also going to include a function called `RenderRobot`, i want to write the render code for the robot in it's own function, to make it re-usable and to keep the render loop clean.

```

public override void Render() {
    GL.Viewport(0, 0, MainGameWindow.Window.Width, MainGameWindow.Window.Height);

    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    Perspective(60.0f, (float)MainGameWindow.Window.Width / (float)MainGameWindow.Window.Height, 0.01f, 1000.0f);

    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
    LookAt(
        10.0f, 5.0f, 15.0f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );
    grid.Render();

    DrawRobot(-1.0f, 1.0f, 0.0f);
}

```

The draw robot function takes 3 arguments, world x, world y and world z. The cameras positon is 10, 5, 15 and it is looking at point 0, 0, 0. The near and far plane are almost 1000 units apart, so we can see just about anything in the scene.

Draw Robot

This is going to be the basic function to draw our (un-animated) robot:

```

void DrawRobot(float x, float y, float z) {
    // Save the matrix state we enter this function with
    GL.PushMatrix();

    // Translate the robot to the desired coordinates
    // because all body parts will be based off of this position
    // we don't need to push a matrix here
    GL.Translate(x, y, z);

    // Draw the head, because it multiplies a position we have to push matrix
    GL.Color3(1.0f, 0.0f, 0.0f); // Red
}

```

```

GL.PushMatrix();
    GL.Translate(1.0f, 4.0f, 0.0f);
    GL.Scale(0.5f, 0.5f, 0.5f);
    DrawCube();
GL.PopMatrix(); // finish head

// Draw the body
GL.Color3(0.0f, 1.0f, 0.0f); // green
GL.PushMatrix();
    GL.Translate(1.0f, 2.5f, 0.0f);
    GL.Scale(0.75f, 1.0f, 0.5f);
    DrawCube();
GL.PopMatrix(); // finish body

// Draw left arm
GL.Color3(0.0f, 0.0f, 1.0f); // blue
GL.PushMatrix();
    GL.Translate(0.0f, 2.25f, 0.0f);
    GL.Scale(0.25f, 1.0f, 0.25f);
DrawCube();
GL.PopMatrix();

// Draw right arm
GL.Color3(1.0f, 0.0f, 1.0f); // magenta
GL.PushMatrix();
    GL.Translate(2.0f, 2.25f, 0.0f);
    GL.Scale(0.25f, 1.0f, 0.25f);
    DrawCube();
GL.PopMatrix();

// Draw left leg
GL.Color3(1.0f, 1.0f, 0.0f); // yellow
GL.PushMatrix();
    GL.Translate(0.5f, 0.5f, 0.0f);
    GL.Scale(0.25f, 1.0f, 0.25f);
    DrawCube();
// Draw left foot. Since nothing else uses the modelview matrix before
// the next pop, we don't have to technically include this push call.
// But I think including it makes it obvious that the foot is being
// drawn relative to the leg!
GL.PushMatrix();
    GL.Translate(0.0f, -1.0f, 1.0f);
    GL.Scale(1.0f, 0.25f, 2.0f);
    DrawCube();
GL.PopMatrix();
GL.PopMatrix();

// Draw right leg
GL.Color3(0.0f, 1.0f, 1.0f); // baby blue
GL.PushMatrix();
    GL.Translate(1.5f, 0.5f, 0.0f);
    GL.Scale(0.25f, 1.0f, 0.25f);
    DrawCube();
// Draw right foot.
GL.PushMatrix();
    GL.Translate(0.0f, -1.0f, 1.0f);
    GL.Scale(1.0f, 0.25f, 2.0f);
    DrawCube();
GL.PopMatrix();
GL.PopMatrix();

// Restore original matrix state, like if this function
// never did anything to matrices ;
GL.PopMatrix();
}

```

The comments in the code above outline what is happening, but let's discuss it real quick anyway. The robot is made up of 8 cubes. We first scale the cube into the shape of a certain body part. Once the body part has the right shape, we translate it into position.

The most interesting bit here are the feet on the legs. Because each foot is rendered inside the push / pop block of

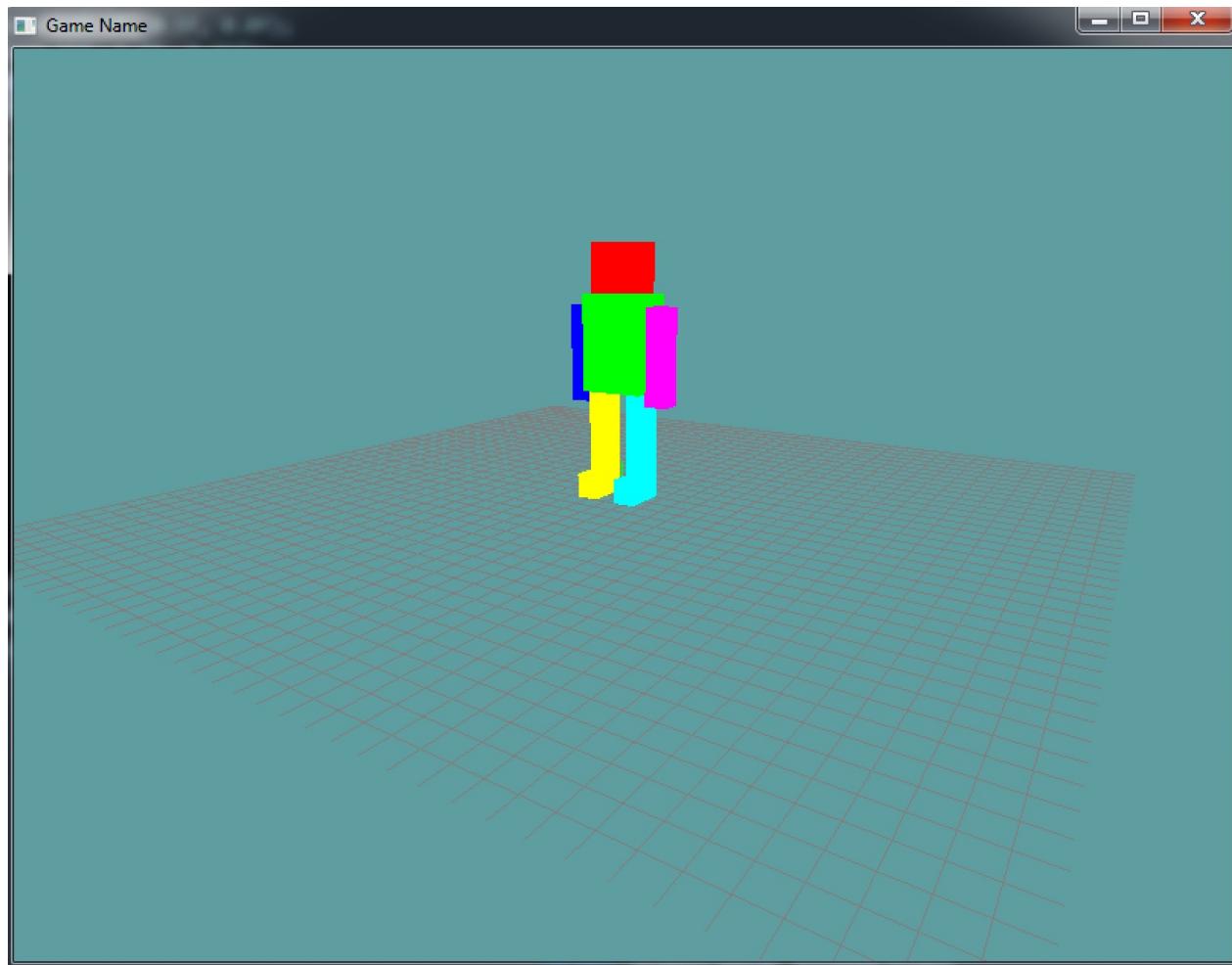
the leg it belongs to, when we rotate the leg we will also rotate the foot.

How did we come up with the scale (1.0f, 0.25f, 2.0f) for the feet? Well, if we don't translate or scale in there, just draw a cube it will draw the leg cube. We want the foot cube to be just as wide as the leg, so the X-scale stays 1. Relative to the leg, we want the foot to be 1/4 the height of the leg. So y scale becomes 0.25. We want the foot to be deeper than the leg, so it can jut out. Relative to the leg we want the foot twice as long, so we scale Z by 2.

This is an important concept. Each time we transform something, we are transforming it relative to the space it's in. The leg is relative to view space, the foot is relative to the leg's model space. If we had toes they would be relative to the foot's model space.

The concept of relative spaces is very important. If you don't understand it talk to me on skype!

With the above render code in place, your robot should render like so:



Animate

Awesome! Now we have a robot! But it's not doing anything.... Lame. Let's animate some movement!

To add animation, all we have to do is account for rotation for all major joints. Left and right arms both rotate. Left and right legs both rotate. The left and right feet don't rotate independently, they rotate with the legs.

Here is how you would rotate the left arm:

```

// Draw left arm
GL.Color3(0.0f, 0.0f, 1.0f); // blue
GL.PushMatrix();
    GL.Translate(0.0f, 2.25f, 0.0f);
    GL.Rotate(leftArmRot, 1.0f, 0.0f, 0.0f);
    GL.Scale(0.25f, 1.0f, 0.25f);
DrawCube();
GL.PopMatrix();

```

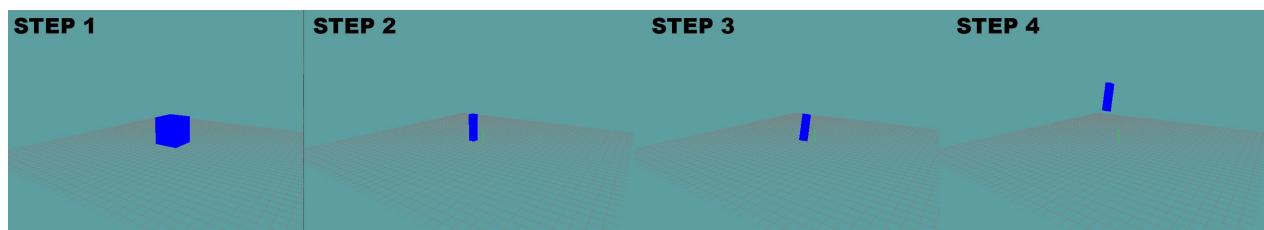
The only thing new is we added Rotate to the transformation matrix of the arm. Try to rotate the other limbs yourself.

Once you have all of this, you should see the limbs swinging. Only one problem, the arms are not rotating at the shoulder, they are rotating at the elbow! Same for the legs...

Pivot

The reason our rotations look messed up is because of how they pivot! To understand what's going on, let's dissect what happens when you draw just one part of the robot, the left arm:

1. We start off with a unit cube, this is just some cube
2. We scale the cube, this gives it the correct proportion
3. We apply a rotation to the cube, this rotates the cube around its own center
4. We position the rotated cube into its final place



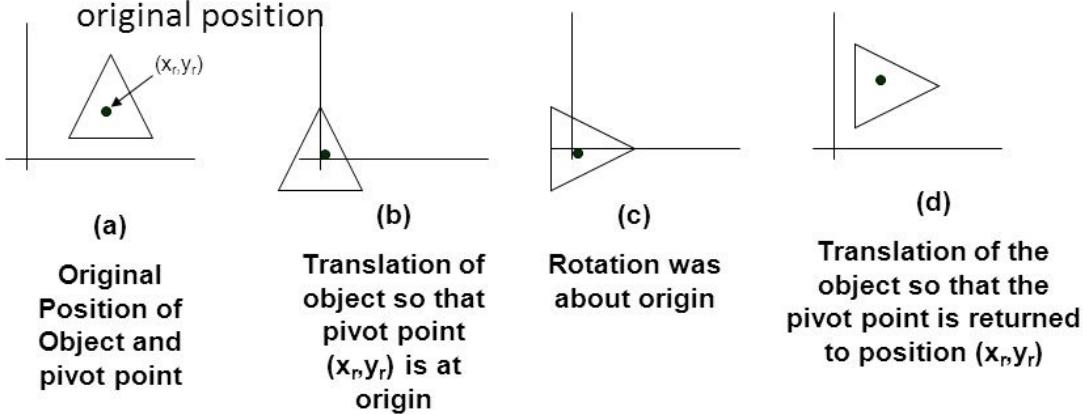
Where things break is step 3. The rotate function rotates the object around the center of the object's coordinate system, and that's why we don't pivot around the shoulder.

How can we fix this? Well, we know that all rotations happen around the origin of the object's model matrix. We can translate the model matrix so that the point we want to pivot is at its origin, then do the rotation and then translate the model matrix back to how it was. This would give us control over where the pivot point goes.

If that sounds confusing, maybe this will clear it up:

General pivot point rotation

- Translate the object so that pivot-position is moved to the coordinate origin
- Rotate the object about the coordinate origin
- Translate the object so that the pivot point is returned to its original position



So how would we do this pivoting for the left arm?

```
// Draw left arm
GL.Color3(0.0f, 0.0f, 1.0f); // blue
GL.PushMatrix();
    GL.Translate(0.0f, 2.25f, 0.0f);
    GL.Translate(0.0f, 1.0f, 0.0f); // UnPivot
    GL.Rotate(leftArmRot, 1.0f, 0.0f, 0.0f);
    GL.Translate(0.0f, -1.0f, 0.0f); // Pivot
    GL.Scale(0.25f, 1.0f, 0.25f);
    DrawCube();
GL.PopMatrix();
```

If you run with that code, the left arm will pivot around the shoulder joint. So, the question is, why do we move this joint by -1 on the y?

To answer that question, we still want to rotate around the center of the arm on the X and Z axis, we just want to rotate at the tip of the Y axis (on the top of the arm).

If you look at the cube being drawn, the Y scale is 1. Remember, the height of the cube is 2, because it goes from -1 to +1. Default rotation is around 0, 0. So to rotate around the tip of the arm we have to subtract 1, our pivot point is 0, -1.

Why -1, wouldn't +1 make more sense? Remember, in OpenGL positive Y goes down, negative Y goes up. If we pivoted to 0, 1 the arm would pivot around the fingers. Kind of the opposite of what we want.

Go ahead and try to add pivots to the rest of the robot to complete its walk-cycle.

Matrix relationships

What happened with the legs and feet of Mr.Roboto is super important. We must understand that each matrix manipulation we do, builds on top of previous matrix manipulations. Take the following code

```
GL.Translate(-2, -1, 7);
// Everything below this will be translated
```

So, when we wanted to move the foot of Mr.Roboto to move with the leg we had to multiply it while the model matrix of the leg was still on the stack.

This made the model matrix of the foot **relative** to the model matrix of the leg.

In OpenGL you can use indentation along with Push and Pop matrix to make these relationships very clear to understand. For example

```
void Draw() {
    GL.PushMatrix();

    // Transform to where the sun is
    Translate(...)
    Rotate(...)
    Scale(...)
    DrawSphere(); // < draws the sun

    // Mars circles the sun, so its model matrix has to be relative to the sun's!
    GL.PushMatrix();
        Translate(...)
        Rotate(...)
        Scale(...)
        DrawSphere(); // < draws mars

    // Mars has two moons, both orbit Mars, so their model matrices have to be relative to Mars'
    // However the moons are NOT relative to each other, so we have to save and restore matrices between rendering
    GL.PushMatrix()
        Translate(...)
        Rotate(...)
        Scale(...)
        DrawSphere(); // < draws Mars' "Phobos" moon
    GL.PopMatrix();
    GL.PushMatrix()
        Translate(...)
        Rotate(...)
        Scale(...)
        DrawSphere(); // < draws Mars' "Demios" moon
    GL.PushMatrix();

    GL.PopMatrix();
    GL.PopMatrix();

    GL.PopMatrix();
}
```

On your own

Make a new github gist. Copy / paste the above code into it. Add the pseudo code to render Earth and its moon.

Send me a skype link when it's finished.

Custom Matrices

Using OpenGL matrix functions is pretty easy. And the stack makes working with matrices even easier. But you already have a Matrix class built. Which already offers everything OpenGL matrices offer, and more!

Tour matrices are also a bit easier to work with than OpenGL's because you have full control and know they are mathematically accurate. Wouldn't it be great if you could use your own matrices to work with OpenGL?

Good news, you can!

OpenGL Matrix

The OpenGL matrix has been confusing programmers since 1992. This next bit might get a bit confusing. So, bear with me; re-read the section if you need to and call me on Skype if you have questions.

Mathematically OpenGL uses **column major** matrices. Because the matrices are *column major*, OpenGL uses **column vectors**. This means OpenGL works **post multiplication** because a vector is a 4×1 matrix, so in order for inner dimensions to match it has to go on the right hand side of a matrix. **matrix * vector**. The matrices are **right handed**, that is **-Z is forward**, it goes *into* the monitor.

Everything bolded in the above paragraph is the theoretical law of the land in OpenGL world. Write them on a notecard, memorize them! That is how the OpenGL specification says OpenGL matrices will work.

A column major matrix has its basis vectors in the first 3 columns and its translation vector in the 4th column, like so:

```
xx  yx  zx  tx  
xy  yy  zy  ty  
xz  yz  zz  tz  
0   0   0   1
```

Topology

Topology refers to how the matrix is stored in memory, THIS is where OpenGL gets confusing. Consider the following column major matrix:

```
xx  yx  zx  tx  
xy  yy  zy  ty  
xz  yz  zz  tz  
0   0   0   1
```

Given that a matrix is stored as an array of floats (size 16) you would expect it to be laid out in memory like so:

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]  
xx  yx  zx  tx  xy  yy  zy  ty  xz  yz  zz  tz  0   0   0   1
```

Which makes sense, the matrix is stored 1 row at a time in a linear array. The X basis vector occupies elements 0, 4, 8 and 12. This is how your matrix works, and to be honest; this is how any sane persons matrix would be written.

In the above example, the matrix is stored linearly one row at a time.

Not OpenGL

In OpenGL the following (still column major) matrix:

```
xx  yx  zx  tx  
xy  yy  zy  ty  
xz  yz  zz  tz  
0   0   0   1
```

Is stored in memory like this:

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]  
xx  xy  xz  0   yx  yy  yz  0   zx  zy  zz  0   tx  ty  tz  1
```

That is, the matrix is stored one column at a time. Instead of storing each row in memory one after another, OpenGL stores each column in memory, one after another.

This is actually a very un-intuitive way to store a matrix. So, why does OpenGL do it this way? When the standard for the OpenGL Matrix was defined in 1992 the hardware of the time could run three times faster with this memory layout. Today it's a nuisance, a major source of confusion and annoyance. But we have to keep doing things this way for historical reasons.

Take a few minutes, let that sink in. See if you can figure out how the following matrix would be stored in OpenGL's memory and in your matrices memory. Keep in mind, the matrix is stored in a 1 dimensional float array that has 16 elements. Let me know on skype before moving on to the next section:

```
A B C D  
E F G H  
I G K L  
M N O P
```

Transpose

The takeaway from the above section is actually rather simple, while you use the same theoretical memory layout as OpenGL (column major, post multiplication) you actually store your data in memory transposed of how OpenGL stores its data in memory.

This just means that before you load your own custom matrix into OpenGL, you must transpose it.

Loading a matrix

Now with all that theory out of the way, how do you actually load a matrix into OpenGL? There are two functions, the first one is `GL.LoadMatrix`. This function takes an array of float's as an argument:

```
void GL.LoadMatrix(float[] matrix);
```

This function will take the matrix passed into it, and **REPLACE** whatever is on the top of the stack with the matrix. It's useful for loading the view matrix, but after that you can't really use it for much else without ruining the view matrix.

The second method is `GL.MulMatrix`, like LoadMatrix it takes an array of 16 floats:

```
void GL.MulMatrix(float[] matrix);
```

Unlike LoadMatrix, MulMatrix respects the top of the stack. It's going to take whatever argument you give it and multiply the top of the matrix stack by that argument. This is the function you want to use to load custom matrices onto an existing stack

Sample

Let's see how to use it. Make a new demo scene that extends the `Game` class. We will need the LookAt, Perspective and DrawCube functions. Also include a grid.

Draw a cube at 1, 0, 0; rotated 45 degrees on the x axis; with a scale of 0.5f.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class MatrixDemo1 : Game {
        Grid grid = null;

        public override void Initialize() {
            grid = new Grid();
        }

        protected static void LookAt(float eyeX, float eyeY, float eyeZ, float targetX, float targetY, float targetZ, float
            // Copy / paste
        }

        public static void Perspective(float fov, float aspectRatio, float zNear, float zFar) {
            // Copy / paste
        }

        public static void DrawCube() {
            // Copy / paste
        }

        public override void Render() {
            GL.Viewport(0, 0, MainGameWindow.Window.Width, MainGameWindow.Window.Height);

            GL.MatrixMode(MatrixMode.Projection);
            GL.LoadIdentity();
            Perspective(60.0f, (float)MainGameWindow.Window.Width / (float)MainGameWindow.Window.Height, 0.01f, 1000.0f);

            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
            LookAt(
                10.0f, 4.0f, 0,
                0.0f, 0.0f, 0.0f,
                0.0f, 1.0f, 0.0f
            );
            grid.Render();

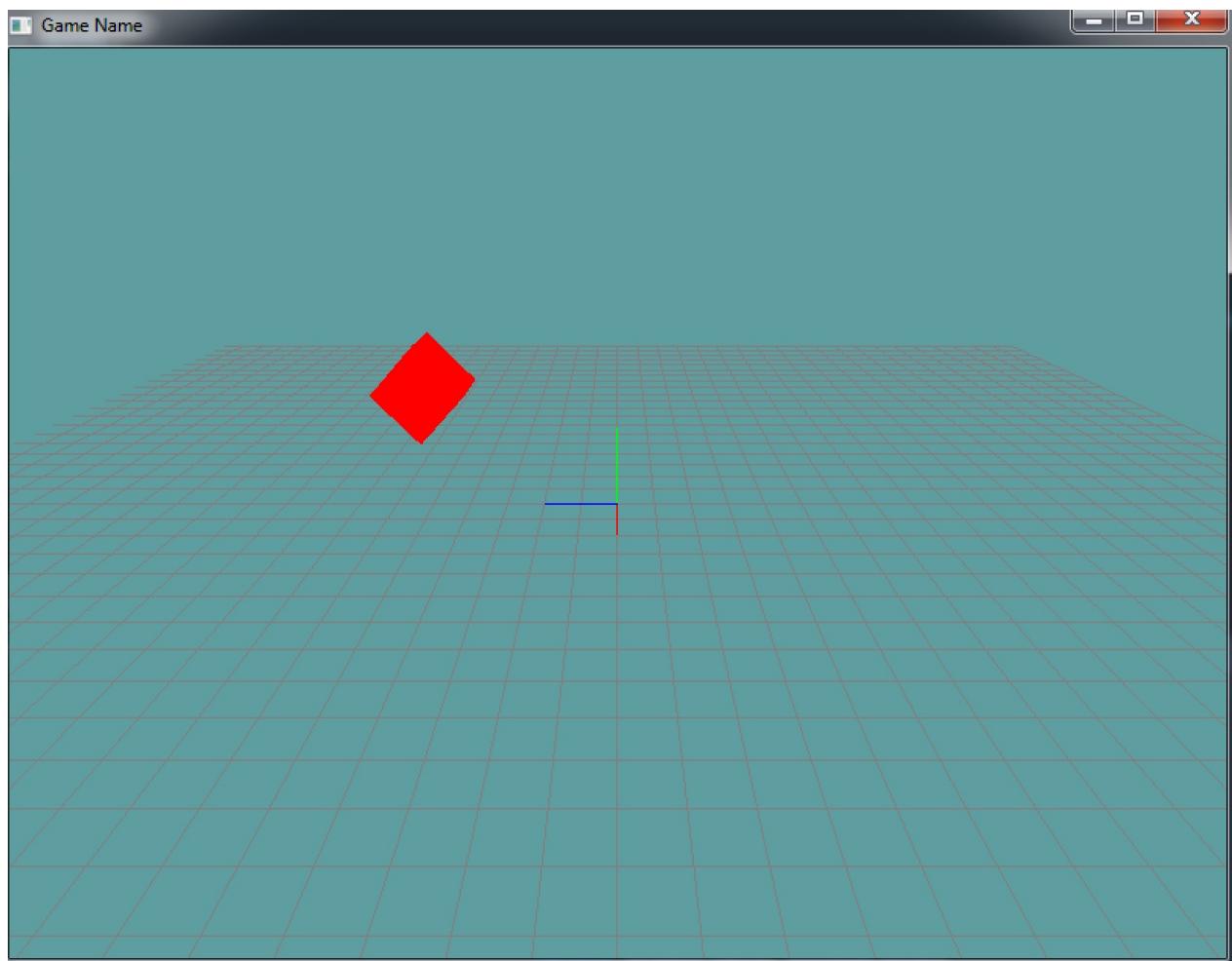
            GL.PushMatrix();
```

```

        GL.Color3(1.0f, 0.0f, 0.0f);
        GL.Translate(-2, 1, 3);
        GL.Rotate(45.0f, 1.0f, 0.0f, 0.0f);
        GL.Scale(0.5f, 0.5f, 0.5f);
        DrawCube();
        GL.PopMatrix();
    }
}
}

```

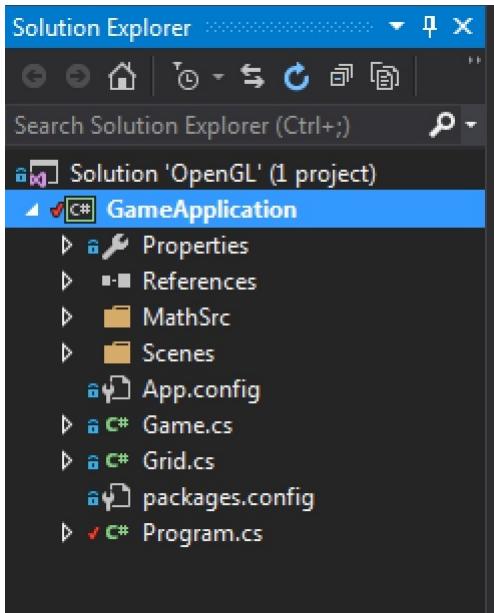
Here is what this screen looks like:



Let's see if we can replace the **model** matrix of the rendered cube with our own custom matrix.

Check **Math Implementation** on github, i opened a new ticket. Fix it before moving on! When you're using custom matrices and things don't render correctly (but they did with OpenGL matrices) the problem is always with your implementation. Even if it passed a few unit tests, the real word scenario sometimes breaks down.

First thing's first, import your math implementation code into the project. Keep things clean, put those files under a folder in visual studios solution view. I'd also like you to put your demo scenes in their own folder, like this:



Make sure to use the namespace these where implemented under `using MathImplementation;`. The namespace `OpenGL` also contains a `Matrix4` class, so make sure you are not using the namespace directly. Meaning this is ok: `using OpenTK.Graphics.OpenGL;` but don't do this: `using OpenTK;`.

Replacing the OpenGL matrices.

Let's look at the render function with your custom matrices in place:

```
public override void Render() {
    GL.Viewport(0, 0, MainGameWindow.Window.Width, MainGameWindow.Window.Height);

    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    Perspective(60.0f, (float)MainGameWindow.Window.Width / (float)MainGameWindow.Window.Height, 0.01f, 1000.0f);

    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
    LookAt(
        10.0f, 4.0f, 0,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );

    grid.Render();

    GL.Color3(1.0f, 0.0f, 0.0f);
    GL.PushMatrix();
    {
        Matrix4 scale = Matrix4.Scale(new Vector3(0.5f, 0.5f, 0.5f));
        Matrix4 rotation = Matrix4.AngleAxis(45.0f, 1.0f, 0.0f, 0.0f);
        Matrix4 translation = Matrix4.Translate(new Vector3(-2, 1, 3));

        // SRT: scale first, rotate second, translate last!
        Matrix4 model = translation * rotation * scale;
        // Remember to transpose your matrix!
        GL.MultMatrix(Matrix4.Transpose(model).Matrix);

        DrawCube();
    }
    GL.PopMatrix();
}
```

As you can see we create 3 matrices: scale, rotation and translation. We make the model matrix for the cube by combining the 3 transformation matrices we created. Because we're in column major (post multiplication) we multiply right to left.

As discussed, we can't apply the matrix directly, because we're storing it in a different manner than OpenGL expects. So, we transpose the matrix before giving it to OpenGL.

We use `GL.MultMatrix`. When that call is reached the only thing on the stack is the view matrix. `GL.MultMatrix` will multiply our matrix into the view matrix to create a `modelView` matrix.

We still need to push and pop matrices. We do this to restore the top of the matrix stack to the view matrix, in case anything else needs to be rendered.

One thing you may have noticed is i put a `{ }` block inside `GL.PushMatrix` and `GL.PopMatrix`. Why did i do this?

Well, we made 4 local matrices: scale, rotation, translation and model. Instead of those matrices being scoped to the function, they are now scoped to those brackets. So, if we have two cubes, we can re-use the matrix names as the ones up there fall out of scope when the `}` is hit.

Like this:

```
public override void Render() {
    // Omitted to save space, everything up to grid.Render is here

    GL.Color3(1.0f, 0.0f, 0.0f);
    GL.PushMatrix();
    {
        Matrix4 scale = Matrix4.Scale(new Vector3(0.5f, 0.5f, 0.5f));
        Matrix4 rotation = Matrix4.AngleAxis(45.0f, 1.0f, 0.0f, 0.0f);
        Matrix4 translation = Matrix4.Translate(new Vector3(-2, 1, 3));

        Matrix4 model = translation * rotation * scale;
        GL.MultMatrix(Matrix4.Transpose(model).Matrix);

        DrawCube();
    }
    GL.PopMatrix();

    GL.Color3(0.0f, 1.0f, 0.0f);
    GL.PushMatrix();
    {
        Matrix4 scale = Matrix4.Scale(new Vector3(0.5f, 0.5f, 0.5f));
        Matrix4 rotation = Matrix4.AngleAxis(45.0f, 0.0f, 1.0f, 0.0f);
        Matrix4 translation = Matrix4.Translate(new Vector3(5, 3, -4));

        Matrix4 model = translation * rotation * scale;
        GL.MultMatrix(Matrix4.Transpose(model).Matrix);

        DrawCube();
    }
    GL.PopMatrix();
}
```

As you can see i can create matrices by the same name, without reusing the old ones because they are scoped to local code blocks denoted by `{ }` pairs.

The missing functions

One of the nice things about being able to use your own matrices in OpenGL is having to rely less on the OpenGL functions. After all, you don't really know what those functions are doing under the hood! If we want to fully break away from OpenGL we have to implement 3 more functions into our Matrix4 class: Ortho, Frustum and LookAt.

Make these changes in the **Math Implementation** repository, once they are working, copy the Matrix4.cs file into your **OpenGL1X** repository, replacing the old file.

All three of these functions are going to be static! They don't modify any existing matrix, just return a new matrix specifying some sort of transformation!

Deriving matrices is hard. I honestly have no idea how to derive an Orthographic projection matrix. What I will present below is the formulas I've memorized and the things that work for me. If you want to be mathematically accurate, [this tutorial](#) is a decent walkthrough of how to derive the matrices. It's not a skill you will ever need, I've never used it once.

Ortho

How OpenGL makes an orthographic matrix is not a secret. It's actually outlined in the OpenGL specification. Use the following image to create your own `Matrix4.Ortho` function:

$$\begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & \frac{-2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You can test this by changing the projection for Mr.Roboto from the perspective projection to:

```
float aspect = (float)MainGameWindow.Window.Width / (float)MainGameWindow.Window.Height;
GL.Ortho(-25.0f * aspect, 25.0f * aspect, -25.0f, 25.0f, -25.0f, 25.0f);
```

Take note of what it looks like, then replace the `GL.Ortho` call with your own matrix:

```
float aspect = (float)MainGameWindow.Window.Width / (float)MainGameWindow.Window.Height;
Matrix4 ortho = Matrix4.Ortho(-25.0f * aspect, 25.0f * aspect, -25.0f, 25.0f, -25.0f, 25.0f);
GL.LoadMatrix(Matrix4.Transpose(ortho).Matrix);
```

The two screens should look the same.

Frustum

Re-implementing OpenGL's Frustum function is not too difficult either, as the math behind it is thoroughly documented. This matrix is one of the few matrices whose element [3,3] is not 1.

$$\begin{bmatrix} \frac{2\text{near}}{\text{right} - \text{left}} & 0 & A & 0 \\ 0 & \frac{2\text{near}}{\text{top} - \text{bottom}} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$A = \frac{\text{right} + \text{left}}{\text{right} - \text{left}}$$

$$B = \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}}$$

$$C = -\frac{\text{far} + \text{near}}{\text{far} - \text{near}}$$

$$D = -\frac{2\text{far}\text{near}}{\text{far} - \text{near}}$$

The formula is available as text on [the OpenGL man page](#). I know D is a bit hard to read it's `D = -((2 * far * near) / (far - near))`

You can test if your function is correct, by changing the Mr.Roboto sample back to using its original projection matrix. Then replace the `Perspective` function with one that uses your frustum instead of the built in one:

```
public static void Perspective(float fov, float aspectRatio, float zNear, float zFar) {
    float yMax = zNear * (float)Math.Tan(fov * (Math.PI / 360.0f));
    float xMax = yMax * aspectRatio;

    //GL.Frustum(-xMax, xMax, -yMax, yMax, zNear, zFar);
    Matrix4 frustum = Matrix4.Frusum(-xMax, xMax, -yMax, yMax, zNear, zFar);
    GL.MultMatrix(Matrix4.Transpose(frustum).Matrix);
}
```

Look At

LookAt is an interesting beast, as it is not a part of the OpenGL specification. It's not really a part of OpenGL. The LookAt function was made popular by a library called GLU (GL Utilities).

I'm going to walk you through creating the LookAt function step by step both the theory and the code.

We start with the function signature. We've already discussed what these arguments do, I won't discuss them here.

```
public static Matrix4 LookAt(Vector3 position, Vector3 target, Vector3 worldUp) {
```

We can create a vector from the camera to the target by subtracting position from target (to get a vector pointing from position to target). If we normalize this vector to have a magnitude of 1 it becomes the forward basis (z) vector for the cameras coordinate system.

```
    Vector3 cameraForward = Vector3.Normalize(target - position);
```

Remember when you cross two vectors the result is a vector that's perpendicular to both. We can cross the world up vector with the cameras forward to get a vector to the right side of the camera. If we normalize this vector, the result will be our right basis (x) vector.

```
    Vector3 cameraRight = Vector3.Normalize(Vector3.Cross(cameraForward, worldUp));
```

Now that we know the up and the right vector of the camera's coordinate system we need to figure out its up vector. The up is going to be perpendicular to forward and right, so we simply take their cross products. Because both matrices are normalized, we don't need to normalize this.

```
    Vector3 cameraUp = Vector3.Cross(cameraRight, cameraForward);
```

Now that we have all 3 basis vectors that make up the cameras coordinate system, let's combine them into a matrix! Remember, in OpenGL -z is forward, so we have to negate our foward vector.

```
Matrix4 rot = new Matrix4(cameraRight.X, cameraUp.X, -cameraForward.X, 0.0f,
    cameraRight.Y, cameraUp.Y, -cameraForward.Y, 0.0f,
    cameraRight.Z, cameraUp.Z, -cameraForward.Z, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f);
```

That takes care of the rotation, but the camera has a position in the world too! Let's create a translation matrix for it:

```
Matrix4 trans = Matrix4.Translate(position);
```

Now it goes to reason the camera matrix will be `rot * trans`. But remember, the **view matrix** is the **inverse** of the **camera matrix**. So we must invert these matrices before returning them:

```
    return Matrix4.Inverse(rot) * Matrix3.Inverse(trans);
}
```

And thats it! You now have a working LookAt function! Make sure it works by replacing the existing LookAt in MR.Roboto with yours, like so:

```
GL.MatrixMode(MatrixMode.Modelview);
GL.LoadIdentity();
//LookAt(10.0f, 5.0f, 15.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
Matrix4 lookAt = Matrix4.LookAt(
```

```

    new Vector3(10.0f, 5.0f, 15.0f),
    new Vector3(0.0f, 0.0f, 0.0f),
    new Vector3(0.0f, 1.0f, 0.0f)
);
GL.MultMatrix(Matrix4.Transpose(lookAt).Matrix);

```

Faster LookAT

By far the most important part of our LookAT function is this line:

```
return Matrix4.Inverse(rot) * Matrix3.Inverse(trans);
```

Doing one matrix inverse is pretty expensive. Doing two is worse! We know that a translation matrix looks like this:

```

1, 0, 0, Tx
0, 1, 0, Ty
0, 0, 1, Tz
0, 0, 0, 1

```

We also know that its inverse simply looks like this:

```

1, 0, 0, -Tx
0, 1, 0, -Ty
0, 0, 1, -Tz
0, 0, 0, 1

```

So it stands to reason that we can speed this up, if instead of inverting the translation matrix we build it using the inverse of the position vector:

```

Matrix4 trans = Matrix4.Translate(position * -1.0f);
return Matrix4.Inverse(rot) * trans;

```

Hey that's awesome! If we run the game with this look at function everything looks the same. Now, the big question is, can we somehow get rid of that other Inverse? Surprisingly the answer is yes.

The `rot` matrix we have above is VERY special. It's what we call an ortho-normal matrix. An ortho normal matrix is a matrix who's basis vectors are perpendicular to each other, and are all of unit length. Our `rot` matrix meets this criteria.

Now for the special part. If an ortho-normal matrix has NO TRANSLATION (which the `rot` matrix does not), it's inverse is the same as its transpose! Crazy! This is super special case and works in only a few specific instances. This is one of those instances. This means we can rewrite the last two lines as:

```

Matrix4 trans = Matrix4.Translate(position * -1.0f);
return trans * Matrix4.Transpose(rot);

```

And now we have no Inverse functions in the LookAt function at all!

Convenience getter

Having to type out

```
Matrix4.Transpose(frustum).Matrix
```

every time we want to use one of our custom matrices with OpenGL is very time consuming. We can save a LOT of time by adding a simple getter to the matrix class that returns the matrix as a transposed array of floating point numbers. How OpenGL is expecting it:

```
public float OpenGL {
    get {
        return Transpose(this).Matrix;
    }
}
```

There, much nicer. Now if i want to load my own matrix, i just have to:

```
Matrix4 frustum = Matrix4.Frustum(-xMax, xMax, -yMax, yMax, zNear, zFar);
GL.MultMatrix(frustum.OpenGL);
```

Convenience perspective:

We now have a Frustum function in our Matrix class. But to set a Perspective projection we still call the `Projection` helper function. Now that we know how a frustum matrix is made, and we know the math of Projection, why don't we just make a function in Matrix4 that returns a straight up projection and makes it so we don't have to mess with any helper functions?

Let's go ahead and do that. All you really have to do is to move the Perspective helper into the matrix class, and return the final matrix.

You know the drill, test it out by replacing the old function.

Your own stack

At this point in our look at the transformation library, we are no longer dependant on the built in OpenGL matrices! Go us! But we're still using the OpenGL stack. Which as we know has a set size. And we don't really know what it's doing under the hood.

Let's remedy this by making our own matrix stack. I'm going to lay out a skeleton here, you fill in the functions

```
class MatrixStack {  
    protected List<Matrix4> stack = null;  
  
    public MatrixStack() {  
        // TODO: make new stack list  
        // TODO: Add identity onto the stack  
    }  
  
    public void Push() {  
        // Take the top of the stack, make a copy of it  
        // add the new copy onto the stack so it becomes  
        // the new top  
    }  
  
    public void Pop() {  
        // Remove the top of the stack  
    }  
  
    public void Load(Matrix mat) {  
        // Replace the top of the stack with whatever was passed in  
    }  
  
    public void Mul(Matrix mat) {  
        // Multiply the top of the stack with whatever was passed in  
    }  
  
    public float[] OpenGL {  
        get {  
            // Return the OpenGL getter of the top matrix on the stack  
        }  
    }  
}
```

That's all there is to it! The matrix stack could not be any more simple! Let's render two cubes using this stack, see how we would use it.

```
void Render(float width, float height) {  
    GL.MatrixMode(MatrixMode.Projection);  
    Matrix4 projection = Matrix4.Projection(60.0f, width / height, 0.01f, 1000.0f);  
    GL.LoadMatrix(projection.OpenGL);  
  
    GL.MatrixMode(MatrixMode.ModelView);  
    MatrixStack stack = new MatrixStack();  
  
    Matrix4 view = Matrix4.LookAt(  
        new Vector3(-5, 5, -2),  
        new Vector3(0, 0, 0),  
        new Vector3(0, 1, 0)  
    );  
    stack.Load(view);  
  
    // Render cube 1  
    stack.Push();  
    {  
        Matrix4 scale = Matrix4.Scale(new Vector3(0.5f, 0.5f, 0.5f));  
        Matrix4 rotation = Matrix4.AngleAxis(45.0f, 1.0f, 0.0f, 0.0f);  
    }
```

```

Matrix4 translation = Matrix4.Translate(new Vector3(-2, 1, 3));

Matrix4 model = translation * rotation * scale;
stack.Mul(model);
GL.LoadMatrix(stack.OpenGL);
DrawCube();
}

// Restore stack
stack.Pop();

// Render cube 2
stack.Push();
{
    Matrix4 scale = Matrix4.Scale(new Vector3(0.5f, 0.5f, 0.5f));
    Matrix4 rotation = Matrix4.AngleAxis(45.0f, 1.0f, 0.0f, 0.0f);
    Matrix4 translation = Matrix4.Translate(new Vector3(-2, 1, 3));

    Matrix4 model = translation * rotation * scale;
    stack.Mul(model);
    GL.LoadMatrix(stack.OpenGL);
    DrawCube();

    // Render sub-cube
    stack.Push();
    {
        Matrix4 scale = Matrix4.Scale(new Vector3(0.5f, 0.5f, 0.5f));
        Matrix4 rotation = Matrix4.AngleAxis(45.0f, 1.0f, 0.0f, 0.0f);
        Matrix4 translation = Matrix4.Translate(new Vector3(-2, 1, 3));

        Matrix4 model = translation * rotation * scale;
        stack.Mul(model);
        GL.LoadMatrix(stack.OpenGL);
        DrawCube();
    }
    stack.Pop();
}
stack.Pop();
}

```

It's not exactly as convenient as the built in push/pop, but it gets the job done. The biggest difference between what OpenGL does and what we do is that OpenGL no longer knows when you multiplied matrices together. Therefore it becomes important for us to tell OpenGL to use the matrix on the top of the stack for rendering, every time we draw something. Here this bit of code:

```

Matrix4 model = translation * rotation * scale;
stack.Mul(model);
GL.LoadMatrix(stack.OpenGL);
DrawCube();

```

On your own

Try to replace the MatrixStack in the Mr.Roboto project with your own matrix stack.

Solar System

It's time to put all of your knowledge up to this point to the test. To finish CH4 you are going to make a small demo with minimal guidance. And because coding is fun, you get to do it twice!

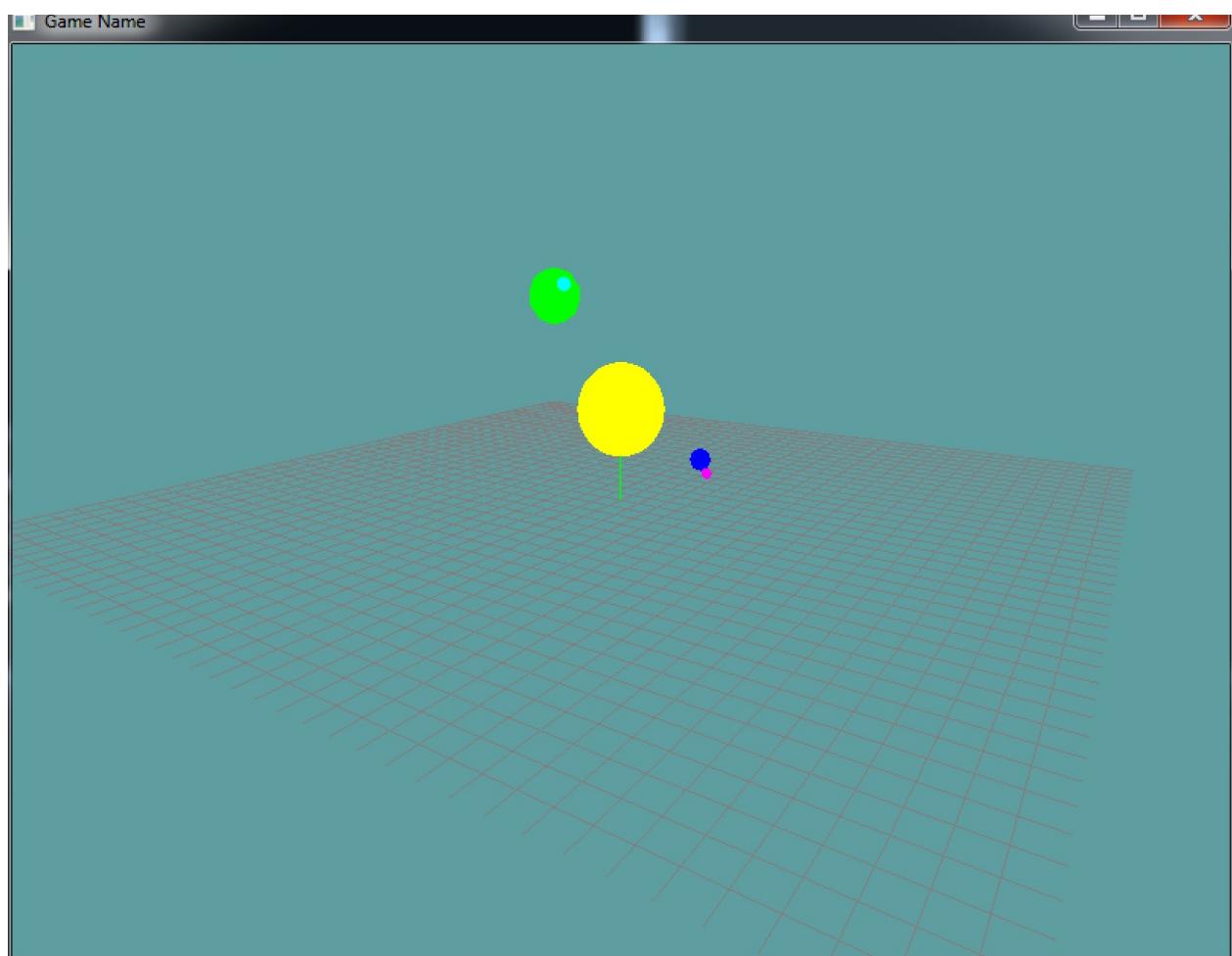
I want you to write the below application two times. They can either look identical, or be different, it's up to you. Once using only OpenGL matrices, and once using only your own matrices. This does mean you need to write two demo classes that extend the game class.

Assignment

I want you to write an application that draws a grid, and on top houses a small solar system. The minimum requirements are this:

- Must have only one sun, this is stationary, it does not move
- Must have minimum two planets, each circling the sun
- Every planet must have at least one moon, circling the planet

Feel free to add more stuff if you want, but those are the minimum requirements. You can [download](#) my version to see how this should work in action. This is what mine looks like:



Drawing a sphere

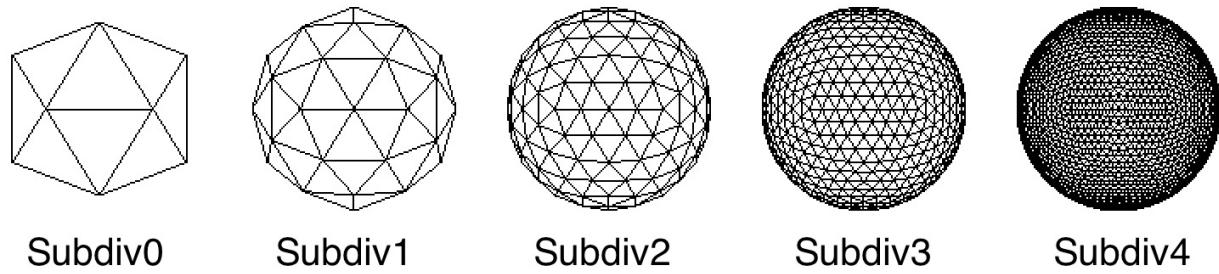
Drawing a sphere is by no means a trivial task. Unlike a cube it's hard to plot down on paper and get the vertex coordinates. The default method of drawing a sphere is to draw a sub-divided icosahedron. 90% of the time the code used to do this copied from [the red book](#).

This is no exception to that rule. I've ported the code from the red-book to C# and we will be using that. Get the code [from here](#), add it to your project.

There is only one static function you need to use:

```
GLGeometry.DrawSphere(int ndiv);
```

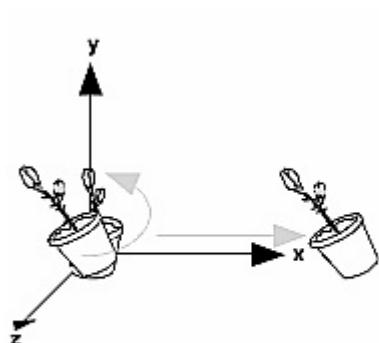
The argument `ndiv` is how many times the sphere should be sub-divided. The more sub-divisions the worse the performance, but the better your scene looks. In my scene i subdivided the sun 3 times, the planets 1 time. Here is the difference between subdiv levels:



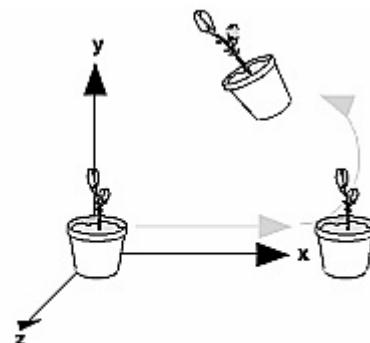
Rotate around things

Rotating around things is a matter of matrix multiplication order. The standard order we use **SRT** rotates an object around its own origin, not around other objects. You can rotate around a point by adding a rotation AFTER we do our translation.

The left of this image is SRT, the right is STR. You want the right.



Rotate then Translate



Translate then Rotate

Structuring your application

Wow, almost done with this chapter! The last thing we need to discuss before wrapping things up is where to put what functions. You may have noticed that the width and height of your window does not change, therefore neither does your projection matrix. Is there any reason to re-calculate it every frame? Not really. So where should this calculation take place?

You only need to set the viewport, and the projection matrix when the window is created or resized. Further-more so long as you set the matrix mode to modelview every time after setting the projection matrix you will not need to select a matrix mode in your render function.

The view matrix by definition needs to be re-calculated each frame. And so does the model matrix. Let's take a look at how all of this fits into your application.

The first thing to do is to add a `Resize` function to your **Game.cs** class. This will be where we set our projection:

```
namespace GameApplication {
    class Game {
        public virtual void Initialize() {

        }
        public virtual void Update(float dTime) {

        }
        public virtual void Render() {

        }
        public virtual void Shutdown() {

        }
        public virtual void Resize(int width, int height) {
        }
    }
}
```

Make sure to call the function in **Program.cs**, like so

```
protected override void OnResize(EventArgs e) {
    base.OnResize(e);
    Rectangle drawingArea = ClientRectangle;
    TheGame.Resize(drawingArea.Width, drawingArea.Height);
}
```

We want to make sure that a default projection matrix is set as soon as the window appears, so we need to change the main function in **Program.cs** to manually call resize as soon as the game is created.

```
[STAThread]
public static void Main(string[] args) {
    //create new window
    Window = new MainGameWindow();
    Axisis = new Grid();
    TheGame = new MrRoboto();
    // -----
    // v This is new
    TheGame.Resize(Window.Width, Window.Height);
    // ^ This is new
}
```

```

// -----
Window.Load += new EventHandler<EventArgs>(Initialize);
Window.UpdateFrame += new EventHandler<FrameEventArgs>(Update);
Window.RenderFrame += new EventHandler<FrameEventArgs>(Render);
Window.Unload += new EventHandler<EventArgs>(Shutdown);

Window.Title = "Game Name";
Window.ClientSize = new System.Drawing.Size(800, 600);
Window.VSync = VSyncMode.On;
//run 60fps
Window.Run(60.0f);

//Dispose at end
Window.Dispose();
}

```

And that's all the infrastructure support we need. From now on, when you make a new demo scene, structure it like this:

```

using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class SimpleScene : Game {
        public override void Resize(int width, int height) {
            GL.MatrixMode(MatrixMode.Projection);
            GL.Viewport(0, 0, width, height);
            // TODO: Set projection matrix
            GL.MatrixMode(MatrixMode.Modelview);
        }

        public override void Initialize() {

        }

        public override void Update(float dTime) {

        }

        public override void Render() {
            // TODO: Set view matrix
            GL.PushMatrix();
            // TODO: Render Scene
            GL.PopMatrix();
        }

        public override void Shutdown() {

        }
    }
}

```

Colors, Lighting, Blending & Fog

Looking at a world of solid colors is confusing. It's hard to judge where objects are and near impossible to tell anything about an object other than its silhouette.

This chapter begins by taking a look at how basic colors work in OpenGL. Then we will move on to more realistic colors using lighting and materials. We will take a quick detour into one method for drawing shadows, then we'll look at how transparency and other effects can be achieved through blending. Finally we will look at OpenGL's built-in fog support.

In this chapter we are going to cover

- Colors in OpenGL
- Shading
- OpenGL lighting
- Light sources
- Materials
- Blending and Transparency
- Fog

Using colors in OpenGL

When you pass primitives to OpenGL, it assigns colors to them by one of two methods: either by using lighting calculation or using the current color. When lighting is used the color for each vertex is computed based on a number of factors, including the position and color of one or more lights, the current materia, the vertex normal and so on. If lighting is disabled, then the current color is used instead.

In RGBA mode, (this is the only mode we are going to be using), OpenGL keeps track of a primary and secondary color consisting of **Red**, **Blue**, **Green** and **Alpha** components. The alternate to RGBA mode is Color Index mode, which has not been used since the mid 90's.

Setting the Color

In RGBA mode, you specify colors by indicating the intensity of the RGBA components. Just because you specify an alpha component doesn't mean OpenGL will draw the shape you are drawing transparent. You need to enable blending for that; something we will discuss later in this chapter.

The color components are usually expressed as a floating point number from 0 to 1, 0 being minimum, 1 being maximum. That means black is (0, 0, 0, 0) and white is (1, 1, 1, 1).

To specify the primary color in OpenGL you will use one of the many variations of:

```
void GL.Color3(...);  
void GL.Color4(...);
```

The one we've been using takes 3 floats as an argument, i suggest you keep using that one. But the function does have many overrides. When using Color3, the alpha value is automatically set to 1. When using Color4, you must specify the alpha value.

Secondary color

In OpenGL 1.4 the concept of a secondary color was introduced. This secondary color was added for a lighting effect known as specular highlights. It never gained much traction, you will probably never use it. But we discuss it for the sake of completeness.

To use the secondary color, you must enable it first

```
GL.Enable(EnableCaps.ColorSum)
```

After it's been enabled, you can set it with

```
GL.SecondaryColor3(...)
```

Shading

So far we've talked about and used GL.Color to set the color of each vertex. But how does OpenGL decide what color to use in the middle of a triangle (not at a vertex)? If all 3 vertices are the same color, then obviously the middle will be that color. But what happens if you use a different color for each vertex? List so:

```
GL.Begin(PrimitiveType.Triangles);
    GL.Color3(1.0f, 0.0f, 0.0f);
    GL.Vertex3(0.0f, 0.5f, 0.0f);

    GL.Color3(0.0f, 1.0f, 0.0f);
    GL.Vertex3(-0.5f, 0.0f, 0.0f);

    GL.Color3(0.0f, 0.0f, 1.0f);
    GL.Vertex3(0.5f, 0.0f, 0.0f);
GL.End();
```

To find out lets simplify the problem. Imagine a line with two different colors. Let's keep things simple, one vertex is black, the other is white. So, what color is the line it's self? Well, that entirely depends on the shading-model being used.

Shading can either be **flat** or **smooth**. When flat shading is used the entire primitive is drawn with a single color.

When using flat shading, each vertex in a primitive can have a different color, OpenGL must chose one of them for the color of the primitives. For lines, triangles and quads the color of the **last** vertex is used. For triangle strips, triangle fans and quad strips the color of the **last** vertex in each sub-triangle is used. For polygons, the color of the **first** vertex is used.

The line we discussed earlier would be white if it was flat shaded. Because the last vertex is white.

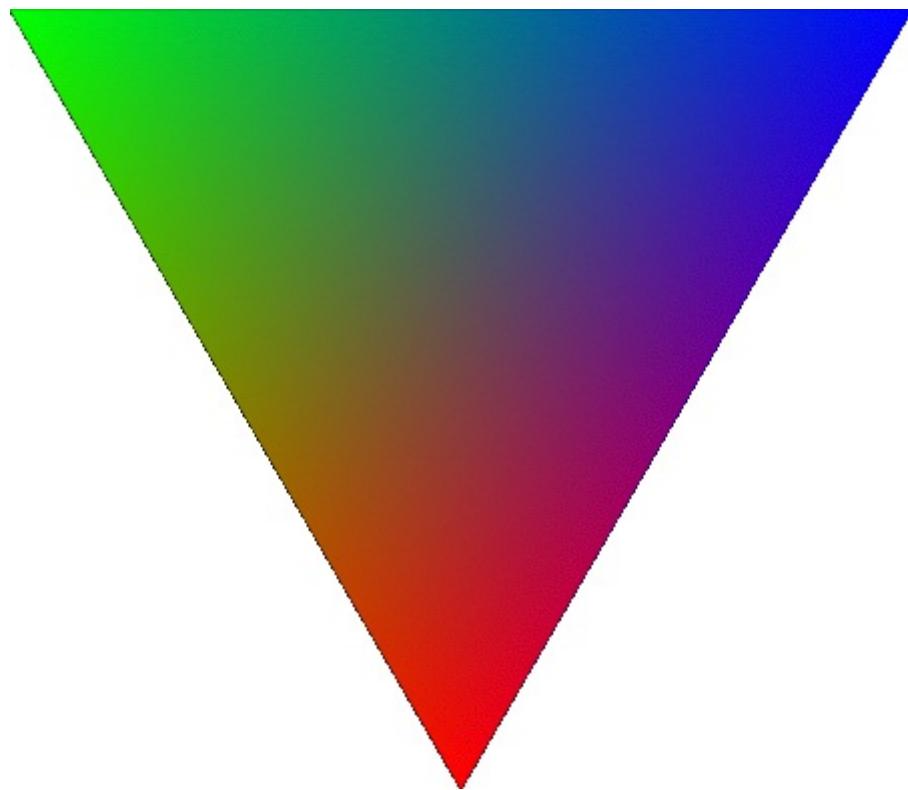
Smooth shading on the other hand is based on the **Gouraud Shading Model**. This is the more realistic of the two shading models. In this mode, colors are interpolated along the face of the primitive.

The line with smooth shading would look like this:



Figure 5.1 Smooth shading of a line with black at the first vertex and white at the second vertex.

Smooth shading on a polygon works the same way as it does for a line. For example, here is a triangle:



Now that you know how these shading models are, how do you use them? The `GL.ShadeModel` function lets you specify the shading model to use. You have to call this before the `GL.Begin` of your geometry.

```
void GL.ShadeModel(ShadingModel);
```

Example

The example below assume the following projection matrix:

```
public override void Resize(int width, int height) {
    GL.Viewport(0, 0, width, height);
    //set projection matrix
    GL.MatrixMode(MatrixMode.Projection);
    float aspect = (float)width / (float)height;
    Matrix4 perspective = Matrix4.Perspective(60, aspect, 0.01f, 1000.0f);
    GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
    //switch to view matrix
    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
}
```

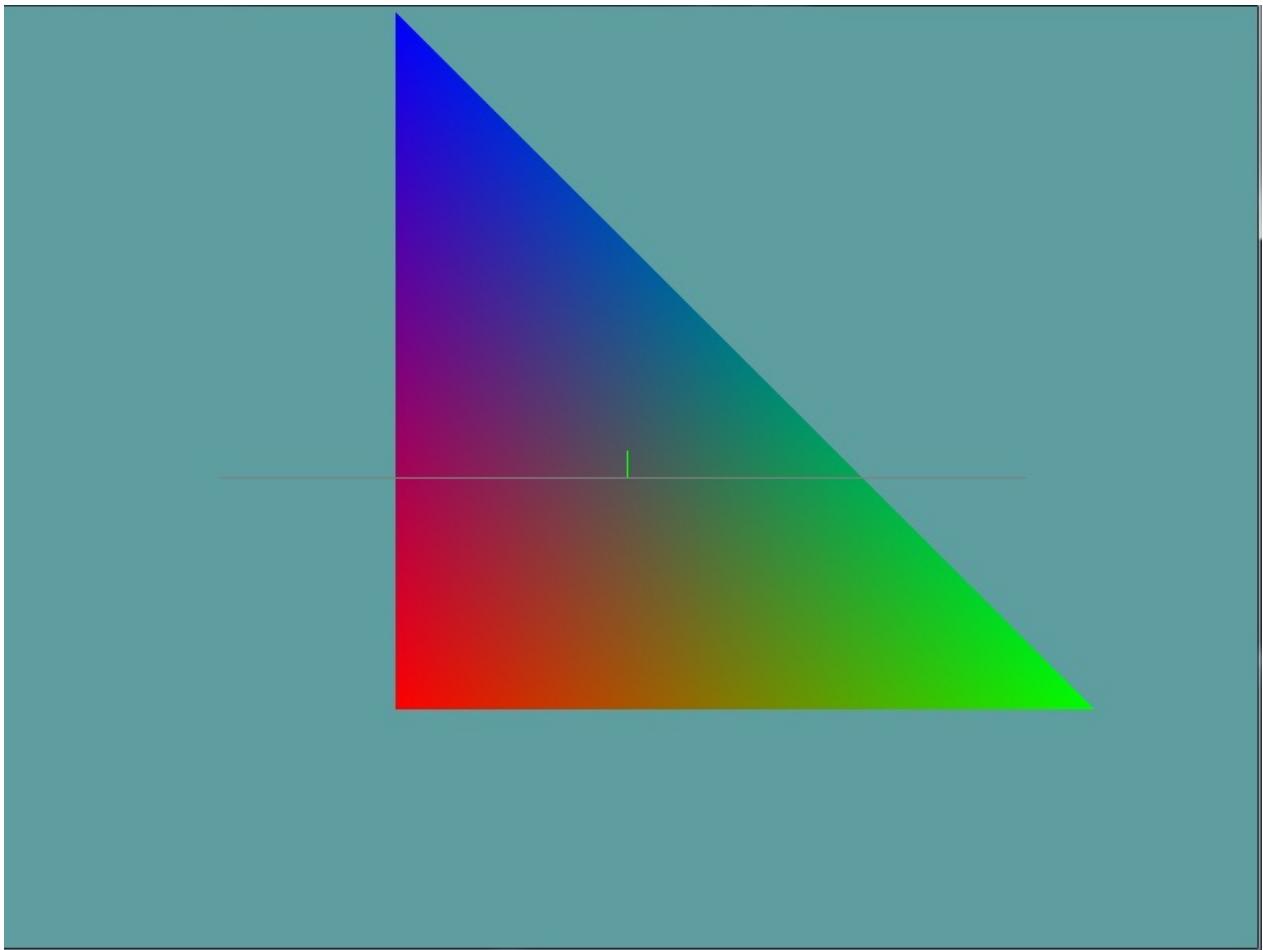
This is bit of code should render a smooth shaded triangle

```
public override void Render() {
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(0.0f, 0.0f, 30.0f), new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f,
    GL.LoadMatrix(lookAt.OpenGL);
    grid.Render();

    // use smooth shading
    GL.ShadeModel(ShadingModel.Smooth);

    // draw our smooth-shaded triangle
    GL.Begin(PrimitiveType.Triangles);
    GL.Color3(1.0f, 0.0f, 0.0f);
    GL.Vertex3(-10.0f, -10.0f, -5.0f); // Red vertex
    GL.Color3(0.0f, 1.0f, 0.0f);
    GL.Vertex3(20.0f, -10.0f, -5.0f); // Green vertex
    GL.Color3(0.0f, 0.0f, 1.0f);
    GL.Vertex3(-10.0, 20.0f, -5.0f); // Blue vertex
    GL.End();
}
```

And this is what it should look like



Try changing the shading model of the triangle from smooth to flat. Try playing around with the colors.

Lighting In OpenGL

You have now arrived at one of the most important aspects of 3D graphics: lighting. It is one of the few elements that can make or break the realism of your 3D game. So far, you've looked at how to build objects, move object, color objects and shade objects. Now let's look at how to make these objects come to life with materials and lights.

OpenGL Lighting and the Real World

Let's take a quick step back and look at a simple explanation of how light works in the real world. Light sources produce photons of many different wavelengths, covering the full spectrum of color. Many of these photons strike objects, which absorb some of them and reflect others. The reflected photons might be reflected uniformly (smooth shiny surfaces) or they may get scattered (rough, matte surfaces). The reflected photons may strike other objects and repeat the process. We are able to see objects because some of these photons eventually enter our eyes.

Modelling the complex interaction of light photons with even a small number of objects (one) is computationally expensive. So much so that it is not feasible to do with an interactive framerate. This type of rendering is referred to as ray tracing, not to be confused with ray-casting which we have done.

OpenGL (and other graphics API's) use simplified lighting models that trade accuracy for speed. Although the results will not match the real world exactly, they are close enough to be believable.

OpenGL calculates lighting by approximating the light into **Red**, **Green** and **Blue** components. This means that the color a light emits is determined by the amount of red, green and blue light it emits.

Light is further broken down into four different terms, which together attempt to simulate the major effects seen in real-world lighting:

- **Ambient Light** simulates the light bouncing between surfaces so many times that the source of the light is no longer apparent. This component is NOT affected by the position of the light or the position of the viewer.
- **Diffuse Light** comes from a certain direction, once it strikes a surface it is reflected equally in all directions. The diffuse light component IS affected by the position (direction) of the light, but NOT the position of the viewer.
- **Specular Light** is directional and is reflected off a surface in a particular direction. Specularity is often referred to as shininess. The specular term is effected by BOTH the position (direction) of the light AND the viewer.
- **Emissive Light** is a cheap way to simulate objects that emit light. OpenGL will not actually use the emissive term to illuminate nearby objects, it simply causes the emissive object to be more intensely lit.

The final results of lighting depend on several major factors, each of which will be discussed in detail later. The factors are:

- One or more light sources. Each light source will have the ambient, diffuse, specular and emissive terms listed above. Each specified as RGBA values. In addition they will either have a position or direction, or have terms that affect the attenuation and may have a limited area of effect.
- The orientation of surfaces in the scene. This is determined through the use of normals, which are associated with each vertex.
- The material each object is made of. Material properties define what percentages of the RGBA values each light term should be reflected. They also define how shiny a surface is.
- The lighting model, which includes a global ambient term (independent of any light source) and whether or not the position of the viewer has an effect on lighting calculations.

When the light strikes a surface, OpenGL uses the material of the surface to determine how much R, G and B light should be reflected by the surface. Even though they are approximations, the equations used by OpenGL can be computed very quickly and produce reasonably good results.

Light sources

It's time to turn some lights on and get on with the show! Like most GL.Enable calls, you want to enable lighting in the initialize of your program.

You can enable lighting like so:

```
GL.Enable(EnableCap.Lighting);
```

This call causes the lighting equation to be applied to every vertex, but it does not turn on any lights. You need to explicitly turn lights on by calling:

```
GL.Enable(EnableCap.LightX);
```

Where X is a number between 0 and 7 (inclusive). The OpenGL specification dictates that OpenGL will support a minimum of 8 lights. Some implementations support more, but 8 is the safe number.

Only 8?

8 lights might not seem like a lot. That's because 8 lights is not a lot. It's a very conservative number. This is largely because lighting is expensive. Enabling just 4 lights might cause a noticeable drop in framerate.

You've no doubt seen games (Even those built on OpenGL 1X, like Doom3) that use way more than 8 lights! How do they do it? Well, if you are standing in the living room, the living room light has an effect on you. But the bedroom light does not.

Lights are enabled and disabled PER OBJECT. This means object A can be drawn with 2 lights, object B can be drawn with 6 and object c can be drawn with 3. Assuming each light is unique, we just drew a scene with 11 lights. But we didn't break OpenGL, because each individual object only enabled the lights that effect it.

Light properties

Each world light has several properties associated with it that define its position or direction in the world, the colors of its ambient, diffuse, specular and emissive terms and whether the light radiates in all directions or shines in a specific direction.

These parameters are configured through the following functions:

```
void GL.Light(LightName name, LightParameter param, float val);
void GL.Light(LightName name, LightParameter param, float[] val);
```

The first argument is an enumeration of type `LightName`, it's going to have a value of `LightName.Light0` through `LightName.Light7`. This argument specifies which light's properties you are modifying.

The second argument is an enumeration of type `LightParameter` which tells OpenGL which value is being modified.

These are the valid values:

- **Ambient** Ambient intensity of the light
- **Diffuse** Diffuse intensity of the light
- **Specular** Specular intensity of the light
- **Position** Position of the light as a vector4 (x, y, z, w)
- **SpotDireriton** Directon of the spot light as a vector3 (x, y, z)
- **SpotExponent** Spotlight exponent
- **SpotCutoff** Spotlight cutoff
- **ConstantAttenuation** Constant attenuation
- **LienarAttenuation** Linear atenuation
- **QuadraticAttenuation** Quadratic attenuation

Lets discuss each of these settings briefly before actually using them in code.

Position and Direction

Each light can have either a position or a direction. Lights with a position are often called **positional lights** or **point lights**. **directional lights** on the other hand are infinitly far away, they have no position. There are no true directional lights in nature, since nothing is infinitly far away. But some light sources are so far away they can be treated as directional lights (like the sun).

So why use a directional light instead of a really big point light (like how the sun actually works)? Performance. Directional lights are (marginally) cheaper than spot lights. Admitedly, on modern hardware this is no longer a problem.

You can set a lights position by passing `LightParameter.Position` to the `GL.Light` function. The function will then take a 4 element array of floats (x, y, z, w) representing either the lights position or direction. The w component is used to indicate if the vector passed in is a position or a direction vector. A w of 0 is directional, a w of 1 is positional.

Here is how you could set the direction of a light

```
// direction must be normalized
float[] lightDir = { 0.5f, 0.5f, 0.0f, 0.0f};
// We pass in a direction vector, this is going to be a directional light
GL.Light(LightName.Light0, LightParamater.Position, lightDir);
```

If i wanted to set up a positional or point light

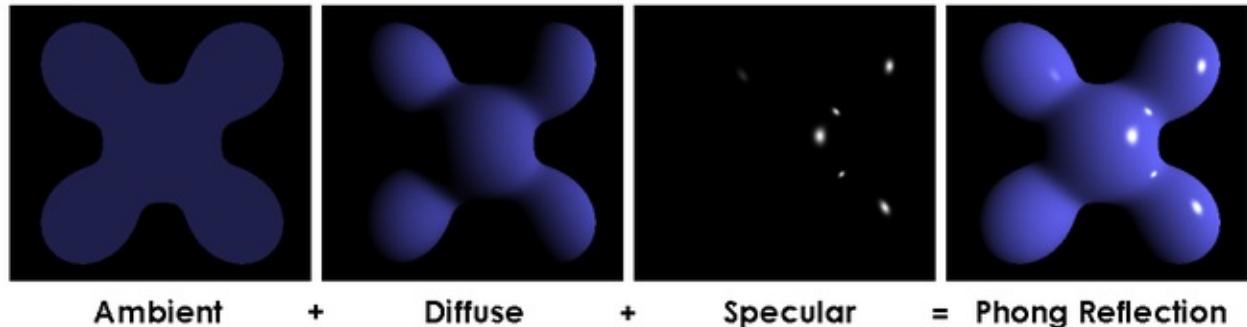
```
float[] lightDir = { 2.5f, 3.5f, 7.0f, 1.0f};
// We pass in a position vector, this is going to be a point light
GL.Light(LightName.Light0, LightParamater.Position, lightDir);
```

The default position for all lights is (0, 0, 1, 0), it is directional, pointing down the negative Z axis.

IMPORTANT: Whenever you make a call to `GL.Light` with `LightParamater.Position` the vector you specify is transformed by the current **modelview** matrix, just as vertices are, and stored in eye coordinates. We will discuss this in more detail later.

Light Color

Light sources are composed of three of the lighting terms we discussed earlier: ambient, diffuse and specular. These 3 terms combined make up the final lighting of the geometry. Illustrated in this image:



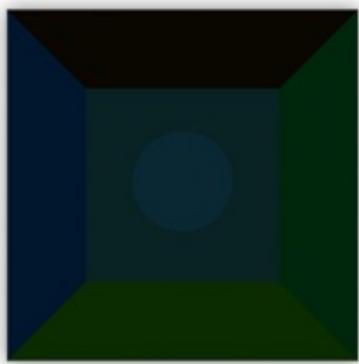
To set up each of these terms you call `GL.Light` with a parameter name of `LightParameter.Ambient`, `LightParameter.Diffuse` or `LightParameter.Specular`. Each of these takes a four component float array. The values passed in represent the RGBA color of the specified term. The following code demonstrates:

```
float[] white = new float[]{1, 1, 1, 1};  
float[] blue = new float[]{0, 0, 1, 1};  
  
// Set ambient term blue  
GL.Light(LightName.Light0, LightParameter.Ambient, blue);  
// Set diffuse term blue  
GL.Light(LightName.Light0, LightParameter.Diffuse, blue);  
// Set specular term white  
GL.Light(LightName.Light0, LightParameter.Specular, white);
```

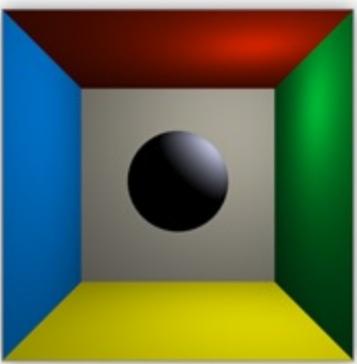
The above code and image match. Note how in the image the ambient and diffuse terms are both blue, and the specular term (the highlight) is white. Adding all the terms up produces the final image.

The default value of every term for all lights is black (0, 0, 0, 1) with two exceptions: Light 0 has a default diffuse and specular term of white (1, 1, 1, 1)

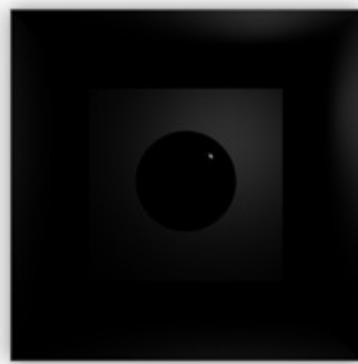
Because this concept is pretty important, here is another image demonstrating it:



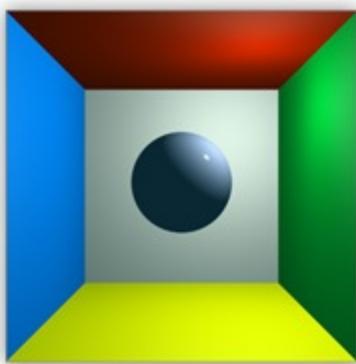
Ambient



Diffuse



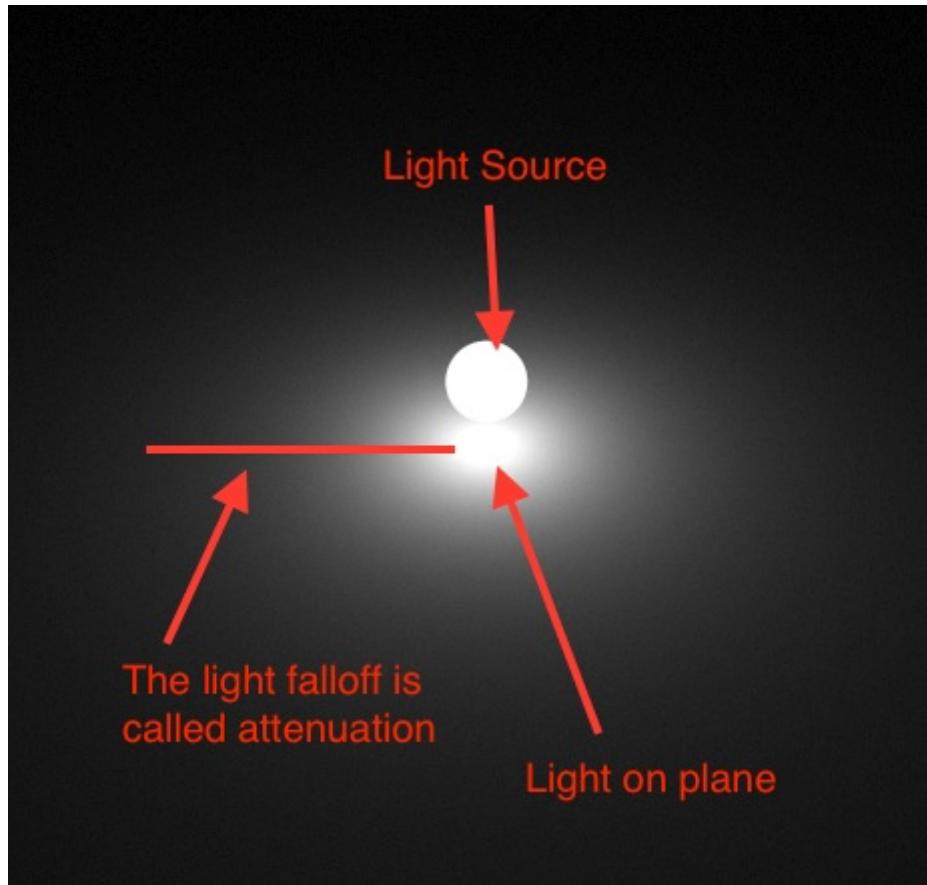
Specular



Ambient + Diffuse
+ Specular

Attenuation

In the real world, the farther an object is away from a light, the less effect that light has on the object. For example, if you look at a street lamp at night (especially in the fog) you'll be able to see the intensity of the light dropping off away from the lamp. This phenomenon is known as *attenuation*.



This effect is modelled by OpenGL using an attenuation factor, which can reduce the effect of a light's contribution to the color of an object based on the distance to the object. The attenuation factor is calculated as follows:

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

d = distance between the light's position and the vertex

k_c = GL_CONSTANT_ATTENUATION

k_l = GL_LINEAR_ATTENUATION

k_q = GL_QUADRATIC_ATTENUATION

Each factor has a default of (1, 0, 0) which results in no attenuation by default. You can change the factors by passing `LightParamater.ConstantAttenuation`, `LightParamater.LinearAttenuation` and `LightParamater.QuadraticAttenuation` to the `GL.Light` function. Each of these attenuation factors takes a single floating point number as an argument.

This sample code sets all attenuation factors

```
GL.Light(LightName.Light0, LightParameter.ConstantAttenuation, 4.0f);
GL.Light(LightName.Light0, LightParameter.LinearAttenuation, 1.0f);
GL.Light(LightName.Light0, LightParameter.QuadraticAttenuation, 0.25f);
```

Attenuation factor affects only positional light sources, it makes no sense in terms of a directional light as the directional light is infinitately far away.

Not intuitive

OpenGL chose this attenuation model because it is pretty close to how nature actually looks / works. The problem is they lost all intuition with this formula. When setting a point light i expect to be able to say "I want a light with a radius of 5, attenuation should start 2 units away from the center".

There is no intuitive way to set the attenuation of a point light. You have to play around with it until it looks the way you want it to. It's actually so bad, i built an application that renders a sphere on a plane with an attenuated spot light and has sliders for all factors. I play with the sliders until it looks good and copy the values into my code

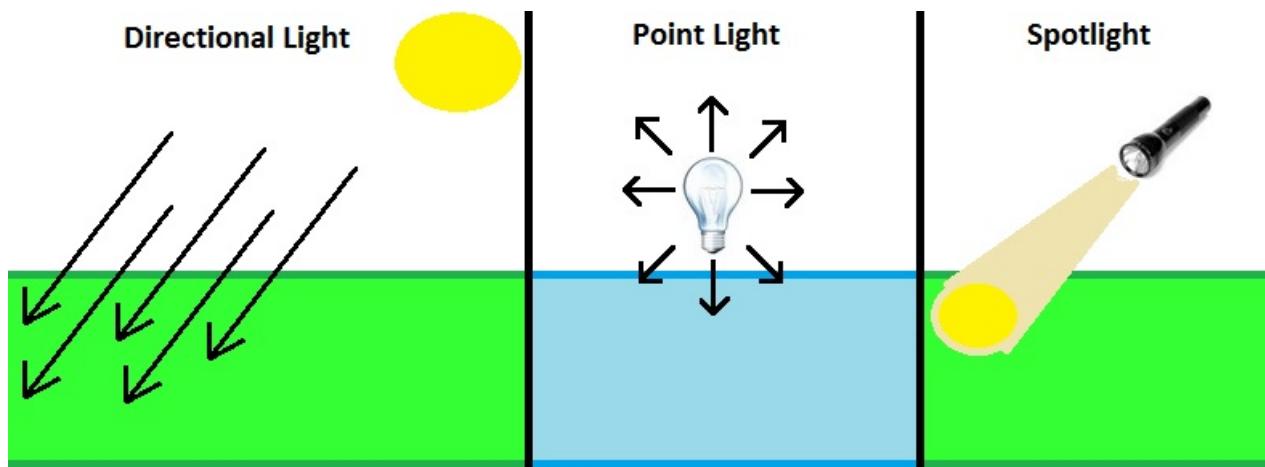
Spotlights

Normally positional lights radiate in all directions. However you can limit the effect of the light to a cone. This would look like a flash light, the effect is called a *spotlight*.

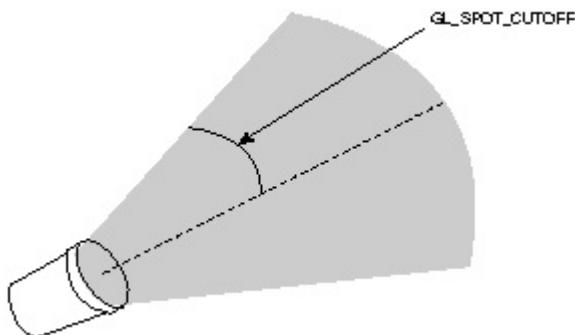
To create a spotlight, you set up a positional light like you normally would and then set a few spotlight-specific parameters: cutoff, direction and focus.

Cutoff

Let's think about what a spotlight looks like for a moment. If you were looking at a spot light in pure darkness, you would see that the light creates a cone of light in the direction that the spotlight is pointing.



With OpenGL, you can define how wide this cone of light should be by specifying the angle between the edge of the cone and its axis by passing `LightParameter.SpotCutoff` to `GL.Light`.



A cutoff value of 10 for example results with a spotlight whose cone spreads out 20 degrees. OpenGL accepts only values between 0 and 90 for this parameter, with the special exception of 180. 180 is the default value and turns the spotlight into a regular point light.

This is the code you would use to create a spotlight with a 30 degree cone

```
GL.Light(LightName.Light0, LightParameter.SpotCutoff, 15.0f);
```

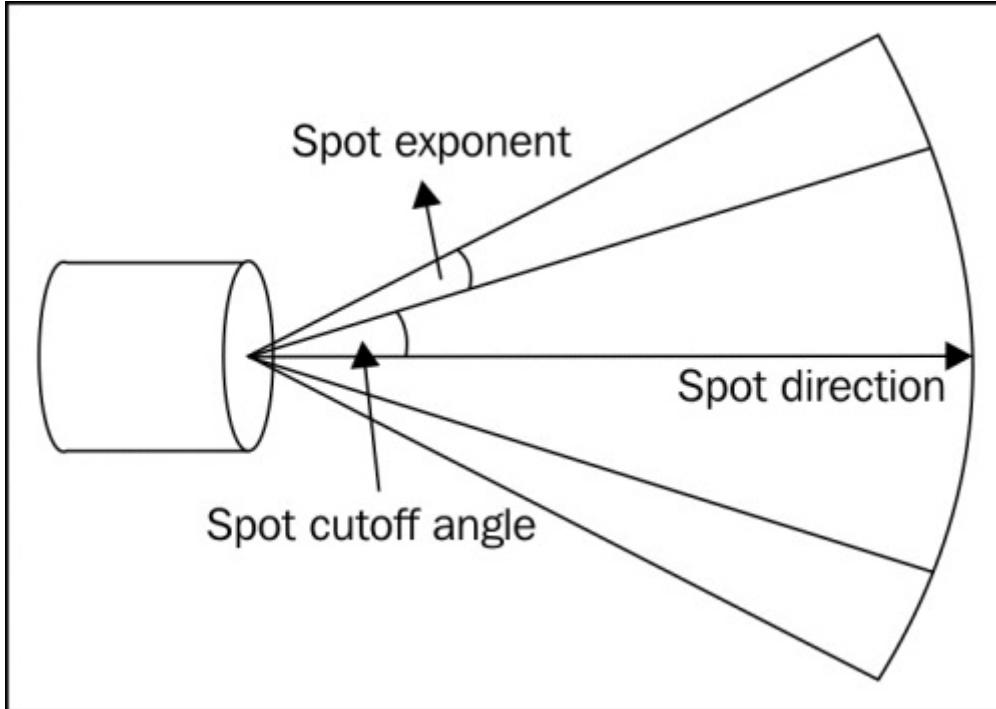
Direction

The next parameter to specify is the direction that the spotlight is facing. This is done with the `LightParameter.SpotDirection` parameter, which takes a 3 component vector (x, y, z). The default direction is (0, 0, -1), which points the spotlight down the negative z axis. You can specify your own direction like so:

```
// Direction vector is normalized
float[] direction = {0, 0.5, -0.5};
GL.Light(LightName.Light0, LightParameter.SpotDirection, direction);
```

Exponent

And finally you can specify the focus of the spotlight, which can be defined as the concentration of the spotlight in the center of the light cone. As you move away from the center of the cone the light is attenuated until there is no more light at the edge of the cone.



You can use `LightParameter.SpotExponent` to control this. A higher exponent results in a more focused light, that drops off quickly. Here is an example of setting the exponent to 10:

```
GL.Light(LightName.Light0, LightParameter.SpotExponent, 10.0f);
```

The spot exponent can range from 0 to 128. A value of 0, which is the default results in no attenuation, so the spot light is evenly distributed

Usage

How often are spot lights used? Not that much. Even modern games, like Assassins Creed tend to use point lights for things like street lamps. The point light is cheaper calculation wise (not by much) and easier for artists to configure.

Moving and rotating

What do you need to do to make a light move around? Think about how you would make any other object in the world move around. One way is to set the position of the object after you translate or rotate it. You can do the same thing with lights. When you call `GL.Light` to define the position or direction of a light, the information you specify is in local space. It will be transformed by the current modelview matrix.

Static Lights

For static lights, lights that never move it makes sense to position the light after you set up the camera (after calling `LookAt` for example) but before applying any other transformations to the modelview matrix. We rendered our reference grid like this. No matter where the objects in the scene or the camera moved, the reference grid is static at origin. Same thing would happen for lights.

Perspective Lights

How about perspective lights? What's a perspective light? It's a light that stays fixed relative to the eye (camera) position. For instance, when you sit in a car, the headlights are perspective lights. They stay in more or less the same position relative to your eyes. In this case you would set the modelview matrix to identity, then define your light at the origin, and then continue transformations as you normally would

```
GL.MatrixMode(MatrixMode.ModelView);
GL.LoadIdentity();

// Position the cars lights at origin, this can be anywhere,
// it doesn't have to be at origin. In the real world it would
// be up a few units on the z axis, and down a few on the y axis
float[] pos = {0,0,0,1}
GL.Light(LightName.Light0, LightParamater.Position, pos);

// Continue rendering as normal
Matrix4 lookAt = Matrix4.LookAt(eyePosition, lookTarget, new Vector3(0.0f, 1.0f, 0.0f));
GL.MulMatrix(lookAt.OpenGL);
```

Dynamic Lights

But what if you had a moving object, like a flash-light. It would need to be a child of a characters arm if it's being held (think of Mr.Roboto and how we did his feet).

```
GL.MatrixMode(MatrixMode.ModelView);
GL.LoadIdentity();

Matrix4 lookAt = Matrix4.LookAt(eyePosition, lookTarget, new Vector3(0.0f, 1.0f, 0.0f));
GL.MulMatrix(lookAt.OpenGL);

// Render character
GL.PushMatrix(); // Push Torso
    GL.Translate(10, 0, 0);
    GL.Rotate(10, 0, 1, 0);
    GL.Scale(1, 1, 1);
    DrawTorso();
    GL.PushMatrix(); // Push Arm
        GL.Translate(0, 5, 0);
        GL.Rotate(2, 1, 0, 0);
        GL.Scale(1, 2, 1);
        DrawArm();
GL.PopMatrix();
```

```
// The arm is drawn, we need to nest the flashlight here.  
// But we want to draw it at the bottom of the arm  
// so we need to translate down a bit  
GL.PushMatrix(); // Push Flashlight  
    GL.Translate(0, 2, 0);  
    // No need to rotate for this example  
    GL.Scale(1, 0.5, 1);  
  
    // Notice we set the light position and direction here!  
    // All other factors can be set elsewhere, but these two  
    // rely on world position data, we MUST set them here!  
    GL.Light(LightName.Light0, LightParamater.Position, lightPosition);  
    GL.Light(LightName.Light0, LightParamater.SpotDirection, spotDirection);  
  
    GL.PopMatrix(); // Pop Flashlight  
  
    GL.PopMatrix(); // Pop Arm  
    // ... Render rest of character  
    GL.PopMatrix(); // Pop Torso
```

Exercises

The last section covered a LOT of ground about lighting, what it is and how it works. But i don't feel like it demonstrated clearly how to actually use lights. That's what this chapter is all about. In this chapter we're going to create several demo scenes, some together, some on your own to create some fully lit scenes

Changing the grid

Before we get started, there are a few things we need to change within our code to get well lit objects. First, in order to actually view lighting, it helps to have a solid ground plane. Let's modify the grid class to optionally draw a solid plane.

First, let's add a public boolean that is going to control the render mode. By default drawing a filled in grid is going to be off (So that we don't break any of the example scenes we've already written)

```
namespace GameApplication {
    class Grid {
        // Control if the grid is rendered solid or not
        public bool RenderSolid = false;

        public Grid() {
            RenderSolid = false;
        }

        public Grid(bool solid) {
            RenderSolid = solid;
        }
    }
}
```

Now, let's modify the render function to actually render the solid grid. Remember, our grid is 20 units wide, and centered around the 0 point. So the far left is -10, the far right is 10. We're going to draw one large quad to cover this area. The quad is going to be a dark gray.

We're going to draw the quad at -0.01f on the Y axis. We do this because i still want the grid to render, this way the quad renders just underneath the grid and will look pretty good.

Lets add this to the begenning of the render function:

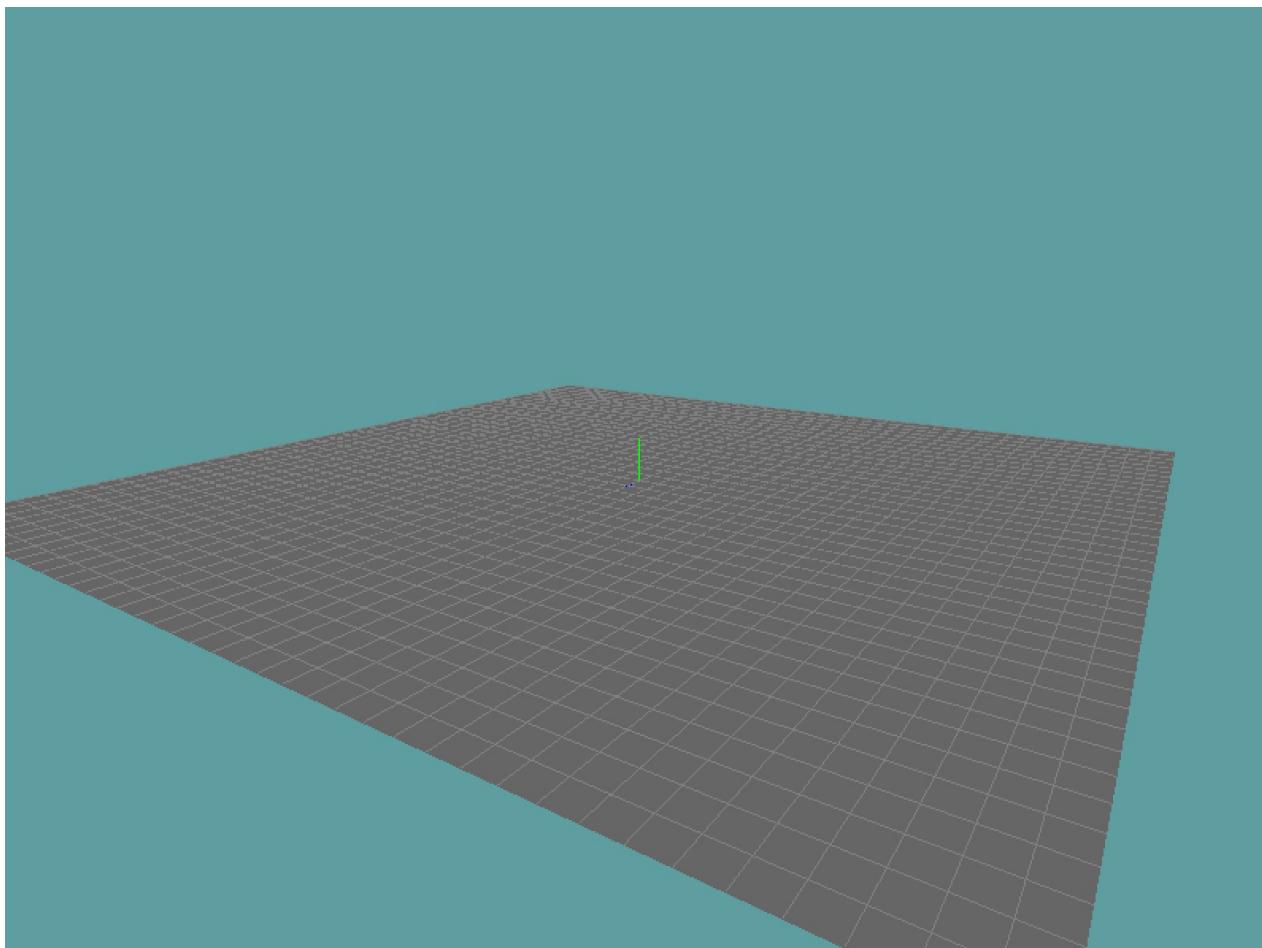
```
public void Render() {
    // Only render this bit if solid is enabled
    if (RenderSolid) {
        // Set the color to a dark gray
        GL.Color3(0.4f, 0.4f, 0.4f);
        // Draw one large rectangular primitive
        // this large rectangle is 2 triangles ;)
        GL.Begin(PrimitiveType.Triangles);
        GL.Normal3(0.0f, 1.0f, 0.0f);
        // Triangle 1
        GL.Vertex3(10.0f, -0.01f, 10.0f);
        GL.Vertex3(10.0f, -0.01f, -10.0f);
        GL.Vertex3(-10.0f, -0.01f, -10.0f);
        // Triangle 2
        GL.Vertex3(10.0f, -0.01f, 10.0f);
        GL.Vertex3(-10.0f, -0.01f, -10.0f);
        GL.Vertex3(-10.0f, -0.01f, 10.0f);
        GL.End();
    }
    // ... The rest of the render function stays the same
}
```

The only thing that's kind of unfamiliar in there is this bit:

```
GL.Normal3(0.0f, 1.0f, 0.0f);
```

The normal of a surface is used in lighting calculations. It tells OpenGL which way an object is facing. A normal is just a unit vector that points in the direction that the primitive faces. We will talk in depth about normals soon.

That's all the changes we need to make to the grid. If you run one of the existing samples, and turn on `RenderSolid` on, your scene should look something like this:



Additional Geometry

You added the `DrawCube` function to every program that uses it. And you added the function `DrawSphere` to a class called `Circle`. There is nothing wrong with this, it's a solid approach. The circle code is better than the cube code for the sake that it is more portable.

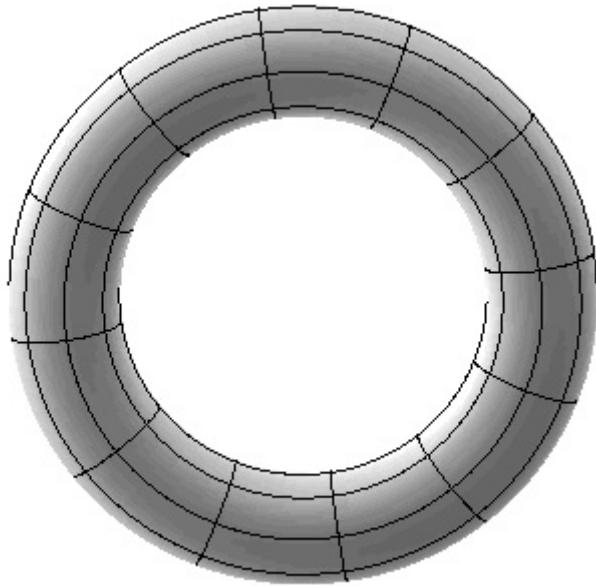
However, neither solution is robust. Ideally you will have many different primitives to render, so we need a class that is a convenient holder for them. I would call this class either `Primitives` OR `Geometry`. Go ahead and create this class in your project.

Add the code found [at this gist](#) to the class. Don't copy any of the existing code in your project, the gist is special in that it adds normal information (we will cover this later) for each primitive. The following static functions are exposed

for you:

```
public static void Geometry.Cube();
public static void Geometry.Sphere(int nDiv = 2);
public static void Torus(float ringRadius = 0.2f, float tubeRadius = 0.8f, int numc = 6, int numt = 12)
```

As you can see, every argument has a default value, so you can call these functions without arguments. Cube and Sphere work like they always have. I've added some sensible defaults. Torus is new, what is a torus? It's essentially a doughnut. This is what a torus looks like:



When calling the function, the first argument is the radius of the ring, that is the radius of the actual tube which makes up the ring. The second argument is the radius of the tube. Imaging, that the ring is stretched around a tube, this is the radius we define here. So, first argument is how thick the geometry is, second argument is how large the hole in the center is.

The last two arguments are the number of segments to subdivide. The first one decides how many slices to give, the second one how many columns. Like with the sphere, the larger these numbers, the more detail the torus will have.

The test scene

With that we now have all the elements we are going to be using for our lighting test scene. For every lighting demo from here on, copy this scene, rename it and implement the light. For the scene, we're going to render a cube a sphere and a torus on top of a grid. And the camera is going to be slowly orbiting the scene to give us a 360 degree view of the lighting.

Let's build out the test scene, we're going to start with the initialize method:

```
namespace GameApplication {
    class LightingScene : Game{
        Grid grid = null;

        public override void Initialize() {
            grid = new Grid(true);
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
```

```

        Resize(MainGameWindow.Window.Width, MainGameWindow.Window.Height);
    }

```

The first thing we do here is make a new solid grid object. We enable DepthTest, because we want the 3D objects to sort properly. We enable CullFace to save some performance by not rendering back-facing triangles. Finally, we call the Resize method, to set up our projection and viewports. Let's take a look at said resize method next:

```

public override void Resize(int width, int height) {
    GL.Viewport(0, 0, width, height);
    float aspect = (float)width / height;

    GL.MatrixMode(MatrixMode.Projection);
    Matrix4 perspective = Matrix4.Perspective(60.0f, aspect, 0.01f, 1000.0f);
    GL.LoadMatrix(perspective.OpenGL);
    GL.MatrixMode(MatrixMode.Modelview);
}

```

This is a pretty standard resize method. Nothing to really talk about here. And finally lets take a look at how we are going to be rendering the scene:

```

public override void Render() {
    Vector3 eyePos = new Vector3();
    eyePos.X = 0.0f;
    eyePos.Y = 5.0f;
    eyePos.Z = 10.0f;

    Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
    GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

    grid.Render();

    GL.Color3(0f, 1f, 0f);
    GL.PushMatrix();
        GL.Translate(0.0f, 2.5f, -2f);
        Primitives.Torus();
    GL.PopMatrix();

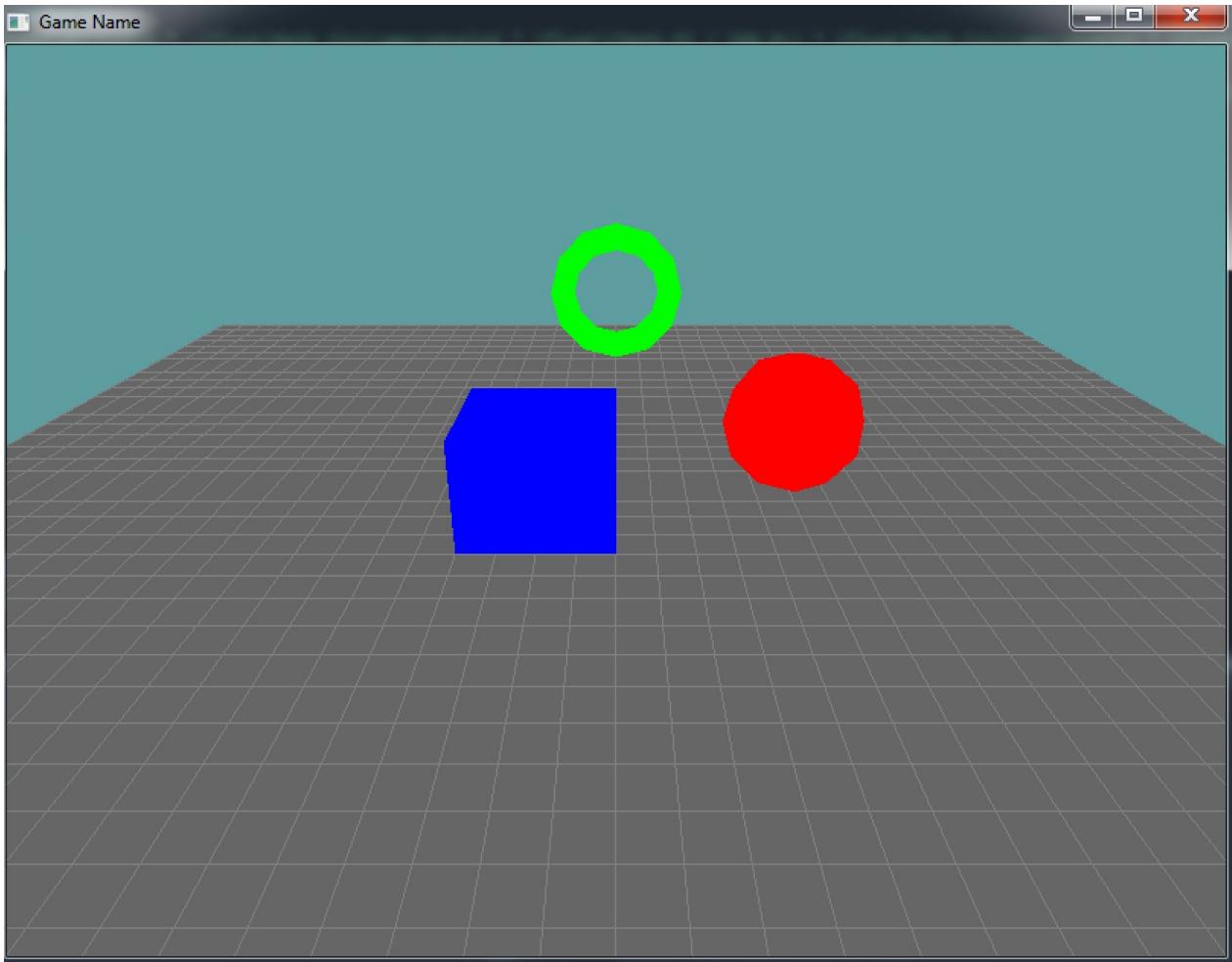
    GL.Color3(1f, 0f, 0f);
    GL.PushMatrix();
        GL.Translate(2.5f, 1.0, -0.5f);
        Primitives.DrawSphere();
    GL.PopMatrix();

    GL.Color3(0f, 0f, 1f);
    GL.PushMatrix();
        GL.Translate(-1f, 0.5f, 0.5f);
        Primitives.Cube();
    GL.PopMatrix();
}

}

```

Again, nothing to brag about here. We set up a pretty standard view matrix, then render the grid, and then translate and render each primitive using their default values. The torus renders green, the sphere red and the cube blue. Your screen should look like this:



Now, let's rotate the camera! To orbit the camera around the scene, let's first add 3 variables to our program: angleX, angleY and distance:

```
namespace GameApplication {
    class LightingScene : Game{
        Grid grid = null;

        float cameraAngleX = 0.0f;
        float cameraAngleY = -25;
        float cameraDistance = 10.0f;

        public override void Initialize() {
            // ... rest of code unchanged
```

The cameras distance from what it's looking at isn't going to change, neither is the height at which the camera is looking from (cameraAngleY). But we're going to rotate the camera around the Y axis. For this, override the update method:

```
public override void Update(float dTime) {
    cameraAngleX += 30f * dTime;
}
```

Now that the angle is updating properly, lets modify the render code to respect these values. The following code should look familiar, it's standard rotation code that you've derived before.

```
public override void Render() {
    Vector3 eyePos = new Vector3();
    eyePos.X = cameraDistance * -(float)Math.Sin(cameraAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(cameraAngleY * (float)(Math.PI / 180.0));
    eyePos.Y = cameraDistance * -(float)Math.Sin(cameraAngleY * (float)(Math.PI / 180.0));
    eyePos.Z = -cameraDistance * (float)Math.Cos(cameraAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(cameraAngleY * (float)(Math.PI / 180.0));

    Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
    // ... rest of code unchanged
```

The only thing new here is that we are multiplying the rotated angle by cameraDistrnace. That's because our angle rotation returns between 0 and 1 (radians). If we multiply that by the distance of the camera, it makes the camera orbit around an imaginary sphere.

Try running the code, and confirm that the camera is rotating the scene. After this we are done, and ready to actually start implementing some lights!

Directional lights

Directional lights are perhaps the easiest to program. They are simply a direction that the light comes from, like sunlight. Let's give it a go.

Follow along, make a new scene for each sub-section here. For this (Directional Light) section alone you should have 6 test scenes.

Adding one light

The first thing we need to do is enable lighting and light 0. This isn't necessarily something that needs to happen every frame, so let's handle it in the initialize function.

```
public override void Initialize() {
    grid = new Grid(true);
    GL.Enable(EnableCap.DepthTest);
    GL.Enable(EnableCap.CullFace);
    Resize(MainGameWindow.Window.Width, MainGameWindow.Window.Height);

    GL.Enable(EnableCap.Lighting);
    GL.Enable(EnableCap.Light0);
}
```

Now we are ready to do some lighting! I'm going to set light 0 to be a blue light coming from the upper front of the scene. For now, the light is static (it doesn't change from frame to frame) so it's safe to configure in the Init function as well.

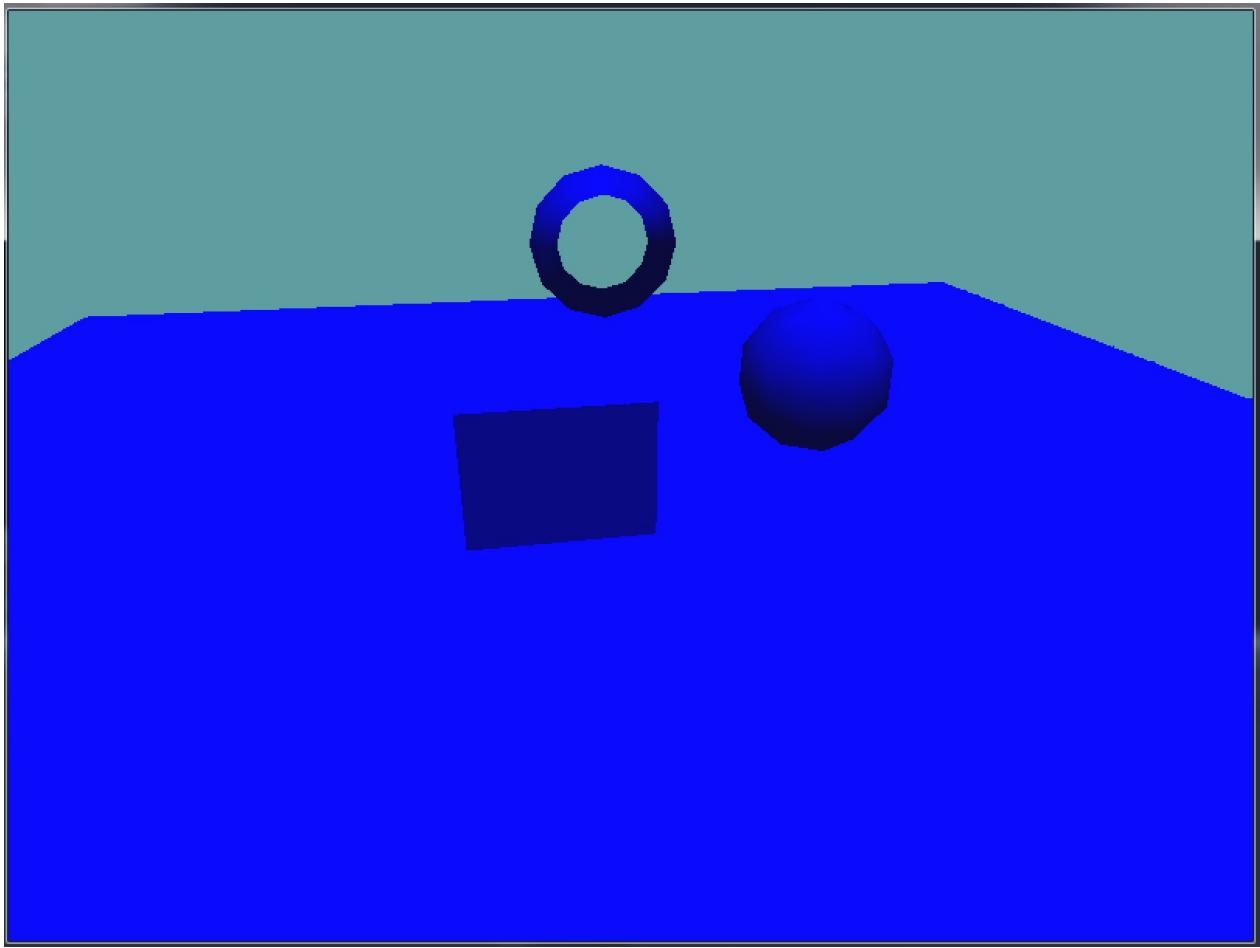
```
public override void Initialize() {
    grid = new Grid(true);
    GL.Enable(EnableCap.DepthTest);
    GL.Enable(EnableCap.CullFace);
    Resize(MainGameWindow.Window.Width, MainGameWindow.Window.Height);

    GL.Enable(EnableCap.Lighting);
    GL.Enable(EnableCap.Light0);

    float[] white = new float[] { 0f, 0f, 0f, 1f };
    float[] blue = new float[] { 0f, 0f, 1f, 1f };

    GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0.0f, 0.5f, 0.5f, 0.0f } );
    GL.Light(LightName.Light0, LightParameter.Ambient, blue );
    GL.Light(LightName.Light0, LightParameter.Diffuse, blue );
    GL.Light(LightName.Light0, LightParameter.Specular, white );
}
```

That's all there is to it. If you run your code, it should look like this:

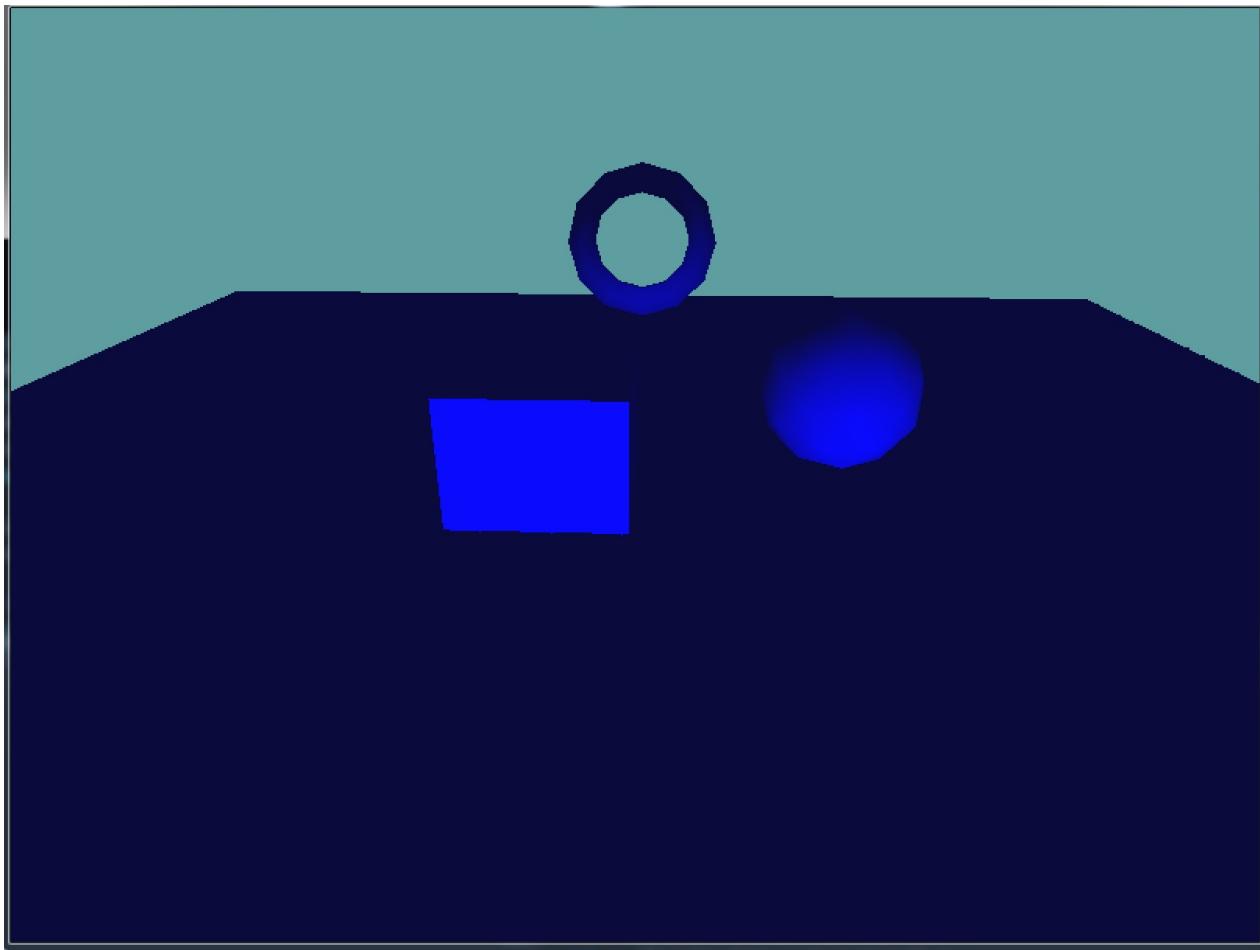


Light direction

It may not be obvious from the previous example, but OpenGL shades (lights) objects independently! That is, no object casts a shadow on another object. When the light is coming from the top, like in the above scene it's easy to miss that. So, let's change the light direction to come from the negative Y axis!

```
GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0.0f, -0.5f, 0.5f, 0.0f } );
```

Let's take a look at what that looks like:



As you can see the shading on the primitives has changed, the floor is the most noticeable. In the first example, the floor got a LOT of light, in the second example, the floor is unlit. This happens because in the second example the floor is facing away from the light.

In the real world, the floor would stop light from reaching the objects. In OpenGL each object is shaded and lit individually, like if no other object was in the scene. It is possible to add code to support shadows, but it's a complicated process that we will talk about later.

Two Lights

So, actually using a light us pretty easy. OpenGL supports up to 8 lights! Let's try to add one more and see what happens. We're going to add a second light, this one is going to be red. It's going to come from the Opposite direction of the blue light from the original example.

First things first, in the Initialize function we have to enable and configure light 1:

```
public override void Initialize() {
    grid = new Grid(true);
    GL.Enable(EnableCap.DepthTest);
    GL.Enable(EnableCap.CullFace);
    Resize(MainGameWindow.Window.Width, MainGameWindow.Window.Height);

    GL.Enable(EnableCap.Lighting);
    GL.Enable(EnableCap.Light0);
    GL.Enable(EnableCap.Light1);

    float[] white = new float[] { 0f, 0f, 0f, 1f };
    float[] blue = new float[] { 0f, 0f, 1f, 1f };
```

```

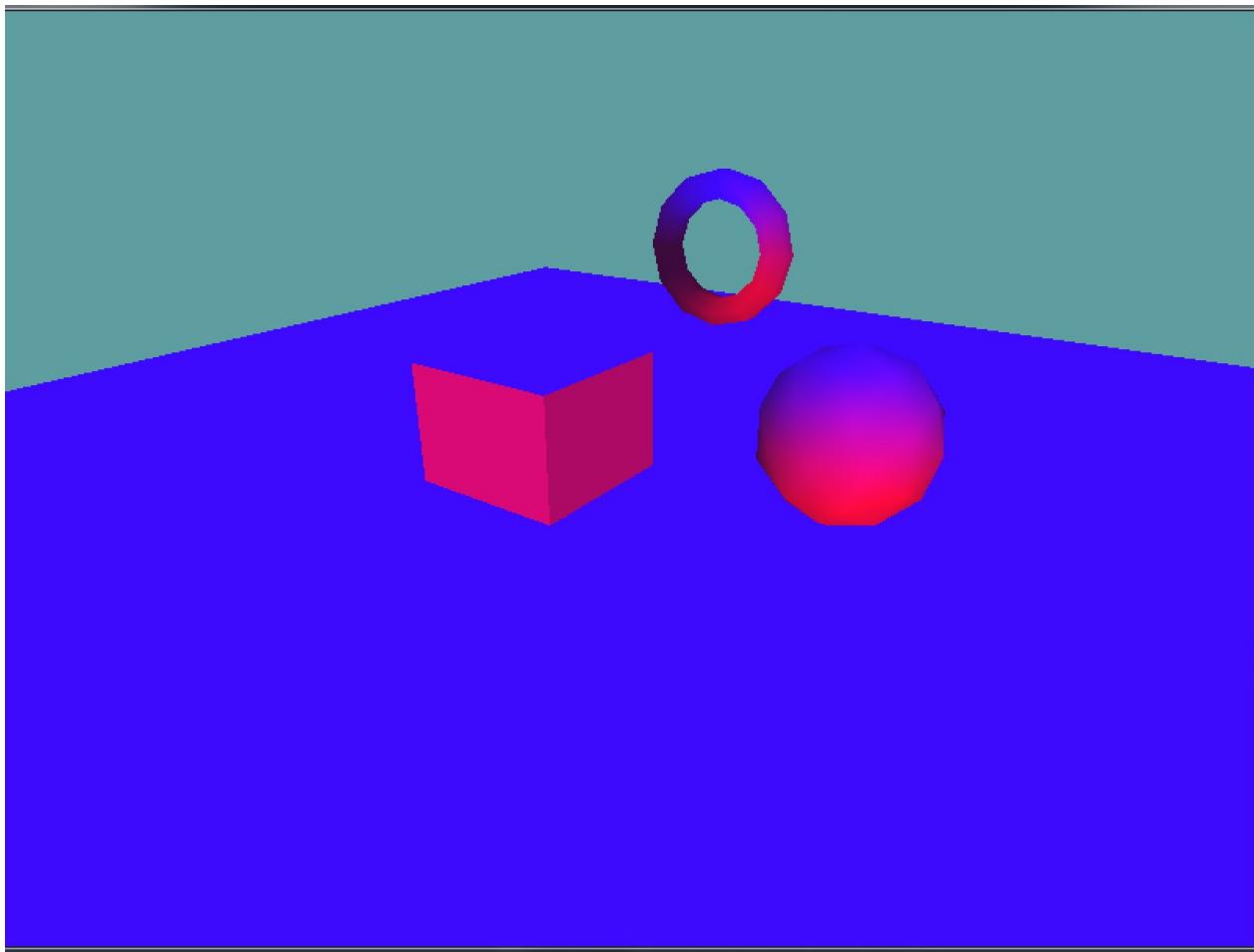
float[] red = new float[] { 1f, 0f, 0f, 1f };

// Config light 0
GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0.0f, 0.5f, 0.5f, 0.0f } );
GL.Light(LightName.Light0, LightParameter.Ambient, blue );
GL.Light(LightName.Light0, LightParameter.Diffuse, blue );
GL.Light(LightName.Light0, LightParameter.Specular, white );

// Config light 1
GL.Light(LightName.Light1, LightParameter.Position, new float[] { 0.0f, -0.5f, 0.5f, 0.0f } );
GL.Light(LightName.Light1, LightParameter.Ambient, red );
GL.Light(LightName.Light1, LightParameter.Diffuse, red );
GL.Light(LightName.Light1, LightParameter.Specular, white );
}

```

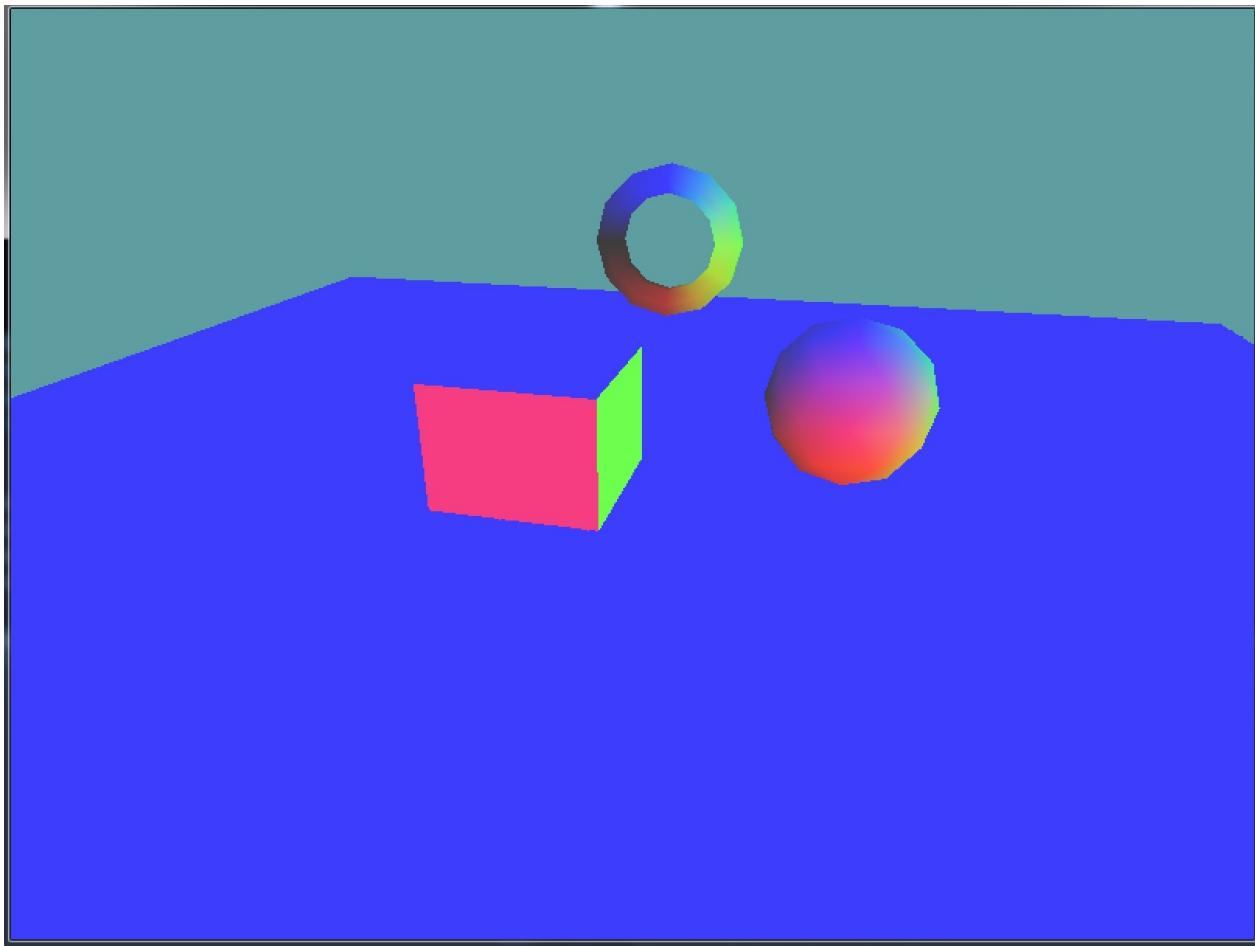
Lets see what that looks like:



The top of any object is shaded blue. The bottom is shaded red. Colors in the middle interpolate between the two lights, based on the direction of the surface normal.

On your own

Using the last example, add a third light (Light2). Configure this light to point straight left (1, 0, 0, 0), and be green. The final scene should look like this:



The torus and cube are especially interesting as we look at this scene. The floor is also noteworthy too. It's completely blue! That's because the floor gets blue light from the top. It's facing away from the red light on the bottom, and is parallel to the green light. So the red and green lights don't directly effect it.

But every time we added a light, the floor became a different shade of blue! If the lights don't directly effect floor, why is it's color changing? Ambient light. Each light adds a bit of ambient light to the scene. Remember, ambient lighting is light that has scattered so much it has no direction, so it uniformly effects everything in the scene!

The lighting models diffuse component is direct lighting, the ambient component is indirect lighting and the specular component is shininess! So, can we turn off the ambient light? Yes. Set the ambient component of any light that you don't want an ambient contribution from to **black**. Because black is $\text{RGB}(0, 0, 0)$, it is absent of color, therefore it has no contribution to lighting.

Dynamic lights

We now have a scene lit by 3 static lights. But lighting in OpenGL can be so much more than just static lights. It can be dynamic! Lights can move with objects! Using the previous scene as a base, let's animate the lights!

We're going to make the green light orbit the scene horizontally, the red light orbit the scene vertically, and we're going to make the camera stay static.

First, let's add some variables to the scene. We're going to use the same orbit logic we used for the camera, so these variables will be similar.

```
namespace GameApplication {
```

```

class LightingScene : Game{
    Grid grid = null;

    float cameraAngleX = 0.0f;
    float cameraAngleY = -25;
    float cameraDistance = 10.0f;

    float redAngleX = 0.0f;
    float redAngleY = 0.0f;

    float greenAngleX = 0.0f;
    float greenAngleY = 0.0f;

    public override void Initialize() {
        // ... rest of code unchanged

```

We have an X and a Y angle for both lights, we don't have a distance, to keep the vectors of unit length, we will use 1.0f for the distance. Next lets change the Update function to stop rotating the camera, and start rotating the lights!

```

public override void Update(float dTime) {
    //cameraAngleX += 30f * dTime;

    redAngleY += 15.0f * dTime;
    greenAngleX += 30.0f * dTime;
}

```

Now that the actual rotation logic is in place, all that's left to do is actually rotate the lights. We do this by changing the `LightParameter.Position` parameter of light 1 (red) and light 2 (green) every time a frame is rendered.

```

public override void Render() {
    Vector3 eyePos = new Vector3();
    eyePos.X = cameraDistance * -(float)Math.Sin(cameraAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(cameraAngleY * (float)(Math.PI / 180.0));
    eyePos.Y = cameraDistance * -(float)Math.Sin(cameraAngleY * (float)(Math.PI / 180.0)) * (float)Math.Cos(cameraAngleX * (float)(Math.PI / 180.0));
    eyePos.Z = -cameraDistance * (float)Math.Cos(cameraAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(cameraAngleY * (float)(Math.PI / 180.0));

    Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
    GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);
    grid.Render();

    // Compute position of red light
    Vector3 redPosition = new Vector3();
    redPosition.X = 1.0f * -(float)Math.Sin(redAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(redAngleY * (float)(Math.PI / 180.0));
    redPosition.Y = 1.0f * -(float)Math.Sin(redAngleY * (float)(Math.PI / 180.0)) * (float)Math.Cos(redAngleX * (float)(Math.PI / 180.0));
    redPosition.Z = -1.0f * (float)Math.Cos(redAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(redAngleY * (float)(Math.PI / 180.0));

    // Compute position of green light
    Vector3 greenPosition = new Vector3();
    greenPosition.X = 1.0f * -(float)Math.Sin(greenAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(greenAngleY * (float)(Math.PI / 180.0));
    greenPosition.Y = 1.0f * -(float)Math.Sin(greenAngleY * (float)(Math.PI / 180.0)) * (float)Math.Cos(greenAngleX * (float)(Math.PI / 180.0));
    greenPosition.Z = -1.0f * (float)Math.Cos(greenAngleX * (float)(Math.PI / 180.0)) * (float)Math.Cos(greenAngleY * (float)(Math.PI / 180.0));

    // Update light positions every frame
    GL.Light(LightName.Light1, LightParameter.Position, new float[] { redPosition.X, redPosition.Y, redPosition.Z, 0.0f });
    GL.Light(LightName.Light2, LightParameter.Position, new float[] { greenPosition.X, greenPosition.Y, greenPosition.Z, 0.0f });

    // ... rest of code unchanged
}

```

Running your scene now, you can see that the lights are changing in real time. But it's pretty hard to get a firm grasp of what is happening. We can kind of see where the lights are coming from, but not really. Lets add some debug code to actually visualize the lights.

We're going to render two lines, one for the red light, one for the green light. They are going to have a local origin

somewhere in space, and will point in the direction that the lights are pointing in. Both of the lines will be of unit length to make debugging easier.

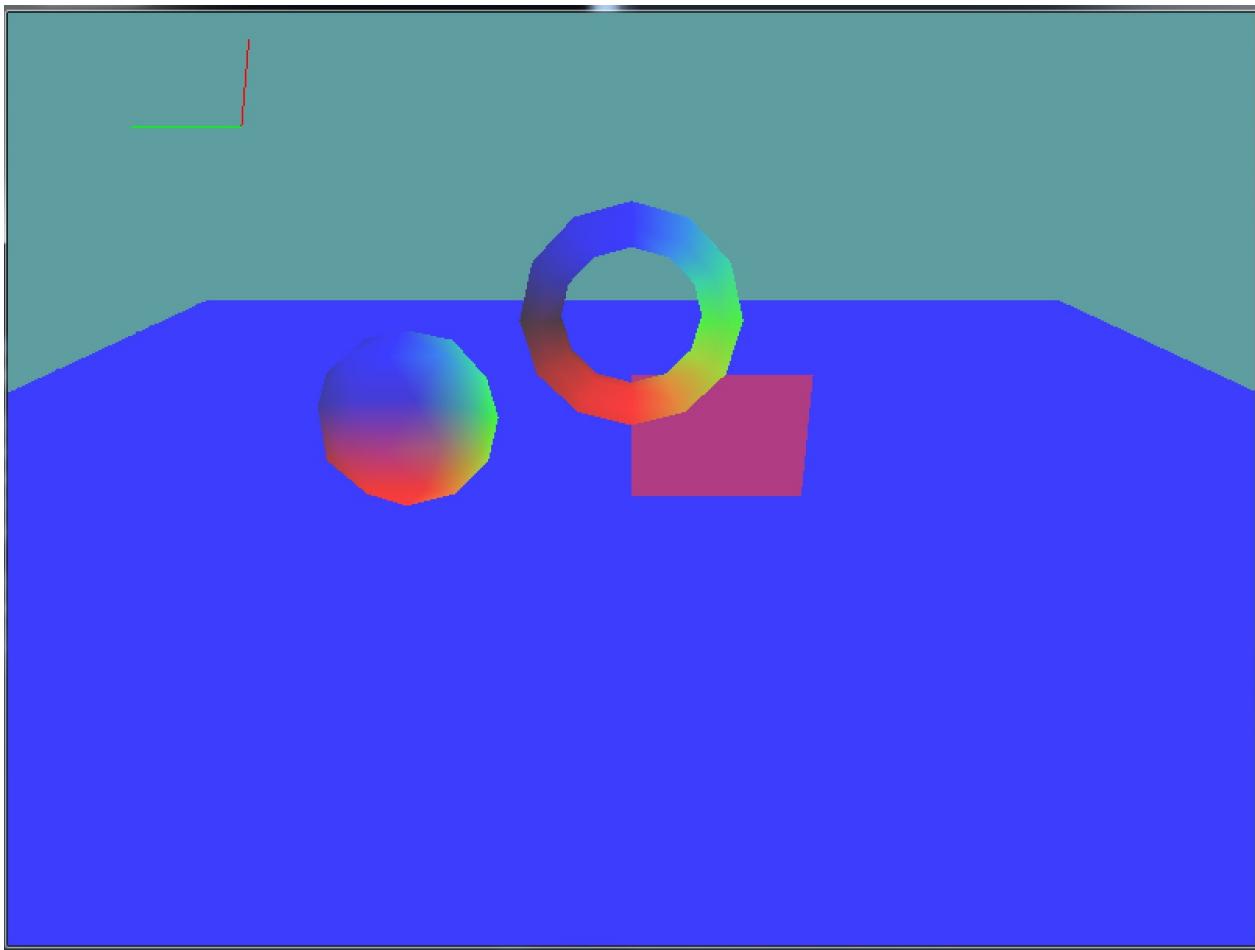
The only catch is, because we are rendering the lines in a lit scene, we can't really define a set color for them; because the lighting equation ignores `GL.Color3`. So to cope with this we're going to disable lighting in the middle of our draw function, draw the indicators and re-enable the lights after.

```
// Update light positions every frame
GL.Light(LightName.Light1, LightParameter.Position, new float[] { redPosition.X, redPosition.Y, redPosition.Z, 0.0f });
GL.Light(LightName.Light2, LightParameter.Position, new float[] { greenPosition.X, greenPosition.Y, greenPosition.Z, 0.0f });

// Add some debug visualization so we can see the direction of the lights
// Disable lights, so the color of the lines comes from GL.Color, not lighting
GL.Disable(EnableCap.Lighting);
GL.PushMatrix();
GL.Translate(4f, 4f, 0f);
GL.Begin(PrimitiveType.Lines);
// Draw red ray
GL.Color3(1f, 0f, 0f);
GL.Vertex3(0f, 0f, 0f);
redPosition.Normalize(); // We want to render a unit vector
redPosition *= -1.0f; // Invert so we see the light direction
GL.Vertex3(redPosition.X, redPosition.Y, redPosition.Z);
// Draw green ray
GL.Color3(0f, 1f, 0f);
GL.Vertex3(0f, 0f, 0f);
greenPosition.Normalize();
greenPosition *= -1.0f;
GL.Vertex3(greenPosition.X, greenPosition.Y, greenPosition.Z);
GL.End();
GL.PopMatrix();
// Re-enable lights, we want the rest of the scene lit
GL.Enable(EnableCap.Lighting);

// ... rest of code unchanged
```

Running your game now, you should have a nice indicator as to which way your lights point, the scene looks like this:



This is a really good example of rendering a lit scene with some unlit objects. At any point during your objects execution you can enable / disable lighting as a whole, you can enable / disable specific lights; OR you can change individual properties of specific lights. This is how we are able to do so much with just 8 lights, they can be 8 different lights for every object!

Independently lit objects

In games (and movies) it's common to have "local lights". A local light is a light that an artist places in a scene, and is attached to a model (using the matrix stack, the same way a foot is attached to a leg). The light moves with a specific character and effects only the character. So a character is lit by the scene lighting, and then some extra character specific lighting to really make it pop.

Movies take this one step further, and have lights only affect certain parts of a character. In [Tangled](#), Flynn had 8 lights attached to him that only affected his hair! Highlights were added to the hair using local lights.

We need independently lit objects to do local lighting. The process for doing local lights and the process for light culling (having the bedroom light not affect objects in the kitchen) is the same. We simply disable lights that don't effect the object, and configure the lights that do affect it appropriately.

We're going to make a new scene using the final Dynamic Lights scene as a base. This is what we are about to render

- The tarus only be lit by the green light
- The sphere lit only by the red light
- The cube will be lit by the blue light and a purple light.

- Instead of adding a 4th light (Light3) to the scene, we're going to recycle the red light and turn it purple
- The purple light will be static.
- The floor will be lit only by the blue light.

The torus and sphere are simple, we just need to enable / disable lights before drawing each of them. Update your render code to reflect this:

```
// We want to set the state of all 3 lights here
// because this is the first lit object that is drawn
// The enable / disable states carry over from last frame
// so setting all states here acts as a kind of reset

// Disable blue light
GL.Disable(EnableCap.Light0);
// Disable green light
GL.Disable(EnableCap.Light2);
// Enable the red light
GL.Enable(EnableCap.Light1);

// Draw torus
GL.PushMatrix();
GL.Color3(0f, 1f, 0f);
GL.Translate(0.0f, 2.5f, -2f);
Primitives.Torus(0.2f, 0.8f, 6,12);
GL.PopMatrix();

// Right now, only the red light is enabled, disable it
GL.Disable(EnableCap.Light1);
// The green light is still disabled, enable it
GL.Enable(EnableCap.Light2);
// We don't need to disable light 0, it's still disabled from last call

// Draw sphere
GL.PushMatrix();
GL.Color3(1f, 0f, 0f);
GL.Translate(2.5f, 1.0, -0.5f);
Primitives.DrawSphere(1);
GL.PopMatrix();
```

Run the game, see what it looks like. The cube and the rest of the scene are only a little bit more complicated as we need to actually update the color and of the red light to essentially become a new purple light. Lets update the render function to reflect these changes:

```
// Disable the green light
GL.Disable(EnableCap.Light2);
// Enable the red light
GL.Enable(EnableCap.Light1);
// Enable the blue light
GL.Enable(EnableCap.Light0);

// Change the color of light 1 from red to purple
float[] purple = new float[] { 1f, 0f, 1f, 1f };
GL.Light(LightName.Light1, LightParameter.Diffuse, purple);
GL.Light(LightName.Light1, LightParameter.Ambient, purple);
// Specular component can stay white
// Update the position of light 1 so it's static (purple will just have a static direction)
GL.Light(LightName.Light1, LightParameter.Position, new float[] { 0.0f, -0.5f, -0.5f, 0.0f });

// Draw cube
GL.PushMatrix();
GL.Color3(0f, 0f, 1f);
GL.Translate(-1f, 0.5f, 0.5f);
Primitives.Cube();
GL.PopMatrix();
```

Run the game to see what it looks like. The sphere is rendered green like we wanted. The cube is rendered purple and blue like we wanted. The ground is rendered purple and blue like we wanted. The torus however is wrong! It's not red, it's purple and blue! Why?



The answer is simple, nothing is reset for you. Ever! At the end of the frame we set light 1 to be purple, which means that at the start of the next frame it will still be purple! This is an easy fix. We want the ground to render blue / purple still, but the next object, the torus to render in red.

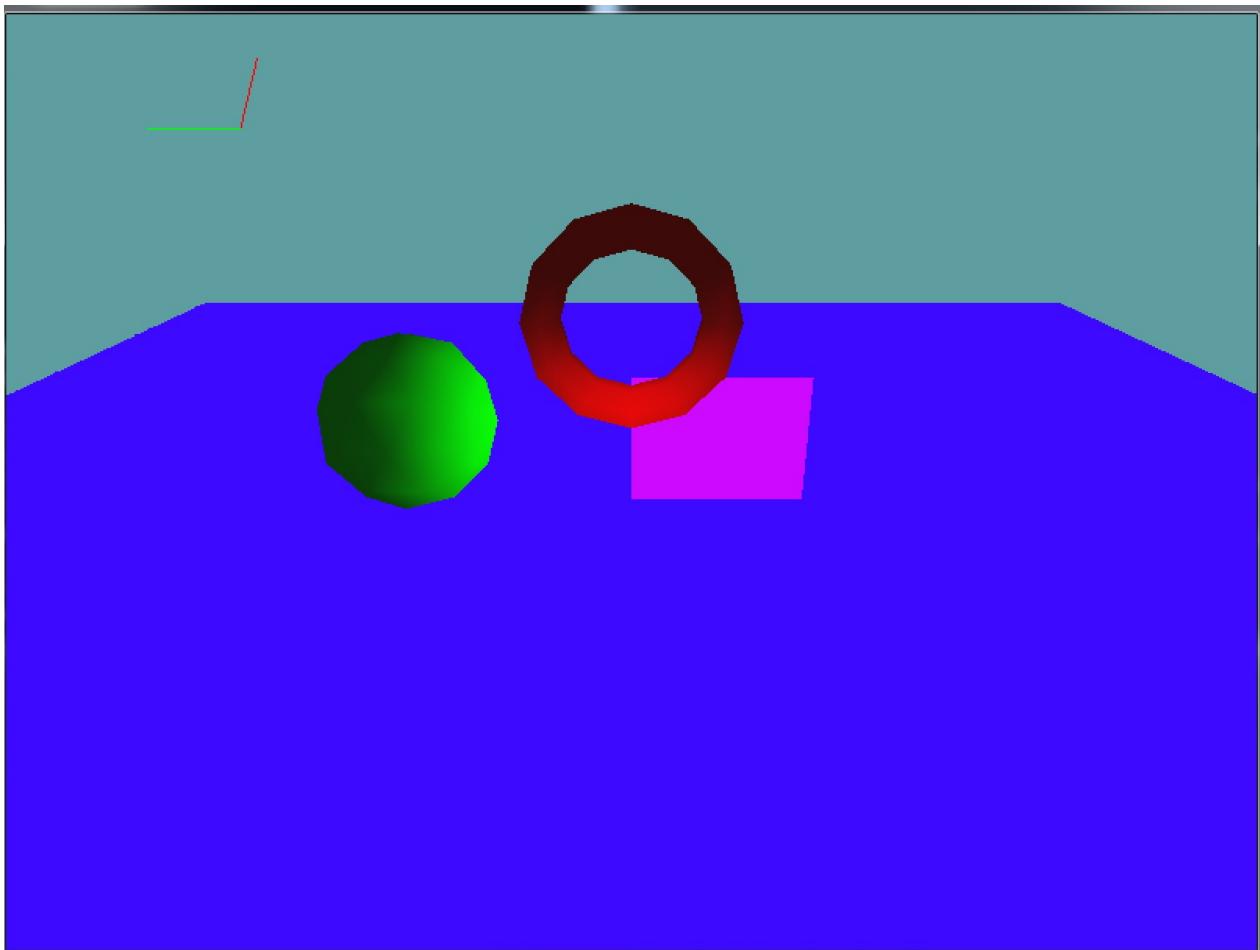
We know that the torus renders with light 1, so let's change the color of light1 to red after drawing the ground. Lets update the render code to reflect this:

```
grid.Render();

float[] red = new float[] { 1f, 0f, 0f, 1f };
GL.Light(LightName.Light1, LightParameter.Diffuse, red);
GL.Light(LightName.Light1, LightParameter.Ambient, red);

// ... rest of code unchanged
```

A bit off topic, but the reason the ground is using the same lighting as the cube is because it renders with whatever configuration the state machine was left in at the end of last frame. In this situation, this is what we want, sometimes it may not be. But this does imply an interesting artifact. It's not visible, because the timing is so short, but the first frame that the application runs the ground is actually colored wrong. Because the render has not executed before, so the state machine is not in the state we expected it to be. However this error is only for 1 frame, so we can ignore it. Your app at this point should look like this:



One thing to note, even though we have independently lit objects in this scene, we don't actually have local lights. This is because all of the lights are static to the scene, none of the lights move with objects in the scene. Local lights don't make much sense with a directional light. We will implement them when doing point lights.

Point Lights

The code for point lights is very similar to the code for directional lights. You can think of a directional light as the sun, a point light is more like a light-bulb. It's a point in space that emits light, the further the light gets from the point, the less it's influence is.

Unlike directional lights, point lights do have a position, which means the matrix stack and the model-view matrix become incredibly important. You **ALWAYS** have to set the position of a point light in your render loop, after the view matrix has been set

Single Static Light

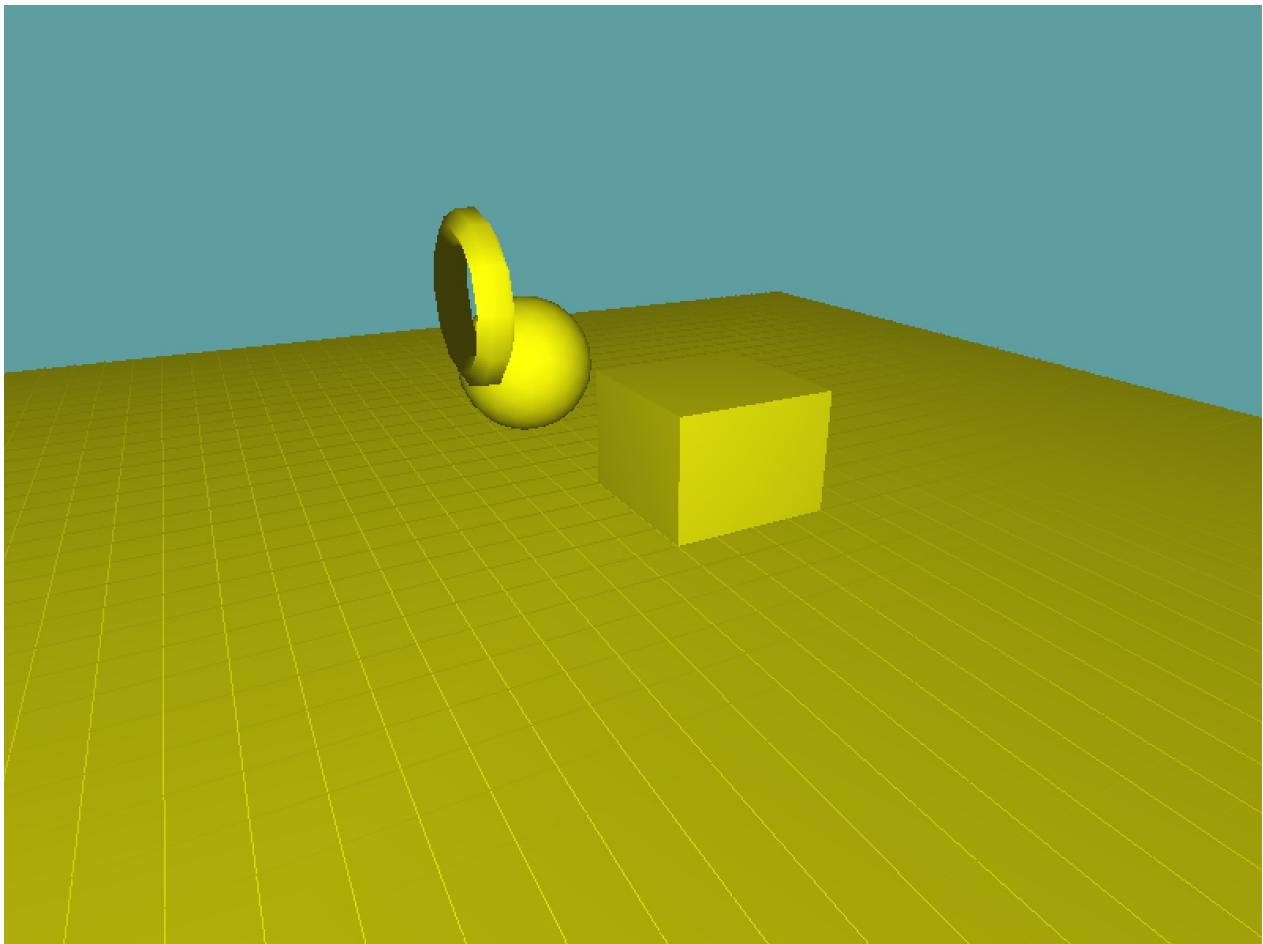
Lets start by adding a single, static point light to our test scene. We will have the light be near origin, and large enough to illuminate all objects in the scene. For this, the light will be yellow.

In the Initialize function enable lighting and light 0. Set the color of the light to yellow RBG(1, 1, 0). Remember, to set the color of a light you need to set its ambient, diffuse and specular components. Also, the light color is a 4 component array, the last component being alpha!

Still in initialize, after the color of the light is set up, we need to set the lights position. The position is a 4 component vector (float array), with the W component being 1.

```
// Having a W of 1 makes the light a point light with a position
float[] position = new float[] { 0f, 1f, 0f, 1f };
GL.Light(LightName.Light0, LightParameter.Position, position);
```

If you run your game now, you get the following scene:



That does not look right. Not at all. The whole scene is lit, nothing about that says point light to me. Before continuing reading, think about why you think this is broken.

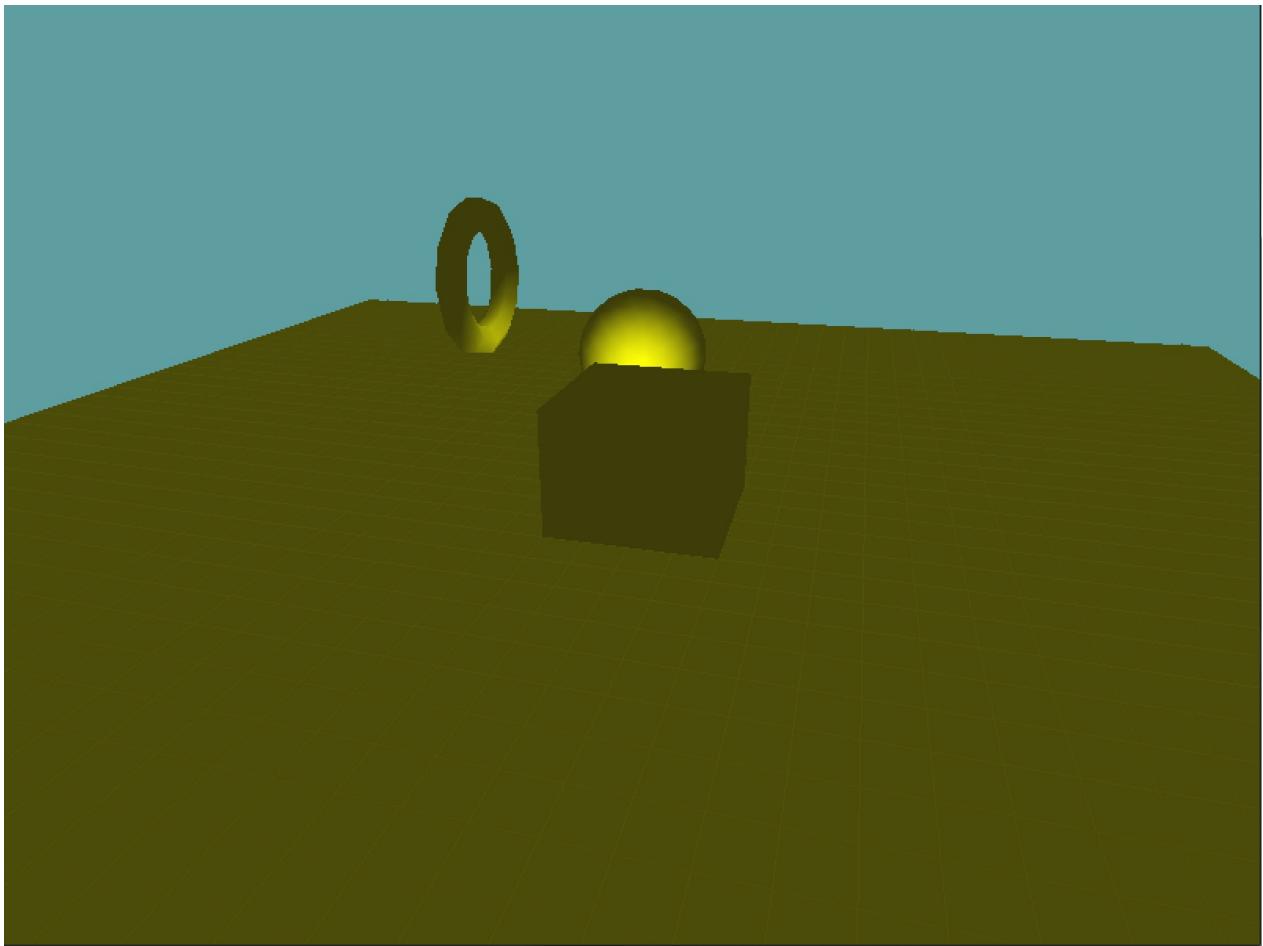
Well, a point light has two major properties, a position and a radius. And frankly, we set the position wrong. Remember, the Position is affected by what's in the **modelview** matrix. But in the Initialize function, that matrix is essentially trash! So, we must also set the position of the light in the render function. After the modelview is built, but before we render anything.

```
GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

float[] position = new float[] { 0f, 1f, 0f, 1f };
GL.Light(LightName.Light0, LightParameter.Position, position);

grid.Render();
```

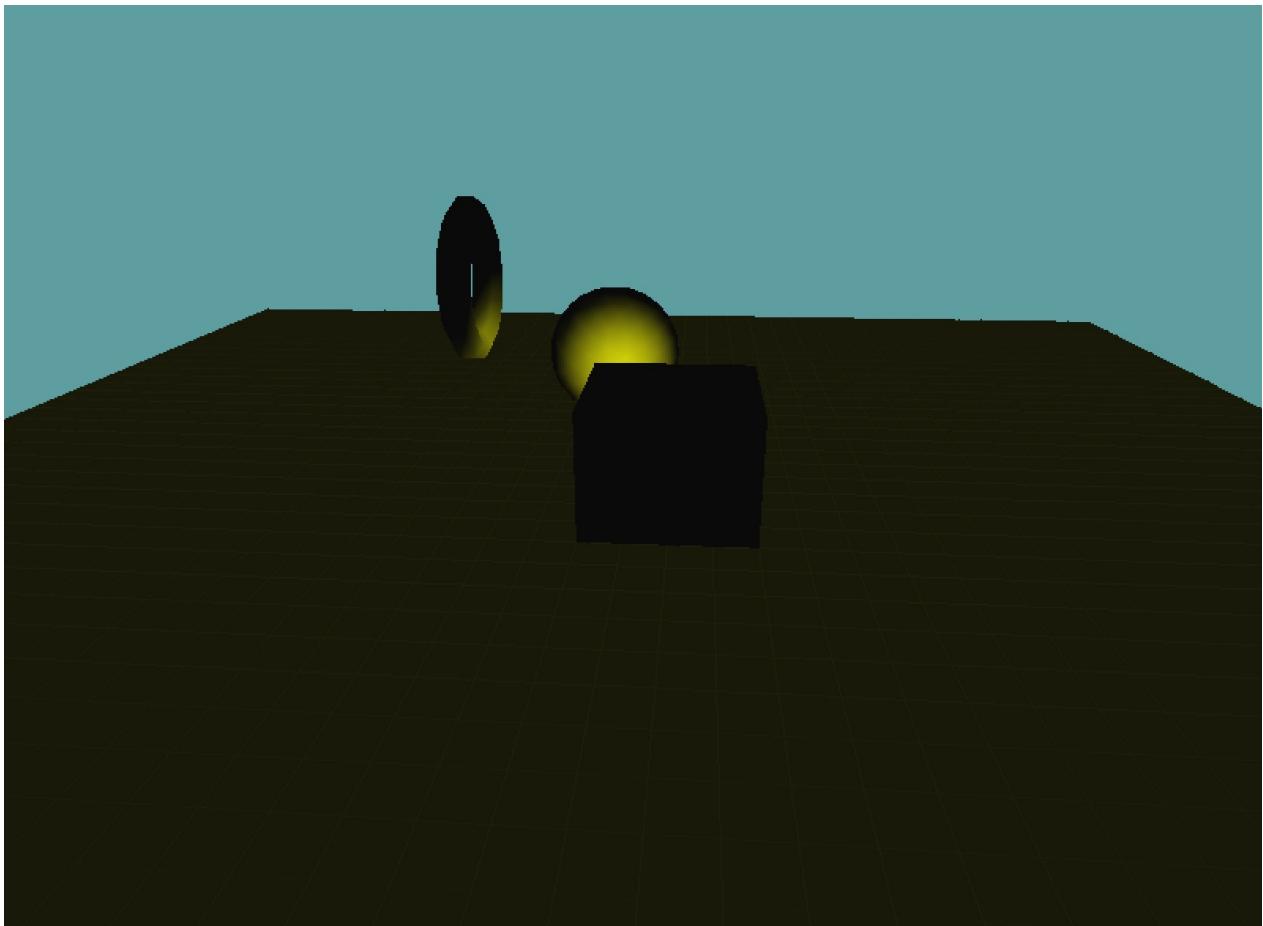
With that code in place, the scene now looks like this:



That looks so much better! It's actually starting to look like a lightbulb illuminating the scene! But why is everything an ugly mustard color? That doesn't look right! Well, that's the ambient component of the light. Remember how light works, the diffuse component is applied to an object where the light touches it. The ambient component is applied everywhere, it represents light that bounced around a lot and is now just ambient. The specular component is shininess.

With a large light, like the sun, or a super intense light, like what they use on a construction site it makes sense to have an ambient component. But with a small light (like a light bulb) it does not. Usually a bulb is only strong enough to light a small area, not a large room! With that being said, change the ambient term from yellow to black. This should leave the diffuse term yellow, the ambient term black and the specular term white.

This will leave your scene looking like this:



Wow, we almost have it! Some things are black, there is a yellow light that fades over distance, there is just one problem. Look at the ground. It's not lit! Before reading on, think about why this might be occurring.

The answer is a tad technical, but this has to do with how OpenGL handles lights. See, OpenGL doesn't calculate lights per pixel, it calculates the light color of every vertex. When drawing a triangle, it calculates the color of all 3 vertices, and then just interpolates them like normal.

Remember how the ground (under the grid) is just two large triangles? Well, none of the triangle's vertices happen to fall into the attenuation zone. Therefore, the ground is rendered as if it was ALL outside the attenuation zone, even the parts that are inside. This is a VERY common problem with per vertex point lights. Large meshes tend to get lit wrong.

How can we fix this? By adding more vertices! The higher density your mesh is (the more tessellated your mesh is), the more accurate your lighting will become. I'm going to give you some code that takes a quad, and divides it into 4 quads before rendering. It's a recursive function, so you can sub-divide a quad as many times as you want. Add this to the `Grid` class:

```
private static void SubdivideQuad(float l, float r, float t, float b, float y, int subdivLevel, int target) {
    if (subdivLevel >= target) {
        GL.Vertex3(l, y, t);
        GL.Vertex3(l, y, b);
        GL.Vertex3(r, y, b);

        GL.Vertex3(l, y, t);
        GL.Vertex3(r, y, b);
        GL.Vertex3(r, y, t);
    }
    else {
        float half_width = Math.Abs(r - l) * 0.5f;
        float half_height = Math.Abs(b - t) * 0.5f;
```

```

        SubdivideQuad(l, l + half_width, t, t + half_height, y, subdivLevel + 1, target);
        SubdivideQuad(l + half_width, r, t, t + half_height, y, subdivLevel + 1, target);
        SubdivideQuad(l, l + half_width, t + half_height, b, y, subdivLevel + 1, target);
        SubdivideQuad(l + half_width, r, t + half_height, b, y, subdivLevel + 1, target);
    }
}

```

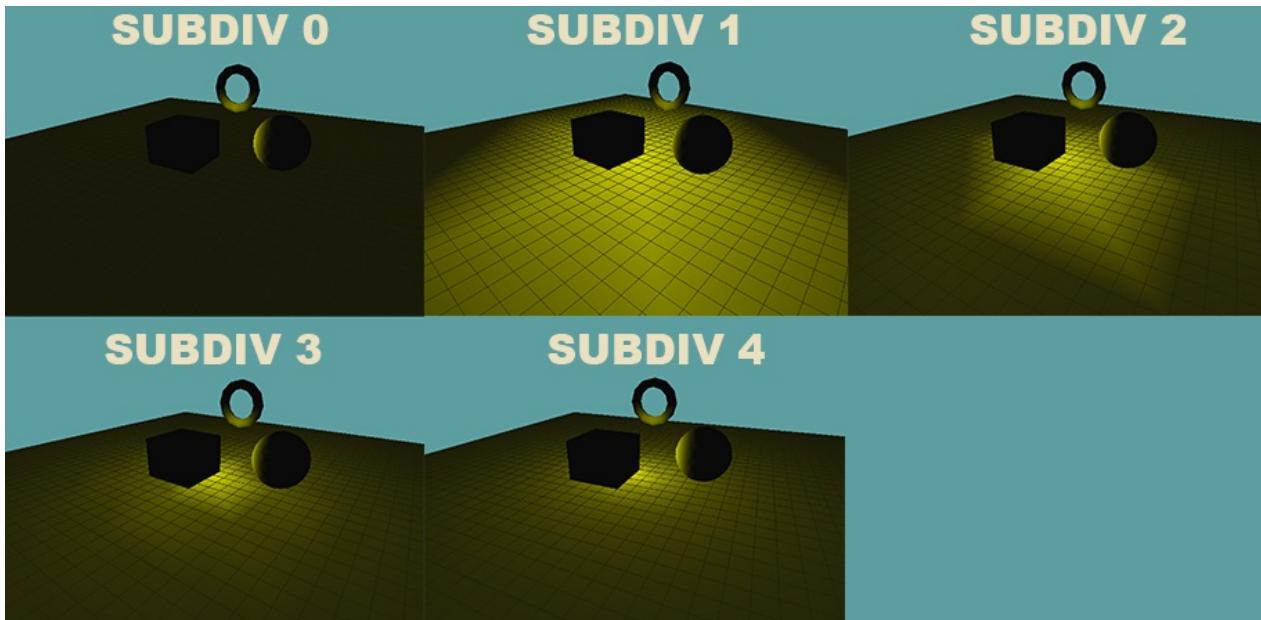
Now modify the `Render` function so it takes an optional subdivision parameter (0 by default), and instead of manually drawing out two large triangles it calls the `subdivide` function i just gave you:

```

public void Render(int subdiv = 0) {
    // Draw grid
    if (RenderSolid) {
        GL.Color3(0.4f, 0.4f, 0.4f);
        GL.Begin(PrimitiveType.Triangles);
        {
            GL.Normal3(0.0f, 1.0f, 0.0f);
            SubdivideQuad(-10, 10, -10, 10, -0.01f, 0, subdiv);
        }
        GL.End();
    }
    // ... rest of code is unchanged
}

```

Now change the render function of your demo scene to use a subdivision of 1. See what it looks like. When it doesn't look good, change it to 2, repeat until it looks good. I decided 4 would be the one i use. This is because 4 is the first subdivision where you can see the light as a circle on the ground!



Almost there, one last problem, why is the cube black! On every side! The answer is so simple it's hard to track the bug down sometimes. The light is INSIDE the box! Because the light is inside the geometry, all the faces face the same way as the light, and no face is lit. Change the light position to:

```
float[] position = new float[] { 0.5f, 1f, 0.5f, 1f };
```

Now as your scene rotates, you will see one of the sides of the cube lit!

That's it, we're done. It may seem like we went through a lot of issues to get a single point light up, that's because we did. I made sure to include all of the major mistakes that get made when setting up lights above. Chances are you will make those mistakes (I still do), when that happens I want you to have the tools to fix them.

Attenuation

For this section, start with the scene we got at the end of the last section. Now that we can render a point light in the scene, and have it look pretty ok, how do we change the radius / size of the light?

That's where attenuation comes in. Like the color of the light, you can set its attenuation in Initialize for now, it doesn't need to be set every frame. If you remember, attenuation is an inverse function (it starts with $1.0f / ...$). This means the lower the attenuation factors, the stronger / larger the light.

Up to this point our point light has used the default attenuation values set by OpenGL. The default attenuation is:

- Constant, 1
- Linear, 0
- Quadratic, 0

Now, seeing how a point light can be visualized as essentially a large sphere, common sense would tell us that there is some way to configure attenuation by defining a radius and some strength. There is not. The attenuation model OpenGL chooses looks physically accurate, but there is no common sense way to configure it. You just have to play with the numbers.

For example, try setting a "neutral" attenuation like so:

```
GL.Light(LightName.Light0, LightParameter.ConstantAttenuation, 0.25f);
GL.Light(LightName.Light0, LightParameter.LinearAttenuation, 0.25f);
GL.Light(LightName.Light0, LightParameter.QuadraticAttenuation, 0.0f);
```

The lighting in the scene has barely changed. The center is a bit more well-lit, but the area of effect for the light is almost unchanged.

Lets try introducing a quadratic term:

```
GL.Light(LightName.Light0, LightParameter.ConstantAttenuation, 0.25f);
GL.Light(LightName.Light0, LightParameter.LinearAttenuation, 0.25f);
GL.Light(LightName.Light0, LightParameter.QuadraticAttenuation, 0.25f);
```

As you can see, the area of effect for the light remains mainly the same, however the brightness of the light has been reduced significantly.

Like I mentioned previously, the closer to 0 your attenuation factors are the stronger the light is. Lets expand the lights area of influence, as well as its brightness. Try the following:

```
GL.Light(LightName.Light0, LightParameter.ConstantAttenuation, 0.01f);
GL.Light(LightName.Light0, LightParameter.LinearAttenuation, 0.1f);
GL.Light(LightName.Light0, LightParameter.QuadraticAttenuation, 0.0f);
```

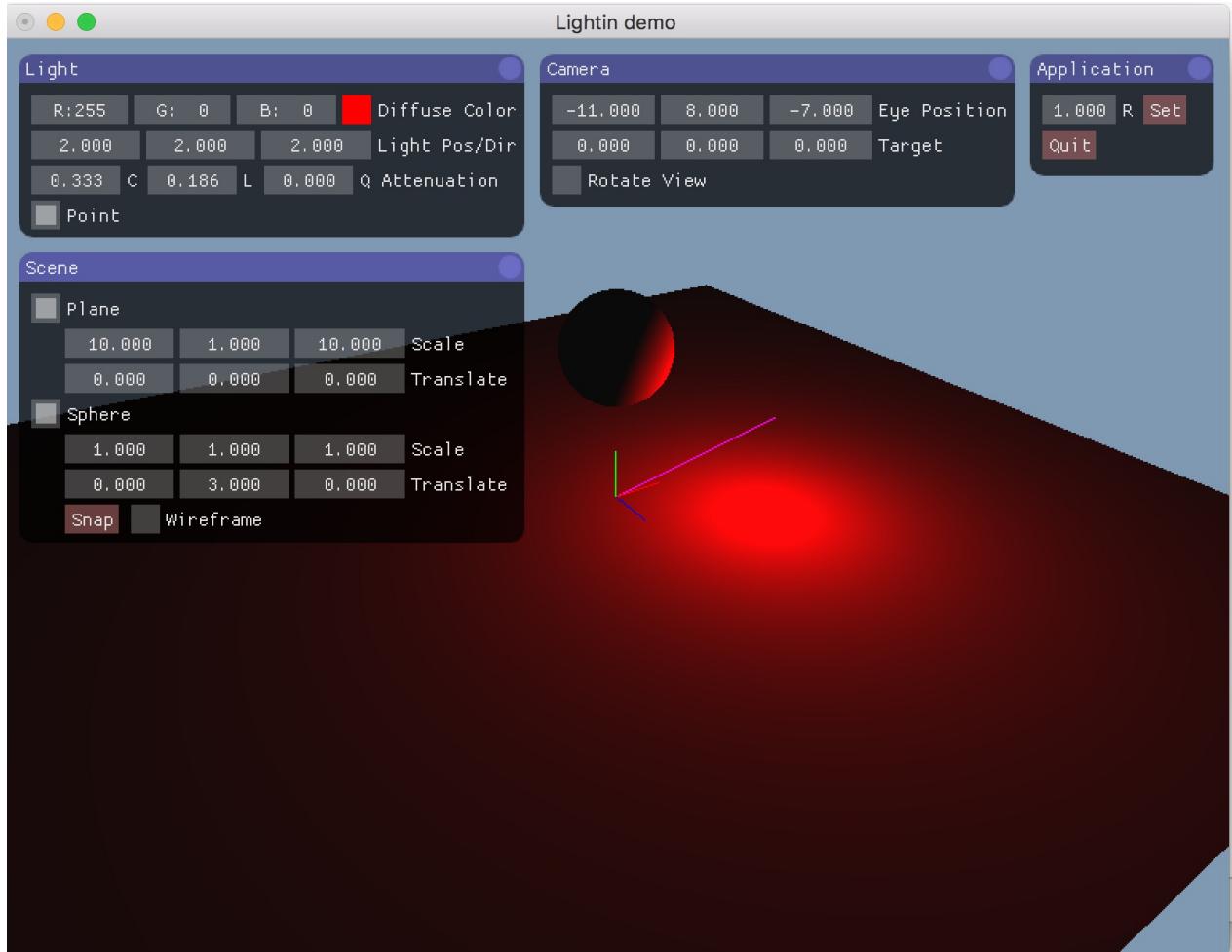
All of a sudden the point light is huge! That's all there is to point lights. You set a position, keeping in mind that the

modelview matrix effects this position, and you make sure to configure the lights attenuation.

Finding good attenuation

One of the hardest things to do with point lights is finding a decent attenuation factor. Artists will play around with the attenuation numbers for hours before choosing the final attenuation of a light.

How did i find the attenuation numbers we used in the above examples? I wrote a small program:



The program is simple, it has a few geometry objects (VERY HIGHLY TESSELATED), i subdivide everything by 10 to ensure there are enough verts that the lighting looks great. This pulls my CPU up to 50% usage whenever i run the program, but i only run it for short ammounts of time.

In this program i can type in numbers for the lights attenuation factors, color, position, etc. I can also click and drag on the text boxes to step each factor the attenuation by 0.001f

I highly suggest making something similar. Tough not required, it will REALLY help you out later. The test scene we have set up right now is actually a really good start. If you don't want to invest time in writing a UI, you can make your interaction be simple.

If i was to write a simple interaction application (which is what i did in school), pressing Q and W would step the constant factor by 0.001f, A & S would step the Linear factor by 0.001f, and Z & X would step the quadratic factor by 0.001f. Similarly i'd have E & R, D & F, C & V also modifying the appropriate factors by 0.01f, and T & Y, G & H, B & N modifying the factors by 0.1f. I'd make the P button reset all attenuation to 0.

So, if you set up your input to modify the attenuation, how can you read back what it is? You could build a font rendering interface, it's something you've already done for the dungeon game. OR you could do what i do for quick and dirty programs, set the title bar every frame!

```
MainGameWindow.Window.Title = "Constant:" + constantFactor + ", Linear: " + linearFactor + ", Quadratic: " + quadraticFacto
```

This way you can build a quick application for yourself that will let you play with attenuation settings in about an hour. It might not be super user friendly, but it will get the job done.

You don't have to build this application as a part of this chapter, but if you decide invest the time in building it now it might save you some trouble down the line.

Local lights

A common use of point lights is to use them as local lights. That is lights that are attached to objects, and move with objects. In an entity-component system, you might have a `PointLightComponent` class for this that you can attach to a game object. I might configure a scene like this

```
Root (game object)
  World (game object)
    Lamp (game object)
      Light (game object with light component)
    Lamp (game object)
      Light (game object with light component)
```

That way each lamp object also has a light object that is offset relative to it. And i don't have to find the world position for every light, because the light is relative to the lamp, each light will have the same relative offset.

A really creative use of a local light is [Navi](#) from zelda. Navi is local to link, wherever link moves Navi follows. But Navi also moves independently, flying around link. Navi has a small radius that illuminates wherever she is. The designers leveraged this, by making navi also be used as a lighting device, whenever you read a sign she flies to it and slightly illuminates it, putting emphasis on the sign being read.

With that in mind, let's create a simple local light. We're going to keep the scene illuminated by our yellow light, but we're going to add a local red point light to the sphere. This will only light the sphere. Once the light is added, we will animate it to orbit the sphere, because local lights don't have to be static.

Start by enabling Light1 in the initialize function. Set its diffuse color to red, ambient color to black and specular color to white. Set the attenuation of this new light to the default values, Constant: 1, Linear: 0, Quadratic: 0. So far everything has been pretty standard. Now comes the part that makes this light a local light.

We are going to enable Light 1 just before the sphere is drawn, and disable it right after, this ensures that only the sphere is lit. Then, we are going to set the light position AFTER the model matrix for the sphere has been configured.

```
// Only affect sphere
GL.Enable(EnableCap.Light1);
GL.Color3(1.0f, 0.0f, 0.0f);
GL.PushMatrix();
{
  GL.Translate(2.5f, 1.0f, -0.5f);
  // Set light position, so it's relative to sphere
```

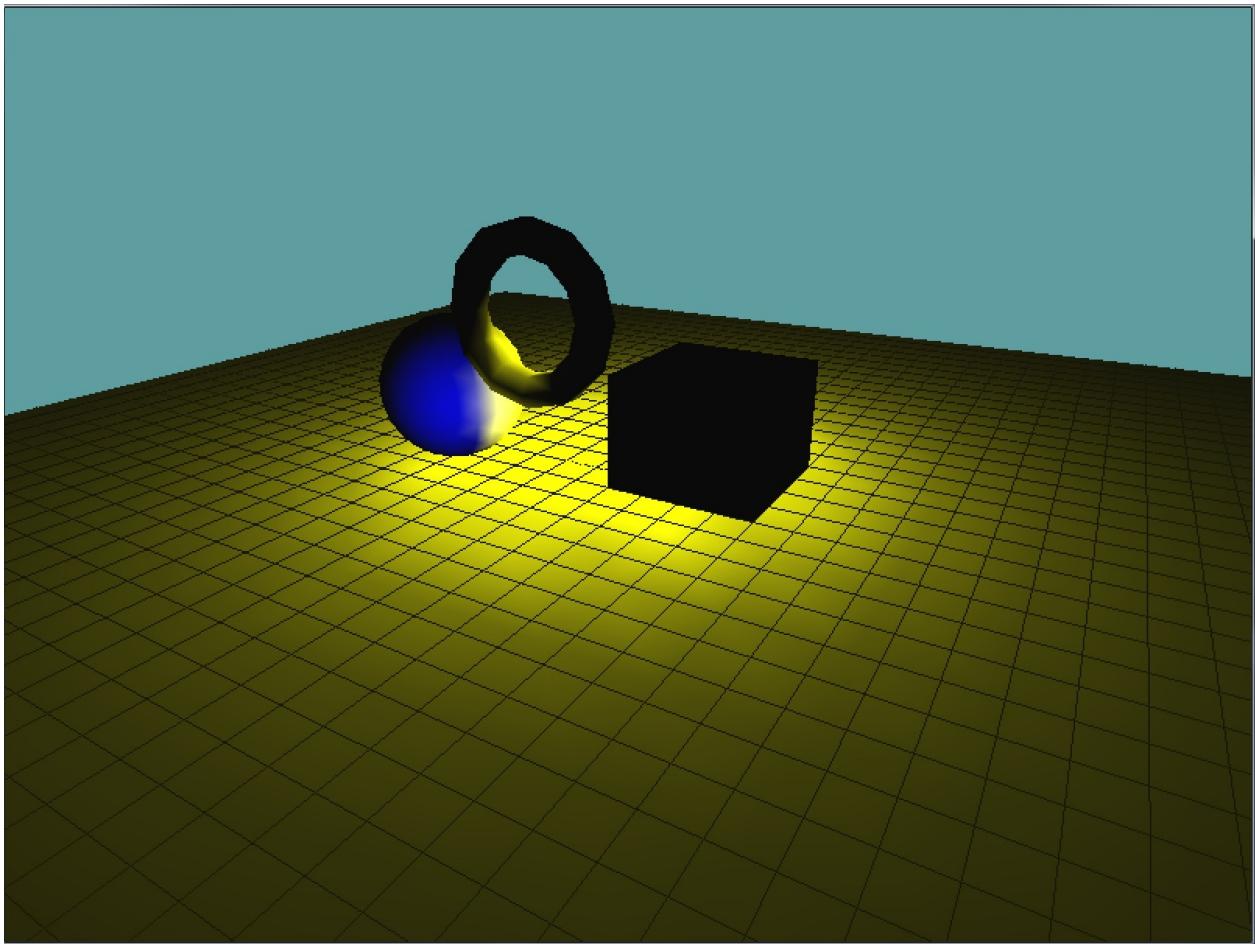
```

        position = new float[] { -2f, 0f, -2f, 1f };
        GL.Light(LightName.Light1, LightParameter.Position, position);

        Primitives.DrawSphere();
    }
    GL.PopMatrix();
    // Only affect sphere
    GL.Disable(EnableCap.Light1);

```

Because the model-view matrix effects the light at the time that we set the lights position, and we set the lights position AFTER the model matrix of the sphere has been applied, this light will move with the sphere. So if you move the sphere, (by changing `GL.Translate(2.5f, 1.0f, -0.5f);`), the light will move with it! This is what your scene should look like:



That's the basic implementation of a local light. Let's try getting a bit more fancy with it by actually orbiting the light around the sphere! First things first, let's add a new variable for the angle of the light orbit to the class:

```

namespace GameApplication {
    class LocalLightSample : Game{
        Grid grid = null;
        Vector3 cameraAngle = new Vector3(0.0f, -25.0f, 10.0f);
        float lightAngle = 0.0f;

```

In update, we're going to rotate this light faster than the camera:

```

public override void Update(float dTime) {
    cameraAngle.X += 30.0f * dTime;
    lightAngle += 90.0f * dTime;
}

```

```
}
```

Now rendering gets a bit tricky. We could do something crazy and figure out how to configure the position of the light based on the light angle, but that involves a bunch of math that we don't really need to do. Instead, we can simply move the **modelview** matrix into the position that we want to render the light at, render the light at (0,0,0) relative to the modelview matrix, then restore the modelview matrix before rendering the sphere.

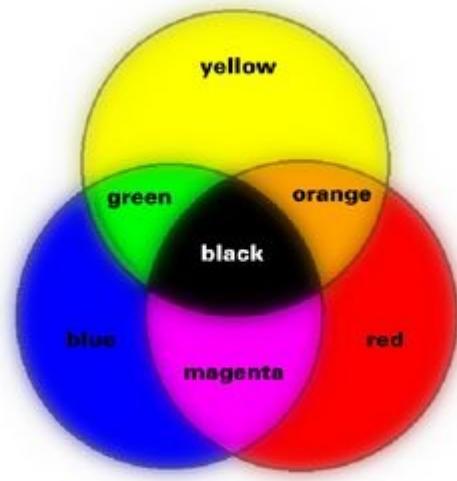
```
GL.Enable(EnableCap.Light1);
GL.Color3(1.0f, 0.0f, 0.0f);
GL.PushMatrix();
{
    // Apply the sphere translation
    GL.Translate(2.5f, 1.0f, -0.5f

    // Move the light into place
    GL.PushMatrix();
    {
        // Orbit the light around the y axis
        GL.Rotate(lightAngle, 0f, 1f, 0.0f);
        GL.Translate(-2f, 0f, -2f);

        // Render light where the model-view matrix is. No translation.
        position = new float[] { 0f, 0f, 0f, 1f };
        GL.Light(LightName.Light1, LightParameter.Position, position);
    }
    // Get rid of the light transform, restore the sphere translation
    GL.PopMatrix();

    Primitives.DrawSphere();
}
GL.PopMatrix();
GL.Disable(EnableCap.Light1);
```

That works, your application now looks the same, but the light is moving around. Take note of what happens when the blue light passes over a section of the sphere that's lit yellow. The geometry turns white! Why is that? Doesn't blue + yellow = green?



In the real world, that would be the case, but computers approximate color with RGB values. The sphere is lit yellow, which is RGB(1, 1, 0), then a blue light passes over it, blue is RGB(0, 0, 1). What happens when you combine (add) the two vectors RGB(1, 1, 0) + RGB(0, 0, 1), the result equals RGB(1, 1, 1), which is white. This is a really good example of where computer graphics fall short, by approximating colors into RGB channels we lose some of the properties that real world colors have.

The scene looks great, it's almost where we want to be, but it's really hard to tell where the blue light is. I mean, it's

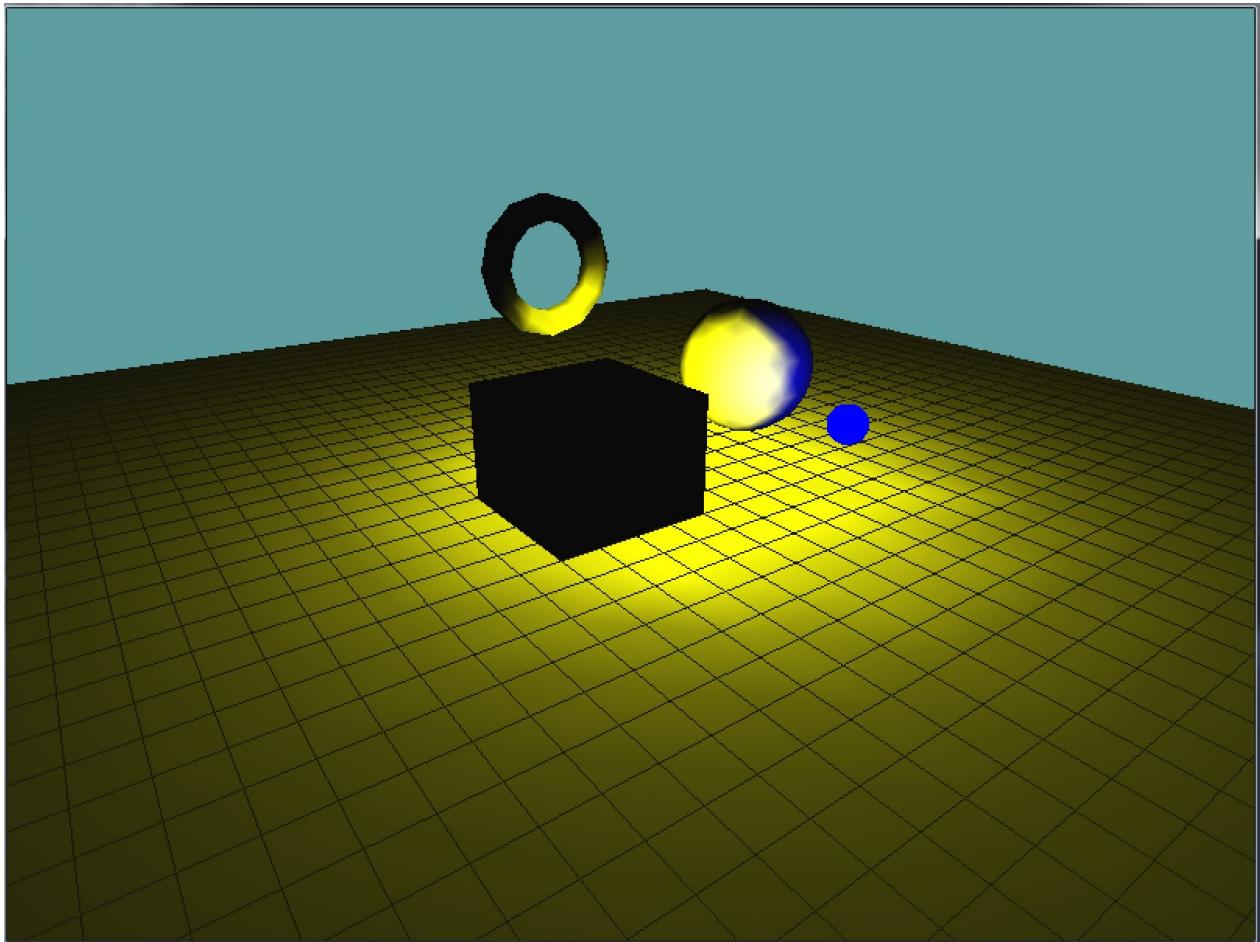
easy to see its effect on the sphere, but it's hard to tell where the actual light is. Let's fix that. We can visualize the blue light by rendering a small, unlit blue sphere at the lights position. This should be easy, since we already have the modelview matrix configured with the lights model matrix.

```
GL.Enable(EnableCap.Light1);
GL.Color3(1.0f, 0.0f, 0.0f);
GL.PushMatrix();
{
    GL.Translate(2.5f, 1.0f, -0.5f);
    GL.PushMatrix();
    {
        GL.Rotate(lightAngle, 0f, 1f, 0.0f);
        GL.Translate(-2f, 0f, -2f);

        position = new float[] { 0f, 0f, 0f, 1f };
        GL.Light(LightName.Light1, LightParameter.Position, position);

        // Disable lighting, we want the visualization sphere to be a solid color
        GL.Disable(EnableCap.Lighting);
        // We want the visual light sphere to be small
        GL.Scale(0.25f, 0.25f, 0.25f);
        // And blue
        GL.Color3(0f, 0f, 1f);
        // Draw the light visualization
        Primitives.DrawSphere();
        // Re-enable lighting for the rest of the scene
        GL.Enable(EnableCap.Lighting);
    }
    GL.PopMatrix();
    // This is still what draws the sphere
    Primitives.DrawSphere();
}
GL.PopMatrix();
GL.Disable(EnableCap.Light1);
```

Now it's easy to tell where the blue light is. The scene looks like this:

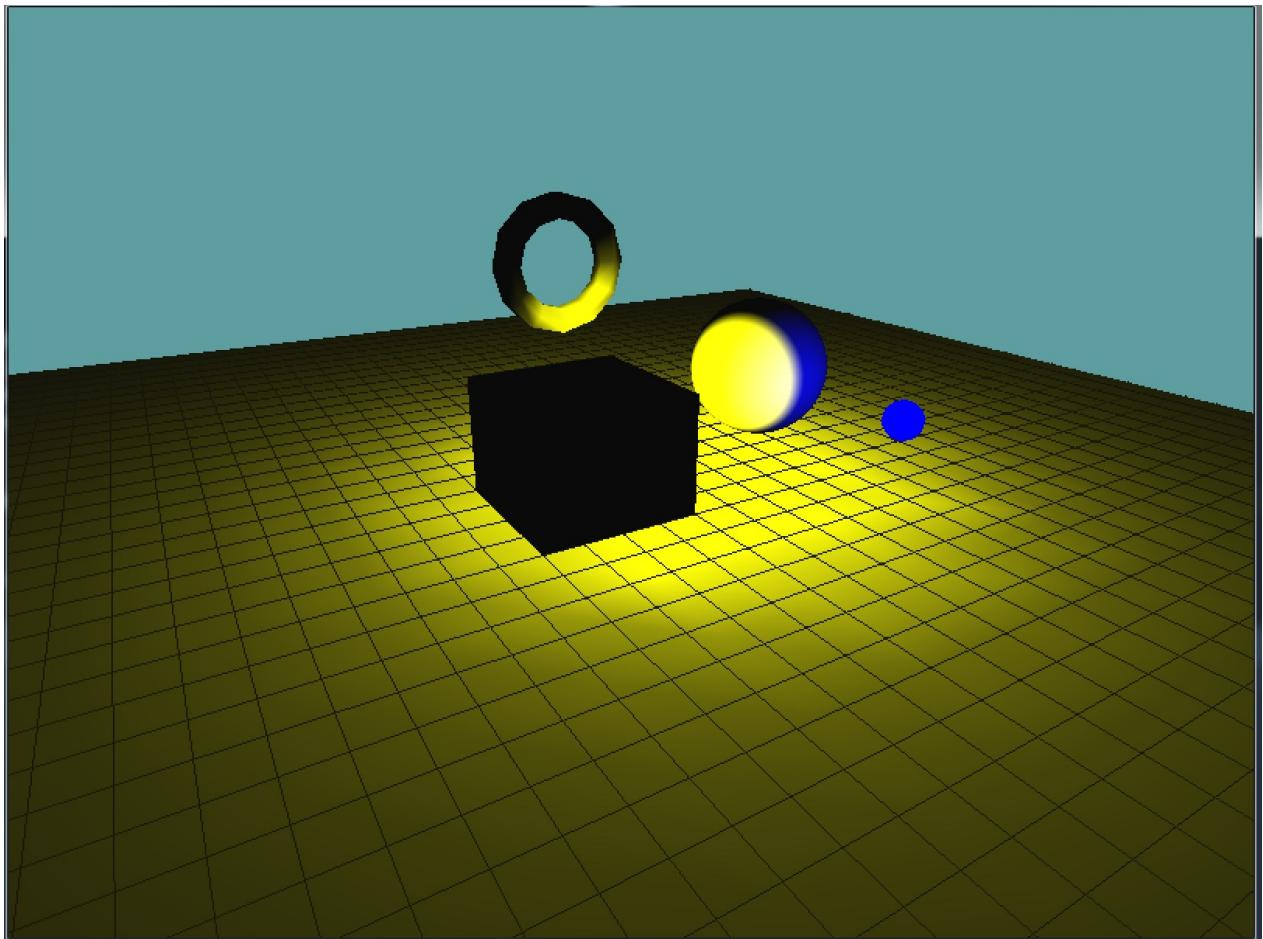


Details

There is one last modification i want to make to the above scene. Right now the lighting on the sphere looks kind of bad. That's because the sphere is made up of a small number of triangles. Remember, lighting is done on a per vertex basis, the further the vertices of a triangle are (that is, the larger a triangle is), the more the lighting has to interpolate. Interpolating essentially kills attenuation and causes the artifacts we see on screen. We can fix this by adding more triangles to the sphere. Change the draw call of the sphere (the one that is rendered, not the light preview) to sub-divide 5 times.

```
Primitives.DrawSphere(5);
```

The resulting scene now has much more accurate lighting for the sphere



Spot Light

The spot light is probably the least used of the three light types OpenGL provides. Why is that? Well, the attenuation calculation for a spot light is the same as that of a point light, but in addition to that OpenGL needs to do math to only light a cone, not a sphere. Limiting the light to a cone shape makes spot lights considerably more expensive than point lights. So much so, that things a spot light would be perfect for like street lamps are most often implemented with point lights instead.

So, if spot lights are more expensive, and point light generally have the same look to them, is there a reason to know about spot lights? Or even to implement them? Yes there is. Sometimes using a PointLight will not suffice. If the shape of the light matters, a spot light is the only way to go. Like luigis manor:



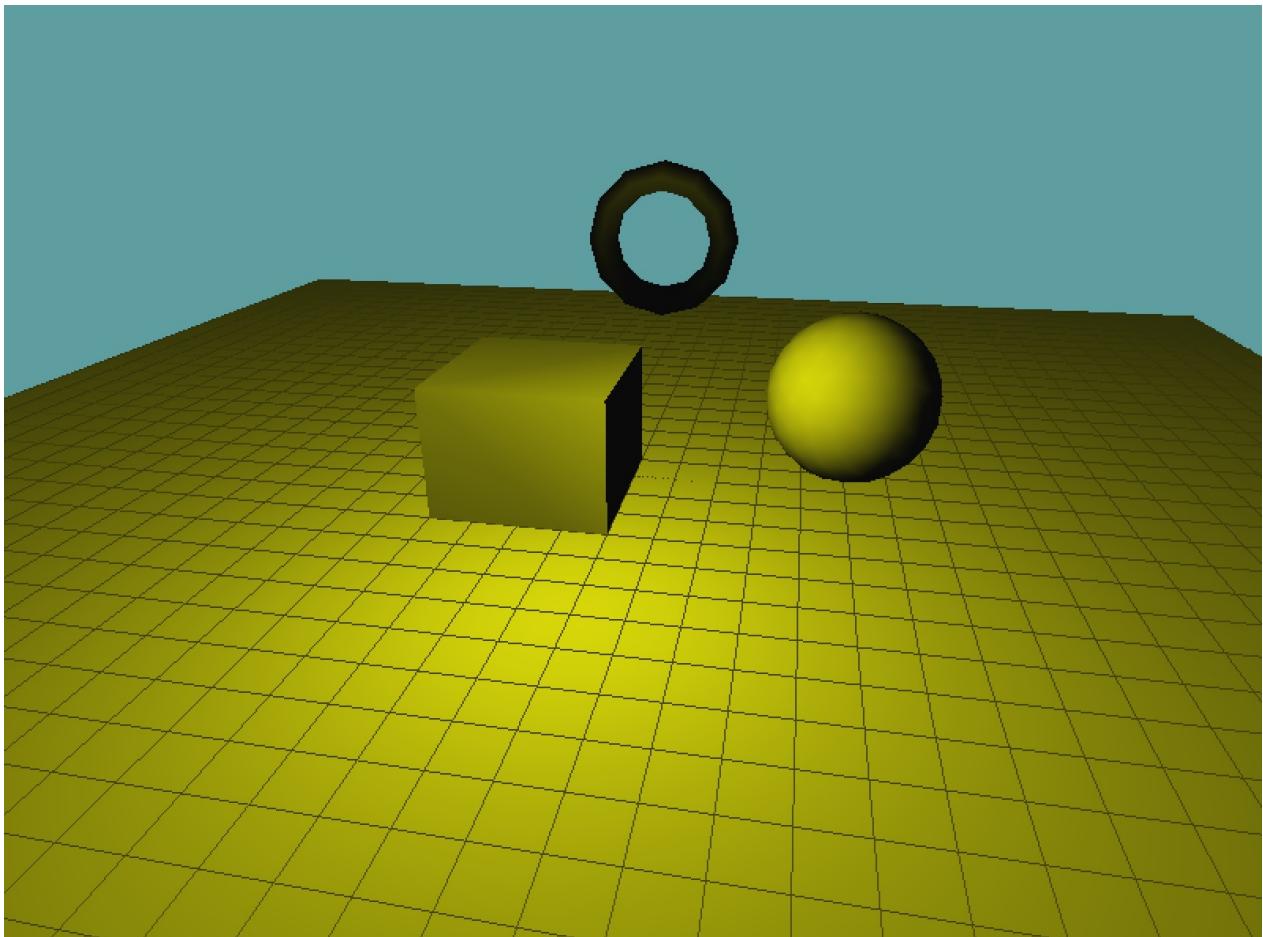
We're going to go through implementing a simple spot light together. All the advanced lighting topics (local lights, moving lights, etc...) have been covered in the Directional Light and Point Light sections. So this section should be pretty short.

Implementation

Lets start with the unlit test scene we set up in the beginning. Enable lighting and light 0. Configure light 0 to have a diffuse term of yellow, an ambient term of black and a specular term of white.

Set the position of this light to $(0, 3, 3, 1)$. Unlike the colors, you can't configure the lights position in the Initialize function. Just like a Point Light, a Spot light must be defined in eye space. Therefore you have to set it after the view matrix is set.

Your scene should be lit like this (once you let it rotate a bit):



Set your grid subdivision to as high as you can while still maintaining a smooth framerate. For me this is 8. At a subdivision level of 8 my game rotates smoothly, at 9 the frame gets super laggy. At 10 my scene moves maybe two seconds every frame and my graphics card catches fire.

Next, let's set the properties that make up a spot light, these are `SpotCutoff`, `SpotExponent` and `SpotDirection`. For a review of how these work, check the "Spotlights" section of "Light Sources", because they have already been covered there i'm not going to cover them here. We're just going to use them.

You can set these properties in the `Initialize` function, as they don't change frame to frame. We're going to set a cutoff of 15, an exponent of 5 and a direction of (0, -1, 0):

```
float[] direction = { 0f, -1f, 0f };
GL.Light(LightName.Light0, LightParameter.SpotCutoff, 15f);
GL.Light(LightName.Light0, LightParameter.SpotExponent, 5f);
GL.Light(LightName.Light0, LightParameter.SpotDirection, direction);
```

If you run your game and let it rotate you should see the below image. If your whole scene is black that means your grid's vertices are all outside of the spot lights reach, it needs to be sub-divided more. (You need a minimum of 4. At subdiv level 4 you will see a light blob). The below image is with subdivision level 8.



Cleanup

That's it. The above code is really all there is to spot lights! Let's do a few quick cleanup steps to pull this demo together.

First and foremost, it always helps to be able to visualize where a light is. To do this, we're going to render a small sphere at the position of the light, and a line in its direction. If you're feeling brave try to implement the code without looking at the below example.

Render the visualization geometry for the light after the grid is rendered:

```
public override void Render() {
    // ... Set eye position, configure look at matrix

    GL.Light(LightName.Light0, LightParameter.Position, pos);

    grid.Render(8);

    GL.Disable(EnableCap.Lighting);
    GL.PushMatrix();
    {
        GL.Translate(pos[0], pos[1], pos[2]);
        GL.Scale(0.25f, 0.25f, 0.25f);
        GL.Color3(1f, 1f, 0f);
        Primitives.DrawSphere();

        GL.Begin(PrimitiveType.Lines);

        GL.Vertex3(0f, 0f, 0f);
        // direction for me = new float[] {0, -1, 0};
```

```

Vector3 lightDirection = new Vector3(direction[0], direction[1], direction[2]);
lightDirection.Normalize();
GL.Vertex3(lightDirection[0] * 5.0f, lightDirection[1] * 5f, lightDirection[2] * 5f);

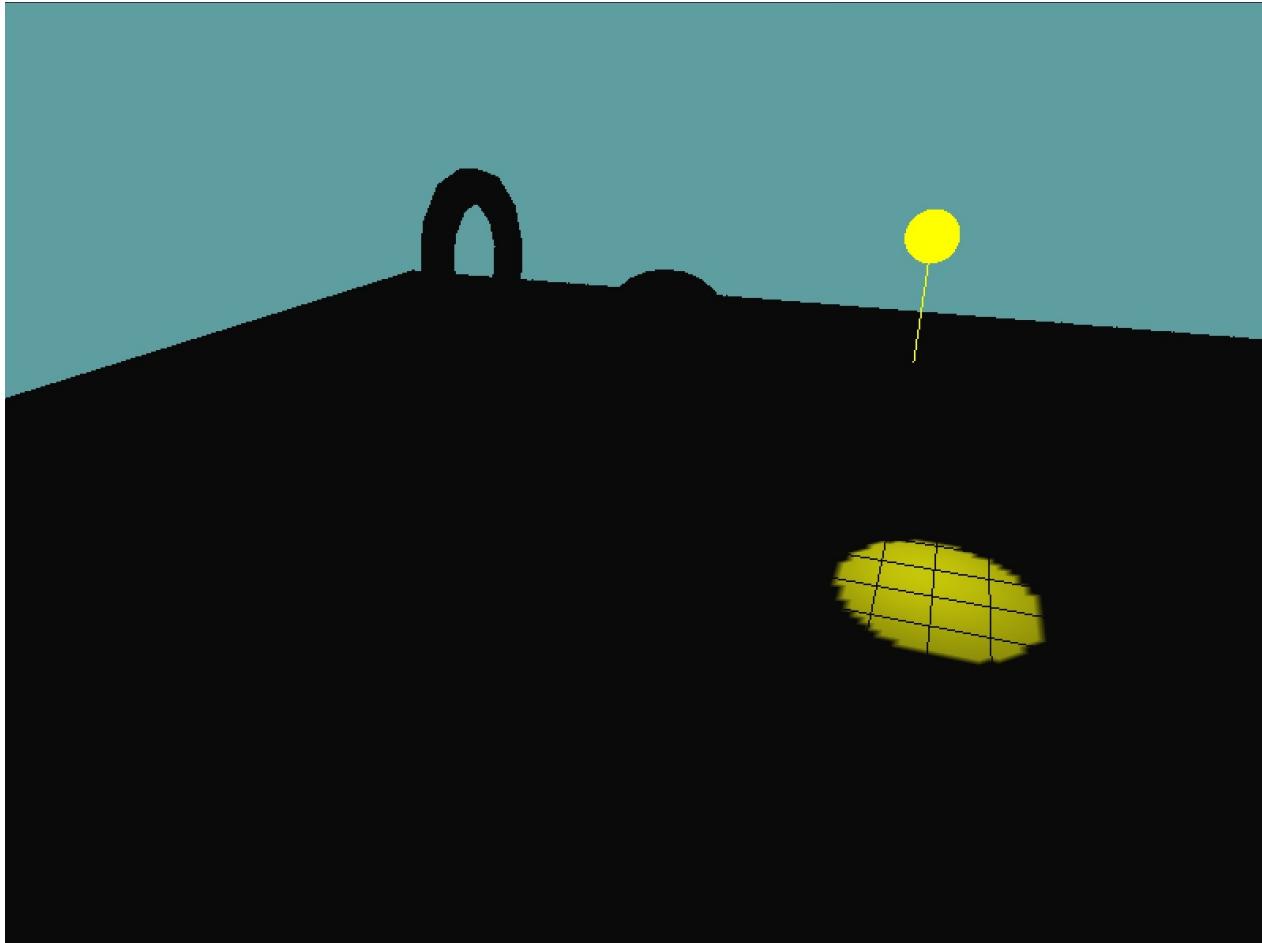
GL.End();
}

GL.PopMatrix();
GL.Enable(EnableCap.Lighting);

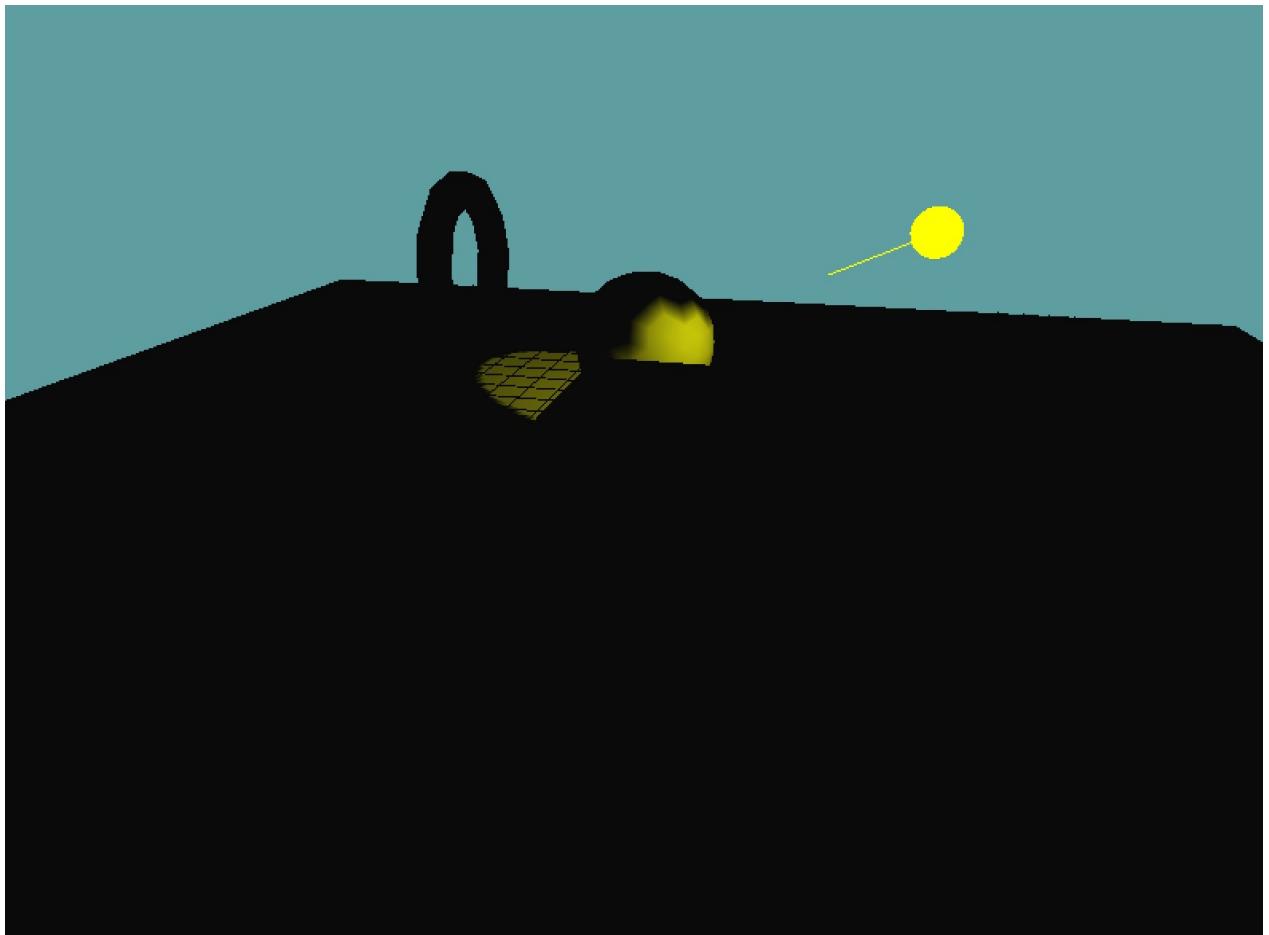
// ... Rest of code is unchanged

```

With the light visualization code in place, your scene should look like this:



And finally, let's tilt the light so it actually lights up some geometry other than the floor! Change the direction fo the light from $(0, -1, 0)$ to $(-2, -1, -3)$. The scene after rotating a bit will look like this:



There are a few artifacts left in here. For example, once the scene rotates enough the light visualizers direction points in a different direction than the light. Or when the light hits the edge of the ground plane, the grid indicators get lit (because the grid vertex will now fall into a light).

The above issues are expected, we're going to acknowledge that they exist, but it's not worth spending time to fix them.

Materials

Lights account for only half of the properties of what we see. The other half is materials. So far every object we've lit has been treated as a mostly white object. Reflecting light almost perfectly. But the real world doesn't work like that. Red light on a blue object might make that object look blue. Some objects are shiny, while other objects are matte. Objects can be assigned these properties through materials.

OpenGL approximates material properties based on the way the material reflects red, green and blue light. For instance, if you have a surface that is green, it will reflect all incoming green light, absorbing all blue and red light. If you were to place this surface under a purple light, it would appear black. This is because the surface only reflects green light. It absorbs the red and blue components of the purple light, there is no green light left to reflect.

If you were to place a green surface under a white light, it would still look green. This is because white is a combination of all colors. The surface will absorb the red and blue components of the light, and reflect the green.

Finally, placing this surface under a green light, it would still look green. There is no color for the surface to absorb, and all incoming green is reflected back at the viewer. It would look exactly the same as if we had placed the surface under a white light.

Materials have the same color terms as lights: **ambient**, **diffuse** and **specular**. These three properties combined will determine how much light a material reflects. A material with high ambient, low diffuse and low specular will reflect only ambient light sources while absorbing light from diffuse and specular sources.

A material with high specular reflectance will appear shiny, even when absorbing the ambient and diffuse light sources. The values defined for the **ambient & diffuse** reflectances (terms) of a material will **determine the color** of the material. They are **usually the same**.

To make sure that the specular highlights end up being the same color as the light being shined, the **specular** reflectance (term) is usually set to **white or gray**. A good way to think about this is a bright white light shining on a blue marble. While the marble is blue the reflected specular highlight remains white.

Defining materials

Now that you have a general understanding of what materials are, lets look at how to use them. Materials do not need to be enabled or anything, the truth is you have been using them all along, just with whatever the default values were. Now, we're going to set some non-default values.

Actually, setting a material is fairly similar to creating a light source. The main difference is the name of the functions being used:

```
void GL.Material(MaterialFace face, MaterialParameter param, float value);
void GL.Material(MaterialFace face, MaterialParameter param, float[] value);
```

The face parameter in these functions specifies how the material will be applied to the objects polygons, implying that materials can affect the front and back faces of polygons differently. It can be one of three values:

- Front
- Back
- FrontAndBack

Only the faces you specify will be modified by the call to `GL.Material`. Most often you will use the same values for front and back faces (and 90% of the time back faces will be culled, so the back face value won't matter). The second parameter `param` tells OpenGL which material property we are assigning a value to. This can be any of the following:

- **Ambient** Ambient color of the material
- **Diffuse** Diffuse color of the material
- **AmbientAndDiffuse** Ambient AND diffuse color of material
- **Specular** Specular color of material
- **Shininess** Specular exponent (power)
- **Emission** Emissive color

Material Colors

The ambient, diffuse and specular components specify how a material interacts with a light source and, thus determine the color of the material. These values are set by passing `Ambient`, `Diffuse` OR `AmbientAndDiffuse` to `GL.Material` as the `MaterialParameter`. Most often the same values are used for both the ambient and diffuse term, so much so that OpenGL provided a convenience parameter `AmbientAndDiffuse` so you can specify both with 1 function call.

For example, if you wanted to set the ambient material color to red for the front and back of polygons, you would call this function:

```
float[] red = { 1f, 0f, 0f, 1f };
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, red);
```

Similarly, to set both the ambient and diffuse parameters to blue would be:

```
float[] blue = { 0f, 0f, 1f, 1f };
```

```
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, blue);
```

Keep in mind that any polygons you draw after calling GL.Material will be affected by the material settings until the next call to GL.Material

Shininess

Try looking at something metallic and something cloth under a direct light. You'll notice that the metallic object appears to be shiny, while the cloth isn't. This is because the light striking the cloth object is mostly scattered by the rough cloth surface, whereas light is reflecting directly on the smooth metal surface.

This image demonstrates the effects of different Specular material colors and Shininess specular exponent values.

- Left: Zero-intensity (0., 0., 0., 1.) specular material color
 - specular exponent is irrelevant.
- Center: Low-intensity (.3, .3, .3, 1.) specular material color
 - with a specular exponent of 10.0.
- Right: Full-intensity (1., 1., 1., 1.) specular material color
 - with a specular exponent of 128.0.



The sphere on the right looks like metal. The illusion of shininess is caused by the bright spot, known as a **specular highlight**. The sphere on the left is a more cloth-like material (t-shirt maybe), and therefore looks more dull. The middle sphere gives the look of most plastics.

The shininess of a material is simulated by the size of the specular highlight. This is controlled with a SINGLE scalar value, which you set with the `MaterialParameter.Shininess` parameter. This value can range from 0 to 128 (NOT 0 to 1), with 128 representing an extremely shiny material and 0 representing a non shiny material.

You would make an object metallic like this:

```
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 128);
```

Emissive materials

The emissive property of a material allows you to cheaply simulate an object that emits light (thing TRON or maybe a glow in the dark toy). It's important to note that the objects don't really emit light. That is, they will not light up or affect nearby objects.

The emissive term is simply added to the objects other lighting components to make the object appear brighter than normal. Here is an example of setting a dark light emission

```
float[] emissiveColor = { .3f, .3f, .3f, 1f };
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Emission, emissiveColor);
```

By default the emissive term is (0, 0, 0, 1), which means it has no contribution and is therefore effectively turned off.

Color Tracking

Note, *color tracking* is not an official term. You will probably not hear it used outside the context of this book. But it's an easy way to describe setting the material color with a call to `GL.Color`.

Color Tracking allows you to set the color of a material with calls to `GL.Color` instead of using `GL.Material`, which often leads to smaller, more easy to read code. You can enable color tracking with the following call:

```
GL.Enable(EnableCaps.ColorMaterial);
```

After enabling color tracking you use the `GL.ColorMaterial` function to specify which material parameters will be affected by future calls to `GL.Color`. The signature of this function is:

```
void GL.ColorMaterial(MaterialFace face, ColorMaterialParameter param);
```

Like always, the face can be set to `Front`, `Back` OR `FrontAndBack`. The mode parameter can be:

- Ambient
- Diffuse
- Specular
- AmbientAndDiffuse
- Emission

Most often you will be using the default values of this function (so you might not even have to call it). The default values are `FrontAndBack`, `AmbientAndDiffuse`. Let's look at some sample code that uses color tracking to set the diffuse color of an object's material:

```
GL.Enable(EnableCap.ColorMaterial);
GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.Diffuse);
GL.Color3(1f, 0f, 0f);
Primitives.DrawSphere();
```

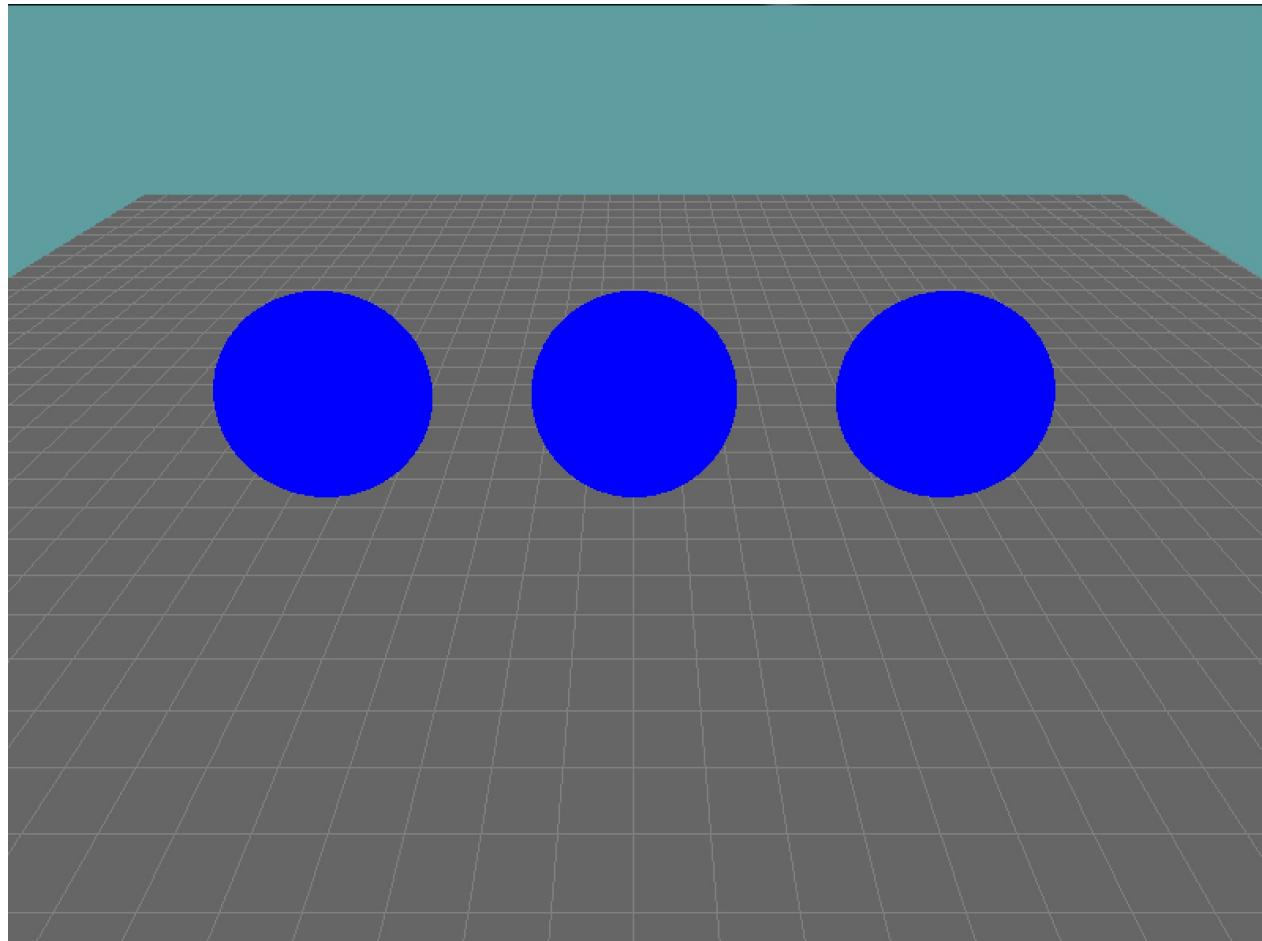
As you can see, color tracking is very simple to set up and use. I almost never use it, I think giving multiple utilities to the same function is a bad idea. I like to explicitly type **Material** when I'm setting a material color component.

Test scene

Lets go ahead and set up a test scene. This scene will be used as the base for all our shading exercises. We're going to render 3 spheres, each will be shaded differently throughout the exercises.

- You should be in a perspective projection, with a 60 degree field of view
 - The base class already does this, but review the code.
 - I'm concerned that this bit isn't being practiced!
 - Take note that it's set in the constructor and resize function
 - Also, pay attention to the viewport. Every time projection is set, so is the viewport
- Set up your view matrix
 - The camera should be positioned at (0, 5, -7)
 - The camera should be looking at (0, 0, 0)
- Add a solid grid as the ground.
 - You can use a subdivision of 0, we're only using directional lights
- Render 3 spheres, each with a subdivision level of 3
 - One at (-3, 1, 0)
 - One at (0, 1, 0)
 - One at (3, 1, 0)
- Make sure the spheres render blue

Running your game, you should see this:



I use a subdivision level of 3 as i think it's a good mix between performance and looks. If you want to see a specific

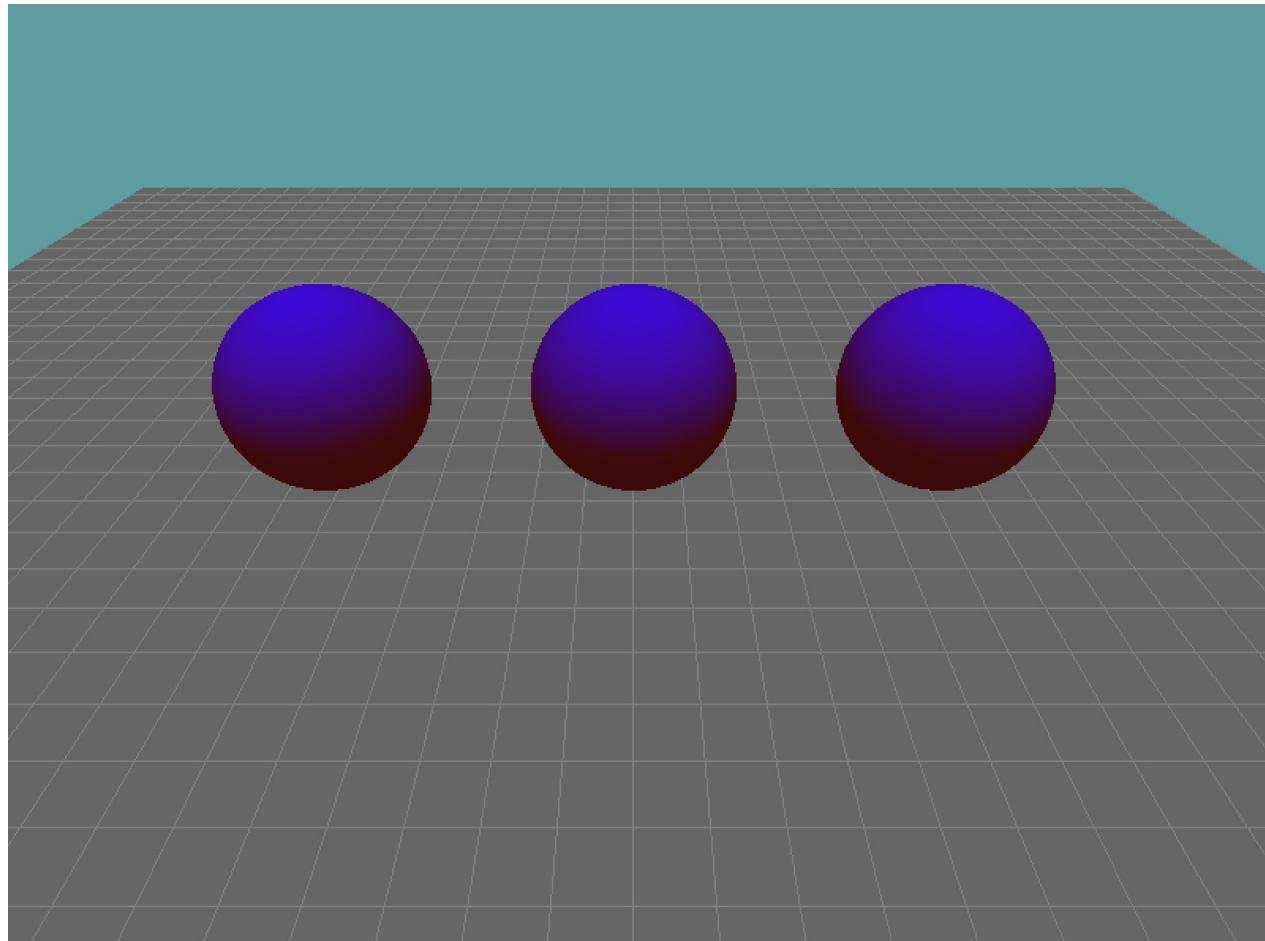
scene rendered WAY more accurately (and pretty), temporarley set your subdiv level to 4 or 5. If the performance cost doesn't bother you, you might even just want to have them sitting on 4.

The camera for this scene **SHOULD NOT ROTATE**

With the basic scene set up, lets add a light. We're going to add a red light to the scene. It's going to shine directly onto the spheres.

- Enable lighting
- Enable light 0
 - All configuration from here on will be affecting light 0
- It's a directional light. It should shine from the direction {0, 1, 1}
 - Remember, the difference between a directional and a point light is the W component
- Set the ambient color to red
- Set the diffuse color to blue
- Set the specular color to white (1, 1, 1)
- Disable lighting for the ground

The default values for everything else are fine. Your scene should now look like this:



Ambient and diffuse

Now that you know how materials work, lets take some time and actually implement some code using materials!

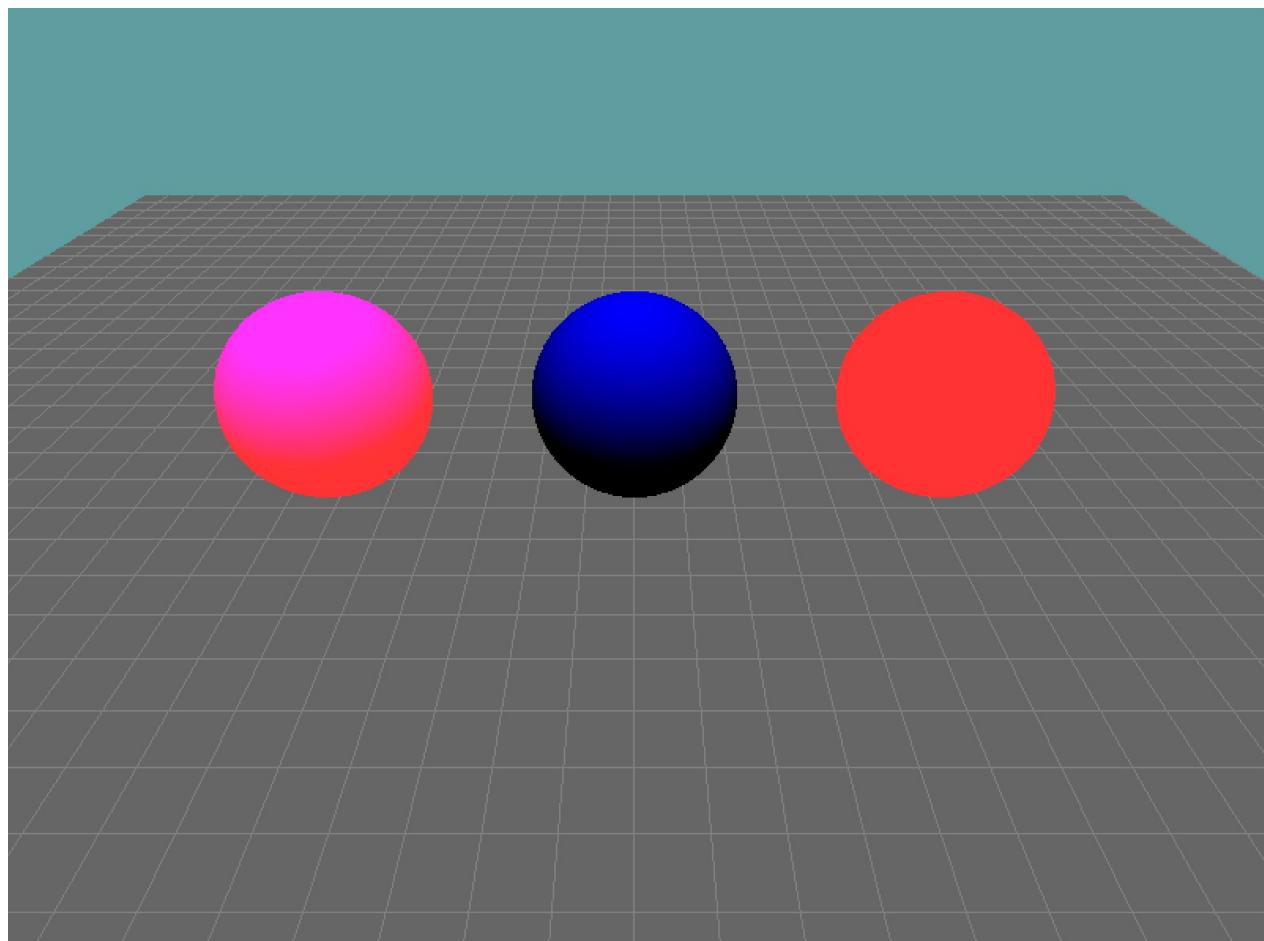
Let's get started by making a new demo class we'll call it `MaterialColors`. Get the code in it up to par with the test scene.

Once you have that, we're just going to play around with material colors. Take note, when setting the color components, OpenGL expects a 4 component (RGBA) array of floating point values (0.0f to 1.0f)

First, lets set the following materials

- Sphere 1
 - Set the material ambient to white
 - Set the material diffuse to black
- Sphere 2
 - Set the material ambient to black
 - Set the material diffuse to white
- Sphere 3
 - Set the material ambient to white
 - Set the material diffuse to white

Try setting the material parameters yourself first. The code for how to do it is given below the image, but you should be able to set these without the sample code. The scene will look like this:



The code to set the material color parameters, looks like this:

```
// Set up camera
// Render unlit grid

GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, new float[] { 1, 1, 1, 1 });
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Diffuse, new float[] { 0, 0, 0, 1 });
// Render first sphere

GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, new float[] { 0, 0, 0, 1 });
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Diffuse, new float[] { 1, 1, 1, 1 });
// Render second sphere

GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, new float[] { 1, 1, 1, 1 });
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Diffuse, new float[] { 1, 1, 1, 1 });
// Render third sphere
```

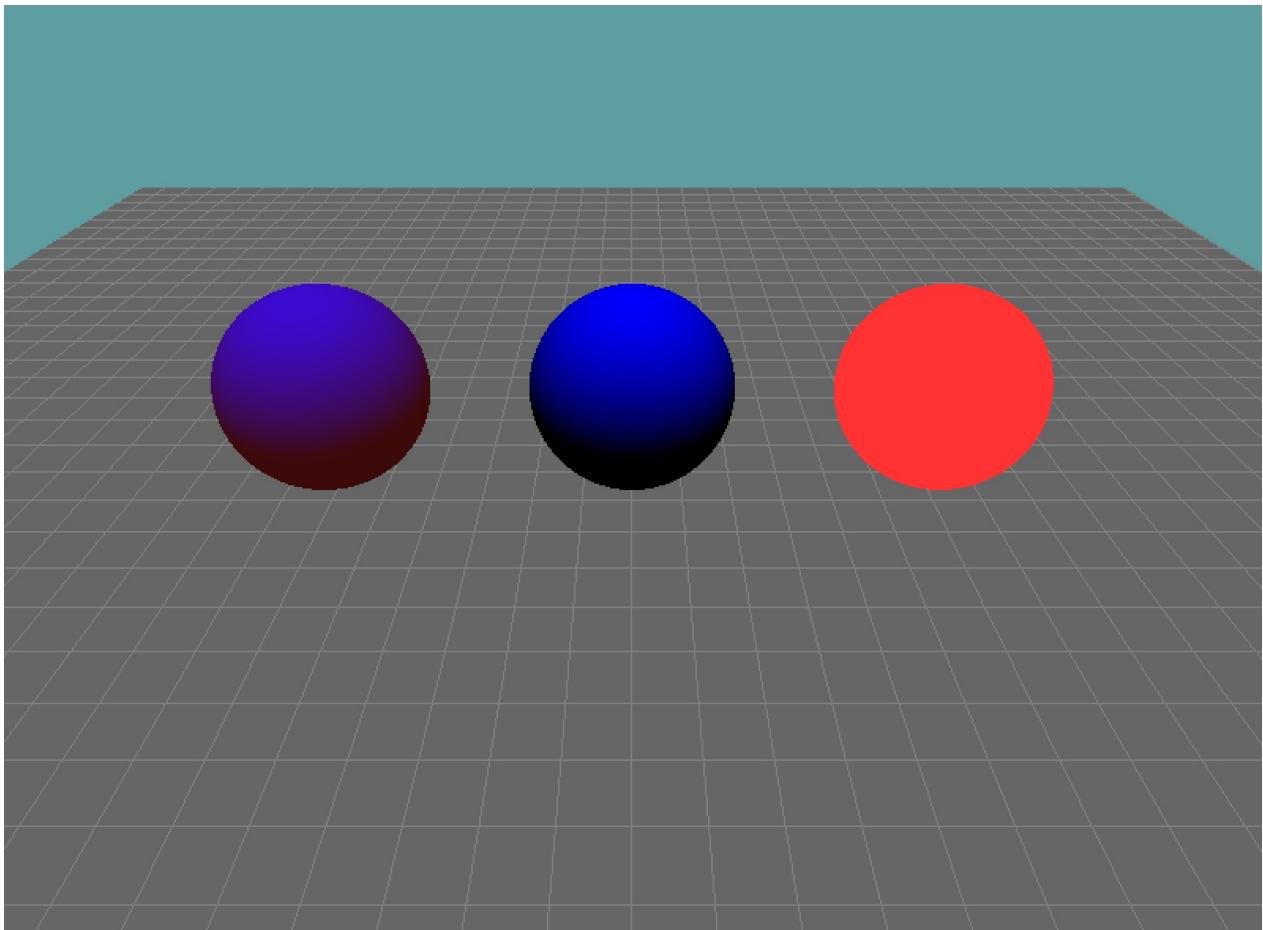
Like we discussed earlier, the materials ambient property determines how much of the lights ambient property effects the object. Same for all other properties. When this is set to white, it means the lights ambient fully effects the object, when it's set to black it means the ambient does not affect the object at all.

A few important observations can be made from this code. Notice that the sphere only affected by ambient lighting isn't shaded. It's solid. That's because ambient light is constant, it doesn't really have a source! Notice, the diffuse lighting IS shaded. This is because the diffuse is the directional component of the light.

Lastly, take a look at the third sphere. We have both ambient and diffuse lights effecting this, but it looks AWFUL! Shouldn't it look like the default lit sphere? No, not exactly. By default the lighting model sets the following values

- Ambient - (0.2, 0.2, 0.2, 1.0)
- Diffuse - (0.8, 0.8, 0.8, 1.0)

The two values when added to come out to (1, 1, 1, 1). This is purely coincidental, not a requirement! Go ahead, update the render code for that last sphere to reflect these numbers. All of a sudden the scene looks like this:



The material code was changed to:

```
// Set up camera  
// Render unlit grid  
  
// Render first sphere  
// Render second sphere  
  
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, new float[] { 0.2f, 0.2f, 0.2f, 1 });  
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Diffuse, new float[] { 0.8f, 0.8f, 0.8f, 1 });  
// Render third sphere
```

Now that third sphere looks the same as it did in the test scene. You can really see how the shading works. To get the final color of this sphere. Let's walk through how this color was obtained.

First we take the ambient component of the light (1, 0, 0) and multiply it with the ambient component of the material (0.2, 0.2, 0.2). The result is the objects ambient component (0.2, 0, 0).

Next we take the diffuse component of the light (1, 0, 1) and multiply it with the diffuse component of the material (0.8, 0.8, 0.8). The result is the objects diffuse component (0, 0, 0.8)

Finally, to get the color of the object, we add the ambient and diffuse components of the object together, resulting in (0.2, 0, 0.8). The color interpolation (dark on bottom, light on top) is contributed by the diffuse component.

Play around with the colors a bit. See if you can predict what your code will look like once its on screen.

Specular

The ambient and diffuse components of the material define the overall color of the object. The specular component defines how shiny the object is.

Let's make a new demo class we'll call it `MaterialSpecular`. Get the code in it up to par with the test scene.

When setting the Ambient and Diffuse colors, OpenGL expected a floating point array. Setting the specular is a two part process. You must set the color (RGBA array) and the power (single float, ranging from 0 to 128).

Quick recap, you set the specular color with:

```
GL.Material(MaterialFace, MaterialParameter.Specular, float[]);
```

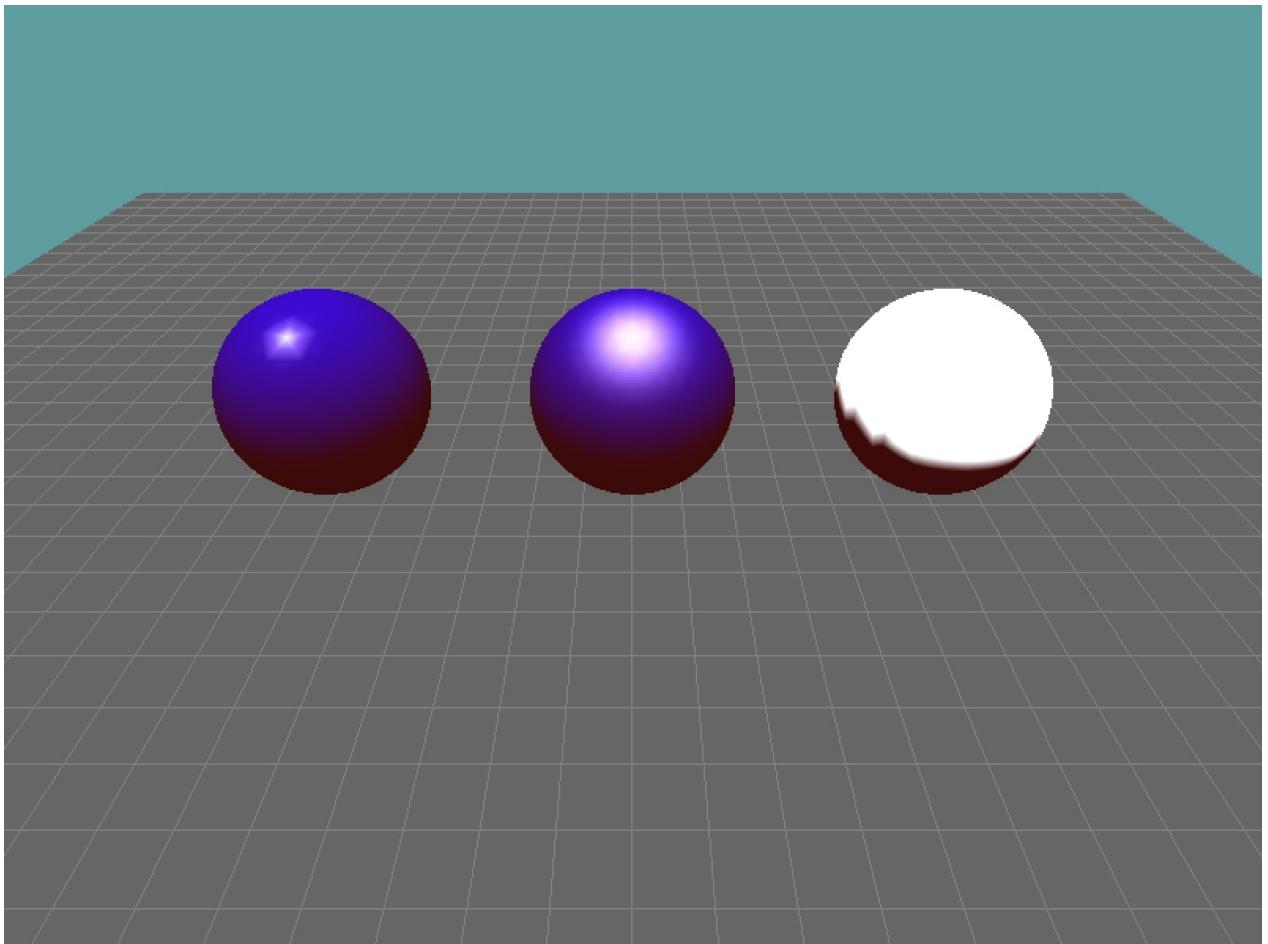
And the specular power with:

```
GL.Material(MaterialFace, MaterialParameter.Shininess, float);
```

Simple Specular

- Sphere 1
 - Set ambient and diffuse to default values
 - Set specular color to white
 - Set specular power to 0
- Sphere 2
 - Set ambient and diffuse to default values
 - Set specular color to white
 - Set specular power to 16
- Sphere 3
 - Set ambient and diffuse to default values
 - Set specular color to white
 - Set specular power to 128

If you set up the properties correctly, your scene should look like this:



To get to that, i set the above material components like so:

```
// Setup camera
// Render unlit grid

// Set ambient and diffuse
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1, 1, 1, 1 } );
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 0.0f);
// Draw sphere 1

// Set ambient and diffuse
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1, 1, 1, 1 } );
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 16.0f);
// Draw sphere 2

// Set ambient and diffuse
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1, 1, 1, 1 } );
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 128.0f);
// Draw sphere 3
```

The sphere with a specular power of 0 is all white. That's because with a low specular power, the specular component dominates the object's color.

The sphere with a specular power of 16 looks like it's lit strongly, the light is dispersing on it evenly.

The sphere with a specular power of 128 looks like a real hard light is shining on it, as the light reflection is super tiny.

The lower your specular power, the larger the specular color will be accross the surface of your object.

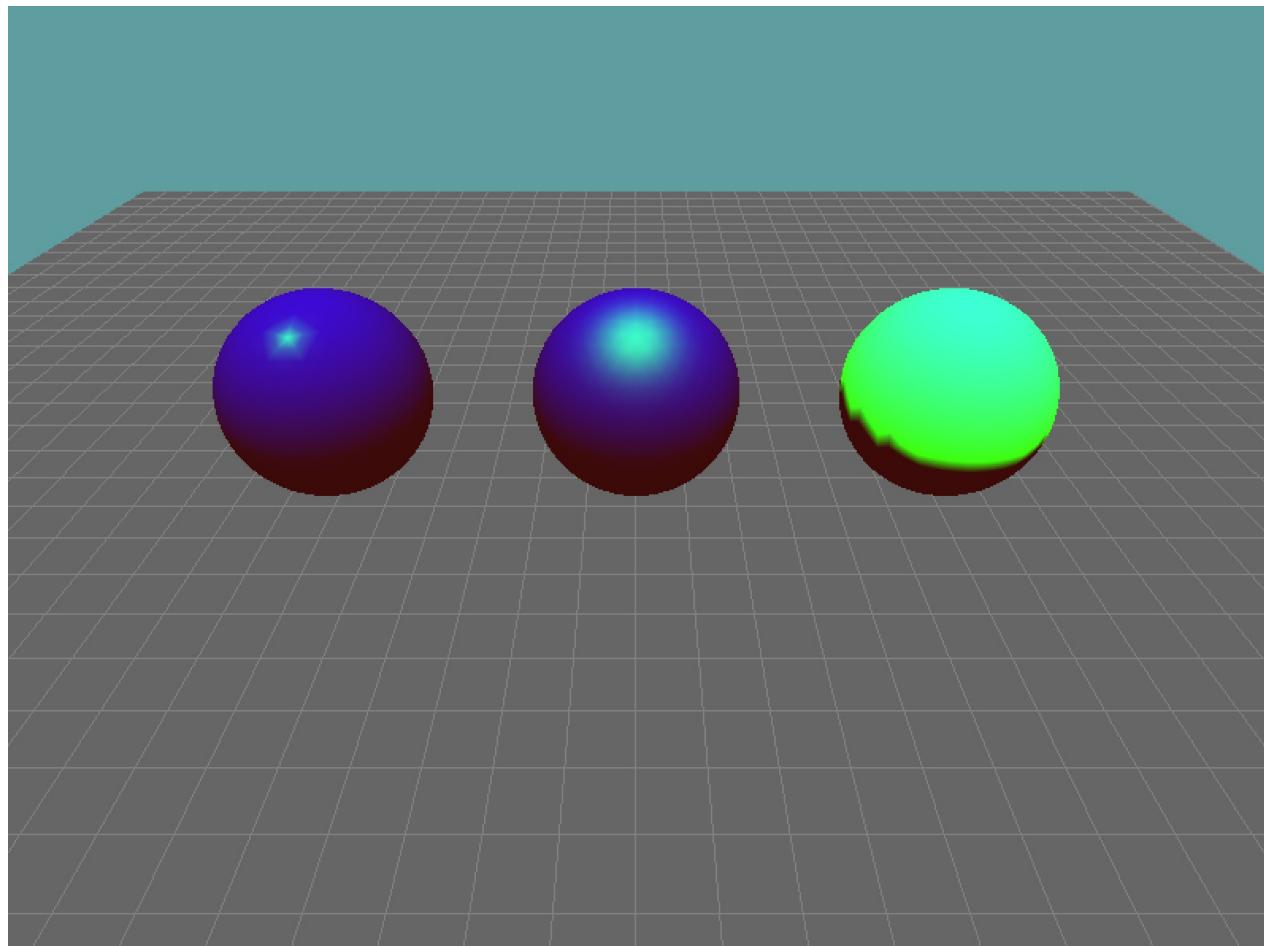
Specular color

Specular color works much like ambient and diffuse. First we take the specular color of the light ($1, 1, 1, 1$), and multiply it with the specular component of the material ($1, 1, 1, 1$). In our example, the object specular color is white.

This object specular color is then added to the object's ambient and diffuse colors. The result is $(0.2, 0, 0) + (0, 0, 0.8) + (1, 1, 1, 1) = (1, 1, 1)$. Notice, the color is capped to 1. This is why the spheres reflect the light as white.

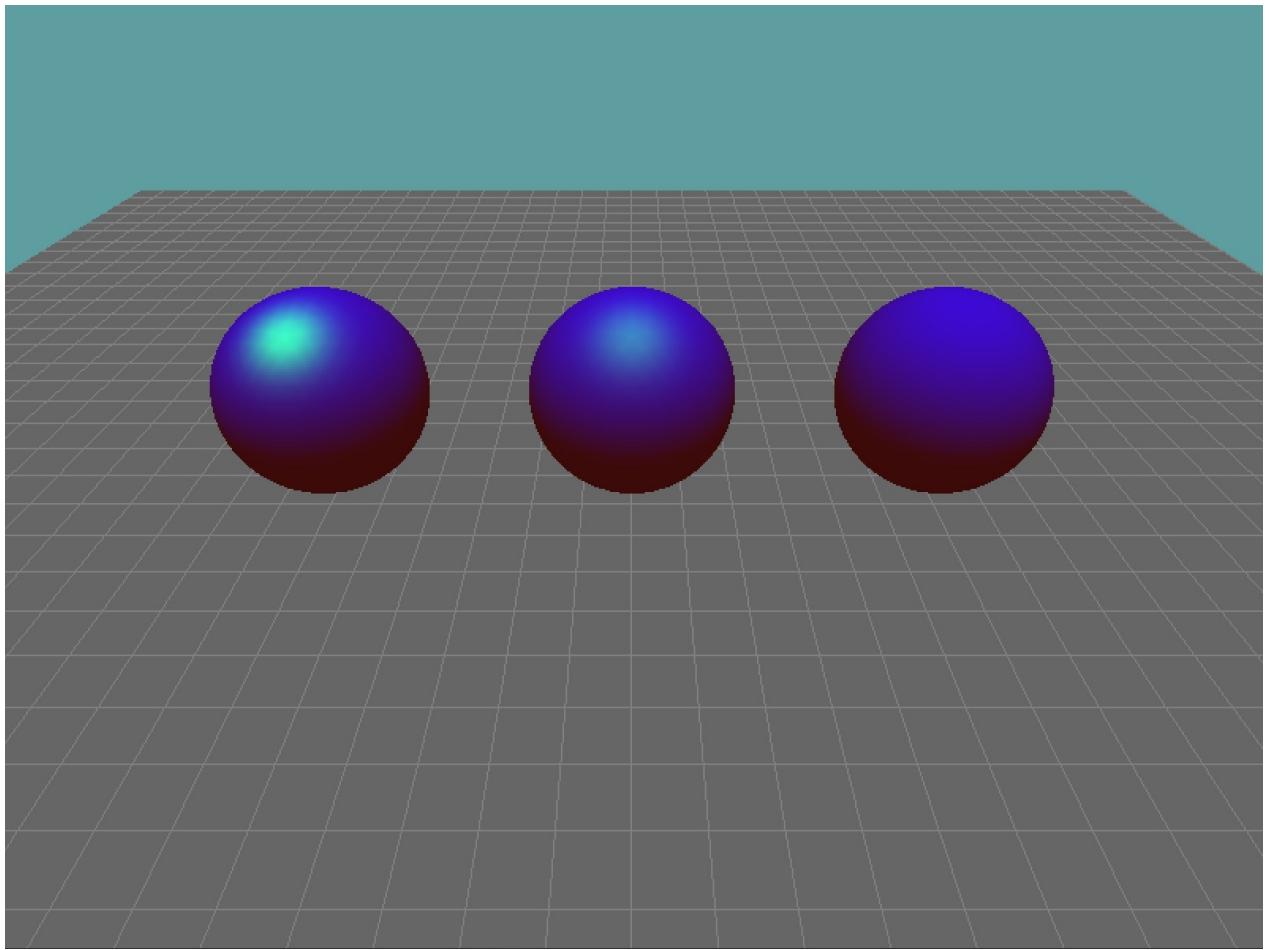
The amount of surface area that is shaded by the specular color is determined by the specular power (shininess). The higher the power, the less of the surface area is shaded.

Let's have some fun, and try to reflect a different color! Where you set up the light, currently it's specular color is white. Set the lights specular color to green!



- Keep the specular color of the light green
- Set the specular power for all the spheres to 16.
- Set the specular color of the first sphere's material to black
- Set the specular color of the second sphere's material to $(0.5, 0.5, 0.5)$
- Set the specular color of the third sphere's material to white

The result looks like this:



By reducing the amount of specular color a sphere takes to 0.5, we dulled out the specular highlight. This makes the middle sphere look more rough, like a plastic as opposed to the first sphere which looks metal.

The last sphere has no specular contribution. This is because the specular color of the light (0, 1, 0) multiplied with the specular component of the material (0, 0, 0) results in black (0, 0, 0). This means it does not affect the lighting.

Color Tracking

We call it color tracking, but more often than not people simply refer to this technique as **Changing material parameters with GL.Color**, sometimes it's also referred to as **color material mode**

Earlier we saw that OpenGL uses the current material color (set with `GL.Material`) when lighting is enabled, but uses the current primary color (set with `GL.Color`) when lighting is disabled. This inconsistency can often get confusing. It's also slightly more expensive to call `GL.Material` than it is to call `GL.Color` because of the safety checks the functions do. Whenever you change a material's color, most often you only change the diffuse and ambient components.

Color material mode addresses these issues. When you enable color material mode, certain current material colors track the current primary color instead. To use color materials, you must first enable them:

```
GL.Enable(EnableCaps.ColorMaterial);
```

This will cause calls to `GL.color` to change not only the primary color, but also the current ambient and diffuse material colors. Of course you can change which properties color tracking affects by calling `GL.ColorMaterial`.

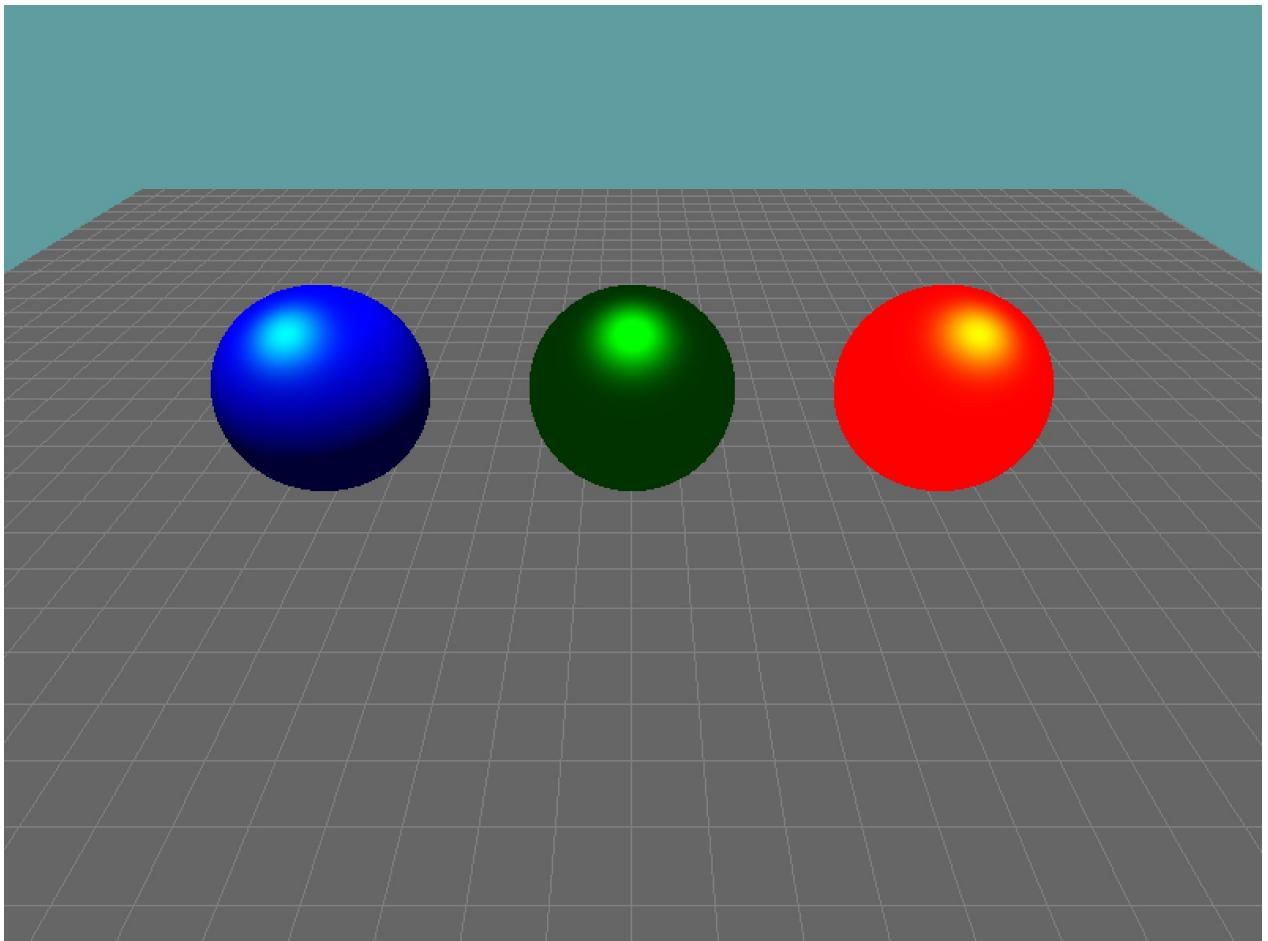
We've already discussed how to change which color is tracked in the **Defining Materials** section.

Example 1

Let's make a new demo scene, call it `ColorTrackingDemo` and get it up to par with the test scene. Once you have that:

- In the initialize function
 - Call `GL.Mateiral` to:
 - Set the specular component to white
 - Set the shininess to 20
 - Enable color material
 - Set color materials to track the ambient AND diffuse components for front and back face
- In the render function
 - Set the first sphere's render color to red
 - Set the second sphere's render color to green
 - Set the third sphere's color to blue

Your scene should look like the below image:



In case your scene doesn't look like that, here is the code i used to get that result

```
public override void Initialize() {
    base.Initialize();

    // Enable lighting and configure light 0
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1, 1, 1, 1 });
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 20.0f);

    GL.Enable(EnableCap.ColorMaterial);
    GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.AmbientAndDiffuse);
}

public override void Render() {
    // Set up view matrix

    // Render unlit grid
    GL.Color3(1f, 0f, 0f);
    // Draw first sphere

    GL.Color3(0f, 1f, 0f);
    // Draw second sphere

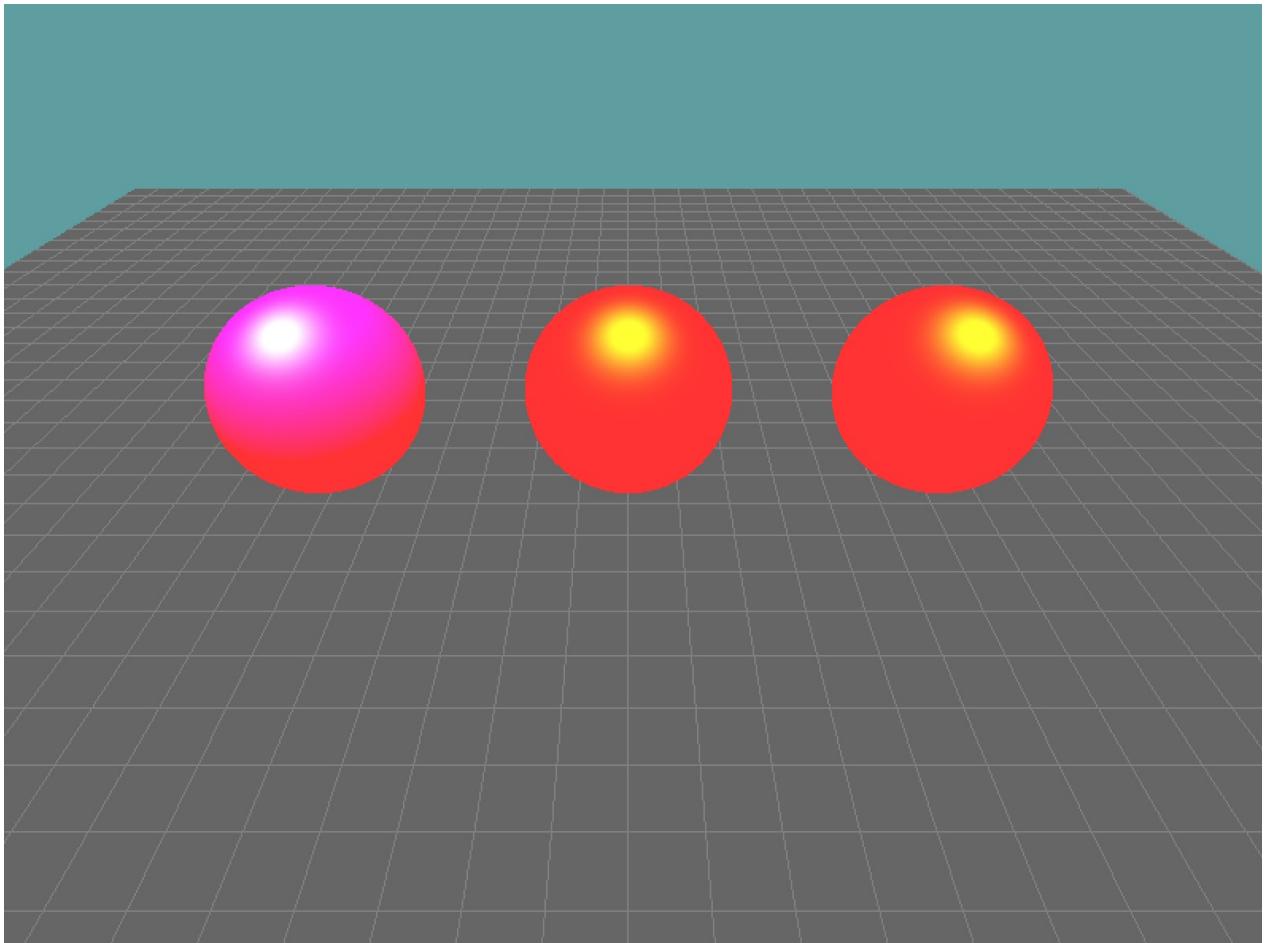
    GL.Color3(0f, 0f, 1f);
    Draw third sphere
}
```

The most important thing to note here is the shading. Because of the way the shading model works, the blue sphere gets a lot of diffuse light, the other two get too much ambient. Let's fix that!

Example 2

Using Example 1 as our starting point.

- In initialize
 - Change the color material tracking point to diffuse
 - Set the materials default ambient to { .2f, .2f, .2f, 1f }



Here is the code change

```
// This is new
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, new float[] { .2f, .2f, .2f, 1f });
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1, 1, 1, 1 });
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 20.0f);

GL.Enable(EnableCap.ColorMaterial);
// This changed
GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.Diffuse);
```

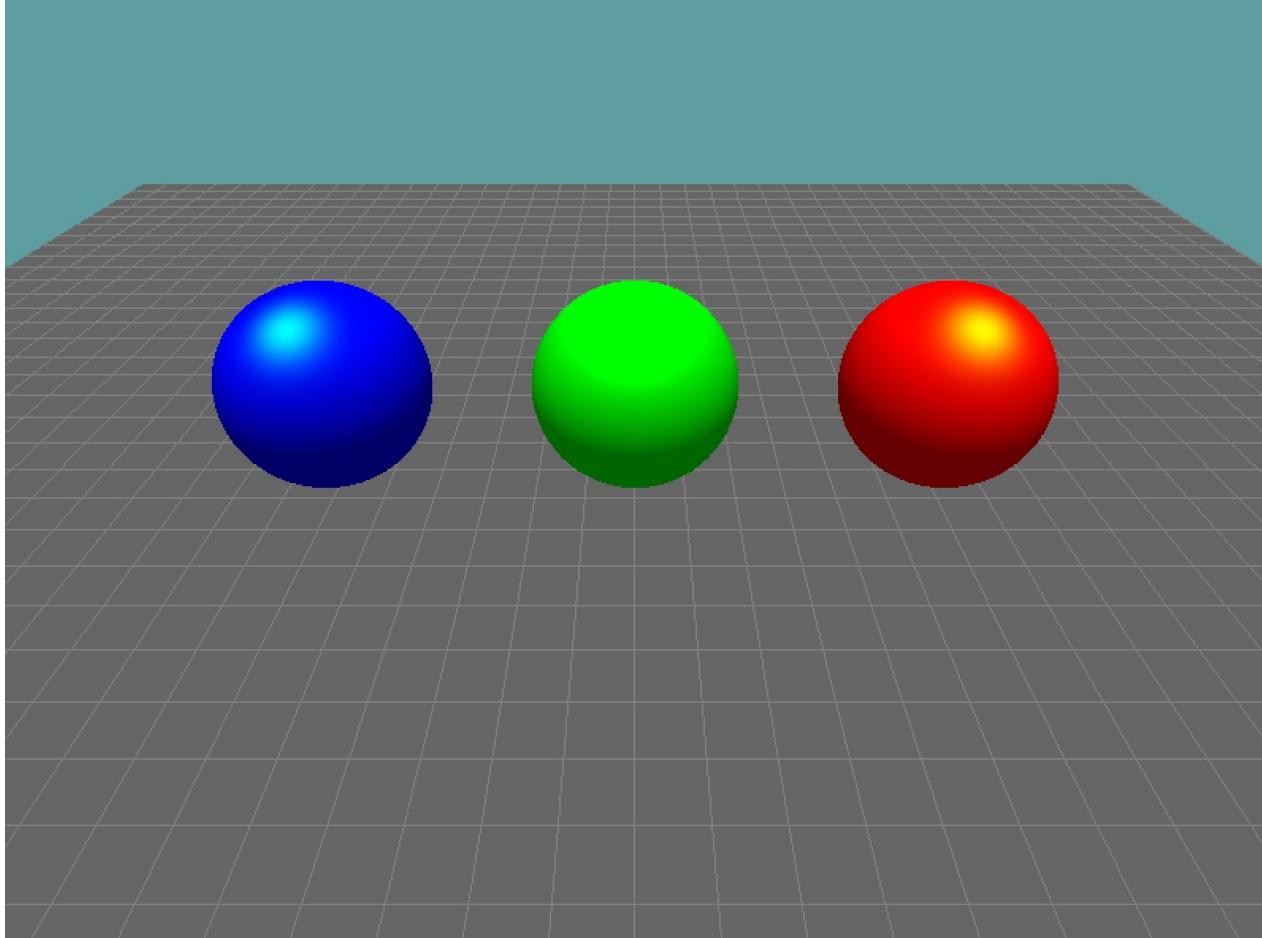
This still doesn't look great. All the spheres now take a uniform ambient term, but only the blue sphere has a blue term to take for its diffuse! That's why the other two look red-ish.

Example 3

The key to good material usage is to think about color a little differently! Instead of thinking about a light having a color and a material having a color, think of the light as an overall brightness (always have a white light), and the material as the color.

We're going to be using example 1 (not example 2!) as a starting point. That is, the color material is going to track both the ambient and diffuse terms. The sphere colors are set with `GL.Color3`, they are: red, green and blue. This is the starting point.

Now, change the LIGHT (not material) ambient color to `{ 0.2f, 0.2f, 0.2f, 1.0f }` and the LIGHT (not mateiral) diffuse color to `{0.8f, 0.8f, 0.8f, 1.0f }`. Run the game and your scene looks the way one would expecte it:



Only the initialize function changed from example 1, here it is in its entirety:

```
public override void Initialize() {
    base.Initialize();

    GL.Enable(EnableCap.Lighting);
    GL.Enable(EnableCap.Light0);

    float[] red = new float[] { 1, 0, 0, 1 };
    float[] blue = new float[] { 0, 0, 1, 1 };
    float[] white = new float[] { 0, 1, 0, 1 };

    // We changed the LIGHT! Not the Material
    GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0, 1, 1, 0 });
    GL.Light(LightName.Light0, LightParameter.Ambient, new float[] { .2f, .2f, .2f, 1f });
    GL.Light(LightName.Light0, LightParameter.Diffuse, new float[] { .8f, .8f, .8f, 1f } );
    GL.Light(LightName.Light0, LightParameter.Specular, white);

    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1, 1, 1, 1 });
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 20.0f);

    GL.Enable(EnableCap.ColorMaterial);
    GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.AmbientAndDiffuse);
}
```

This works because the light now defines how strong the ambient and diffuse components are. The actual material defines the color. Because if the material diffuse is blue (0, 0, 1) and the light diffuse is (.8, .8, .8) then the two multiplied together create the objects diffuse (0, 0, .8).

This (example 3) is the configuration i do 90% of my lighting with! It's something you should be comfortable setting up.

Normals

We've already discussed normal vectors a little bit as a mathematical concept, but seeing how normals are the key to realistic lighting, they warrant further discussion.

Normals in terms of lighting are vectors that are perpendicular to a surface, and have a length of 1.

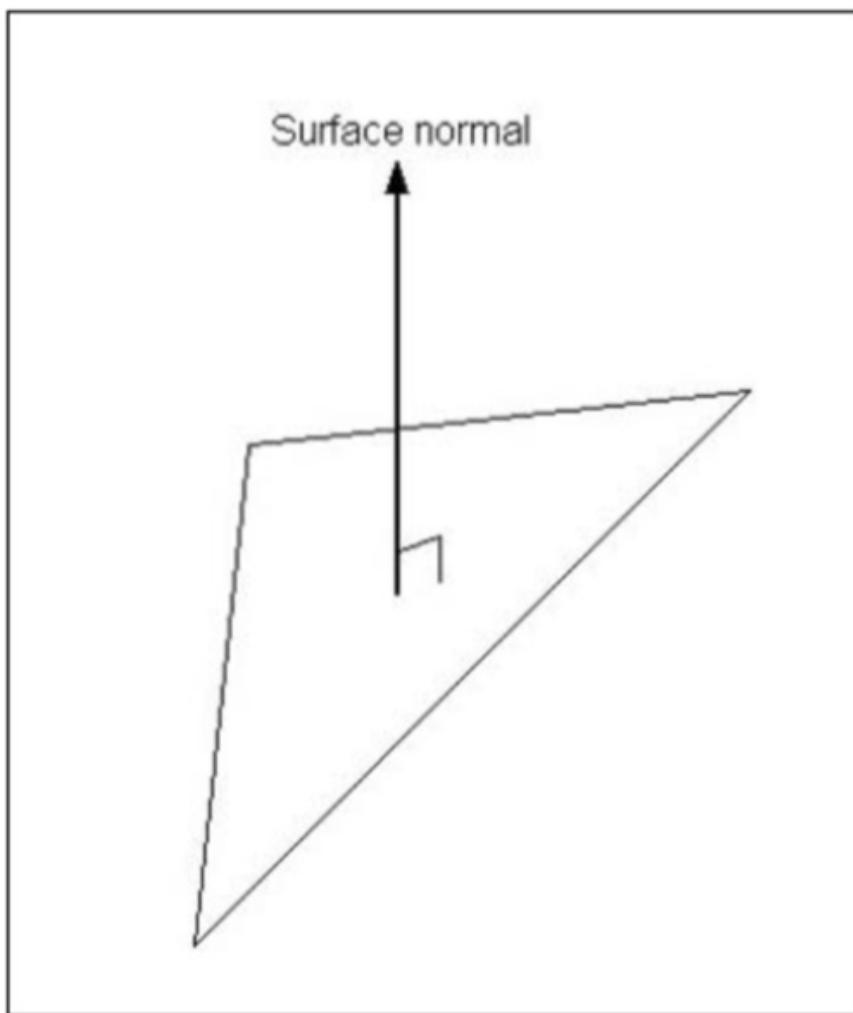
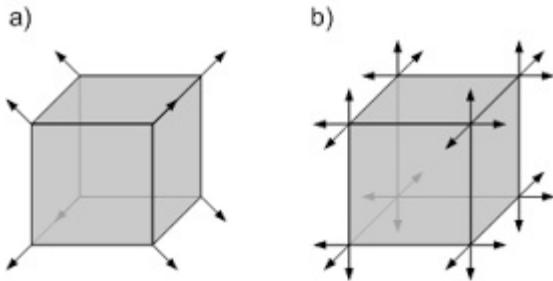


Figure 5.6 The surface normal.

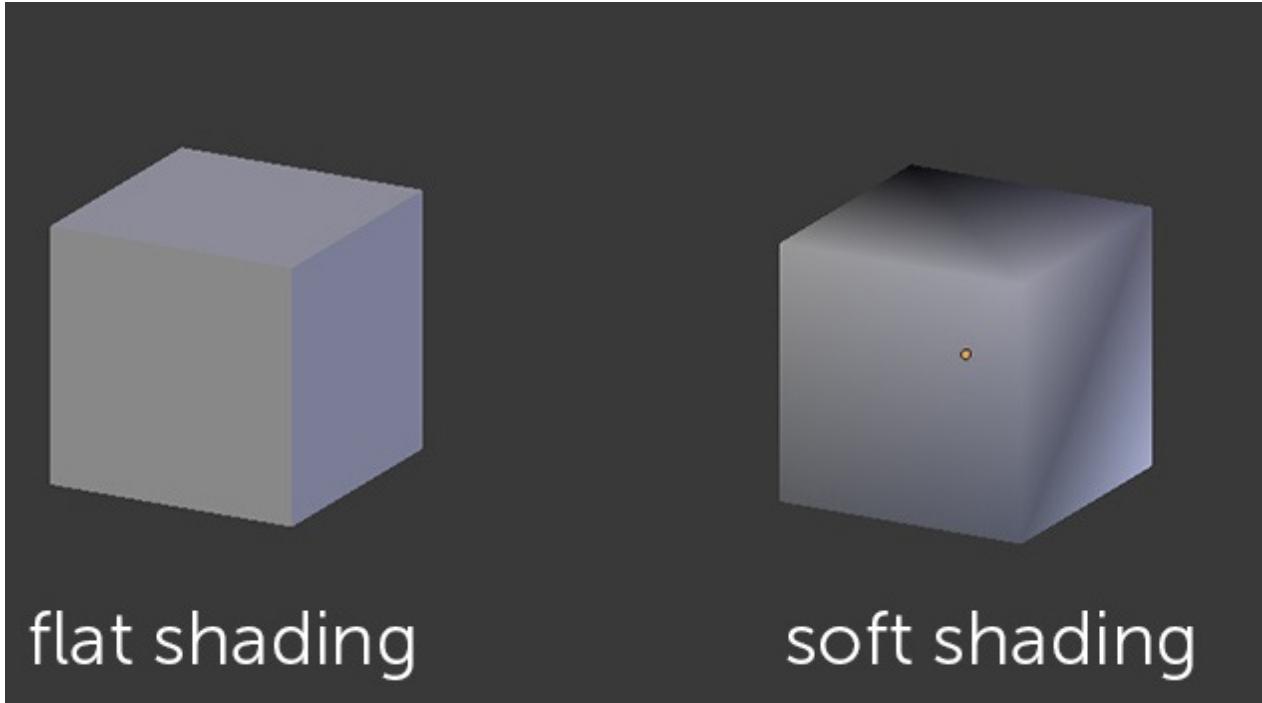
They are pivotal to lighting because normals are used to describe the orientation of a surface. When you specify a light you either specify it at a point in space, or at some direction. When you draw an object, light rays from the light source will strike the surface at some angle. The color of a surface is calculated using the angle that the ray hit the surface with and the normal of the surface.

The actual math behind how the normal is used to calculate a surface color is pretty involved, we will cover it, but much later.

In OpenGL normals are defined per vertex. This allows us a great deal of flexibility in making an object look smooth or sharp. Take the following cube for example, there are two ways to possibly define its normals:



Method A) is soft shaded, method B) is flat shaded. This is what they would look like lit:



Notice, soft shading does not really have defined edges. Flat shading does. You generally want to use flat shading for things like houses, and soft shading for organic things like people.

If you want to use normals in lighting, every time a vertex is specified, a normal must also be given. You can specify a normal with the function:

```
void GL.Normal3(float, float, float);
```

The function works the same as `GL.Vertex3`. The values you pass to it represent a 3 dimensional vector. This vector **MUST BE NORMALIZED**, or your lighting results will be wrong. This is how you could specify a triangle:

```
GL.Begin(PrimitiveType.Triangles);
    GL.Normal3(0f, 1f, 0f);
    GL.Vertex3(-3f, 0f, 2f);

    GL.Normal3(0f, 1f, 0f);
    GL.Vertex3(2f, 0f, 0f);

    GL.Normal3(0f, 1f, 0f);
    GL.Vertex3(-1f, 0f, -3f);
GL.End();
```

Notice how every vertex has a normal. Because of the way the OpenGL state machine works, you could simply specify one normal, and all subsequent vertices would use it. That is, this code is valid:

```
GL.Begin(PrimitiveType.Triangles);
    GL.Normal3(0f, 1f, 0f);
    GL.Vertex3(-3f, 0f, 2f);
    GL.Vertex3(2f, 0f, 0f);
    GL.Vertex3(-1f, 0f, -3f);
GL.End();
```

But that becomes hard to manage later on. I encourage you to specify a normal for every vertex, even if it's a duplicate. You can take a look at the static Primitives class we use to draw primitives. It draws all primitives **WITH** normals.

Calculating normals

Finding the normal of a flat surface is easy, given a little vector math, in particular the cross product. You might remember, given two 3D vectors A and B, the cross product will produce a vector that is perpendicular to both A and B.

Check your math code for implementation details.

This means, that you need two vectors, A and B to calculate the normal of a surface. Where can you find these vectors? All triangles are made up of 3 points: P1, P2 and P3. You can use them to find the vectors. Here is a visual example:

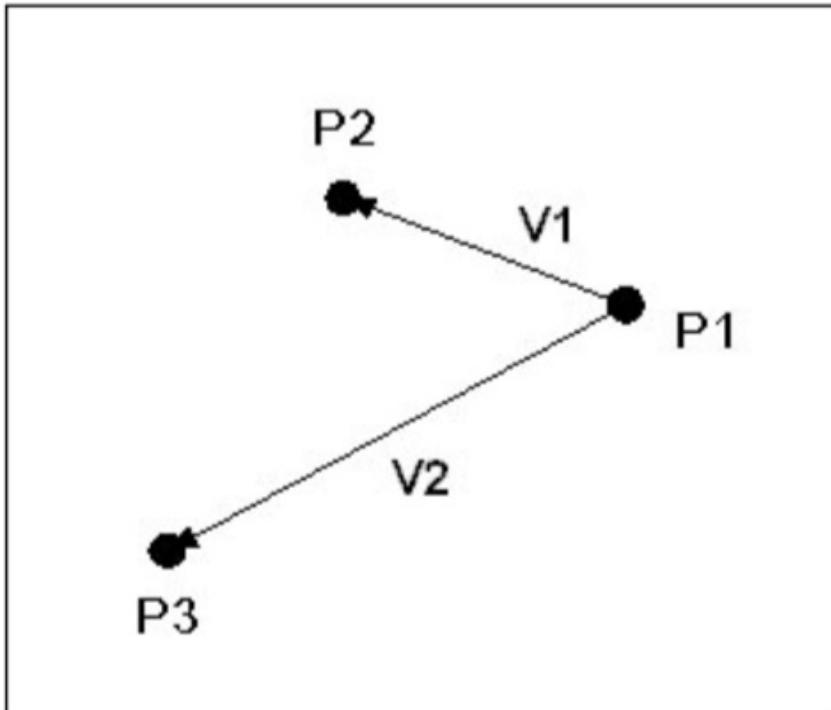


Figure 5.7 You can define two vectors, $V1$ and $V2$, out of three points.

So, the full code to get the normal of a triangle would look like this:

```
Vector3 GetNormal(Triangle tri) {
    Vector3 v1 = tri.p2 - tri.p1;
    Vector3 v2 = tri.p3 - tri.p1;

    Vector3 cross = Vector3.CrossProduct(v1, v2);
    Vector2 norm = Vector3.Normalize(cross);

    return norm;
}
```

Remember, the cross product just returns a perpendicular vector. You still have to normalize it!

OpenGL Normalization

You can ask OpenGL to do the normalization for you, so you can pass non-normal vectors into the `GL.Normal3` function. To do so, you just have to enable Normalization:

```
GL.Enable(EnableCaps.Normalize);
```

However, I strongly discourage doing this. It is viewed as bad practice, asking OpenGL to normalize is actually a lot slower than just doing the normalization yourself.

The lighting model

In addition to individual lights and materials, and normals there are additional global components of the lighting model that determine the final color of rendered objects. These are:

- A global ambient term
- Whether the location of the viewer is local or infinite
 - This only effects the specular highlight
- Whether the lighting is one or two sided
- Whether the calculated specular color is stored separately from other color values
 - Meaning it's passed as its own color to the final rendering stage

You control these elements of the lighting model with the `GL.LightModel` function, which is defined as:

```
void GL.LightModel(LightModelParameter, float);
void GL.LightModel(LightModelParameter, float[]);
```

The first parameter specifies which property you are modifying, the second is the value you are setting it to. The argument will either be a single float, or a 4 component array. The values for the first argument are:

- **LightModelAmbient** Ambient intensity of the scene (RGBA)
 - The default value is (0.2, 0.2, 0.2, 1.0)
- **LightModelLocalViewer** Is the viewpoint local or infinite.
 - The default value is 0 (INFINITE).
 - This takes either 1 (LOCAL) or 0 (INFINITE) as arguments
- **LightModelTwoSide** One or two sided lighting.
 - Default is 0 (ONE-SIDED)
 - This can be either 0 (ONE-SIDED) or 1 (TWO-SIDED)
- **LightModelColorControl** Is the specular color stored separate or together with ambient and diffuse
 - Default value is 0 (Single color, not separate)
 - 0 is single, 1 is separate

We will discuss each of these properties below, however they are not important enough to each get their own coding section. So pay attention, this is the only page they will be mentioned on.

Of course if you want to play around with them and make some practice scenes on your own, you can certainly do so.

Global ambient light

In addition to the ambient light contributed by individual light sources, there is a **global ambient light** that is present whether or not any light sources are enabled. This is used to model lights for which the source can not be determined.

This value is controlled with `LightModelAmbient`. The following code sets the global ambient light to a blue-green color. Kind of like what you might expect to see under water.

```
float[] ambient = new float[] { 0f, 0.2f, 0.3f, 1f };
```

```
GL.LightModel(LightModelParamater.LightModelAmbient, ambient);
```

Note: I don't really like a magical light source that i can't see, more often than not i will straight up set this to black to avoid it from contributing to my scene.

Local or Infinite View

When calculating the specular term, the direction from the vertex being calculated to the viewpoint effects the intensity of the specular highlight. The `LightModelLocalViewer` parameter lets you specify whether a viewpoint is local (based on the viewer's actual world position) or an infinite distance away.

Having a local viewpoint will look more realistic, but will be significantly more expensive calculation wise, as OpenGL will now need to calculate the direction for each vertex. An infinite view is used by default, it looks good enough 90% of the time.

If the default isn't realistic enough for your application, you can change it any time with:

```
// This takes an int (or float),  
// 1 is ON  
// 0 is OFF  
GL.LightModel(LightModelParamater.LightModelLocalViewer, 1);
```

Two Sided or One Sided Lighting

The next parameter you can specify is `LightModelTwoSide`. This parameter deals with whether you want to calculate the lighting for the back of polygons correctly. For example, if you were to take an enclosed object, like a cube and cut it in half the inside would not be lit correctly. To light the inside correctly, you must enable two sided lighting like so:

```
// 0 or 1  
GL.LightModel(LightModelParamater.LightModelTwoSide, 1);
```

Because most of the time you will not see inside closed meshes, and you will not see the back side of visible meshes, keeping this off is a good idea. It's off by default to save on performance.

Separate Specular Color

The final light model property you can set is the `LightModelColorControl` property. This control was added because when you do lighting with texturing the specular highlight tends to get washed out once the texture is applied.

When you enable this property, OpenGL stores two colors per vertex. One that is a combination of the ambient, diffuse, and emissive light combined with the texture of the object. And one that is its specular value. At render time it applies the specular value, this keeps the specular highlight from getting washed out by the texture colors.

Enabling this property is pretty straight forward:

```
// 0 or 1  
GL.LightModel(LightModelParamater.LightModelColorControl, 1);
```

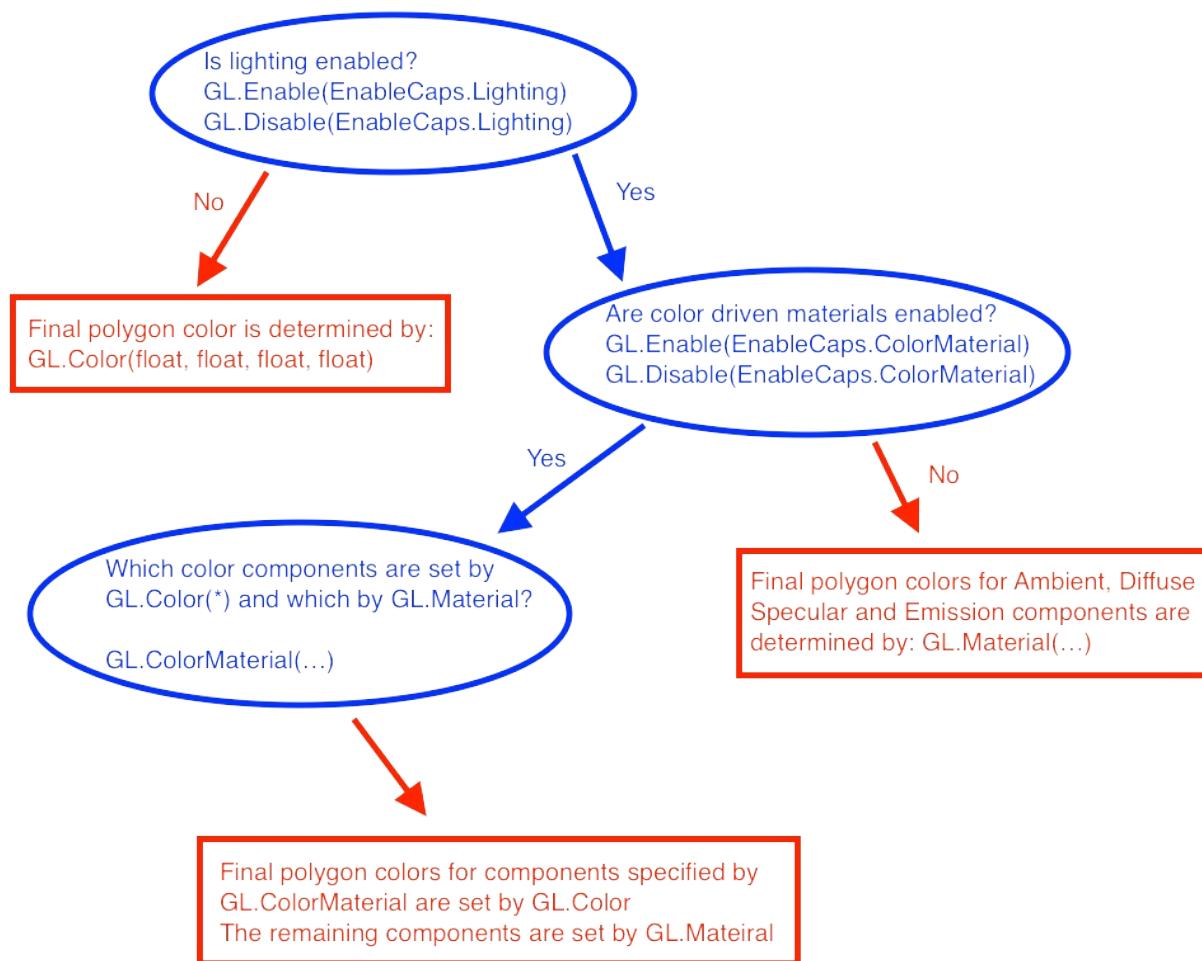
Because i've never rendered anything where the specular highlight **REALLY** mattered, i've never had a need to turn this component on

Where render color comes from

This page is a straight up rip of [Steve Baker's OpenGL Lighting](#) tutorial. Some of this stuff we've already covered and will serve as a review, other information might be brand new. Think of this as a one page cheat-sheet for lighting.

Introduction

Many people starting out with OpenGL are confused by the way that OpenGL's built-in lighting works - and consequently how color functions. I hope to be able to clear up some of the confusion. What is needed to explain this clearly is a flow chart:



Lighting ENABLED or DISABLED?

The first - and most basic - decision is whether to enable lighting or not.

```
GL.Enable(EnableCaps.Lightning);
```

...or...

```
GL.Disable (EnableCaps.Lighting);
```

If it's disabled then all polygons, lines and points will be colored according to the setting of the various forms of the `GL.color` command. Those colors will be carried forward without any change other than is imparted by texture or fog if those are also enabled. Hence:

```
GL.Color3(1.0f, 0.0f, 0.0f) ;
```

...gets you a pure red triangle no matter how it is positioned relative to the light source(s).

With `EnableCaps.Lighting` enabled, we need to specify more about the surface than just its color - we also need to know how shiny it is, whether it glows in the dark and whether it scatters light uniformly or in a more directional manner.

The idea is that OpenGL switches over to using the current settings of the current "*material*" instead of the simplistic idea of a polygon "color". This is an over-simplistic explanation - but keep it firmly in mind.

glMaterial and glLight

The OpenGL light model presumes that the light that reaches your eye from the polygon surface arrives by four different mechanisms:

- **AMBIENT** - light that comes from all directions equally and is scattered in all directions equally by the polygons in your scene. This isn't quite true of the real world - but it's a good first approximation for light that comes pretty much uniformly from the sky and arrives onto a surface by bouncing off so many other surfaces that it might as well be uniform.
- **DIFFUSE** - light that comes from a particular point source (like the Sun) and hits surfaces with an intensity that depends on whether they face towards the light or away from it. However, once the light radiates from the surface, it does so equally in all directions. It is diffuse lighting that best defines the shape of 3D objects.
- **SPECULAR** - as with diffuse lighting, the light comes from a point source, but with specular lighting, it is reflected more in the manner of a mirror where most of the light bounces off in a particular direction defined by the surface shape. Specular lighting is what produces the shiny highlights and helps us to distinguish between flat, dull surfaces such as plaster and shiny surfaces like polished plastics and metals.
- **EMISSION** - in this case, the light is actually emitted by the polygon - equally in all directions.

So, there are three light colors for each light - Ambient, Diffuse and Specular (set with `GL.Light`) and four for each surface (set with `GL.Material`). All OpenGL implementations support at least eight light sources - and the `GL.Material` can be changed at will for each polygon.

The final polygon color is the sum of all four light components, each of which is formed by multiplying the `GL.Mateiral` color by the `GL.Light` color (modified by the directionality in the case of Diffuse and Specular). Since there is no Emission color for the `GL.Light`, that is added to the final color without modification.

A good set of settings for a light source would be to set the Diffuse and Specular components to the color of the light source, and the Ambient to the same color - but at MUCH reduced intensity, 10% to 40% seems reasonable in most cases. So, a blue light might be configured like so:

```
float[] blue = new float[] { 0f, 0f, 1f, 1f };
float[] dimBlue = new float[] { 0f, 0f, 0.1f, 1f};
```

```
GL.Light(LightName.Light0, LightParameter.Ambient, dimBlue );
GL.Light(LightName.Light0, LightParameter.Diffuse, blue );
GL.Light(LightName.Light0, LightParameter.Specular, blue );
```

For the `GL.Material`, it's usual to set the Ambient and Diffuse colors to the natural color of the object and to put the Specular color to white. The emission color is generally black for objects that do not shine by their own light. So, a yellow object might have the following material:

```
float[] red = { 1f, 0f, 0f, 1f };
float[] white = { 1f, 1f, 1f, 1f }
float[] black = { 0f, 0f, 0f, 1f }

GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Ambient, red);
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Diffuse, red);
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, white);
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Emission, black);
```

Before you can use an OpenGL light source, it must be positioned using the `GL.Light` command and enabled using `GL.Enable(EnableCaps.LightN)` where 'N' is 0 through 7. There are additional commands to make light sources directional (like a spotlight or a flashlight) and to have it attenuate as a function of range from the light source.

glColorMaterial

This is without doubt the most confusing thing about OpenGL lighting - and the biggest cause of problems for beginners. The problem with using `GL.Material` to change polygon colors is two-fold:

- You frequently need to change `GL.Mateiral` properties for both Ambient and Diffuse to identical values - this takes two OpenGL function calls which is annoying.
- You cannot change `GL.Mateiral` settings with many of the more advanced polygon rendering techniques such as Vertex arrays and `GL.DrawElements`.

For these reasons, OpenGL has a feature that allows you do drive the `GL.Material` colors using the more flexible `GL.color` command (which is not otherwise useful when lighting is enabled).

To drive (say) the Emission component of the `GL.Material` using `GL.Color`, you must say:

```
GL.Enable(EnableCap.ColorMaterial);
GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.Emission);
```

From this point performing a `GL.Color` command has the exact same effect as calling:

```
GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Emission, ...colors...);
```

One especially useful option is:

```
GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.AmbientAndDiffuse);
```

This causes `GL.Color` commands to change both Ambient and Diffuse colors at the same time. That's a very common thing to want to do for real-world lighting models.

Light Sources.

OpenGL's lights are turned on and off with `GL.Enable(EnableCaps.LightN)` and `GL.Disable(EnableCaps.LightN)` where 'N' is a number in the range zero to the maximum number of lights that this implementation supports (typically eight).

The `GL.Light` call allows you to specify the color (ambient, diffuse and specular), position, direction, beam width and attenuation rate for each light.

By default, it is assumed that both the light and the viewer are effectively infinitely far from the object being lit. You can change that with the `GL.LightModel` call - but doing so is likely to slow down your program - so don't do it unless you have to. `GL.LightModel` also allows you to set a global ambient lighting level that's independent of the other OpenGL light sources.

There is also an option to light the front and back faces of your polygons differently. That is also likely to slow your program down - so don't do it.

glNormal

When lighting is enabled, OpenGL suddenly needs to know the orientation of the surface at each polygon vertex. You need to call `GL.Normal` for each vertex to define that - OpenGL does not provide a useful default.

Good Settings.

With this huge range of options, it can be hard to pick sensible default values for these things. Even harder is to debug broken lighting! Sometimes you start a scene and everything is black. The best way to debug this is to set some sensible defaults, and work from there.

My advice for a starting point is to:

- Set `EnableCaps.Light0`'s position to something like 45 degrees to the "vertical". Coordinate (1,1,0) should work nicely in most cases.
- Set `EnableCaps.Light0`'s Ambient color to 0,0,0,1
- Set `EnableCaps.Light0`'s Diffuse color to 1,1,1,1
- Set `EnableCaps.Light0`'s Specular color to 1,1,1,1
- Set the `GL.LightModel`'s global ambient to 0.2,0.2,0.2,1 (this is the default).
- Don't set any other `GL.Light` or `GL.LightModel` options - just let them default.
- Enable `Enable.Lighting` and `EnableCaps.Light0`.
- Enable `EnableCaps.ColorMaterial` and set `colorMaterialParameter.ColorMaterial` to `ColorMaterialParameter.AmbientAndDiffuse`. This means that `GL.Material` will control the polygon's specular and emission colours and the ambient and diffuse will both be set using `GL.Color`.
- Set the `GL.Material`'s Specular color to 1,1,1,1
- Set the `GL.Material`'s Emission color to 0,0,0,1
- Set the `GL.Material` to whatever color you want each polygon to basically appear to be. That sets the Ambient and Diffuse to the same value which is what you generally want.

Using Alpha with lighting enabled.

One confusing thing is that each of the color components (Ambient,Diffuse, Specular and Emission) has an associated "*alpha*" component for setting transparency.

It is important to know that only the DIFFUSE color's alpha value actually determines the transparency of the polygon. If you have taken my advice and used `GL.colorMaterial` to cause `GL.color` to drive the ambient and diffuse components then this seems perfectly natural.

But people sometimes want to use the `GL.color` to drive one of the other material components and are then very confused about their inability to set the transparency using `GL.Color`. If that happens to you - remember to use `GL.Material` to set the diffuse color - and hence the alpha for the polygon overall.

Blending and Fog

Up to this point we've used RGBA for a lot of things. Every material color has an alpha component. Even lights have an alpha component! Yet, setting this to 0.5 does not add alpha to our scene at all.

OpenGL let's you actually use the Alpha component to draw translucent and transparent primitives through a process known as blending.

In addition to blending, OpenGL also offers fog. Fog is great to fade objects in the distance out. If something is at the end of your view frustum, it's better to fade it out and make your world look like it's in a fog than it is to have a hard cut-off point that shows players that they are viewing the world through a limited window.

Perhaps the most famous use of fog is the original silent hill. The PS1 could not draw many triangles (weak processor). To avoid drawing too many triangles and losing framerate, the developers had to set the camera frustums far plane pretty close to a distance of about 25 (compared to the 1000 we set ours to).

This short frustum caused objects farther than 25 game units to not render, making things like buildings pop in and out of view. To compensate for this, the developers added a heavy fog to the game to limit the players visibility. Really, it was just used to hide the fact that anything further than 25 units would pop out of existence.

They used this heavy fog to add an air of mystery to the game, and accidentally gave rise to modern horror games.



Blending

OpenGL allows you to blend incoming fragments with pixels already on screen. Which enables you to introduce effects such as transparency into your scenes. With transparency you can simulate water, windows, glass and other objects in the world you can normally see through.

The term **fragment** might be new to you, but it will come up several times from here on out. This is a good time to discuss what it is.

As OpenGL processes the primitives you pass to it, during the rasterization stage it breaks them down into pixel size chunks called fragments. Sometimes the terms pixel and fragment are used interchangeably, but there is a subtle difference. A pixel is a value that gets written to the color buffer. A fragment is a piece of a primitive that *might* eventually become a pixel after it is depth-tested, alpha-tested, blended, combined with a texture, combined with another fragment or it may just become a pixel without being modified.

So, essentially a fragment is data that can become a pixel, but isn't necessarily a pixel yet.

Remember the alpha value we've been ignoring all this time? Well, now that we're talking about blending it's time to learn how to use it. When enabling blending, you are telling OpenGL to combine the color of the incoming primitive with the color that is already in the frame buffer, and store that combined color back into the frame buffer.

Blending operations are typically specified with the RGB values representing color, and the Alpha value representing opacity. But other combinations are possible. From now on, we will refer to the incoming fragment as SOURCE, and the pixel that is already in the frame buffer as DESTINATION.

To enable blending, call the `GL.Enable` method with `EnableCapsBlend` as its argument. Just enabling blending isn't enough to make blending happen. You must also tell OpenGL what formula to use to blend pixel colors. You do this by defining a blending function for the source and destination fragments.

You can call the function `GL.BlendFunc` to define the source and destination blend factors. Blend factors are values in the 0 to 1 range, that are multiplied by the RGBA components of both the source and destination colors. The resulting colors are then combined (usually by adding them) and clamped to the range of 0 to 1.

```
void GL.BlendFunc(BlendingFactorSrc sourceFactor, BlendingFactorDest destFactor);
```

The first argument is the source blend factor, the second argument is the destination blend factor. Both enums contain the same enumerated values, with the exception of `OneMinusSrcColor` which is only available as a source blend factor and `OneMinusDstColor`, which is only available as a destination blend factor. The enumerated values are:

- **Zero** Each component is multiplied by 0, effectively setting the color to black
- **One** Each component is multiplied by 1, leaving the color unchanged
- **SrcColor** Each component is multiplied by the corresponding component
- **OneMinusSrcColor** Each component is multiplied by 1 - source color
- **DstColor** Each component is multiplied by the corresponding component in the destination color
- **OneMinusDstColor** Each component is multiplied by 1 - destination color
- **SrcAlpha** Each component is multiplied by the source alpha value
- **OneMinusSrcAlpha** Each component is multiplied by 1 - source alpha value
- **DstAlpha** Each component is multiplied by the destination alpha value

- **OneMinusDstAlpha** Each component is multiplied by $1 - \text{dest alpha value}$
- **ConstantColor** Each component is multiplied by a constant color, set using `GL.BlendColor`
- **OneMinusConstantColor** Each component is multiplied by $1 - \text{constant color}$, set using `GL.BlendColor`
- **ConstantAlpha** Each component is multiplied by an alpha value set using `GL.BlendColor`
- **OneMinusConstantAlpha** Each component is multiplied by $1 - \text{a constant alpha value}$ set using `GL.BlendColor`
- **SrcAlphaSaturate** Multiplies the source color by the minimum of source and $(1 - \text{destination})$. The alpha value is not modified. Only valid as the source blend factor

The default values are One for the source and Zero for the destination. Which produces the same results as not using blending at all.

Many different effects can be created with these blending factors, some of which are more useful in medical imaging than games. To better understand how this voodoo works, let's look at the application that is most often used in games: transparency! Typically transparency is implemented as follows:

```
GL.Enable(EnableCapsBlend)
GL.BlnFunc(BlendingFactorSrc.SrcAlpha, BlendingFactorDest.OneMinusSrcAlpha);
```

Memorize that function! It's the one you will use 90% of the time!

To get an idea of how this works, Let's examine how it would work for a single pixel.

- Say we drew a red triangle onto screen
 - The frame buffer has a red pixel
 - DST: $(1, 0, 0, 1)$
- Next, we draw a blue triangle with 50% opacity
 - SRC: $(0, 0, 1, 0.5)$
- With the blend factors we've chose, the source color is multiplied by the source alpha
 - $(0, 0, 1) * 0.5 = (0, 0, 0.5)$
 - SRC: $(0, 0, 0.5, 0.5)$
- The destination will be multiplied by $1 - \text{source alpha}$ (0.5)
 - $(1, 0, 0) * (1 - 0.5) = (0.5, 0, 0)$
 - DST: $(0.5, 0, 0, 1)$
- Finally, the source and destination colors are added together.
 - SRC + DST = $(0, 0, 0.5) + (0.5, 0, 0)$
 - Result: $(0.5, 0, 0.5)$
- This added number is written to the frame buffer
 - $(0.5, 0, 0.5)$ written to frame buffer

This simple example ignores one very important thing: you have to pay attention to the depth of the objects and the order in which they are rendered when using transparency.

When we draw without transparency, the order of drawing commands doesn't matter, as the Z-Buffer will take care of hiding overlapping objects for us.

When you use transparency however, order most definitely matters! Lets assume you have two objects, a solid and a transparent one. If you draw a solid object first, then a transparent object, you will see the image get blended as expected. But if you draw the transparent object first, then the solid object, the transparent object will draw and blend with the clear color. Then parts of the solid object not overlapping the transparent object will draw. And your scene will look broken.

The most common way to handle this is to make two render passes at the scene. First, render only solid objects. Next, enable blending and render only transparent objects. This will fix the problem in 90% of the cases, but sometimes you have overlapping transparent objects!

The full fix is to do the two render passes as described above, but also sort the transparent objects so the furthest objects render first. As you will see later, even this isn't a perfect solution

Blend Practice

We're going to real quick try to render a scene which uses blending. This exercise will also re-inforce lessons we've learned in the materials section. The code i used to get this practice challenge done is provided in the next section "Solution". Try to do this code on your own before looking at my implementation.

First, set up a test scene. In this scene your camera will not be moving!

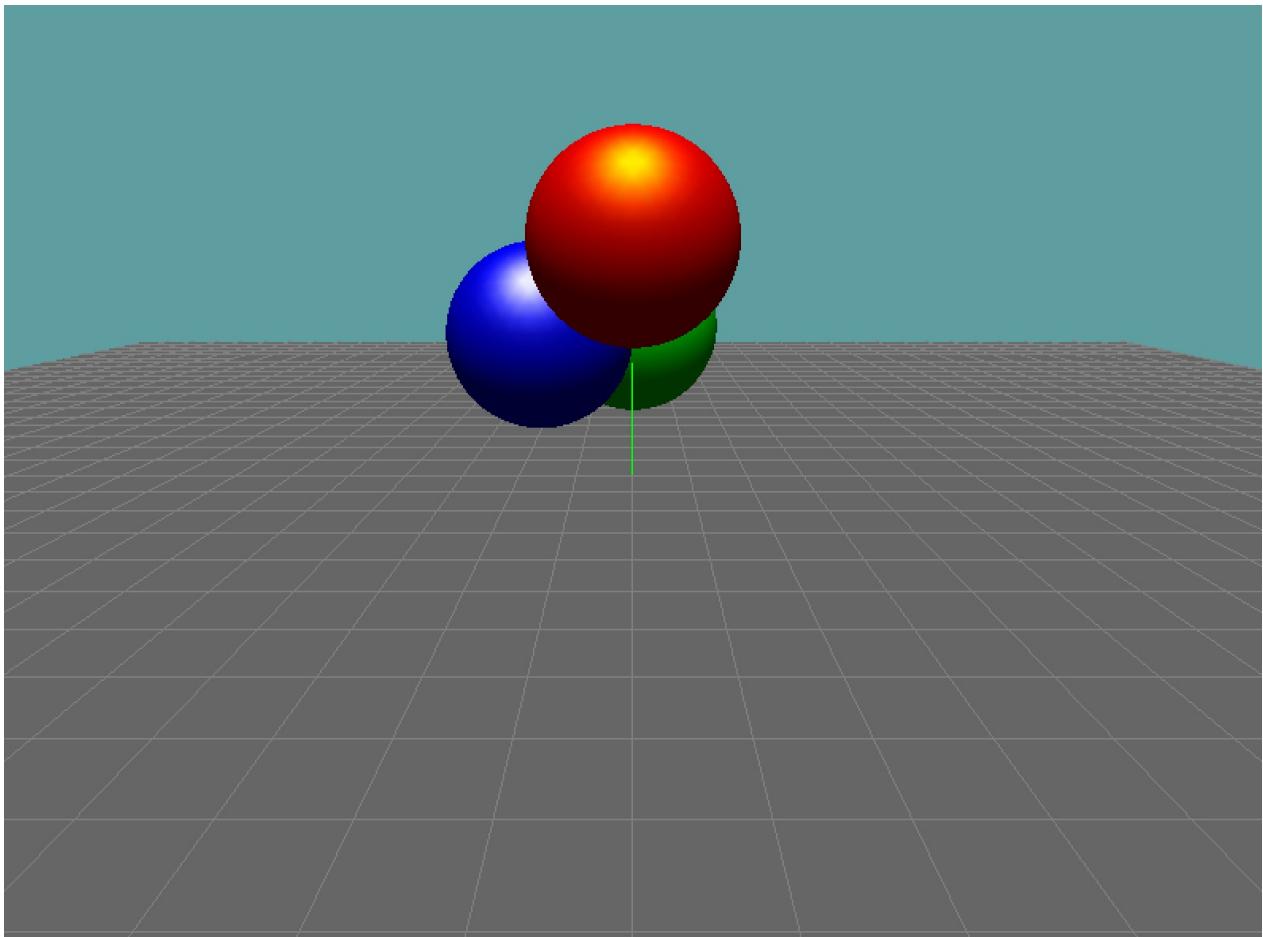
Initialize

- Enable lighting, we're going to use only one light
- Set the lights position to 0, 1, 1, it's a directional light
 - REMEMBER TO SET A W
- Set a uniform low ambient color (dark gray)
 - I suggest (.2, .2, .2, 1)
- Set a uniform high diffuse color (light gray)
 - I suggest (.8, .8, .8, 1)
- Set a specular color of white
- Set the specular property of the front and back of the material to white
- Set the material specular exponent to 20
- Enable color tracking for the ambient AND diffuse components
- Turn OFF the global ambient light by setting it to black
 - We never wrote code for this, it was discussed in "The Lighting Model"

Render

- Position your camera at (0, 2, -7)
- Have the camera looking at (0, 0, 0)
- Render an UNLIT solid grid
- Set the material ambient and diffuse color to green
 - Remember, color tracking (Material Color) is turned on!
- Draw a sphere at: (0, 1, 3)
- Set the material color to blue
- Draw a sphere at: (1, 1, 2)
- Set the material color to red
- Draw a sphere at (0, 2, 1)

At this point, your scene should look like this:



Now for some blending

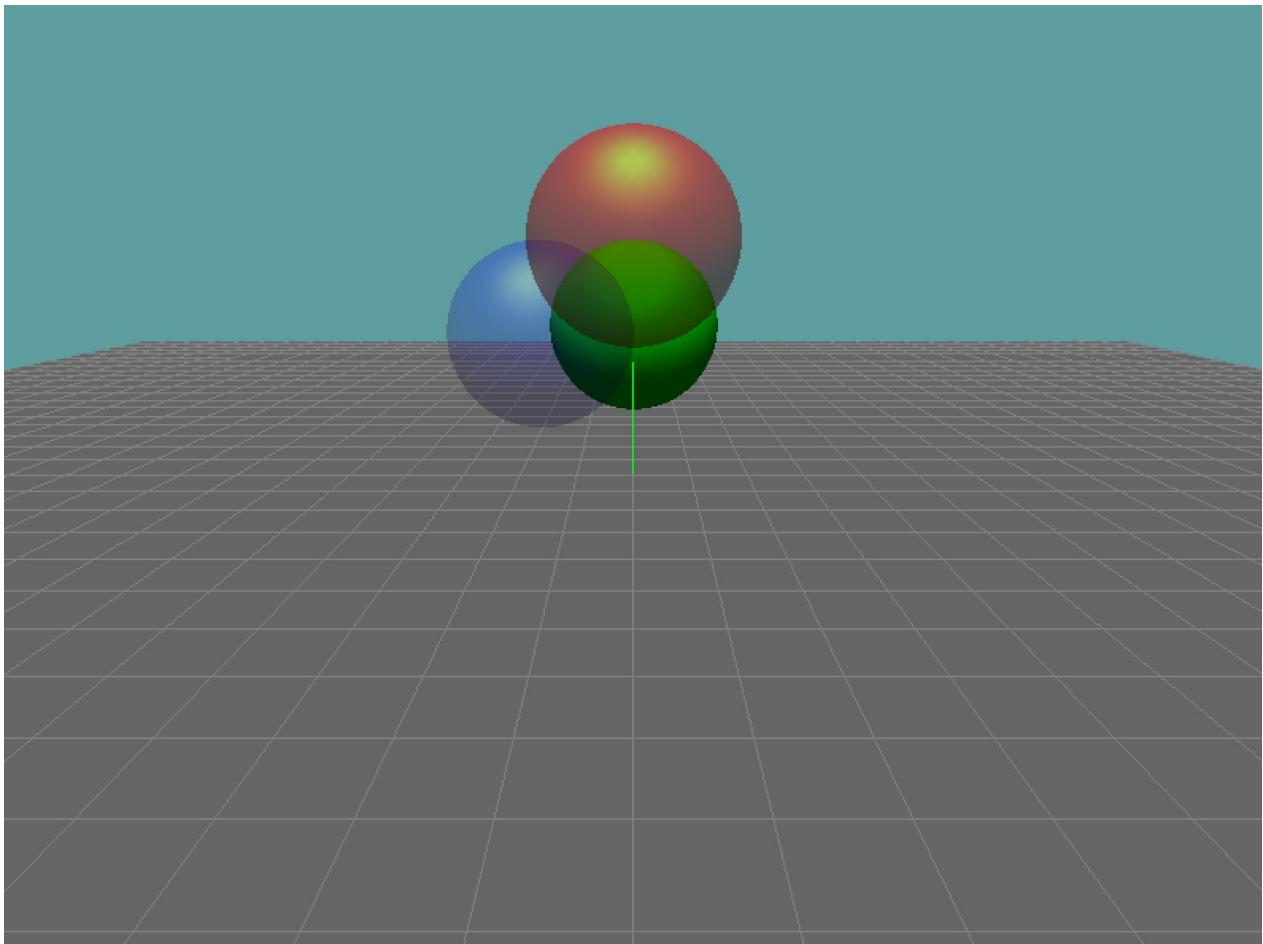
Inside Initialize:

- Enable blending
- Set the blend func to the transparency function we talked about in the last section

Inside Render:

- To indicate we're going to use alpha, change `GL.Color3` to `GL.Color4`
- Set the alpha component of the green sphere at 1
- Set the alpha component of the blue sphere to 0.25
- Set the alpha component of the red sphere to 0.5

Having made these changes, your scene should now look like this:

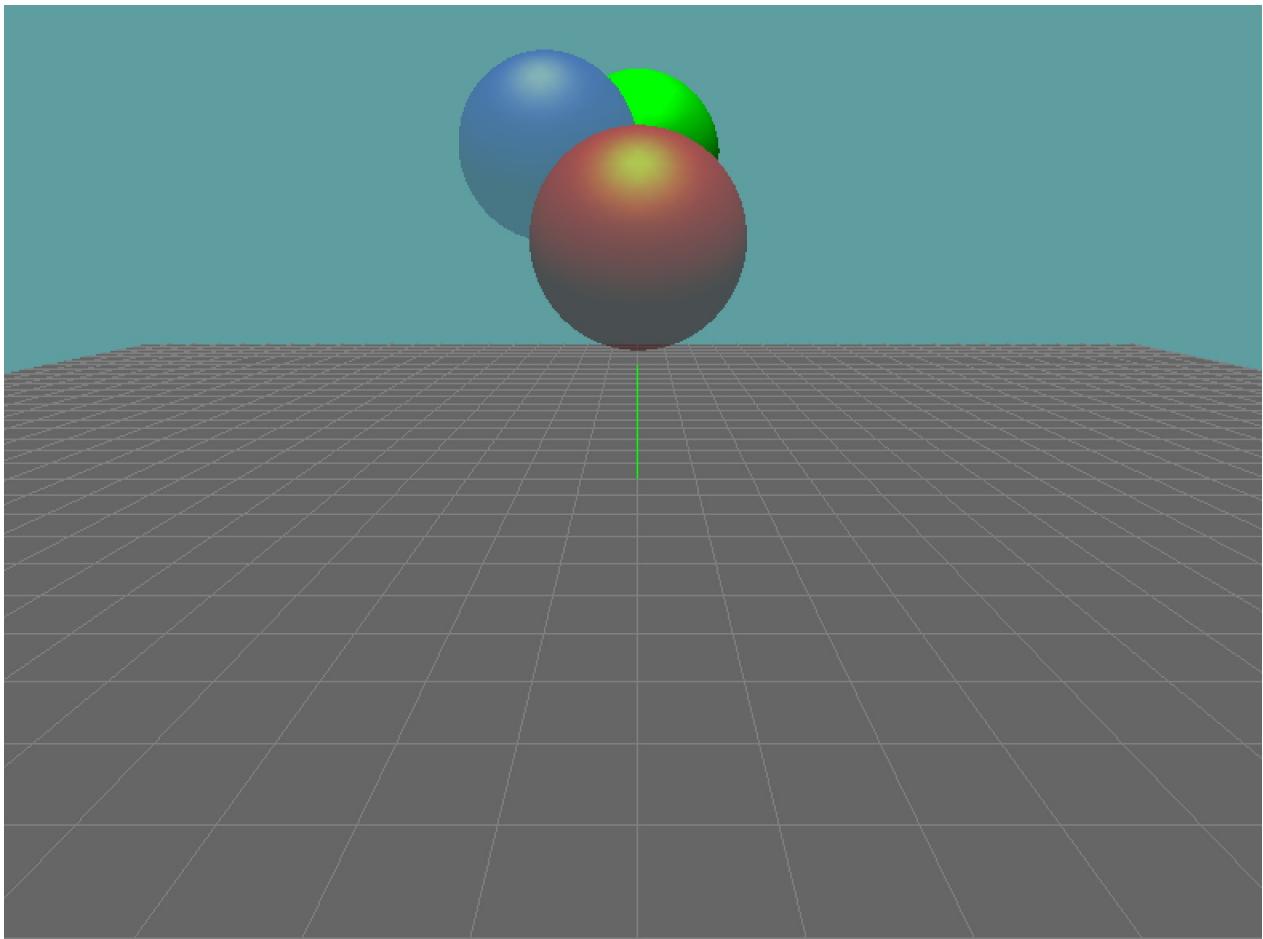


Thats it. That's all there is to transparency. If you look at the source of our 2D game framework, it enables blending and sets the transperancy function when the framework is initialized, and that's all it does. Alpha values come from textures.

Mess it up!

Want to see how alpha rendering can go wrong? Change the order up. Render the red sphere first, blue second and green last. Do this without modifying the Z values or color values, just change the order you render them in, to intentionally mess the scene up.

The rendered scene will look like this:



It messes up so bad because we draw the red sphere with nothing in the frame buffer except the clear color, so the red sphere blends on top of the clear color (except a small sliver on its bottom that blends with the plane).

Next we render the blue sphere, again in its visible parts we can only blend it with the clear color. You might expect the blue sphere to draw over the red one, but the objects are recorded in the Z buffer as if they were solid.

The Z buffer knows no alpha. As far as Z is concerned, the red sphere is closer than the blue sphere, so the fragments of the blue sphere that are behind the red one are simply discarded.

The green sphere is only partially visible for the same reason, most of its fragments are discarded during the z-test.

During the initial setup function

This is the code we used to get the demo scene set up

Initialize

```
public override void Initialize() {
    base.Initialize();
    GL.Enable(EnableCap.Lighting);
    GL.Enable(EnableCap.Light0);

    GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0f, 1f, 1f, 0f });
    GL.Light(LightName.Light0, LightParameter.Ambient, new float[] { .2f, .2f, .2f, 1f });
    GL.Light(LightName.Light0, LightParameter.Diffuse, new float[] { .8f, .8f, .8f, 1f });
    GL.Light(LightName.Light0, LightParameter.Specular, new float[] { 1f, 1f, 1f, 1f });

    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1f, 1f, 1f, 1f });
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 20f);

    GL.Enable(EnableCap.ColorMaterial);
    GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.AmbientAndDiffuse);

    GL.LightModel(LightModelParameter.LightModelAmbient, new float[] { 0f, 0f, 0f, 1f });
}
```

Render

```
public override void Render() {
    Matrix4 lookat = Matrix4.LookAt(new Vector3(0f, 2f, -7f), new Vector3(0f, 0f, 0f), new Vector3(0f, 1f, 0f));
    GL.LoadMatrix(Matrix4.Transpose(lookat).Matrix);

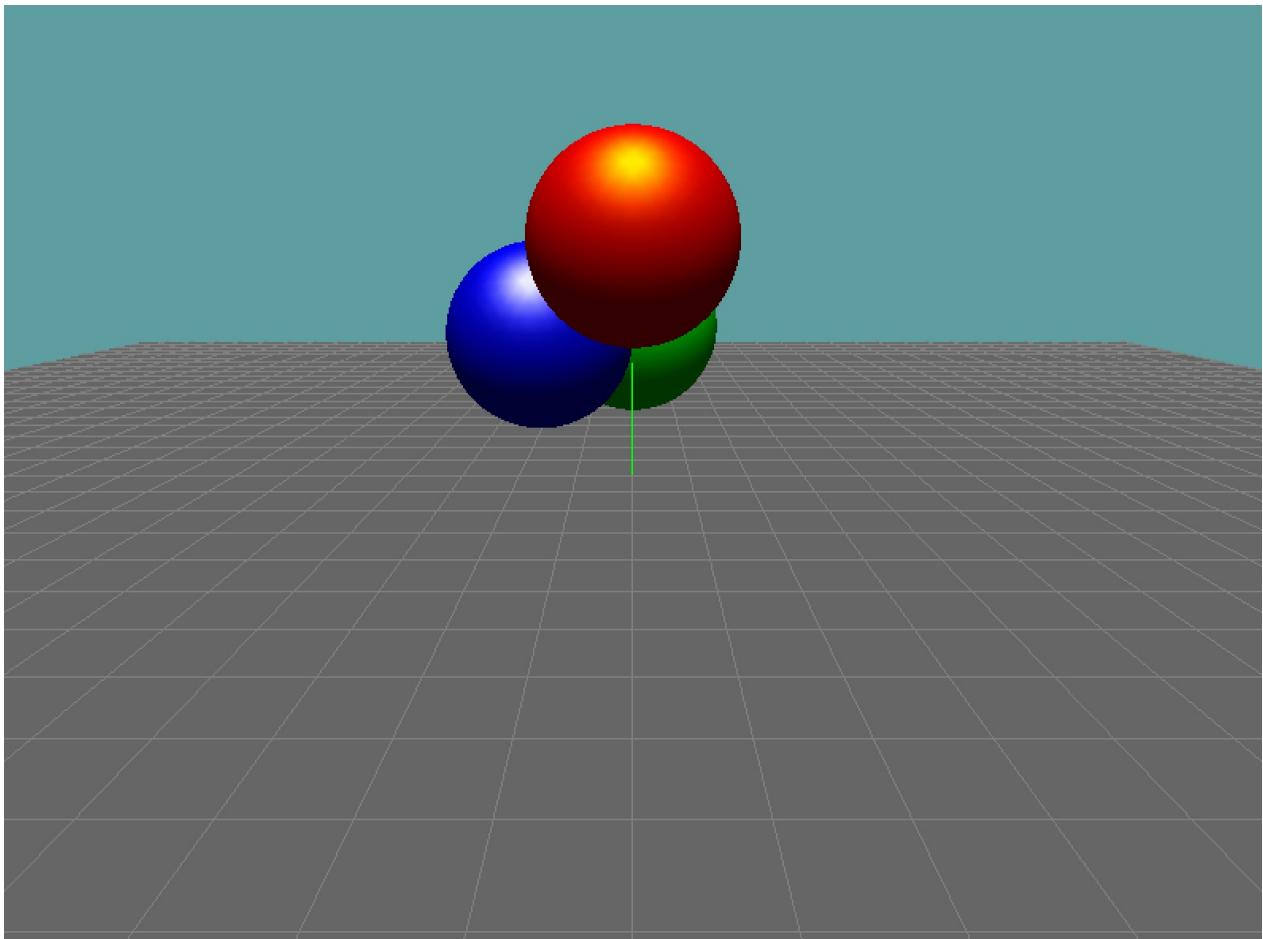
    GL.Disable(EnableCap.Lighting);
    grid.Render();
    GL.Enable(EnableCap.Lighting);

    GL.Color3(0f, 1f, 0f);
    GL.PushMatrix();
        GL.Translate(0f, 1f, 3f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();

    GL.Color3(0f, 0f, 1f);
    GL.PushMatrix();
        GL.Translate(1f, 1f, 2f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();

    GL.Color3(1f, 0f, 0f)
    GL.PushMatrix();
        GL.Translate(0f, 2f, 1f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();
}
```

At this point, your scene should look like this:



Now that the initial scene has been set up, let's take a look at the code changes we had to make in order to have transparency work

Initialize

```
public override void Initialize() {
    base.Initialize();
    GL.Enable(EnableCap.Lighting);
    GL.Enable(EnableCap.Light0);

    GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0f, 1f, 1f, 0f });
    GL.Light(LightName.Light0, LightParameter.Ambient, new float[] { .2f, .2f, .2f, 1f });
    GL.Light(LightName.Light0, LightParameter.Diffuse, new float[] { .8f, .8f, .8f, 1f });
    GL.Light(LightName.Light0, LightParameter.Specular, new float[] { 1f, 1f, 1f, 1f });

    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Specular, new float[] { 1f, 1f, 1f, 1f });
    GL.Material(MaterialFace.FrontAndBack, MaterialParameter.Shininess, 20f);

    GL.Enable(EnableCap.ColorMaterial);
    GL.ColorMaterial(MaterialFace.FrontAndBack, ColorMaterialParameter.AmbientAndDiffuse);

    GL.LightModel(LightModelParameter.LightModelAmbient, new float[] { 0f, 0f, 0f, 1f });

    // NEW
    GL.Enable(EnableCap.Blend);
    GL.BlendFunc(BlendingFactorSrc.SrcAlpha, BlendingFactorDest.OneMinusSrcAlpha);
}
```

Render

```

public override void Render() {
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(0f, 2f, -7f), new Vector3(0f, 0f, 0f), new Vector3(0f, 1f, 0f));
    GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

    GL.Disable(EnableCap.Lighting);
    grid.Render();
    GL.Enable(EnableCap.Lighting);

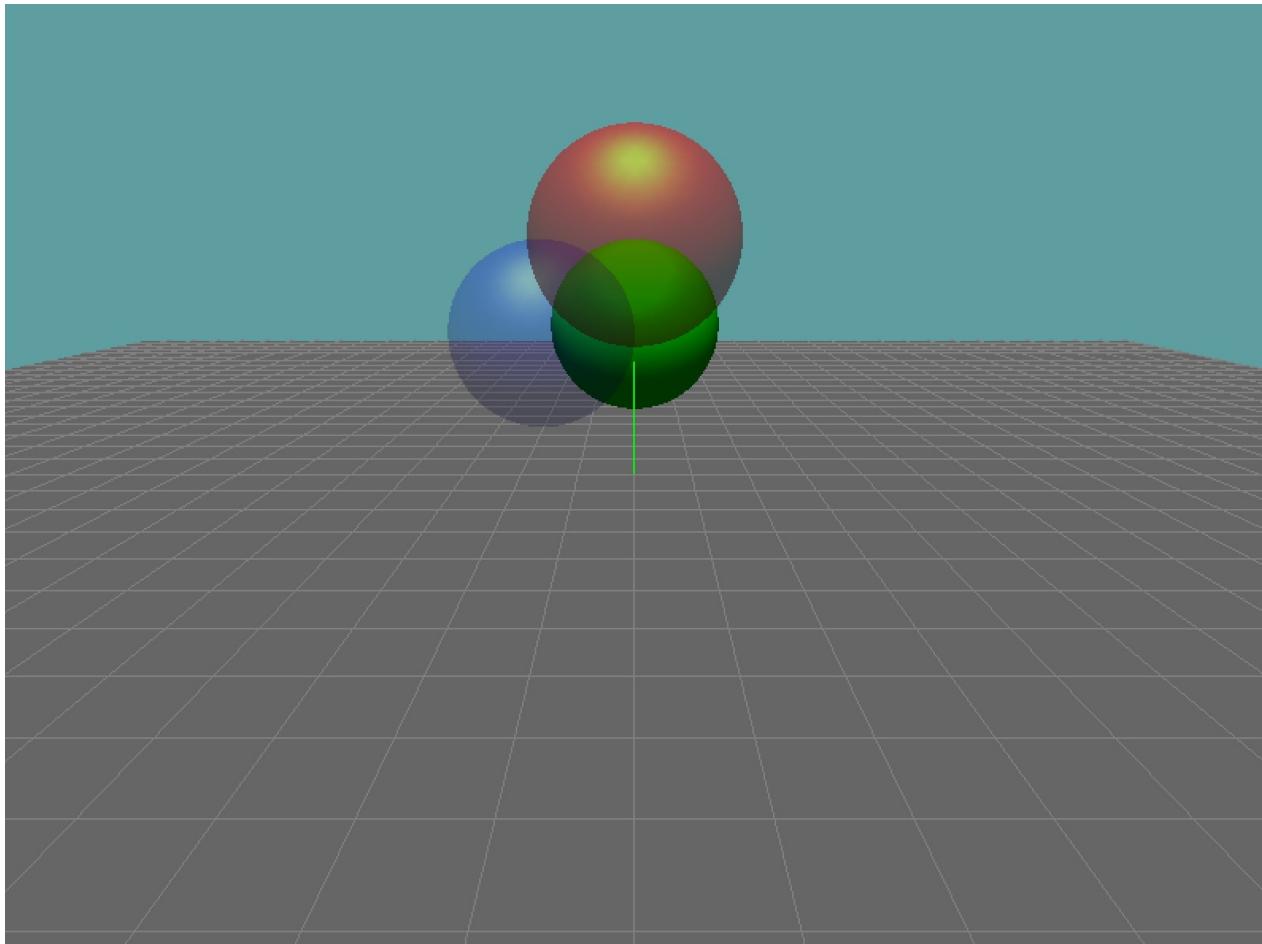
    GL.Color4(0f, 1f, 0f, 1f); // CHANGED
    GL.PushMatrix();
        GL.Translate(0f, 1f, 3f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();

    GL.Color4(0f, 0f, 1f, .25f); // CHANGED
    GL.PushMatrix();
        GL.Translate(1f, 1f, 2f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();

    GL.Color4(1f, 0f, 0f, .5f); // CHANGED
    GL.PushMatrix();
        GL.Translate(0f, 2f, 1f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();
}

```

Having made these changes, your scene should now look like this:



To mess things up, simply change the render order:

Render

```

public override void Render() {
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(0f, 2f, -7f), new Vector3(0f, 0f, 0f), new Vector3(0f, 1f, 0f));
    GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

    GL.Disable(EnableCap.Lighting);
    grid.Render();
    GL.Enable(EnableCap.Lighting);

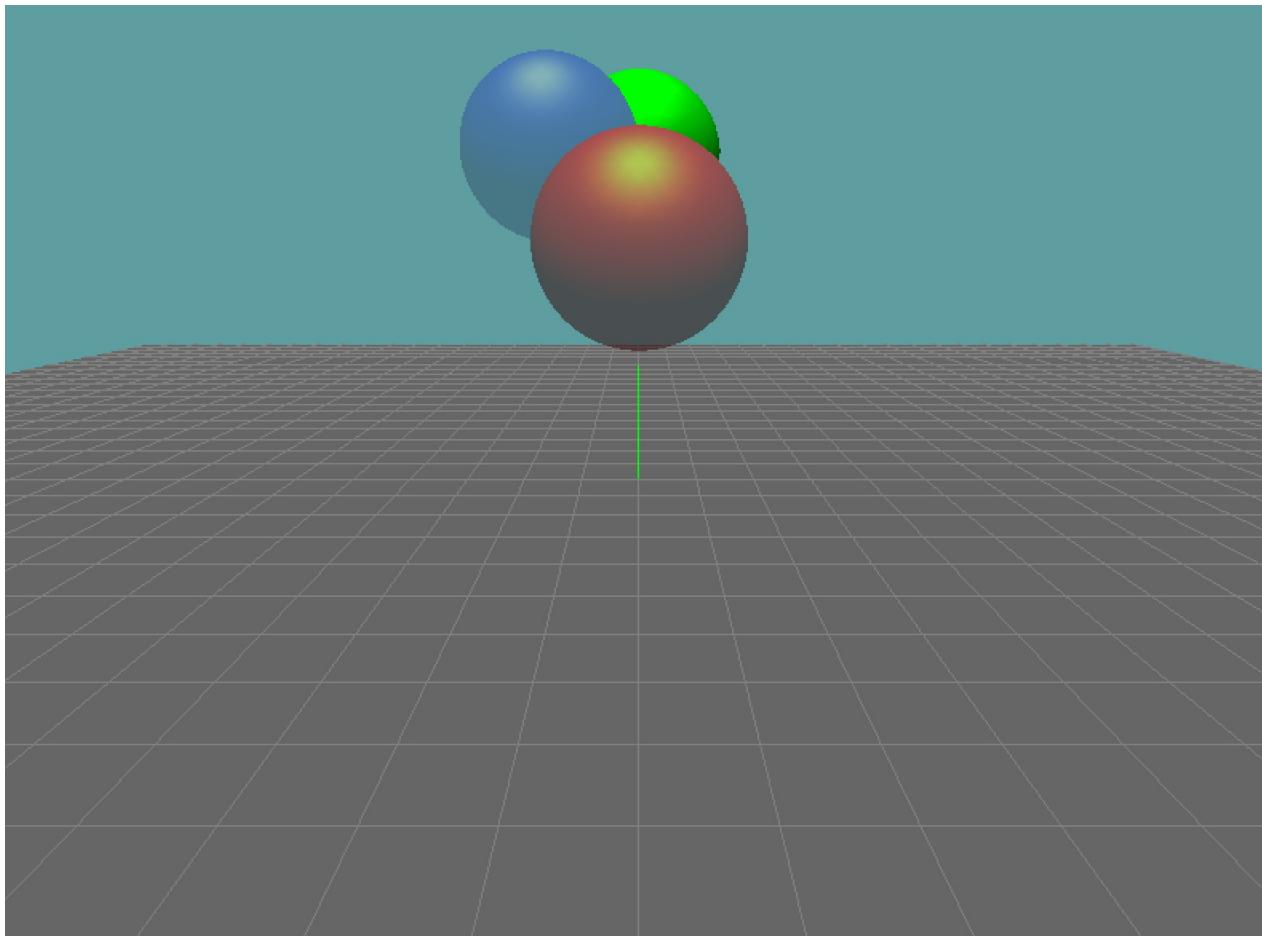
    // WAR LAST, NOW FIRST
    GL.Color4(1f, 0f, 0f, .5f);
    GL.PushMatrix();
        GL.Translate(0f, 2f, 1f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();

    // STAYED IN CENTER
    GL.Color4(0f, 0f, 1f, .25f);
    GL.PushMatrix();
        GL.Translate(1f, 1f, 2f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();

    // WAS FIRST, NOW LAST
    GL.Color4(0f, 1f, 0f, 1f);
    GL.PushMatrix();
        GL.Translate(0f, 1f, 3f);
        Primitives.DrawSphere(3);
    GL.PopMatrix();
}

```

The messed up scene looks like this:



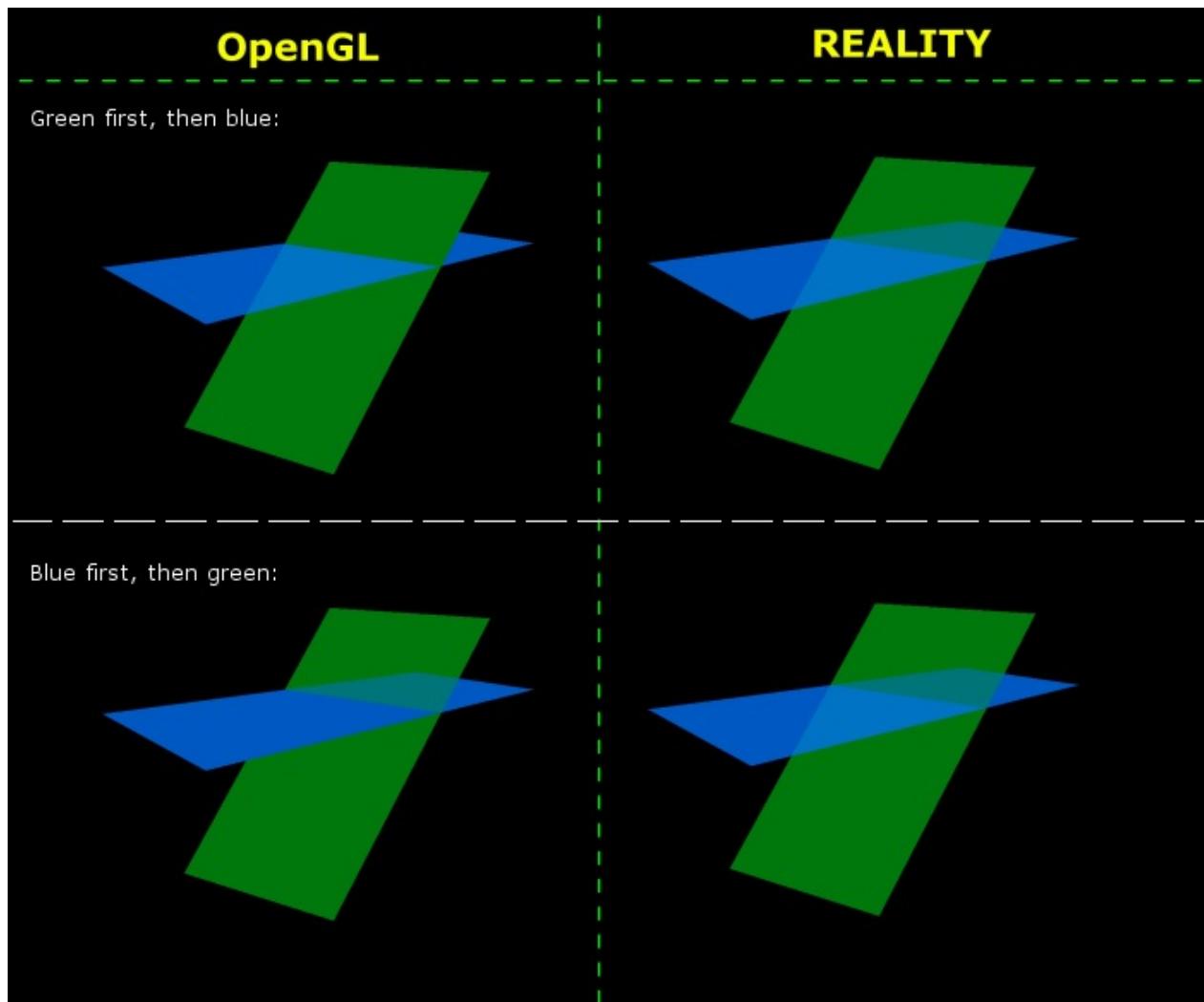
Problems with blending

Even if you order your polygons perfectly, and use the perfect blend functions there are still blending scenarios OpenGL will fail to handle. Some of these are unsolvable, so we avoid situations that may cause them in games.

Unsolvable is a lie. We can solve them, in OpenGL with a technique known as depth-peeling, but it's an advanced technique that is expensive and will slow the games framerate WAY down. So we just avoid these issues.

Intersecting alpha

What happens when two primitives with alpha intersect? Which one should be drawn first? Consider this image:



- Both blue and green planes have alpha
- In the top, the part of the blue plane that's behind the green isn't showing
- In the bottom, the part of the green that's behind the blue isn't showing

You can see how this is a very sticky situation. Don't ever have two transparent objects cross each other.

Polygon Cycle

When multiple polygons cross each other, we refer to that as cycling. You only need 3 triangles to create an unsortable cycle, consider the following:



If all 3 of those triangles have alpha, which one do you draw first?

Transparent frame buffer

The alpha value of the frame buffer should always be 1. If for some reason it's not, then the blending is going to be off. Luckily, so long as you use the standard transparency function:

```
GL.BlnndFunc(BlendingFactorSrc.SrcAlpha, BlendingFactorDest.OneMinusSrcAlpha);
```

This will be a non-issue

Fog

Adding fog to your world has more than one purpose. Besides trying to give the player the impression of actual fog, it can be used to obscure objects in the distance and have them gradually become clearer as the player gets closer. This is beneficial both in allowing you to reduce the number of geometry on the screen at one time (thus improving performance) and in preventing objects from suddenly popping into the view frustum.

There is more than one way to implement fog, but because OpenGL has a built in solution we will simply use that.

OpenGL Fog

OpenGL's built-in fog support works by blending each pixel with the color of the fog, using a blend factor dependent on the distance from the viewer, the density of the fog and the currently selected fog mode.

To use fog, you must first enable it:

```
GL.Enable(EnableCaps.Fog);
```

Fog has several parameters associated with it, which you can control with calls to `GL.Fog`

```
void GL.Fog(FogParameter, int);
void GL.Fog(FogParameter, float);
void GL.Fog(FogParameter, float[])
```

The values of the `GL.Fog(FogParameter)` enumeration are:

- **FogMode** This parameter specifies which blend equation to use when calculating fog. By default it's `Exp`. It can be set to:
 - `(int)FogMode.Linear`
 - `(int)FogMode.Exp`
 - `(int)FogMode.Exp2`
- **FogDensity** This parameter is a single value, representing the density of the fog. The default value is 1.0
- **FogStart** A single float, representing how far from the near plane the fog should start
- **FogEnd** A single float, representing how far from the near plane the fog should end
- **FogIndex** Specifies the color index to be used in 8-bit color mode
 - Don't bother, we don't use this!
- **FogColor** This specifies the color to be used for fog. It's an array, RGBA
- **FogCoordSrc** We don't touch this. It specifies the depth term of the fog blend equation. This is an extension, and is not always available.

That's all there is to fog. Because modern computers can handle far planes that are insanely far away (1000 to 5000) we tend to not use fog to hide things, we only tend to use it as an environmental effect when appropriate.

Because you will likely not need to use fog (I've NEVER used it) we're not going to do an exercise on it. Just know that you have to first enable it, then you can set parameters if the default fog doesn't look good.

Texture mapping

Nothing we have discussed so far can bring as much realism to a 3D scene as texture mapping. Lighting comes close, but it doesn't have near the impact a simple texture map can have when applied to a set of polygons. In this chapter you will learn to achieve a high level of realism through an introduction to the concept and implementation of texture mapping techniques in OpenGL.

In this chapter you will learn:

- The basics of texture mapping
- Texture coordinates
- Texture objects and texture binding
- Texture specifications with 1D, 2D, 3D and cube map textures
- Texture filtering
- Mipmaps and auto mip generation
- Texture parameters
- Wrap modes
- Level of details
- Texture environments and functions

An overview of texture mapping

In a nutshell *texture mapping* allows you to attach images to polygons in order to provide more realistic graphics. As an example, you could apply an image of the front of a book to a rectangular polygon; the polygon would appear as a visual representation of the front of the book. Another example would be to take a map of earth and texture map it to a sphere. You then have a 3D visual representation of earth. Nowdays, texture maps are used everywhere in 3D graphics, you're gonna have a hard time finding a game that doest use textures.

Texture maps are composed of rectengular arrays of date, each element of these arrays is called a **texel**. Altough they are rectangular arrays, textures can be mapped to non-rectangular objects.

In a picture, each pixel has 4 components (RGBA). Each of these components would be 1 entry in the rectangular texture array. Each component is represented by a value of 0 to 255. So the size of a texture is: `width * height * 4 * sizeof(char)` Each pixel of that texture is called a **texel**.

Why **texel**, why not pixel? As we map a texture to a 3D object, and the 3D object to screen, a single pixel in the source texture could occupy multiple pixels on screen (Because of wrapping around a 3D object that could be scaled, rotates, skewed, etc...). Because the mapping is not 1 to 1, the source of the image comes from texel coordinates and the destination on screen is pixel coordinates.

Usually developers use two-dimensional textures for graphics, however using one dimensional or even 3 dimensional textures is not unheard of. 1D textures have a width. 2D textures have a width and a height (these are the images we are used to as png files). 3D textures have a width, a height and a depth. 3D textures are sometimes called *volume textures*. We generally only use 1D and 2D textures. 1D is used almost exclusivley for "toon shading"

When you map a texture to a polygon, it will deform with the polygon. In other words, if you rotate the earth, it's texture will rotate along with the sphere. This applies to all deformations and transformations.

Using a texture map

As mentioned, textures are images that you apply to polygons. These images are either loaded from a file, or created in code. Once you have a texture in memory, you need to map it to an OpenGL texture object. But before you can do ANY of this, you must first enable texture mapping!

To enable / disable texture mapping you will most often use this call:

```
GL.Enable(EnableCap.Texture2D);
```

Other acceptable arguments are:

- EnableCap.Texture1D
- EnableCap.Texture2D
- EnableCap.Texture3D
- EnableCap.TextureCubeMap

You must enable the type of texture you want to use. More often than not that is going to be `Texture2D`. It's important that you only enable 1 texture type at a time. If you have `Texture1D` AND `Texture2D` enabled the render operation might or might not work. It's undefined. You must disable the old texturing before using the new. For example:

```
GL.Enable(EnableCap.Texture2D);
// Draw textured objects
GL.Disable(EnableCap.Texture2D);

GL.Enable(EnableCap.TextureCubeMap);
// Draw reflective objects
GL.Disable(EnableCap.TextureCubeMap);
```

This brings up the next point. When you have texturing enabled, all color info will come from the texture. So, what if you want to draw a textured polygon, but still render an untextured grid? You just disable the texture after you drew the polygon! You could also disable texturing before you draw the grid!

```
// Draw the grid unlit and untextured
GL.Disable(EnableCap.Texture2D);
GL.Disable(EnableCap.Lighting);
grid.Render();
// Now that the grid is drawn, re-enable lighting and texturing!
GL.Enable(EnableCap.Lighting);
GL.Enable(EnableCap.Texture2D);
```

Texture Objects

Texture objects are internal memory to OpenGL that hold texture data and parameters. Essentially, it's memory on the GPU. Managing memory on the GPU is dangerous, because of this OpenGL will not give you direct access to it. Instead you can't track this memory with a unique unsigned integer that acts as a handle.

This concept should already be familiar. In our 2D OpenTK Framework, when you loaded a texture it returned an integer. You then used this integer to draw the texture onto screen, and eventually to unload the texture. OpenGL does the same thing.

There are two ways to generate new texture handles. You can either request a single one at a time, or if you know how many textures you will need ahead of time you can request them in one big batch.

```
// Generate a single texture
int GL.GenTexture();
// Generate multiple textures
void GL.GenTextures(int n, out int[] textures);
```

Generating a single texture handle is straight forward. When generating multiple, the function takes as its first parameter the number of textures you want. As its second parameter it takes an out qualified integer array that is large enough to hold all requested handles.

Before generating textures, you must enable texturing. This is what we've learned so far put to code:

```
// Enable Texturing
GL.Enable(EnableCap.Texture2D);
// Generate a texture handle
int handle = GL.GenTexture();
// ??? Profit?
```

Generating a texture handle is useless without being able to actually put data in it. We will cover how this works over the next two sections.

Texture Binding

In most API's that use handles, functions which effect the handle take the handle as their first argument. Take for example how we did our 2D framework:

```
// This is NOT OpenGL, it's from the 2D framewrok  
  
// First we intialize the texture manager  
TextureManager.Instance.Initialize(Window);  
  
// Next we generate a texture handle  
int texBird = TextureManager.Instance.LoadTexture("Assets/Bird.png");  
  
// Finally we draw the texture by passing the handle  
// as the first parameter to the draw function  
TextureManager.Instance.Draw(texBird, new Point(0, 0));
```

OpenGL does not work like this.

Binding

OpenGL has the concept of a "currently bound texture". When you plan to use a texture for anything (rendering usually), you bind it to the OpenGL context (with the `GL.BindTexture` function). After a texture is bound, all subsequent operations that use textures will use the currently bound texture.

This is what the function call for binding a texture looks like:

```
GL.BindTexture(TextureTarget proxy, int texId);
```

To bind no texture, set the second argument (`texId`) to 0. By default, no texture is bound. What is the first argument? That `TextureTarget` enumeration.... It's a list of texture types that can be bound. We only care about the following entries:

- `TextureTarget.Texture1D`
- `TextureTarget.Texture2D`
- `TextureTarget.Texture3D`
- `TextureTarget.TextureCubeMap`

There are a lot of other options, but seeing how those are the four texture types we can actually create, they are the only ones we care about.

What happens if you enable `Texture2D`, then try to bind a `Texture1D` object? Nothing. No errors, nothing. But also, nothing will be drawn. More often than not you will be using `Texture2D` for this argument. If you are not getting anything rendering on screen, make sure you have the right texture unit enabled.

A bound texture will remain bound until it is either unbound, or deleted. It's **GOOD PRACTICE** to unbind your texture (by binding 0), after you are done drawing with it.

Here is what we've learned so far:

```
// Enable Texturing
GL.Enable(EnableCap.Texture2D);
// Generate a texture handle
int handle = GL.GenTexture();
// Bind the handle we generated as the active texture object
GL.BindTexture(TextureTarget.Texture2D, handle);
// ???
```

Now that we have generated a texture, and told OpenGL that we want to make changes to it by binding the texture handle, it's time to actually fill the texture with some data. We will discuss how to do this in the next section.

Specifying Textures

Now that a texture object has been created on the GPU and its associated handle has been bound as the active texture, it's time to actually fill that texture object with data.

It's important to make the distinction between GPU memory and CPU memory. You have control of CPU memory, but not GPU memory. Rendering is done mostly using GPU memory. Therefore, to render a texture, you must upload it to GPU memory.

The process of getting a texture into GPU memory is fairly straight forward.

- First, you must have the texture object you want to fill with data bound.
- Read the texture data into CPU memory.
- Upload to the GPU with `GL.TexImage2D`
- At this point, you can manually delete CPU memory
 - Or let the garbage collector do it, depends on how you loaded the texture

The `GL.TexImage2D` function is fairly straight forward, but it does have a lot of arguments

```
GL.TexImage2D(TextureTarget target, int level, PixelInternalFormat internalFormat, int width, int height, int border, PixelFormat sourceFormat)
```

Let's break each argument down

- **TextureTarget target**
 - Which texture are we loading data into? Texture1D, Texture2D, etc..
 - Most often (99% of the time) this will be `TextureTarget.Texture2D`
- **int level**
 - Specifies how many levels of detail the image has.
 - Level 0 is the base, each additional level is a new mip-map
 - We will cover mip-maps later. For now, keep this at 0
- **PixelInternalFormat internalFormat**
 - Specifies what the image is formatted for, Color, Light, etc..
 - 99% of the time you will use: `PixelInternalFormat.Rgba`
 - The other two that we use are:
 - `PixelInternalFormat.Rgb`
 - `PixelInternalFormat.Alpha`
 - There are other options, but they are not useful for games.
- **int width**
 - Specifies the width of the texture being loaded
 - Remember, you should be using a power of 2!
- **int height**
 - Specifies the height of the texture being loaded
 - Remember, you should be using a power of 2!
- **int border**
 - Specifies a memory border, NOT a pixel border
 - When written to OpenGL throws an error
 - THIS MUST BE 0 (I don't know why they included this parameter)
- **PixelFormat sourceFormat**

- o Specifies how the source data is laid out
- o Values are Argb, Bgra, Rgb, Alpha, etc...
- **PixelType sourceType**
 - o Specifies what data type we are storing the source as
 - o Most often this will be an unsigned byte, values are:
 - **PixelType.UnsignedByte**
 - **PixelType.Short**
 - **PixelType.Int**
 - o There are more types, but those are the 3 you will most often use
- **IntPtr data**
 - o An IntPtr is an unsafe data type
 - o It can point to any array of numeric data (int[], byte[], etc...)
 - o This is a long, single dimensional array containing the pixels to upload to the GPU

The function looks complicated, but you only have to really write it once in a friendly wrapper. After calling this function, you can discard the pixel data you are holding onto on the CPU, as the function will have uploaded it to the GPU

Decoder

So, how do we get that array of bytes that represents the texture? We have to decode a texture from its source format (png, jpg, tga, etc...) into an array of bytes. We're going to use built-in windows functions to do this.

In order to use windows to decode, you must include a reference to **System.Drawing**, as it contains the **Bitmap** class that we will use to decode the texture. The following block of code is commented to explain what is involved in decoding a texture.

```
private int LoadGLTexture(string filename, out int width, out int height) {
    if (string.IsNullOrEmpty(filename)) {
        Error("Load texture file path was null");
        throw new ArgumentException(filename);
    }

    // Generate a handle on the GPU
    int id = GL.GenTexture();

    // Bind the handle to be the active texture.
    GL.BindTexture(TextureTarget.Texture2D, id);

    /* TODO:
     * Set appropriate min and mag filters
     * we will talk about these in the next section
     */

    // Allocate CPU system memory for the image
    // This will load the encoded texture into CPU memory
    Bitmap bmp = new Bitmap(filename);

    // Decode the image data and store the byte array into CPU memory
    BitmapData bmp_data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height), ImageLockMode.ReadOnly, System.Drawing.;

    /* TODO:
     * Check bmp.Width and bmp.Height, if they are not a power
     * of two, throw an error
     */

    // Upload the image data to the GPU
    GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0, OpenTK.Graphics.;

    // Mark CPU memory eligible for GC, disposing it
}
```

```

        bmp.UnlockBits(bmp_data);
        bmp.Dispose();

        // Return the textures width, height and GPU ID
        width = bmp.Width;
        height = bmp.Height;
        return id;
    }

```

Using this function is pretty easy. It returns the texture handle and gives you the width & height of the loaded texture as out parameters:

```

int texWidth = -1;
int texHeight = -1;
int texHandle = LoadGLTexture("File.png", out texWidth, out texHeight);

```

So far

Reading the above code, you will notice that we've wrapped not only `GL.TexImage2D`, but also `GL.GenTexture` and `GL.BindTexture` into the `LoadGLTexture` helper function. So, our code to load (and eventually display) a texture becomes:

```

// Enable Texturing
GL.Enable(EnableCap.Texture2D);
// Generate a texture handle, bind it and load it with data
int width = -1;
int height = -1;
int handle = LoadGLTexture(file.png, out width, out height);

```

Width & Height

By far the easiest way to obtain the width and height of a texture is to store it at the time of loading that texture. However, you don't HAVE to do it this way. You can get the width or height of a texture anytime with the `GetTextureParameter` function.

```

int GetWidth(int textureId) {
    GL.BindTexture(TextureTarget.Texture2D, textureId);
    int result = 0;
    GL.GetTexLevelParameter(TextureTarget.Texture2D, 0, GetTextureParameter.TextureWidth, out result);
    return result;
}

```

But just storing the width / height at load time is much easier and much more performant!

Other loading methods

The method we used to load textures here is by far the simplest. It relies on Windows to decode texture files for us. Sometimes, this isn't an option though. For example, on an iPhone. So, how can we decode textures on non-windows platforms?

A common method is to use [stb_image](#) or [LodePng](#). Both are C libraries that you have to compile into a .dll file and

link against. Once compiled, you can use C#'s interop features to access the C functions.

If that sounds like a lot of work just to load a texture, well that's because it is! Another way to get texture loading to work is to browse NuGet for a png or jpg decoder package. NuGet is the Visual Studio package manager we used to link against OpenGL (In the form of OpenTK) and NAudio.

We actually use NAudio that we got through NuGet to decode mp3 files in the OpenTK framework. You can find LodePNG on NuGet if you want to play around with a third party decoder. LodePNG is actually faster than the built in Windows decoder.

Min and Mag Filters

We're almost done! We've managed to load some memory onto the GPU at this point, but if we tried to render that memory as a texture, nothing would show up. This is because our texture loading code neglected to specify min and mag filters. There is a TODO comment section in the function.

Min and mag filters describe to OpenGL what to do when we are trying to render a 256x256 texture on a 512x512 surface, or even a 128x128 surface! There is no clear way to map the pixels of a smaller image to a larger surface or a larger image to a smaller surface. Each approach in handling the problem has its own ups and downs, so OpenGL lets you pick what to do.

What is a Min filter

Min filter is the minification filter. It is applied when an image is zoomed out so far that multiple pixels (texels) on the source image make up a single pixel (fragment) on the display screen. There are two common settings

- **Nearest**
 - Nearest neighbor filtering, does not attempt to scale the image
 - Result is sharp / pixelated
- **Bilinear**
 - Bilinear filtering, attempts to resize the image
 - Result is soft / blurred

Either way you set the min filter, the resulting image will be less than ideal because certain pixels have to be skipped. Usually, min filtering only needs to effect far away objects, so this isn't too big of an issue.



GL_NEAREST



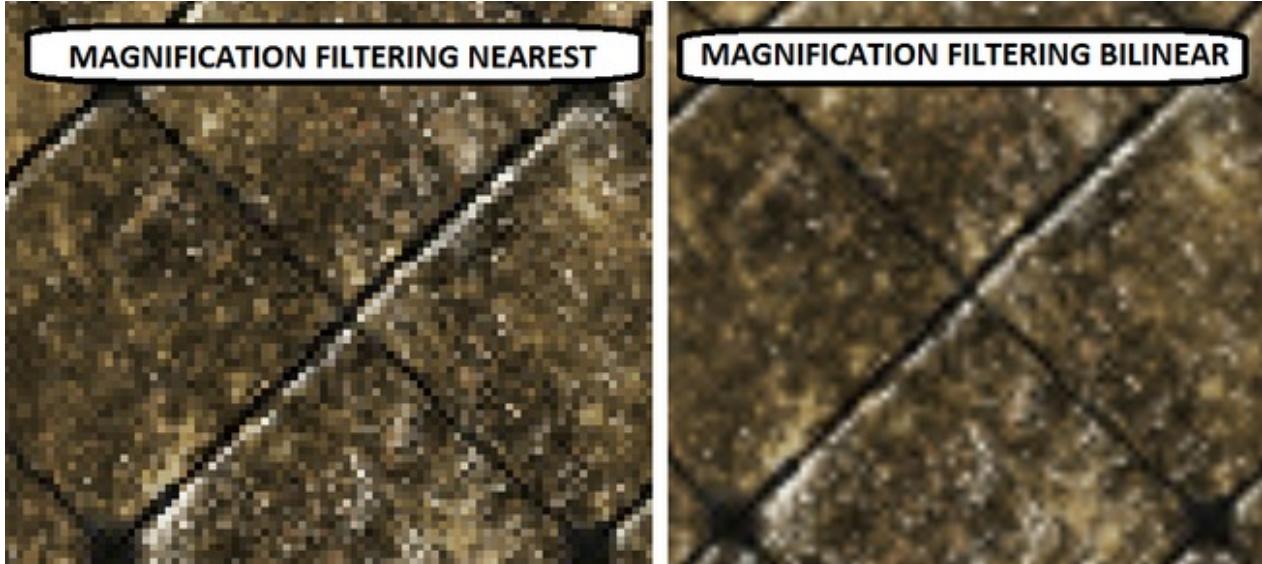
GL_LINEAR

What is a Mag filter

Mag filter is the magnification filter. It is applied when an image is zoomed in so close that one pixel (texel) on the source image takes up multiple pixels (fragments) on the display screen. There are two common settings for mag filtering:

- **Nearest**

- Nearest neighbor filtering means no scale is applied to the texture
- The resulting image will be sharp, but very pixelated
- **Bilinear**
 - Use bilinear filtering to resize (enlarge) the source image
 - The result will be blurred, but the image will look smooth



Together now

99% of the time you will set these properties to be the same. That is you will have a nearest min and mag, or a bilinear min and mag. It's RARE (I have never done this) to need a nearest mag and a bilinear min.

2D games (Like our OpenTK framework) tend to use nearest neighbor filtering. This helps keep pixels looking sharp and crisp! It's an art style, after all you don't want mario to have blurry edges.

In contrast 3D games tend to use bilinear filtering. Because the world is smooth and continuous, you really want to maintain that illusion, even if it means blurring your image a little.

There are of course exceptions. Minecraft for instance uses nearest neighbor filtering despite being a 3D game. And most Klei games use bilinear filtering, even though they are 2D games.

Code

Setting the min and mag filters in code is pretty straight forward. You call the `GL.TexParameter` function two times, once for the min and once for the mag filters.

```
GL.TexParameter(TextureTarget target, TextureParameterName param, int value)
```

The first argument, **target** is of course which texture this command targets. More often than not the value of this is going to be `Texture2D`. The second parameter, **param** is the important one, it tells OpenGL what texture parameter you are setting. We want to set `TextureMagFilter` OR `TextureMinFilter`. The last parameter is an integer, this is a bit of a magic number. There is an enumeration `TextureMagFilter`, you can cast the values of this enum into an int for the last parameter

Knowing what the function looks like, this is how you would go about setting a linear filter

```
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);
```

And this is how you would set a nearest filter

```
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Nearest);
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Nearest);
```

Load Texture

Let's retrofit setting the min and mag filters into the `LoadGLTexture` function we wrote in the last section. We're going to take advantage of the fact that both min and mag filters tend to be set to the same value by simply adding one new argument to the function.

This new argument, **bool nearest** if true will make the function use nearest filtering. If false, it will use bilinear.

```
private int LoadGLTexture(string filename, out int width, out int height, bool nearest) {
    if (string.IsNullOrEmpty(filename)) {
        Error("Load texture file path was null");
        throw new ArgumentException(filename);
    }

    // Generate a handle on the GPU
    int id = GL.GenTexture();

    // Bind the handle to the be the active texture.
    GL.BindTexture(TextureTarget.Texture2D, id);

    /////////////////////////////////
    // THIS IS NEW
    Set appropriate filters
    if (nearest) {
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Nearest);
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Nearest);
    }
    else {
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);
    }
    /////////////////////////////////

    // Allocate CPU system memory for the image
    // This will load the encoded texture into CPU memory
    Bitmap bmp = new Bitmap(filename);

    // Decode the image data and store the byte array into CPU memory
    BitmapData bmp_data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height), ImageLockMode.ReadOnly, System.Drawing.);

    /* TODO:
     *   Check bmp.Width and bmp.Height, if they are not a power
     *   of two, throw an error
     */

    // Upload the image data to the GPU
    GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0, OpenTK.Graphics.;

    // Mark CPU memory eligible for GC, disposing it
    bmp.UnlockBits(bmp_data);

    // Return the textures width, height and GPU ID
}
```

```
    width = bmp.Width;
    height = bmp.Height;
    return id;
}
```

In this example we set the texture filter after the texture was bound, but before it is filled with data. So long as the texture is bound, we can set its filtering mode any time, you don't HAVE to set it before it is filled with data. As a matter of fact, you can change this during runtime!

However it's considered best practice to set the filtering before filling a texture with data, and changing the filtering at runtime has a MUCH higher performance penalty than just having a second, duplicate texture with different filtering. So, follow the above convention.

Done loading

Wow, that's it. This image is now ready to be displayed! In general you will want to load all your images in an initialize function, delete them all in a shutdown function and draw them all in a render function.

We've finally gone through all the code required in the initialize section of your application. Let's take a look at what is involved in deleting textures on shutdown before we actually render anything!

Deleting textures

Now that we know exactly what we have to do in the initialize function of a textured application, let's take a quick look at how to clean up a texture handle. This could not be any easier.

First, make sure the texture you are deleting isn't bound! Remember, you can always bind 0 as the active texture (effectively binding null). Next, you need to pass the textures handle to the `GL.DeleteTexture` function. This function has two variations:

```
void GL.DeleteTexture(int texture);
void GL.DeleteTextures(int n, int[] textures);
// The first argument of the second function (n) is
// the length of the textures array.
```

One of them takes a single handle and deletes it, the other takes an array of handles and deletes them all. It should be common sense, but once a texture is deleted you should not use it. If you do, no error will be thrown, but undefined behaviour will happen!

So Far

We're almost ready to draw some textured stuff! Let's recap the texturing process so far:

```
int texture1 = -1;
int texture2 = -1;

void Initialize() {
    // Enable Texturing
    GL.Enable(EnableCap.Texture2D);

    int width = -1;
    int height = -1;

    // Load in textures, we don't even care
    // about the width / height right now
    // Be sure to use the version of this
    // function that specifies a min & mag filter!
    texture1 = LoadGLTexture("file.png", out width, out height, true);
    texture2 = LoadGLTexture("file2.png", out width, out height, false);
}

void Render() {
    // TODO: Render textures!
}

void Shutdown() {
    // Now that we are done, delete the texture handles!
    GL.DeleteTexture(texture1);
    GL.DeleteTexture(texture2);

    // And i like to set any int references to invalid values
    // just so i know that these are no longer usable
    texture1 = texture2 = -1;
}
```

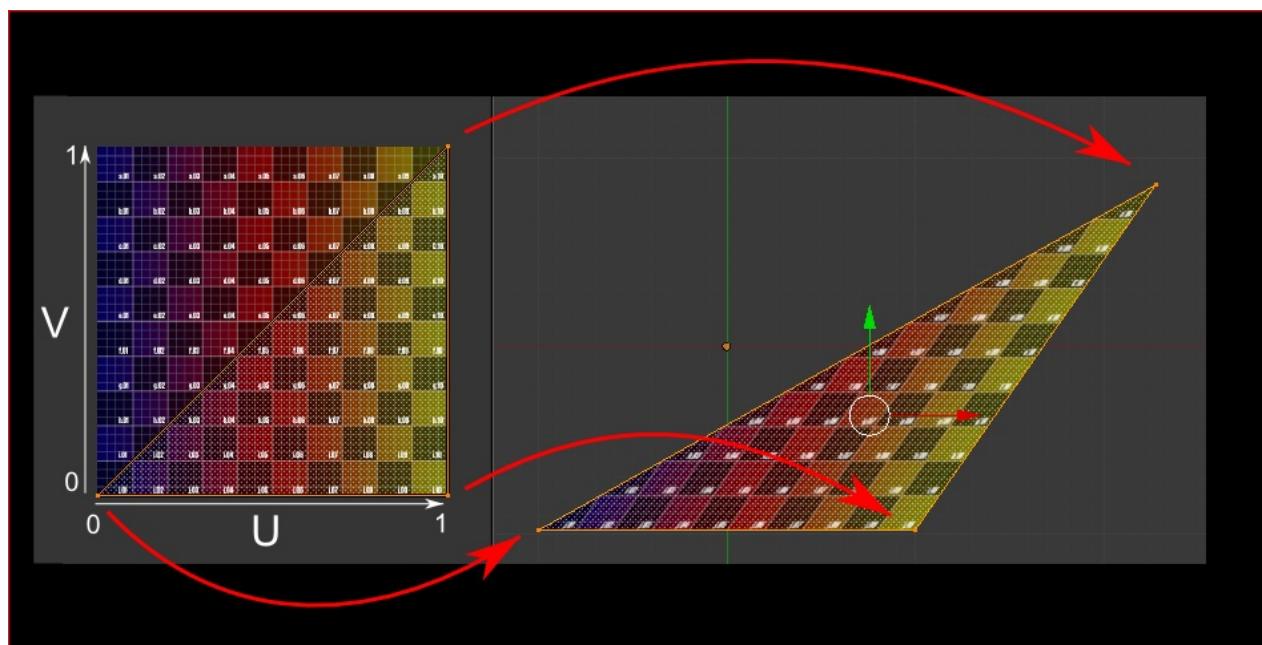
Texture Coordinates

Now that we can load and unload textures it's finally time to texture something! In order to apply a texture to a 3D model, you must first specify some texture coordinates for each vertex of that model. So far we've given a vertex a position, a color, a normal; now we add texture coordinates to the list of attributes a vertex can have.

Texture coordinates have some special vocabulary. We call the texture coordinates **UV coordinates**. This is because a texture is not read on an X-Y axis like you might expect, rather a texture is read using a U-V axis. The U axis is horizontal, the V axis is vertical. 0, 0 is in the bottom left. When we use subscript notation, U comes first: (U, V)

The other thing to know about U-V space is that it is normalized. It doesn't matter if a texture is 124x124, 256x256, 512x1024 or whatever. The U-V coordinate space always extends from 0 to 1. The center pixel of an image is always at point (0.5f, 0.5f).

In this image, the left side shows a png picture, the right side shows the picture being applied onto a triangle. The triangle is also outlined on top of the image. The UV coordinates of this triangle's vertices are (0, 0), (1, 1), (1, 0).



What happens when your texture isn't a square? Let's say you have 512x1024. Then the UV coordinate (0.5f, 0.5f) maps to the pixel at 256x512. I can't stress this enough, **images should always be square and a power of two**. This will help keep coordinates simple, and will run faster on your graphics card.

While we're talking about textures, let's review why a power of two texture is faster on a graphics card. A U-V coordinate doesn't map to a pixel, it maps to a **texel**. As such, each pixel on an image might take up one or more pixels on the actual render target, or no pixels at all. In order to convert texels to pixels, a graphics card must divide by 2.

If you remember WAY back to when we started programming, we briefly talked about bitshifting. Let's take the binary number for 4 for example: `0000 0100`. If we right shift this by 1 `>> 1` the result is two: `000 0010`. We just achieved the same thing as a division by two, except this is about 500 times less expensive than division! This trick only works with powers of two.

Tex Coords

Alright, enough theory. You can specify texture coordinates for vertices using the `GL.TexCoord2` function. This function has the following signature:

```
GL.TexCoord2(float s, float t);
```

Where S is the U axis and T is the V axis. So, if we wanted for example to draw a textured quad on screen, we would specify UV coordinates for each vertex, like so:

```
GL.Begin(PrimitiveType.Quads);
    GL.TexCoord2(0, 1);           // What part of the texture to draw
    GL.Vertex3(left, bottom, 0.0f); // Where on screen to draw it

    GL.TexCoord2(1, 1);           // What part of the texture to draw
    GL.Vertex3(right, bottom, 0.0f); // Where on screen to draw it

    GL.TexCoord2(1, 0);           // What part of the texture to draw
    GL.Vertex3(right, top, 0.0f); // Where on screen to draw it

    GL.TexCoord2(0, 0);           // What part of the texture to draw
    GL.Vertex3(left, top, 0.0f); // Where on screen to draw it
GL.End();
```

Notice how I define the UV coordinate before the vertex. You specify whatever attributes a vertex needs, you could even specify all the attributes we've used until now for a single vertex!

```
GL.Begin(PrimitiveType.Triangles);
    GL.TexCoord2(1f, 1f);
    GL.Normal3(0f, 1f, 0f);
    GL.Color3(0f, 1f, 0f);
    GL.Vertex3(0f, 1f, 0f);

    GL.TexCoord2(1f, 0f);
    GL.Normal3(0f, 1f, 0f);
    GL.Color3(1f, 0f, 0f);
    GL.Vertex3(1f, 0f, 0.0f);

    GL.TexCoord2(0f, 0f);
    GL.Normal3(0f, 1f, 0f);
    GL.Color3(0f, 0f, 1f);
    GL.Vertex3(-1f, 0f, 0f);
GL.End();
```

The order in which you define the vertex attributes does not matter so long as A) you are consistent within the `Begin` / `End` calls and B) the `GL.Vertex3` call specifying position comes last.

Most of the time when you specify a texture you will not specify a color. But position and normal will usually be specified for all 3D models. UI and 2D games tend to only use UV-Coords and position.

So far

It's not enough to just specify texture coordinates. Texturing must be enabled, and a valid texture must be bound. Let's see some code as to what it takes to draw a textured quad:

```
int textureHandle = -1;
```

```

int textureWidth = -1;
int textureHeight = -1;

void Initialize() {
    // Enable Texturing
    GL.Enable(EnableCap.Texture2D);

    // Take note, we store the width and height!
    textureHandle = LoadGLTexture("file.png", out textureWidth, out textureHeight, true);
}

void Render() {
    // World coordinates of the quad we are drawing
    float left = 0f;
    float right = 20f;
    float top = 0f;
    float bottom = 10f;

    // We have to bind a valid texture to draw
    GL.BindTexture(TextureTarget.Texture2D, textureHandle);

    GL.Begin(PrimitiveType.Quads);
        GL.TexCoord2(0, 1);
        GL.Vertex3(left, bottom, 0.0f);

        GL.TexCoord2(1, 1);
        GL.Vertex3(right, bottom, 0.0f);

        GL.TexCoord2(1, 0);
        GL.Vertex3(right, top, 0.0f);

        GL.TexCoord2(0, 0);
        GL.Vertex3(left, top, 0.0f);
    GL.End();

    // You don't have to do this, but i don't like leaving bound textures
    GL.BindTexture(TextureTarget.Texture2D, 0);
}

void Shutdown() {
    // Now that we are done, delete the texture handles!
    GL.DeleteTexture(textureHandle);

    // Invalidate references
    textureHandle = -1;
}

```

That's a lot of code, and it's even more when you consider that all our texture loading code is nicely wrapped up in the `LoadGLTexture` function, not shown here. But that's what it takes to draw a textured primitive.

So, let's pretend that the quad we drew is a health bar. And the health bar is on an atlas. Our artist tells us that the UV-Coordinates for this health bar are at top-left:125x34, bottom-right(150, 50). How do we take those pixel coordinates and convert them into UV-coords?

By **normalizing** them. That is, divide the X pixel by the width or the image, and the Y pixel by the height of the image. This will put the pixel coordinates into a normal space. Let's take a look at how we might change the render function to do this:

```

void Render() {
    // Normalized uv coordinates, remember we kept a reference
    // to the width and height of the image, returned
    // from the LoadGLTexture function.
    float uv_left = 125f / textureWidth;
    float uv_right = 150f / textureWidth;
    float uv_top = 34f / textureHeight;
    float uv_bottom = 50f / textureHeight;
}

```

```

// World coordinates of the quad we are drawing
float left = 0f;
float right = 20f;
float top = 0f;
float bottom = 10f;

// We have to bind a valid texture to draw
GL.BindTexture(TextureTarget.Texture2D, textureHandle);

GL.Begin(PrimitiveType.Quads);
    GL.TexCoord2(uv_left, uv_bottom);
    GL.Vertex3(left, bottom, 0f);

    GL.TexCoord2(uv_right, uv_bottom);
    GL.Vertex3(right, bottom, 0f);

    GL.TexCoord2(uv_right, uv_top);
    GL.Vertex3(right, top, 0f);

    GL.TexCoord2(uv_left, uv_top);
    GL.Vertex3(left, top, 0f);
GL.End();

// You don't have to do this, but i don't like leaving bound textures
GL.BindTexture(TextureTarget.Texture2D, 0);
}

```

You will notice in my code that I map the Y axis of my UV-Coordinates in reverse. This is because having (0, 0) on the bottom left makes no sense to me. So I map the UV's in reverse, effectively putting (0, 0) at the top left of uv-space.

One trick I like to use to avoid dividing twice is reciprocal multiplication. This code:

```

float uv_top = 34f / textureHeight;
float uv_bottom = 50f / textureHeight;

```

can be written as

```

float inverseHeight = 1f / textureHeight;
float uv_top = 34f * inverseHeight;
float uv_bottom = 50f * inverseHeight;

```

The final values of `uv_top` and `uv_bottom` will be the same, but instead of doing two divisions we now do 1 division and two multiplications. Which is actually faster. Not by a lot, but still faster.

What's next

Before we move on to the "putting it all together" section where we actually write code I want you to take a peek into the 2DOpenTKFramework we've been using to make 2D games. Specifically, I want you to check out the [TextureManager.cs](#) file, it's the one with all the texture goodies.

There is a specific function in there, it takes a texture-id and a screen rectangle. It draws the texture in its entirety to the screen. The signature of this function is:

```

public void Draw(int textureId, Point screenPosition)

```

It's heavily commented, it should be easy to see how it works. If you have a pretty good grasp on how that function works, and feel up to a challenge, take a look at it's cousin with the overrides:

```
public void Draw(int textureId, Point screenPosition, PointF scale, Rectangle sourceSection)
```

This function does a bunch of math (mostly multiplication) to convert pixel coordinates on the source texture into normalized texture coordinates.

Putting it all together

Now we know everything we need to know to make a simple textured scene! Nothing left to do except actually make something textured! In this section i'm going to walk you trough setting up a simple textured scene, but the actual texturing work will be up to you.

Putting together the test scene

The name of this new test scene will be `TexturedPlanes`, so create it in a file called `TexturedPlanes.cs`. For the last few examples, they have all extended a sample scene, not the empty game scene. I want to make sure we start from scratch here, notice that the `TexturedPlanes` class extends the `Game` scene, not the `LightingExample` class like the lighting examples before this.

So, we're going to need to make a member grid, inside the initialize function we will make a new grid. Also inside of initialize we need to enable depth testing for a proper depth buffer, as well as face culling. The shutdown function is going to stay empty for now.

```
using System.Drawing;
using System.Drawing.Imaging;
using OpenTK.Graphics.OpenGL;
using Math_Implementation;

namespace GameApplication {
    class TexturedPlanes : Game {
        protected Grid grid = null;

        public override void Initialize() {
            base.Initialize();
            grid = new Grid(true);
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
        }

        public override void Shutdown() {
            base.Shutdown();
        }
    }
}
```

The render function is going to set the camera at position (-7, 5, -7), looking at point (0, 0, 0). So we are going to load the appropriate view matrix.

Then we're going to render our standard grid background. Take note, when we draw the grid we disable texturing, then re-enable texturing. This is because the grid color should come from `GL.Color3`, not the active texture.

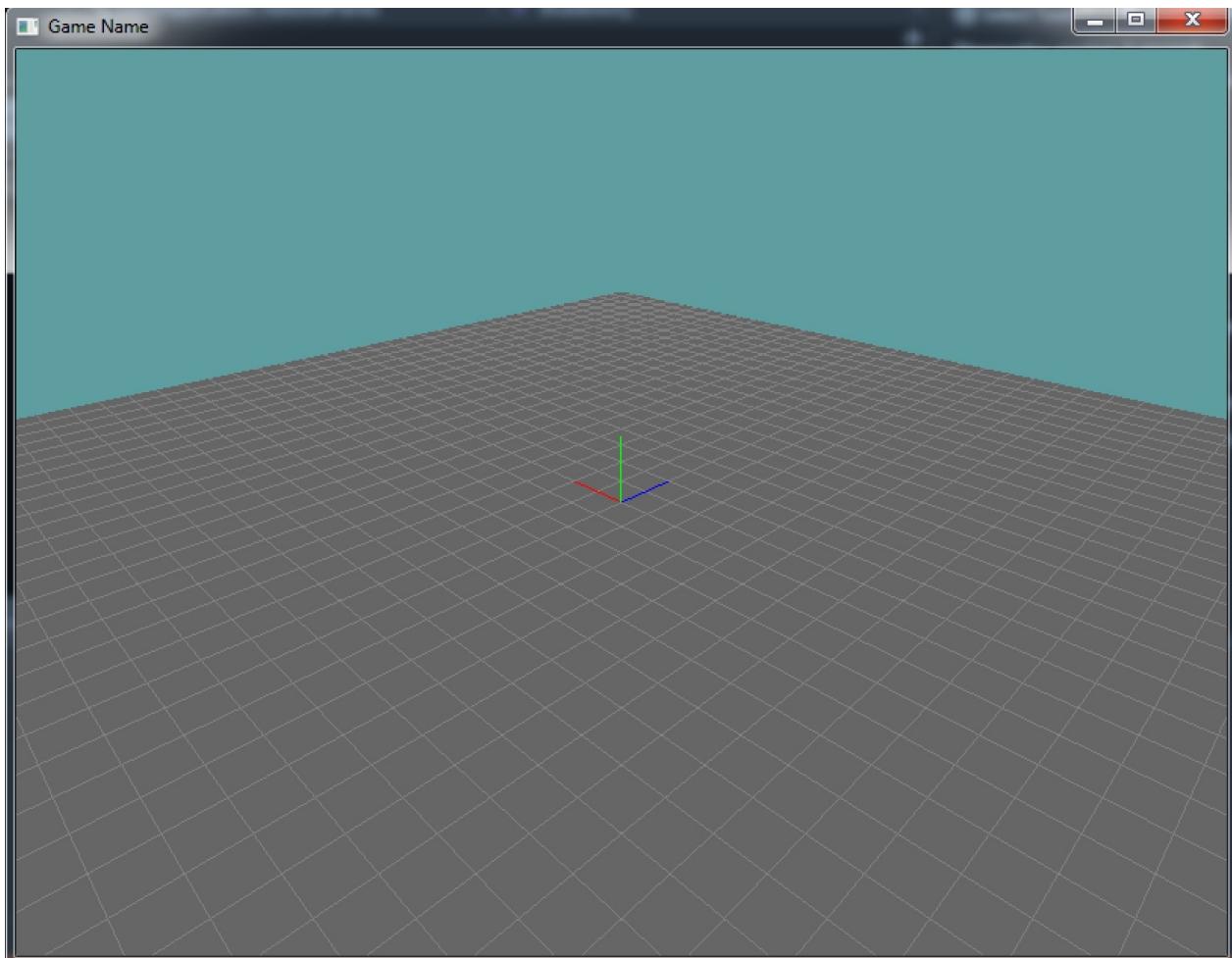
```
public override void Render() {
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(-7.0f, 5.0f, -7.0f), new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
    GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

    GL.Disable(EnableCap.Texture2D);
    GL.Disable(EnableCap.DepthTest);
    grid.Render();
    GL.Enable(EnableCap.DepthTest);
    GL.Enable(EnableCap.Texture2D);
}
```

Finally the resize function is going to set the viewport and load an updated projection matrix, then set the active matrix back to the modelview matrix.

```
public override void Resize(int width, int height) {
    GL.Viewport(0, 0, width, height);
    GL.MatrixMode(MatrixMode.Projection);
    float aspect = (float)width / (float)height;
    Matrix4 perspective = Matrix4.Perspective(60.0f, aspect, 0.01f, 1000.0f);
    GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
    GL.MatrixMode(MatrixMode.Modelview);
}
}
```

At this point, the test scene should look like this:



Now, let's add a quad to the scene that is to be rendered. We are going to draw this quad using two triangles. I'll make sure to comment the code to specify which vertex is which corner of the quad. Because we use two triangles, two of the corners will be defined twice.

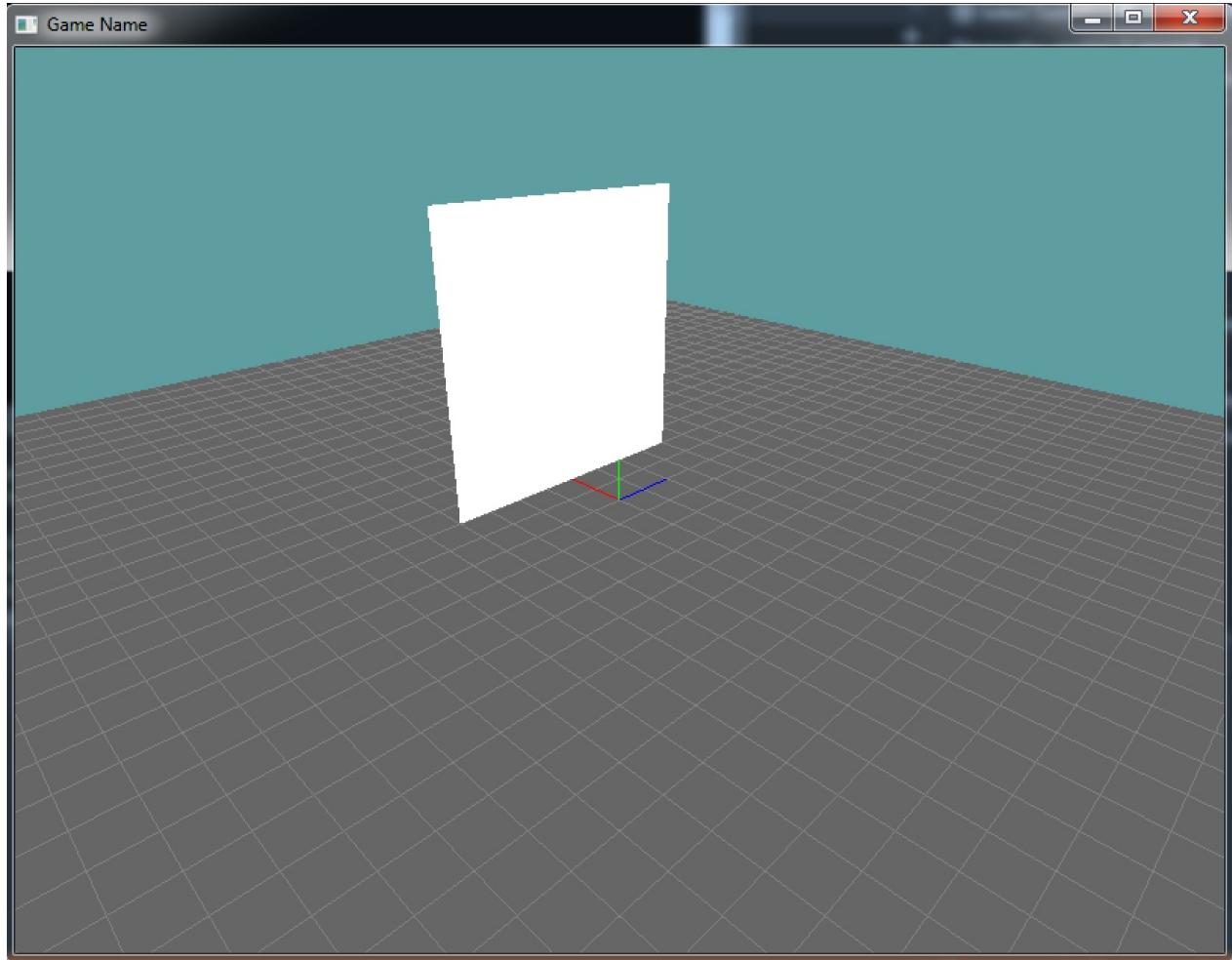
Modify the Render function, by adding this code at it's end:

```
GL.Color3(1f, 1f, 1f);
GL.Begin(PrimitiveType.Triangles);

GL.Vertex3(1, 4, 2); // Top Right
GL.Vertex3(1, 4, -2); // Top Left
GL.Vertex3(1, 0, -2); // Bottom Left
```

```
GL.Vertex3(1, 4, 2); // Top Right  
GL.Vertex3(1, 0, -2); // Bottom Left  
GL.Vertex3(1, 0, 2); // Bottom Right  
  
GL.End();
```

Your scene should now look like this:



On your Own

First, make an assets directory and save this image into it:

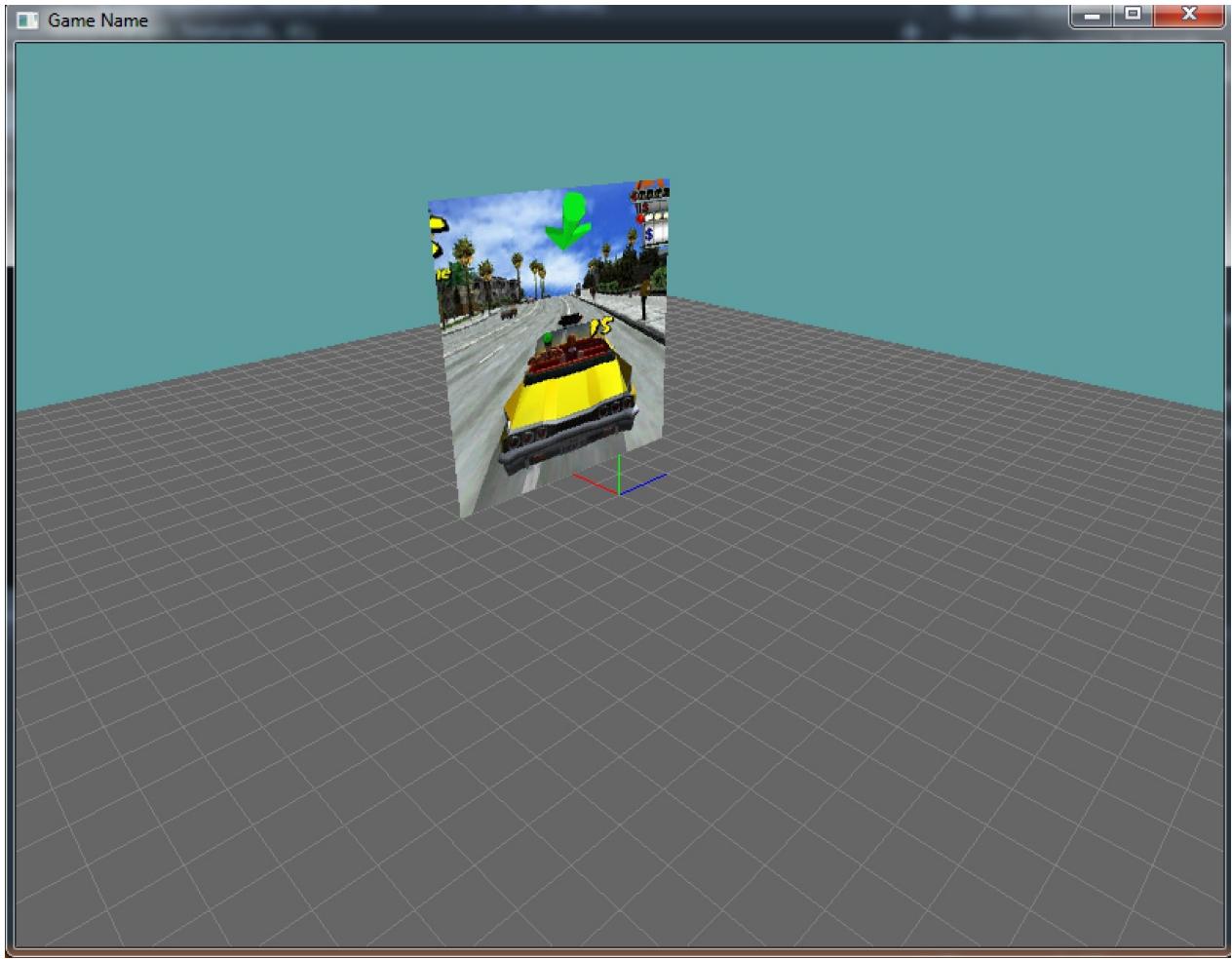


Remember, you have to set visual studio's working directory to one above the asset directory for loading resources! Just like with the 2D games.

We are going to do everything inline for now, so no LoadTexture helper function. Back to our code, do the following:

- Make a new integer member variable
 - This is going to be a texture handle
- In initialize, enable texturing
- In initialize, generate a texture handle
 - Assign the result to the member variable you created earlier
- In initialize, bind the new texture handle
- In initialize, set the min and mag filters to linear
- In initialize, load the texture data into the handle
 - This can be done in 5 lines of code, again no helper function
 - If you get stuck, look at the "Loading Help" sub page
- In shutdown, delete the texture handle
 - Remember to unbind it first!
- In render, bind the texture handle before drawing the quad
 - You MUST do this before GL.Begin
- In Render, add UV coordinates to each vertex

Running your game should show the textured quad. It should look like this:



Before we render the quad we set the color to white with this code: `GL.Color3(1f, 1f, 1f);`. Try setting that to blue to see how vertex colors affect textures.

Adding some detail

Rendering a textured quad is interesting, but we can make this a bit better. Let's add two more quads and explore how to render images with some alpha in them! Just like above, i'm going to walk you through adding the geometry for these images to the scene, but then it's going to be all you when it comes to actually texturing them.

First things first though, save the following image to your Assets directory. I call my version of it houses.png



House 1

X: 0, Y: 0
W: 186, H: 326



House 2

X: 332, Y: 0
W: 180, H: 336

Let's modify the render function. After the crazy taxy texture is rendered, add the following code:

```
// TODO: When you do texturing, remove the Disable call
GL.Disable(EnableCap.Texture2D);

// House 1
GL.Color3(1f, 1f, 1f);
GL.PushMatrix();
    GL.Translate(-1f, 0.5f, -1f);
    GL.Rotate(-130f, 0f, 1f, 0f);
    GL.Scale(0.57f, 1f, 1f);
    GL.Scale(3f, 3f, 3f);
    GL.Begin(PrimitiveType.Triangles);
    GL.Vertex3(0.5, 0.5, 0); //top right
    GL.Vertex3(-0.5, 0.5, 0); //top left
    GL.Vertex3(-0.5, -0.5, 0); //bottom left

    GL.Vertex3(0.5, 0.5, 0); //top right
    GL.Vertex3(-0.5, -0.5, 0); //bottom left
    GL.Vertex3(0.5, -0.5, 0); //bottom Right
GL.End();
GL.PopMatrix();

// House 2
GL.Color3(1f, 1f, 1f);
GL.PushMatrix();
    GL.Translate(-2f, 0.5f, -3f);
    GL.Rotate(-130f, 0f, 1f, 0f);
    GL.Scale(0.53f, 1f, 1f);
    GL.Scale(3f, 3f, 3f);
    GL.Begin(PrimitiveType.Triangles);
        GL.Vertex3(0.5, 0.5, 0); //top right
```

```

GL.Vertex3(-0.5, 0.5, 0); //top left
GL.Vertex3(-0.5, -0.5, 0); //bottom left

GL.Vertex3(0.5, 0.5, 0); //top right
GL.Vertex3(-0.5, -0.5, 0); //bottom left
GL.Vertex3(0.5, -0.5, 0); //bottom Right

GL.End();
GL.PopMatrix();

// TODO: When you do texturing, remove the Enable call
GL.Enable(EnableCap.Texture2D);

```

For now we disable texturing before drawing the quads, then enable it after we're done. We do this so we can confirm that the white boxes are rendered correctly. If we didn't do this, the color of those boxes would be undefined, as OpenGL would try to read the color from the texture, for a model with no UV coordinates.

What this does is simple. We render a plane with a width of 1 and a height of 1 (a depth of 0). Then we use matrix transformations to move the plane into position in the world, rotate it towards the camera, adjust its aspect ratio and scale it up. The following lines adjust the aspect ratio:

```

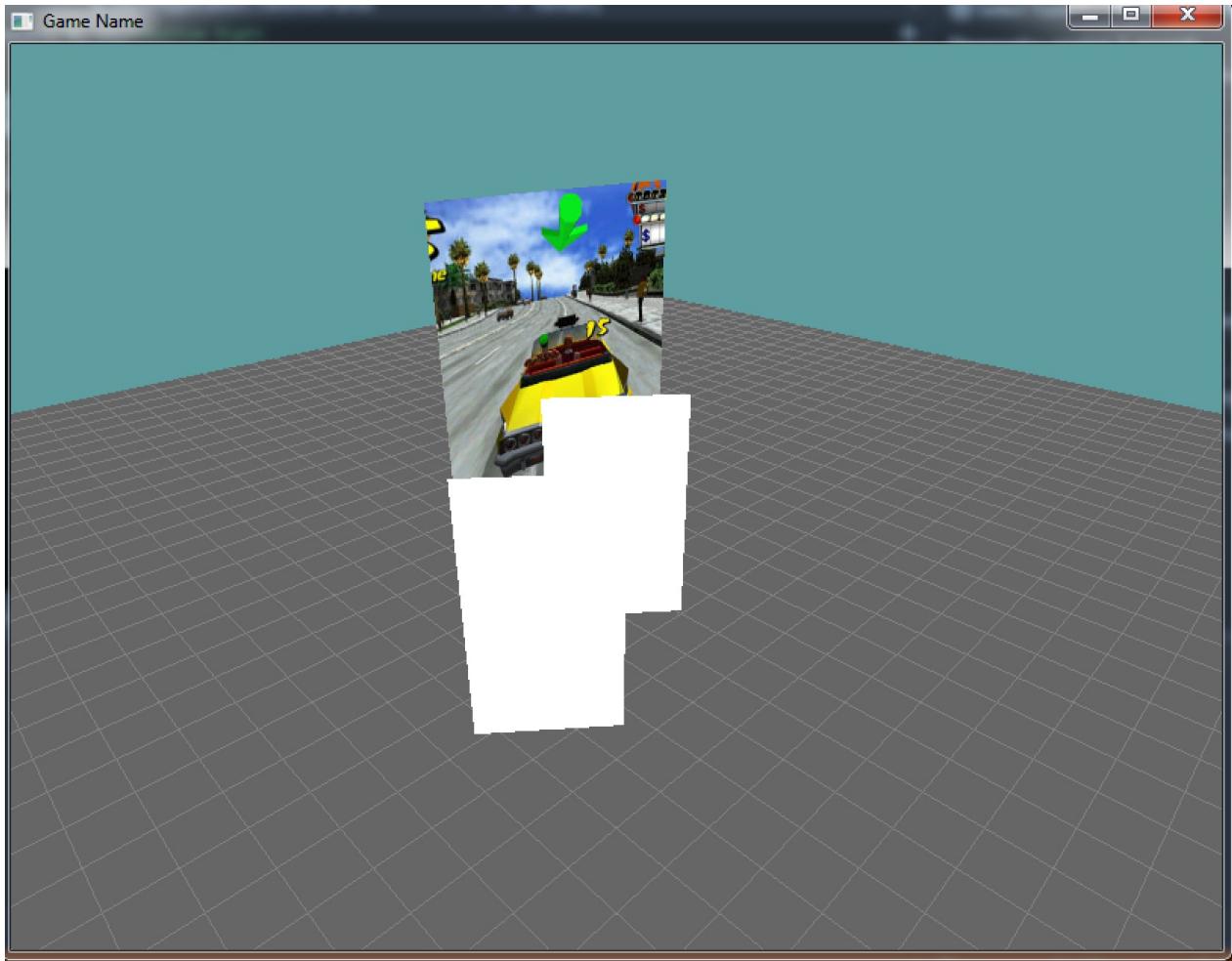
...
    GL.Scale(0.57f, 1f, 1f);
...
    GL.Scale(0.53f, 1f, 1f);
...

```

Where do 0.57 and 0.53 come from? Just like with the screen aspect ratio, any aspect is width divided by height. The first building is 186/326 which equals 0.57. The second building is 180/336 which equals 0.53. Of course I'm rounding the number a bit ;).

The size and location of the texture bits is on the actual texture sheet. The aspect ratio of an image is the scale of its width compared to its height. So at a scale of 1, the width of an image is its aspect.

Running the game, your scene should look like this:

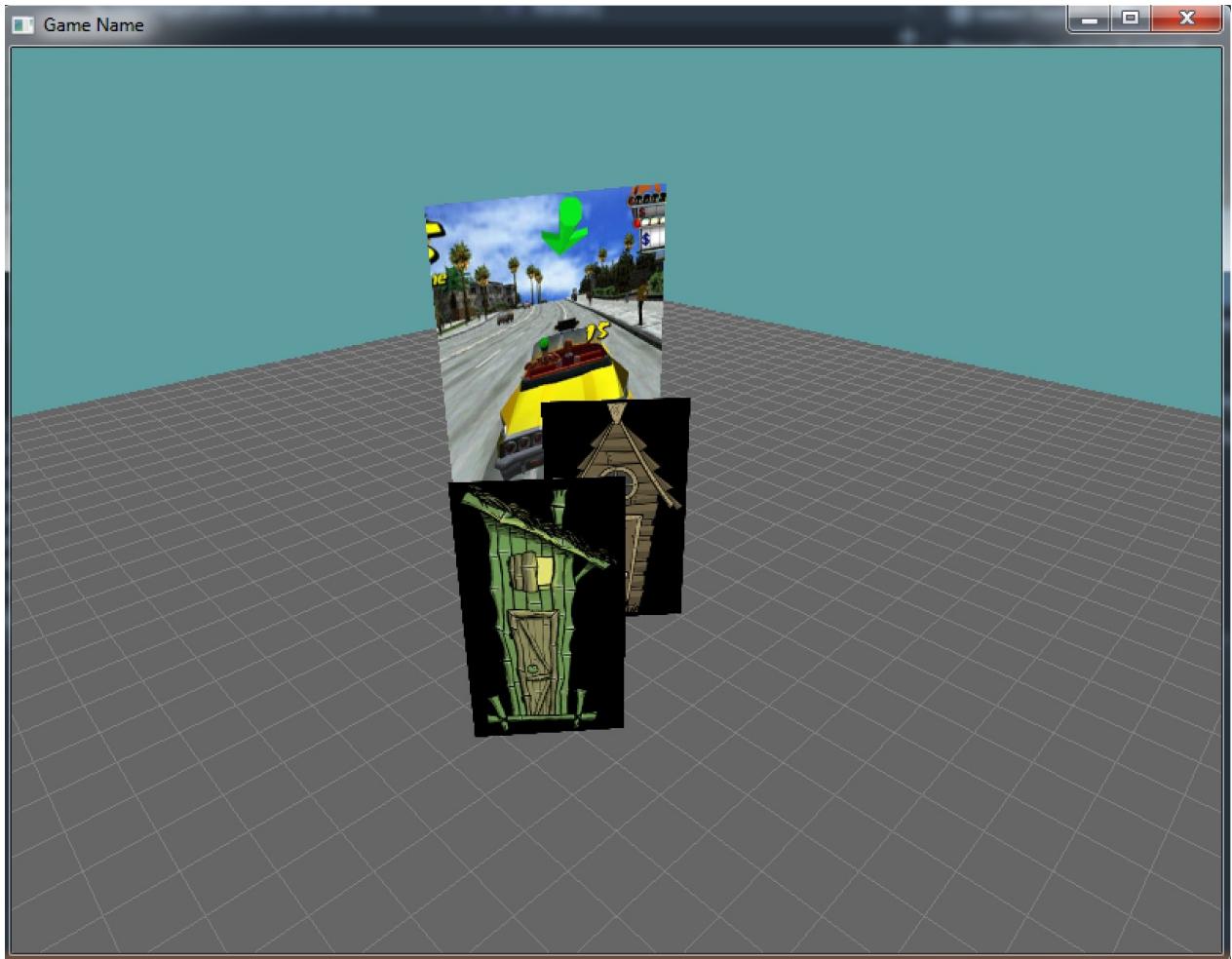


On Your Own

Now that we have some sample planes rendering, let's actually texture them. You are going to have to do this bit on your own. Just like above, don't make a helper function, write out all the code.

- First, create another class scope texture handle
 - Even tough we're texturing 2 objects, they are in an atlas
- In initialize, generate a new handle for this texture
- In initialize, bind the newly created handle
- In initialize, set min and mag filters
- In initialize, load the png file into this new texture handle
- In shutdown, delete the actual texture handle
 - Set the member variable to -1, to signify that it's invalid
- In Render, remove the enable and disable texturing around the two new quads
- Before rendering the quads, bind the houses texture
 - It's a good idea to bind 0 at the end of the render loop
- In render, add UV coordinates for each square
 - To display each house you will need to know where it's rectangle is and bring it into (normalized) uv space.

Once you have done all of that, running the game should look like this:

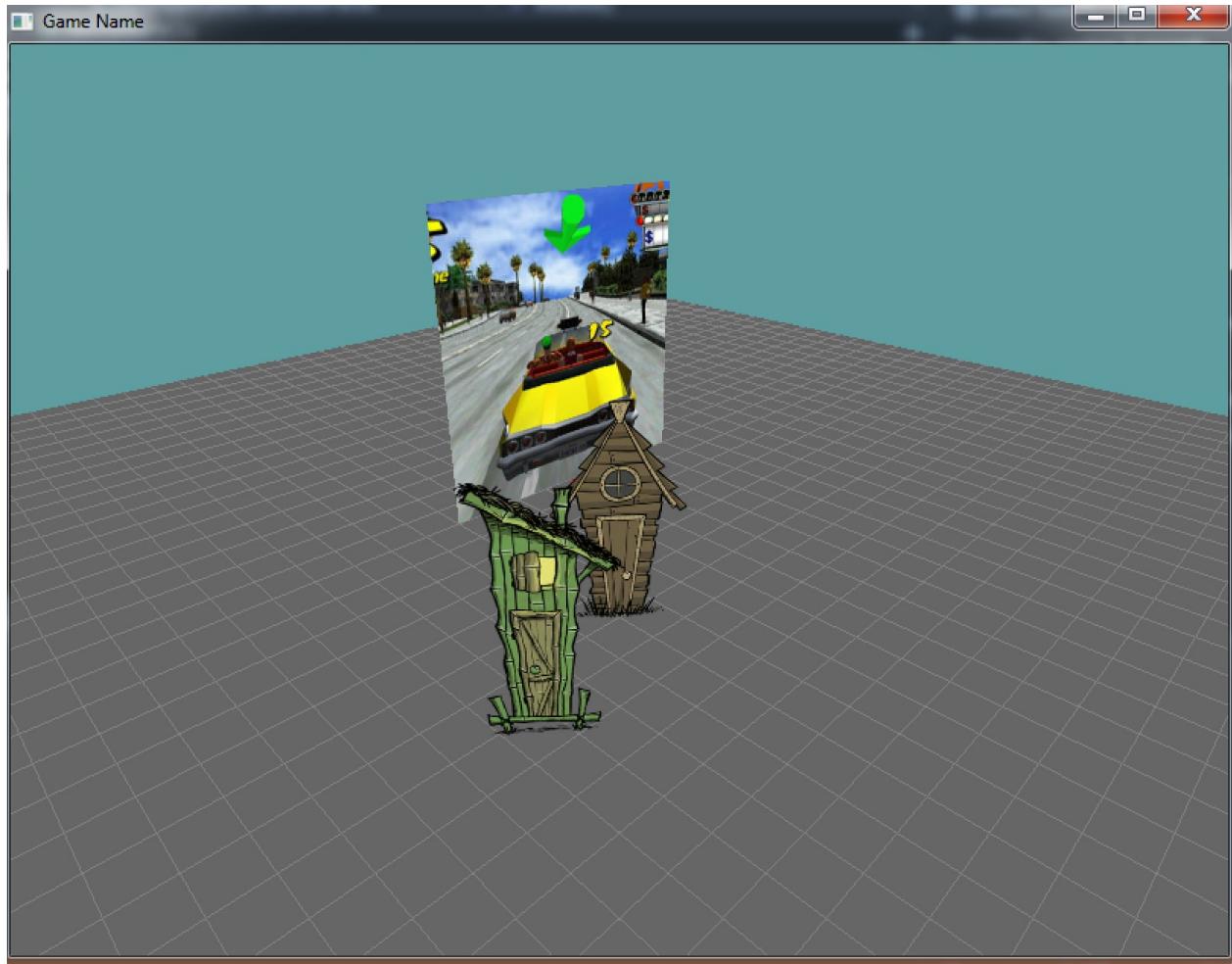


Well, that's cool. But it's not. Why is the alpha section of the image black? It has alpha pixels! Shouldn't it be transparent?!?!

Yes and no. The fact is, those are transparent pixels. But if you remember, transparency is not magic. It's not free. And it's not on by default. Let's fix this. Do the following

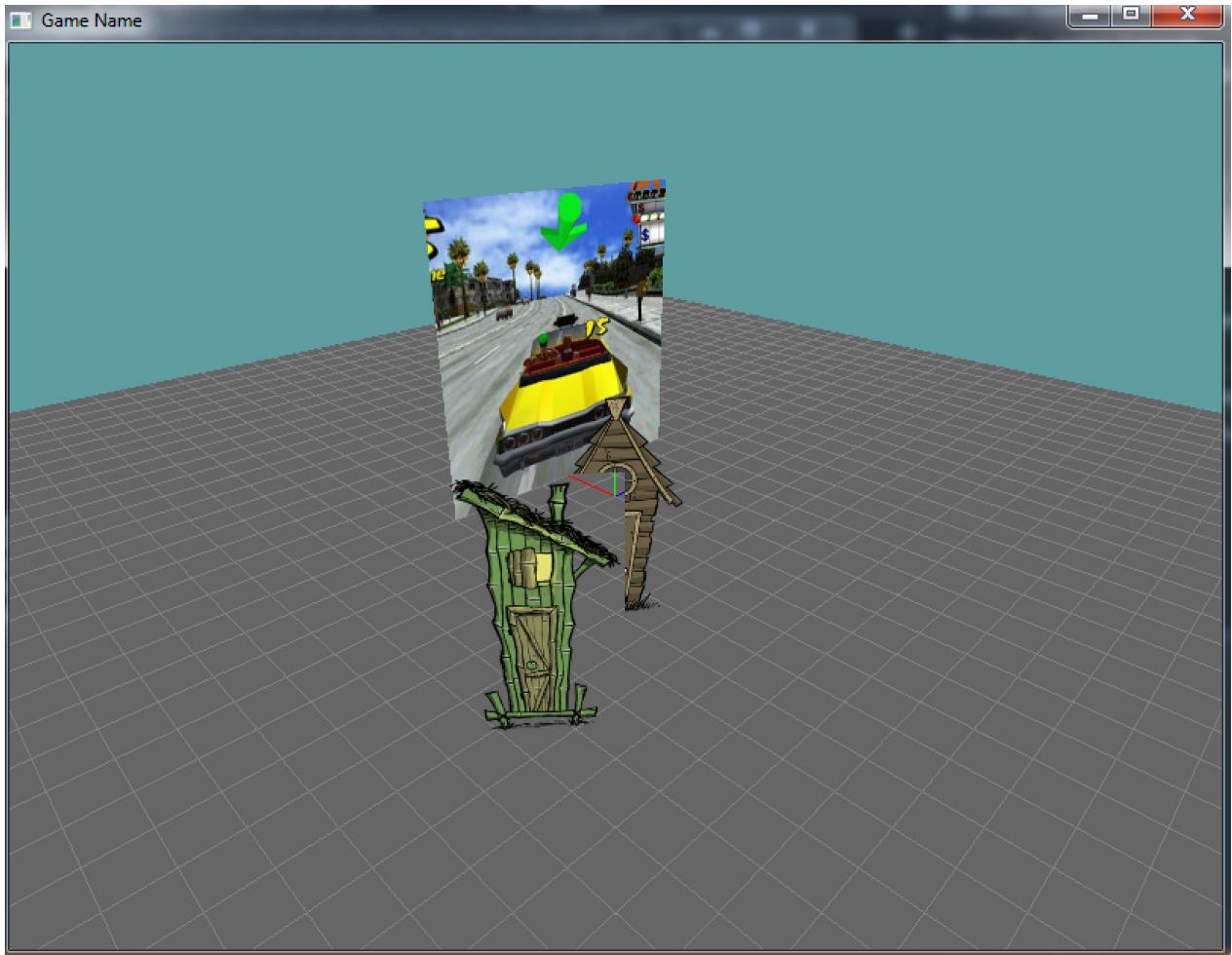
- In initialize enable blending
- In initialize set the blend function

Once you have done that, your scene will look like this:



One more thing you can try. To see how the order of the sprites effects transparency, try moving the code that renders house 2 above the code that renders house 1. Just to see the artifacts it causes. You need to know what it looks like when you have wrong z-order for alpha blending, and this is the perfect example to check out.

Here is what that would look like, but try it anyway:



This happens because the rendering goes like this now:

- First the grid renders
- Then crazy taxy renders
- Then the closest (green) building renders.
 - It's alpha pixels blend with whats on screen (grid & taxi)
- Finally the further (brown) building renders
 - It blends with whats on screen (grid, taxi, green building)
 - The green building is still rendered on a QUAD geometry
 - So a quad is written to the Z-Buffer
 - This quad has a closer Z than the new geometry,
 - New geometry which fails z-test and does not render in area

And this is why order matters when you are rendering alpha blended objects. You **ALWAYS** should be rendering the objects furthest away first, and then the closer objects.

Loading an OpenGL image

Loading an OpenGL image can be done in 5 lines of code.

```
Bitmap bmp = new Bitmap("SomeImageFile.png");
BitmapData bmp_data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height), ImageLockMode.ReadOnly, System.Drawing.Imaging.PixelFormat.Format32bppArgb);
GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0, OpenTK.Graphics.OpenGL.PixelFormat.Format32bppArgb);
bmp.UnlockBits(bmp_data);
bmp.Dispose();
```

Three of these 5 lines are just fillers. This is the only line that matters:

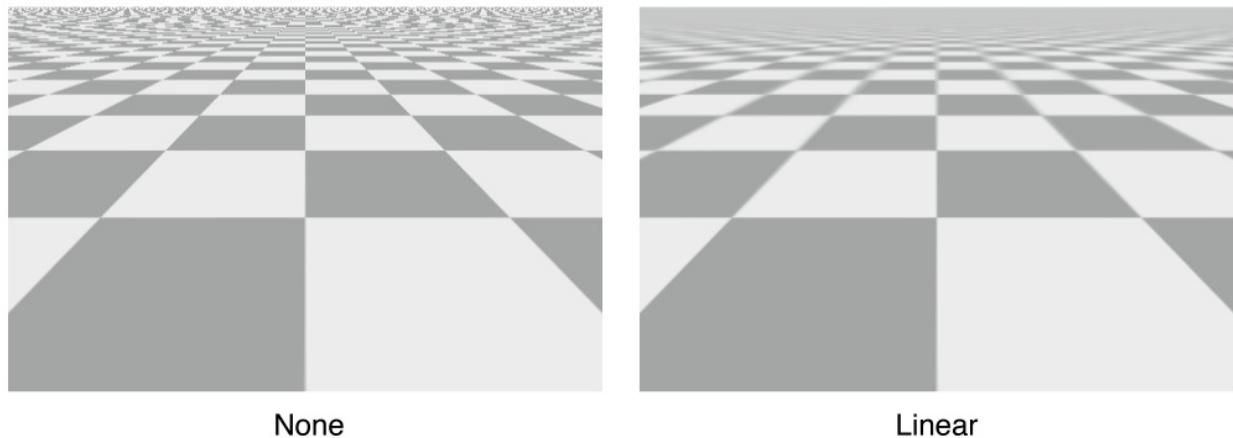
```
GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0, OpenTK.Graphics.OpenGL.PixelFormat.Format32bppArgb);
```

Make sure you understand the parameters of the `GL.TexImage2D` function, as it is what uploads the large array of pixel data to the GPU.

The loading code is mostly copied [from here](#)

Mip Mapping

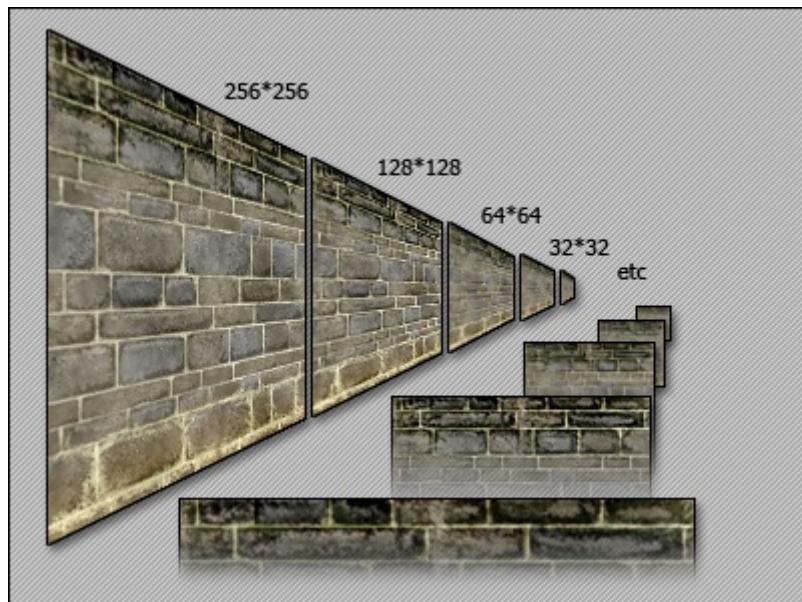
Mip mapping is a technique used to prevent texel swimming. Before we can talk about how mip mapping works, we need to discuss what texel swimming is, and why it's a problem. Consider the following image:



The left side shows a scene without mip mapping, the right side shows the same scene with linear mip mapping. Notice how even though the image is a straight grid, off in the distance a wavy distortion starts to happen. This phenomenon is called texel swimming.

Texel swimming happens because the further away a textured pixel is, the more super sampled it becomes. That is, the span of 3 pixels on the source image are compressed into perhaps a single pixel on screen. When applying a perspective distortion to an image, the swimming artifacts appear.

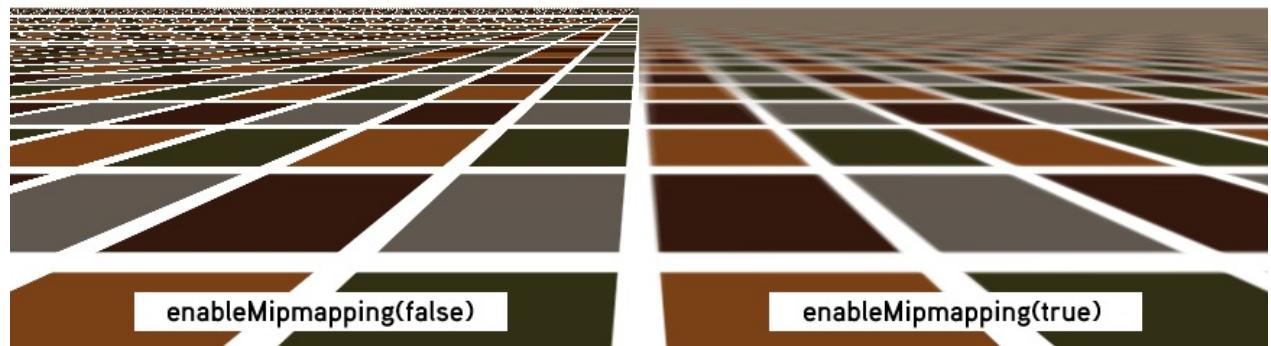
How does mip mapping solve this? By using several resolutions of an image in memory, and sampling the correct one depending on how far away screen pixel is. This way you are more likely to get a 1 to 1 texture mapping from source to screen and less likely to see texel swimming. Consider the following image:



The further away you view the wall segment, the smaller the version of the image you need to sample. That is, the closer a wall the more detail it needs (sample a 256x256 version of the wall texture) but the further away the wall is,

the less textures it is on screen. That means when the wall is far away a 32 x 32 version of the wall texture has enough detail.

Of course because you are sampling an image at different resolutions the resulting image will look blurred based on what resolution the specific image is sampled at. Take a look:

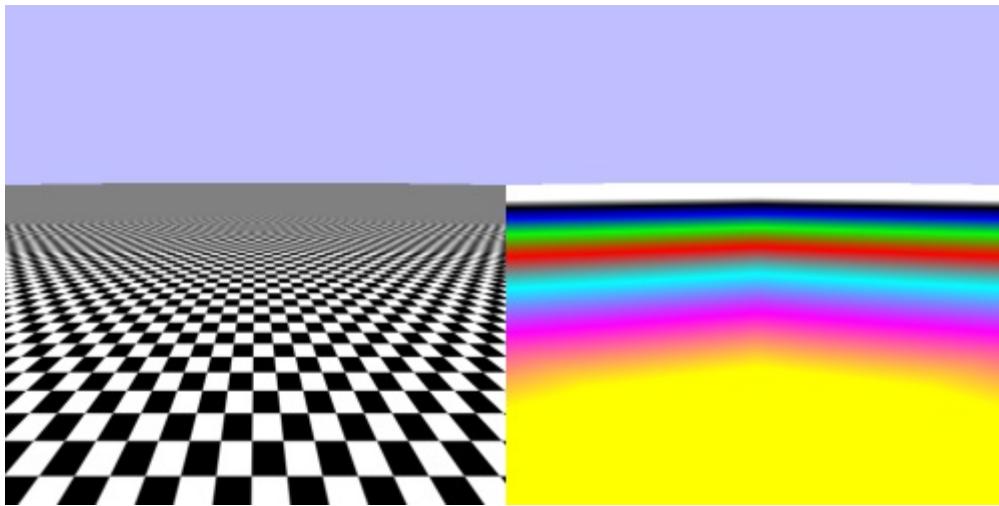


This may seem unreasonable, i mean doesn't blurring the image make the world look worse? In simple cases yes. But look at this example scene, it's not a complex scene, but adding mip-mapping makes a LOT of difference in the overall quality:

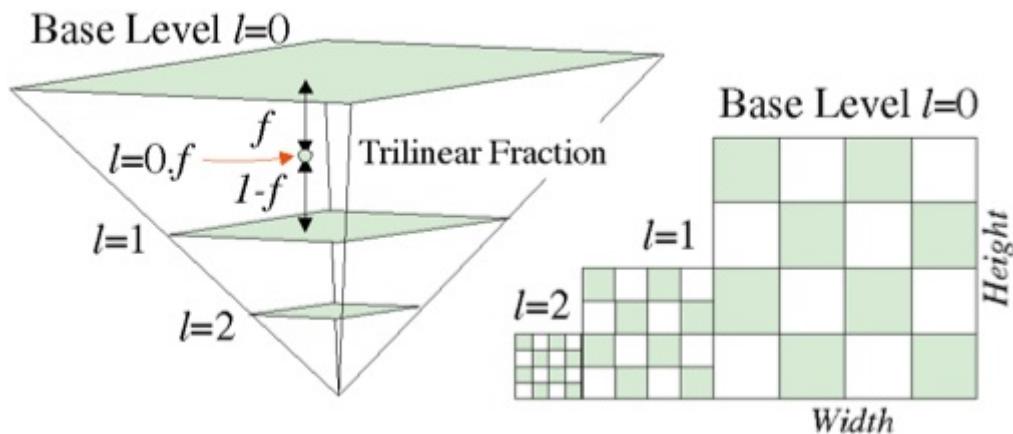


Texture pyramid

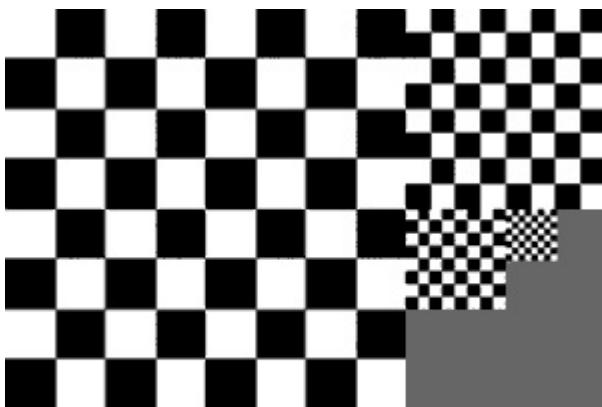
Like i said mip mapping is based on the pixel being rendered's distance to the camera. This image demonstrated the different mip levels being used



The way to visualize what pixel uses what mip is by thinking of the pixel as a texture pyramid:



The closer a pixel is to the camera, the closer it is the mip level 0. The further it is from the camera, the closer it is to a deeper mip level (like 2). So what does a mip mapped image look like in memory? Like this:



This simple MIP map holds four copies of a chequered texture, all at different scales

It's still a single texture, but that single texture is now larger, and holds multiple copies of the image at different scales. For example, if a base texture is 1024x1024, the mipmapped version of that texture will have a resolution of 1536x1024 and will contain resized versions of the image. ($1536 = 1024 + 512$)

When a mip map texture is generated, the image keeps being halved until it is only a single pixel. Each time the

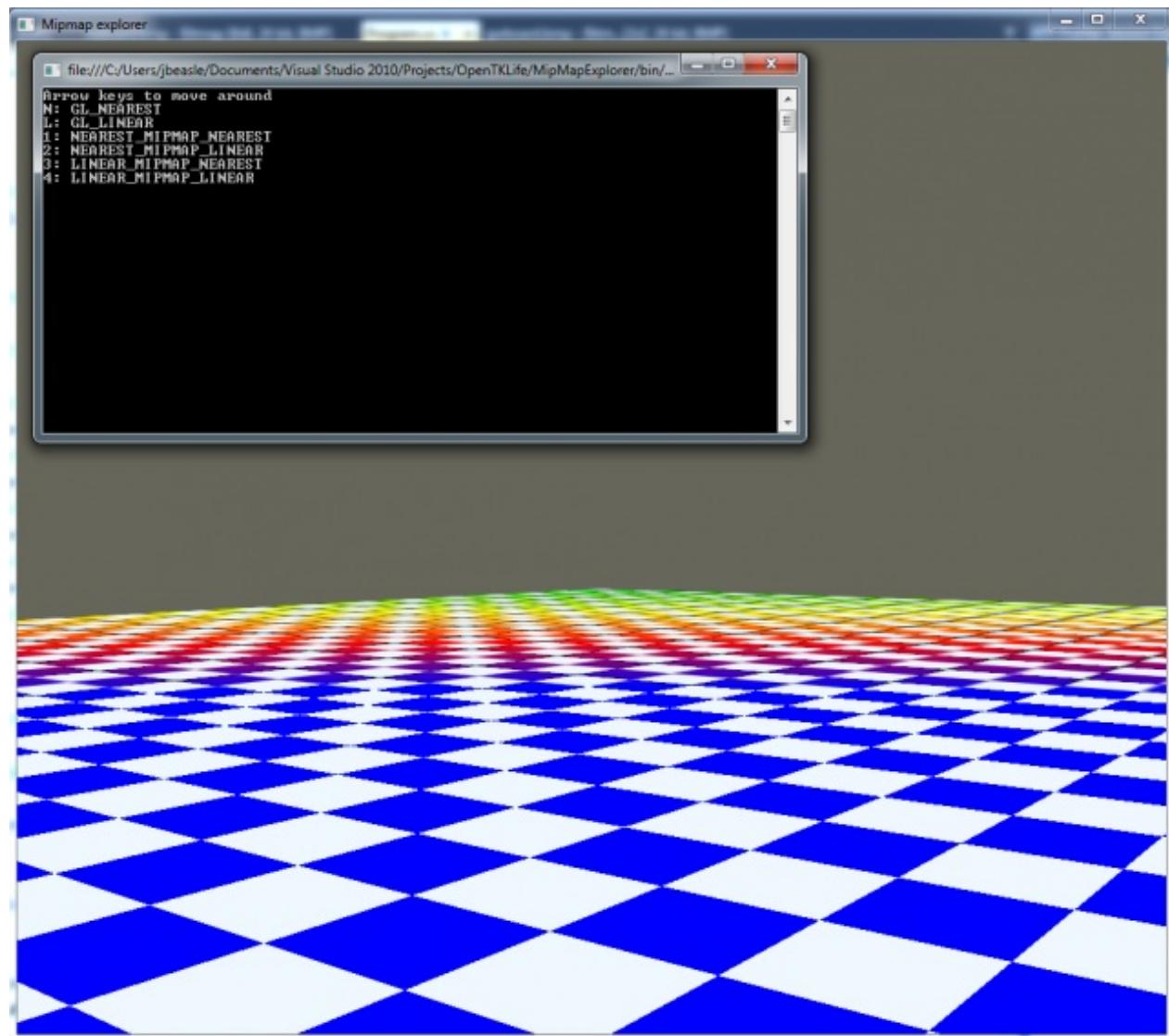
image is halved it's called a new mip level. The base image is level 0, the last level could be anything depending on the size of the original image.

Where do mip maps come from?

A natural next question to ask is where do mip maps come from? There are two possible sources for mip maps. Most of the time you will have OpenGL generate mip maps for you. But it's also possible to upload a texture that is pre-mipmapped.

Some graphics cards don't support uploading pre-mipmapped images as they store the mip format in a proprietary way. Because of this, to reach as many devices as possible, we tend to let OpenGL generate mipmap for us as the default.

The best argument for uploading pre-mipmapped images comes from [this OpenTK article](#). They use a different colored image for each mip-level to debug the application and make it look better. Their multi-image mip looks like this when rendered:



Lets see some code!

You don't have to follow along with this code, BUT when you do make a 3D world you will need it. As such, while you don't HAVE to follow along, you most certainly CAN. You can mip-map any scene that uses textures. The best place to start would be to add mip-maps to the scene from the [Putting it all together](#) section.

Mip mapping happens when a texture is being put through the min and mag filters. As such, the first step is to modify the min filter. Instead of using `TextureMinFilter.Linear` as the argument use `TextureMinFilter.LinearMipmapLinear`.

Take note, mip mapping is for min filters only, as such the `TextureMinFilter` enum contains this option, but the `TextureMagFilter` enum does not.

With all that being said, keep the mag filter on linear and configure the min filter like so:

```
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.LinearMipmapLinear);
```

Then, you simply call `GL.GenerateMipmap` and specify a texture target to generate the mipmap for. Like so:

```
GL.GenerateMipmap(GenerateMipmapTarget.Texture2D);
```

And that's all there is to it. Mip mapping is now enabled, and mip-maps are now generated. But just to be complete, let's take a look at loading a full texture with mip-mapping:

```
int textureHandle = GL.GenTexture();
GL.BindTexture(TextureTarget.Texture2D, textureHandle);

// Enable mip mapping here, with the min filter
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.LinearMipmapLinear);
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter);

Bitmap bmp = new Bitmap("SomeImageFile.png");
BitmapData bmp_data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height), ImageLockMode.ReadOnly, System.Drawing.Imaging.PixelFormat.Format32bppArgb);
GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0, OpenTK.Graphics.OpenGL.PixelFormat.Bgra);
// Generate mipmap here, after pixel data is uploaded to GPU
GL.GenerateMipmap(GenerateMipmapTarget.Texture2D);

bmp.UnlockBits(bmp_data);
bmp.Dispose();
```

There is a decent example of this code being used on the [Official OpenTK website](#)

Texture Parameters

You can tell OpenGL how to treat different properties of textures using the `GL.TexParameter` function. We've already used the function when telling OpenGL how to handle the min and mag filters for the textures being loaded. What other texture parameters can we control?

Texture Wrapping

There are a few parameters we can control using `GL.TexParameter`, but the only ones we care about are mip-mapping and texture wrapping. So what is texture wrapping?

We've seen what happens when you have a quad which is mapped within the 0 to 1 uv range, like so:

```
GL.Begin(PrimitiveType.Quads);
    GL.TexCoord2(0, 1);           // What part of the texture to draw
    GL.Vertex3(left, bottom, 0.0f); // Where on screen to draw it

    GL.TexCoord2(1, 1);           // What part of the texture to draw
    GL.Vertex3(right, bottom, 0.0f); // Where on screen to draw it

    GL.TexCoord2(1, 0);           // What part of the texture to draw
    GL.Vertex3(right, top, 0.0f); // Where on screen to draw it

    GL.TexCoord2(0, 0);           // What part of the texture to draw
    GL.Vertex3(left, top, 0.0f); // Where on screen to draw it
GL.End();
```

But what happens if you exceed that range? For instance, what if your texture coords don't go to one, but they go to 2... Or -7. Like this:

```
GL.Begin(PrimitiveType.Quads);
    GL.TexCoord2(0, 3);           // What part of the texture to draw
    GL.Vertex3(left, bottom, 0.0f); // Where on screen to draw it

    GL.TexCoord2(3, 3);           // What part of the texture to draw
    GL.Vertex3(right, bottom, 0.0f); // Where on screen to draw it

    GL.TexCoord2(3, 0);           // What part of the texture to draw
    GL.Vertex3(right, top, 0.0f); // Where on screen to draw it

    GL.TexCoord2(0, 0);           // What part of the texture to draw
    GL.Vertex3(left, top, 0.0f); // Where on screen to draw it
GL.End();
```

This is where texture wrapping comes into play. The texture wrapping parameter tells OpenGL how to handle this exact scenario. There are 4 possible values you could set wrapping to:

- **Repeat** Will tile the texture
- **MirroredRepeat** Will tile the texture, with each tile being flipped
- **ClampToEdge** Will use the edge pixel of a texture to determine the remaining colors
- **ClampToBorder** Will use a solid border color to fill the remaining pixels

This might sound a bit confusing at first, so let's take a look at how each option acts in a real world situation



GL_REPEAT



GL_MIRRORED_REPEAT



GL_CLAMP_TO_EDGE



GL_CLAMP_TO_BORDER

You can set different parameters for the X and Y axis. You specify each axis as `TextureParameterName.TextureWrapS` for the X and `TextureParameterName.TextureWrapT` for the Y. For example, if you wanted to set a texture to repeat on both axis, you would use this code:

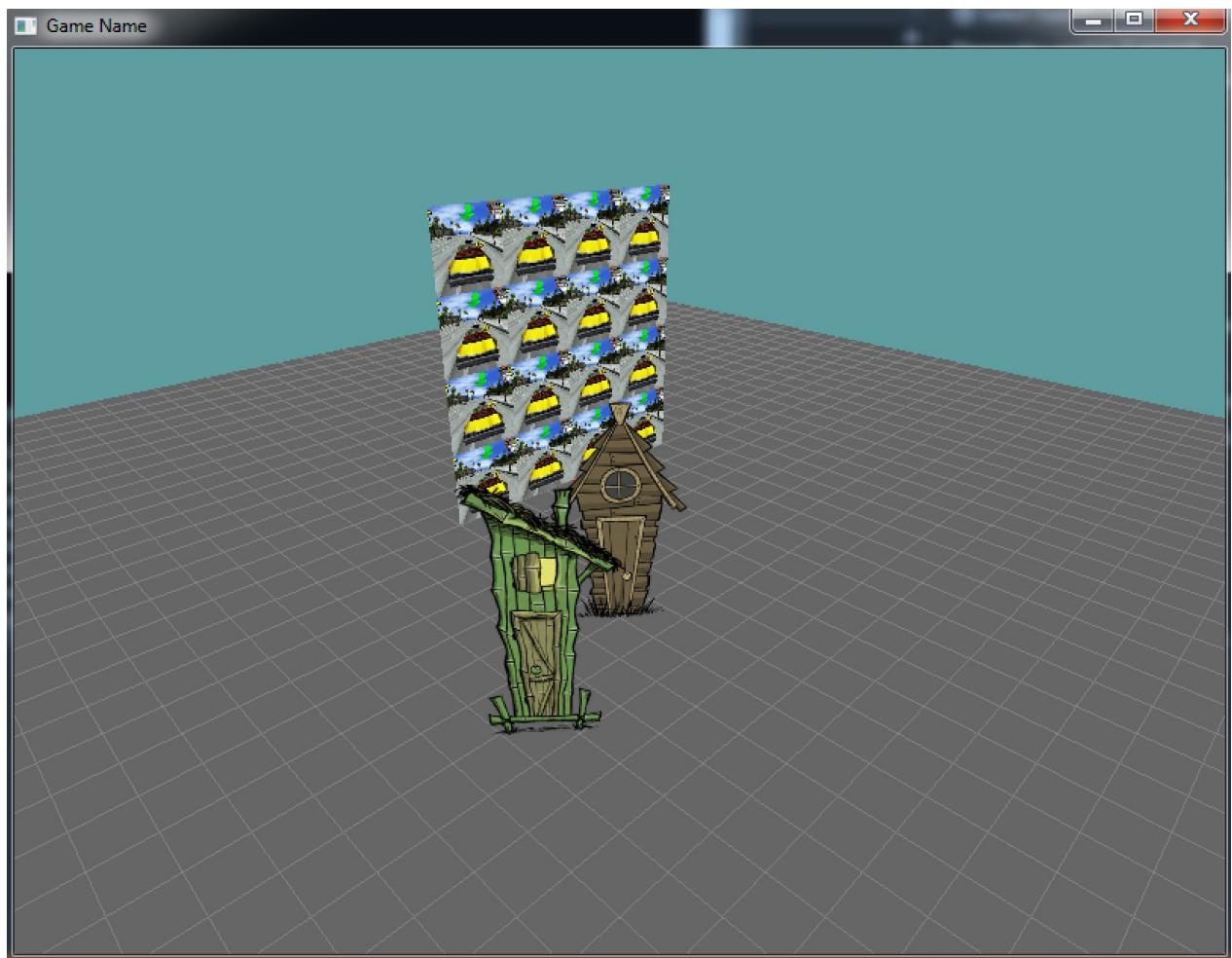
```
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapS, (int)TextureWrapMode.Repeat);
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapT, (int)TextureWrapMode.Repeat);
```

There is one more parameter you have to set if you use the `ClampToBorder` parameter, and that's the border color. You can set the border color like this:

```
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureBorderColor, new float[] { r, g, b, a});
```

On your own

Modify the sample scene we've been working with so that the crazy taxi image is uv'd from -2 to 2 instead of 0 to 1. Make sure the wrapping parameters on both axis are set to repeat. The final image should look like this:



Textures for UI

Texturing 3D objects is cool, but one of the most often used places for textures is UI. In this section i'm going to walk you trough how to add ui to any scene. Then, in the on your own section you will implement some UI on top of the 3D textured test scene we've been working with so far.

Multiple coordinates

UI relies on using multiple coordinates. After all, if you have a world with a perspective camera, you don't want your UI to be perspective. Unless some UI is in world space (Like the health bar of an RTS game) the UI should be in an orthographic projection.

We not only give the UI it's own projection matrix, we also give it it's own modelview matrix. This allows us to work easily in ui space. Generally, the code for UI will go something like this:

```
void Render() {
    // FIRST, We render the 3D scene!

    // Assume the matrix mode is ModelView
    // Load world modelview matrix for the 3D scene
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(-7.0f, 5.0f, -7.0f), new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
    GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

    // Render 3D scene as normal
    RenderWorld();

    // THEN, render the UI

    // Clear only the depth buffer. This way our UI will not z-test against
    // the 3D scene, because the depth buffer will be empty.
    GL.Clear(ClearBufferMask.DepthBufferBit);

    // Switch to projection matrix mode, backup 3D projection, load UI projection
    GL.MatrixMode(MatrixMode.Projection); // Switch
    GL.PushMatrix(); // Backup
    GL.LoadIdentity(); // Clear
    GL.Ortho(-1, 1, -1, 1, -1, 1); // Load UI Projection

    // Switch back to modelview mode, backup 3D modelview, clear
    GL.MatrixMode(MatrixMode.ModelView); // Switch
    GL.PushMatrix(); // Backup
    GL.LoadIdentity(); // Clear

    // Render the UI
    RenderUI();

    // Restore world (3D) projection
    GL.MatrixMode(MatrixMode.Projection());
    GL.Pop();

    // Restore world (3D) modelview
    GL.MatrixMode(MatrixMode.ModelView());
    GL.Pop();

    // We make sure the matrix mode is modelview for the next render iteration
}
```

That's a LOT of code, but it kind of shows you how rendering the UI is almost an entireley different render path altogether. It's like rendering an overlay. Perhaps the most important thing when rendering the UI is the call to clear

the depth buffer:

```
GL.Clear(ClearBufferMask.DepthBufferBit);
```

This call clears all Z-values that have been written into the depth buffer so far. By doing this, we ensure that our UI never clips against anything in world space. This should look familiar, a similar call is used in Program.cs to clear the depth and color buffers.

After the depth buffer is cleared, we back up the world space projection and reset it to UI space, next we back up the world space view matrix and replace it with the UI view matrix. And that's all there is to it, now we just render the UI.

I STRONGLY suggest breaking your scene into two render functions, `RenderWorld` and `RenderUI`, and calling them like I did above. This will keep the main render function `Render` of your scene from getting unmaintainable.

Positioning the UI

Right now positioning the UI is a pain in the ass. Because the screen goes from -1 to 1 we have to figure out how much space a UI element takes up, then normalize that into the screen. We might end up with UI code that looks like this:

```
GL.Begin(PrimitiveType.Quads);
GL.TexCoord2(0, 1);
GL.Vertex3(0.1345f, 0.3345f, 0.0f);

GL.TexCoord2(1, 1);
GL.Vertex3(0.435f, 0.3345f, 0.0f);

GL.TexCoord2(1, 0);
GL.Vertex3(0.435f, 0.1245f, 0.0f);

GL.TexCoord2(0, 0);
GL.Vertex3(0.1345f, 0.1245f, 0.0f);
GL.End();
```

And that is kind of awful! I mean, what if you need to change the screen position later, this becomes a nightmare to maintain. Worse yet, it does not account for aspect ratio. Worse, worse yet, because of the aspect ratio error compounded with possible floating point error it's nearly impossible to get a pixel perfect UI.

And that's the real issue here, sometimes we NEED for UI to be pixel perfect. Other times it doesn't matter, but if you are serious about making something 2D, you need to at least have the option of being pixel perfect.

Pixel Perfect

The key to a pixel perfect UI lies in the orthographic projection matrix. Right now we have the space map to NDC space, that is a cube ranging from -1 to 1, like so:

```
GL.Ortho(-1, 1, -1, 1, -10, 10); // Load UI Projection
```

That maps 1 unit to half the width of the screen. What if we didn't do this? Instead we could change our orthographic space to map 1 unit to 1 pixel? Can we even do this? Yes, yes we can, like so:

```

int screenWidth = MainGameWindow.Window.Width;
int screenHeight = MainGameWindow.Window.Height;
// Ortho: left, right, bottom, top, near, far
GL.Ortho(0, screenWidth, screenHeight, 0, -1, 1);

```

Now the left of the screen is at 0, the right is at `screenWidth` units. This makes our orthographic projection space map 1 to 1 with pixels, meaning if we want a quad to render in the top left, 10 pixels from the top, 15 pixels from the left 20 pixels wide and 30 pixels tall, we could just do this:

```

int left = 15; // Left is 15 pixels from edge of screen
int top = 10; // Top is 10 pixels from edge of screen
int right = 15 + 20; // 20 pixels wide, right side is left + 20
int bottom = 10 + 30; // 30 pixels tall, bottom is top + 30

GL.Begin(PrimitiveType.Quads);
    GL.TexCoord2(0, 1);
    GL.Vertex3(left, bottom, 0.0f);

    GL.TexCoord2(1, 1);
    GL.Vertex3(right, bottom, 0.0f);

    GL.TexCoord2(1, 0);
    GL.Vertex3(right, top, 0.0f);

    GL.TexCoord2(0, 0);
    GL.Vertex3(left, top, 0.0f);
GL.End();

```

And that's that. We just defined a texture to be drawn using pixel coordinates. All because the orthographic projection matrix mapped the projections NDC space 1 to 1 with our windows pixel space.

What if you wanted that same quad to be 15 pixels from the right of the screen instead of the left?

```

int quadWidth = 20;
int right = screenWidth - 15; // Place right side of quad 15 pixels from width of screen
int left = right - quadWidth; // Place left side of quad 20 pixels from right side of quad

// Top and bottom like normal
// Draw like normal

```

The point is, once you are in pixel space, you can figure out how to anchor UI to different parts of a window

Utility

I suggest writing a utility function. Something along the lines of

```

void DrawTexture(int texId, Rect screenRect, Rect sourceRect, Size sourceImageSize) {
    // TODO, figure out how to render quad to screen
}

```

This way, you won't have to write the above verbose `GL.Begin / GL.End` every time you want to render a ui quad. Also, it will make your life super easy if your projection is pixel perfect.

Try to write the function on your own, maybe load some UI texture to test it with. If you want you can send it to me for review before doing the "On Your Own" section. The `Primitives` class might be a good place for this function to live.

Implementing a simple UI

We're going to modify the demo scene we've been working with to have a simple UI. I suggest reading this whole page over at least one time before implementing any code, just so you have an idea of everything that will be happening.

If you want to write or use some helper functions for this example, that's fine. By the time we're done, the demo scene will look like this:



First, download the following texture into your assets directory:



Health: X: 2, Y: 2, W: 421, H: 87
Home: X: 16, Y: 104, W: 92, H: 92
Help: X: 120, Y: 104, W: 92, H: 92
FB: X: 230, Y: 102, W: 92, H: 92

The first thing you want to do is make a `RenderUI` function. You can leave it blank for now, just have it stubbed in. This is where the code to render your UI will go.

- Add a new integer variable to the class to hold the UI
- In `initialize`, load the UI texture
 - Remember to unload it in shutdown
- Refactor your render function into a `RenderWorld` function
 - That is, render should set the modelview matrix
 - Then call `RenderWorld`
- After the world is rendered:
 - Set a pixel perfect Orthographic projection matrix
 - Remember, back up the old one
 - Load identity for the view matrix
 - Remember, back up the old one
 - Call your `RenderUI` function
 - Restore the projection and view matrices

Running your game at this point, everything should render as it did before. It's a really good place to check and make sure nothing is broken yet. After you've confirmed that nothing is broken, it's time to start working on the actual UI.

- First, clear your depth buffer!
- Make sure the right texture is bound
- Render the health-bar on screen.

- Screen coordinates (in pixels):
 - X: 10
 - Y: 10
 - W: 210
 - H: 43
- UV Coordinates (in pixels):
 - X: 2
 - Y: 2
 - W: 421
 - H: 87
- You can either figure out the normalized UV position by hand
 - Or normalize it in code, all you need is the texture width / height
- The 3 buttons need to be rendered independently,
 - All 3 buttons should be **Relative** to the bottom right corner
 - When the window is resized, these stay the same distance from the corner
 - The spacing between buttons is 10 pixels
- Render facebook button on screen
 - UV Coordinates: *Written on texture*
 - Screen Coordinates:
 - Bottom: 10 pixels above the window bottom
 - Top: The height of the button above "Bottom"
 - Right: 10 pixels from the right of the window
 - Left: the width of the button to the right of "Right"
- Render help button on screen
 - UV Coordinates: *Written on texture*
 - Screen Coordinates:
 - Bottom: 10 pixels above the window bottom
 - Top: The height of the button above "Bottom"
 - Right: 10 pixels from the right of facebook buttons "Left"
 - Left: the width of the button to the right of "Right"
- Render home button on screen
 - UV Coordinates: *Written on texture*
 - Screen Coordinates:
 - Bottom: 10 pixels above the window bottom
 - Top: The height of the button above "Bottom"
 - Right: 10 pixels from the right of help buttons "Left"
 - Left: the width of the button to the right of "Right"

Hint, not seeing anything on screen? Try `GL.Disable(EnableCap.CullFace)` to see if your UI rect's are being culled improperly. Keeping cull face disabled is not an option, but it's a debug step towards seeing what is broken.

When resizing the screen, the health bar is relative to the top left. This is why you could just do screen coordinates. But when you render the buttons in the bottom right, they need to move with the screen size. So you need to adjust their X-Y coordinates accordingly.

For example:

```
int buttonSpace = 10;
int homeButtonX = screen.Width - buttonSpace - home.Width;
int helpButtonX = homeButtonX - buttonSpace - help.Width;
int fbButtonX = helpButtonX - buttonSpace - fb.Width;
```


Sorting

So far we've been writing all of this rendering code inline, that is whenever we needed to render anything, we just do! The only thing we really wrapped into an object has been the grid. And maybe a "Draw Texture" function.

Rendering a 3D game will get complex. A 3D mixes and matches solid and transparent objects sometimes. In order to draw transparent objects, you must first draw solid objects, then draw transparent objects. Transparent objects need to be sorted based on who is furthest from the camera.

I'll walk you through how it would normally work. This is not something you need to implement right now, but it is something to be aware of, as you might need to do this at some point.

Transparent objects

When reading this remember, you can render solid objects in any order. The Z-Buffer will make sure that objects get rendered correctly.

Game Objects only need to be sorted when they are rendered with transparency! And even then, it's advised to render in two passes. First, render the solid objects, next render the transparent ones.

The Component (and render component)

We're going to use a fairly simple component base class

```
class Component {
    GameObject owner;

    public virtual void Render() { }
    public virtual void Update(float deltaTime) { }
}
```

And the render component is going to be pretty simple too. It will however need to do some logic to see if a model is textured, or has normals.

```
class MeshRenderer : Component {
    public int textureHandle;
    public bool UsingAlpha; // Set if object is transparent

    protected List<Vector3> vertices;
    protected List<Vector3> normals;
    protected List<Vector2> uvs;

    public override void Render() {
        // Enable texturing if we use it
        if (textureHandle != -1) {
            GL.Enable(EnableCaps.Texture2D);
        }

        GL.Begin(PrimitiveType.Triangles);
        for (int i = 0; i < vertices.Count; ++i) {
            if (normals != null) {
                GL.Normal3(normals[i].x, normals[i].y, normals[i].z);
            }

            if (uvs != null && textureHandle != -1) {
```

```

        GL.TexCoord2(uvs[i].x, uvs[i].y);
    }

    GL.Vertex3(vertices[i].x, vertices[i].y, vertices[i].z);
}
GL.End();

// Disable texturing if it was enabled
if (textureHandle != -1) {
    GL.Disable(EnableCaps.Texture2D);
}
}

}

}

```

The Game Object

For this example, let's assume we have a super simple 3D game object class. The class is going to be super minimal for this example, it's going to have a list of components, a list of children, a potential parent and a 3D transform (a matrix).

```

class GameObject {
    public string Name;
    public List<Component> Components;
    public GameObject Parent;
    public List<GameObject> Children;
    public Matrix4 LocalTransform;

    public Matrix4 WorldTransform {
        get {
            // The order of this multiplication might be wrong
            return LocalTransform * Parent.LocalTransform;
        }
    }

    public void Update(float deltaTime) {
        foreach (Component component in Components) {
            component.Update(deltaTime);
        }
        foreach(GameObject child in Children) {
            child.Update(deltaTime);
        }
    }

    public void RenderSolid() {
        foreach (Component component in Components) {
            if (component is MeshRenderer) {
                MeshRenderer renderer = component as MeshRenderer;
                if (!renderer.UsingAlpha) {
                    // Backup view matrix
                    GL.PushMatrix();

                    // Apply game object transform
                    GL.MulMatrix(Matrix4.Transpose(WorldTransform).Matrix);

                    // Render the object
                    component.Render();

                    // Restore view matrix
                    GL.PopMatrix();
                }
            }
            foreach(GameObject child in Children) {
                child.RenderSolid();
            }
        }
    }
}

```

The Scene

The scene class, is going to for the most part be what we are used to, a root game object, that in turn has many children. The update function is actually going to be recursive, like we are used to. One thing that's different in this scene is it's going to have a view matrix defined. This is essentially the camera.

Remember, you can get the world position of the camera (the viewer) by taking the inverse of the view matrix. And you can get the view matrix by taking the inverse of the world position of the camera matrix.

```
class Scene {
    public string Name;
    public GameObject Root;
    public Matrix View;

    public void Update(float deltaTime) {
        if (Root != null) {
            Root.Update(deltaTime);
        }
    }

    public void Render() {
        // load the view matrix
        GL.LoadMatrix(Matrix4.Transpose(View).Matrix);

        if (Root != null) {
            // Each object will load it's own model matrix
            Root.RenderSolid();
        }
    }
}
```

This will render all solid objects in the scene. Now, let's see what we need to do to render transparent objects!

Rendering transparent objects

In order to render transparent objects, we have to do a 2 step process. First, we need to collect all transparent object. Then, we need to sort them based on distance to camera. I'm going to make a new class (a protected helper of scene) and create a method to collect all transparent objects

```
class Scene {
    protected class RenderCommand {
        MeshRenderer component;
        Matrix worldTransform;
        float DistanceToCamera;

        public void Execute() {
            // Backup view matrix
            GL.PushMatrix();

            // Apply game object transform
            GL.MulMatrix(Matrix4.Transpose(worldTransform).Matrix);
            // Render component
            component.Render();

            // Restore view matrix
            GL.PopMatrix();
        }
    }

    public string Name;
    public GameObject Root;
    public Matrix View;
```

```

public void Update(float deltaTime) {
    if (Root != null) {
        Root.Update(deltaTime);
    }
}

public void Render() {
    // load the view matrix
    GL.LoadMatrix(Matrix4.Transpose(View).Matrix);

    if (Root != null) {
        // Each object will load it's own model matrix
        Root.RenderSolid();
    }

    // Collect transparent objects
    List<RenderCommand> transparentObjects = CollectTransparent(Root);

    // Sort the transparent objects back to front
    SortList(transparentObjects);

    // Finally, render all transparent objects, back to front
    foreach (RenderCommand command in transparentObjects) {
        command.Execute();
    }
}

void SortList(List<RenderCommand> list) {
    // Implement bubble sort,
    // sort the list using the DistanceToCamera field
    // of each RenderCommand
}

List<RenderCommand> CollectTransparent(GameObject object) {
    List<RenderCommand> result = new List<RenderCommand>();

    foreach(Component component in object.Components) {
        if (component is MeshRenderer) {
            MeshRenderer renderer = component as MeshRenderer;

            if (renderer.UsingAlpha) {
                RenderCommand command = new RenderCommand();
                command.component = renderer;
                command.worldTransform = object.WorldTransform;

                // Find the distance to camera. We do this by taking two points at 0, 0, 0
                // Next transform one point to where the camera is
                // and the other point to where the game obejct is
                // With the transformed vectors, check their difference

                Vector3 cam = Matrix4.Inverse(View) * new Vector3(0, 0, 0);
                Vector3 obj = object.WorldTransform * Vector3(0, 0, 0);
                command.DistanceToCamera = Vector3.Length(cam - obj);

                result.Add(command);
            }
        }
    }

    foreach(GameObject child in object) {
        List<RenderCommand> childRenderers = CollectTransparent(child);
        if (childRenderers != null && childRenderers.Count > 0) {
            result.AddRange(childRenderers);
        }
    }
}

return result;
}
}

```

It's a lot of code, but our renderer finally has a list of transparent objects. Instead of storing GameObjects i made a

helper class called RenderCommand. We could have just stored objects, then found the render component on those objects later when it's time to draw them, but this is slightly more efficient.

Render commands are also pretty standard in 3D games. Usually even solid objects get grouped into a set of render commands, they are not sorted; but commands only get created for objects that the camera can see. We will talk about this more in detail later.

Once we have a list of render commands, you need to sort them based on distance to camera. You can use whatever sort you like. Then, in the correct back to front order, render all the game objects.

Particles

Everything is better with particles! Case and point, watch [this video](#).

Explosions, sparks, underwater bubbles, other special fx, particle systems in modern games are used to add eye-candy to a game. Usually individual particles have no collision reaction, there are just too many of them. But some sort of generic sphere collider could encapsulate the entire particle system.

Particles in effect are a number of entities that behave according to some pre-defined rules. Take rain for example. Each rain drop is a particle, all drops behave according to gravity. That is, a cloud spawns rain particles, they fall down with gravity. When a rain drop entity hits the ground it dies (is recycled as a different drop).

A more formal definition: *A particle system is a collection of any number of entities, either related or unrelated, that behave according to a set of logical rules, (a system)*.

See, the concept of a particle system is rather abstract. Any number of things will fit into that description, the roads of GTA, the speckles of a screensaver, rain... This is what makes particles so hard for most programmers. Two games can both have particle systems, and the implementation can be completely different.

When it comes to particles all we care about is what the final image looks like. Do these entities acting under some system look the way we expect / want? If so, the particle system works. If not, it just needs some additional work.

"We don't try to model the outcome of the system, we try to model the system and see its outcome" - Andre LaMothe

Design

Let's design a particle system. In general there are two ways to go about this. You can either design a base "ParticleSystem" class and then extend it into specific sub classes such as "FireParticles" or "SnowParticles" OR you can make a single mega particle system that exposes all sorts of properties such as absolute and relative velocity, then tweek these until your system is the way you want it.

As always, there is no right or wrong answer, what you use depends on what your needs are. Unity for example goes with the second route, the mega particle system. This is because they want the particles to be editable easily from the editor without code.

We on the other hand are not going to be shy about getting our hands dirty. We are going to design a particle system using the first approach. Where each new effect will take a little bit of code to create.

Lets take a bit of time and talk about things a particle system might need. Not everything we talk about on this page will make it into our implementation. Some of the stuff will be just for the purpose of discussion.

Particles

Lets start at the most basic element, the individual particle. To begin, you ned to decide what attributes a particle is going to need. Possible attributes might include:

- Position
- Velocity
- Life Span
- Size
- Weight
- Representation
- Color
- Owner

There are of course many other attributes you may want to try out, but these are the basics. When designing a particle system it's important to know to assign attributes that affect individual particles to the particle class, but attributes that effect all particles to the particle system class. Let's take a look at each of the above attributes in a little bit more detail.

Position

You need to know where the particle is in 3D space so that you can render it correctly. This is an attribute that will almost certainly belong to the particle, not the particle system. You may also want to track the particles last position to achieve effects such as trails. Note that the particles position will be affected by velocity.

Velocity

Your particles are probably going to be moving, so you need to store their velocity. It's most convenient to store this as a vector representing both speed and direction so you can use the vector to update the particles position.

Velocity will likely be affected by such factors as wind and gravity which we'll discuss later in this chapter, in the "Forces" subsection under "Particle Systems". If the particle is capable of accelerating itself, that could affect the velocity as well, and you'd want to create an additional vector to store the acceleration. More often tough, the factors that affect the velocity of a particle are external.

Life Span

For most effects, particles are going to be emitted from their source, and after some period of time are going to disappear. For this reason, you need to either keep track of how long a particle has been alive, or how long it has left to live. The life span may affect other attributes as a particle might grow, shrink or fade over time.

Size

Size in an attribute that may not need to be handled by individual particles. In fact, unless the size changes based on something, it's a useless attribute! However you might want to increase the size of particles, for example to represent a large fire, then decrease them as the fire dwindles.

Weight

Weight is a lot like size in terms of whether it's needed as an attribute or not. If a particle might accelerate differently

based on weight, or if it is affected by gravity differently you should include weight. Otherwise it's useless.

Representation

To have particles produce some kind of effect, you're going to have to see them. The real question is, how are you going to represent them on screen? There are three commonly used methods, though you may find other ways to represent them depending on your needs. The most common representations are:

- **Points** Points can be used for a number of effects, especially those that aren't viewed closely. Each particle might be a 3D point.
- **Lines** These can be used to create a trailing effect, which is often useful. The line connects the particles current position and its last one. Often used for rain
- **Textured Quads** This offers the most flexibility, and is the most widely used method. The particles itself is a quad or a triangle, and has a (possibly alpha blended) texture on it. Fire, sparks, smoke all use this method.

Note, if you decide to use quads, it's a good idea to use billboarding to make them always face the camera. Billboarding will be covered two chapters from now, under the "Advanced Topics" section.

Color

If you choose the point or line representation discussed in the preceding section you will probably want each particle to have a color. Even if you are using textured quads, a color tint might be a good idea. Color may change over time.

Owner

Sometimes particles benefit from knowing what particle system owns them. That way they can call methods in their owner system.

Particle Systems

Particles are owned by a particle system. Each particle system will control a set of particles, each of which acts autonomously, but share some common attributes. It's the job of the particle system to assign these attributes in such a way that, collectively, the particles create a desired effect. Some of the things a particle system might handle include:

- Particle list
- Position
- Emission Rate
- Forces
- Default particle attributes and ranges
- Current state
- Blending
- Representation

Just like with the discussion of particles, not everything we discuss about particle systems here will be implemented in our demo. And when you design your own particle systems, you might opt to include more attributes than the ones listed here. With that being said, let's go through each attribute:

Particle List

First and foremost, the system needs to be able to access the particles it is managing. So it needs a list of them, often stored as an array. The particle system should also have some maximum number of particles that it is allowed to create, effectively defining the size of the array.

Position

The particle system must be located somewhere to determine where particles start. Although this is usually modelled as a single point in space (For example, a campfire), but they don't have to be. You could represent the position of a system as a rectangle, so particles can be emitted randomly within that space (For example, rain), which would make the rectangle the part of the sky that is raining.

Emission Rate

The emission rate determines how often a new particle is created. To maintain a regular emission rate, the system will also need to keep track of how long it's been since the last particle was emitted.

Forces

Adding one or more forces to a particle system can create a greater degree of realism, or lack thereof. Forces can range anywhere from gravity, gravity wells to exploding forces and torque.

Default particle attributes and ranges

When a system creates a new particle, it should do so with some default attributes. For certain systems, having each particle spawn the same size will work perfectly. Other systems might need a default range. That is any particle might spawn between a range of 2 and 4 in terms of size. Whether your system uses default attributes or ranges depends on the effect you want to create.

Current State

You might want your particle system to change its behaviour over time; this may simply involve turning a system on or off. For example, a snow particle system should be off during summer. This is essentially enabling / disabling the system.

Sometimes you want to manually turn the system on or off, but other times you might want to have the system turn off on its own after some time. An explosion would be a great example of this.

Blending

Most particle systems use some sort of blending. For this reason you might want to include source and destination blend parameters as attributes of the particle system. Then again, most systems will make due with the default alpha blending, in which case this attribute is useless.

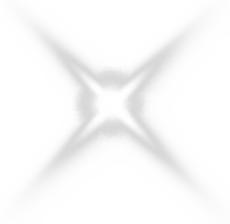
Representation

In most cases all particles within a system are going to be represented the same way. Because of this you might want to store a particles representation as a system attribute, instead of as an individual particle attribute.

Implementation

Now that we have talked a bit about how particle systems work, let's go ahead and actually implement one. I'll get you started by walking you through the base classes needed to create particles. After the base classes are made I'll also walk you through how to extend the system to create a simple snowing particle effect.

We're going to use this snow texture:



The final scene we will have at the end of the implementation will look like this:



Notice that the snow falls above the world, but below the UI. This is because of where we will position the snow's draw call.

Particle Class

Let's start by making a particle class. We're going to include a small number of attributes in this class. We're also going to include two unused attributes. The idea behind the unused attributes is to demonstrate that not all systems use the same attributes.

For example, if we have a snow particle system and a fire particle system they will both utilize the same particle class. The fire system might use attributes that the snow system doesn't and visa versa. If there are extra, unused variables in this class, it's not a problem.

The other thing to note about this class is that it has no methods. It's a class strictly to hold data. I've decided that my particle systems are going to know how to render each particle, so I gave the responsibility to render and update the particles to the system. This makes the particle class super simple.

```
using Math_Implementation;

namespace GameApplication {
    class Particle {
        // These are used in our demo
        public Vector3 position = new Vector3();
        public Vector3 velocity = new Vector3();
        public float size = 0f;

        // Unused in our demo, might be nice for other systems
        public float[] color = new float[4];
    }
}
```

That's all there is to it. The **position** variable is used to determine where to render the particle. The **velocity** variable determines how the particle will travel. The **size** variable sets the size of the particle.

The **color** variable is not going to be used in our demo, but other systems might need it. You should add attributes to the particle class every time a specific system needs them!

Particle System

The `ParticleSystem` class is a base class. It will need to be subclassed for each type of particle effect that your game might need. I included a minimum number of common attributes that you might need, more might be added as necessary.

We start off by declaring some member variables. We're going to need a list to store all the particles in the system. We're going to need to know the maximum number of particles the system will be able to hold as well as how many are currently alive. It also helps to know where in 3D space the system is. Because of how the particles are going to be spawned, let's keep track of how long the system has been alive. And finally, I'll include a convenience variable for random numbers.

```
using System;
using System.Collections.Generic;
using Math_Implementation;

namespace GameApplication {
    class ParticleSystem {
        protected List<Particle> particleList;
        protected int maxParticles;
        protected int numParticles;
        protected Vector3 systemOrigin;
        protected float accumulatedTime;
        protected Random random;
```

Next, let's make the constructor. It takes two arguments and sets all the class member variables accordingly. This constructor will also allocate all the particles we need.

```
public ParticleSystem(int maxParticles, Vector3 origin) {
    particleList = new List<Particle>();
    this.maxParticles = maxParticles;
    systemOrigin = new Vector3(origin.X, origin.Y, origin.Z);
    numParticles = 0;
    accumulatedTime = 0f;
    random = new Random();

    for (int i = 0; i < maxParticles; ++i) {
        particleList.Add(new Particle());
    }
}
```

Next, let's define some strictly virtual functions. Because each particle effect might behave differently, they will all need to override these functions. There is the standard update and render functions. The Shutdown function should be used in case the constructor loads a texture or something.

The interesting function here is `InitParticle`. This function is responsible for taking an unused particle and initializing it with some semi random values. This will of course need to be overridden.

Because all the particles we have are stored in a list, the `InitParticle` function just needs an index into that list. It will initialize the appropriate variables.

```
public virtual void Update(float deltaTime) { }
public virtual void Render() { }
public virtual void Shutdown() { }
```

```
public virtual void InitParticle(int index) {
    Console.WriteLine("ParticleSystem.InitParticle should never be called directly");
    Console.WriteLine("Only subclasses should call this function!");
}
```

We're going to use a function called `Emit` to actually spawn particles. When you call `Emit` you pass in a number, the number of particles that you would like spawned. `Emit` will return an integer, representing how many particles it failed to spawn. With any luck the return number should be 0.

Particles might or might not start out with some randomized values. For this reason when a particle is created we call the `InitParticle` function on it. That function will ensure that the particle starts out right.

```
public int Emit(int request) {
    // If we can make particles, make them!
    while (request > 0 && (numParticles < maxParticles)) {
        // Initialize a particle, and increase the number of particles
        InitParticle(numParticles++);
        // Decrease particles left to spawn
        --request;
    }

    // Return how many particles we could not create. Ideally, should be 0
    return request;
}
```

That's all there is to the particle system. It's a base class, so a lot of the effect related heavy lifting is going to be done by child classes. Speaking of which, let's go ahead and create an example of a snow particle system in the next section!

Snow particle example

Lets start the class out with some constants. There are two parts to each constant. The default value that the constant will have, along with some random range that affects that value.

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using System.Drawing;
using System.Drawing.Imaging;

namespace GameApplication {
    class SnowstormParticleSystem : ParticleSystem {
        public static Vector3 SNOWFLAKE_VELOCITY = new Vector3(0f, -2f, 0f);
        public static Vector3 VELOCITY_VARIATION = new Vector3(0.2f, 0.5f, 0.2f);
        public static float SNOWFLAKE_SIZE = 0.25f;
        public static float SIZE_DELTA = 0.25f;
        public static float SNOWFLAKE_PER_SEC = 500;
```

Next, let's define some member values. The snow storm is a volumetric effect, so we need to define a cube for it. We're going to leverage the fact that the `ParticleSystem` base class already has an `origin` variable. We will define a cube by augmenting `origin` with `width`, `height` and `depth`.

```
protected float height;
protected float width;
protected float depth;
protected int texture;
```

The constructor takes 1 more variable than the base constructor, which is the size of the snow volume. Because of this, the default constructor is called using the `:` convention with the first two arguments only.

After we store the dimensions of the snow volume, we go ahead and load a snow texture. You can download the texture from [here](#).

```
public SnowstormParticleSystem(int maxParticles, Vector3 origin, Vector3 size)
    : base(maxParticles, origin) {
    height = size.Y;
    width = size.X;
    depth = size.Z;

    texture = GL.GenTexture();
    GL.BindTexture(TextureTarget.Texture2D, texture);
    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMagFilter.Linear);
    GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);
    Bitmap bmp = new Bitmap("Assets/snow.png");
    BitmapData bmp_data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height), ImageLockMode.ReadOnly, System.Drawing.Imaging.PixelFormat.Format32bppArgb);
    GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0, OpenTK.Graphics.OpenGL.PixelFormat.Bgra);
    bmp.UnlockBits(bmp_data);
    bmp.Dispose();
}
```

The shutdown function is straight forward, it deletes the snow texture.

```
public override void Shutdown() {
```

```

        GL.DeleteTexture(texture);
    }
}

```

Update is a little funky, you might be able to write it cleaner than i could. Basically all particles that are alive in a list go from 0 to numParticles. This function goes ahead and updates each particles position by its velocity. If a particle is below the ground plane, we go ahead and copy the last active particle to its index. By not updating the loop counter when this happens we ensure that the next iteration will update the newly copied value. We also replace the old last particle with a new one. If the particle is still alive, we increase the iterator.

```

public override void Update(float deltaTime) {
    for (int i = 0; i < numParticles; /*omitted*/) {
        // Update particles position based on velocity
        particleList[i].position = particleList[i].position + particleList[i].velocity * deltaTime;

        // If the particle hit the ground kill it
        if (particleList[i].position.Y <= systemOrigin.Y) {
            // Copy last active particle into this slot, decrease active particle count.
            // Because we dont step the loop counter here, the next iteration fo the for loop executes on the same
            particleList[i] = particleList[--numParticles];
            particleList[numParticles] = new Particle();
        }
        // Move on to next particle
        else {
            i++;
        }
    }

    // Store elapsed time
    accumulatedTime += deltaTime;

    // Determine how many particles should be alive
    int newParticles = (int)(SNOWFLAKE_PER_SEC * accumulatedTime);

    // Reduce stored time by number of newly spawned particles
    accumulatedTime -= 1.0f / SNOWFLAKE_PER_SEC * newParticles;

    // Request that more particles get spawned
    // Remember, Emit is only present in the base class
    Emit(newParticles);
}
}

```

The render function is relativley easy, it just loops trough all the particles in the list and renders a textured quad for each of them.

```

public override void Render() {
    GL.BindTexture(TextureTarget.Texture2D, texture);

    GL.Begin(PrimitiveType.Quads);
    for (int i = 0; i < numParticles; ++i) {
        Vector3 startPos = particleList[i].position;
        float size = particleList[i].size;

        GL.TexCoord2(0f, 1f);
        GL.Vertex3(startPos.X, startPos.Y, startPos.Z);

        GL.TexCoord2(1f, 1f);
        GL.Vertex3(startPos.X + size, startPos.Y, startPos.Z);

        GL.TexCoord2(1f, 0f);
        GL.Vertex3(startPos.X + size, startPos.Y - size, startPos.Z);

        GL.TexCoord2(0f, 0f);
        GL.Vertex3(startPos.X, startPos.Y - size, startPos.Z);
    }
}

```

```
        GL.End();
    }
```

We've saved the best for last, the `InitParticle` function is what makes the snow look like snow. It sets each variable to some default value plus some random value. The values are defined by the first constants that the class defined on the top. Other than that, this should be pretty straight forward

```
public override void InitParticle(int index) {
    particleList[index].position.X = systemOrigin.X + (float)random.NextDouble() * width;
    particleList[index].position.Y = height;
    particleList[index].position.Z = systemOrigin.Z + (float)random.NextDouble() * depth;

    particleList[index].size = SNOWFLAKE_SIZE + (float)random.NextDouble() * SIZE_DELTA;

    particleList[index].velocity.X = SNOWFLAKE_VELOCITY.X + (float)random.NextDouble() * VELOCITY_VARIATION.X;
    particleList[index].velocity.Y = SNOWFLAKE_VELOCITY.Y + (float)random.NextDouble() * VELOCITY_VARIATION.Y;
    particleList[index].velocity.Z = SNOWFLAKE_VELOCITY.Z + (float)random.NextDouble() * VELOCITY_VARIATION.Z;
}
}
```

Adding snow to the game

Let's head back to the latest texture mapped scene (the one we rendered the UI in) and add some particles to it! First thing is first, add a new `snowstormParticleSystem` member variable, set it to null by default. I called mine `snow`

In initialize, set the particle system to a new system. Use the below values.

```
snow = new SnowstormParticleSystem(5000, new Vector3(0f, 0f, 0f), new Vector3(10f, 10f, 10f));
```

Call the particle systems shutdown function in the scene shutdown.

In the scene render, after the world is rendered, but before anything else happens, render the particle system:

```
RenderWorld();
snow.Render();
```

The particle system also needs to be updated. This means you will have to override the scenes update function, and call the particle systems update function in it. When all is done, wait a few seconds in your scene for all the snow to spawn. The final scene should look like this:



Of course from this scene it's obvious that i'm not an artist . Usually a technical artist will play around with a particle systems attributes to create the desired effects.

Other systems

Other particle systems like fire and lighting are done in similar fashions. If you are confused about how what we did so far has worked, or how that knowledge translates into creating a fire system give me a call on skype and i'll be happy to explain in more detail.

Optimization Techniques

At this point you should be fairly comfortable with OpenGL. We've gone through and done most of the things you would normally do on the job. But a lot of times we skipped best practices, especially in terms of performance. This made the code easier to read, but more costly to execute.

This section will focus on two things. How to optimize your rendering code to render more things faster, and how to load and manage resources; so you can render more interesting things.

We're going to learn how to manage textures, meshes and other objects so that they load and unload automatically, without us having to do much work to keep track of them. We're also going to learn how to parse 3D models from .obj files to load more interesting looking geometry.

Vertex Arrays

Thus far, all examples have used the `GL.Begin / GL.End` method of drawing objects. This method is referred to as **Immediate Mode Rendering**. Immediate mode is useful for simple applications and to prototype code, as it is easy to understand and visualize. However it comes with some performance penalties that make it less useful for applications that need to render lots of geometry at a high framerate, like games.

For example, let's say you're rendering a model containing 2,000 lit and textured triangles using immediate mode. Assume that you're able to pack all of the vertex data into a single triangle strip (This is the best case scenario, it's often not possible). Your rendering code *might* look like this:

```
GL.Begin(PrimitiveType.TriangleStrip);

for (int i = 0; i < 2002; ++i) { // 2002 because of the triangle STRIP
    GL.Normal3(normals[i][0], normals[i][1], normals[i][2]);
    GL.TexCoord2(uvs[i][0], uvs[i][1]);
    GL.Vertex3(verts[i][0], verts[i][1], verts[i][2]);
}

GL.End();
```

There are several problems here, the first of which is that in order to render this model we make over 6000 function calls! Every function call has a very tiny overhead, but with over 6000 calls, this overhead adds up really fast! Remember, no function call is free!

The second and third problems are illustrated in this image:

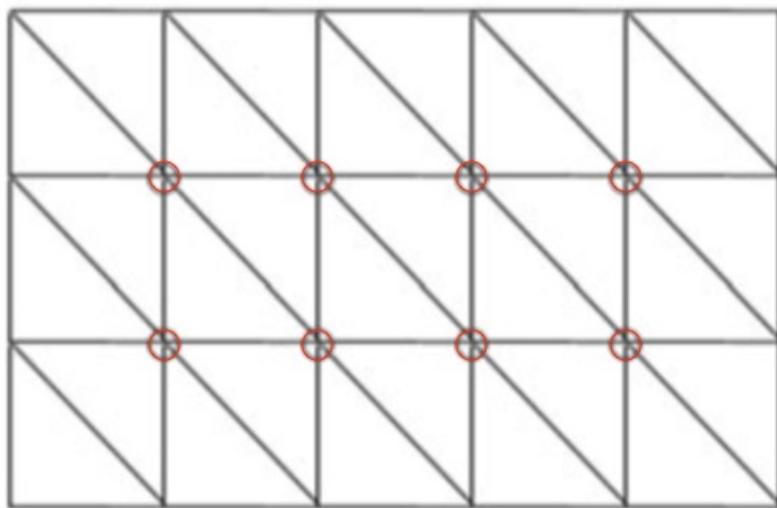


Figure 10.1 A mesh with redundant vertices.

Assuming that this mesh is made up of triangle strips (perhaps it's a portion of the mesh created in pseudo-code above), each circled vertex is redundant. In other words, these vertices are shared by more than 3 triangles, but since a triangle strip can represent at most three triangles per vertex, each of the circled vertices have to be sent to

the graphics card more than once.

This results in additional bandwidth when uploading data to the video card. More importantly, those vertices will need to be transformed and lit multiple times, once for each duplicate vertex.

To address these issues OpenGL provides vertex arrays. A Vertex array has the following advantages:

- Large batches of data can be sent with a small number of function calls.
 - Using vertex arrays we could reduce the above example from 6000 function calls to 2
- Through the use of indexed vertex arrays, vertices can be sent exactly once per triangle mesh, reducing bandwidth and lighting calculations
 - This is the reason 90% of the things we render are triangles, not quads or triangle strips
 - When rendered with indexed arrays we can avoid sending duplicate data

It's important to understand how Vertex Arrays work, as they are the next logical step from **immediate mode** vertex rendering. This does not mean that they are the **BEST** option to render, but they are important to understand.

No professional game ships with immediate mode, vertex arrays used to be the goto solution, you could actually ship a game using vertex arrays (I have). Now that we've discussed the reasons for using vertex arrays, let's see how they are used.

Array based data

The biggest challenge for graphics is getting data. So far we've used simple primitives like cubes and planes. These are easy to describe directly in code using `GL.Vertex3`. Complicated models however will not be easily described in code.

Most games use two approaches to solve this. First, it's possible to **generate geometric data procedurally**. This just entails running some algorithm that will eventually spit out vertices to 3D geometry. We render the sphere in our demos using this method.

More often than not though, geometric data will be **loaded from an external file**. Later on we will actually discuss how to load complex 3D models from an OBJ file. There are standard model formats out there, like OBJ, FBX and DAE but most games use some custom format. This custom format tends to be one that simply makes sense to the programmers. It's likely to be a sphere.

Whichever approach is used, it should be fairly obvious that you don't want to repeat all the work every frame. You certainly don't want to be constantly loading or generating a model. Instead you want to load the data into arrays in `initialize`, and then just use those arrays in the `render` function.

This is the point of vertex arrays, the data is stored in **LARGE** floating point arrays. Perhaps arrays of 2 to 6 thousand elements.

Enabling vertex arrays

Like most OpenGL features, to be able to use vertex arrays, you must first enable them. You might expect this to be done with `GL.Enable`, but it's not. OpenGL provides a separate pair of functions to control vertex array support:

```
void GL.EnableClientState(ArrayCap stateArray);
void GL.DisableClientState(ArrayCap stateArray);
```

The `stateArray` parameter is a flag indicating which type of array you're enabling (or disabling). Each type of vertex attribute (position, normal, color, uv) can be stored in an array, you need to enable whichever attributes you are using individually. These are the valid flags:

- **ArrayCap.VertexArray** Enables an array containing the position of each vertex
- **ArrayCap.NormalArray** Enables an array containing the normal of each vertex
- **ArrayCap.ColorArray** Enables an array containing color information for each vertex
- **ArrayCap.SecondaryColorArray** Enables an array containing color information for each vertex
- **ArrayCap.IndexArray** Enabled an array containing indices into a *color palette* for each vertex
- **ArrayCap.TextureCoordArray** Enabled an array containing the uv coordinates for each vertex
- **ArrayCap.EdgeFlagArray** Enables an array containing an edge flag for each vertex

For example, if you wanted to render a model that has vertex positions, normals and texture coordinates, you'd have to do the following:

```
void GL.EnableClientState(ArrayCap.VertexArray);
void GL.EnableClientState(ArrayCap.NormalArray);
void GL.EnableClientState(ArrayCap.TextureCoordArray);

// TODO: Render

void GL.DisableClientState(ArrayCap.TextureCoordArray);
void GL.DisableClientState(ArrayCap.NormalArray);
void GL.DisableClientState(ArrayCap.VertexArray);
```

Working with arrays

After you have enabled the array types that you will be using, the next step is to give OpenGL some data to work with. It's up to you to create arrays and fill them with the data you will be using. After you have filled the arrays with data, you need to tell OpenGL about these arrays so it can draw them. The function used to do this depends on the type of array you're using, let's look at each function in detail:

In each of the following functions, **stride** indicates the byte offset between array elements. If the array is **tightly packed** (meaning there is no padding between elements) you can set this to 0. Otherwise, you use the stride to compensate for padding, or to pack data for multiple attributes into a single array.

A tightly packed array of two vertices may look like this:

```
float[] verts = new float[] {  
    3f, 2f, 1f,  
    9f, 5f, 6f  
}
```

There is no padding between vertices in the above array. Therefore the stride of that array is 0. But we could use a single array to hold both vertices and normals, like this:

```
float[] modelData = new float[] {  
    3f, 2f, 1f, // VERTEX 1  
    0f, 1f, 0f, // NORMAL 1  
    9f, 5f, 6f, // VERTEX 2  
    0f, 1f, 0f // NORMAL 2  
}
```

In the above example vertex 1 and vertex 2 are separated by 6 floating point numbers. Similarly Normal 1 and Normal 2 are also separated by 6 numbers. Therefore, the stride of the above array is `6 * sizeof(float)`

The datatype of the array (float, int short, etc...) is indicated by **type**.

data is an array of floating points. While it is possible to use strides, I suggest having a separate array for each data type. If your model has 4 triangles, and is rendered as a `PrimitiveType.Triangles` primitive, this array will have $4 * 3$ floats. 4 because you are rendering 4 triangles, 3 because each triangle has 3 vertices.

Other parameters will be explained as they are used in the functions.

```
void GL.VertexPointer(int size, VertexPointerType type, int stride, float[] data);
```

This array contains positional data for vertices. **size** is the number of coordinates per vertex, it must be 2, 3, or 4. The above example has 3 floats for every vertex, so its size is 3. **type** can be Short, Int, Float or Double.

```
void GL.TexCoordPointer(int size, TexCoordPointerType type, int stride, float[] data);
```

This array contains texture coordinates for each vertex. **size** is the number of coordinates, it must be 1, 2, 3 or 4. **type** can be Short, Int, Float or Double.

```
void GL.NormalPointer(NormalPointerType type, int stride, float[] data);
```

This array contains normal vectors for each vertex. Normals are always stored with exactly 3 coordinates (x, y, z) so there is **no size parameter**. **type** can be Byte, Short, Int, Float or Double.

```
void GL.ColorPointer(int size, ColorPointerType type, int stride, float[] data);
```

This specifies the primary color array (vertex color). **size** is the number of components per color, which is either 3 or 4 (RGB or RGBA). **type** can be Byte, UnsignedByte, Short, UnsignedShort, Int, UnsignedInt, Float or Double.

After having specified which arrays OpenGL should use for each vertex attribute, you can begin to have it access the data for rendering. There are several functions you can render with, next we will talk about each of them.

Pinning

The following section is copied from [This](#) article, which explains how vertex arrays should be used. [This](#) page also provides some very usable information on rendering in OpenTK.

You don't have to store data in linear arrays. So long as your data is laid out linearly, you can use pointers to access it as if it were arrays. If you want to use strides within your arrays, that is have all vertex data in a large array instead of having a color and a position array, you have to use pointers

Advanced

You will NOT need to do this if you just store your vertex data in linear array. This is actually super advanced, but I figured I'd dedicate a page to it for the sake of completeness.

Vertex Arrays use client storage, because they are stored in system memory (not video memory). Since .Net is a Garbage Collected environment, the arrays must remain pinned until the GL.DrawArrays() or GL.DrawElements() call is complete.

Pinning an array means that while the array is pinned the garbage collector is not allowed to touch it. If the arrays are unpinned prematurely, they may be moved or collected by the Garbage Collector before the draw call finishes. This will lead to random access violation exceptions and corrupted rendering, issues which can be difficult to trace.

Due to the asynchronous nature of OpenGL, `GL.Finish()` must be used to ensure that rendering is complete before the arrays are unpinned. Let's take a look at how this might be used in context:

```
struct Vertex {
    public Vector3 Position;
    public Vector2 TexCoord;
}
// Vertex size = 5 * sizeof(float)

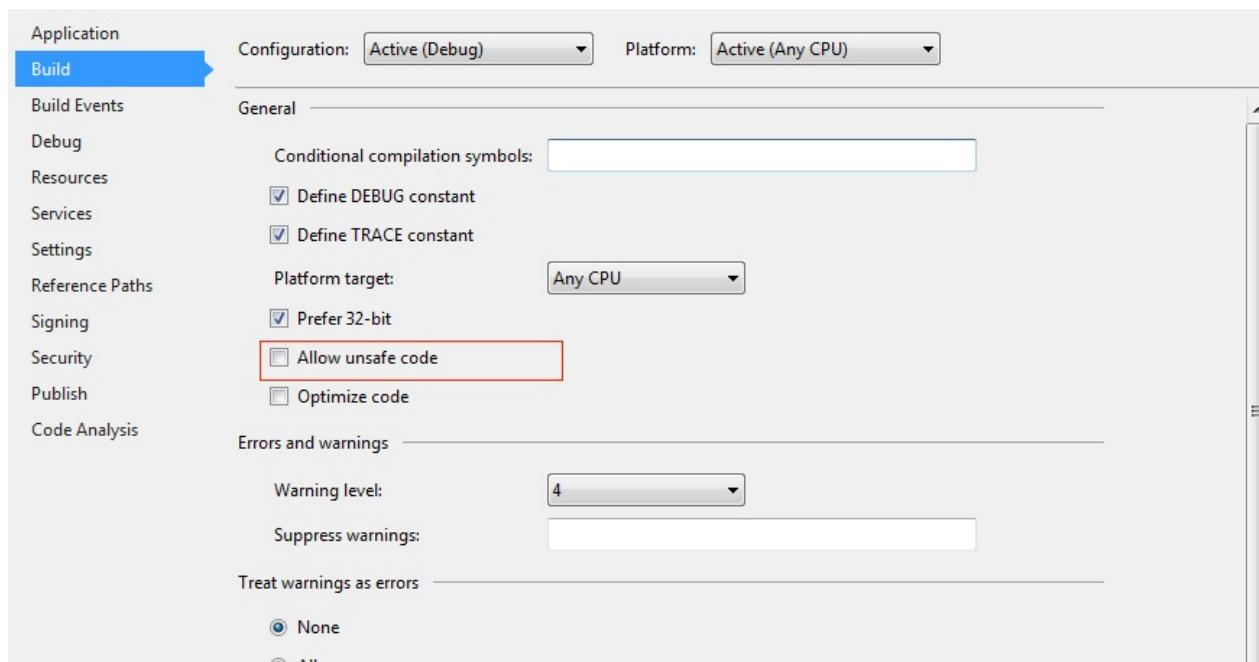
Vertex[] vertices = new Vertex[100];

unsafe { // MUST BE CALLED TO ACCESS POINTERS
    fixed (float* pvertices = vertices) { // Pins memory
        GL.VertexPointer(3, VertexPointerType.Float, 5 * sizeof(float), pvertices);
        GL.TexCoordPointer(2, VertexPointerType.Float, 5 * sizeof(float), pvertices + sizeof(Vector3));
        GL.DrawArrays(BeginMode.Triangles, 0, vertices.Length); // Discussed in next section
        GL.Finish(); // Force OpenGL to finish rendering while the arrays are still pinned.
    }
}
```

Notice that all of the code is wrapped in an **unsafe** section. In order to use pointers you must tell C# that the code you are using is going to be touching memory directly, since this is considered risky business in the C# world we put all such code in an unsafe block.

The **fixed** keyword will pin memory to the desired variable until the fixed block is exited. A pointer datatype is denoted by adding an asterix (*) after the variable type. So a pointer to an array becomes `float*`, but a pointer to a single floating point value would also be `float*`.

In order to run unsafe code, you need to turn it on as a compiler option. You can just check the option under project settings:



GL.DrawArrays

When this function is called, OpenGL iterates over each of the currently enabled arrays, rendering primitives as it goes. Lets take a look at the function to understand how it works:

```
void GL.DrawArrays(PrimitiveType type, int first, int count);
```

type serves the same basic function as the parameter of `GL.Begin`. It specifies what type of primitives the vertex data creates. Valid values are: Points, LineStrip, LineLoop, Lines, TriangleStrip, TriangleFan, Triangles, QuadStrip, Quads and Polygon.

first specifies the index at which we should start drawing. This means you can choose to only draw a part of the array data.

count is how many indices to draw. If your model has 92 vertices, this will be 92. **it's the number of vertices, NOT the number of triangles**. Even after 5 years, this argument still trips me up sometimes!

It should be noted that after calling `GL.DrawArrays` the state of the arrays being processed is undefined. Meaning you have to re-bind them or the behaviour of the next `DrawArrays` call is undefined.

For example, lets try to render two meshes from the same arrays. The following code is bad:

```
void GL.EnableClientState(ArrayCap.VertexArray);
void GL.EnableClientState(ArrayCap.NormalArray);

GL.VertexPointer(3, VertexPointerType.Float, 0, pverts);
GL.NormalPointer(NormalPointerType.Float, 0, pnorms);

GL.DrawArrays(BeginMode.Triangles, mesh1.offset, mesh1.Length); // Draw mesh 1
GL.DrawArrays(BeginMode.Triangles, mesh2.offset, mesh2.Length); // Draw mesh 2

void GL.DisableClientState(ArrayCap.NormalArray);
void GL.DisableClientState(ArrayCap.VertexArray);
```

The proper way to render them would be like this:

```
void GL.EnableClientState(ArrayCap.VertexArray);
void GL.EnableClientState(ArrayCap.NormalArray);

GL.VertexPointer(3, VertexPointerType.Float, 0, pverts);
GL.NormalPointer(NormalPointerType.Float, 0, pnorms);
GL.DrawArrays(BeginMode.Triangles, mesh1.offset, mesh1.Length); // Draw mesh 1

GL.VertexPointer(3, VertexPointerType.Float, 0, pverts);
GL.NormalPointer(NormalPointerType.Float, 0, pnorms);
GL.DrawArrays(BeginMode.Triangles, mesh2.offset, mesh2.Length); // Draw mesh 2

void GL.DisableClientState(ArrayCap.NormalArray);
void GL.DisableClientState(ArrayCap.VertexArray);
```

Take note of how the `VertexPointer` and `Normal` pointers are re-defined between calls to `DrawArrays`

GL.DrawElements

This function is very similar to `GL.DrawArrays`, but it is even more powerful! With `GL.DrawArrays`, your only option is to draw all vertices in the array sequentially. Meaning you can't reference the same vertex more than once. So `GL.DrawArrays` still has the problem of needing duplicate verts.

`GL.DrawElements` on the other hand allows you to specify the array elements in any order, and access each element (vertex) as many times as needed. Let's take a look at the function prototype:

```
void GL.DrawElements(BeginMode mode, int count, DrawElementsType type, uint indices);
```

mode and **count** are used the same as in `GL.DrawArrays`. **type** is the type of the values in the indices array, it should be `UnsignedByte`, `UnsignedShort`, or `UnsignedInt`. **indices** is an array containing indexes for the vertices you want to render.

The last argument for `GL.DrawElements` is an array of unsigned integers `uint[]`. It could also be `unsigned byte` or `short`, depending on the **type** parameter.

To understand the value of this method, it must be reiterated that not only can you specify the indices in any order, you can also specify the same vertex repeatedly in the series. In games, most vertices will be shared by more than one polygon. By storing the vertex once and accessing it repeatedly by its index, you can save a substantial amount of memory.

In addition, OpenGL will only do lighting calculations once for each vertex, this means that by re-using vertices you save on performance by not having to repeat the same computation for identical vertices. Remember, lighting is the most expensive part of the pipeline.

In the next section we're going to implement a demo program using `GL.DrawElements`.

Example

You should go ahead and code along with this example.

First, we include the required libraries, and define our standard application:

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;

namespace GameApplication {
    class DrawElementsExample : Game {
```

Next up are member variables. This demo will have the standard grid, and a cube that has different colors for each vertex. So we're going to make 3 arrays. One for the vertex positions, one for the vertex colors, and one for the indices into the previous two arrays to draw.

```
    Grid grid = null;
    float[] cubeVertices = null;
    float[] cubeColors = null;
    uint[] cubeIndices = null;
```

Next up is a standard copy / paste resize function:

```
public override void Resize(int width, int height) {
    GL.Viewport(0, 0, width, height);
    GL.MatrixMode(MatrixMode.Projection);
    float aspect = (float)width / (float)height;
    Matrix4 perspective = Matrix4.Perspective(60.0f, aspect, 0.01f, 1000.0f);
    GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
    GL.MatrixMode(MatrixMode.Modelview);
}
```

In the `Initialize` function we're going to make a solid grid. We're also going to populate the box arrays. Take note, we only define vertices for the front (1 - 4) and back (5 - 8) faces of the cube. Same with colors. This is because those corners are shared with the top, bottom, left and right faces. We can just include them using the index array.

```
public override void Initialize() {
    grid = new Grid(true);

    cubeVertices = new float[] {
        -1.0f, -1.0f, 1.0f, // Vertex 1
        1.0f, -1.0f, 1.0f, // Vertex 2
        1.0f, 1.0f, 1.0f, // Vertex 3
        -1.0f, 1.0f, 1.0f, // Vertex 4
        -1.0f, -1.0f, -1.0f, // Vertex 5
        1.0f, -1.0f, -1.0f, // Vertex 6
        1.0f, 1.0f, -1.0f, // Vertex 7
        -1.0f, 1.0f, -1.0f // Vertex 8
    };

    cubeColors = new float[] {
        1.0f, 0.0f, 0.0f, // Vertex 1
        0.0f, 1.0f, 0.0f, // Vertex 2
        0.0f, 0.0f, 1.0f, // Vertex 3
        1.0f, 1.0f, 1.0f, // Vertex 4
        1.0f, 0.0f, 0.0f, // Vertex 5
```

```

    0.0f, 1.0f, 0.0f, // Vertex 6
    0.0f, 0.0f, 1.0f, // Vertex 7
    1.0f, 1.0f, 1.0f // Vertex 8
};

```

Still in `Initialize` we next need to define the index array. Of course we need two triangles / cube face. The way to read this is each integer here is an index into the `cubeVertices` and `cubeColors` array.

```

cubeIndices = new uint[] {
    // Front
    0, 1, 2, // Front triangle 1
    2, 3, 0, // Front triangle 2
    // Top
    1, 5, 6, // Top triangle 1
    6, 2, 1, // Top triangle 2
    // Back
    7, 6, 5, // Back triangle 1
    5, 4, 7, // Back triangle 2
    // Bottom
    4, 0, 3, // Bottom triangle 1
    3, 7, 4, // Bottom triangle 2
    // Left
    4, 5, 1, // Left triangle 1
    1, 0, 4, // Left triangle 2
    // Right
    3, 2, 6, // Right triangle 1
    6, 7, 3 // Right triangle 2
};
}

```

Finally let's get started on the `Render` function. First we load up a modelview matrix, then render the grid:

```

public override void Render() {
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(-7.0f, 5.0f, -7.0f), new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
    GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

    GL.Disable(EnableCap.DepthTest);
    grid.Render();
    GL.Enable(EnableCap.DepthTest);
}

```

In order to use `DrawElements` (Or `DrawArrays`) we must first enable client states for each vertex attribute we're going to be rendering:

```

GL.EnableClientState(ArrayCap.VertexArray);
GL.EnableClientState(ArrayCap.ColorArray);

```

Next, we need to tell OpenGL where to find that vertex data:

```

GL.VertexPointer(3, VertexPointerType.Float, 0, cubeVertices);
GL.ColorPointer(3, ColorPointerType.Float, 0, cubeColors);

```

And finally, we just call `DrawElements` with the right arguments. Remember, the count argument is how many vertices you are rendering, NOT TRIANGLES.

```

GL.DrawElements(PrimitiveType.Triangles, cubeIndices.Length, DrawElementsType.UnsignedInt, cubeIndices);

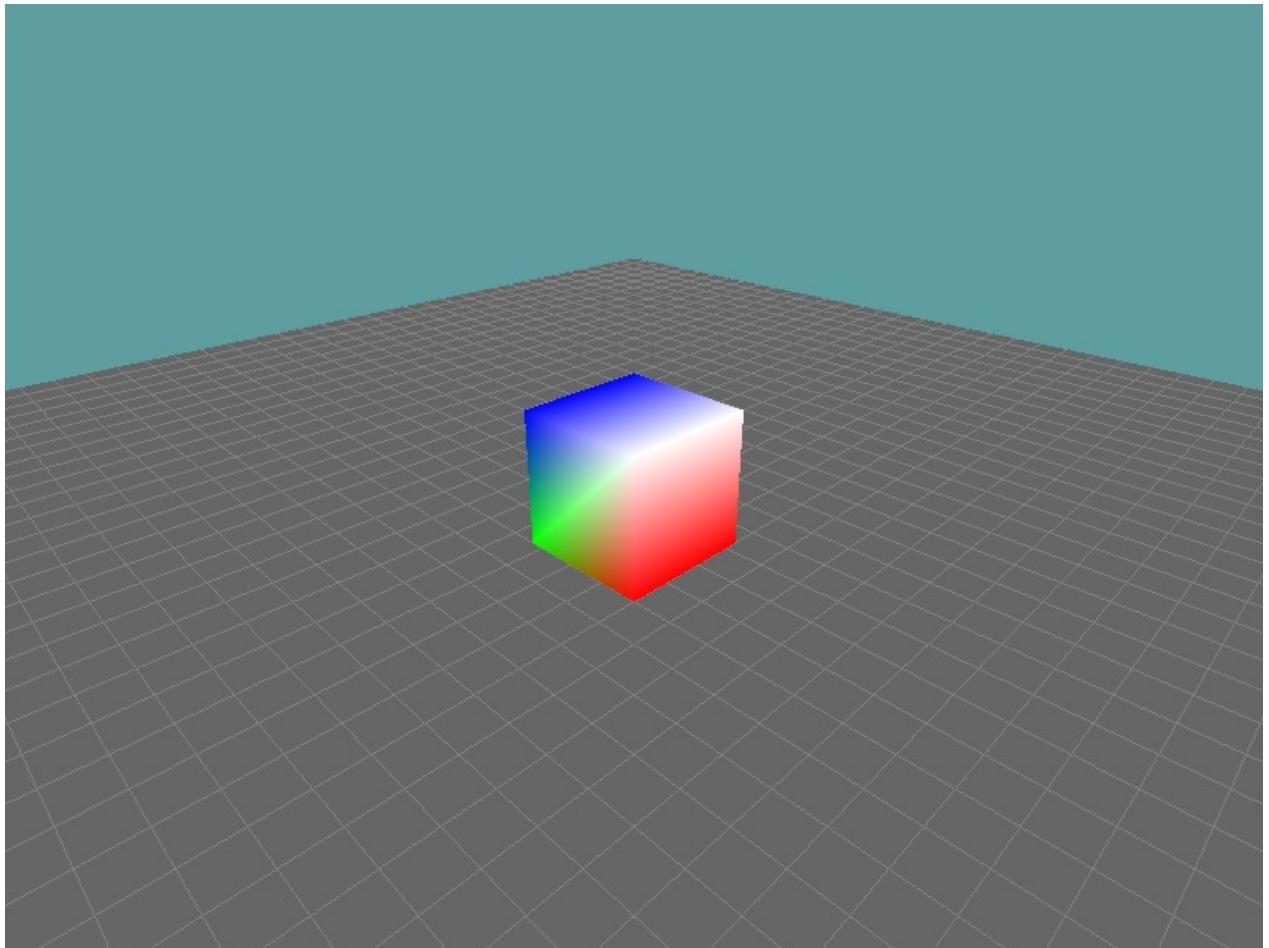
```

After the cube has ben drawn we should Disable any client states that we enabled. This is just so the next draw call can work.

```
        GL.DisableClientState(ArrayCap.VertexArray);
        GL.DisableClientState(ArrayCap.ColorArray);
    }
}
```

Compare what we just wrote to the [Cube function in Primitives.cs](#) (line 100). You can see how much cheaper this is, it doesn't use duplicate vertices, and is using FAR less draw calls!

The final application should look like this:



Vertex Buffers

Vertex Arrays are awesome, they are leaps and bounds more performant than the immediate mode functions. But even vertex arrays have drawbacks. The biggest one is that vertex data lives in CPU memory. This means, that every frame when you call a function like `GL.VertexPointer` that memory has to be uploaded to the GPU.

This means not only are we paying extra performance to upload the memory every frame, for some time identical data exists in both CPU and GPU memory.

Vertex Buffer Objects fix this. A vertex buffer is simply memory that lives on the GPU, much like a texture. Using Vertex buffers is actually very similar to using textures. first you create a buffer, then you fill it with data. Now you are ready to render every frame. Once you're done, you tell OpenGL to delete the buffer.

Code Re-Use

The interesting thing about vertex buffers is that they re-use 90% of the same code that array buffers use. So, even though we have a buffer filled with data, we still need to enable vertex arrays; we just tell OpenGL that instead of using an actual array for the vertex array (VA) data, we want to use a vertex buffer object (VBO).

This does get a little confusing at times. Because both techniques use the same function calls, using a VBO versus using a VA comes down to whether or not an active buffer is bound.

If an active buffer is bound, OpenGL assumes you want to use Vertex Buffers, otherwise it assumes you want Vertex arrays. This will become clear in the next section.

Sample

After having read through the general guide, if the process still doesn't make sense, there is a decent [VBO Example](#) on the OpenTK website.

Generating a vertex buffer

You can generate a new vertex buffer with the following functions:

```
int GL.GenBufer();
void GL.GenBuffers(int n, out int buffers);
```

These functions work the same way that generating textures works. And just like with textures, a generated buffer is not active unless it is bound to be the active buffer.

Binding buffers

You can use the following function to bind your generated buffer ID to an actual buffer:

```
void GL.BindBuffer(BufferTarget target, int buffer);
```

Just like with textures, if your buffer isn't bound, the function calls will not do what you think they will.

The second argument is a buffer id that was previously created. The first argument is what you want to use that buffer as. There are two values we care about:

- **BufferTarget.ArrayBuffer** is used when you want to specify vertex data such as position, normals, uv's etc...
- **BufferTarget.ElementArrayBuffer** is used if you are doing indexed rendering. This buffer will contain the indices for you to render.

Unbind

When you are done drawing with a buffer, you should unbind that buffer. Just like with textures this is done by binding 0 to the active target. Like so:

```
GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);
```

Fill Data

Now that you have generated a new vertex buffer, and you bound it it's time to fill it with data. Just like with textures, once you fill the buffer object with data you no longer need to keep a CPU copy of that data. This is because the data we put into the vertex buffer will be managed by OpenGL.

You fill a buffer with data using the following function:

```
void GL.BufferData(BufferTarget target, System.IntPtr size, float[] data, BufferUsageHint usage);
```

The first argument is super important, it tells OpenGL WHAT the **currently bound** buffer will be used for. There are two enum values we care about here:

- **BufferTarget.ArrayBuffer** contains a large array of floating point values. These are your positions, normals and uv's
- **BufferTarget.ElementArrayBuffer** contains a large array of unsigned integers. This is used to index your geometry if you want to render with `GL.DrawElements`.

The second argument is misleading. Even though it's technically a pointer, the value of that pointer should be an integer, representing how many bytes are to be uploaded to the GPU. It's used like this:

```
new System.IntPtr(cubeData.Length * sizeof(float))
```

The third argument is either a `float` or a `uint` array. This of course depends on the first argument. It's a large array, while you could fill it with interleaved data, I suggest filling it with consecutive data segments.

- **interleaved data** is just data with a stride. That is the memory is laid out as: position, normal, uv, position, normal, uv, position, normal, uv....
- **consecutive data** is laid out linearly. You have all your positions at the front of the array, then all your normals, then all your uvs, like so: position, position, position, normal, normal, normal, uv, uv, uv

Using consecutive data helps speed up the rendering process just a little bit, and makes it easier to write the actual render code.

The last argument is a hint. Some OpenGL implementations will use this hint to optimize how they store the data, based on what you intend to do with it. There are two values to this that we care about:

- **BufferUsageHint.StaticDraw** is used for static objects. That is, any object you draw whose vertices don't change from frame to frame.
- **BufferUsageHint.DynamicDraw** is used for *skinned animation*. That is, 3D characters who animate their arms or legs in a realistic way. We won't use this until we talk about animation.

Rendering

This is where things get interesting! OpenGL tries to re-use the same functions it did for vertex array rendering, but with the arguments interpreted differently. This in my opinion is a VERY bad idea, but it's how they decided to do it.

So if the functions are the same then what tells OpenGL that we want to use vertex buffers instead of arrays? The bound buffer. If a buffer is bound, OpenGL assumes you are rendering VBO's, if not it assumes you are rendering vertex arrays.

Example

Let's take a look at some code that we used to render vertex arrays:

```
GL.EnableClientState(ArrayCap.VertexArray);
GL.EnableClientState(ArrayCap.ColorArray);

GL.VertexPointer(3, VertexPointerType.Float, 0, cubeVertices);
GL.ColorPointer(3, ColorPointerType.Float, 0, cubeColors);

GL.DrawElements(PrimitiveType.Triangles, cubeIndices.Length, DrawElementsType.UnsignedInt, cubeIndices);

GL.DisableClientState(ArrayCap.VertexArray);
GL.DisableClientState(ArrayCap.ColorArray);
```

Now let's convert this to use VBO's. The first thing we need to do is to enable buffers before the call to `GL.VertexPointer`. I'm also going to make sure to disable them once i'm done rendering. Like so:

```
GL.EnableClientState(ArrayCap.VertexArray);
GL.EnableClientState(ArrayCap.ColorArray);

// NEW
GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBuffer);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, indexBuffer);

GL.VertexPointer(3, VertexPointerType.Float, 0, cubeVertices);
GL.ColorPointer(3, ColorPointerType.Float, 0, cubeColors);

GL.DrawElements(PrimitiveType.Triangles, cubeIndices.Length, DrawElementsType.UnsignedInt, cubeIndices);

// NEW
GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);

GL.DisableClientState(ArrayCap.VertexArray);
GL.DisableClientState(ArrayCap.ColorArray);
```

Next up we need to change the `GL.VertexPointer` and `GL.ColorPointer` functions. The only thing that changes is the last argument. Instead of being an array, it now becomes an int pointer, but it does not point to anything!

Instead of using an int-pointer we just pack an integer into the pointer If this sounds stupid, that's because it is, it's a leftover thing from OpenGL being a C-API. So this last parameter now signifies the offset from the beginning of the buffer that our data is located at.

```
GL.VertexPointer(3, VertexPointerType.Float, 0, new System.IntPtr(0));
```

```
GL.ColorPointer(3, ColorPointerType.Float, 0, new IntPtr(sizeof(float) * 8 * 3));
```

In my buffer the first $8 * 3$ floats are the position of my cube. So, the cube vertices are at offset 0. But then the colors come after, so those are

$8 * 3 * \text{sizeof}(\text{float})$ bytes away from the front of the buffer.

Now, if we didn't have an index buffer bound, this would already render. Just because you have a vertex buffer doesn't mean you *have* to use an index buffer. But more often than not you will. This means that we must also change the `GL.DrawElements` function.

The `GL.DrawElements` function changes the same way the others did. The last argument becomes an int-pointer. Its value is an integer, not a real pointer. This integer specifies how many bytes from the beginning of the buffer the data we are rendering is located. The new call becomes:

```
GL.DrawElements(PrimitiveType.Triangles, numIndices, DrawElementsType.UnsignedInt, System.IntPtr.Zero);
```

Putting it all together

Let's take a look at the new render function in full context

```
public override void Render() {
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(-7.0f, 5.0f, -7.0f), new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f,
        GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

    GL.Disable(EnableCap.DepthTest);
    grid.Render();
    GL.Enable(EnableCap.DepthTest);

    GL.EnableClientState(ArrayCap.VertexArray);
    GL.EnableClientState(ArrayCap.ColorArray);

    // Bind vertex and index buffer, OpenGL will use VBO's from here on out
    GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBuffer);
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, indexBuffer);

    // 0 bytes from the beginning of the buffer
    GL.VertexPointer(3, VertexPointerType.Float, 0, new IntPtr(0));
    // The buffer first contains positions, which are 8 vertices, made up of 3 floats each.
    // after that comes the color information, therefore the colors are:
    // 8 * 3 * sizeof(float) bytes away from the beginning of the buffer
    GL.ColorPointer(3, ColorPointerType.Float, 0, new IntPtr(sizeof(float) * 8 * 3));

    // The index buffer only contains indices we want to draw, so they are 0 bytes
    // from the beginning of the array. You can use the constant i use here instead
    // of making a new IntPtr, if the offset you are looking for is 0
    GL.DrawElements(PrimitiveType.Triangles, numIndices, DrawElementsType.UnsignedInt, System.IntPtr.Zero);

    // Unbind vertex and index buffers, OpenGL will draw VA's from here on out
    GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);

    GL.DisableClientState(ArrayCap.VertexArray);
    GL.DisableClientState(ArrayCap.ColorArray);
}
```

Cleanup

Cleaning up vertex buffers is pretty easy, the function call is:

```
void GL.DeleteBuffer(int bufferId);
```

This call signals to OpenGL that it's ok to delete the data in the buffer. After this call i suggest setting bufferId to -1 to signify that it is an invalid buffer.

Review

That was a lot of information in a very short ammount of time. Let's real quick review all the things we've learned.

Globals

You will need 3 pieces of information, a global reference to your vertex buffer, to your index buffer and you will need to know how many indices you are rendering:

```
class SomeClass {
    protected int vertexBuffer;
    protected int indexBuffer;
    protected int numIndices;
```

Initialize

Inside the initialize function you need to generate your buffers and fill them with data. Make sure to unbind at the end, so that rendering doesn't automatically assume it's supposed to render with VBO's.

```
public override void Initialize() {
    grid = new Grid(true);

    float[] vertexData = new float[] {
        ...
    };

    uint[] indexData = new uint[] {
        ...
    };

    numIndices = indexData.Length;

    vertexBuffer = GL.GenBuffer();
    GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBuffer);
    GL.BufferData(BufferTarget.ArrayBuffer, new IntPtr(vertexData.Length * sizeof(float)), vertexData, BufferUsageH:

    indexBuffer = GL.GenBuffer();
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, indexBuffer);
    GL.BufferData(BufferTarget.ElementArrayBuffer, new IntPtr(indexData.Length * sizeof(uint)), indexData, BufferUsa

    GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);
}
```

Render

Rendering is almost the same as rendering vertex arrays. You use the same functions. All you have to do is bind your VBO (Vertex Buffer Object) and possibly IBO (Index Buffer Object), then treat the last element of each function call as a byte offset into the appropriate buffer.

```
public override void Render() {
    Matrix4 lookAt = Matrix4.LookAt(new Vector3(-7.0f, 5.0f, -7.0f), new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f,
```

```

GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

GL.EnableClientState(ArrayCap.VertexArray);
GL.EnableClientState(ArrayCap.ColorArray);

// Bind vertex and index buffer, OpenGL will use VBO's from here on out
GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBuffer);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, indexBuffer);

// 0 bytes from the beginning of the buffer
GL.VertexPointer(3, VertexPointerType.Float, 0, new System.IntPtr(0));
// The buffer first contains positions, which are 8 vertices, made up of 3 floats each.
// after that comes the color information, therefore the colors are:
// 8 * 3 * sizeof(float) bytes away from the beginning of the buffer
GL.ColorPointer(3, ColorPointerType.Float, 0, new System.IntPtr(sizeof(float) * 8 * 3));

// The index buffer only contains indices we want to draw, so they are 0 bytes
// from the beginning of the array. You can use the constant i use here instead
// of making a new IntPtr, if the offset you are looking for is 0
GL.DrawElements(PrimitiveType.Triangles, numIndices, DrawElementsType.UnsignedInt, System.IntPtr.Zero);

// Unbind vertex and index buffers, OpenGL will draw VA's from here on out
GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);

GL.DisableClientState(ArrayCap.VertexArray);
GL.DisableClientState(ArrayCap.ColorArray);
}

```

Shutdown

Finally, when your program is done, you should delete your buffers. Make sure nothing is bound when doing so:

```

public override void Shutdown() {
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);

    GL.DeleteBuffer(vertexBuffer);
    GL.DeleteBuffer(indexBuffer);
}

```

Practice

I want you to go ahead and convert the example from the VA (Vertex Array) section to use a VBO (Vertex Buffer Object) and an IBO (Index Buffer Object) to render.

You will only have 4 member variables: the grid, a vertex buffer handle, an index buffer handle and an integer containing how many indices to draw (size of the index buffer).

Take the current global arrays and make them local to initialize. The final image should look the same.

Meshes

Up to this point we've used a lot of placeholder geometry for spheres, cubes and planes. None of it was very impressive, and most of them render in a very sub-optimal kind of way. Let's go ahead and fix that.

In this section i'm going to show you how to load meshes from possibly the most common format out there, OBJ files. We're going to build a mesh class that can load these OBJ models and render them using vertex buffer objects.

The Format

An OBJ file is just a text file. It's a standard format that supports a LARGE standard with many features. We're only going to implement loading a small specific subset of obj files.

Every OBJ we load we're going to assume has a set vertex format containing a position, a texture coordinate and a normal. Not all OBJ files have all of this information, but the subset we load we assume will.

Let's take a look at what the inside of an OBJ file looks like:

```
# Blender3D v249 OBJ File: untitled.blend
# www.blender3d.org
mtllib cube.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 0.748573 0.750412
vt 0.749279 0.501284
vt 0.999110 0.501077
vt 0.999455 0.750380
vt 0.250471 0.500702
vt 0.249682 0.749677
vt 0.001085 0.750380
vt 0.001517 0.499994
vt 0.499422 0.500239
vt 0.500149 0.750166
vt 0.748355 0.998230
vt 0.500193 0.998728
vt 0.498993 0.250415
vt 0.748953 0.250920
vn 0.000000 0.000000 -1.000000
vn -1.000000 -0.000000 -0.000000
vn -0.000000 -0.000000 1.000000
vn -0.000001 0.000000 1.000000
vn 1.000000 -0.000000 0.000000
vn 1.000000 0.000000 0.000001
vn 0.000000 1.000000 -0.000000
vn -0.000000 -1.000000 0.000000
usemtl Material_ray.png
s off
f 5/1/1 1/2/1 4/3/1
f 5/1/1 4/3/1 8/4/1
f 3/5/2 7/6/2 8/7/2
f 3/5/2 8/7/2 4/8/2
f 2/9/3 6/10/3 3/5/3
f 6/10/4 7/6/4 3/5/4
f 1/2/5 5/1/5 2/9/5
f 5/1/6 6/10/6 2/9/6
f 5/1/7 8/11/7 6/10/7
```

```
f 8/11/7 7/12/7 6/10/7  
f 1/2/8 2/9/8 3/13/8  
f 1/2/8 3/13/8 4/14/8
```

Every line begins with some kind of a letter or phrase to describe the information the line will hold. Then, the information appears in a specific format. There are more line markers than what is listed above / what we will be using. This means its important for your parser to ignore unknown line starts.

Here are the lines we will care about:

- v is a vertex
- vt is the texture coordinate of one vertex
- vn is the normal of one vertex
- f is a face (triangle)

And these are the things we don't care about:

- Any line that begins with # is a comment, just like // in C#
- usemtl and mtllib describe the look of the model. We won't use this in this tutorial.
- s stands for specular, it can be off or it can be some number

A material describes how to shade a model. OBJ files often come with .mtl files which then link to textures. We're not going to care about mtl files. We just want to get vertex data into our game!

Data format

Now that we know that we only care about v, vt, vn and f, let's take a look at the format of each of these lines:

v, vt and vn are simple to understand. The letter is followed by a space, then three space seperated floats. That's the line.

f is more tricky. So, for f 8/11/7 7/12/7 6/10/7 :

- 8/11/7 describes the first vertex of the triangle
- 7/12/7 describes the second vertex of the triangle
- 6/10/7 describes the third vertex of the triangle
- For the first vertex, 8 says which vertex to use. So in this case, -1.000000 1.000000 -1.000000 (index start to 1, not to 0 like in C#)
- 11 says which texture coordinate to use. So in this case, 0.748355 0.998230
- 7 says which normal to use. So in this case, 0.000000 1.000000 -0.000000

A triangle is described as 3 sets of indices into the vertex / uv / normal arrays. It's important to note that arrays in C# are indexed starting at 0, in OBJ land they start at 1. A conversion needs to be made when you write your parser.

Implementation

It may not be immediateley obvious, but because an OBJ file has 3 indices per vertex attribute (Look at the f paramater in the last section) it does not suit its-self well for indexed rendering. This is because a vertex position and normal might not appear at the same index.

Because of this we will need to serialize the data, meaning make it linear. So first we're going to load the data into temp arrays, then loop trough the arrays and flatten them out. Then we can render un-indexed using `GL.DrawArrays`.

Skeleton

I'm going to get you started with a class skeleton for the OBJ loader:

```
using System;
using System.IO;
using System.Collections.Generic;
using OpenTK.Graphics.OpenGL;

namespace GameApplication {
    class OBJModel {
        protected int vertexBuffer = -1;

        protected bool hasNormals = false;
        protected bool hasUvs = false;

        protected int numVerts = 0;
        protected int numNormals = 0;
        protected int numUvs = 0;

        public OBJModel(string path) {
            // TODO: Load
        }

        public void Destroy() {
            // TODO: Destroy
        }

        public void Render(bool useNormals = true, bool useTextures = true) {
            if (vertexBuffer == -1) {
                return;
            }

            if (!hasNormals) {
                useNormals = false;
            }

            if (!hasUvs) {
                useTextures = false;
            }

            // TODO: Render
        }
    }
}
```

The member variables are all protected becasue they are only used for rendering.

Loading

Loading can be broken up into two parts, first reading all of the data in, then parsing all of the data. I'm going to provide some skeleton code for you to work in for this one.

First let's make 6 arrays. One to hold sequential vertex information, one for normals and one for texCoords. When you encounter a line like

```
v 0 10.0 20.0
```

That should add 3 floats to the vertices array, 0, 10 and 20. Same for normals and tex-coords.

Then we have three more arrays, all unsigned integers. These are for the actual triangle data. For example, if a triangle is listed as such

```
f 1//2 4//5 6//8
```

That should put 1, 4 and 6 into the vertex index array, 2, 5 and 8 into the normal index array and nothing into the uv index array. Do take note, ANY of those numbers could be missing, be in the double digits, etc...

```
public OBJModel(string path) {
    List<float> vertices = new List<float>();
    List<float> normals = new List<float>();
    List<float> texCoords = new List<float>();

    List<uint> vertIndex = new List<uint>();
    List<uint> normIndex = new List<uint>();
    List<uint> uvIndex = new List<uint>();

    using (TextReader tr = File.OpenText(path)) {
        string line = tr.ReadLine();
        while (line != null) {
            if (string.IsNullOrEmpty(line) || line.Length < 2) {
                continue;
            }

            // TODO Parse Line, fill out above arrays

            line = tr.ReadLine();
        }
    }
}
```

Once all the data is parsed, it's time to process it into something that's a bit more sequential. For this i'm going to make 3 new arrays that contain positions, normals and uv's all in order.

```
List<float> vertexData = new List<float>();
List<float> normalData = new List<float>();
List<float> uvData = new List<float>();
```

Then, we're going to loop trough the index arrays we've build up and fill the sequential data up in order. One of the things you will notice is `index * 3 + 1`, why is this needed all over the place?

Because indexin assumes that we have an array of float3's, that is, each array element is 3 floats. C# would support this with a multidimensional array, but we can modify the indexing of our big array to emulate that effect.

```
for (int i = 0; i < vertIndex.Count; ++i) {
```

```

        vertexData.Add(vertices[(int)vertIndex[i] * 3 + 0]);
        vertexData.Add(vertices[(int)vertIndex[i] * 3 + 1]);
        vertexData.Add(vertices[(int)vertIndex[i] * 3 + 2]);
    }
    for (int i = 0; i < normIndex.Count; ++i) {
        normalData.Add(normals[(int)normIndex[i] * 3 + 0]);
        normalData.Add(normals[(int)normIndex[i] * 3 + 1]);
        normalData.Add(normals[(int)normIndex[i] * 3 + 2]);
    }
    for (int i = 0; i < uvIndex.Count; ++i) {
        uvData.Add(texCoords[(int)uvIndex[i] * 2 + 0]);
        uvData.Add(texCoords[(int)uvIndex[i] * 2 + 1]);
    }
}

```

We now have enough data to fill in the class member variables:

```

hasNormals = normalData.Count > 0;
hasUvs = uvData.Count > 0;

numVerts = vertexData.Count;
numUvs = uvData.Count;
numNormals = normalData.Count;

```

Finally it's time to upload all this data to the GPU, i'm going to make one last array, this is going to be used to transfer ALL of the above properties to OpenGL. Then we're just going to make a buffer and populate it with data from this new array.

```

List<float> data = new List<float>();
data.AddRange(vertexData);
data.AddRange(normalData);
data.AddRange(uvData);

vertexBuffer = GL.GenBuffer();
GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBuffer);
GL.BufferData(BufferTarget.ArrayBuffer, new IntPtr(data.Count * sizeof(float)), data.ToArray(), BufferUsageHint.Static);
GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
}

```

Unloading

The destroy function is pretty simple:

- Make sure no buffers are bound (unbind buffer)
- Actually delete the buffer
- Set buffer handle to -1

Rendering

Rendering is pretty simple too, i want you to try it on your own

First, enable the appropriate client states. Take note of the arguments the function takes, if for example use normals is false, then don't enable the normals array.

Next, bind your array buffer

Then set your client pointers, the vertex pointer will always start at offset 0, the normal pointer is at numverts * sizeof(float), and the uv pointer is at (numverts + numnorms) * sizeof(float). You have all the data for these offsets in member variables. Mind the parameters of the function!

Finally, call `GL.DrawArrays` to render, the topology is ALWAYS going to be triangles.

Test it

With your new OBJ class loading and presumably rendering primitives it's time to take it for a spin. I'm going to provide a test scene, download [this](#) file and put it in assets to test with.

```
using System;
using OpenTK.Graphics.OpenGL;
using Math_Implementation;

namespace GameApplication {
    class OBJLoadingExample : Game {
        Grid grid = null;
        OBJModel model = null;
        protected Vector3 cameraAngle = new Vector3(0.0f, -25.0f, 10.0f);
        protected float rads = (float)(Math.PI / 180.0f);

        public override void Resize(int width, int height) {
            GL.Viewport(0, 0, width, height);
            GL.MatrixMode(MatrixMode.Projection);
            float aspect = (float)width / (float)height;
            Matrix4 perspective = Matrix4.Perspective(60.0f, aspect, 0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }

        public override void Initialize() {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            Resize(MainGameWindow.Window.Width, MainGameWindow.Window.Height);

            grid = new Grid(true);
            model = new OBJModel("Assets/test_object.obj");

            GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0.0f, 0.5f, 0.5f, 0.0f });
            GL.Light(LightName.Light0, LightParameter.Ambient, new float[] { 0f, 1f, 0f, 1f });
            GL.Light(LightName.Light0, LightParameter.Diffuse, new float[] { 0f, 1f, 0f, 1f });
            GL.Light(LightName.Light0, LightParameter.Specular, new float[] { 1f, 1f, 1f, 1f });
        }

        public override void Shutdown() {
            model.Destroy();
        }

        public override void Update(float dTime) {
            cameraAngle.X += 30.0f * dTime;
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)Math.Sin(cameraAngle.X * rads) * (float)Math.Cos(cameraAngle.Y * rads);
            eyePos.Y = cameraAngle.Z * -(float)Math.Sin(cameraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)Math.Cos(cameraAngle.X * rads) * (float)Math.Cos(cameraAngle.Y * rads);

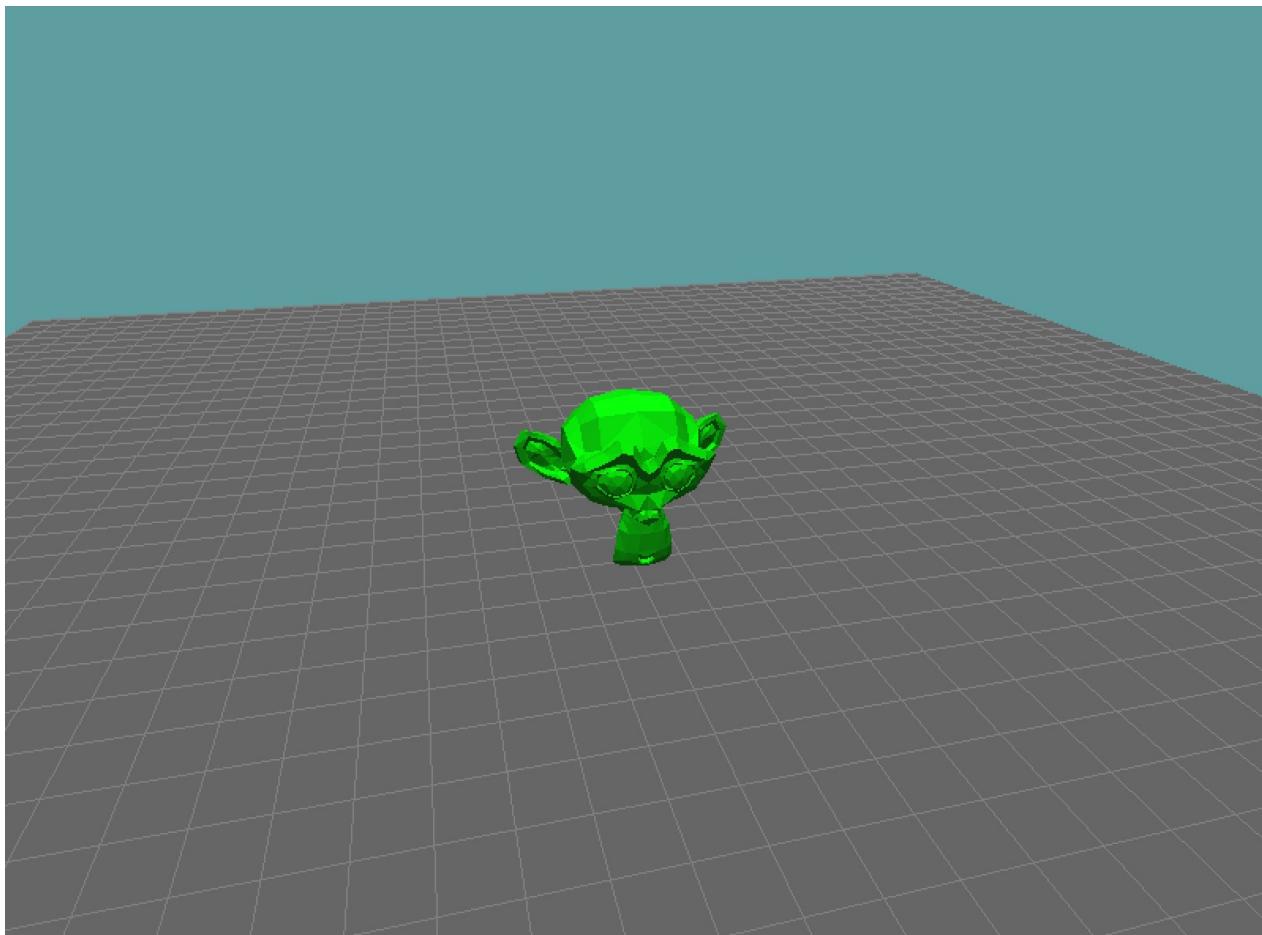
            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            GL.Disable(EnableCap.Lighting);
            GL.Disable(EnableCap.DepthTest);
            grid.Render();
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.Lighting);

            GL.Color3(1f, 1f, 1f);
            model.Render(true, false);
        }
    }
}
```

```
    }  
}  
}
```

If your OBJ loads and renders properly, the above scene should look like this, with a rotating camera:



Resource Management

Loading meshes and textures one at a time isn't the worst thing you can do, but it does make managing them extremely difficult.

Let's say you have a scene with 1 texture used 20 times. You obviously don't want to load the same texture 20 times. So do you keep a counter around? How do you manage the lifecycle of those textures? This is where resource management comes in handy.

Reference counting

The basic idea behind resource management is simple, it's called reference counting. You put all of the code related to loading a specific type of asset into a class. Let's assume we're working with meshes. We could easily make a `MeshManager` class. This class would contain all the code needed to load meshes.

The manager needs to keep a counter for every mesh. This is the reference counter. It increments every time a mesh is loaded and decrements every time a mesh is unloaded. Once this reference counter reaches 0 the mesh in question is deleted, because it's no longer being used.

Handles

When a manager loads an asset, it owns the resource. You are essentially leasing an instance of that resource, eventually you have to give it back. In the case of a mesh you will need a way to access the array of verts, in the case of a texture you will need the OpenGL texture ID. So, how do you get a reference to this?

The answer here is a handle. Once a manager loads a resource it needs to give you a resource handle. When it gives you the resource handle, the reference count is increased. When you give the resource handle back, it is decreased. So what is a resource handle?

It can be anything! Well, anything that will uniquely identify an object. The most common things to use as resource handles are integers and strings. For example, the 2DOpenTKFramework we used returned integer handles.

Previous framework

Speaking of the 2D framework we used, let's take a quick look at how it functioned. Take some time and browse through the `TextureManager` class. Knowing the above information should help you navigate the class.

In the `TextureManager` i have an internal class that represents each texture instance, this class is called `TextureInstance`. The `TextureManager` contains a `List` of `TextureInstance`s.

When a new texture is requested i loop through this list, and check the path of each instance. If a path identical to the texture being requested is found, its reference count is incremented and a handle is returned.

If the texture was not found in the list, i loop through the list, looking for any texture instances with a reference count of 0.

This manager utilizes lazy unloading, meaning when the reference count reaches 0 the texture is not unloaded. Instead textures are unloaded if a new texture is requested and a slot with a 0 reference is available.

If there was no 0 reference slot available and the texture was not already loaded, a new one is made and added to the list. And a handle to that is returned.

So what are the handles in the framework? The handle of a texture is its index in the list. Why did i choose that, it is the cheapest way to retrieve a texture. Loading a texture is expensive, but retrieving it is relatively cheap.

Whats next?

Next up, i want to walk you through actually creating a simple resource manager.

There is no one size fits all solution for asset management, and the methods i show you are far from perfect. Each game figures out what they need and build the asset pipeline around their needs. Engines like Unity or Unreal handle this for you.

I like to create a separate manager for each asset type. That is, i'll have a `TextureManager` a `MeshManager`, an `AudioManager` and so on. Some companies just make a single class called `AssetManager` that manages every possible asset type. Like i said before, there is no wrong or right way. What you use depends on what makes sense to you and what your game needs.

Designing a Manager

Let's go through and create a texture manager together. What functions will this manager need?

- **Initialize** All managers should have some kind of initialize function (that's not the constructor). Even if the function does nothing, it helps keep code readable
- **Destroy** Whatever is initialized must be destroyed! This is also a good place to warn the rest of the code if any resources were not unloaded
- **LoadTexture** This function is the powerhouse, it does the actual work of loading a texture, OR incrementing the reference on an already loaded texture
- **UnloadTexture** `LoadTexture` requests memory, this function will release memory
- **GetSize** We need a way to retrieve the width / height of a texture, we might need it to normalize coordinates.
- **GetTexture** We need a way to access the raw OpenGL texture handle. This is it.

Are there any other considerations to designing this system? Tons! Do we want insertion of new data to be fast and retrieval to be slow? Or do we want insertion to be slow and retrieval to be fast?

This is an important question because it affects the data structures we will use internally. If we want to retrieve the data fast, we must store it in a linear array, but every time we insert that array must be searched; making retrieval slow.

Because most games use a loading screen when loading assets, we're going to design this system for fast retrieval, slow insertion. This means the underlying data structure is going to be an array, so our handles will be integers (Indices into the array)

Helper classes

We can't just have an array of OpenGL handles. We need more data! Like a unique way of identifying each texture (the texture path) and a reference count (an integer).

We could maintain parallel arrays, but it would become a nightmare to maintain them all. Instead we're going to use OOP design, and make private inner classes to the manager.

A private inner class means that the class will not be referenced (or known) from outside of the manager, and that's a good thing, it's just housekeeping data.

Implementation

Let's start laying out the skeleton of the main class and implementing the helper class. One of the new things you will notice is i have blocks of functions wrapped in `#region` and `#endregion` tags.

These tags have no effect on the final executable, they are there to visually help the code be more easily readable and navigatable. They get compiled out like comments.

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using OpenTK.Graphics.OpenGL;
using System.Collections.Generic;

namespace GameFramework {
    public class TextureManager {

        #region Singleton
            private static TextureManager instance = null;
            public static TextureManager Instance {
                get {
                    if (instance == null) {
                        instance = new TextureManager();
                    }
                    return instance;
                }
            }
        private TextureManager() {
        }
        #endregion

        #region HelperClass
            private class TextureInstance {
                public int glHandle = -1;
                public string path = string.Empty;
                public int refCount = 0;
                public int width = 0;
                public int height = 0;
            }
        #endregion

        #region MemberVariables
            private List<TextureInstance> managedTextures = null;
            private bool isInitialized = false;
        #endregion

        #region HelperFunctions
            private void Error(string error) { }
            private void Warning(string error) { }
            private void InitCheck(string errorMessage) { }
            private void IndexCheck(int index, string function) { }
            private bool IsPowerOfTwo(int x) { }
            private int LoadGLTexture(string filename, out int width, out int height, bool nearest) { }
        #endregion

        #region PublicAPI
            public void Initialize() { }
            public void Shutdown() { }
            public int LoadTexture(string texturePath, bool UseNearestFiltering = false) { }
            public void UnloadTexture(int textureId) { }
            public int GetTextureWidth(int textureId) { }
            public int GetTextureHeight(int textureId) { }
            public Size GetTextureSize(int textureId) { }
            public int GetGLHandle(int textureId) { }
        #endregion
    }
}
```

```
    }  
}
```

As you can see this class is a singleton, therefore it has a public `Instance` accessor, and its constructor is marked as **private** to ensure that only this class can make a new instance of its self.

We defined a private inner class called `TextureInstance`. Because its an inner class, you can only make a new instance of it as `new TextureManager.TextureInstance` outside of `TextureManager`, but inside you can just use `new TextureInstance`. This however doesn't matter, because the inner class is private, nothing outside of `TextureManager` can ever reference it.

We have two member variables, one is a list of `TextureInstance`, this is all the textures we are managing, the other is a bool to keep track of weather the manager has been initialized or not.

We have 6 private helper functions. These are private because they will never be used outside of the texture manager class. We will talk about them more in detail as we implement each of them.

Finally, the public API consists of 8 functions. These are the functions that the rest of the code can use to interact with our manager. Again, we will talk about these in more detail as we implement them

Helper Functions

Let's start off by implementing all of the helper functions we're going to need

Error

This function will print something to the console in red. We use it to message any errors to the developer. When an error occurs it will not break the game, but fixing it should be the number 1 priority

```
private void Error(string error) {
    ConsoleColor old = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(error);
    Console.ForegroundColor = old;
}
```

Warning

This function will print something to the console in yellow. Unlike errors warnings are not super high up on the priority list, they just have to be fixed before the game ships.

```
private void Warning(string error) {
    ConsoleColor old = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine(error);
    Console.ForegroundColor = old;
}
```

InitCheck

This function takes a look at the member variable `isInitialized` and throws an error if the manager has not been initialized. We use this in EVERY public API function to make sure no function is called before the manager is initialized.

```
private void InitCheck(string errorMessage) {
    if (!isInitialized) {
        Error(errorMessage);
    }
}
```

IndexCheck

A large number of our public API functions take a handle (index into the `managedTextures` array) as an argument. Those functions will call this function to make sure that the index passed in is valid.

```
private void IndexCheck(int index, string function) {
    if (managedTextures == null) {
        Error(function + " is trying to access element " + index + " when managedTextures is null");
        return;
    }
    if (index < 0) {
```

```

        Error(function + " is trying to access a negative element: " + index);
        return;
    }
    if (index > managedTextures.Count) {
        Error(function + " is trying to access out of bounds element at: " + index + ", bounds size: " + managedTextures.Count);
        return;
    }
    if (managedTextures[index].refCount <= 0) {
        Warning(function + " is acting on a texture with a reference count of: " + managedTextures[index].refCount);
    }
}

```

IsPowerOfTwo

Given an integer, this function returns true if the integer is a power of two, false otherwise. Remember, we only want to load textures that are POT. This function is called in the LoadTexture function to ensure that the loaded texture is a POT texture.

I didn't actually write this function, I just googled "C# Is Power Of Two" and copied the best answer from a stack overflow article. I'd give you the link, but I've since lost the article.

```

private bool IsPowerOfTwo(int x) {
    if (x > 1) {
        while (x % 2 == 0) {
            x >>= 1;
        }
    }
    return x == 1;
}

```

LoadGLTexture

This function is the powerhouse of the manager. It actually loads a texture and returns its OpenGL texture handle. It also returns the textures width and height in arguments marked as `out`.

Most of the code should be fairly familiar, you've loaded plenty of textures before. Pay attention to the error logging, this is the only place that `IsPowerOfTwo` is actually called.

```

private int LoadGLTexture(string filename, out int width, out int height, bool nearest) {
    int id = GL.GenTexture();
    GL.BindTexture(TextureTarget.Texture2D, id);

    if (nearest) {
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Nearest);
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Nearest);
    }
    else {
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);
    }

    Bitmap bmp = new Bitmap(filename);
    BitmapData bmp_data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height), ImageLockMode.ReadOnly, System.Drawing.PixelFormat.Format32bppArgb);

    if (!IsPowerOfTwo(bmp.Width)) {
        Warning("Texture width non power of two: " + filename);
    }

    if (!IsPowerOfTwo(bmp.Height)) {
        Warning("Texture height non power of two: " + filename);
    }
}

```

```
if (bmp.Width > 2048) {
    Warning("Texture width > 2048: " + filename);
}

if (bmp.Height > 2048) {
    Warning("Texture height > 2048: " + filename);
}

GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bmp_data.Width, bmp_data.Height, 0, OpenTK.Graphics.ES2.CompletionType.Immediate);

bmp.UnlockBits(bmp_data);

width = bmp.Width;
height = bmp.Height;
return id;
}
```



Public API

With the inner class and helper functions done, let's actually write the public API that the rest of the code will interact with.

Initialize

The initialize function is pretty simple, first it makes sure that the game is not already initialized, if it is an error gets printed. Then we just make a new `TextureInstance` list and set the `isInitialized` variable to true.

Why is this in the code `managedTextures.Capacity = 100;` ? By setting the initial capacity to 100 we ensure that the List (Vector) has enough room for at least 100 elements. Remember, when a vector data structure runs out of room it doubles in size, when the List runs out it will resize to 200, 400, 800, etc...

```
public void Initialize(OpenTK.GameWindow window) {
    if (isInitialized) {
        Error("Trying to double intialize texture manager!");
    }
    managedTextures = new List<TextureInstance>();
    managedTextures.Capacity = 100;
    isInitialized = true;
}
```

Shutdown

Like initialize shutdown is fairly straight forward. It first checks to make sure that the manager has been initialized.

It then loops trough all the managed textures. If anything has a reference count that is greater than 0 a warning is thrown, but the texture is deleted to make sure that we don't leak memory.

After that we just clear the list, set it to null and set `isInitialized` back to false. The Manager is now shut down, it can be re-initialized if needed.

```
public void Shutdown() {
    InitCheck("Trying to shut down a non initialized texture manager!");

    for (int i = 0; i < managedTextures.Count; ++i) {
        if (managedTextures[i].refCount > 0) {
            Warning("Texture reference is > 0: " + managedTextures[i].path);

            GL.DeleteTexture(managedTextures[i].glHandle);
            managedTextures[i] = null;
        }
        else if (managedTextures[i].refCount < 0) {
            Error("Texture reference is < 0, should never happen! " + managedTextures[i].path);
        }
    }

    managedTextures.Clear();
    managedTextures = null;
    isInitialized = false;
}
```

LoadTexture

The public facing `LoadTexture` can be misleading. It actually calls to the private helper function `LoadGLTexture` to load the texture data. The public facing function is concerned with referenced. To make sure that no texture is loaded twice.

This function returns an integer, it is a **handle**, it's an index into the `managedTextures` array. There are three possible situations this function needs to handle:

First, if the texture path requested is already loaded, increment it's reference count and return it's handle. If the texture is not already loaded, a new texture must be allocated, the next two steps deal with this allocation.

Next, it loops trough all the loaded textures in the array. If any of them have a reference count of < 0, that means that texture is already unloaded. We take this unloaded texture, and load the new texture into it's slot, essentially recycling the texture handle.

Finally, if the texture was not already loaded and we cant recycle any texture handles; we will load the texture and add it to the array, creating a new texture handle.

```
public int LoadTexture(string texturePath, bool UseNearestFiltering = false) {
    InitCheck("Trying to load texture without initializing texture manager!");

    if (string.IsNullOrEmpty(texturePath)) {
        Error("Load texture file path was null");
        throw new ArgumentException(texturePath);
    }

    // Check if the texture is already loaded. If it is, increment it's reference count and return it's handle.
    for (int i = 0; i < managedTextures.Count; ++i) {
        if (managedTextures[i].path == texturePath) {
            managedTextures[i].refCount += 1;
            return i;
        }
    }

    // Try to recycle an old texture handle
    for (int i = 0; i < managedTextures.Count; ++i) {
        if (managedTextures[i].refCount <= 0) {
            managedTextures[i].glHandle = LoadGLTexture(texturePath, out managedTextures[i].width, out managedTextures[i].height);
            managedTextures[i].refCount = 1;
            managedTextures[i].path = texturePath;
            return i;
        }
    }

    // Texture was not loaded, and there are no re-cyclable slots in the array
    // Load texture into a new array slot
    TextureInstance newTexture = new TextureInstance();
    newTexture.refCount = 1;
    newTexture.glHandle = LoadGLTexture(texturePath, out newTexture.width, out newTexture.height, UseNearestFiltering);
    newTexture.path = texturePath;
    managedTextures.Add(newTexture);
    return managedTextures.Count - 1;
}
```

UnloadTexture

Compared to `LoadTexture` , `UnloadTexture` is pretty simple. It decreases the reference count of the provided texture handle by 1. If the reference count of a texture reaches 0, that texture is deleted, it's handle will become eligible to be recycled.

```
public void UnloadTexture(int textureId) {
```

```

    InitCheck("Trying to unload texture without initializing texture manager!");
    IndexCheck(textureId, "UnloadTexture");

    managedTextures[textureId].refCount -= 1;
    if (managedTextures[textureId].refCount == 0) {
        GL.DeleteTexture(managedTextures[i].glHandle);
    }
    else if (managedTextures[textureId].refCount < 0) {
        Error("Ref count of texture is less than 0: " + managedTextures[textureId].path);
    }
}

```

GetTextureWidth

This function is a simple getter that returns the width of a texture. This is information that `LoadTexture` stored inside the actual `managedTextures` element when the texture was loaded.

```

public int GetTextureWidth(int textureId) {
    InitCheck("Trying to access texture width without initializing texture manager!");
    IndexCheck(textureId, "GetTextureWidth");

    return managedTextures[textureId].width;
}

```

GetTextureHeight

This function is a simple getter that returns the height of a texture. This is information that `LoadTexture` stored inside the actual `managedTextures` element when the texture was loaded.

```

public int GetTextureHeight(int textureId) {
    InitCheck("Trying to access texture height without initializing texture manager!");
    IndexCheck(textureId, "GetTextureHeight");

    return managedTextures[textureId].height;
}

```

GetTextureSize

This function works just like `GetTextureWidth` and `GetTextureHeight`. It packs the return values into a `Size` struct.

```

public Size GetTextureSize(int textureId) {
    InitCheck("Trying to access texture size without initializing texture manager!");
    IndexCheck(textureId, "GetTextureSize");

    return new Size(managedTextures[textureId].width, managedTextures[textureId].height);
}

```

GetGLHandle

This function returns the OpenGL handle when provided with a handle to the TextureManager. It's crucial, because you will need the OpenGL handle to actually bind a texture for rendering.

```

public int GetGLHandle(int textureId) {
    InitCheck("Trying to access texture ID without initializing texture manager!");
    IndexCheck(textureId, "GetGLHandle");
}

```

```
    return managedTextures[textureId].glHandle;  
}
```

On your own

Design and create a `MeshManager` for OBJ files. Utilize the OBJ loader created in the last section for this. You will need an inner helper class just like the TextureManager did, instead of tracking an OpenGL handle you will be tracking OBJ objects.

You don't need to include any of the loading code in your manager class, because the OBJLoader classes constructor already takes care of that. Essentially, only make a new instance of OBJLoader when something needs to be loaded.

If you want you can write a class skeleton and get my feedback, or if you feel comfortable you can just build the whole thing in one go.

Example

So now that we have a few managers let's put them to good use. I'm going to do this by making a `TexturedModel` class that will let me display textured 3D models.

```
class TexturedModel {
    protected int objHandle = -1;
    protected int texHandle = -1;

    public TexturedModel(string obj, string texture) {
        texHandle = TextureManager.Instance.LoadTexture(texture);
        objHandle = ModelManager.Instance.LoadModel(obj);
    }

    public void Shutdown() {
        TextureManager.Instance.UnloadTexture(texHandle);
        ModelManager.Instance.UnloadModel(objHandle);
        objHandle = texHandle = -1;
    }

    public void Render() {
        int texture = TextureManager.Instance.GetGLHandle(texHandle);
        OBJLoader model = ModelManager.Instance.GetModel(objHandle);

        GL.BindTexture(TextureTarget.Texture2D, texture);
        model.Render();
        GL.BindTexture(TextureTarget.Texture2D, 0);
    }
}
```

That's it. That's how simple it is to use our new manager classes to display any textured obj model. The constructor takes two arguments, both file paths. Each object is loaded up, a handle to each is kept and released accordingly.

Camera

So far the most interesting camera we've had has been the one rotating around the scene when we want it to. It's ok to look at lighting, but it's not great. In this chapter we're going to go ahead and create a FPS camera.

There are plenty of tutorials out there on this topic, my favorite one is [this](#), it's written for C++ OpenGL. But there are a lot of OpenTK specific ones too, like [this one](#) or [this one](#).

Instead of following those tutorials (which can be overcomplicated at times) we're just going to set up our own.

Making a camera is simple, you need to first figure out the world position of the camera. For this you need to figure out its rotation and orientation. You then take the world position and invert it to get a view matrix.

How you figure out the world position of a camera is the difference between the FPS, RTS, 3rd person, etc... camera models. For an FPS camera, the rotation on the Y and X axis updates based on mouse movement, while the position updates based on the WASD keys.

Framework

I'm going to provide the code for the framework we're going to be working in. It's more or less a copy / paste of the OBJ test scene. Take note of the Using directives, we want to use `openTK.Input` for mouse / keyboard and `Math_Implementation` for Matrix and Vector classes

```
using System;
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using OpenTK.Input;

namespace GameApplication {
    class CameraExample : Game {
        Grid grid = null;
        OBJModel model = null;

        // TODO: Set based on camera input
        protected Matrix4 viewMatrix = new Matrix4();

        public override void Resize(int width, int height) {
            GL.Viewport(0, 0, width, height);
            GL.MatrixMode(MatrixMode.Projection);
            float aspect = (float)width / (float)height;
            Matrix4 perspective = Matrix4.Perspective(60.0f, aspect, 0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }

        public override void Initialize() {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            Resize(MainGameWindow.Window.Width, MainGameWindow.Window.Height);

            grid = new Grid(true);
            model = new OBJModel("Assets/test_object.obj");

            GL.Light(LightName.Light0, LightParameter.Position, new float[] { 0.0f, 0.5f, 0.5f, 0.0f });
            GL.Light(LightName.Light0, LightParameter.Ambient, new float[] { 0f, 1f, 0f, 1f });
            GL.Light(LightName.Light0, LightParameter.Diffuse, new float[] { 0f, 1f, 0f, 1f });
            GL.Light(LightName.Light0, LightParameter.Specular, new float[] { 1f, 1f, 1f, 1f });
        }
    }
}
```

```

    }

    public override void Shutdown() {
        model.Destroy();
    }

    public override void Update(float dTime) {
        // TODO: Move 3D camera
    }

    public override void Render() {
        GL.LoadMatrix(Matrix4.Transpose(viewMatrix).Matrix);

        GL.Disable(EnableCap.Lighting);
        GL.Disable(EnableCap.DepthTest);
        grid.Render();
        GL.Enable(EnableCap.DepthTest);
        GL.Enable(EnableCap.Lighting);

        GL.Color3(1f, 1f, 1f);
        model.Render(true, false);
    }
}
}
}

```

The new thing to note here is the `viewMatrix` matrix. We use it to set the view when rendering. Right now it's set to identity, so you should see almost nothing when rendering. We're going to be updating this based on the move code.

Variables

Let's start by adding some class variables.

```

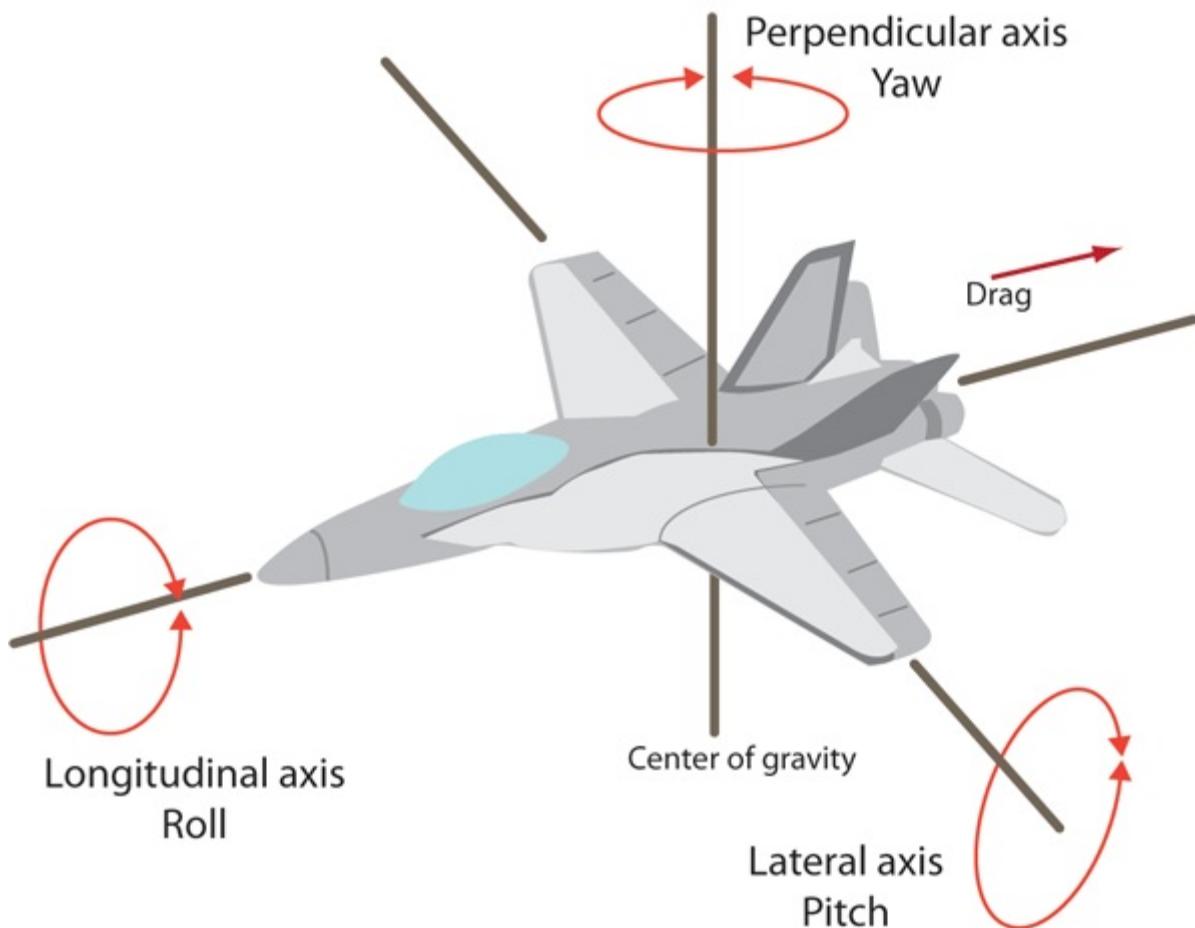
protected float Yaw = 0f;
protected float Pitch = 0f
protected Vector3 CameraPosition = new Vector3(0, 0, 10);
protected Vector2 LastMousePosition = new Vector2();
/*Already exists*/ protected Matrix4 viewMatrix;

```

Yaw and Pitch are the Y and X rotation of the camera respectivley. This represents the cameras orientation in the world.

When we're talking about orientation, we use the terms yaw, pitch and roll. You multiply these together to get an orientation. Order matters!

```
orientation = roll * pitch * yaw;
```



Next we've added a `Vector3` to represent the cameras position in the world. We're going to start the camera off at 10 units in the Z axis. If we started it at 0 it would start INSIDE the 3D model, instead we want to be looking at it.

Last we need to add a `Vector2` to maintain the last position of the mouse. We need this because we have to calculate the delta movement of the mouse. Depending on the input library you are using this might not be needed, some input handlers will have a `GetMouseDelta` function. OpenTK by default does not.

And of course the `viewMatrix` variable was already there in the skeleton framework.

The Camera

We're going to implement our FPS camera in a helper function. I've documented this function using comments, be sure to read them!

```
// Returns the view matrix. Takes in delta time and a movement speed.
Matrix4 Move3DCamera(float timeStep, float moveSpeed = 10f) {
    // Helper variables, we need to know the mouse and keyboard states
    const float mouseSensitivity = .01f;
    MouseState mouse = OpenTK.Input.Mouse.GetState();
    KeyboardState keyboard = OpenTK.Input.Keyboard.GetState();

    // Figure out the delta mouse movement
    Vector2 mousePosition = new Vector2(mouse.X, mouse.Y);
    var mouseMove = mousePosition - Last.mousePosition;
    Last.mousePosition = mousePosition;

    // If the left button is pressed, update Yaw and Pitch based on delta mouse
    if (mouse.LeftButton == ButtonState.Pressed) {
        Yaw -= mouseSensitivity * mouseMove.X;
    }
}
```

```

        Pitch -= mouseSensitivity * mouseMove.Y;
        if (Pitch < -90f) {
            Pitch = 90f;
        }
        else if (Pitch > 90f) {
            Pitch = -90f;
        }
    }

    // Now that we have yaw and pitch, create an orientation matrix
    Matrix4 pitch = Matrix4.XRotation(Pitch);
    Matrix4 yaw = Matrix4.YRotation(Yaw);
    Matrix4 orientation = /*roll */ pitch * yaw;

    // Get the orientations right and forward vectors
    Vector3 right = Matrix4.MultiplyVector(orientation, new Vector3(1f, 0f, 0f));
    Vector3 forward = Matrix4.MultiplyVector(orientation, new Vector3(0f, 0f, 1f));

    // Update movement based on WASD
    if (keyboard[OpenTK.Input.Key.W]) {
        CameraPosition += forward * -1f * moveSpeed * timeStep;
    }
    if (keyboard[OpenTK.Input.Key.S]) {
        CameraPosition += forward * moveSpeed * timeStep;
    }
    if (keyboard[OpenTK.Input.Key.A]) {
        CameraPosition += right * -1f * moveSpeed * timeStep;
    }
    if (keyboard[OpenTK.Input.Key.D]) {
        CameraPosition += right * moveSpeed * timeStep;
    }

    // Now that we have a position vector, make a position matrix
    Matrix4 position = Matrix4.Translate(CameraPosition);
    // Using position and orientation, get the camera in world space
    Matrix4 cameraWorldPosition = position * orientation;
    // The view matrix is the inverse of the cameras world space matrix
    Matrix4 cameraViewMatrix = Matrix4.Inverse(cameraWorldPosition);

    return cameraViewMatrix;
}

```

The function is verbose, but it's pretty simple. Following the steps outlined below you can construct just about any kind of camera.

Update pitch and yaw by the mouse delta position. Because Pitch looks up and down, clamp it to the -90 to 90 range. Once we have these, construct a new orientation.

Update the position vector based on the WASD key states and orientation. Once we have an updated position, make a position matrix.

We can't just move the camera based on world forward and up, we want to take the orientation of the camera into account when moving. That's why we take the right vector (1, 0, 0) and multiply it by the orientation matrix, to get the local right vector of that matrix. Same with forward.

Once we have a position and orientation matrix we can figure out where the camera is in world space.

Once you know where the camera is in world space, the view matrix is just the inverse of that.

Applying the camera

Now that we have the code to move our camera in 3D, we still need to call it.

```
public override void Update(float dTime) {
    viewMatrix = Move3DCamera(dTime);
}
```

With that, go ahead and run the application. You should be able to move with WASD, and when you click your mouse button, dragging it should look around the screen.

You can adjust the mouse sensitivity using the `mouseSensitivity` constant in the move function. If the WASD movement is too slow, you can adjust it using the functions second argument.

Half Space Culling

So far we've rendered everything, no matter what. And that's ok, for small demos. But even the simplest game could not get away from that. Think of a modern game, like Assassins Creed. They have MILLIONS of objects per scene. Rendering them all would make the game unplayable.

So what's the solution? Don't render what you can't see! This is called **view culling**. We cull out all invisible objects, and NEVER actually render them. Over the years view culling has gotten increasingly more complex, but we're going to start off with the simplest implementation, **half space culling**.

The theory behind half space culling is simple. Construct a plane at the position of the camera, the plane will span the cameras X and Y axis, and will face in the direction of its Z axis. Then, test all models we are rendering against this plane. Only render models that are in front of the plane. This way, we don't render anything behind the camera!

How do we actually test what side of the plane a point is on? Using the game programmers bread and butter, the **dot product**! If you want to google how to do this, you should be looking for [Half Space Test](#).

Plane

A *plane* in 3D space can be thought of as a flat surface extending indefinitely in all directions. It can be described in several different ways. For example by:

- Three points not on a straight line
- A normal, and a point on the plane
- A normal, and a distance from origin

Let's define a plane class in code:

```
class Plane {  
    public Vector3 n; // Plane normal. Points x on the plane satisfy Dot(n, x) = d  
    public float d; // Distance from origin, d = Dot(n, p), for a given point p on the plane  
}
```

So we've chosen to represent a plane as a normal and a distance from origin. We can calculate this plane, from 3 (3D) points

```
static Plane ComputePlane(Vector3 a, Vector3 b, Vector3 c) {  
    Plane p = new Plane();  
    p.n = Vector3.Normalize(Vector3.Cross(b - a, c - a));  
    p.d = Dot(p.n, a);  
    return p;  
}
```

When two planes are not parallel to each other, they intersect in a line. Similarly, three planes (two not parallel to each other) intersect at a 3D point.

Theory

The theory of a half space test is simple. Given a plane, with a normal. What is the angle between the plane and the

point!

Remember how the dot product works! If we take the planes up vector, and a vector from the plane to the point we are testing we can do a dot product between the two vectors.

The result of this dot product is the half space test, remember if the result of the dot product is:

- 0, the vectors are perpendicular (90 degrees)
- positive, the vectors are less than 90 degrees apart
- negative, the vectors are more than 90 degrees apart

So, if the dot product of the planes vector and the vector from the plane to the point is:

- 0, the point is on the plane
- positive, the point is in front of the plane
- negative, the point is behind the plane

Half Space Test

Before we talk about the half space test, let's take a quick look at the plane equation (Equasion of a plane).

$$a * x + b * y + c * z + d = 0$$

Where, ABC is the normal of the plane, D is the distance of the plane from origin and XYZ is some point. This equasion states that if point XYZ is on the plane, the result of the above equasion is 0. If a point is in front of the plane the result will be a positive number (distance from the plane) and if the point is behind the plane the result will be a negative number.

Because you already know what ABC and D are, you could just plug int XYZ into the above equasion and return the result. Try implementing this function in code

```
static float HalfSpace(Plane p, Vector3 v) {
    // TODO: Return result of plane equasion
}
```

The above code is the preffered method of implementing the half space test. A solution is given at the end of the page

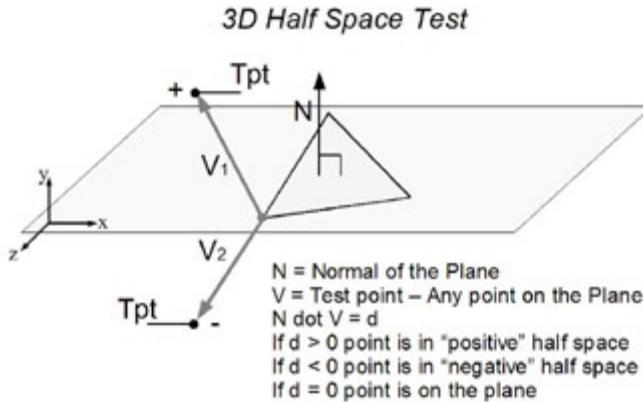
Alternate representation

There are other ways of representing planes, and doing the half space test. For example, a plane can be represented by a normal and any point on the plane. As opposed to what we have now, a normal and a distance from origin.

If this was the case, the half space test would in involve subtracting the plane point from the point you are testing to get a direction vector, then doing a dot product with the plane normal and direction vector.

Because the dot product returns a number that is relative to an angle this representation might be easier to understand.

- If the dot of the plane normal and direction vector is negative, then the angle is greater than 90 degrees
- if it is 0 then the angle is exactly 90 degrees (on the plane)
- If it is positive, then the angle is less than 90 degrees



We can actually make a point on the plane by multiplying the plane normal by the plane distance. Once we have a point on the plane we could use the dot product to perform a half space test.

```
public static float HalfSpace(Plane p, Vector3 v) {
    Vector4 N = new Vector4(p.n.X, p.n.Y, p.n.Z, 0f);
    Vector4 PointOnPlane = new Vector4(p.n.X * p.d, p.n.Y * p.d, p.n.Z * p.d, 1);
    Vector4 V = PointOnPlane - new Vector4(v.X, v.Y, v.Z, 1f);
    return Vector4.Dot(N, V);
}
```

Implementation

Cool, now that we understand the half space test (If you don't call me on skype) let's go ahead and implement it in our test scene.

Implement a `Plane` class, and the `HalfSpace` function. For ease of use, make the `HalfSpace` function a static member of the `Plane` class. Also, make the `ComputePlane` function a static member of the plane class.

Inside the example application, after all the camera variables, add a new `Plane` variable:

```
... old code
protected Vector3 CameraPosition = new Vector3(0, 0, 10);
protected Vector2 LastMousePosition = new Vector2();
protected Matrix4 viewMatrix;
// NEW
Plane cameraPlane = null;
```

We're going to construct this new plane inside the `Move3DCamera` function. We know we can construct a plane using 3 points, so what 3 points of the camera can we use to make a plane?

Well we know the forward and right and up of the camera. We can use the right and up to construct a camera plane whose normal will be the camera forward. But that's only two vectors, what about the third one? We can invert the right vector to get a left vector.

This means we can create the camera plane using the camera world matrix left, right and up planes. Let's see how

this would look in code:

```

...old code
Matrix4 position = Matrix4.Translate(CameraPosition);
Matrix4 cameraWorldPosition = orientation * position;
Matrix4 cameraViewMatrix = Matrix4.Inverse(cameraWorldPosition);

// NEW
right = new Vector3(cameraWorldPosition[0, 0], cameraWorldPosition[1, 0], cameraWorldPosition[2, 0]);
Vector3 left = new Vector3(-right.X, -right.Y, -right.Z);
Vector3 up = new Vector3(cameraWorldPosition[0, 1], cameraWorldPosition[1, 1], cameraWorldPosition[2, 1]);

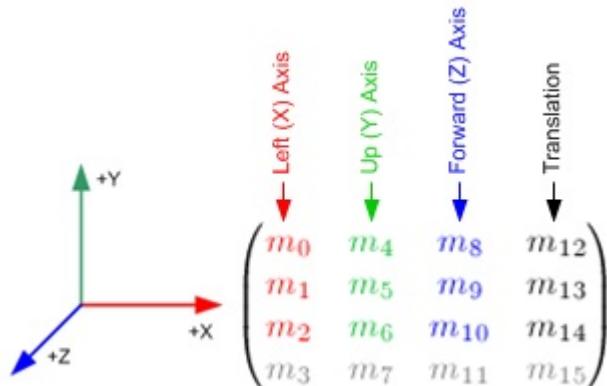
right = Matrix4.MultiplyPoint(cameraWorldPosition, right);
left = Matrix4.MultiplyPoint(cameraWorldPosition, left);
up = Matrix4.MultiplyPoint(cameraWorldPosition, up);

cameraPlane = Plane.ComputePlane(left, right, up);

// OLD
return cameraViewMatrix;
... old code

```

We get the right, and up out of the matrix, then invert right to get left. Remember, you can extract a matrices forward, right and up basis vectors because they make up the upper 3x3 matrix:



We could have multiplied the right and up vectors by the orientation matrix, like earlier in this same function, but extracting the vectors straight from the matrix is much simpler (and a lot less expensive). And you should be aware of both ways to extract the vector.

However, extracting these vectors is not enough. These are vectors, not points, so the translation has not been applied yet. That's why we multiply each of these vectors by the world matrix of the camera.

After we have left, right and up in world space (each one unit away from the cameras position), we can construct the camera plane.

Finally, lets modify the render code:

```

if (Plane.HalfSpace(cameraPlane, new Vector3(0f, 0f, 0f)) >= 0) {
    GL.Color3(1f, 1f, 1f);
    model.Render(true, false);
}
else {
    Console.WriteLine("Green susane culled");
}

```

Now if you move past susane, or rotate the camera that she's off screen, we wont see her. But some text will be

written to the console instead.

Why are we testing 0, 0, 0 for susane, because she is rendered at origin. If you have a model rendered at 2, 4, 6 you would test that point.

Solution

Above i mentioned a solution would be given at the end of the page:

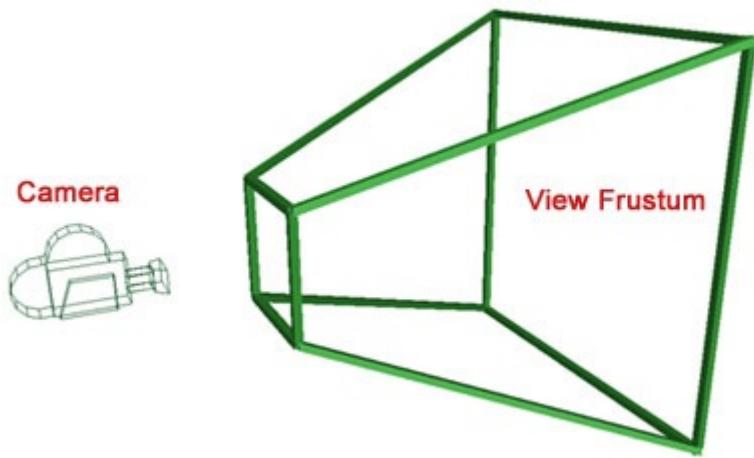
```
static float HalfSpace(Plane p, Vector3 v) {
    return (p.n.X*v.X) + (p.n.Y*v.Y) + (p.n.Z*v.Z) + p.d;
}
```

Frustum Culling

Halfspace culling was a good start, but it's far from optimal. For one, you don't have a 180 degree field of view, the camera can usually only see about 60 degrees wide. Also, it does not take any of the other planes into account. Things might be out of view to the left, right, top, bottom, they may be too far or behind the camera.

This is where Frustum Culling comes in. We don't use frustum culling in conjunction with halfspace culling, INSTEAD OF halfspace culling we do frustum culling.

This method revolves around the cameras frustum:



Basically if an object is not inside the green frustum it will not be rendered. The math behind frustum culling isn't much different than the math behind Half Space Culling. It's the same thing actually!

To do Furstum culling, we first extract the 6 planes that make up the Frustum from the modelview matrix. Then, we do a half space test with ALL 6 planes. If the point is in front of all 6 planes then we render the object. If any one fails, we reject the object from being rendered.

Extracting planes

Recall the plane equation

$$a * x + b * y + c * z + d = 0$$

To extract the planes of a frustum, you need the **viewProjection** matrix, which you get by multiplying the projection and view matrices together. The plane values can be found by adding or subtracting one of the first three rows of the **viewProjection** matrix from the fourth row.

- **Left Plane** Row 1 (addition)
- **Right Plane** Row 1 Negated (subtraction)
- **Bottom Plane** Row 2 (addition)
- **Top Plane** Row 2 Negated (subtraction)
- **Near Plane** Row 3 (addition)
- **Far Plane** Row 3 Negated (subtraction)

So, assuming that matrix `mp` is the view-projection matrix, you could get the left and right planes like so:

```
Matrix4 mv = perspective * view;

Vector4 row1 = new Vector4(mv[0, 0], mv[0, 1], mv[0, 2], mv[0, 3]);
Vector4 row4 = new Vector4(mv[3, 0], mv[3, 1], mv[3, 2], mv[3, 3]);

Vector4 p1 = row4 + row1;
Vector4 p2 = row4 - row1;

Plane left = new Plane();
left.n = new Vector3();
left.n.X = p1.X;
left.n.Y = p1.Y;
left.n.Z = p1.Z;
left.d = p1.w;

Plane right = new Plane();
right.n = new Vector3();
right.n.X = p2.X;
right.n.Y = p2.Y;
right.n.Z = p2.Z;
right.d = p2.w;
```

Frustum Culling

Once you have all the planes of a matrix, the actual Frustum Culling comes down to doing 6 half space tests. Assume that we have our frustum defined like so:

```
Plane[] frustum = new Plane[6];

void Init() {
    for (int i = 0; i < 6; ++i) {
        frustum[i] = new Plane();
    }

    ExtractPlanes(); // Fill the global frustum variable with planes
}
```

Now that we have a frustum, the culling code becomes very simple, first we use a helper function to test if a point is inside a frustum:

```
public bool PointInFrustum(Plane[] frustum, Vector3 point) {
    foreach (Plane plane in frustum) {
        if (Plane.HalfSpace(plane, point) < 0) {
            return false;
        }
    }
    return true;
}
```

And finally you just use it in your render function for anything that you want to have culled:

```
void Render() {
    if (PointInFrustum(frustum, object.Position)) {
        object.Render();
    }
}
```

Implementation

As you can see above it's useful to make a Vector4 into a Plane, so let's add a new helper function to the Plane class to create one from a Vector:

```
public static Plane FromNumbers(Vector4 numbers) {
    Plane p = new Plane();
    p.n = new Vector3();
    p.n.X = numbers.X;
    p.n.Y = numbers.Y;
    p.n.Z = numbers.Z;
    p.d = numbers.W;
    return p;
}
```

Next we need to make two member variables:

```
Plane[] frustum = new Plane[6];
float aspect = 0f;
```

The reason for frustum is pretty self explanatory, you need a frustum to do frustum culling. The aspect, not so much. We need an aspect ratio to recreate the projection matrix to extract planes from.

Aspect is actually already defined in the `Resize` function, take out the local definition of aspect from `Resize` so that it sets the value of the member variable.

Now, we have an array of 6 Planes, but we do not have 6 planes. We only allocated the array. Let's allocate the actual planes in the `Initialize` function.

```
... old code
MouseState mouse = OpenTK.Input.Mouse.GetState();
LastMousePosition = new Vector2(mouse.X, mouse.Y);
// NEW
for (int i = 0; i < 6; ++i) {
    frustum[i] = new Plane();
}
// OLD
viewMatrix = Move3DCamera(0f);
... old code
```

Now that the actual planes have been created we have a frustum. Let's update the `Move3DCamera` function to populate this frustum. We're not going to take out any code (the near plane generation), we're just going to add new code to generate a frustum as well.

After the cameraPlane (near plane) has been created, make a projection matrix, then use that to make a `viewProjection` matrix. Then, extract each row into a Vector4:

```
... old code
cameraPlane = Plane.ComputePlane(left, right, up);
// NEW
Matrix4 perspective = Matrix4.Perspective(60.0f, aspect, 0.01f, 1000.0f);
Matrix4 mv = perspective * cameraViewMatrix;

Vector4 row1 = new Vector4(mv[0, 0], mv[0, 1], mv[0, 2], mv[0, 3]);
Vector4 row2 = new Vector4(mv[1, 0], mv[1, 1], mv[1, 2], mv[1, 3]);
```

```

Vector4 row3 = new Vector4(mv[2, 0], mv[2, 1], mv[2, 2], mv[2, 3]);
Vector4 row4 = new Vector4(mv[3, 0], mv[3, 1], mv[3, 2], mv[3, 3]);

... old code

```

After we have those 4 rows calculating the camera planes becomes super simple, right after they where made, add this code to populate the frustum:

```

frustum[0] = Plane.FromNumbers(row4 + row1);
frustum[1] = Plane.FromNumbers(row4 - row1);
frustum[2] = Plane.FromNumbers(row4 + row2);
frustum[3] = Plane.FromNumbers(row4 - row2);
frustum[4] = Plane.FromNumbers(row4 + row3);
frustum[5] = Plane.FromNumbers(row4 - row3);

```

Next, let's add a helper function to test if a point is inside the frustum:

```

public bool PointInFrustum(Plane[] frustum, Vector3 point) {
    foreach (Plane plane in frustum) {
        if (Plane.HalfSpace(plane, point) < 0) {
            return false;
        }
    }
    return true;
}

```

And finally, we have to actually use this function in the Render function to determine what gets drawn and what does not:

```

if (PointInFrustum(frustum, new Vector3(0f, 0f, 0f))) {
    GL.Color3(1f, 1f, 1f);
    model.Render(true, false);
}
else {
    Console.WriteLine("Green susane culled");
}

```

That's it. If you run the game now you should see susane, until the center point of the world goes off-screen.

Non-Points

You might have noticed a flaw in all of this visibility optimization. With susane when the center of the world goes out of bounds (susanes position), she disappears. But, really half of the geometry is still on screen.

You can imagine larger geometry will have MUCH more noticeable popping. Generally we don't want this. I'd rather an unseen object render than a seen object not render.

This is expected. To do proper Frustum Culling involves more math and primitive geometry testing. These topics are outside the realm of rendering, we will discuss them when we talk about scene management.

Advanced

First, this isn't the end of our OpenGL journey. We will revisit OpenGL later with "Modern OpenGL" (as opposed to legacy). We're also going to revisit OpenGL for "Animations".

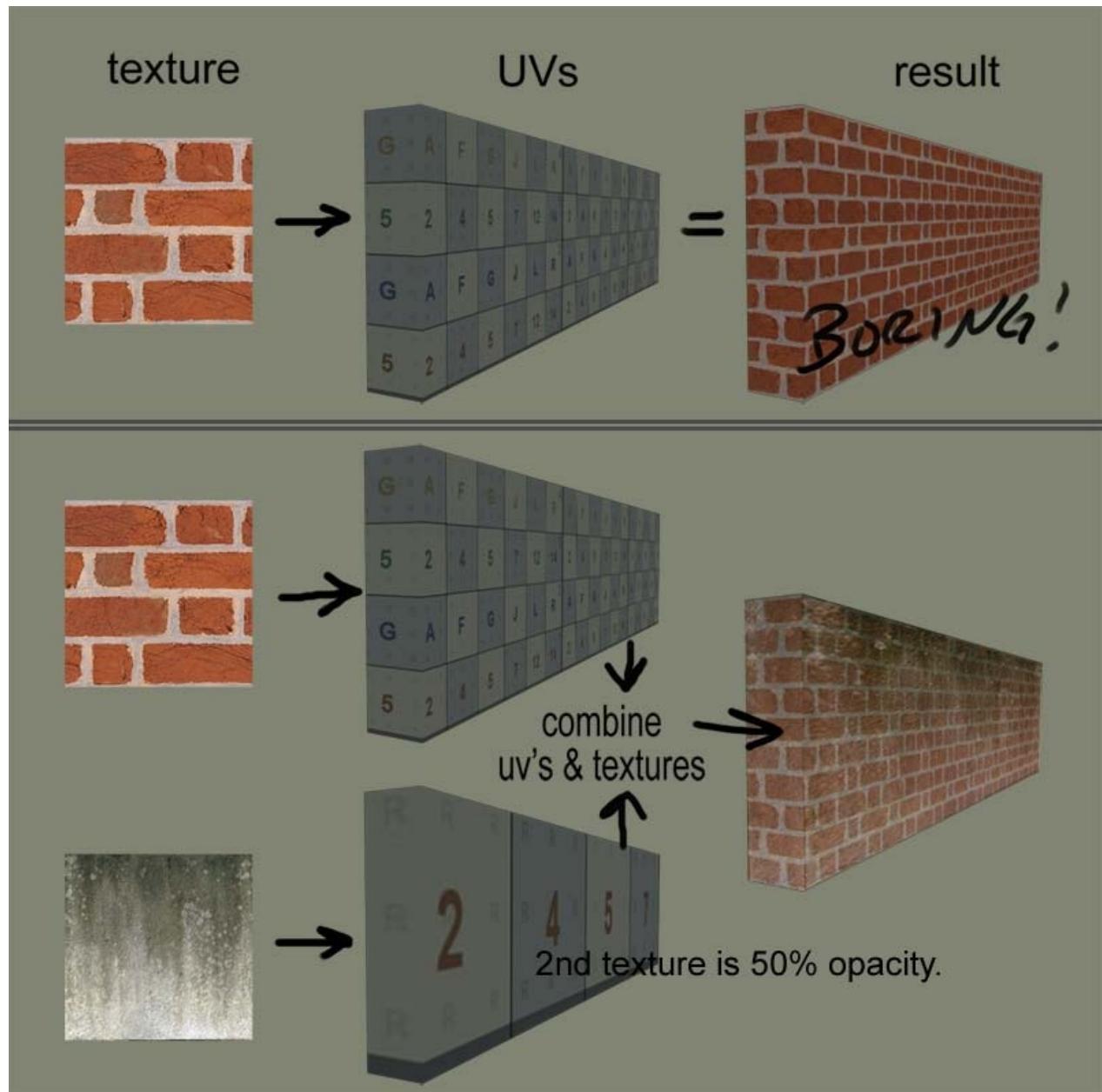
However graphics programming is vast. I decided to include a section on commonly used techniques, that you will not need to implement (this whole section is a read-along); but you should still know how they work.

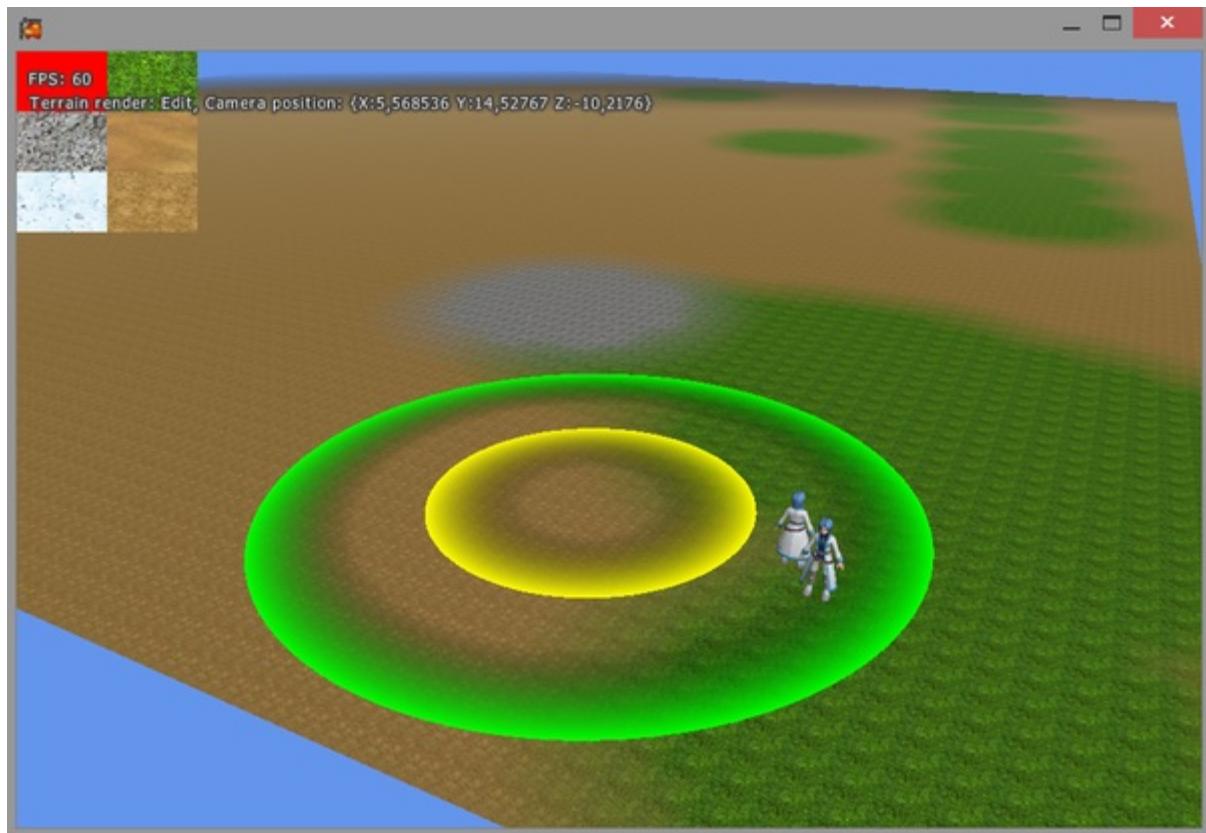
Most of these i have not implemented myself, but they come up in conversation. I usually have to google these, sometimes you can find them on youtube, or in online tutorials.

Multi Texturing

OpenGL has facilities to support multi-texturing / texture environments. Multitexturing isn't too expensive, it's mainly used to blend dirt and grass textures on terrains. Some lighting effects also use multitexturing.

The effect is mostly done by OpenGL, you just have to look up the right commands to issue, and the right order to issue them in. Here are some examples of multi texturing in action:





Sub Textures

You already know how to texture atlas. When you use an atlas you have a large image that's packed with smaller images. You access those images by modifying the uv coordinate that you draw.

Sub-Textures is almost the same thing. With sub-textures you can load sections of a texture as their own texture handles, essentially making them new textures.

This feature is mainly deemed to be useless, not many people use it. Especially for games.

Billboarding

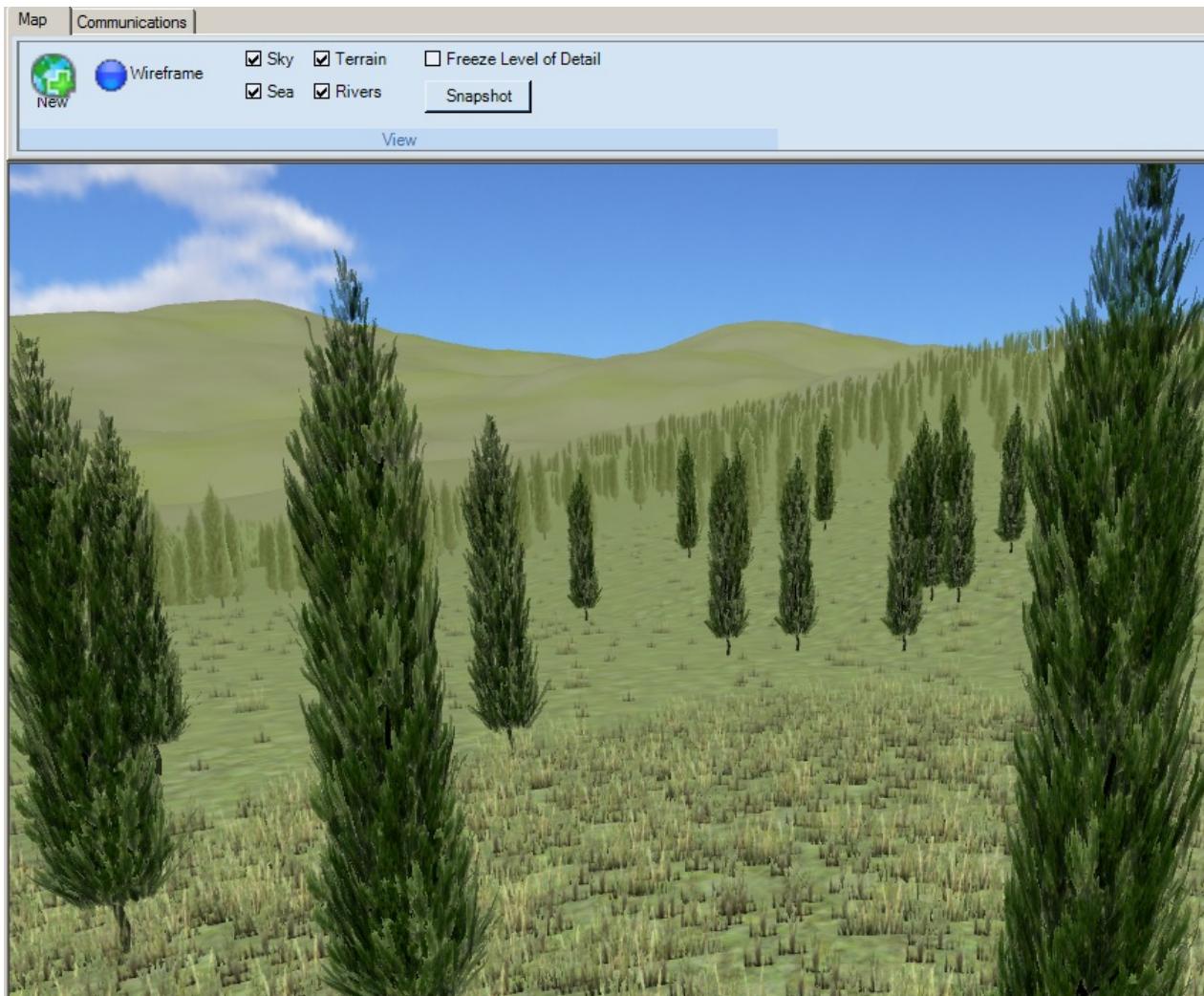
Billboarding is a technique of making polygons always face the viewer. Why is this useful? Suppose we're making an outdoor scene. There are trees, LOTS of trees. They have LOTS of polygons. But far away trees have little detail to them. You could replace far away trees with textured quads that always face the camera.

Changes are the player wouldn't even notice this. I can't, here is an example:

Trees: 122607
Meshes (draw calls): 49
Quads: 122607
Triangles: 245214



In some games all foliage may be billboarded:



Flashes, Special FX, grenades and other objects are also good candidates for billboarding.

Making a billboard

If every object has a matrix, making a billboard is easy. When we did frustum culling we looked at how to extract the forward, right and up vectors from a matrix. Billboarding is mainly that.

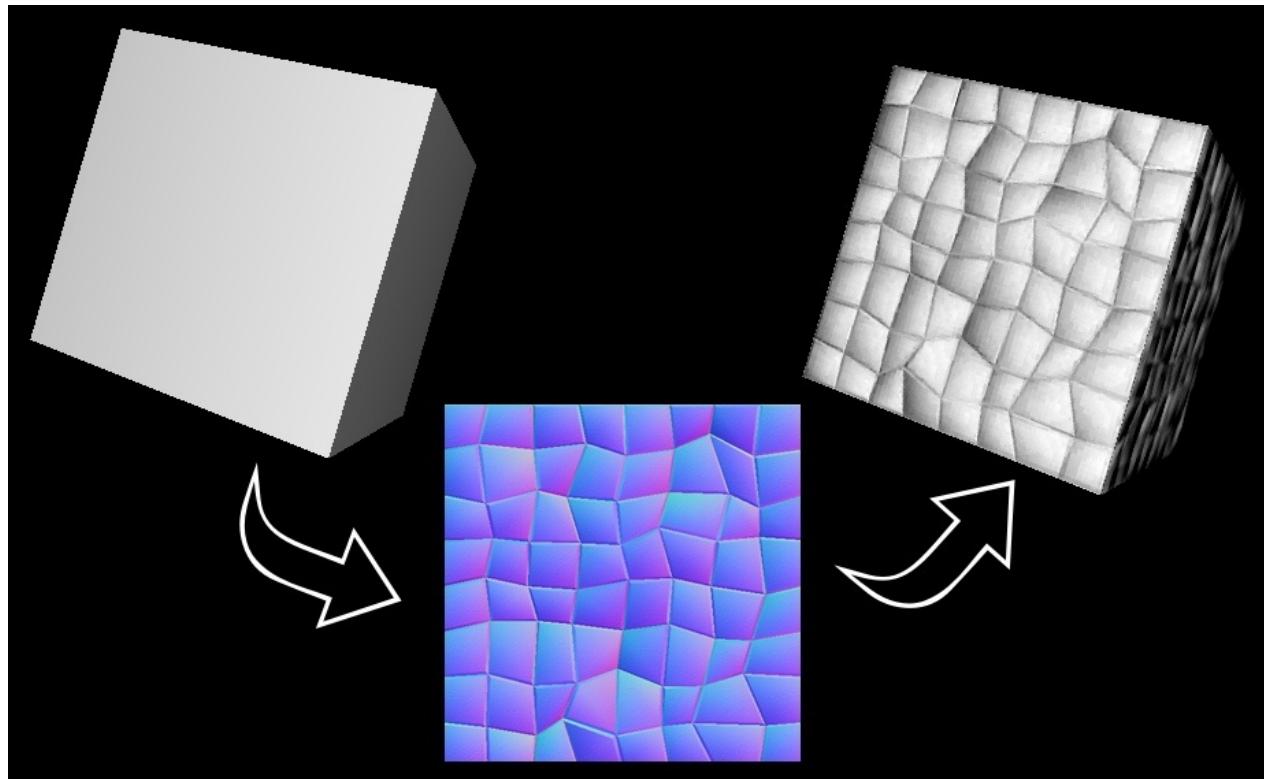
Take the cameras forward vector and invert it. Set the billboarded objects forward vector to this. We don't want the up axis of the billboard to change, but we do want to make sure that the right vector stays orthogonal. So take the billboards up and forward vectors and do a cross product. Set the billboards right vector to the result of the cross product.

Normal Mapping

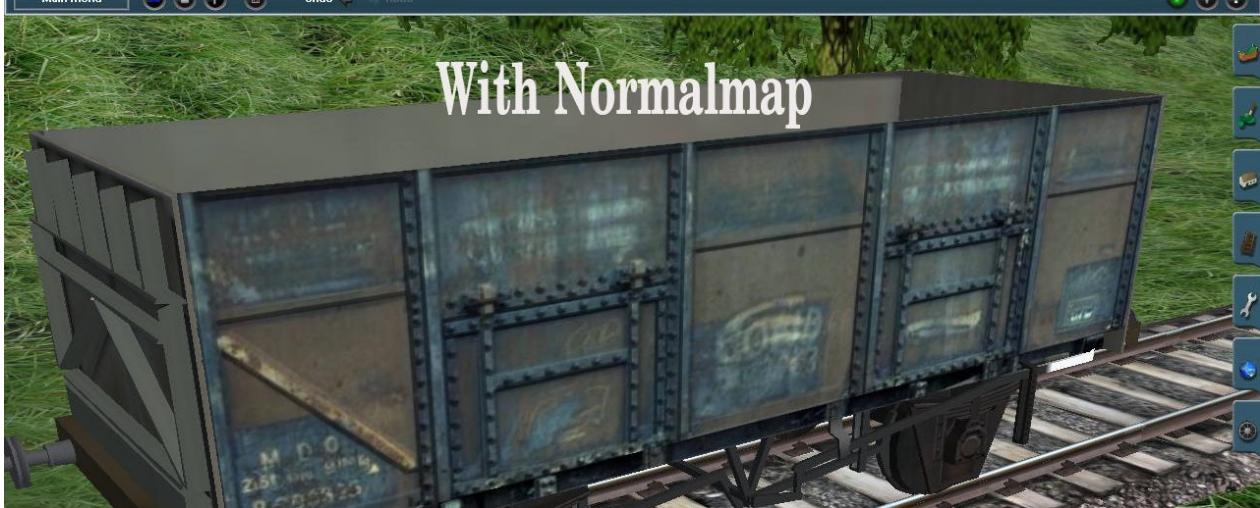
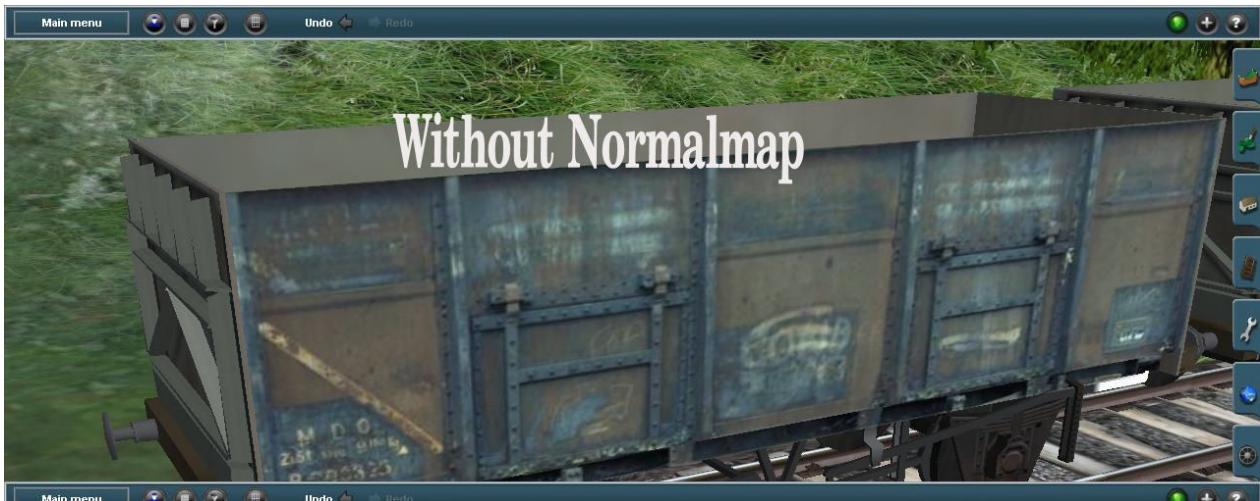
Normal Mapping is a great way to add realism to any 3D model. Recall from when we learned about the lighting calculations in OpenGL that lighting is calculated on a per vertex level. This means the closer your vertices, the better the lights look.

Normal mapping essentially encodes normal data into a texture (The R component of a pixel is the X component of the encoded normal, the G is the Y and the B is the Z). OpenGL then uses this texture with uv coordinates like normal textures, but instead of drawing the texture as a color, it uses it as a normal.

This gives us PER PIXEL normal mapping! By calculating the normals (and therefore light reflection) per pixel we can fake having detail on flat geometry.



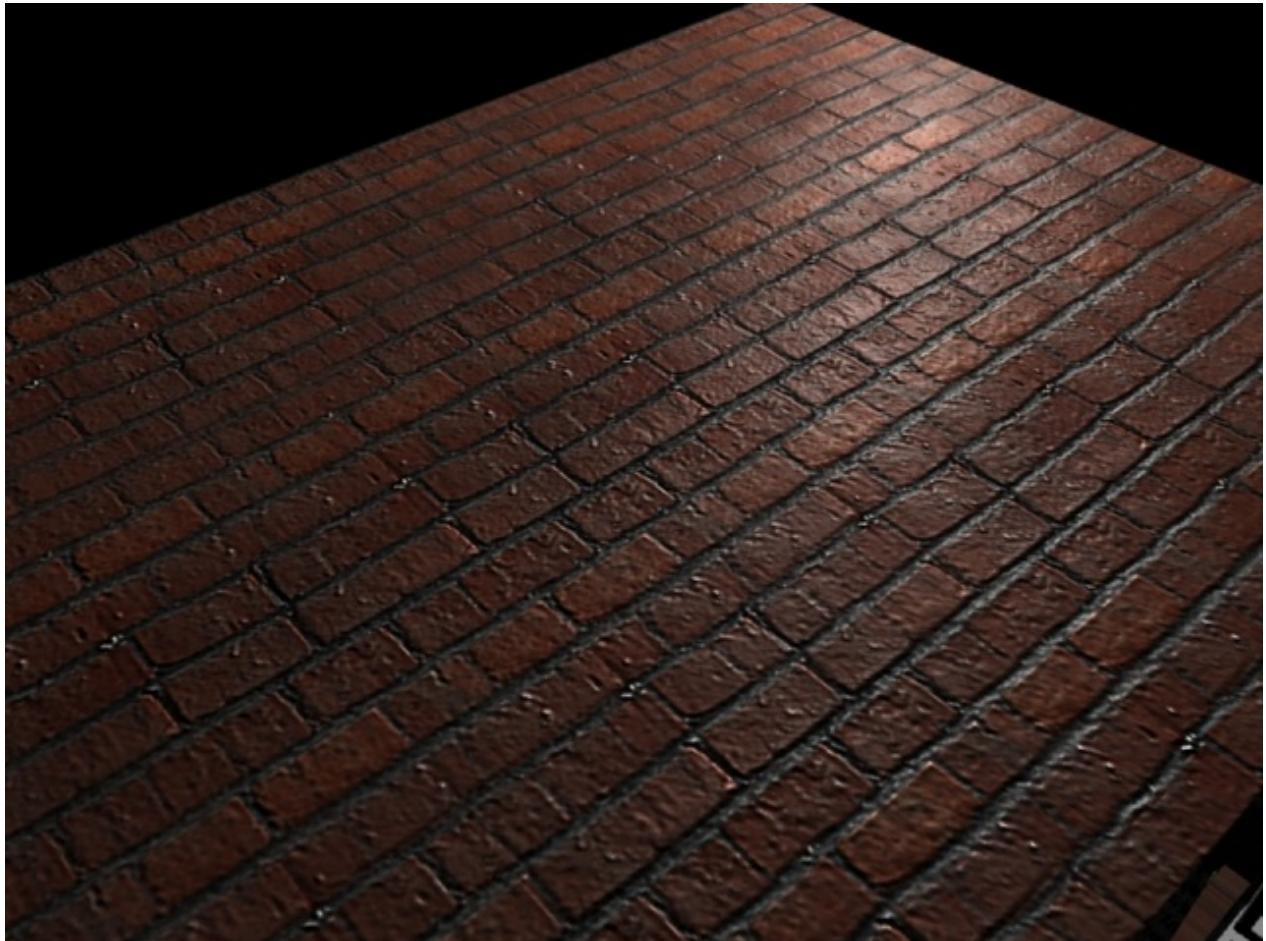
This is a method we will implement when we learn about "Modern OpenGL"



Specular Mapping

Specular Mapping is almost like normal mapping, but for the specular component. Because the specular component is just a single range value, the spec map is often added as the alpha channel of the normal map.

All specular mapping is calculating the specular reflection of light per pixel. For example, here is a flat plane with a diffuse texture, normal map and specular map. Notice how it looks a little wet. That's because of the spec map:



Ambient Occlusion

Ambient Occlusion is the small shadows that ambient lighting casts. When you're in a well lit room look at the corner of a wall. It's usually a little bit darker than the rest of the wall.

A better example is the wrinkles on a persons face. They cast subtle shadows. Because wrinkles are expressed using normal maps, there is no geometry for them to cast shadows. This is where ambient occlusion maps come into play. We basically render out all the subtle shadows, then apply them with a texture.



Ambient Occlusion = On



Ambient Occlusion = Off

Ambient maps look cool, they look like the scene is rendered into some solid dark white material. Modern games apply ambient occlusion maps to entire cities:



Outlines

Outlines are a pretty classic effect in video games. Sometimes they are core to a game's style, just look at Borderlands:



The way Borderlands does outlines is interesting, it's an old hack. First, you turn off Z-Write. Then you scale your model to 1.1 on all axis. Turn off textures and lighting, and render the model all black. Next, re-enable all the stuff that was disabled, and render the model normally at a scale of 1. The result is an outline!

To this day the above method is probably the fastest way to render an outline, but there are more precise ways to do it too. Sometimes you need precision!

If you need precise outlines you must have access to a model's triangles. You have to have a list of edge objects, an edge is the line that connects two triangles. And you need a light direction. If the dot product of one of the triangles of an edge is < 0 , but the other is > 0 then you know that edge is a part of the model's outline.

Planar Shadows

Planar shadows are the cheapest way to achieve "nice" looking shadows in games. They look OK, but they have a lot of artifacts. First, an object can not shadow it's-self. Second, the shadow is literally a plane, so if the player stands on an elevated surface, like a stump, the shadow will float.

This method was popular on the PS2 because of how cheap it is to implement performance wise. The method is simple, construct a special matrix that collapses your 3D geometry down onto a 2D plane. Render this collapsed version of the model black. This becomes the shadow. Then, render the model normally.

The extra matrix multiplication has minimal impact. The only challenging part of this method is finding the matrix that will project your 3D model onto a 2D plane in 3D space!

I don't currently have this book with me, but [OpenGL Game Programming](#) covers how to derive the matrix.

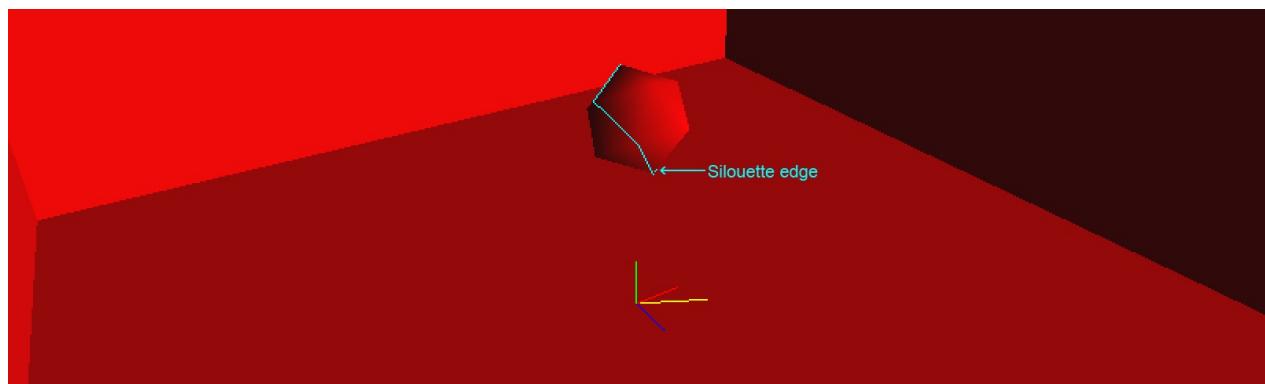
We may or may not implement this shadowing method, i'm not sure yet. It's kind of useless because of the artifacts it produces, and the next two shadowing methods we discuss both are capable of self-shadowing.

Stencyl Shadows

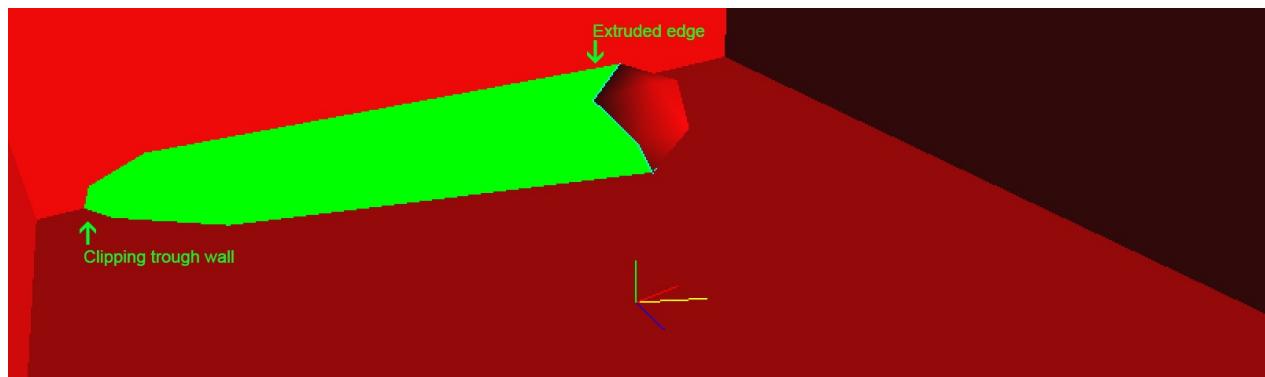
This is perhaps the crispest way to implement shadows, it was made famous by DOOM3. Take a look at the detail of their shadow tech:



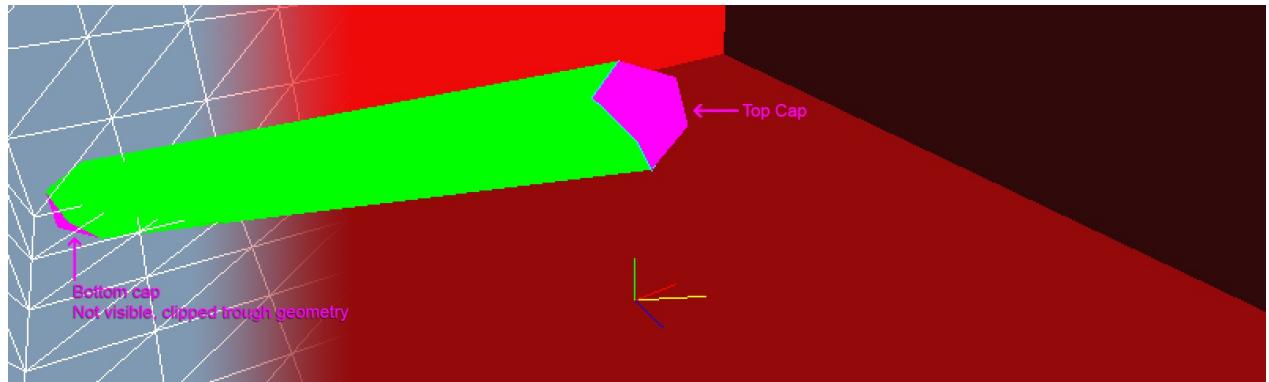
Stencil shadows are also called volume shadows. To get a stencil shadow you first find the outline of an object:



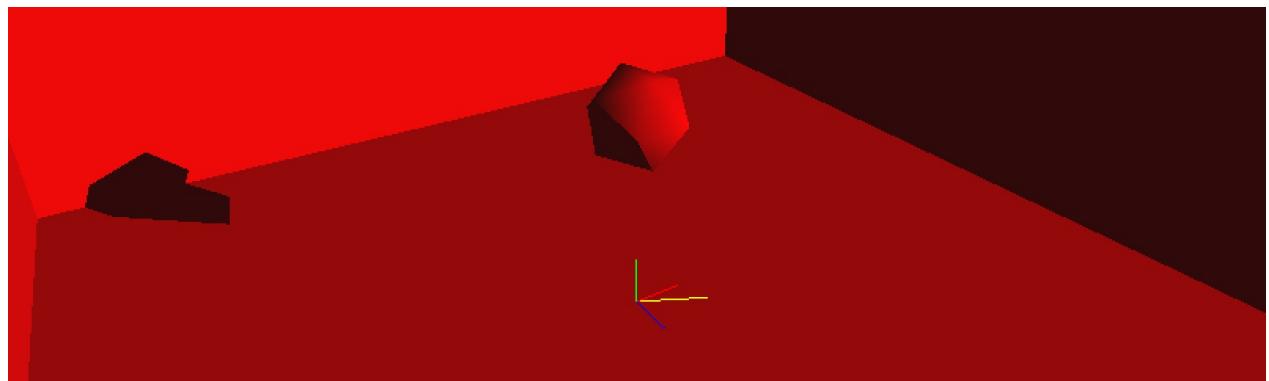
Once you have the outline, you must extrude it in the direction of the light. In theory this extrusion should be to infinity, but really it's just a really big number (50,000)



This extruded geometry is never rendered. What you do is test for intersections. Where this geometry intersects other geometry, you draw a shadow.



Above I peeled back the color to show where the shadow volume intersects the room geometry. Below is the final image. Notice how the shadow volume also covered half of the object casting the shadow, so the object is **self shadowed**



Stencyl?

So, why is this method called stencil shadows? Well, you don't actually do any intersection testing. You "Render" the scene into the stencil buffer (Not color buffer). Then you "Render" the shadow volume into the stencil buffer too.

If you render the shadow volume and it draws on-top of other geometry we know an intersection happened.

It's a pretty complex topic, it took me about 5 years to actually implement. The best part is, no games actually use this method of shadowing anymore!

Anyway, I wrote an article about it, with some sample c++ code:

<https://github.com/gszauer/GLExperiments/tree/master/ZFail>

Shadow Mapping

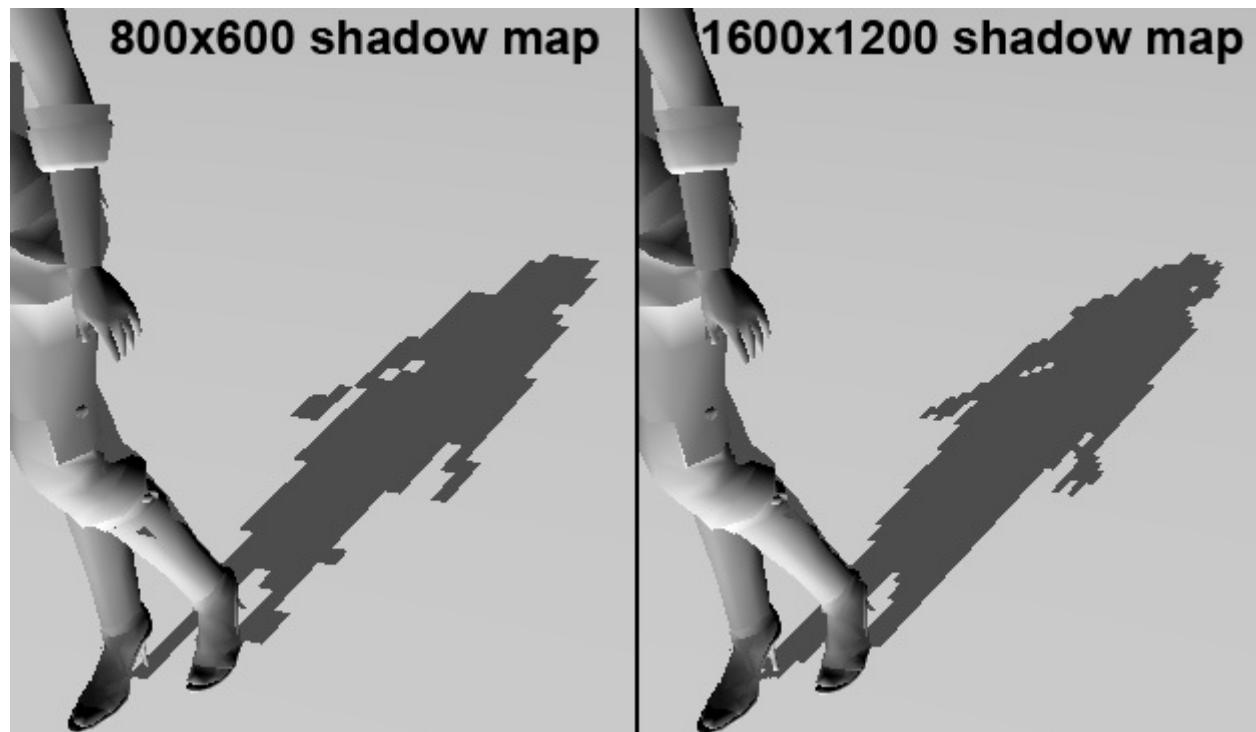
Shadow mapping is the "modern" way to render shadows. The quality of shadow mapped shadows is MUCH lower than that of Stencyl Shadows, but performance wise shadow maps are MUCH faster. We will probably implement this method of shadowing later.

The concept of a shadow map is simple, render the entire scene into an off-screen color buffer from the point of view of the light. Then render the scene normally from the point of view of the camera. If the Z value of the camera pixel being rendered is closer than the Z value of the shadow pixel, the object is in shadow.

Shadow mapping suffers from resolution issues. Because the off screen buffer is just pixels, it can alias. A low resolution shadow map looks like crap!

You can find a really good tutorial on how to do shadow mapping with OpenGL 1 (Legacy OpenGL, what we've been using) [here](#), but it becomes MUCH easier to implement with OpenGL 2.

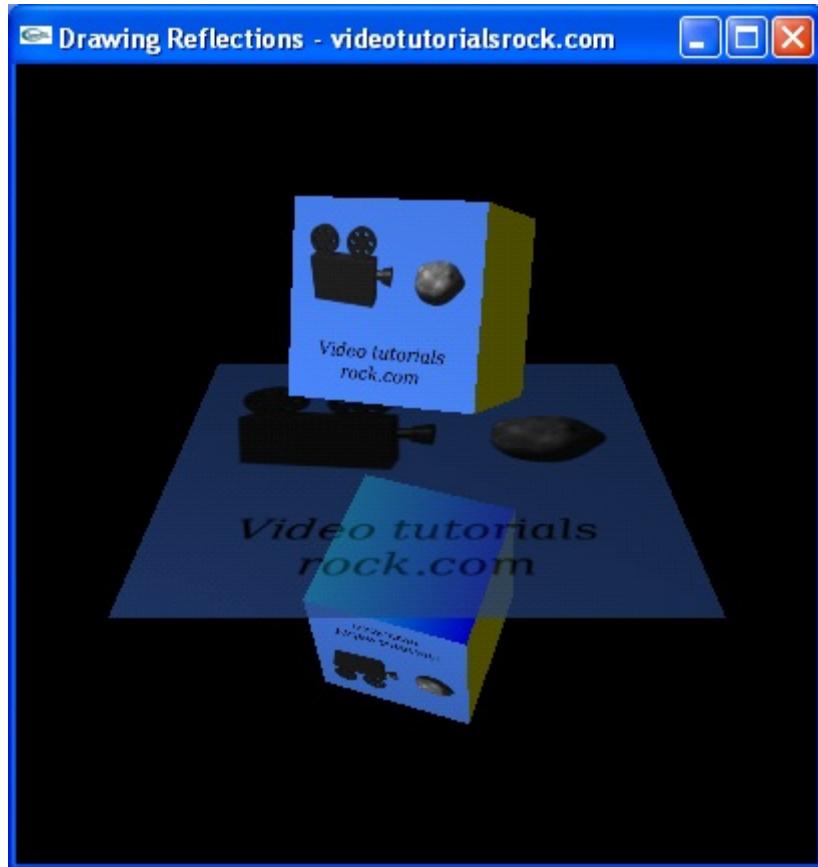
Here is an image comparing different resolution shadow maps. The quality can be awful!



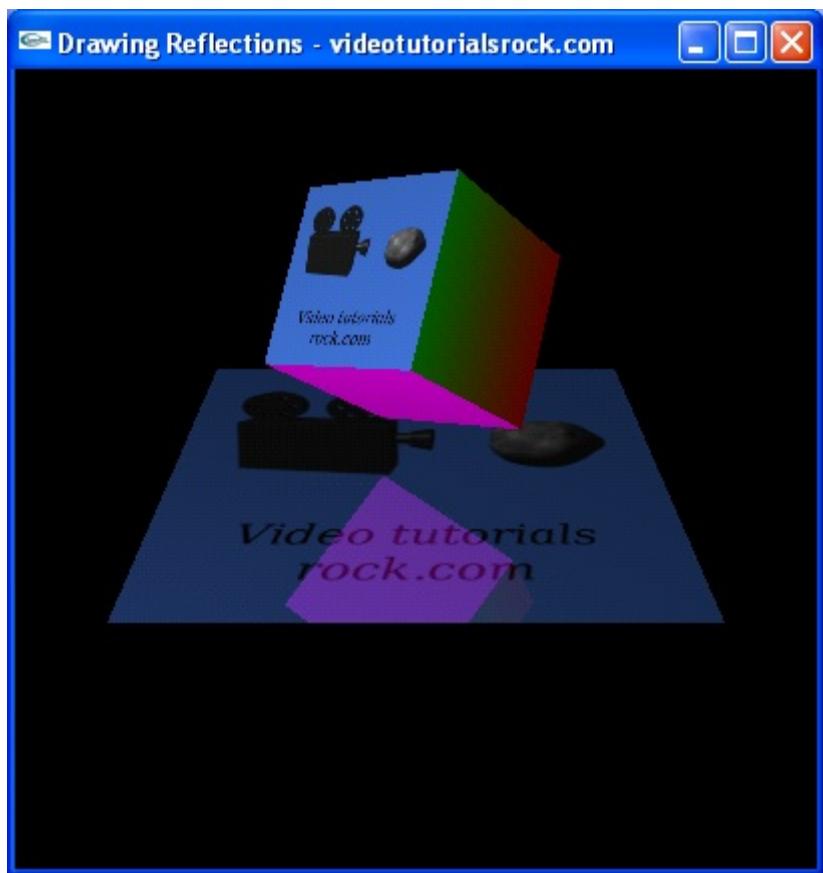
Reflections

Planar Reflections is the oldest reflection technique used in video games. It's used for things like water, glass and mirrors. Basically any surface that is somewhat of a plane. This technique will not work on complex shapes, like a car.

The technique is simple, you reflect your camera around the reflection plane and render the scene upside down:



The plane is just geometry rendered with opacity. This doesn't look great though, you can see the upside down geometry! Because of this, the Stencil buffer is used to clip the upside down image to the shape of the plane:



Cubemap reflection

Cube map reflections solve the major issue of planar reflections, complex shapes. Any racing game you play where the cars have some sort of real reflection on the paint, or any time you see a complex shape reflect something, it's a cubemap.

A cubemapped reflection will look like this:



So, how do we achieve this effect? First the scene has to be rendered 6 times from the point of view of the object, into a cube map. If you remember, when we bind textures, one of the texture targets to bind to is **CubeMap**. This is what a rendered map will look like:



Now comes the magic, when you render the object, OpenGL will take pixel color from the cube map. Imagine a cube that's just big enough to surround the object being rendered (The green outline around the tea pot). The cube map is wrapped around this cube:



Now when a pixel is rendered, OpenGL figures out what direction the light on that pixel reflects from, and casts a ray out. Wherever that ray hits the enclosing cube, that pixel is used for the color of the object.

This is a super advanced graphics method, that i've never had to implement. Mainly because engines like Unity do the hard work for you, you just have to tag an object as cube-map refected. But at some point in time i will try to implement this for fun.

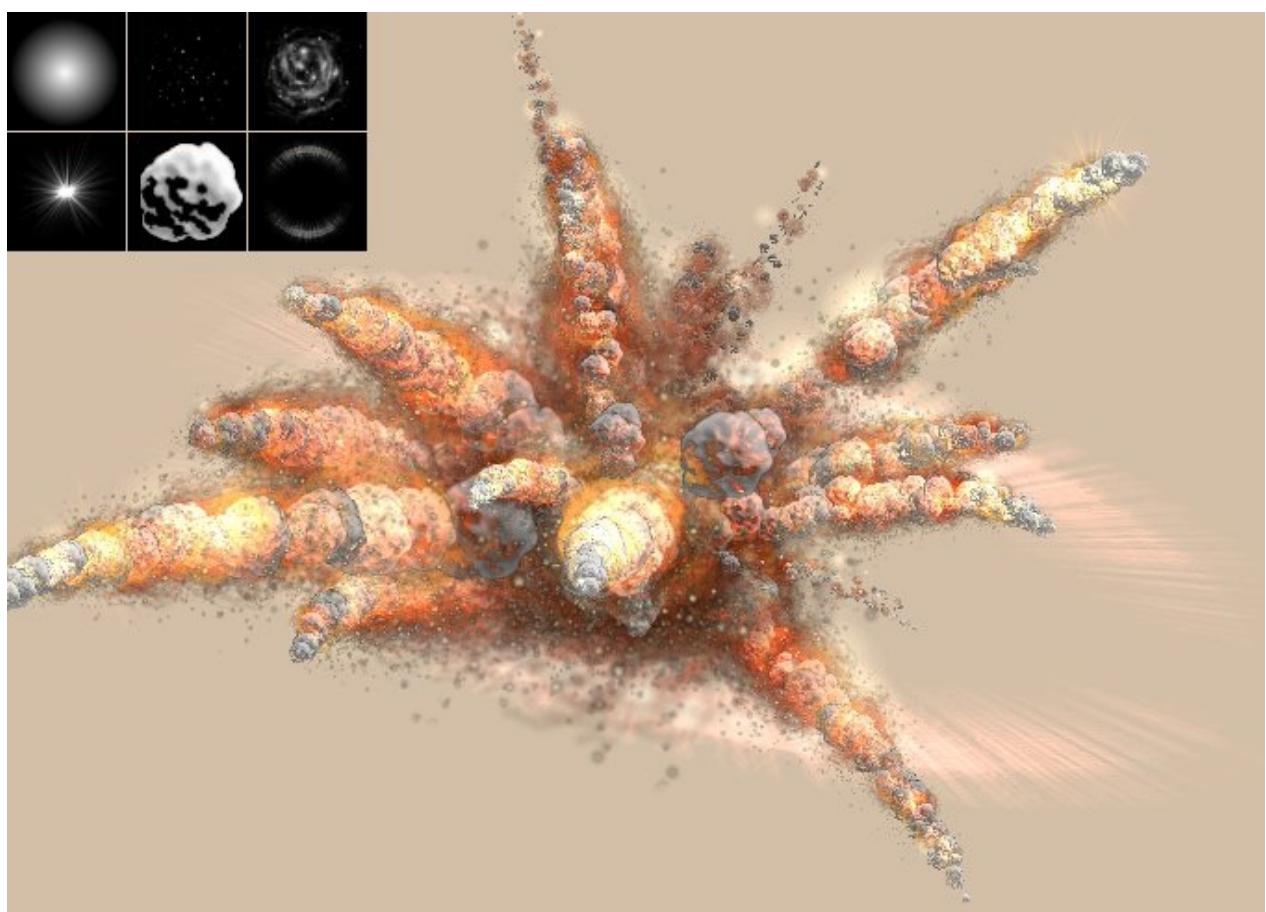
Particles

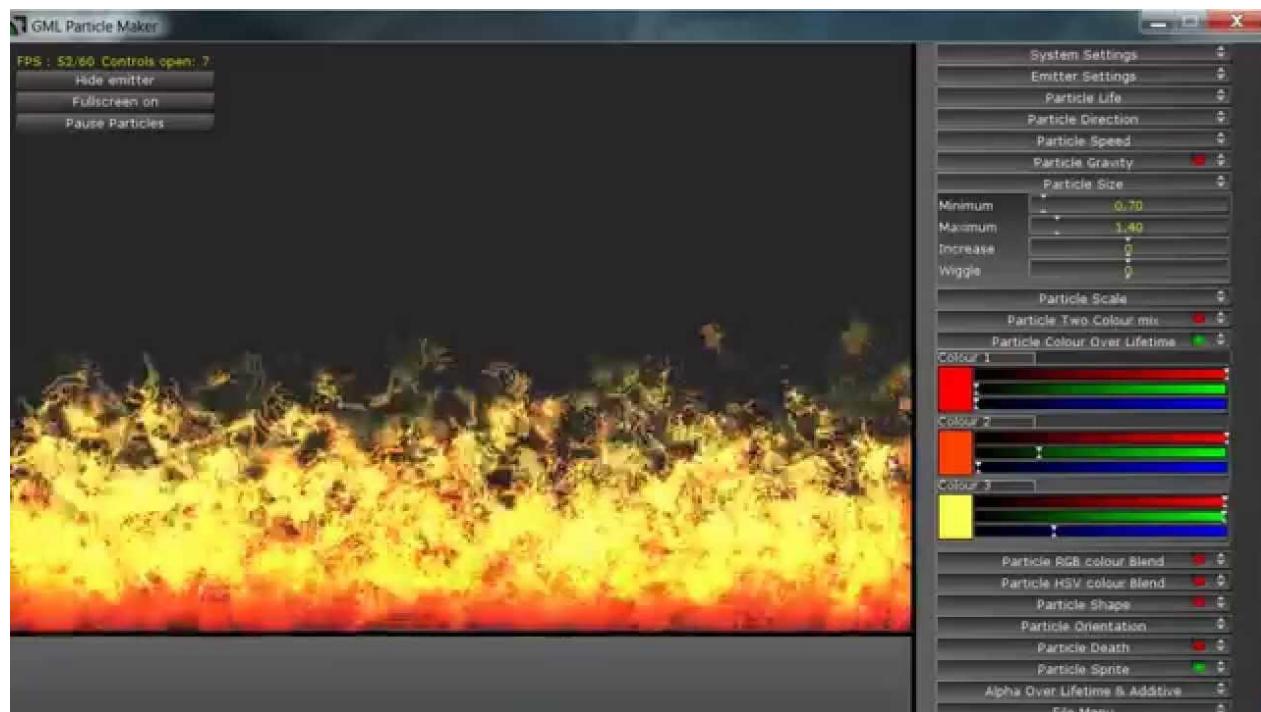
We've already built a simple particle system, but i don't think that gave us the appreciation for just how amazing particles in games are. One of the things to note is that not all particle systems emit just quads. Some emit full blown meshes. Other systems emit billboarded quads, or like our implementation just simple quads.

Particles are used to create anything from rain, fog, fire to waterfalls and the such. When you see an animation of water pouring, it's usually particles.

I'm just going to dump a bunch of images here showing different particle effects. Also, watch [the unreal 4 elemental demo](#). The particles are what really make that video!









Motion Blur

I personally find motion blur super annoying in games, but it seems like every new game that comes out uses it. So i guess this warrants a brief discussion. What does motion blur in games look like:

Applied to the character:



Applied to everything but the character



Applied to everything:



I think the only place motion blur *really* makes sense is in racing games:



The effect

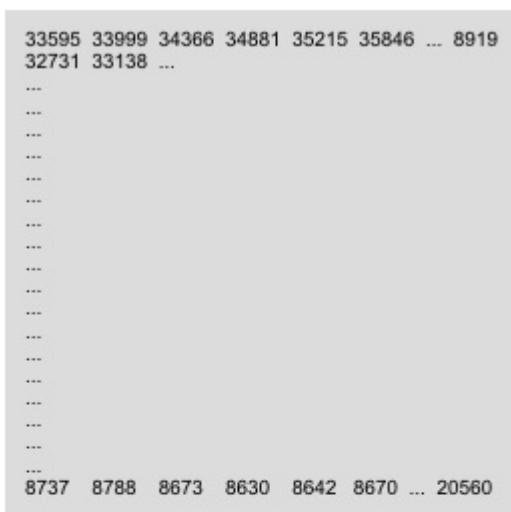
So how is motion blur achieved? OpenGL has something called an accumulation buffer. You can store the color buffer from the last N frames (N can be any number depending on your GPU support, the OpenGL specs guarantee it will at least be 3) and blend them together.

Blending the last 3 to 8 frames causes them to look blurred. You can render everything blurred, or use the stencil buffer to render some things not blurred at all. OR you could render things from your scene (Not blurred) at 80% opacity to make them look kind of blurred, but less so than the rest of the scene.

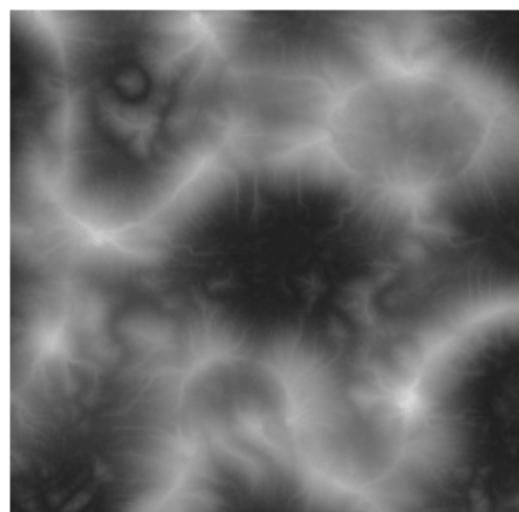
Terrain

Terrain's are created using height maps. Height maps are very popular in games. I've never had to implement a height map in my career, and i don't see that changing. Engines like Unreal or Unity have built in support for height-maps, and unless you are planning to make a game outside an engine that happens to be set out in the wilderness i don't think you'll need to implement this method.

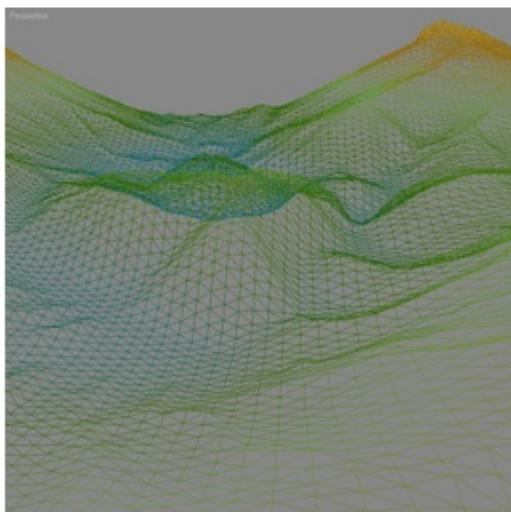
So, how does height mapping work? Well you have a black and white image. White areas represent low height and black areas represent tall height. Then you have a large, subdivided plane. You map each vertex of the plane to a pixel on the height-map. You set the Y position of each vertex to whatever the heightmap wants you to set it to.



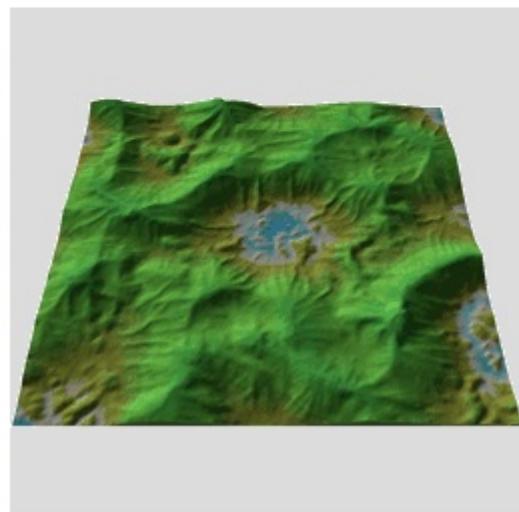
Heightmap array of altitude values



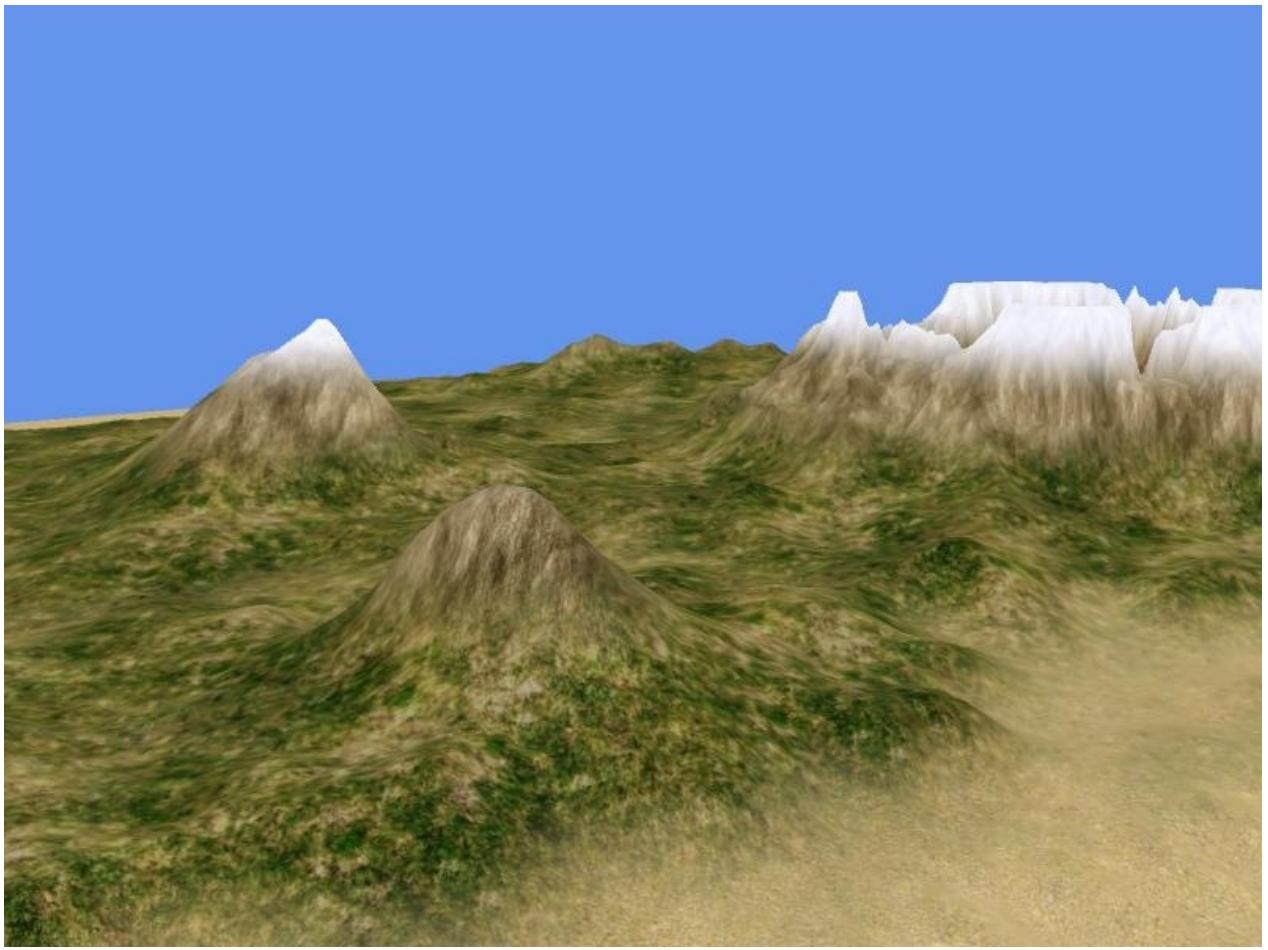
Heightmap equivalent grayscale image



Heightmap equivalent 3D mesh



3D Mesh with colors-by-altitude



As you can imagine, outdoor games like Skyrim or Far Cry use this method a lot. Indoor games have 0 use for the method.