Table of Contents

Introduction 1.1

New Repo 1.2

- Applicaiton.cs 1.2.1
- Program.cs 1.2.2
- Samples 1.2.3
  - Console 1.2.3.1
  - Windowed 1.2.3.2

Shapes And Points 1.3

- Point 1.3.1
- Sphere 1.3.2
  - Point in Sphere 1.3.2.1
  - Closest point on Sphere 1.3.2.2
- AABB 1.3.3
  - Point in AABB 1.3.3.1
  - Closest point on AABB 1.3.3.2
- Plane 1.3.4
  - Point on Plane 1.3.4.1
  - Closest point on Plane 1.3.4.2
  - 3 Plane Intersection 1.3.4.3
- Line 1.3.5
  - Point on line 1.3.5.1
  - Closes point on Line 1.3.5.2
- Ray 1.3.6
  - Point on ray 1.3.6.1
  - Closest point on Ray 1.3.6.2

Intermission 1.4
Static Intersections 1.5
Sphere-Sphere 1.5.1
Sphere-AABB 1.5.2
Sphere-Plane 1.5.3
AABB-AABB 1.5.4
AABB-Plane 1.5.5
Plane-Plane 1.5.6
Lines and Rays 1.6
Ray casting 1.6.1
Against Sphere 1.6.1.1
Against AABB 1.6.1.2
Against Plane 1.6.1.3
Line testing 1.6.2
Against Sphere 1.6.2.1
Against AABB 1.6.2.2
Against Plane 1.6.2.3
Dynamic Intersections 1.7
Time Slicing 1.7.1
Linear Impact Search 1.7.2
Binary Impact Search 1.7.3
Multiple Moving Objects 1.7.4
3D Models 1.8
Triangle 1.8.1
Point in Triangle 1.8.1.1
Closest point to triangle 1.8.1.2
Sphere-Triangle 1.8.1.3
AABB-Triangle 1.8.1.4

Plane-Triangle 1.8.1.5
Generic SAT 1.8.1.6
Triangle-Triangle 1.8.1.7
Robust SAT 1.8.1.8
Raycast Triangle 1.8.1.9
Line test Triangle 1.8.1.10
Broad-Phase Collision 1.8.2
Adding Triangles 1.8.2.1
AABB 1.8.2.2
Sphere 1.8.2.3
Collision Tests 1.8.3
Sphere 1.8.3.1
AABB 1.8.3.2
Plane 1.8.3.3
Triangle 1.8.3.4
Ray 1.8.3.5
Raycast Model 1.8.4
BVH - Node 1.8.4.1
BVH - Split 1.8.4.2
BVH - Shake 1.8.4.3
BVH - Raycast 1.8.4.4
Scene Management 1.9
Hierarchy 1.9.1
Making a scene 1.9.2
Octree 1.9.3
Octree-Node 1.9.3.1
Raycast 1.9.3.2
Picking 1.9.4

Other structures 1.9.5
Camera 1.10
Debug Info 1.10.1
Frustum 1.10.2
Point in Frustum 1.10.2.1
Sphere in Frustum 1.10.2.2
AABB in Frustum 1.10.2.3
Culling 1.10.3
Further Research 1.11
OBB 1.11.1
Capsule 1.11.2
Elipsoid 1.11.3
Cylinder 1.11.4
Physics Engines 1.11.5

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/

New Repo Time!

Now that we're done with OpenGL it's time to start a new repo on github. Let's call this one "CollisionDetection". Make the repo, clone it, add the appropriate .gitignore, commit the changes. Make a new Visual Studio solution, inside the cloned repo. The solution only needs to have 1 project in it.

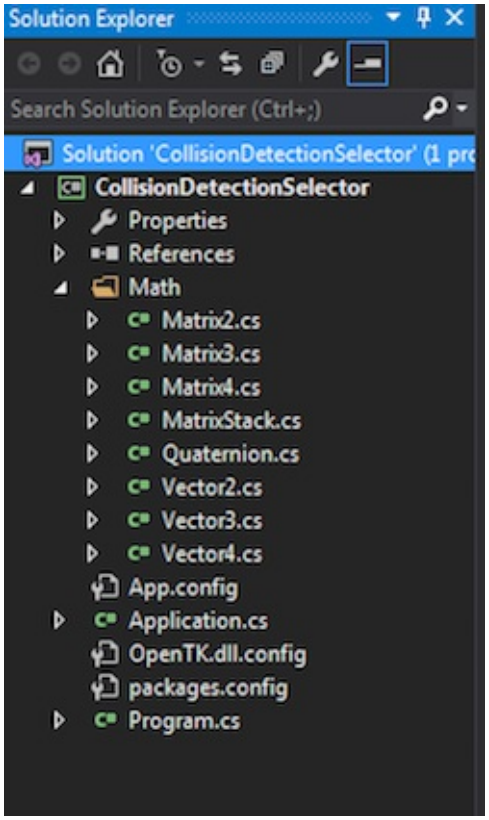
Once you have a solution and project set up, put the math libary we wrote into it. Collisions are going to be math heavy. Also set up OpenTK. Even tough we will not focus on rendering, we will want to visualize this stuff.

Sample Framework

I'll provide you with code for a compleate sample framework. The way it will work is there is a base class `Application`, you will subclass this to make your own samples. When you run the application a console will pop up, it will scan your code for all subclasses of application. Then it will ask you which one to run.

This should make running sample application super simple! **Program.cs** already exists in your application, go ahead and add a new class, name it **Application.cs**

At this point, your directory tree should resamble this, only two .cs files.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'CollisionDetectionSelector' (1 pro
CollisionDetectionSelector
Properties
References
Math
Matrix2.cs
Matrix3.cs
Matrix4.cs
MatrixStack.cs
Quaternion.cs
Vector2.cs
Vector3.cs
Vector4.cs
App.config
Application.cs
OpenTK.dll.config
packages.config
Program.cs

Application

This is going to be your **base class** for all the applications we build with the collision detection projects. All your classes will subclass this one. There are two member variables your classes will inherit:

- `Application Instance` a static instance to the active subclass of Application. However, the type is of the base class. Singleton access is convenient sometimes
- `OpenTK.GameWindow Window` this is the actual OpenTK window that the Application class launches. Chances are you won't need to do anything with this, but it's included just in case.

Extending Application

When you extend the application class, there are a few functions you will be able to override to add your own logic.

```
public virtual void Main(string[] args)
```

Main is the entry point. You will be forwarded all the default arguments from the program. By default, the base class version of this function creates the OpenTK window. If you want to make a console application, you just override `Main` and not call the base class version of it!

The next few functions are pretty self explanatory, as we've been using similar functions all along. You NEVER need to call the base versions of these functions (as they are empty). Usually if you override one or more of these, you don't need to override main.

```
public virtual void Intialize(int width, int height)
public virtual void Resize(int width, int height)
public virtual void Update(float deltaTime)
public virtual void Render()
public virtual void Shutdown()
```

The code

Here is the code for the `Application` class in it's entirety. Other than what was discussed above, the class has a constructor and a few other private OpenTK related functions.

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

// Lives in global namespace
class Application {
    public static Application Instance {
        get; private set;
    }

    public OpenTK.GameWindow Window = null;

    public Application() {
        Instance = this;
    }

    /// These are the logic driving functions that classes whom su
bclass application
    /// will need to override. There is no need to call the base v
ersions!
    #region Inheritable Logic
    public virtual void Intialize(int width, int height) {
```

```
    }

    public virtual void Resize(int width, int height) {

    }

    public virtual void Update(float deltaTime) {

    }

    public virtual void Render() {

    }

    public virtual void Shutdown() {

    }
    #endregion

    /// Entery point of the application, by default it creates a n
ew window
    /// Your subclasses should not have to override this, but if y
ou want a
    /// sample that does not have a window, you need to.
    #region Entery Point
    public virtual void Main(string[] args) {
        Window = new OpenTK.GameWindow();
        Window.Load += new EventHandler<EventArgs>(OpenTKInitializ
e);
        Window.UpdateFrame += new EventHandler<FrameEventArgs>(Ope
nTKUpdate);
        Window.RenderFrame += new EventHandler<FrameEventArgs>(Ope
nTKRender);
        Window.Unload += new EventHandler<EventArgs>(OpenTKShutdow
n);
```

```
        Window.Title = "Sample Application";
        Window.ClientSize = new System.Drawing.Size(800, 600);
        Instance.Resize(800, 600);
        Window.VSync = VSyncMode.On;
        Window.Run(60.0f);

        Window.Dispose();
    }
    #endregion

    /// If the default entery point (Main function) is used, these
 are the OpenTK callbacks
    /// Which are issued, they in turn call member functions that
subclasses need to override
    #region OpenTK Callbacks
    private void OpenTKInitialize(object sender, EventArgs e) {
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.DarkGray;
        Console.WriteLine("OpenGL Vendor: " + GL.GetString(StringN
ame.Vendor));
        Console.WriteLine("OpenGL Renderer: " + GL.GetString(Strin
gName.Renderer));
        Console.WriteLine("OpenGL Version: " + GL.GetString(String
Name.Version));
        Console.ResetColor();
        Instance.Intialize(800, 600);
    }

    private void OpenTKUpdate(object sender, FrameEventArgs e) {
        float dTime = (float)e.Time;
        Instance.Update(dTime);
    }

    private void OpenTKRender(object sender, FrameEventArgs e) {
        GL.ClearColor(Color.CadetBlue);
        GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.
DepthBufferBit | ClearBufferMask.StencilBufferBit);
```

```
        Instance.Render();
        Window.SwapBuffers();
    }

    private void OpenTKShutdown(object sender, EventArgs e) {
        Instance.Shutdown();
    }

    private void OpenTKResize(EventArgs e) {
        Instance.Resize(Instance.Window.Width, Instance.Window.Hei
ght);
    }
    #endregion
}
```

Program

The nice thing about this `Program` class is that it's automated. It scans your executable (assembly) for subclasses of `Application`, and dynamically creates a selected subclass of `Application`.

This magic is done trough reflection, using the built-in `Assembly.GetTypes` method. Because this class will never be edited, i'm just going to give you the code listing. If you are interested in how it works, ask me or do some google-ing on any code you don't recognize.

You select the subclass of `Application` based on class name of the subclasses!

The code

The one thing we do here that we haven't done before is that the `Program` class is not in a specific namespace! This is not an error. C# has an anonymous global namespace. If you fail to specify a namespace for a class, it gets put into this global namespace.

Both `Program` and `Application` live in the global namespace.

```
using System.Linq;

// Lives in global namespace
class Program {
    static void Main(string[] args) {
        System.Type[] samples = FindAndPrintApplicationChildClasse
s();
        System.Type selectedSampleType = null;

        if (samples == null || samples.Length == 0) {
            System.Console.WriteLine("No samples found, press any
key to quit");
```

```
            System.Console.ReadKey();
            return;
        }

        if (args.Length > 0) {
            selectedSampleType = ParseSampleFromNumber(samples, ar
gs[0]);
        }

        System.Console.WriteLine("Select a sample, or q to quit");
        while (selectedSampleType == null) {
            System.Console.Write("Enter a sample number: ", System
.ConsoleColor.White);
            string input = System.Console.ReadLine();
            if (!string.IsNullOrEmpty(input) && input.ToLower()[0]
 == 'q') {
                return;
            }
            selectedSampleType = ParseSampleFromNumber(samples, in
put);
        }

        Application game = (Application)System.Activator.CreateIns
tance(selectedSampleType);
        game.Main(args);
    }

    static System.Type[] FindAndPrintApplicationChildClasses() {
        System.Type[] samples = typeof(Application).Assembly.GetTy
pes().Where(t => t.IsSubclassOf(typeof(Application)) && t != typeof
(Application)).ToArray();
        int blockSize = samples.Length / 2;
        for (int i = 0; i < blockSize; i++) {
            System.Console.WriteLine(string.Format("{2,2}. {0,-30}
{3,2}. {1,-30}", samples[i].Name, samples[i + blockSize].Name, i,
i + blockSize), System.ConsoleColor.DarkGray);
```

```
        }
        if (blockSize * 2 < samples.Length) {
            System.Console.WriteLine($"{samples.Length - 1,36}. {s
amples[samples.Length - 1].Name}", System.ConsoleColor.DarkGray);
        }
        if (samples.Length != 0) {
            System.Console.WriteLine();
        }

        return samples;
    }

    static System.Type ParseSampleFromNumber(System.Type[] samples
, string input) {
        int number;
        if (!int.TryParse(input, out number)) {
            System.Console.WriteLine("Invalid format.", System.Con
soleColor.Red);
            return null;
        }

        if (number >= samples.Length || number < 0) {
            System.Console.WriteLine("Invalid number.", System.Con
soleColor.Red);
            return null;
        }

        return samples[number];
    }
}
```

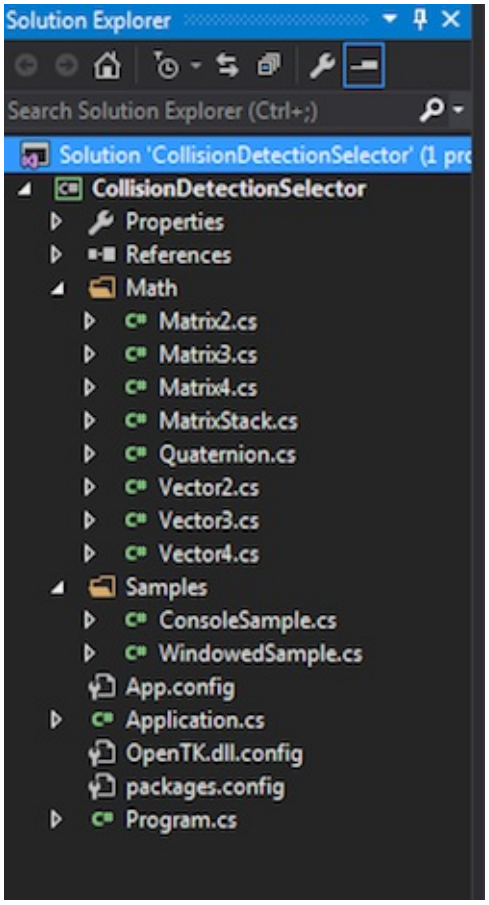
Samples

Let's go trough and implement two samples real quick. This should give you a general idea of how to make new samples. We're also going to be creating all these in a folder to keep our project orgonized.

- First, add a new folder, lets call it **Samples**
- Add a new file **ConsoleSample.cs**
- Add a new file **WindowedSample.cs**

From now on, all new samples will be made in this folder. ONLY put classes that create a window (or console) in here. No helper classes! We are going to make lots of classes that are just objects, don't put them in here. If you want you can create a new "Collision" directory for those, or just put them into the root, next to Program.cs; but keep the "Samples" directory clean!

The project layout should look like this:


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'CollisionDetectionSelector' (1 pro
CollisionDetectionSelector
Properties
References
Math
Matrix2.cs
Matrix3.cs
Matrix4.cs
MatrixStack.cs
Quaternion.cs
Vector2.cs
Vector3.cs
Vector4.cs
Samples
ConsoleSample.cs
WindowedSample.cs
App.config
Application.cs
OpenTK.dll.config
packages.config
Program.cs

Console

Making a new console application is pretty easy. All you have to to is subclass `Application` and override the `Main` method. When you do this, make sure that you don't call `base.Main`! `base.Main` is what would otherwise create the OpenTK window!

```
using System;

namespace CollisionDetectionSelector.Samples {
    class ConsoleSample : Application {
        public override void Main(string[] args) {
            Console.WriteLine("Testing, testing");
            Console.WriteLine("No window should pop up");
            Console.WriteLine("Press any key to quit");
            Console.ReadKey();
        }
    }
}
```

Window

Making a windowed sample is as easy as making a console one. If you want a window, you probably don't need to override the `Main` method, but you will need to override one or more of the following:

- Intialize(int width, int height)
- Resize(int width, int height)
- Update(float deltaTime)
- Render()
- Shutdown()

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

namespace CollisionDetectionSelector.Samples {
    class WindowedSample : Application {
        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, 0
.0f));
            GL.LoadMatrix(lookAt.OpenGL);

            GL.ShadeModel(ShadingModel.Smooth);

            GL.Begin(PrimitiveType.Triangles);
            GL.Color3(1.0f, 0.0f, 0.0f);
            GL.Vertex3(-10.0f, -10.0f, -5.0f);//red
            GL.Color3(0.0f, 1.0f, 0.0f);
            GL.Vertex3(20.0f, -10.0f, -5.0f);//green
```

```
            GL.Color3(0.0f, 0.0f, 1.0f);
            GL.Vertex3(-10.0f, 20.0f, -5.0f);//blue
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Run it!

After having created both samples, run your application. You can select which one to run based on the name of the class.

Shapes & Points

We're going to start out by implementing some basic shapes (We're going to make classes for each shape) and methods to test if a point is inside any of the shapes. This may seem simple but it's the basis of collision detection. In later chapters we're going to add functionality to our existing shape classes.

You SHOULD NOT out put primitives in the **Samples** directory. Remember, only code that creates a window (or console) goes in sample. You might want to consider making a "Collision" directory or something.

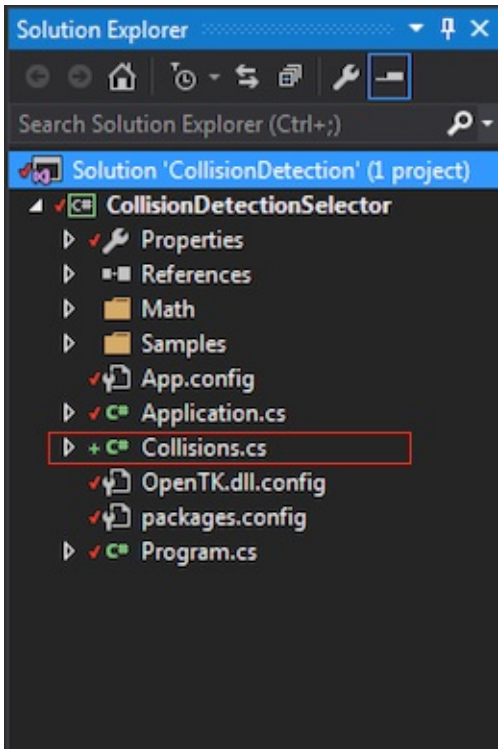
Collision class

Aside from shapes and points, we're going to make a helper class called `Collision`. This class will hold the actual collision logic. It's going to contain only public static methods. This is an example of how collisions will work when we're done:

```
// THIS IS JUST SAMPLE CODE, DON'T COPY IT!

// Integrate function causes moving sphere to rest on plane
void Integrate(Sphere sphere, Plane plane, float velocity) {
    // CollisionResult is a helper class, it contains information
    // about a collision that could happen
    CollisionResult collision = Collision.MovingSpherePlane(sphere
, plane, velocity);
    if (collision.happened) {
        // Collision happened, offset the sphere to be above the p
lane
        sphere.position = collision.point - collision.incident;
    }
}
```

Let's Get Started!

- Add a new `Collision.cs` file to the project
- For now, this class is going to be empty



And let's just add the skeleton, like `Program` and `Application` this class is going to live in global scope.

```
using System;
using Math_Implementation;

class Collisions {

}
```

Getting Started

Troughout this course, i'm going to be providing you with skeleton code for shape classes. **THIS CODE WILL NOT COMPILE**. You don't even have to use my code for implementation! The code is just there to give you an idea of what a class needs to have. It's pseudo-code.

I will render it for you. The difference being the `Render` function. Because this course is about collisions, not rendering, i will provide the rendering code for each primitive in full.

I will also try to provide unit-tests or sample applications for everything **THIS CODE WILL COMPILE**, just copy / paste.

THERE IS NOTHING TO CODE UNTIL THE ON YOUR OWN SECTION

And that's going to stay that way troughout the rest of these chapters. Just read and digest the information until you get to the *On Your Own* section. That's the only place where you need to implement stuff.

Point

The point is the simplest 3D primitive we will use. That makes it a good primitive to start with. A point is just a 3D touple, a collection of 3 numbers. Therefore, it makes sense to use a Vector3 to represent a point.

Remember, this layout is to give you an idea of WHAT the class is and should do. You don't HAVE to implement the class like this, just be sure to understand what the clas does.

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;

class Point {
  protected Vector3 position = new Vector3();

  public Vector3 Position {
    get; // TODO (Make new vector)
    set; // TODO (Make new vector)
  }

  public float X { get; set; } // TODO: get, set
  public float Y { get; set; } // TODO: get, set
  public float Z { get; set; } // TODO: get, set

  public Point(); // TODO
  public Point(float x, float y, float z); // TODO
  public Point(Vector3 v); // TODO (Make new)

  public Vector3 ToVector(); // TODO (Return new)
  public void FromVector(Vector3 v); // TODO (Make new)

#region Rendering
  public void Render() {
      GL.Begin(PrimitiveType.Points);
      GL.Vertex3(position.X, position.Y, position.Z);
      GL.End();
  }

  public override string ToString() {
    return "(" + X + ", " + Y + ", " + Z + ")";
  }
#endregion
}
```

The actual Vector3 that is the position is protected. All getters and setters should more often than not need to make a new internal vector. I've noted in comments which ones i think need to do this? Why? Let's take a look at the constructor that takes a vector.

Let's say we implement this constructor wrong, without creating a new vecotr:

```
public Point(Vector3 v) {
    position = v;
}
```

Now if we run this code:

```
void DoPointUnitTest() {
    Vector3 v = new Vector3(2, 3, 4);
    Point unitPoint1 = v;
    Console.WriteLine("p1: " + unitPoint1.ToString());
    v.X = 7;
    Point unitPoint2 = v;
    Console.WriteLine("p1: " + unitPoint1.ToString());
    Console.WriteLine("p2: " + unitPoint2.ToString());
}
```

The output of the unit test will be:

```
p1: (2, 3, 4)
p1: (7, 3, 4)
p2: (7, 3, 4)
```

As you can see, the X value of p1 would get changed. This is wrong. It happens because by doing a simple assignment like that, we're setting the `position` vector as a reference to the `v` argument. Essentially both variables point to the same vector!

Instead the constructor needs to create a NEW vector, using the values of `v`

```
public Point(Vector3 v) {
    position = new Vector3(v.X, v.Y, v.Z);
}
```

This works better. The vector that `position` is assigned to was made right there on the spot. If we run the same unit test as above, the new output will be:

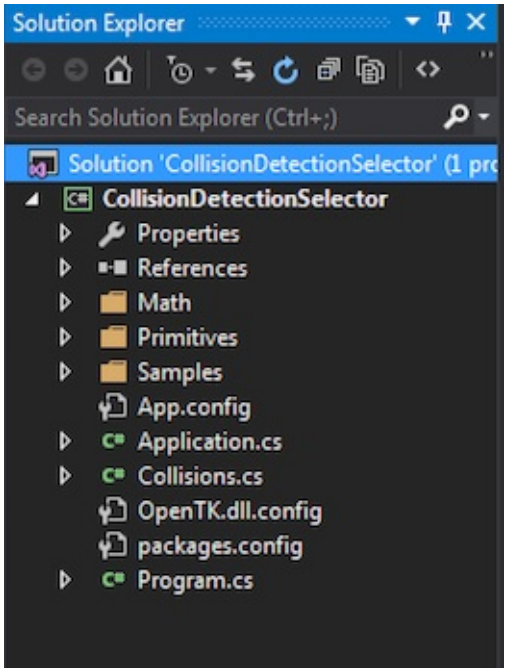
```
p1: (2, 3, 4)
p1: (2, 3, 4)
p2: (7, 3, 4)
```

And that is what we'd expect. The takeaway here is to ALWAYS pay attention to what you're assigning, a reference, a new object or a struct. And know what ramifications that assignement might have! When can you get away with a reference assign, and when do you need to do a deep copy that duplicates data into a new object. It's a lot to think about!

On Your own

Go ahead and implement the `Point` class on your own. There isn't much to do, just implement the stubs (Or figure out how you would implement it and do that, like i said you don't have to follow my pseudo-code i give, but you certainly can).

I made all my geometry classes under a geometry folder, to keep the project clean:



I'm going to be putting all geometry (Point, Sphere, AABB, Plane, etc...) under the Primitives folder. You can orgonize your project however you want, but i think this is the cleanest way of doing things.

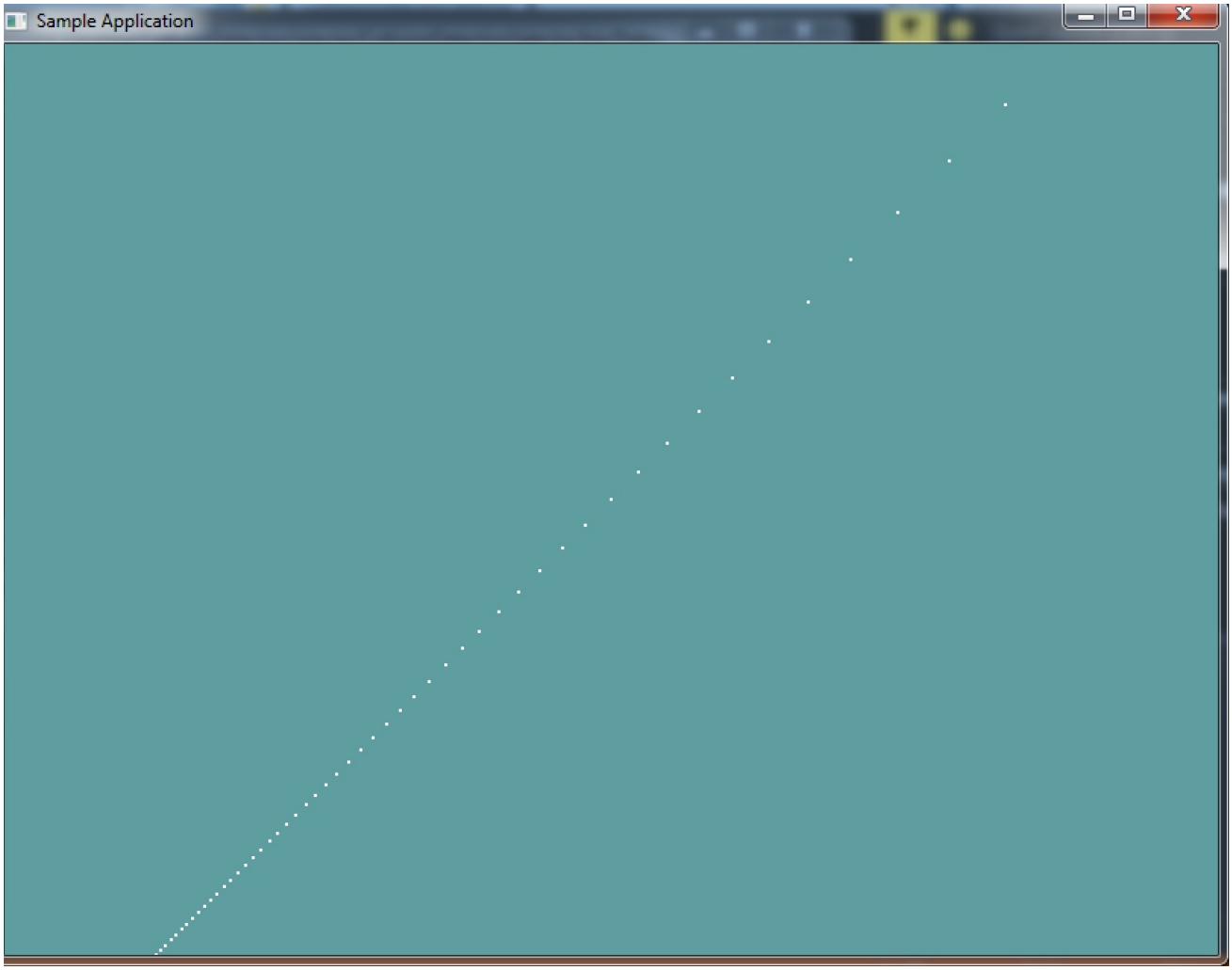
Sample / Unit Test

For every section i'll try to provide a sample application, or a unit test. For Points, make a new sample application (I called mine **PointSample.cs** and add the following code to it:

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class PointSample : Application {
        Point[] points = null;

        public override void Intialize(int width, int height) {
            points = new Point[100];
            for (int i = 0; i < 100; ++i) {
```

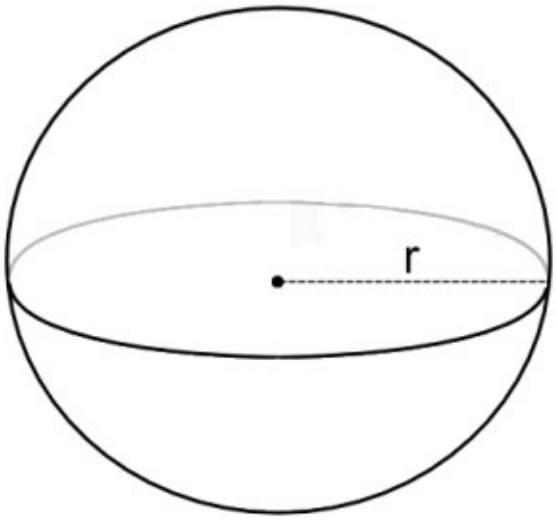
```
                points[i] = new Point(i -50, i - 50, i - 50 );
            }
            GL.PointSize(2f);
        }

        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, 0
.0f));
            GL.LoadMatrix(lookAt.OpenGL);

            for (int i = 0; i < points.Length; ++i) {
                points[i].Render();
            }
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

If you implemented your `Point` class well, the resulting sample window should look like this:


Sample Application

Sphere

Other than a point, the next most basic 3D shape. A sphere consists of a point and a radius. The radius of a sphere goes half-way accross it. So, a sphere with a radius of 1, will have a diameter of 2. This can be counter-intuitive when modelling a world, but keep it in mind!



Because we're not going to actually expose a lot of the data underneath the sphere, i'm going to store the point as a `Vector3`, and the radius as a `float`, but you could store the point as a `Point`

Based on the above information, and the image, the simplest implementation of a sphere would be:

```
// THIS BLOCK IS JUST SAMPLE CODE, DON'T COPY IT!
class Sphere { // Could be a struct?
  public Vector3 position; // Could be 3 floats?
  public float radius = 1f;
}
```

Code Guide

Everything in and after the **Rendering** region will compile. The code above it is more of a suggestion than a hard guide, tough i do think this is good design. The `CreateRenderArrays` function **MUST** be called in **EVERY** constructor.

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

class Sphere {
    Vector3 point = new Vector3();
    float radius = 1f;

    public Point Position { /* TODO */ } // Gets / Sets point vect
or
    public float Radius { /* TODO */} // Self explanatory

    public Sphere() {
        CreateRenderArrays(); // Mush call this to create Arrays
    }

    public Sphere(Point p, float r) {
        // TODO: Set member variables
        CreateRenderArrays(); // Mush call this to create Arrays
    }

    public Sphere(Vector3 p, float r) {
        // TODO: Set member variables
        CreateRenderArrays(); // Mush call this to create Arrays
    }

    public Sphere(float x, float y, float z, float r) {
        // TODO: Set member variables
        CreateRenderArrays(); // Mush call this to create Arrays
    }

#region Rendering
    private float[] verts = null;
```

```
    private float[] norms = null;
    private uint[] indices = null;

    private void CreateVBO(uint rings = 10, uint sectors = 15) {
        // From:
        // http://stackoverflow.com/questions/5988686/creating-a-3
d-sphere-in-opengl-using-visual-c/5989676#5989676
        // http://stackoverflow.com/questions/7957254/connecting-s
phere-vertices-opengl
        float R = 1f/ (float)(rings - 1);
        float S = 1f/ (float)(sectors - 1);
        float M_PI = 3.14159265358979323846f;
        float M_PI_2 = 1.57079632679489661923f;

        verts = new float[rings * sectors * 3];
        norms= new float[rings * sectors * 3];
        indices = new uint[rings * sectors * 4];

        int v = 0;
        int n = 0;
        int i = 0;

        for (int r = 0; r < rings; r++) {
            for (int s = 0; s < sectors; s++) {
                float y = (float)Math.Sin(-M_PI_2 + M_PI * r * R);
                float x = (float)Math.Cos(2f * M_PI * s * S) * (fl
oat)Math.Sin(M_PI * r * R);
                float z = (float)Math.Sin(2f * M_PI * s * S) * (fl
oat)Math.Sin(M_PI * r * R);

                verts[v++] = (x /* * radius*/);
                verts[v++] = (y /* * radius*/);
                verts[v++] = (z /* * radius*/);

                norms[n++] = (x);
                norms[n++] = (y);
                norms[n++] = (z);
```

```
            }
        }

        if (v != verts.Length) {
            Console.WriteLine("ERROR, Wrong number of verts!");
        }
        if (n != norms.Length) {
            Console.WriteLine("ERROR, Wrong number of norms!");
        }

        for (int r = 0; r < rings - 1; r++) {
            for (int s = 0; s < sectors - 1; s++) {
                indices[i++] = ((uint)(r * sectors + s));
                indices[i++] = ((uint)(r * sectors + (s + 1)));
                indices[i++] = ((uint)((r + 1) * sectors + (s + 1)
));
                indices[i++] = ((uint)((r + 1) * sectors + s));
            }
        }

        if (i != indices.Length) {
            Console.WriteLine("ERROR, Wrong number of indices!");
        }
    }

    public void Render() {
        GL.PushMatrix();
        GL.Translate(point.X, point.Y, point.Z);
        GL.Scale(radius, radius, radius);

        GL.EnableClientState(ArrayCap.VertexArray);
        GL.EnableClientState(ArrayCap.NormalArray);

        GL.VertexPointer(3, VertexPointerType.Float, 0, verts);
        GL.NormalPointer(NormalPointerType.Float, 0, norms);
        GL.DrawElements(PrimitiveType.Quads, indices.Length, DrawE
lementsType.UnsignedInt, indices);
```

```
        GL.DisableClientState(ArrayCap.VertexArray);
        GL.DisableClientState(ArrayCap.NormalArray);

        GL.PopMatrix();
    }
#endregion

    public override string ToString() {
        return "Position: (" + point.X + ", " + point.Y + ", " + p
oint.Z + "), Radius: " + radius;
    }
}
```

On Your Own

Implement a sphere class. Feel free to use the above code as a guide, or come up with your own implementeation.

Sample / Unit Test

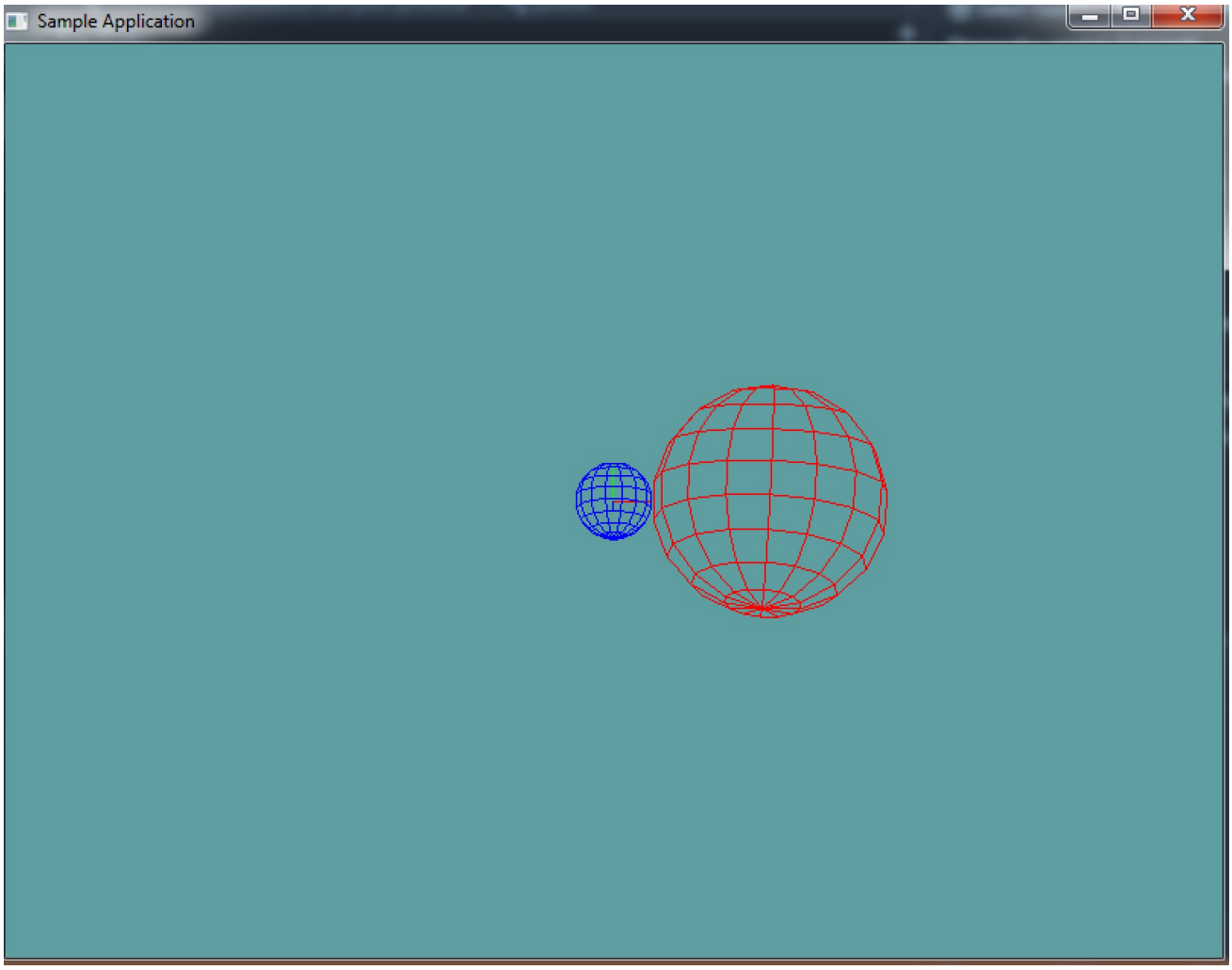
You can Download the samples for this chapter to see if your result looks like the unit test.

This sample is short and simple, it makes a new sphere at origin, configures some render settings and draws the sphere. There is an image of what it's supposed to look like after the code.

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
// Include the namespace where your primitives live!

namespace CollisionDetectionSelector.Samples {
    class SphereSample : Application{
        Sphere sphere = new Sphere(new Point(4, 0, 0), 3);
```

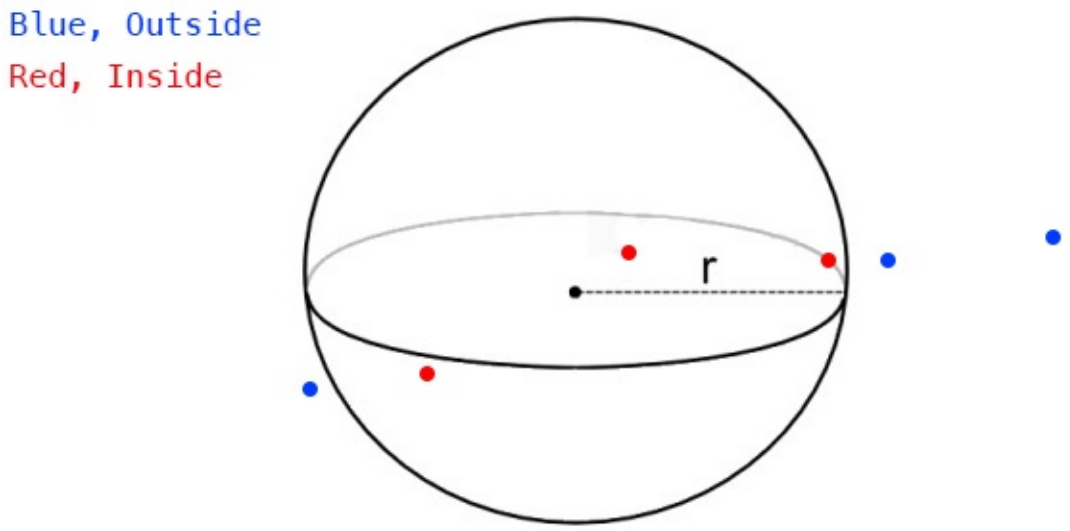
```
        Sphere sphere2 = new Sphere();

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
        }

        public override void Render() {
            Matrix4 lookAt = Matrix4.LookAt(new Vector3(0.0f, 5.0f
, 20), new Vector3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f
));
            GL.LoadMatrix(lookAt.OpenGL);

            GL.Begin(PrimitiveType.Lines);
            Line(1f, 0f, 0f);
            Line(0f, 1f, 0f);
            Line(0f, 0f, 1f);
            GL.End();

            GL.Color3(0f, 0f, 1f);
            sphere2.Render();
            GL.Color3(1f, 0f, 0f);
            sphere.Render();
        }

        void Line(float x, float y, float z) {
            GL.Color3(x, y, z);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(x, y, z);
        }

        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, aspect,
```

```
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```



Point in Sphere

Testing if a point is inside a sphere is one of the simplest 3D tests. Take a look at this image, and see if you can figure out the algorithm to perform the test.



The algorithm

The actual algorithm is very simple. If the distance from the point to the center of the sphere is less than the radius of the sphere, the sphere contains the point!

Prototype

You could achieve this by accessing the position of the sphere and point being tested as vectors. Subtract one vector from the other (order doesn't matter) and take the length of the resulting vector.

```
// THIS BLOCK IS JUST SAMPLE CODE, DON'T COPY IT!
function PointInSphere(Vector3 spherePos, float sphereRad, Vector3
 point) {
  Vector3 difference = spherePos - point;
  float distance = Vector3.Length(difference);
  return distance < sphereRad;
}
```

That's a start, but it's a very sub-optimal way to do things. Remember, `Vector3.Length` requires a square root. Which is a relativley expensive operation! Before reading the next paragraph (because i will answer this question in it), how could you avoid doing a SquareRoot?

The answer is `LengthSquared`! It's a function we implemented that does the same thing as `Length`, minus the square root. If you are going to check the squared length, all you have to do is bring the sphere radius into square space as well. Like so:

```
// THIS BLOCK IS JUST SAMPLE CODE, DON'T COPY IT!
...
  float distanceSq = Vector3.LengthSquared(difference);
  return distanceSq < sphereRad * sphereRad;
```

That, is the most optimal way to check if a sphere contains a point.

But Why?

Why do we care if a sphere contains a point? Or a bounding box contains a point? Or anything really? Triggers! You've played games where after running down an alley a cutscene triggers. Or when you enter an enemy area all the enemies attack. Perhaps running accross a bridge caused it to collapse behind you, forcing you to move forward?

How do they know you entered the enemy area? Or that you got far enough down an alley to show the cutscene? There is an invisible sphere (or box, or other geometry), they take the point in space represented by the characters world matrix and test it agains the invisible geometry.

If the invisble geometry contains the point, it is deleted (to keep from triggering the same event twice) and the cutscene is spawned, or the enemy AI is alerted to your presence.

Of course this is a small, trivial example of where a point containment test can help, there are HUNDREDS of uses for this simple technique.

On Your Own

We're going to keep geometry implemented as mostly data. The collision functions don't really belong inside the geometry classes.

Go ahead and implement a static `PointInSphere` function inside the `Collisions` class:

```
using System;
using Math_Implementation;
// If your geometry is in a namespace, pull it in

class Collisions {
    public static bool PointInSphere(Sphere sphere, Point point) {
        // TODO
        return false;
    }
}
```

Unit Test

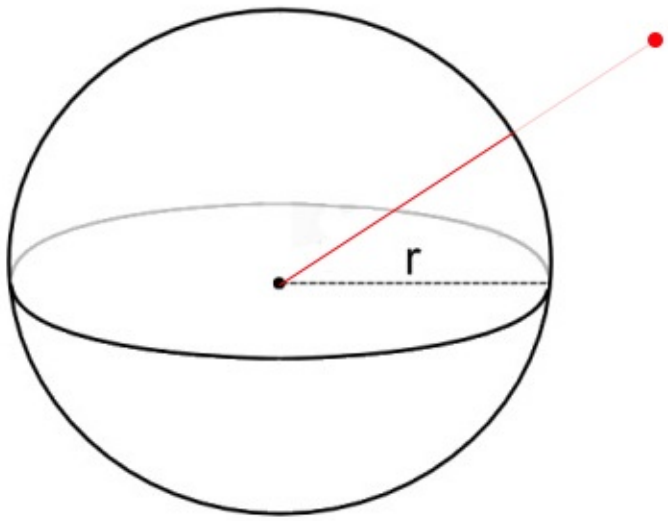
The sample / Unit test code for this section will be included with the next (Closest point on Sphere) section.

Closes Point On Sphere

Finding the closest point on a sphere can be just as useful as knowing if a point is inside a sphere or not. The closest point on sphere is often used for AI, Physics simuation and special effects.

The key to the closest point on sphere algorithm is to realize that no matter where that point may be, it will always be the same distance from the center of the sphere. That distance is equal to the radius.

Before reading the next paragraph (Because i will answer the question there), see if you can figure out given a point and a sphere how to get the closest point on the sphere to that point. I'll provide an image:



The algorithm

The algorithm is not as straight forward as Point in Sphere, but it's pretty simple. First, you need to subtract the point from the center of the sphere. Subtraction order matters on this one. You are left with a vector that points from the center of the sphere, to the test point.

Normalize this vector, then multiply it by the radius of the sphere. You now have a vector that points from the center of the sphere, to the edge of the sphere, towards the point in question.

Finally, add the position of the sphere (as a vector) to the new vector you have. This gives you a vector that points from the origin of the world to the edge of the sphere (A Point)

Prototype

The sample code below is probably not needed, it's a literal translation of the paragraph above.

```
// THIS BLOCK IS JUST SAMPLE CODE, DON'T COPY IT!
Point ClosestPoint(Vector3 spherePos, float sphereRad, Vector3 poi
nt) {
    // First, get a vetor from the sphere to the point
    Vector3 sphereToPoint = point - spherePos;
    // Normalize that vector
    sphereToPoint = Vector3.Normalize(sphereToPoint);
    // Adjust it's length to point to edge of sphere
    sphereToPoint *= sphereRad;
    // Translate into world space
    Vector3 worldPoint = spherePos + sphereToPoint;
    // Return new point
    return new Point(worldPoint.x, worldPoint.y, worldPoint.z);
}
```

On Your Own

Add the following function to the `Collisions` class:

```
public static Point ClosestPoint(Sphere sphere, Point point)
```

And provide an implementation for it!

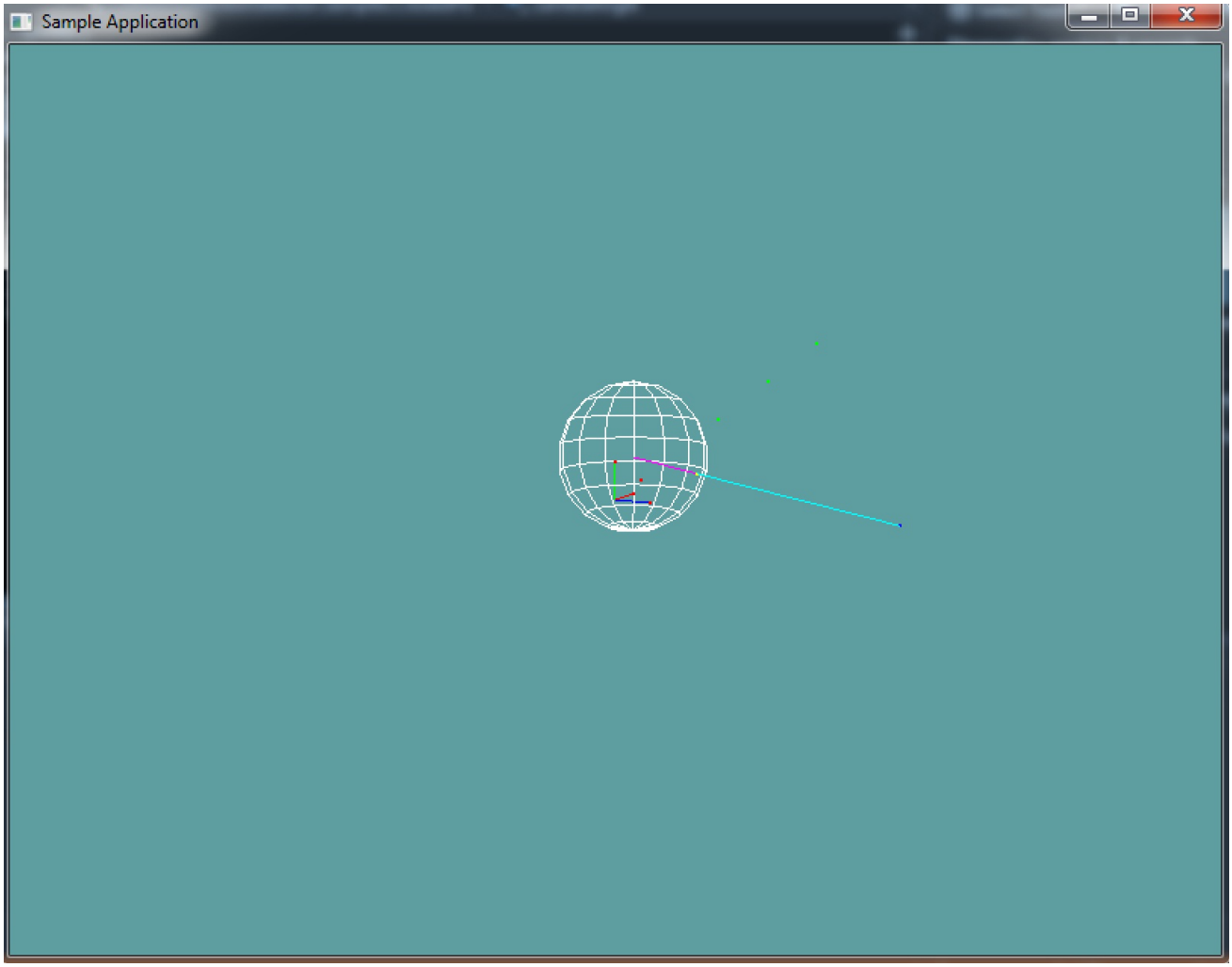
Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will test both the point in sphere, and closest point to sphere sections of code. If any kind of error occurs, red text should be printed to the console. The camera in this example orbits, but you can disable that by commenting out the update method.

If you don't see anything on screen, it's because the math library is probably wrong. This would happen if you forgot to copy the math implementation from OpenGL1X into Math-Implementation, and used the old Math-Implementation library.

Here is a screenshot of what the unit test will look like:




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
// The namespace where your primitives live!

namespace CollisionDetectionSelector.Samples {
    class ClosestPointSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Point[] testPoints = new Point[] {
            new Point(1, 0, 0),
            new Point(0, 1, 0),
            new Point(0, 0, 1),
            new Point(0.5f, 0.5f, 0.5f),
```

```
            new Point(2f, 2f, 2f),
            new Point(3f, 3f, 3f),
            new Point(4f, 4f, 4f),
        };

        Point farPoint = new Point(0f, 0f, 7f);
        Sphere testSphere = new Sphere(1f, 1f, 0f, 2f);

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.PointSize(2f);

            bool[] results = new bool[] { true, true, true, true,
false, false, false };
            for (int i = 0; i < testPoints.Length; ++i) {
                bool collision = Collisions.PointInSphere(testSphe
re, testPoints[i]);
                if (collision != results[i]) {
                    System.Console.ForegroundColor = System.Consol
eColor.Red;
                }
                System.Console.Write("Point: " + testPoints[i]);
                if (collision) {
                    System.Console.WriteLine(" is in sphere");
                }
                else {
                    System.Console.WriteLine(" is not in sphere");
                }
                System.Console.ResetColor();
            }

            Vector3 expected = new Vector3(0.719944f, 0.719944f, 1
.960392f);
            Point closest = Collisions.ClosestPoint(testSphere, fa
rPoint);
```

```
            if (expected != closest.ToVector()) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
            }
            System.Console.WriteLine("Closest point: " + closest);
            if (expected != closest.ToVector()) {
                System.Console.WriteLine("Expected: " + expected.X
 + ", " + expected.Y + ", " + expected.Z);
            }
            System.Console.ResetColor();
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawOrigin();

            GL.Color3(1f, 1f, 1f);
            testSphere.Render();

            foreach(Point point in testPoints) {
                if (Collisions.PointInSphere(testSphere, point)) {
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 1f, 0f);
                }
```

```
                point.Render();
            }

            GL.Color3(0f, 0f, 1f);
            farPoint.Render();

            Point closest = Collisions.ClosestPoint(testSphere, fa
rPoint);
            GL.Color3(1f, 0f, 1f);
            GL.Begin(PrimitiveType.Lines);
            GL.Vertex3(testSphere.Position.X, testSphere.Position.
Y, testSphere.Position.Z);
            GL.Vertex3(closest.X, closest.Y, closest.Z);
            GL.End();

            GL.Color3(1f, 1f, 0f);
            closest.Render();

            GL.Color3(0f, 1f, 1f);
            GL.Begin(PrimitiveType.Lines);
            GL.Vertex3(closest.X, closest.Y, closest.Z);
            GL.Vertex3(farPoint.X, farPoint.Y, farPoint.Z);
            GL.End();
        }

        public override void Update(float deltaTime) {
           cameraAngle.X += 45.0f * deltaTime;
        }

        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
```

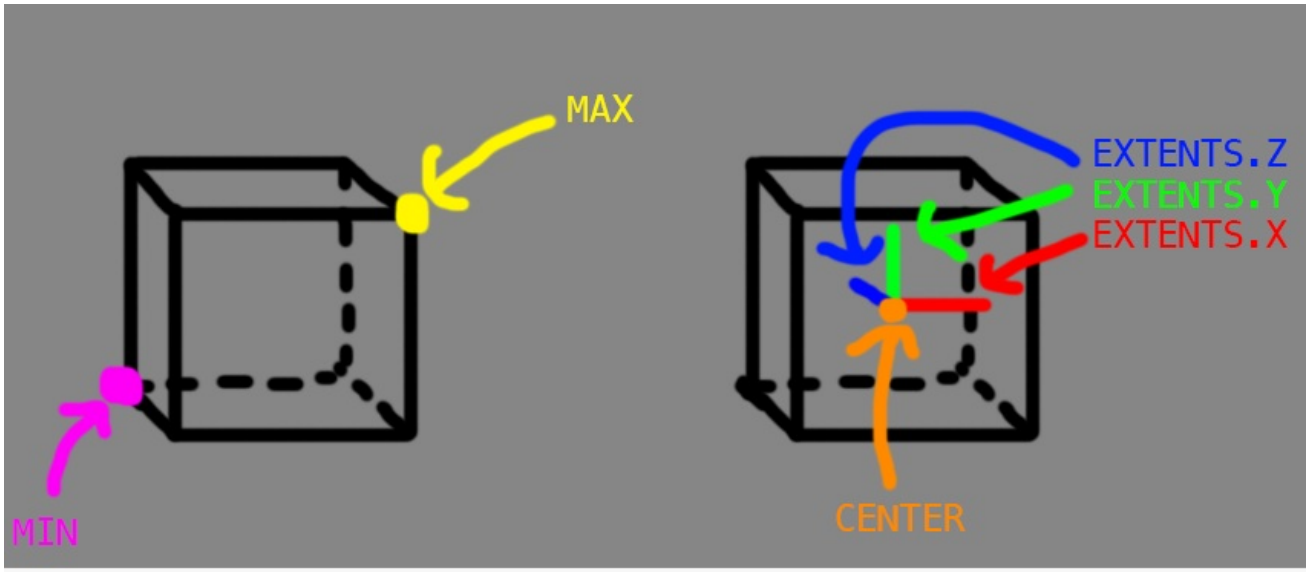
```
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

AABB - Axis Aligned Bounding Box

An AABB (Axis Aligned Bounding Box) is a 3D box. It's width / height / depth don't have to be equal, but the width is always aligned to the X axis, the height to the Y axis and depth to the Z axis. That is to say, this box can't be rotated.

AABB's are pivotal to spacial partitioning, they let us cut a section of 3D space up into smaller sections of 3D space. AABB's also serve as a great acceleration structure for hit detection, but more on that later.

The first question we have to ask is how to represent an AABB? There are two common ways, by storing the leftmost and rightmost corners, OR by storing the center point and a vector of how far the box extends on each side.




```
class AABB {
    Vector3 min;
    Vector3 max;
}
```

```
class AABB {
    Vector3 center;
    Vector3 extents;
}
```

Most mathematical texts describe an AABB as a min and max point. Because of this, i like to design my AABB classes as such. However, there is a bit of a problem, what if a user sets the min.X of the AABB to be greater than the max.X? For this reason i

usually include 2 bits of code:

```
// THIS BLOCK IS JUST SAMPLE CODE, DON'T COPY IT!
public bool IsValid {
    get {
        return Min.X < Max.X && Min.Y < Max.Y && Min.Z < Max.Z;
    }
}

public void Fix() {
    if (Min.X > Max.X) {
        // SWAP
    }
    if (Min.Y > Max.Y) {
        // Swap
    }
    if (Min.Z > Max.Z) {
        // Swap
    }
}
```

At the end of the day, your representation doesn't matter much. If you represent your AABB as a min and max internally, you can expose getter only `Center` and `Extents` properties.

Code Guide

This code guide is actually valid C# code. If you do all the TODO sections, it will compile. Like always, you don't have to use / follow this implementation.

One thing to keep in mind, the constructor that takes a min / max point as argument should call the `Fix` function because we can't trust user provided values.

That's the only constructor that needs to call Fix, the one that takes a point and extents builds min and max in-line, so you know it will be correct!

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

namespace CollisionDetectionSelector.Primitives {
    class AABB {
        public Point Min = new Point();
        public Point Max = new Point();

        public Point Center {
            get {
                // FIGURE OUT CENTER POINT
            }
            // NO SET!
        }

        public Vector3 Extents {
            get {
                // FIGURE OUT EXTENTS
            }
            // NO SET!
        }

        public bool IsValid {
            // TODO: Implement, getter only, no setter!
        }

        public void Fix() {
            // TODO: Implement
        }

        public AABB() {
            // TODO: Make a unit AABB
            // Min = -1, max = +1 on all axis
        }

        public AABB(Point min, Point max) {
```

```
            // TODO: Set Min and Max
            // Don't forget, could be invalid
        }

        public AABB(Point center, Vector3 extents) {
            // TODO: Set Min and Max
        }

        // No need to change below this, render and to-string
        // functions are provided
        public void Render() {
            GL.Begin(PrimitiveType.Quads);

            GL.Vertex3(Min.X, Min.Y, Max.Z);
            GL.Vertex3(Max.X, Min.Y, Max.Z);
            GL.Vertex3(Max.X, Max.Y, Max.Z);
            GL.Vertex3(Min.X, Max.Y, Max.Z);

            GL.Vertex3(Max.X, Min.Y, Max.Z);
            GL.Vertex3(Max.X, Min.Y, Min.Z);
            GL.Vertex3(Max.X, Max.Y, Min.Z);
            GL.Vertex3(Max.X, Max.Y, Max.Z);

            GL.Vertex3(Min.X, Max.Y, Max.Z);
            GL.Vertex3(Max.X, Max.Y, Max.Z);
            GL.Vertex3(Max.X, Max.Y, Min.Z);
            GL.Vertex3(Min.X, Max.Y, Min.Z);

            GL.Vertex3(Min.X, Min.Y, Min.Z);
            GL.Vertex3(Min.X, Max.Y, Min.Z);
            GL.Vertex3(Max.X, Max.Y, Min.Z);
            GL.Vertex3(Max.X, Min.Y, Min.Z);

            GL.Vertex3(Min.X, Min.Y, Min.Z);
            GL.Vertex3(Max.X, Min.Y, Min.Z);
            GL.Vertex3(Max.X, Min.Y, Max.Z);
            GL.Vertex3(Min.X, Min.Y, Max.Z);
```

```
            GL.Vertex3(Min.X, Min.Y, Min.Z);
            GL.Vertex3(Min.X, Min.Y, Max.Z);
            GL.Vertex3(Min.X, Max.Y, Max.Z);
            GL.Vertex3(Min.X, Max.Y, Min.Z);

            GL.End();
        }

        public override string ToString() {
            string result = "Min: (" + Min.X + ", " + Min.Y + ", "
 + Min.Z + "), ";
            result += "Max: ( " + Max.X + ", " + Max.Y + ", " + Ma
x.Z + ")";
            return result;
        }
    }
}
```

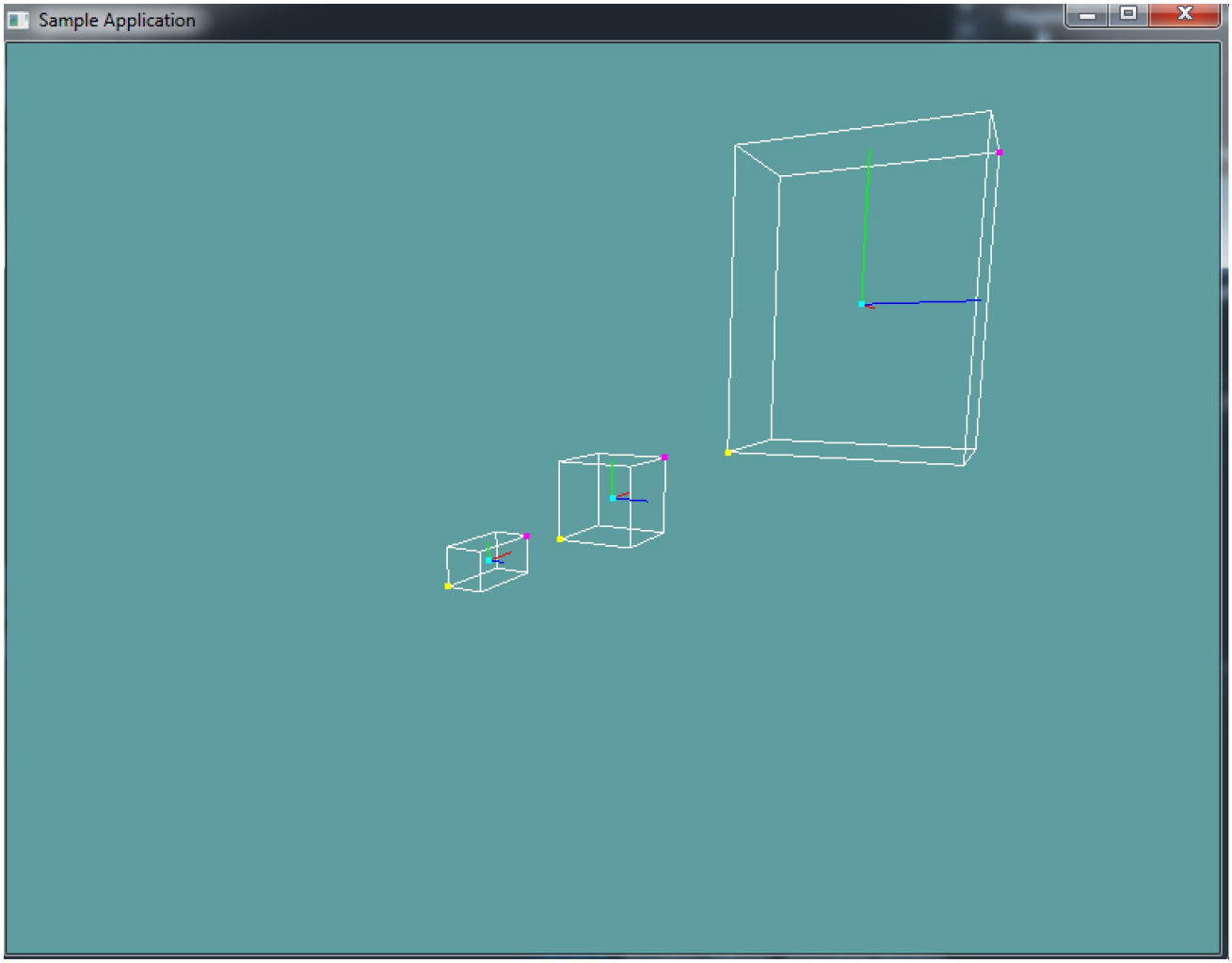
On Your Own

Implement the AABB class. You can use the code guide above, or use your own implementation

Sample / Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The sample will render 3 AABB's constructed using the 3 different constructors above. It will render each AABB, it's min and max points (in magenta and yellow), it's center point (In light blue) and it's extents (r, g, b for x, y, z). If there are any errors, the console should print red. This is what the sample will look like:




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class AABBSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        AABB aabb1 = new AABB();
        AABB aabb2 = new AABB(new Point(-3, -2, -3), new Point(-1,
-1, -2));
        AABB aabb3 = new AABB(new Point(5, 5, 5), new Vector3(2, 4
, 3));
```

```
        public override void Intialize(int width, int height) {
            // Seeing the back face of a square gives a better
            // overview of the actual geometry
            //GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.PointSize(4f);

            System.Console.ResetColor();
            CheckAABBMinMax(aabb1, 1, -1, -1, -1, 1, 1, 1);
            CheckAABBCenterExtents(aabb1, 1, 0, 0, 0, 1, 1, 1);
            System.Console.WriteLine("AABB1: " + aabb1);
            System.Console.WriteLine("\tCenter: " + aabb1.Center);
            System.Console.WriteLine("\tExtents: " + aabb1.Extents
.X + ", " + aabb1.Extents.Y + ", " + aabb1.Extents.Z);

            System.Console.ResetColor();
            CheckAABBMinMax(aabb2, 2, -3, -2, -3, -1, -1, -2);
            CheckAABBCenterExtents(aabb2, 2, -2, -1.5f, -2.5f, 1,
0.5f, 0.5f);
            System.Console.WriteLine("AABB2: " + aabb2);
            System.Console.WriteLine("\tCenter: " + aabb2.Center);
            System.Console.WriteLine("\tExtents: " + aabb2.Extents
.X + ", " + aabb2.Extents.Y + ", " + aabb2.Extents.Z);

            System.Console.ResetColor();
            CheckAABBMinMax(aabb3, 3, 3, 1, 2, 7, 9, 8);
            CheckAABBCenterExtents(aabb3, 3, 5, 5, 5, 2, 4, 3);
            System.Console.WriteLine("AABB3: " + aabb3);
            System.Console.WriteLine("\tCenter: " + aabb3.Center);
            System.Console.WriteLine("\tExtents: " + aabb3.Extents
.X + ", " + aabb3.Extents.Y + ", " + aabb3.Extents.Z);
        }

        protected void CheckAABBCenterExtents(AABB aabb, int num,
```

```
float cx, float cy, float cz, float ex, float ey, float ez) {
            if (aabb.Center.X != cx || aabb.Center.Y != cy || aabb
.Center.Z != cz) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
                System.Console.WriteLine("AABB" + num + ", center
is wrong");
                System.Console.WriteLine("\tGot: (" + aabb.Center.
X + ", " + aabb.Center.Y + ", " + aabb.Center.Z + ")");
                System.Console.WriteLine("\tExpected: (" + cx + ",
 " + cy + ", " + cz + ")");
            }

            if (aabb.Extents.X != ex || aabb.Extents.Y != ey || aa
bb.Extents.Z != ez) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
                System.Console.WriteLine("AABB" + num + ", extents
 is wrong");
                System.Console.WriteLine("\tGot: (" + aabb.Extents
.X + ", " + aabb.Extents.Y + ", " +aabb.Extents.Z + ")");
                System.Console.WriteLine("\tExpected: (" + ex + ",
 " + ey + ", " + ez + ")");
            }
        }

        protected void CheckAABBMinMax(AABB aabb, int num, float m
inX, float minY, float minz, float maxX, float maxY, float maxZ) {
            Point t = aabb.Min;
            if (t.X != minX || t.Y != minY || t.Z != minz) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
                System.Console.WriteLine("AABB" + num + ", min is
wrong");
                System.Console.WriteLine("\tGot: (" + t.X + ", " +
 t.Y + ", " + t.Z + ")");
                System.Console.WriteLine("\tExpected: (" + minX +
```

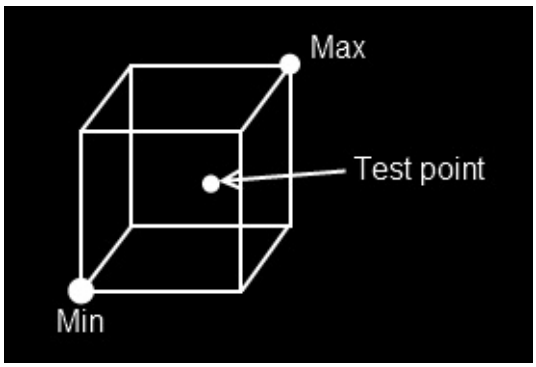
```
", " + minY + ", " + minz + ")");
            }
            t = aabb.Max;
            if (t.X != maxX || t.Y != maxY || t.Z != maxZ) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
                System.Console.WriteLine("AABB" + num + ", max is
wrong");
                System.Console.WriteLine("\tGot: (" + t.X + ", " +
 t.Y + ", " + t.Z + ")");
                System.Console.WriteLine("\tExpected: (" + maxX +
", " + maxY + ", " + maxZ + ")");
            }
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawAABBDebug(aabb1);
            DrawAABBDebug(aabb2);
            DrawAABBDebug(aabb3);
        }

        protected void DrawAABBDebug(AABB aabb) {
            GL.Color3(1f, 1f, 1f);
            aabb.Render();
            GL.Color3(1f, 1f, 0f);
```

```
            aabb.Min.Render();
            GL.Color3(1f, 0f, 1f);
            aabb.Max.Render();
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(aabb.Center.X, aabb.Center.Y, aabb.Center.Z
);
            GL.Vertex3(aabb.Center.X + aabb.Extents.X, aabb.Center
.Y, aabb.Center.Z);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(aabb.Center.X, aabb.Center.Y, aabb.Center.Z
);
            GL.Vertex3(aabb.Center.X, aabb.Center.Y + aabb.Extents
.Y, aabb.Center.Z);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(aabb.Center.X, aabb.Center.Y, aabb.Center.Z
);
            GL.Vertex3(aabb.Center.X, aabb.Center.Y, aabb.Center.Z
 + aabb.Extents.Z);
            GL.End();
            GL.Color3(0f, 1f, 1f);
            aabb.Center.Render();
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
```

```
            }
        }
    }
```

Point in AABB

Figuring out if a point is inside of an AABB is pretty simple, especially if you think of an AABB as being represented by a min and a max point. Take a look at the image below and try to figure out the algorithm before i describe it in the next paragraph



The algorithm

Looking at the above image, we can figure out that if a point is inside of a AABB:

- The relationship of point to min is such:
  - `Point.X` , `Point.Y` and `Point.Z` are all **greater** than `Min.X` , `Min.Y` and `Min.Z` .
- The relationship of point to max is such:
  - `Point.X` , `Point.Y` and `Point.Z` are all **less** than `Min.X` , `Min.Y` and `Min.Z` .

That is to say, if `Point > min && Point < max` then the point is inside the AABB.

On Your Own

Add the following function to the `Collisions` class:

```
public static bool PointInAABB(AABB aabb, Point point)
```

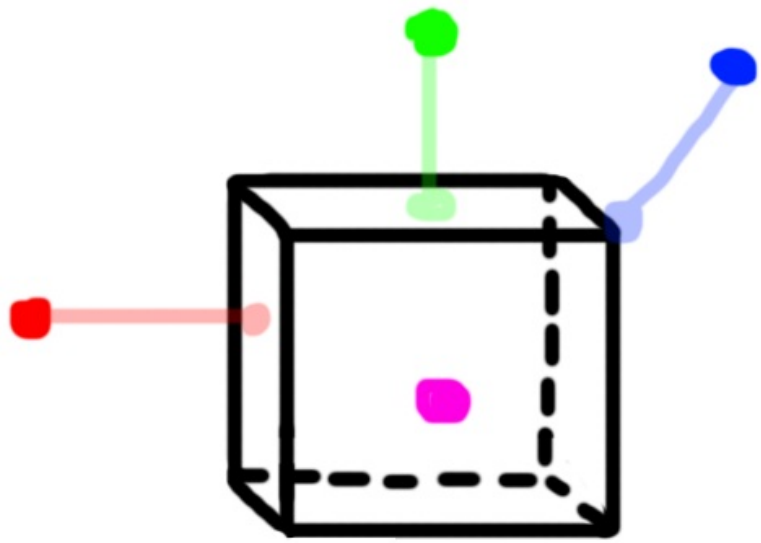
And provide an implementation for it!

Unit Test

The sample / Unit test code for this section will be included with the next (Closest point on AABB) section.

Closest point on AABB

Finding the closest point on a plane is not too difficult. Consider the following image:



The red, green and blue dots are outside the AABB, the closest point to them on the AABB is shown in a transparent projection. The purple dot is INSIDE the aabb. The closest point to the purple one is its-self.

Given that image, can you figure out the rules of the closest point?

The algorithm

To find the closest point on an AABB, we just clamp the X, Y and Z positions of the point to the AABB! Consider the following if sequence for the X axis test:

- If point.X > aabb.max.X
  - Set result.X = aabb.max.X
- Else if point.X < aabb.min.X
  - Set result.X = aabb.min.X
- Else

- Set result.X = point.X

You might have noticed something odd, if the point is outside the min or max x it is clamped to either the min or max. This puts the point on the edge of the AABB. But if the test point is inside of the aabb, it is not clamped to an edge, rather it just returns the test point's position (on the axis being tested).

This might sound like a bug at first, but it's not. In order to use the closest-point algorithm, we need the closest point to be inside the volume sometimes.

On Your Own

Add the following function to the `Collisions` class:

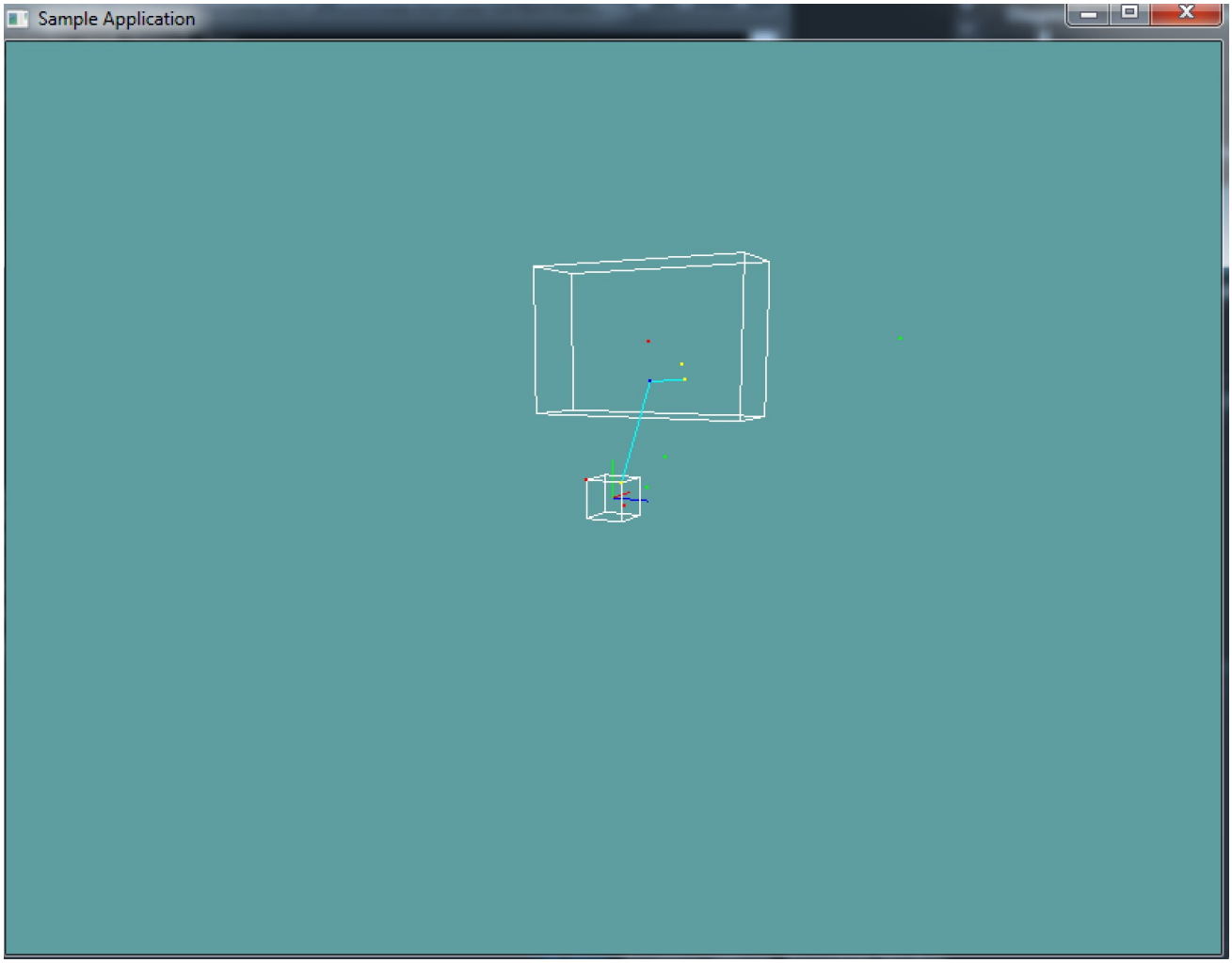
```
public static Point ClosestPoint(AABB aabb, Point point)
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

If you comment out the update function, your image should look like the one attached. The constructor only checks for one




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class AABBPointSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        AABB[] testAABBs = new AABB[] {
            new AABB(new Point(2f, 4f, 0f), new Vector3(1f, 2f, 3f
)),
             new AABB(new Point(-0.5f, -0.5f, -0.5f), new Point(0.5
f, 0.5f, 0.5f))
```

```
        };

        Point[] testPoints = new Point[] {
            new Point(0.2f, -0.2f, 0.2f),
            new Point(1f, 1f, 1f),
            new Point(-0.5f, 0.5f, -0.5f),
            new Point(2f, 4f, 0f),
            new Point(2f, 0f, 0f),
            new Point(2f, 4f, 6.5f),
        };

        Point farPoint = new Point(-1f, 3f, 1.5f);
        Point secondFarPoint = new Point(2f, 3.4f, 1f);

        public override void Intialize(int width, int height) {
            // Seeing the back face of a square gives a better
            // overview of the actual geometry
            //GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.PointSize(2f);

            Point closest = Collisions.ClosestPoint(testAABBs[0],
secondFarPoint);
            if (closest.ToVector() != new Vector3(2f, 3.4f, 1f)) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
            }
            System.Console.WriteLine("closest: " + closest + " / 2
, 3.4, 1");
            System.Console.ResetColor();
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
```

```
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawOrigin();

            GL.Color3(1f, 1f, 1f);
            foreach(AABB aabb in testAABBs) {
                aabb.Render();
            }

            foreach (Point point in testPoints) {
                bool collides = false;
                foreach (AABB aabb in testAABBs) {
                    if (Collisions.PointInAABB(aabb, point)) {
                        collides = true;
                        break;
                    }
                }

                if (collides) {
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 1f, 0f);
                }
                point.Render();
            }

            Point closest;
            foreach (AABB aabb in testAABBs) {
                closest = Collisions.ClosestPoint(aabb, farPoint);
```

```
                GL.Color3(0f, 1f, 1f);
                GL.Begin(PrimitiveType.Lines);
                GL.Vertex3(closest.X, closest.Y, closest.Z);
                GL.Vertex3(farPoint.X, farPoint.Y, farPoint.Z);
                GL.End();

                GL.Color3(1f, 1f, 0f);
                closest.Render();
            }

            GL.Color3(0f, 0f, 1f);
            farPoint.Render();
            secondFarPoint.Render();

            closest = Collisions.ClosestPoint(testAABBs[0], second
FarPoint);
            GL.Color3(0f, 1f, 1f);
            GL.Begin(PrimitiveType.Lines);
            GL.Vertex3(closest.X, closest.Y, closest.Z);
            GL.Vertex3(secondFarPoint.X, secondFarPoint.Y, secondF
arPoint.Z);
            GL.End();

            GL.Color3(1f, 1f, 0f);
            closest.Render();
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
```

```
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Plane

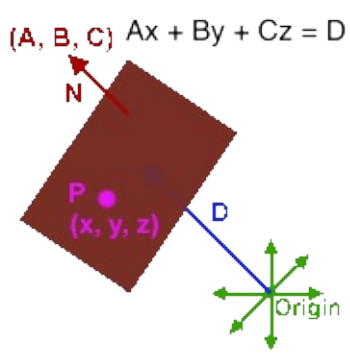
A plane is a flat surface that extends infinateley in all directions. There are three common ways to represent a plane:

- Three points (not on a straight line)
- A normal and a point on the plane
- **A normal and the distance from origin**

Your native plane implementation should represent the plane as the third, a normal and a distance from origin. This is the plane equation. Given the normal and distance, any point `X` that satisfies the equation

```
Dot(X, Normal) = Distance; // Plane equasion
```

Is on the plane. The above dot product example can also be written as the formal plane equasion: `NormalX * PointX + NormalY * PointY + NormalZ * PointZ = Distance` or `A * x + B * y + C * z = Distance`



When researching this topic, you will often see the equation represented as:

```
Ax + By + Cz + D = 0
```

instead of

```
Ax + By + Cz = d
```

Why is that? They re-dedine D!

```
Ax + By + Cz = d
Ax + By + Cz - d = 0
--------------------
D = -d
Ax + By + Cz + D = 0
```

That gets super confusing. But for some reason developers seem to like adding D instead of subtracting it, even tough it adds un-necesary complication to the equation.

Because of this, moving forward i'm going to write the equation as:

```
Ax + By + Cz + (-D) = 0
```

We can represent a plane as a normal and a distance from origin, but we can still construct it from 3 points using the following formula:

```
// THIS BLOCK IS JUST SAMPLE CODE, DON'T COPY IT!
Plane ComputePlane(Point a, Point b, Point c) {
    Plane result = new Plane();
    result.Normal = Normalize(Cross(b - a, c - a));
    result.Distance = Dot(result.Normal, a);
    return result;
}
```

Let's break this apart. We create two vectors (ba and ca) by subtracting point a from b and c from c. Both of these vectors lie on the plane. We can get the normal of the plane (A vector perpendicular to ba and ca) by taking the cross product of ba and ca.

Becuase we don't know that ba and ca are of unit length or not, we need to assume they are not. In order to get a normal, we have to normalize the result of the cross product.

Now we have the normal (ABC), all we need to fill out the plane equasion is D. To get the distance of the plane from origin, take the dot product of any point on the plane (In this case i chose a because it is already a known) and the normal we just found.

Code Guide

Because the plane's normal is finicky, i suggest not exposing it directly. Instead, have a private `_normal` variable, and a `Normal` getter / setter. Whenever `Normal` is set, it just sets `_normal` and calls `Normalize` on it.

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

namespace CollisionDetectionSelector.Primitives {
    class Plane {
        protected Vector3 _normal = new Vector3();

        public Vector3 Normal {
            // TODO: Getter and setter
            // remember, this just accesses and sets _normal
            // Which must be normalized when set!
        }
        public float Distance = 0f;

        public Plane() {
            // TODO: Make a default plane
            // by default normal is (0, 0, 1) and distance is 0
        }

        public Plane(Vector3 norm, float dist) {
            // TODO
```

```
        }

        public Plane(Point a, Point b, Point c) {
            // TODO: Construct this plane from a point
        }

        // No need to edit anything below this
        public void Render(float scale = 1f) {
            //Debug Normal
            /* // GL.Color3(1f, 1f, 0f);
            GL.Begin(PrimitiveType.Lines);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(Normal.X * 500, Normal.Y * 500, Normal.Z *
500);
            GL.End(); */

            // Construct plane orientation
            Vector3 forward = new Vector3(Normal.X, Normal.Y, Norm
al.Z);
            Vector3 up = new Vector3(0f, 1f, 0f);
            Vector3 right = Vector3.Cross(forward, up);
            up = Vector3.Cross(right, forward);

            // Because this is going to be a matrix, it needs to b
e normalized
            forward.Normalize();
            right.Normalize();
            up.Normalize();

            // Create plane model matrix
            Matrix4 rot = new Matrix4(right.X, up.X, -forward.X, 0
.0f,
                right.Y, up.Y, -forward.Y, 0.0f,
                right.Z, up.Z, -forward.Z, 0.0f,
                0.0f, 0.0f, 0.0f, 1.0f);
            Matrix4 trans = Matrix4.Translate(Normal * Distance);
            Matrix4 model = trans * rot;
```

```
            // Load matrix and render plane
            GL.PushMatrix();
            GL.MultMatrix(model.OpenGL);
            GL.Scale(scale, scale, scale);
            //GL.Color3(1f, 1f, 1f);
            GL.Begin(PrimitiveType.Quads);
            GL.Vertex3(-1, -1, 0);
            GL.Vertex3(-1, 1, 0);
            GL.Vertex3(1, 1, 0);
            GL.Vertex3(1, -1, 0);
            GL.End();
            GL.PopMatrix();
         }

        public override string ToString() {
            return "N: (" + Normal.X + ", " + Normal.Y + ", " + No
rmal.Z + "), D: " + Distance;
        }
    }
}
```

Unlike the previous primitives the `Render` function for the plane takes a scale. By default, this function renders a plane that ranges from -1 to 1. But because a plane is infinate, you may need it to be larger. If that's the case, provide a large number as the scale argument when calling `Render`

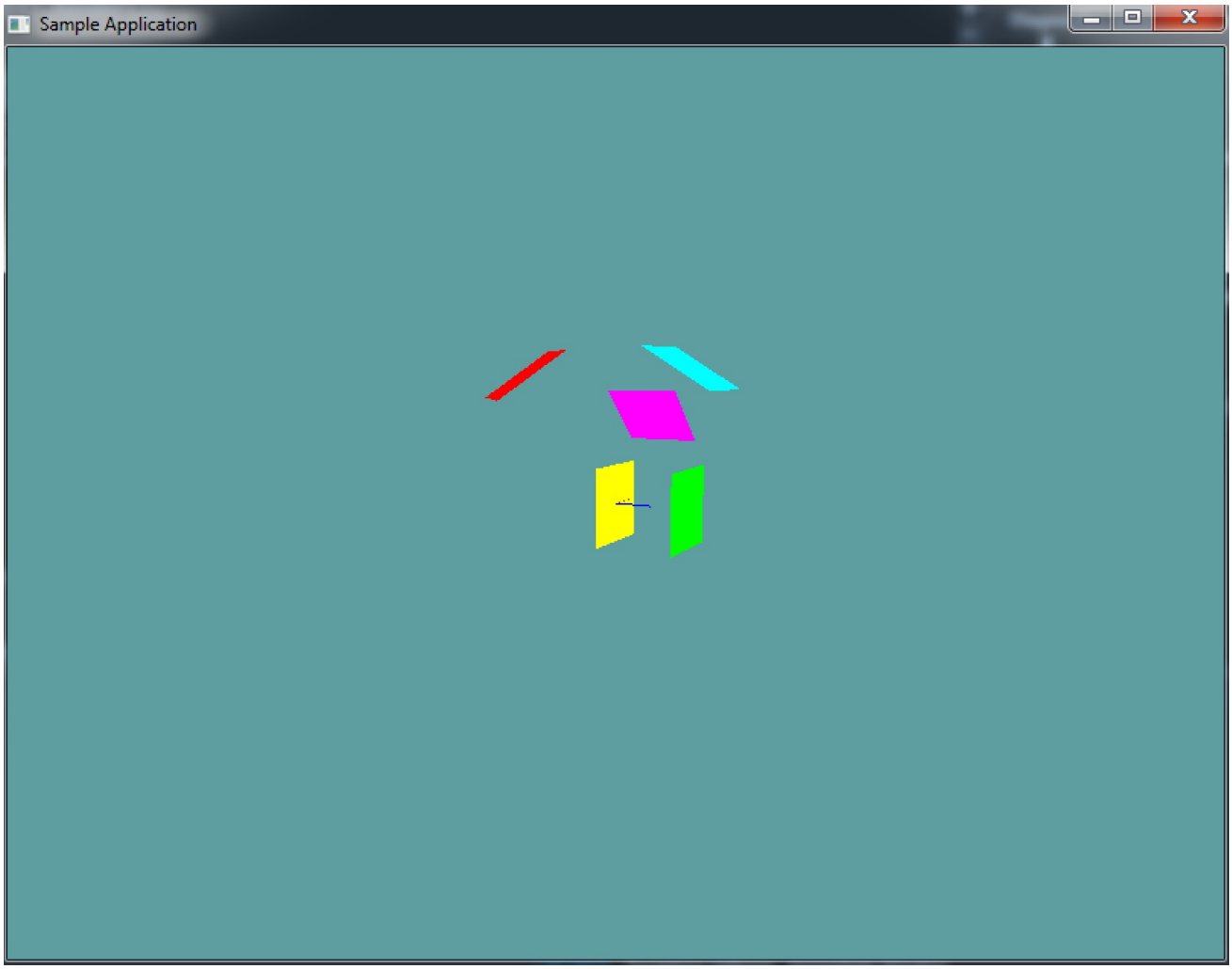
On Your Own

Implement the Plane class. You can use the code guide above, or use your own implementation

Sample / Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This example is visual only, no errors will be printed to the console if the code is bad. This is what the final render should look like:




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class PlaneSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);
```

```
        Plane[] plane = new Plane[] {
            new Plane(),
            new Plane(),
            new Plane(new Vector3(0f, 5f, 3f), 4f),
            new Plane(new Point(5, 6, 7), new Point(6, 5, 4), new
Point(1, 2, 3)),
            new Plane(new Point(0, 0, 2), new Point(1, 1, 2), new
Point(2, 0, 2))
        };

        public override void Intialize(int width, int height) {
            //GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.DepthTest);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Fill);
            GL.PointSize(2f);

            plane[1].Normal = new Vector3(1f, 1f, 0f);
            plane[1].Distance = 3.0f;
            plane[3].Distance = 4f;

        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);
```

```
            DrawOrigin();

            float[][] renderColors = new float[][] {
                new float[] { 1f, 1f, 0f },
                new float[] { 1f, 0f, 1f },
                new float[] { 0f, 1f, 1f },
                new float[] { 1f, 0f, 0f },
                new float[] { 0f, 1f, 0f }
            };

            for (int i = 0; i < plane.Length; ++i) {
                GL.Color3(renderColors[i][0], renderColors[i][1],
renderColors[i][2]);
                plane[i].Render();
            }
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Point on plane

Testing if a point is on a plane or not is fairly straight forward. In fact, we've done it multiple time. The key to this is the plane equasion!

```
// Plane representation
class Plane {
  float A, B, C; // Normal
  float D; // Dostance
}

// Plane equasion (XYZ is a point on the plane)
A*X + B*Y + C*Z + = Distance

// We can represent the above as:
Dot(Normal, UnknownPoint) = Distance
```

Given this information we can figure out if a point is on a plane or not. Let's rearrange the distance equasion, so the distance is on the left side and the right side is 0:

```
Dot(UnknownPoint, Normal) - Distance == 0
```

If a point satisfies the above equasion it is on the plane!

```
A*X + B*Y + C*Z - D = Result
```

The resulting number is what we care about.

- If it is 0, the point is on the plane
- If it is positive, the point is in front of the plane
- If it is negative, the point is behind the plane

On Your Own

Add the following functions to the `Collisions` class:

```
public static bool PointOnPlane(Point point, Plane plane);
// Should return a positive number, a negative number or 0
public static float DistanceFromPlane(Point point, Plane plane);
```

And provide an implementation for them!

HINT `PointOnPlane` not behaving as expected? Remember, floating point numbers are finicky. You WILL need to do an epsilon comparison!

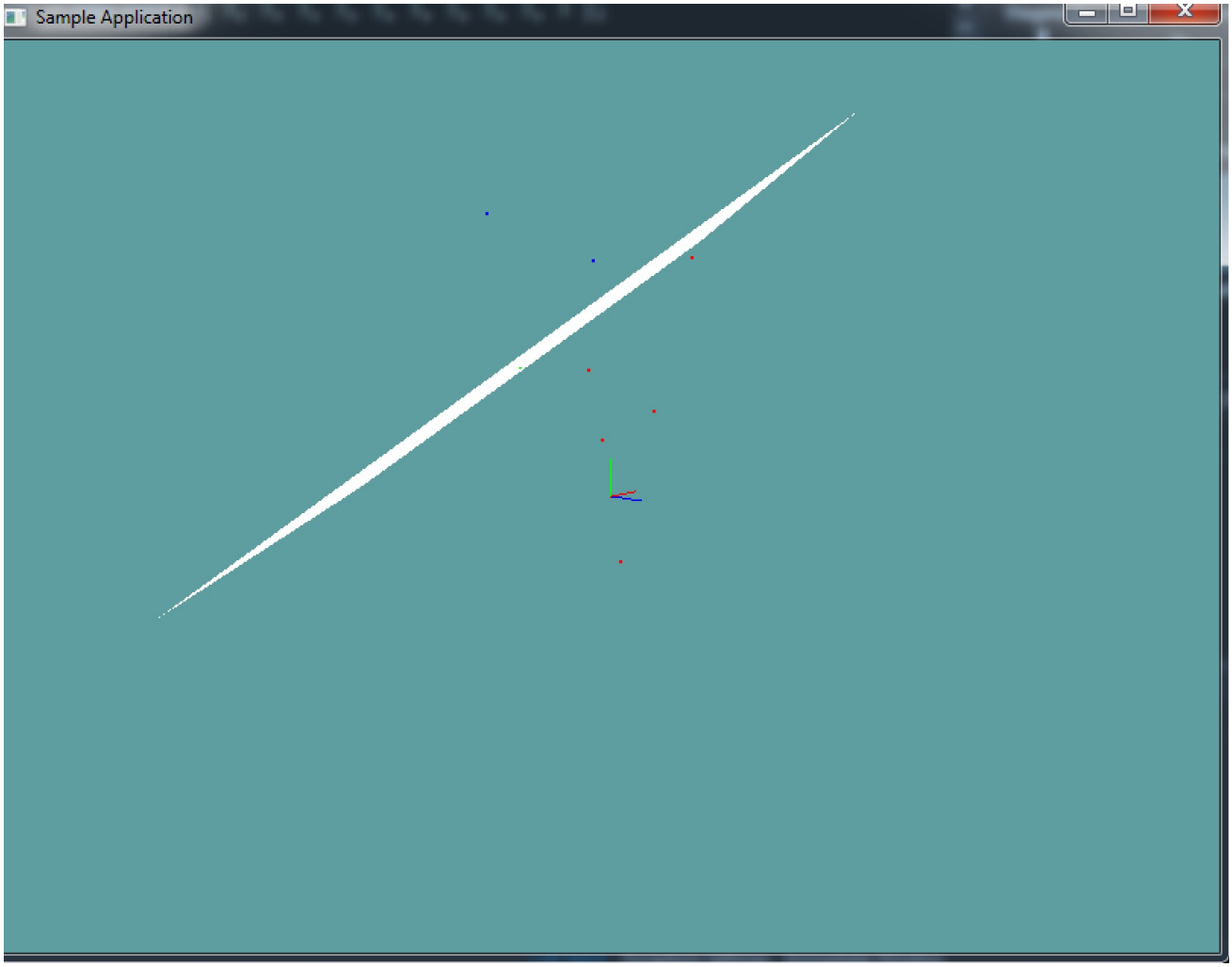
`DistanceFromPlane` does not need an epsilon, as it returns the result of an equation!

Try to implement the epsilon comparison on your own, if you can't figure it out, here is how to do it.

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The unit test renders a large plane and colors the dots around it. Any point on the plane is green, points behind the plane are red, points in front of it are blue. The constructor will spit out errors if either of the above equasions is bad.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class PointOnPlaneSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Plane plane = new Plane(new Point(5, 6, 7), new Point(6, 5
, 4), new Point(1, 2, 3));
        Point[] points = new Point[] {
            new Point(0f, 0f, 0f),
            new Point(2f, 6f, 1f),
```

```
            new Point(3f, 1f, -3f),
            new Point(-2f, -1f, 2f),
            new Point(7f, 7f, -7f),
            new Point(3f, 2f, -1f),
            new Point(10f, 3f, -10f),
            new Point(2f, 8f, -7f),
            new Point(-1.632993f, 3.265986f, -1.632993f),
            new Point(-1.020621f, 3.265986f, -2.245366f)
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(2f);
            plane.Distance = 4f;

            int[] expected = new int[] { -1, -1, -1, -1, 1, -1, -1
, 1, 0, 0 };
            int[] result = new int[] { 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
 };

            if (expected.Length != result.Length) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
                System.Console.WriteLine("Gabor messed up!");
            }

            if (result.Length != points.Length) {
                System.Console.ForegroundColor = System.ConsoleCol
or.Red;
                System.Console.WriteLine("Gabor messed up!");
            }

            for (int i = 0; i < points.Length; ++i) {
                Point point = points[i];
                if (Collisions.PointOnPlane(point, plane)) {
                    result[i] = 0;
                }
```

```
                else {
                    if (Collisions.DistanceFromPlane(point, plane)
 < 0f) {
                        result[i] = -1;
                    }
                    else {
                        result[i] = 1;
                    }
                }
            }

            for (int i = 0; i < points.Length; ++i) {
                if (result[i] == 3) {
                    System.Console.ForegroundColor = System.Consol
eColor.Red;
                    System.Console.WriteLine("Gabor messed up!");
                }

                if (expected[i] != result[i]) {
                    System.Console.ForegroundColor = System.Consol
eColor.Red;
                    System.Console.Write("Point " + i + " was expe
cted ");
                    if (expected[i] == 0) {
                        System.Console.Write("on ");
                    }
                    else if (expected[i] == 1) {
                        System.Console.Write("in front of ");
                    }
                    else if (expected[i] == -1) {
                        System.Console.Write("behind the ");
                    }
                    System.Console.Write("plane, but was found ");
                    if (result[i] == 0) {
                        System.Console.Write("on");
                    }
                    else if (result[i] == 1) {
```

```
                        System.Console.Write("in front");
                    }
                    else if (result[i] == -1) {
                        System.Console.Write("behind");
                    }
                    System.Console.WriteLine("\n\tPlane: " + plane
.ToString());
                    System.Console.WriteLine("\tPoint: " + points[
i].ToString() + "\n");
                }
            }
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawOrigin();

            GL.Color3(1f, 1f, 1f);
            plane.Render(7f);

            foreach(Point point in points) {
                if (Collisions.PointOnPlane(point, plane)) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    if (Collisions.DistanceFromPlane(point, plane)
```

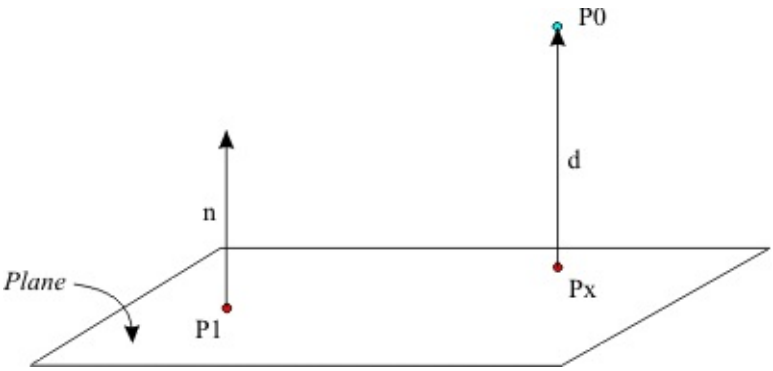
```
< 0f) {
                        GL.Color3(1f, 0f, 0f);
                    }
                    else {
                        GL.Color3(0f, 0f, 1f);
                    }
                }
                point.Render();
            }
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
```

```
                GL.LoadIdentity();
            }
        }
    }
```

Closest Point on Plane

Sometimes knowing if a point is on a plane or not isn't enough. Imagine your character walking in a room. You want the feet of your character to be on the floor. This is where knowing the closest point on plane helps. You can get your characters position, and the closest point on the ground plane is where you need to place him.

Technically, the we find the closest point to a plane using an orthographic projection, but there is an easier way to understand this.



Here we have a plane with a normal N. We want to find the closest point on the plane to P0, on the image that would be Px. Take note, the point Px is some distance away from P0, but that distance is in the direciton of the normal!

The closest point on a plane will always be in the direction of the plane normal from the test point. All we have to figure out is **how far is the test point from the plane**? In the above picture, that distance is **d**.

The algorithm

Thanks to the above image, we can deduct that given a plane, and a point, the closest point on the plane from that point will be the point minus the distance of the point from the plane in the direction of the planes normal.

The question now becomes, how do we find the distance between a point and a plane? The last section *Point on Plane* mentioned this as the formula:

```
Dot(SomePoint, Normal) == Distance
```

This becomes pretty easy to implement in a function like so:

```
// THIS BLOCK IS JUST SAMPLE CODE, DON'T COPY IT!
Point ClosestPointOnPlane(Plane plane, Point point) {
    // This works assuming plane.Normal is normalized, which it sh
ould be
    float distance = DOT(plane.Normal, point) - plane.Distance;
    // If the plane normal wasn't normalized, we'd need this:
    // distance = distance / DOT(plane.Normal, plane.Normal);

    return point - distance * plane.Normal;
}
```

On Your Own

Add the following function to the `Collisions` class:

```
public static Point ClosestPoint(Plane plane, Point point);
```

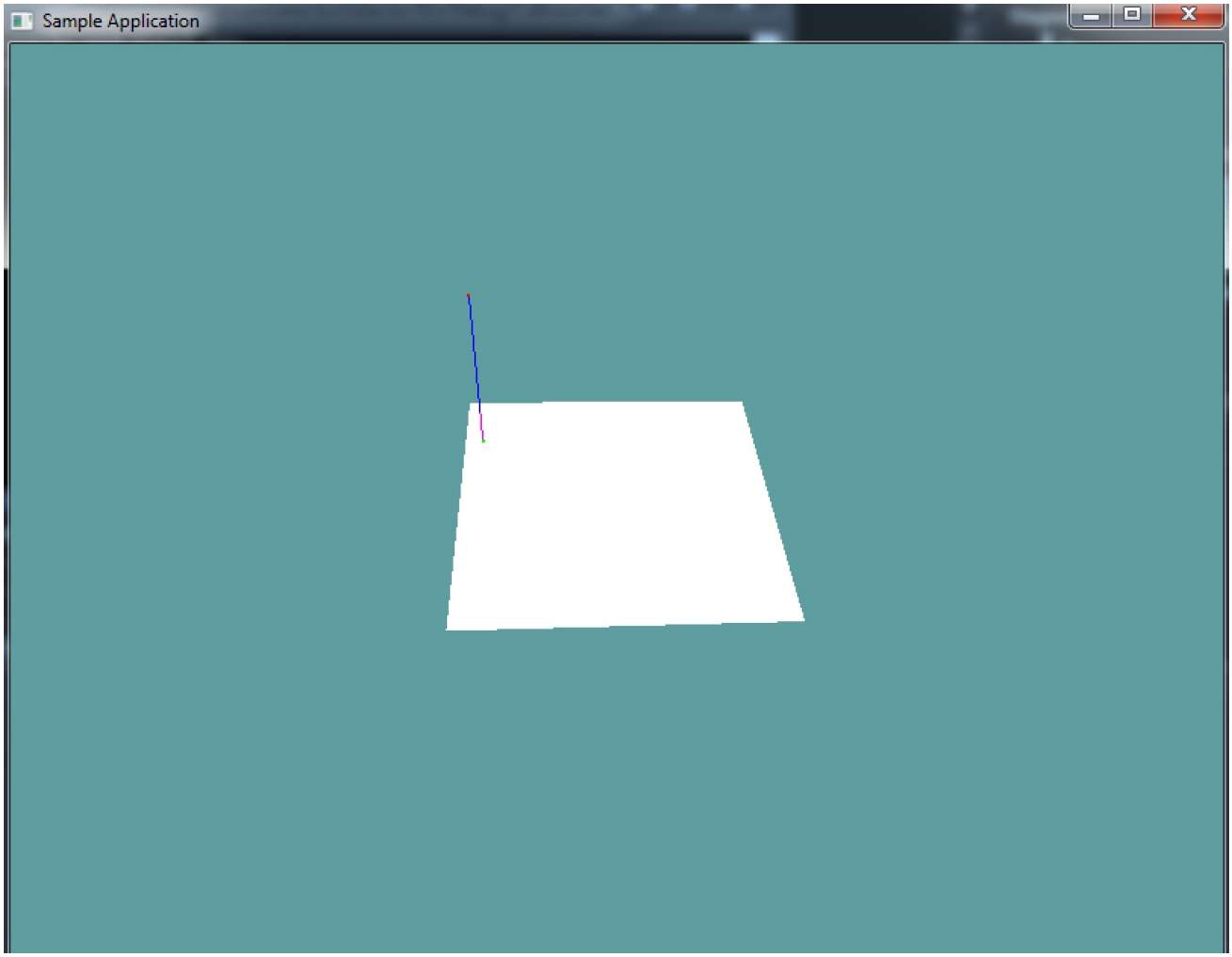
And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The following code is visual only, if you make any mistakes no error is printed!

The image is straight forward, there is a plane, the test point is red, the closest point is green. There is a blue line going from the test point to the closest point. The magenta line is the normal of the plane (rendered on top of the blue line)




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class ClosestPointPlaneSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Plane plane = new Plane(new Point(5, 6, 7), new Point(6, 5
, 4), new Point(1, 2, 3));

        Point point = new Point(2f, 5f, -3f);
```

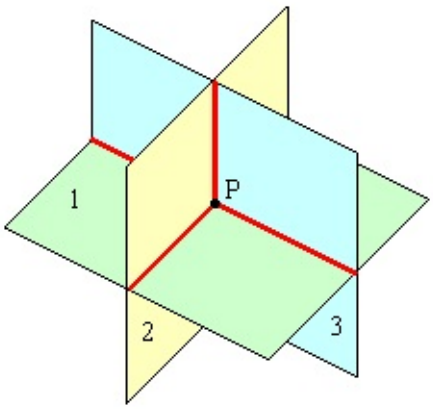
```
        public override void Intialize(int width, int height) {
            GL.PointSize(2f);
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawOrigin();

            GL.Color3(1f, 1f, 1f);
            plane.Render(4f);

            Point closest = Collisions.ClosestPoint(plane, point);
            float distance = Collisions.DistanceFromPlane(point, p
lane);
            Vector3 vec = point.ToVector() - plane.Normal * distan
ce;

            GL.Color3(0f, 0f, 1f);
            GL.Begin(PrimitiveType.Lines);
            GL.Vertex3(point.X, point.Y, point.Z);
            GL.Vertex3(vec.X, vec.Y, vec.Z);
            GL.End();

            GL.Color3(1f, 0f, 1f);
            GL.Begin(PrimitiveType.Lines);
            GL.Vertex3(closest.X, closest.Y, closest.Z);
```

```
            GL.Vertex3(closest.X + plane.Normal.Z, closest.Y + pla
ne.Normal.Y, closest.Z + plane.Normal.Z);
            GL.End();

            GL.Color3(1f, 0f, 0f);
            point.Render();

            GL.Color3(0, 1f, 0f);
            closest.Render();
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
```

```
                GL.LoadIdentity();
            }
        }
    }
```

3 Plane Intersection

Planes have a pretty special property. Given 3 unique planes, they intersect at exactly one point!



Any 3 dimensional cordinate system has 3 axis (x, y, z) which can be represented by 3 planes. Where those axis meet is considered (0, 0, 0) or the origin of the coordinate space.

Other than debug visualization, i've never used this intersection test in production. But it's useful to know in order to visualize some debug properties.

The algorithm

I don't fully understand this algorithm, usually i copy it out of a book. In that spirit, i'm just going to go ahead and provide it:

```
Point IntersectPlanes(Plane p1, Plane p2, Plane p3) {
    Vector m1 = new Vector(p1.Normal.X, p2.Normal.X, p3.Normal.X);
    Vector m2 = new Vector(p1.Normal.Y, p2.Normal.Y, p3.Normal.Y);
    Vector m3 = new Vector(p1.Normal.Z, p2.Normal.Z, p3.Normal.Z);
    Vector d = new Vector(p1.Distance, p2.Distance, p3.Distance);

    Vector u = Cross(m2, m3);
    Vector v =Cross (m1, d);

    float denom = Dot(m1, u);

    if (Abs(denom) < EPSILON) {
        // Planes don't actually intersect in a point
        // Throw exception maybe?
        return new Point(0, 0, 0);
    }

    return new Point(
        Dot(d, u) / denom,
        Dot(m3, v) / denom,
        -Dot(m2, v) / denom
    );
}
```

Of course, there are ways for 3 planes to intersect that do not end in a point:

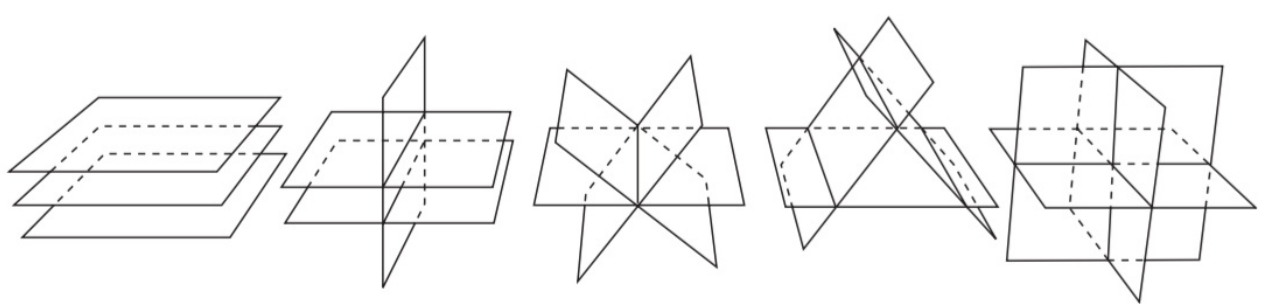


Figure 5.31 The five essentially different intersection configurations of three planes.

In that case we just return 0, but it might be appropriate to throw an exception. Or it might not, depends on what you use this function for!

On Your Own

Add the following function to the `Collisions` class:

```
public static Point Intersection(Plane p1, Plane p2, Plane p3);
```

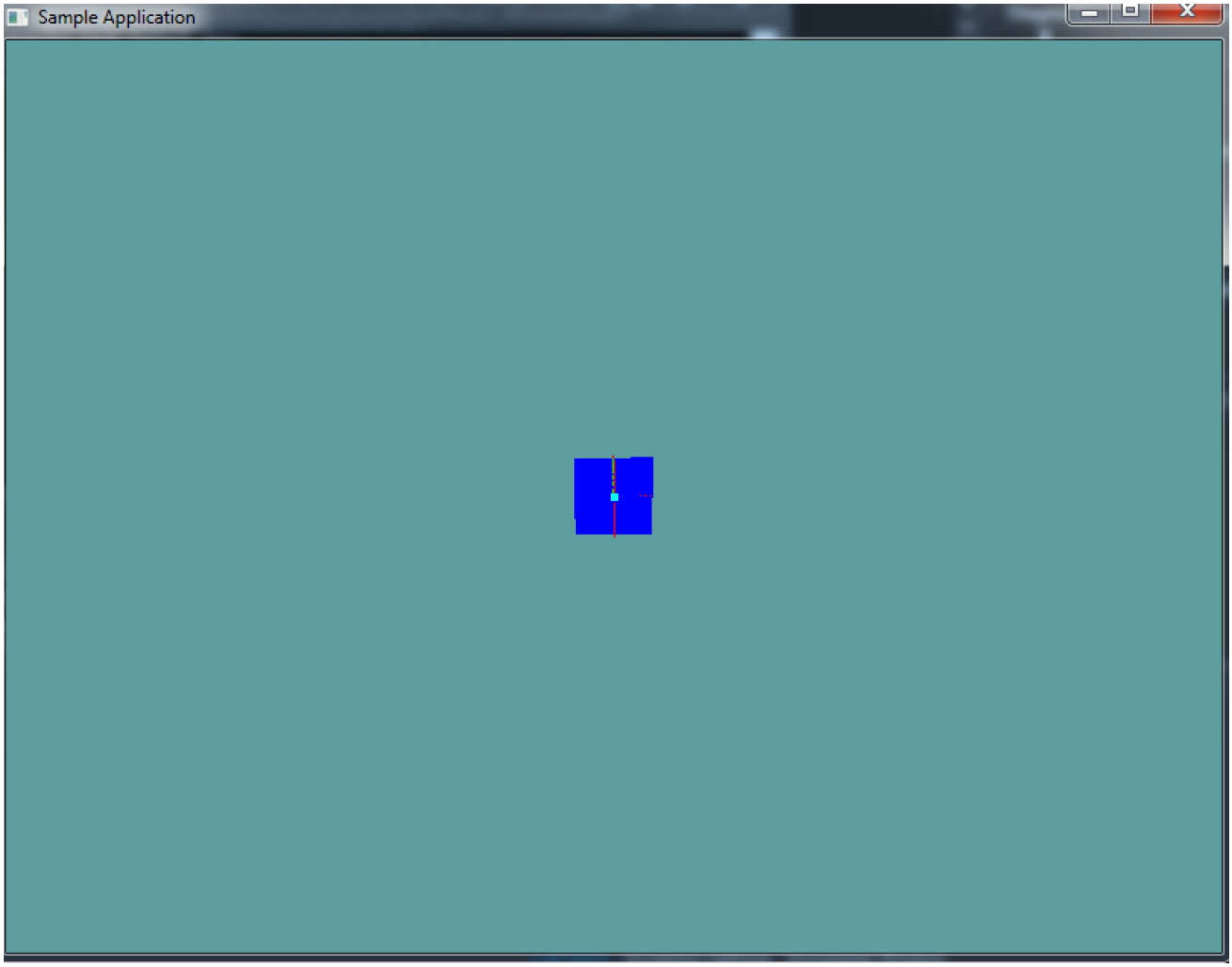
And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The following code is visual only, if you make any mistakes no error is printed!

The red and blue planes render just fine, the green plane will not. This is because of a minor flaw in the render code of the plane class. The intersection of the 3 planes renders in a light blue color.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class ThreePlaneIntersectionSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Plane plane1 = new Plane(new Vector3(1f, 0f, 0f), 0f);
        Plane plane2 = new Plane(new Vector3(0f, 1f, 0f), 0f);
        Plane plane3 = new Plane(new Vector3(0f, 0f, 1f), 0f);

        public override void Intialize(int width, int height) {
```

```
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawOrigin();

            GL.Color3(1f, 0f, 0f);
            plane1.Render();
            GL.Color3(0f, 1f, 0f);
            plane2.Render();
            GL.Color3(0f, 0f, 1f);
            plane3.Render();

            GL.Disable(EnableCap.DepthTest);
            GL.Color3(0f, 1f, 1f);
            Point midPoint = Collisions.Intersection(plane1, plane
2, plane3);
            midPoint.Render();
            GL.Enable(EnableCap.DepthTest);
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }
```

```
        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Line

Technically a line is the intersection of 2 planes. It runs infinatley in all directions. What we're going to calla line is actually a **Line Segment**. It's a line between two points. Unlike a line, a line segment does have a length.

Given the above definition, defining a line is pretty simple:

```
class Line {
    public Point start;
    public Point end;
}
```

There is really no magic to a line. For example if you want to get the length of a line, it's the same as getting the length of a vector! You get a vector by subtracting start from end, then find the length!

Code Guide

The `Line` class is straight forward. You can follow this guide for implementation, or make your own.

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

namespace CollisionDetectionSelector.Primitives {
    class Line {
        public Point start = new Point();
        public Point end = new Point();

        public float Length { get { /*TODO*/ } }
        public float LengthSquared { get { /*TODO*/ } }
```

```
        public Line() {
            // Make a line of 0 length,
            // both points should default to (0, 0, 0)
        }

        public Line(Line other) {
            // TODO: Copy the other line
            // Pay attention, are you assigning by reference or va
lue
        }

        public Line(Point p1, Point p2) {
            // TODO
        }

        public Line(Vector3 p1, Vector3 p2) {
            // TODO
        }

        public Vector3 ToVector() {
            // TODO: return end - start
        }

        //Render and toString must be there!
        public void Render() {
            GL.Begin(PrimitiveType.Lines);
            GL.Vertex3(start.X, start.Y, start.Z);
            GL.Vertex3(end.X, end.Y, end.Z);
            GL.End();
        }

        public override string ToString() {
            string result = "Start: (" + start.X + ", " + start.Y
+ ", " + start.Z + "), ";
            result += "End: ( " + end.X + ", " + end.Y + ", " + en
d.Z + ")";
```

```
            return result;
        }
    }
}
```

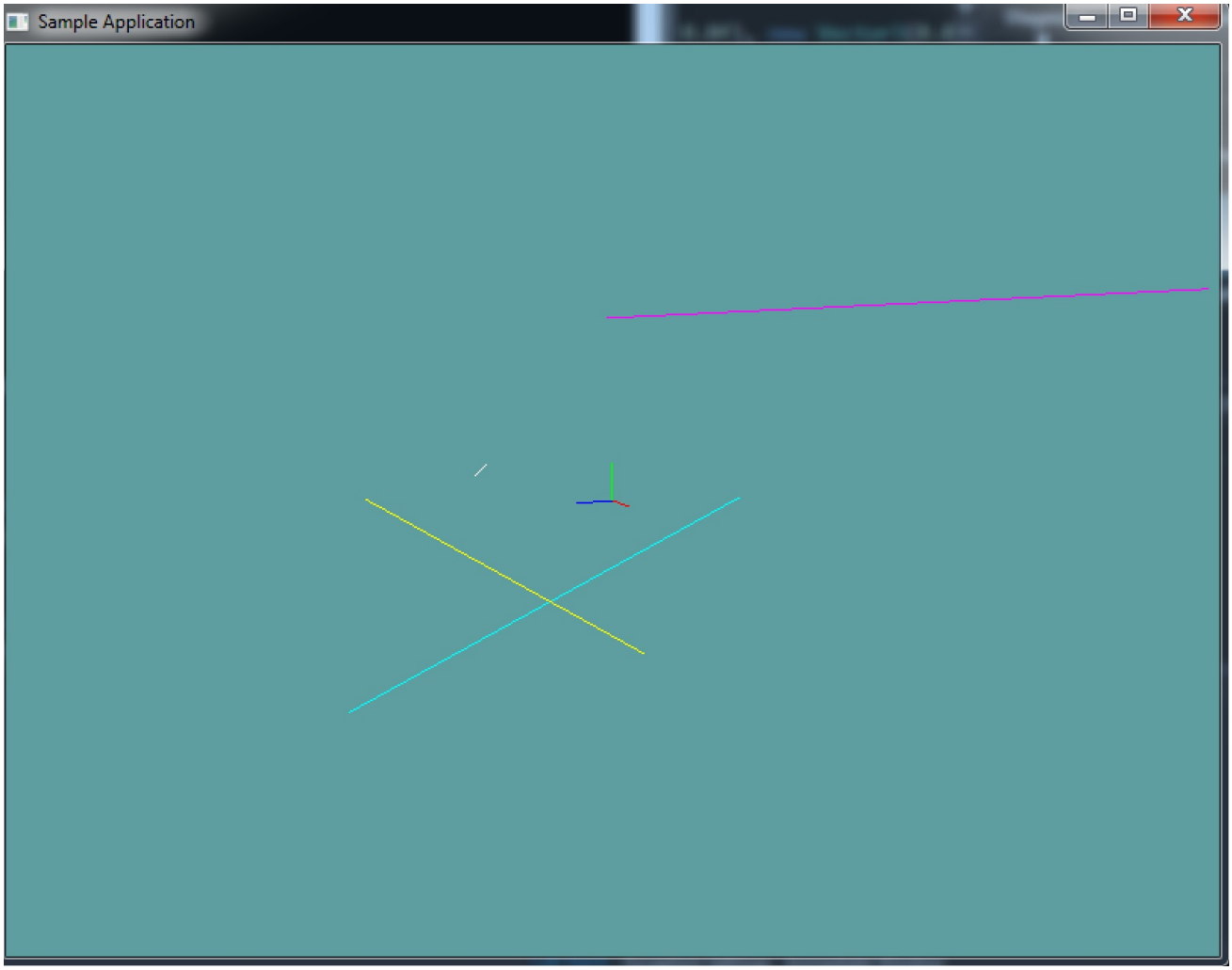
On Your Own

Implement the Line class. You can use the code guide above, or use your own implementation

Sample / Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This example is visual only, no errors will be printed to the console if the code is bad. This is what the final render should look like:




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class LineSample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Line[] origin = new Line[] {
            new Line(new Point(0, 0, 0), new Point(1, 0, 0)),
            new Line(new Point(0, 0, 0), new Point(0, 1, 0)),
            new Line(new Point(0, 0, 0), new Point(0, 0, 1))
        };
```

```
        Line[] random = new Line[] {
            new Line(new Point(-7, 5, -3), new Point(8, 5, -10)),
            new Line(new Point(-8, -8, 6), new Point(-9, -2, -10))
,
            new Line(new Point(-4, -6, -3), new Point(-9, -1, 5)),
            new Line(new Point(-6, 0, 2), new Point(-1, 1, 3))
        };

        float[][] colors = new float[][] {
            new float[] { 1f, 0f, 1f },
            new float[] { 0f, 1f, 1f },
            new float[] { 1f, 1f, 0f },
            new float[] { 1f, 1f, 1f },
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);

            /*System.Random r = new System.Random();
            string output = "new Line(new Point(" + (r.Next() % 20
 - 10) + ", " + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 - 10)
 + "), new Point(" + (r.Next() % 20 - 10) + ", " + (r.Next() % 20
- 10) + ", " + (r.Next() % 20 - 10) + ")),\n";
            for (int i = 0; i < 3; ++i) {
                output += "new Line(new Point(" + (r.Next() % 20 -
 10) + ", " + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 - 10) +
 "), new Point(" + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 -
10) + ", " + (r.Next() % 20 - 10) + ")),\n";
            }
            System.IO.File.WriteAllText(@"C:\Users\WinVPC\Desktop\
Lines.txt", output);*/
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
```

```
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            GL.Color3(1f, 0f, 0f);
            origin[0].Render();
            GL.Color3(0f, 1f, 0f);
            origin[1].Render();
            GL.Color3(0f, 0f, 1f);
            origin[2].Render();

            for (int i = 0; i < random.Length; ++i) {
                GL.Color3(colors[i][0], colors[i][1], colors[i][2]
);
                random[i].Render();
            }
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
```

```
            }
        }
    }
```

Point on line

In order to determine if a point is on a line or not we need to know the slope intercept form of a line. That is, we need to express the line as the equation:

```
y = m * x + b
```

Im this equation m and b will be given. We plug in the X and Y position of the point being tested, if the result of the equation is true, the point will be on the line!

However, we express lines by two points, a start point and an end point. Not in slope intercept form. So we have to convert our line into said form.

`m` in the above equation is the slope of the line, slope is defined as rise over run, or `rise / run` ; that is, the y of the line divided by it's x. So if we convert the line to a vector (end - start), then the slope is said vectors y divided by it's x.

```
M = (line.end.y - line.start.y) / (line.end.x - line.start.x);
```

`b` is the y-intercept. That is, it's the x value of the line where it crosses the y axis. We can find this with our slope, by subtracting the slope from the starting points y and multiplying it by the x value of the point

```
b = line.start.y - M * line.start.x;
```

If any of that is unclear, i suggest watching this khan academy video

The algorithm

Now that we can express a line in y intercept form, we can write some code to test if a point is on a line:

```
bool IsPointOnLine(Line line, Point point)  {
   float m = (line.end.y - line.start.y) / (line.end.x - line.star
t.x);
   float b = line.start.y - m * line.start.x;

   // At this point, evaluating the equation should be:
   // return point.y == m * point.x + b;
   // But that won't work, because we're working with floats
   // floating point error can accumulate, we have to use an epsil
on

   // using an epislon, we can compare the distance of the
   // equations result to 0 by the epsilon
   if (ABS(point.y - (m * point.x + b)) < EPSILON) {
       return true;
   }
   return false;

   // EPSILON is just a small number, like 0.001f;
}
```

On Your Own

Add the following function to the `Collisions` class:

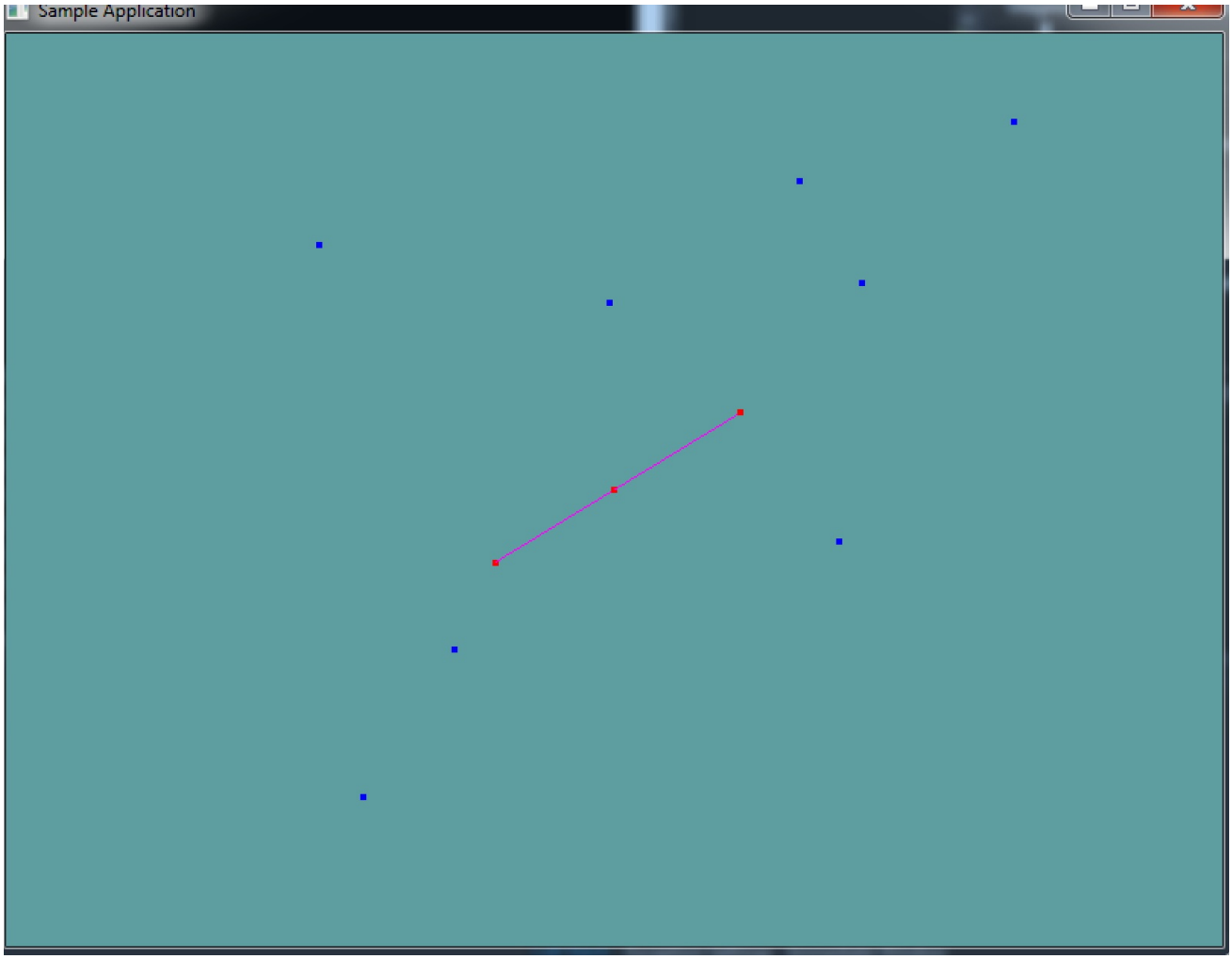
```
public static bool PointOnLine(Point point, Line line);
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The constructor of this code will spit out errors if they are present. A line is rendered, along with a random sampling of points. Any point that falls on the line is rendered in red, points not on the line are rendered in blue.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class PointOnLine : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Line testLine = new Line(new Point(-3, -2, -1), new Point(3
```

```
, 2, 1));
        Point[] testPoints = new Point[] {
            new Point(0, 0, 0),
            new Point(-3, -2, -1),
            new Point(3, 2, 1),
            new Point(7, -2, -1),
            new Point(-4, -7, -8),
            new Point(7, 8, 5),
            new Point(1, 5, -5),
            new Point(-6, 5, 7),
            new Point(1, 6, 8),
            new Point(-7, -10, -4),
            new Point(5, 5, 3)
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);

            for (int i = 0; i < 3; ++i) {
                if (!Collisions.PointOnLine(testPoints[i], testLin
e)) {
                    System.Console.ForegroundColor = System.Consol
eColor.Red;
                    System.Console.WriteLine("Expected point: " +
testPoints[i].ToString() + " to be on Line!");
                }
            }

            for (int i = 3; i < testPoints.Length; ++i) {
                if (Collisions.PointOnLine(testPoints[i], testLine
)) {
                    System.Console.ForegroundColor = System.Consol
eColor.Red;
                    System.Console.WriteLine("Expected point: " +
testPoints[i].ToString() + " to be on Line!");
                }
```

```
            }
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            GL.Color3(1f, 0f, 1f);
            testLine.Render();

            foreach (Point point in testPoints) {
                if (Collisions.PointOnLine(point, testLine)) {
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 0f, 1f);
                }
                point.Render();
            }
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Closest Point On Line

Finding the closest point on a line to another point is simple vector math. Think of the line as a vector, let's call it AB. The point is another vector, C. To get the closest point on the line to the vector we simply project C onto AB.

This works for a Line, but we don't actually have a line, we have a **line segment**. For this, you also have to interpolate the projected point. The interpolation formula is simple:

```
I(t) = START + t * (END - START)
```

`t` in the above example is time. It's how far we've interpolated. `I` is the interpolation result. `START` and `END` are self explanatory. This also works with regular numbers, let's try to interpolate 25% (0.25f) between 1 and 3:

```
// Formula
I(t) = START + t * (END - START)
// Substitute start and end
I(t) = 1 + t * (3 - 1)
// Substitute t
I(0.25) = 1 + 0.25 * (3 - 1)
// Evaluate
I(0.25) = 1.5
```

This works, let's reason it out. The range from 1 to 3 is 2. So the total distance between 1 and 3 is 2. 1/4 (0.25) of 2 is .5. If we want to interpolate 25% between 1 and 3 we add .5 to 1, resulting in 1.5.

Similarly, if you wanted to interpolate 1/2 between 1 and 3, the result would be 2. This is becuase 2 is half-way between 1 and 3.

The algorithm

The above formula will intepolate in all dimensions. Back to what i was saying, there are 3 ways to project a point (C) onto a line (AB):

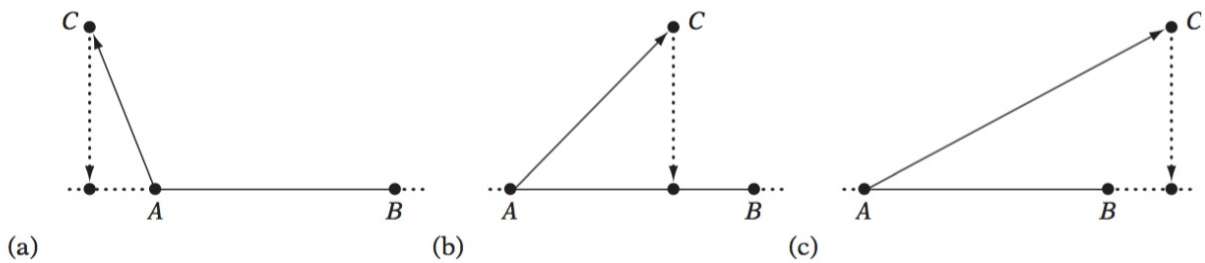


Figure 5.2 The three cases of *C* projecting onto *AB*: (a) outside *AB* on side of *A*, (b) inside *AB*, and (c) outside *AB* on side of *B*.

In the first and last example T would be **< 0** and **> 1**. In the middle example T is between 0 and 1. In order for the projected point to be ON THE SEGMENT, we need to clamp T between 0 and 1.

```
// This is Pseudo code, the types are not 100% correct,
// that is, casting is needed. But the formula is fully
// implemented for demonstration purposes

// Given segment AB and Point C, compute the closest point D
// on segment AB. Also return T where D(T) = A + T * (B - A)
Point ClosestPoint(Line ab, Point c, out float t) {
  // Break ab appart into components a and b
  Vector a = ab.start;
  Vector b = ab.end;

  // Project c onto ab, computing the
  // paramaterized position d(t) = a + t * (b - a)
  t = Dot(c - a, ab) / Dot(ab, ab);

  // Clamp T to a 0-1 range. If t was < 0 or > 1
  // then the closest point was outside the line!
  t = Clamp(t, 0f, 1f);

  // Compute the projected position from the clamped t
  Point d = new Point(a + t * ab.ToVector());

  // Return result
  return d;
}

// Provide a useful overload, we don't always care about T
Point ClosestPoint(Line ab, Point c) {
    float t = 0f;
    return ClosestPoint(ab, c, out t);
}
```

On Your Own

Add the following functions to the `Collisions` class:

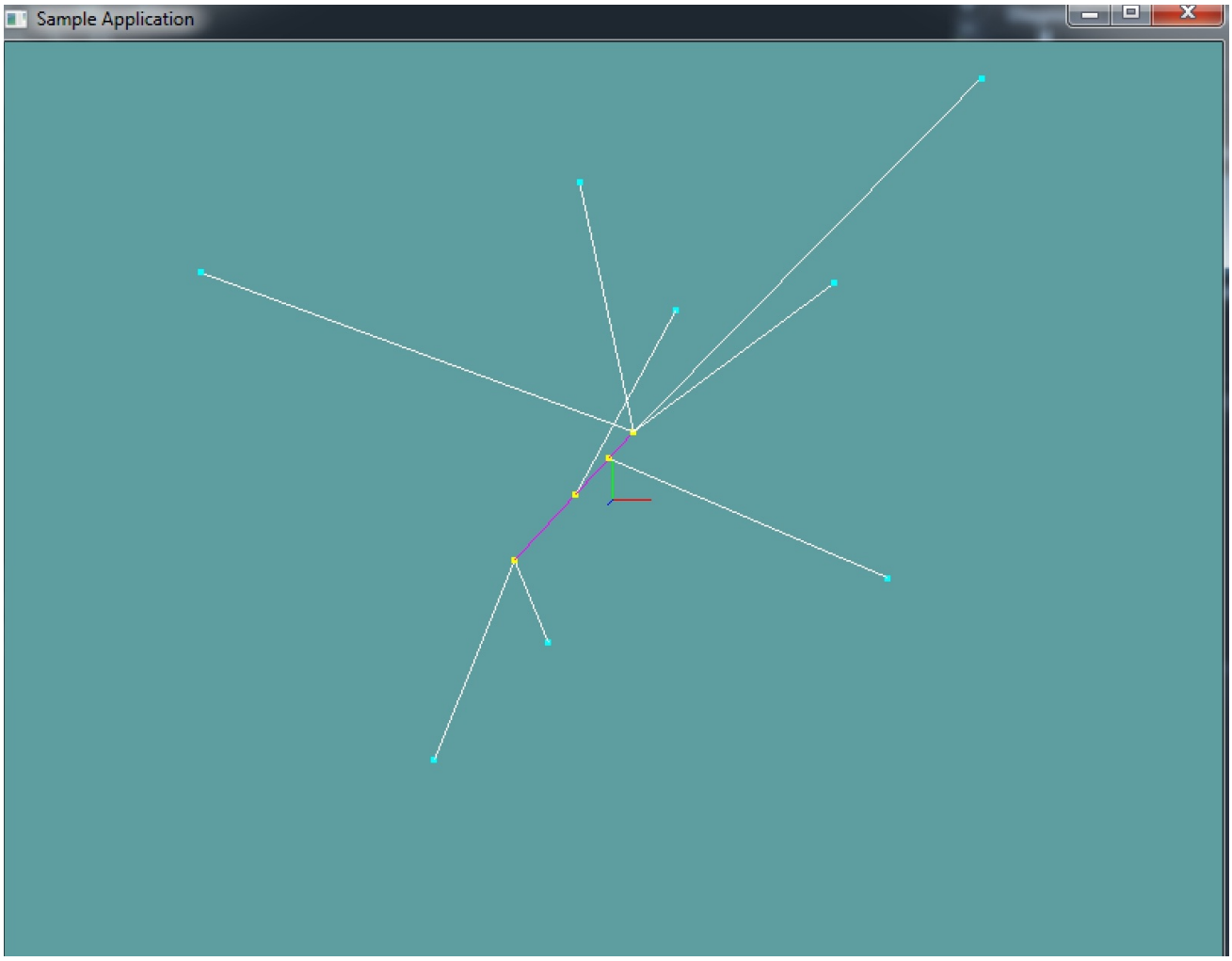
```
public static Point ClosestPoint(Line ab, Point c, out float t)
public static Point ClosestPoint(Line ab, Point c)
```

And provide an implementation for them! The second function can just call the first function. It's for conveniance, because we don't always care about t.

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This code doesn't spit out any errors. It's all visual. There is 1 line and several points. The points are projected to the line and lines are drawn between the points.



```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class ClosestPointLine : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Line testLine = new Line(new Point(-3, -2, -1), new Point(1
, 2, 3));
        Point[] testPoints = new Point[] {
            new Point(7, -2, -1),
            new Point(-4, -7, -8),
            new Point(7, 8, 5),
            new Point(1, 5, -5),
            new Point(-6, 5, 7),
            new Point(1, 6, 8),
            new Point(-7, -10, -4),
            new Point(5, 5, 3)
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
```

```
            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawOrigin();

            GL.Color3(1f, 0f, 1f);
            testLine.Render();

            GL.Color3(0f, 1f, 1f);
            foreach(Point point in testPoints) {
                point.Render();
            }

            GL.Color3(1f, 1f, 0f);
            foreach (Point point in testPoints) {
                Point closest = Collisions.ClosestPoint(testLine,
point);
                closest.Render();
            }

            GL.Color3(1f, 1f, 1f);
            foreach (Point point in testPoints) {
                Point closest = Collisions.ClosestPoint(testLine,
point);
                Line newLine = new Line(closest, point);
                newLine.Render();
            }
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
```

```
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Ray

A ray is similar to a line (not line segment). A ray is a single point in space with a direction. It extends infinatley far away in that single direction (as opposed to a line, which extends in both directions).

With the above definition, the code for a ray is pretty simple:

```
class Ray {
    public Point position;
    public Vector3 direction;
}
```

Much like line segments, rays are fairly straight forward objets. Because i ray is a directed line (not line segment tough), a lot of the collision code is going to be very similar to that of a line.

Code Guide

The `Ray` class is straight forward. You can follow this guide for implementation, or make your own. Notably, i made the direction a private member called `_normal`. I do this to ensure that it is normalized at all times. You might want to do that differently, this means you can't to this in my code:

```
ray.Normal.X = 7;
```

The reason i'm ok with that is you rarley set only a single component of a normal! More often than not you must set the whole normal anyway.

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

```
namespace CollisionDetectionSelector.Primitives {
    class Ray {
        public Point Position;
        private Vector3 _normal;

        public Vector3 Normal {
            get {
                // TODO: Returns _normal
                return _normal;
            }
            set {
                // TODO: Sets (and normalizes) _normal
                _normal = value;
                _normal.Normalize();
            }
        }

        public Ray() {
            // Make a ray that is at origin,
            // pointing down the Z axis
            Position = new Point(0, 0, 0);
            _normal = new Vector3(0, 0, 1);
        }

        public Ray(Point p, Vector3 d) {
            // Don't forget to normalize _normal
            Position = new Point(p.X, p.Y, p.Z);
            _normal = new Vector3(d.X, d.Y, d.Z);
            _normal.Normalize();
        }

        public Ray(Vector3 p, Vector3 d) {
            // Don't forget to normalize _normal
            Position = new Point(p.X, p.Y, p.Z);
            _normal = new Vector3(d.X, d.Y, d.Z);
            _normal.Normalize();
```

```
        }

        //Render and toString must be there!
        public void Render() {
            GL.Begin(PrimitiveType.Lines);
            Vector3 start = Position.ToVector();
            Vector3 end = start + _normal * 5000f;
            GL.Vertex3(start.X, start.Y, start.Z);
            GL.Vertex3(end.X, end.Y, end.Z);
            GL.End();
        }

        public override string ToString() {
            string result = "P: (" + Position.X + ", " + Position.
Y + ", " + Position.Z + "), ";
            result += "D: ( " + _normal.X + ", " + _normal.Y + ", "
 + _normal.Z + ")";
            return result;
        }
    }
}
```

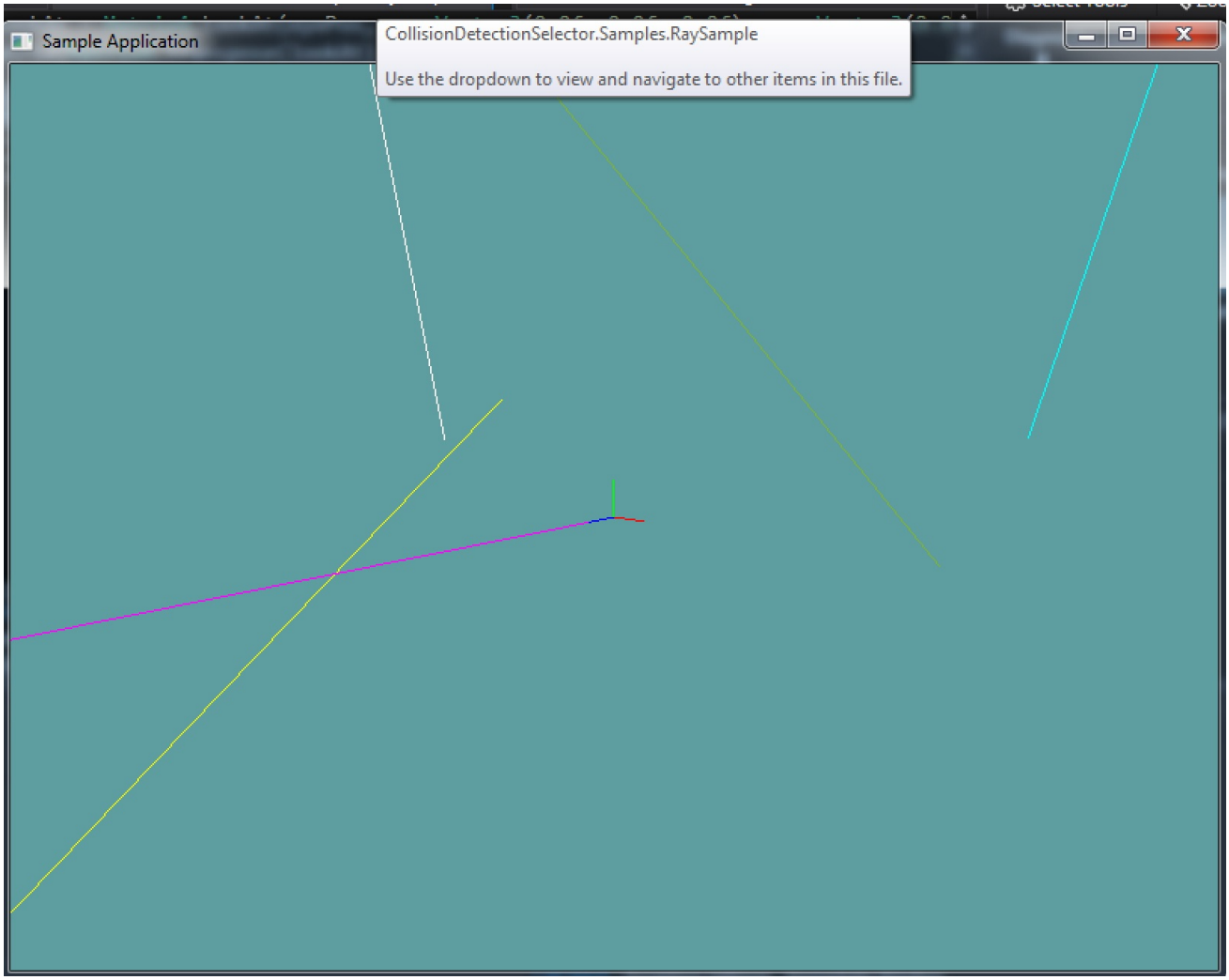
On Your Own

Implement the Ray class. You can use the code guide above, or use your own implementation

Sample / Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This example is visual only, no errors will be printed to the console if the code is bad. This is what the final render should look like:




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class RaySample : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Line[] origin = new Line[] {
            new Line(new Point(0, 0, 0), new Point(1, 0, 0)),
            new Line(new Point(0, 0, 0), new Point(0, 1, 0)),
            new Line(new Point(0, 0, 0), new Point(0, 0, 1))
        };
```

```
        Ray[] random = new Ray[] {
            new Ray(),
            new Ray(new Point(8, 2, -7), new Vector3(1, 7, -3)),
            new Ray(new Point(-2, 3, 2), new Vector3(-4, -5, 3)),
            new Ray(new Vector3(-3, 2, 3), new Vector3(0, 5, 1)),
            new Ray(new Vector3(7, -1, -4), new Vector3(-9, 5, -8)
),
        };

        float[][] colors = new float[][] {
            new float[] { 1f, 0f, 1f },
            new float[] { 0f, 1f, 1f },
            new float[] { 1f, 1f, 0f },
            new float[] { 1f, 1f, 1f },
            new float[] { 0.5f, 0.7f, 0.2f },
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);

            /*System.Random r = new System.Random();
            string output = "new Ray(new Point(" + (r.Next() % 20
- 10) + ", " + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 - 10)
+ "), new Vector3(" + (r.Next() % 20 - 10) + ", " + (r.Next() % 20
 - 10) + ", " + (r.Next() % 20 - 10) + ")),\n";
            for (int i = 0; i < 3; ++i) {
                output += "new Ray(new Point(" + (r.Next() % 20 -
10) + ", " + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 - 10) +
"), new Vector3(" + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 -
 10) + ", " + (r.Next() % 20 - 10) + ")),\n";
            }
            System.IO.File.WriteAllText(@"C:\Users\WinVPC\Desktop\
Rays.txt", output);*/
        }

        public override void Render() {
```

```
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            GL.Color3(1f, 0f, 0f);
            origin[0].Render();
            GL.Color3(0f, 1f, 0f);
            origin[1].Render();
            GL.Color3(0f, 0f, 1f);
            origin[2].Render();

            for (int i = 0; i < random.Length; ++i) {
                GL.Color3(colors[i][0], colors[i][1], colors[i][2]
);
                random[i].Render();
            }
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
```

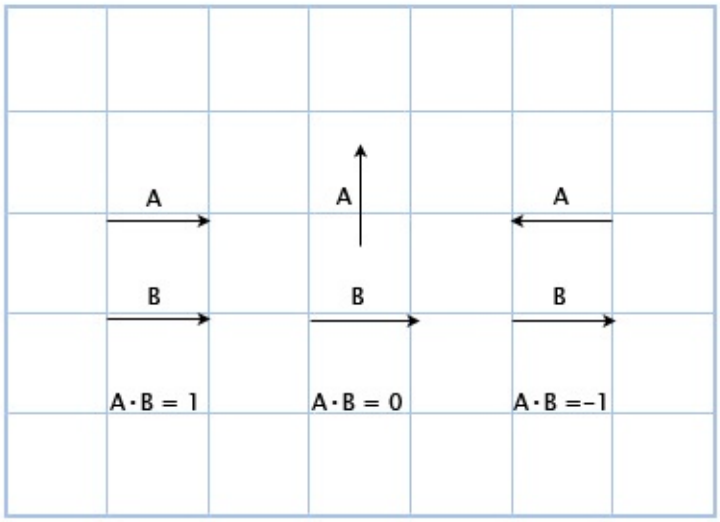
```
                GL.MatrixMode(MatrixMode.Modelview);
                GL.LoadIdentity();
            }
        }
    }
```

Point on ray

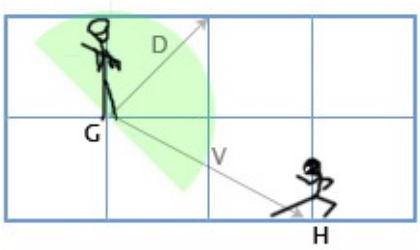
Determining if a point is on a ray or not comes down to the dot product. The key here is to realize that given the direction of the ray we have a normal vector. If we subtract the point from the ray, and normalize the resuling vector, we have a normal from the origin of the ray to the test point. At this point we have two normals. Remember, the dot product of two normals is:

- **0** if the two vectors point in the same direction
  - That is to say, if the vectors are perpendicular
- **positive** If the vectors point in the same direction
- **negative** If the vectors point in opposing direction

Of course you need to use a very small epsilon here, because of floating point error. A visual example of the above bullet points:



In a more practical example, if we know the forward vector of a guard, a player is in the guards view if the dot product of the guards forward and the vector from the guard to the player are positive.



Now, there is one edge case to all this. What happens if the point we are testing and the origin of the ray are the same point? Well, the dot product breaks down and we get no new normal vector. But that point is on the ray. You should add a special case if check for that.

See if you can implement this on your own before looking at the code below.

The algorithm

Implementing the above in code is fairly straight forward:

```
bool PointOnRay(Point point, Ray ray) {
    // If point and ray are the same, return true

    Vector3 newNorm = point - ray.Position;
    newNorm.Normalize();

    float d = Vector3.Dot(newNorm, ray.Normal);
    return Math.Abs(1f - d) < 0.000001f; // SUPER SMALL EPSILON!
}
```

On Your Own

Add the following function to the `Collisions` class:

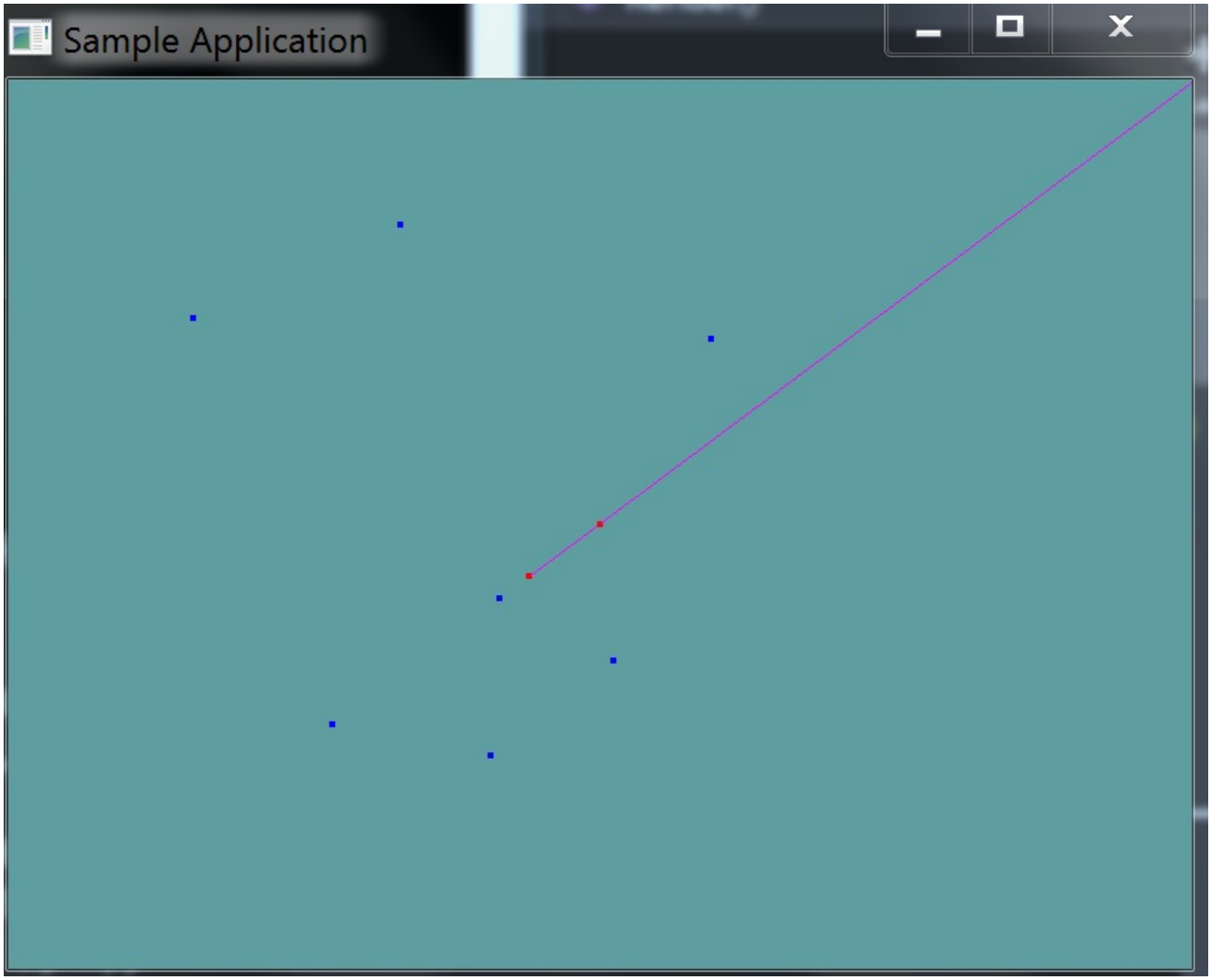
```
public static bool PointOnRay(Point point, Ray ray)
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The constructor of this code will spit out errors if they are present. A ray is rendered, along with a random sampling of points. Any point that falls on the ray is rendered in red, points not on the ray are rendered in blue.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
```

```
    class PointOnray : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Ray testRay = new Ray(new Point(-3, -2, -1), new Vector3(3
, 2, 1));
        Point[] testPoints = new Point[] {
            new Point(-3, -2, -1),
            new Point(12, 8, 4),
            new Point(0, 0, 0),
            new Point(-18, -12, -6),
            new Point(-4, -7, -8),
            new Point(7, 8, 5),
            new Point(1, 5, -5),
            new Point(-6, 5, 7),
            new Point(1, 6, 8),
            new Point(-7, -10, -4),
            new Point(-4.5f, -3f, -1.5f)
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);

            for (int i = 0; i < 3; ++i) {
                if (!Collisions.PointOnRay(testPoints[i], testRay)
) {
                    System.Console.ForegroundColor = System.Consol
eColor.Red;
                    System.Console.WriteLine("Expected point: " +
testPoints[i].ToString() + " to be on Ray!");
                }
            }

            for (int i = 3; i < testPoints.Length; ++i) {
                if (Collisions.PointOnRay(testPoints[i], testRay))
```

```
 {
                    System.Console.ForegroundColor = System.Consol
eColor.Red;
                    System.Console.WriteLine("Expected point: " +
testPoints[i].ToString() + " to NOT be on Ray!");
                }
            }
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            GL.Color3(1f, 0f, 1f);
            testRay.Render();

            foreach (Point point in testPoints) {
                if (Collisions.PointOnRay(point, testRay)) {
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 0f, 1f);
                }
                point.Render();
            }
        }

        public override void Update(float deltaTime) {
```

```
            cameraAngle.X += 45.0f * deltaTime;
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Closest point on Ray

Finding the closest point on a ray is similar to finding the closest point on a line! When we check if a point is on a line segment, we do so by projecting the point onto an infinate line. Then, we clamp the projection to a range of 0 to 1. The 0 to 1 range is what makes the check be on a line segment.

A ray goes infinatley in one direction. If we change the formula to only cap to 0, the same formula will work! Becuase a ray has no length, the interpolated t value is useless.

Now you might think to yourself, a point has a start and end, a ray only has a start and direction, how do i get an end point?!?! Well, because we don't cap the interpolation at 1, the end point doesn't matter. It just has to be some point on the ray. You can easily get some point by adding the normal to the origin of the ray.

If the above doesn't sound familiar, re-read Closest Point On Line and make sure you understand the algorithm behind it! Call me if you have questions!

The algorithm

Again, it's so similar to the closest point on line algorithm, i just copied it and made some subtle changes. You shouldn't even need this guide, you should be able to make the same changes based on the above paragraph.

```
// This is Pseudo code, the types are not 100% correct,
// that is, casting is needed. But the formula is fully
// implemented for demonstration purposes

Point ClosestPoint(Ray r, Point c) {
  // t is now local, there is no need to pass it out of the functi
on
  float t = 0f;
  // Construct a line segment out of the ray
  Line ab = new Line(r.Position, r.Position + r.Normal);
  // Break ray into a (start) and b (end) components
  Vector a = r.Position;
  Vector b = r.Position + r.Normal;

  // Project c onto ab, computing the
  // paramaterized position d(t) = a + t * (b - a)
  t = Dot(c - a, ab) / Dot(ab, ab);

  // We only want to clamp t in the positive direction.
  // The ray extends infinatley in this direction!
  t = Max(t, 0f);

  // Compute the projected position from the clamped t
  // Notice we multiply r.Normal by t, not AB.
  // This is becuase we want the ray in the direction
  // of the normal, which technically the line segment is
  // but this is much more explicit and easy to read.
  Point d = new Point(a + t * r.Normal);

  // Return result
  return d;
}
```

On Your Own

Add the following function to the `Collisions` class:

```
public static Point ClosestPoint(Ray r, Point c)
```

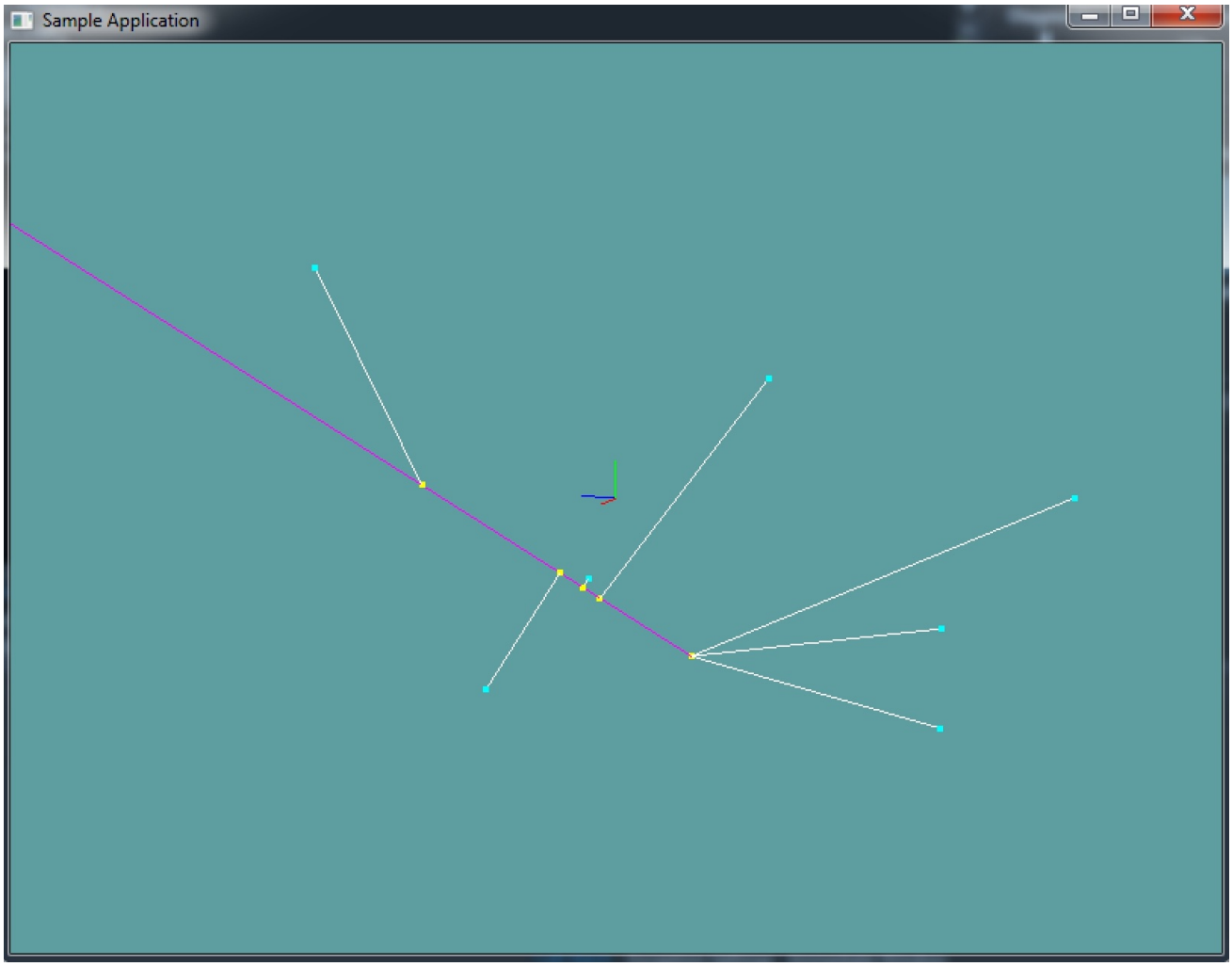
And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This code doesn't spit out any errors. It's all visual. There is 1 ray and several points. The points are projected to the ray and lines are drawn between the points.

This will look very similar to the Closest Point On Line Sample!



```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class ClosestRayLine : Application {
        protected Vector3 cameraAngle = new Vector3(120.0f, -10f,
20.0f);
        protected float rads = (float)(System.Math.PI / 180.0f);

        Ray testRay = new Ray(new Point(-10, -8, 1), new Vector3(1
, 2, 3));
        Point[] testPoints = new Point[] {
            new Point(-8, -5, -8),
            new Point(-6, -9, 8),
            new Point(-9, -5, 5),
            new Point(-4, 3, -3),
            new Point(2, 1, -10),
            new Point(8, 5, 2),
            new Point(4, -3, -8)
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);

            /*System.Random r = new System.Random();
            string output = "new Point(" + (r.Next() % 20 - 10) +
", " + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 - 10) + "),\n"
;
            for (int i = 0; i < 6; ++i) {
                output += "new Point(" + (r.Next() % 20 - 10) + ",
 " + (r.Next() % 20 - 10) + ", " + (r.Next() % 20 - 10) + "),\n";
            }
            System.IO.File.WriteAllText(@"C:\Users\WinVPC\Desktop\
Points.txt", output);*/
```

```
        }

        public override void Render() {
            Vector3 eyePos = new Vector3();
            eyePos.X = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));
            eyePos.Y = cameraAngle.Z * -(float)System.Math.Sin(cam
eraAngle.Y * rads);
            eyePos.Z = -cameraAngle.Z * (float)System.Math.Cos(cam
eraAngle.X * rads * (float)System.Math.Cos(cameraAngle.Y * rads));

            Matrix4 lookAt = Matrix4.LookAt(eyePos, new Vector3(0.0
f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));
            GL.LoadMatrix(Matrix4.Transpose(lookAt).Matrix);

            DrawOrigin();

            GL.Color3(1f, 0f, 1f);
            testRay.Render();

            GL.Color3(0f, 1f, 1f);
            foreach (Point point in testPoints) {
                point.Render();
            }

            GL.Color3(1f, 1f, 0f);
            foreach (Point point in testPoints) {
                Point closest = Collisions.ClosestPoint(testRay, p
oint);
                closest.Render();
            }

            GL.Color3(1f, 1f, 1f);
            foreach (Point point in testPoints) {
                Point closest = Collisions.ClosestPoint(testRay, p
oint);
                Line newLine = new Line(closest, point);
```

```
                newLine.Render();
            }
        }

        public override void Update(float deltaTime) {
            cameraAngle.X += 45.0f * deltaTime;
        }

        protected void DrawOrigin() {
            GL.Begin(PrimitiveType.Lines);
            GL.Color3(1f, 0f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(1f, 0f, 0f);
            GL.Color3(0f, 1f, 0f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 1f, 0f);
            GL.Color3(0f, 0f, 1f);
            GL.Vertex3(0f, 0f, 0f);
            GL.Vertex3(0f, 0f, 1f);
            GL.End();
        }

        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, aspect,
0.01f, 1000.0f);
            GL.LoadMatrix(Matrix4.Transpose(perspective).Matrix);
            GL.MatrixMode(MatrixMode.Modelview);
            GL.LoadIdentity();
        }
    }
}
```

Intermission

This is a good stopping point to stop and reflect on your code-base. Make any changes that you feel are necessary for moving forward, then proceed. Iteration and reflection like this is key to writing clean, well organized code.

Application Behaviours

One of the things that bothers me is that all of our sample code contains a lot of copy/paste code. So i want to move a lot of that into the base `Application` class. The things i really want to move are: Rendering the origin, resizing the window and making the camera rotate.

Additionally, the unit tests write red text constantly, i want to add a LogError function to make writing unit tests less verbose.

We have to move this functionality into the `Application` class without breaking the existing samples. I propose we just copy / paste the code into the implementation of the base class, with new variable names.

This way, new functions can call the inherited base class functions to do what we want them to, but because old code doesn't call the base functions (for example, `base.Update(dt);`) it will not be affected.

The new `Application` class becomes:

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

// Lives in global namespace
public class Application {
    // We need to fully scope Math_Implementation.Vector3
```

```
    // because a Vector3 class exists in the OpenTK namespace
    private Math_Implementation.Vector3 _baseCameraAngle =
        new Math_Implementation.Vector3(120.0f, -10f, 20.0f);
    private float _baseRads = (float)(System.Math.PI / 180.0f);

    public static Application Instance {
        get; private set;
    }

    public OpenTK.GameWindow Window = null;

    public Application() {
        Instance = this;
    }

    #region Common methods
    public void LogError(string error) {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Error: " + error);
        Console.ResetColor();
    }

    public void DrawOrigin() {
        GL.Begin(PrimitiveType.Lines);
        GL.Color3(1f, 0f, 0f);
        GL.Vertex3(0f, 0f, 0f);
        GL.Vertex3(1f, 0f, 0f);
        GL.Color3(0f, 1f, 0f);
        GL.Vertex3(0f, 0f, 0f);
        GL.Vertex3(0f, 1f, 0f);
        GL.Color3(0f, 0f, 1f);
        GL.Vertex3(0f, 0f, 0f);
        GL.Vertex3(0f, 0f, 1f);
        GL.End();
    }
    #endregion
```

```
    /// These are the logic driving functions that classes whom su
bclass application
    /// will need to override. There is no need to call the base v
ersions!
    #region Inheritable Logic
    public virtual void Intialize(int width, int height) {

    }

    // Has a default implementation, making the need to override i
n child
    // classes purley optional
    public virtual void Resize(int width, int height) {
        GL.Viewport(0, 0, width, height);
        GL.MatrixMode(MatrixMode.Projection);
        float aspect = (float)width / (float)height;
        Math_Implementation.Matrix4 perspective =
            Math_Implementation.Matrix4.Perspective(60, aspect, 0.
01f, 1000.0f);
        GL.LoadMatrix(Math_Implementation.Matrix4.Transpose(perspe
ctive).Matrix);
        GL.MatrixMode(MatrixMode.Modelview);
        GL.LoadIdentity();
    }

    // If the example wants a default rotating camera, it needs to
 call base.Update();
    public virtual void Update(float deltaTime) {
        _baseCameraAngle.X += 45.0f * deltaTime;
    }

    // If the example wants a default rotating camera, it needs to
 call base.Render();
    public virtual void Render() {
        // Again, Vector3 and Matrix4 must be fully scoped, because

        // the OpenTK namespace contains both classes already
```

```
        Math_Implementation.Vector3 eyePos = new Math_Implementati
on.Vector3();
        eyePos.X = _baseCameraAngle.Z * -(float)System.Math.Sin(_b
aseCameraAngle.X * _baseRads * (float)System.Math.Cos(_baseCameraA
ngle.Y * _baseRads));
        eyePos.Y = _baseCameraAngle.Z * -(float)System.Math.Sin(_b
aseCameraAngle.Y * _baseRads);
        eyePos.Z = -_baseCameraAngle.Z * (float)System.Math.Cos(_b
aseCameraAngle.X * _baseRads * (float)System.Math.Cos(_baseCameraA
ngle.Y * _baseRads));

        Math_Implementation.Matrix4 lookAt =
            Math_Implementation.Matrix4.LookAt(eyePos,
            new Math_Implementation.Vector3(0.0f, 0.0f, 0.0f),
            new Math_Implementation.Vector3(0.0f, 1.0f, 0.0f));
        GL.LoadMatrix(Math_Implementation.Matrix4.Transpose(lookAt
).Matrix);
    }

    public virtual void Shutdown() {

    }
    #endregion

    /// Entery point of the application, by default it creates a n
ew window
    /// Your subclasses should not have to override this, but if y
ou want a
    /// sample that does not have a window, you need to.
    #region Entery Point
    public virtual void Main(string[] args) {
        Window = new OpenTK.GameWindow();
        Window.Load += new EventHandler<EventArgs>(OpenTKInitializ
e);
        Window.UpdateFrame += new EventHandler<FrameEventArgs>(Ope
nTKUpdate);
        Window.RenderFrame += new EventHandler<FrameEventArgs>(Ope
```

```
nTKRender);
        Window.Unload += new EventHandler<EventArgs>(OpenTKShutdow
n);

        Window.Title = "Sample Application";
        Window.ClientSize = new System.Drawing.Size(800, 600);
        Instance.Resize(800, 600);
        Window.VSync = VSyncMode.On;
        Window.Run(60.0f);

        Window.Dispose();
    }
    #endregion

    /// If the default entery point (Main function) is used, these
 are the OpenTK callbacks
    /// Which are issued, they in turn call member functions that
subclasses need to override
    #region OpenTK Callbacks
    protected void OpenTKInitialize(object sender, EventArgs e) {
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.DarkGray;
        Console.WriteLine("OpenGL Vendor: " + GL.GetString(StringN
ame.Vendor));
        Console.WriteLine("OpenGL Renderer: " + GL.GetString(Strin
gName.Renderer));
        Console.WriteLine("OpenGL Version: " + GL.GetString(String
Name.Version));
        Console.ResetColor();
        Instance.Intialize(800, 600);
    }

    protected void OpenTKUpdate(object sender, FrameEventArgs e) {
        float dTime = (float)e.Time;
        Instance.Update(dTime);
    }
```

```
    protected void OpenTKRender(object sender, FrameEventArgs e) {
        GL.ClearColor(Color.CadetBlue);
        GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.
DepthBufferBit | ClearBufferMask.StencilBufferBit);
        Instance.Render();
        Window.SwapBuffers();
    }

    protected void OpenTKShutdown(object sender, EventArgs e) {
        Instance.Shutdown();
    }

    protected void OpenTKResize(EventArgs e) {
        Instance.Resize(Instance.Window.Width, Instance.Window.Hei
ght);
    }
    #endregion
}
```

Collision Handling Code

I noticed you have different files to handle different collision types:

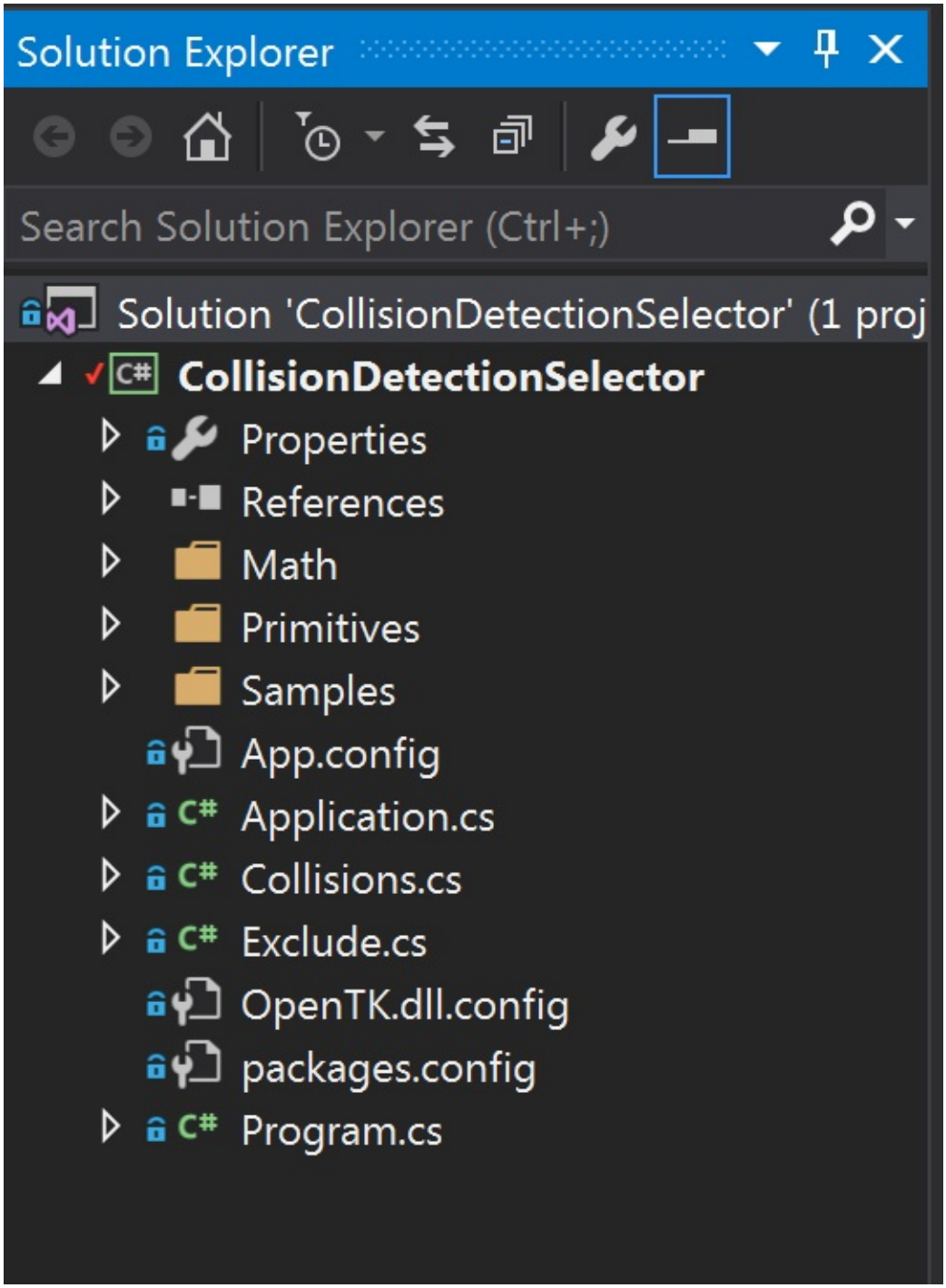
- AABBCollisions.cs
- PlaneCollision.cs
- SphereCollisions.cs

I only have 1 file

- Collisions.cs

And it only contains 1 class named `Collisions`, this is where i put all my collision functions.

I'm not saying you need to change anything, you should proceed with whatever method makes more sense to you! I just wanted to point out what my code is looking like. This is what my hierarchy is:



Take a minute, make sure your hierarchy is one you are happy with and that you are happy with how your code is formatted.

Intersections

Now that we've defined the basic primitives we are going to work with, let's get started on some intersection tests. An intersection test is essentially asking, is this spehere intersecting that AABB in any way?

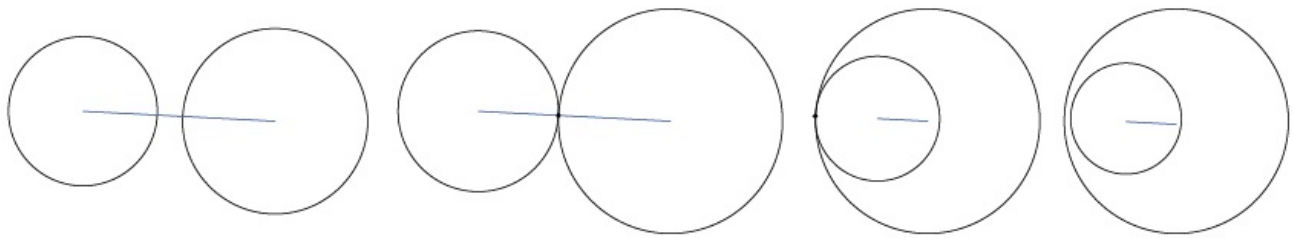
For intersection testing, we're going to focus on solids. Line and Ray tests (Ray casting and Line casting) are going to get their own section.

It's also important to understand that the tests for static objects and dynamic objects are not necesarley the same. More on this in sections to come.

Sphere Sphere Intersection

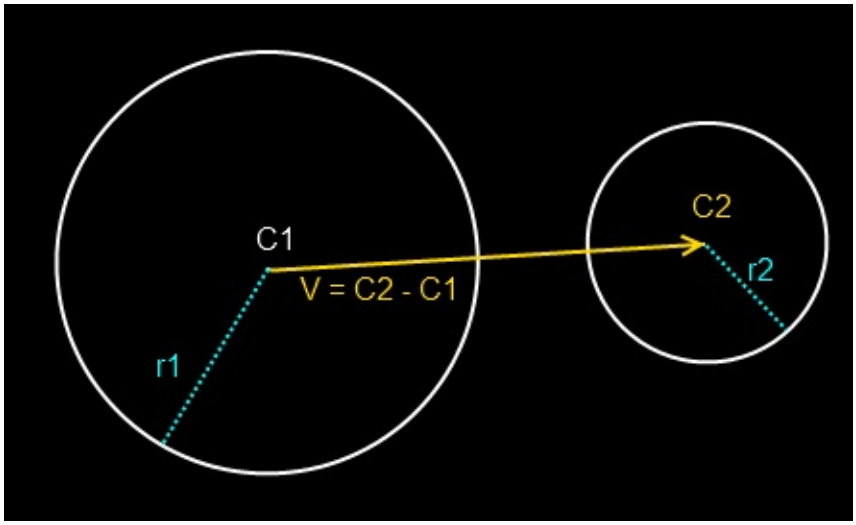
The overlap test between two spheres is very simple. The distance between the sphere centers is computed and compared against the sum of the sphere radii. To avoid an often expensive square root operation, the squared distances are compared.

The test looks like this:



The blue line is the distance between the center of the spheres. In the first example this line is greater than the sum radii of the spheres, they do not intersect. The second example, it's equal, they intersect. The third and fourth examples see the distance be less than the sum of the radii as well, they also intersect.

The above example fails to show the radii of the spheres, in the below image, the yellow line does that.



The Algorithm

The code to test intersecting spheres is super straight forward.

```
bool TestSphereSphere(Sphere a, Sphere b) {
    // Calculate squared distance between centers
    Vector d = a.position - b.position;
    float squaredDistance = Dot(d, d); // Same as d.LengthSquared()


    // Spheres intersect if squared distance is less than squared
sum of radii
    float radiusSum = a.radius + b.radius;
    float squaredRadii = radiusSum * radiusSum;

    return squaredDistance <= squaredRadii;
}
```

On Your Own

Add the following function to the `Collisions` class:

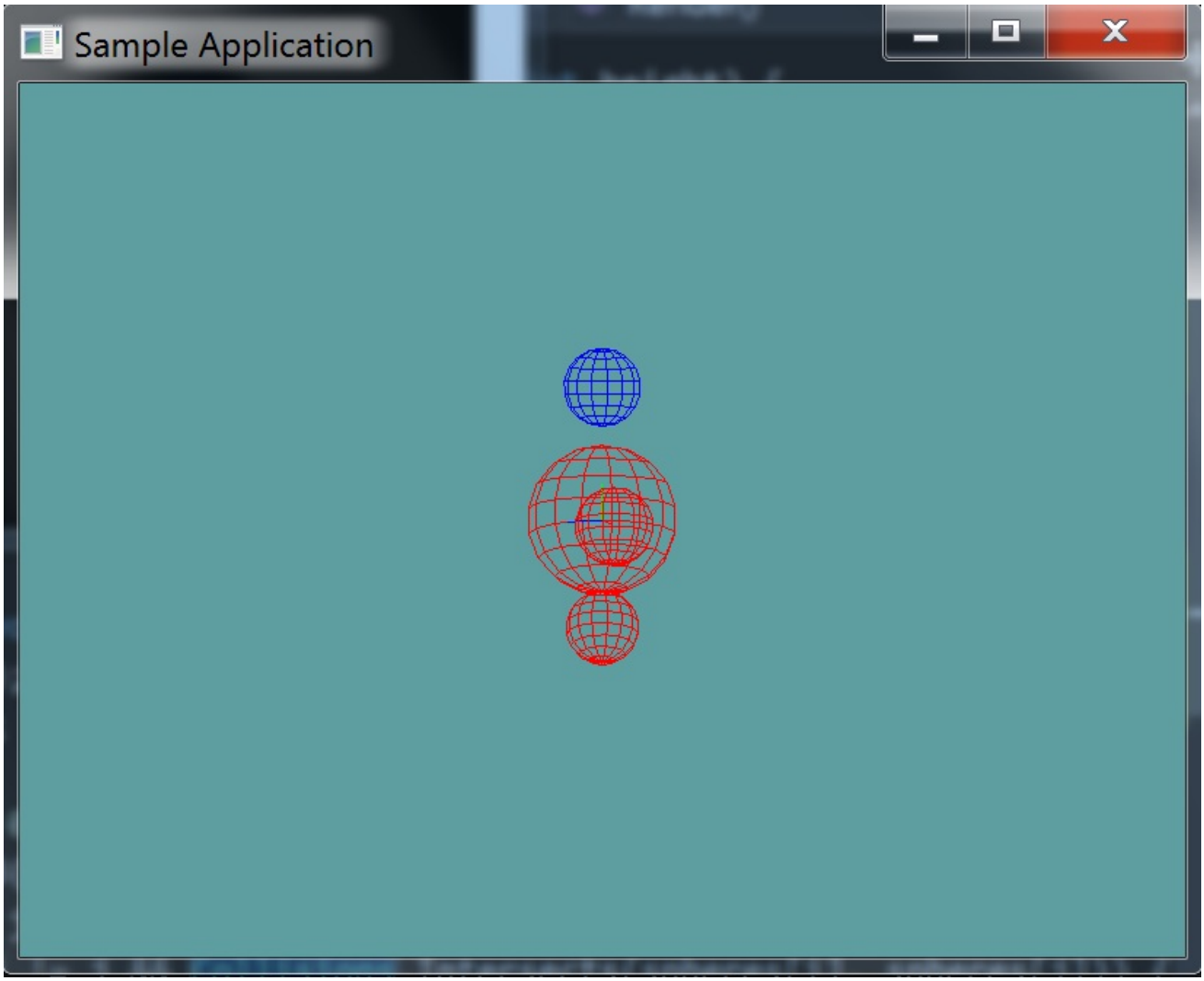
```
public static bool Intersects(Sphere s1, Sphere s2)
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will render 4 spheres, in verious test cases. Any sphere that intersects another sphere is rendered as red. If a sphere does not intersect any other spheres, it will render blue. The constructor will log error messages.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class SphereSphereIntersection : Application{
        Sphere[] spheres = new Sphere[] {
            null, null, null, null // Size = 4
        };
```

```
        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            spheres[0] = new Sphere(new Point(0f, 0f, 0f), 2f);
            spheres[1] = new Sphere(new Point(0f, 3.5f, 0f), 1f);
            spheres[2] = new Sphere(new Point(0f, -3f, 0f), 1f);
            spheres[3] = new Sphere(new Point(1f, 0f, 0f), 1f);

            bool[] results = new bool[] {
                true, false, true, true, false, true, false, false
, true,
                false, true, false, true, false, false, true
            };
            int t = 0;

            for (int i = 0; i < spheres.Length; ++i) {
                for (int j = 0; j < spheres.Length; ++j) {
                    if (Collisions.Intersects(spheres[i], spheres[
j]) != results[t++]) {
                        LogError("sphere " + i + " and " + j + " s
hould " +
                            (results[t - 1] ? "" : "not ") + "inte
rsect"
                        );
                    }
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            for (int i = 0; i < spheres.Length; ++i) {
```

```
                GL.Color3(0f, 0f, 1f);
                for (int j = 0; j < spheres.Length; ++j) {
                    if (i != j && Collisions.Intersects(spheres[i]
, spheres[j])) {
                        GL.Color3(1f, 0f, 0f);
                        break;
                    }
                }
                spheres[i].Render();
            }
        }
    }
}
```

Sphere AABB Intersection

Testing whether a sphere intersects an axis-aligned bounding box is best done by computing the distance between the sphere center and the AABB. This is done using the "Closest Point To AABB" function, with the sphere center being the point.

Once you have the closest point compare its distance with the radius of the sphere. `Distance = Sphere.Position - ClosestPoint` If the distance is less than the radius, the sphere and AABB must be intersecting.

To avoid expensive square root operations, both distance and radius can be squared before the comparison is made without changing the result of the test.

The Algorithm

Again, this is pseudo-code. The types might not match up to what you expect them to be.

```
bool Intersects(Sphere sphere, AABB aabb) {
    Vector3 closestPoint = ClosestPoint(aabb, sphere.Position);
    Vector3 differenceVec = sphere.Position - closestPoint;

    float distanceSquared = Vector3.LengthSquared(differenceVec);
    float radiusSquared = sphere.Radius * sphere.Radius;

    return distanceSquared < radiusSquared;
}
```

On Your Own

Add the following function to the `Collisions` class:

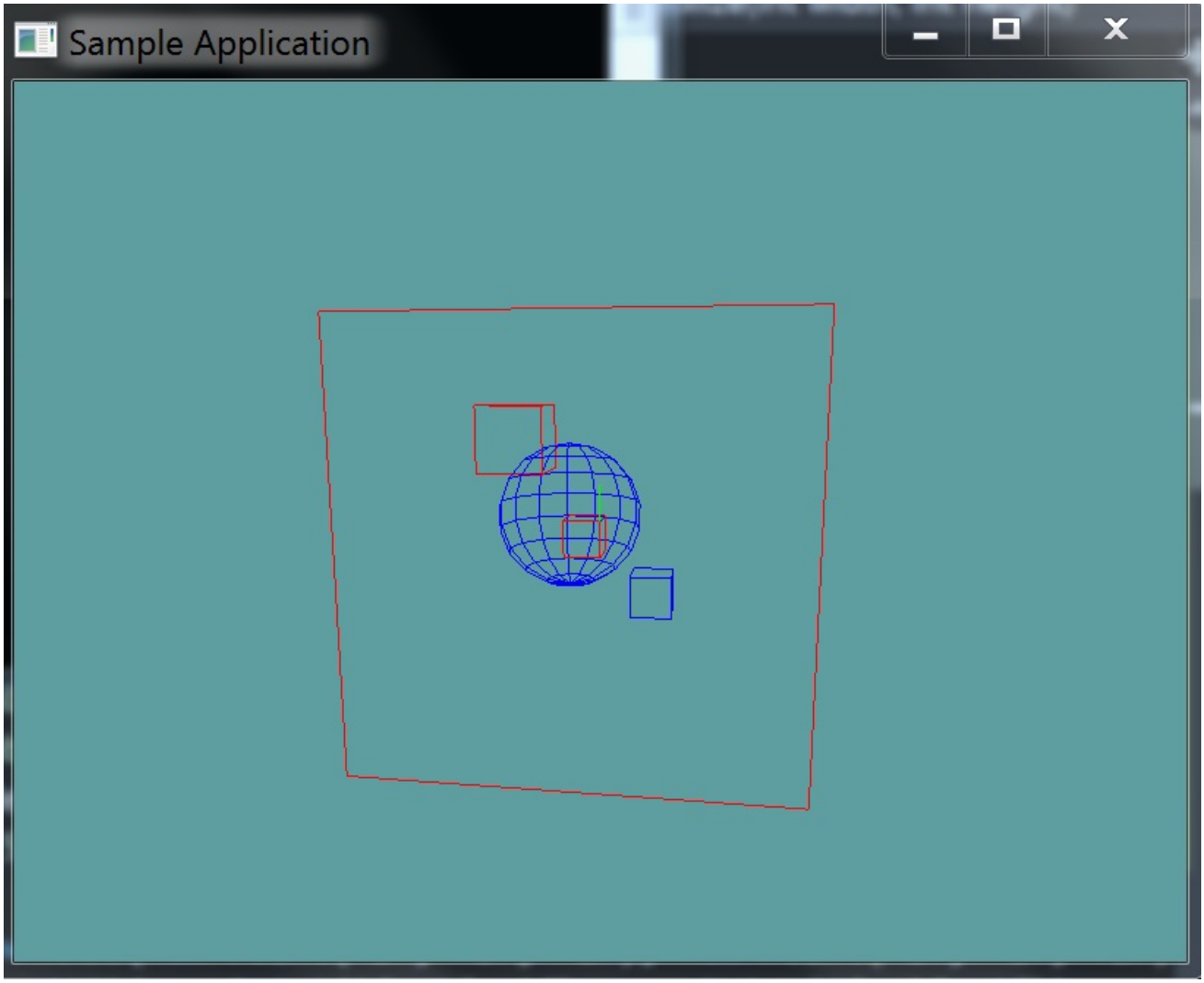
```
// TODO: Implement this
public static bool Intersects(Sphere sphere, AABB aabb)

// Just a conveniance function, so argument order wont matter!
public static bool Intersects(AABB aabb, Sphere sphere) {
    return Intersection(sphere, aabb);
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit tests will spit out some errors in the constructor if your intersection code is returning the wrong values. There is a blue sphere rendered and 4 cubes. 3 of the cubes intersect the sphere. Any intersecting cubes are rendered red, while non-intersecting ones are rendered blue.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class SphereAABBIntersection : Application {
        Sphere test = new Sphere(new Point(1f, 0f, 1f), 2f);

        AABB[] aabbs = new AABB[] {
            null, null, null, null // Size = 4
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
```

```
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.PointSize(5f);

            aabbs[0] = new AABB(new Point(-2f, -2f, -2f), new Poin
t(-1f, -1f, -1f));
            aabbs[1] = new AABB(new Point(2f, 1f, 2f), new Point(4
f, 3f, 4f));
            aabbs[2] = new AABB(new Point(1f, 0f, 1f), new Point(0
f, -1f, 0f));
            aabbs[3] = new AABB(new Point(5f, 5f, 5f), new Point(-5
f, -5f, -5f));

            bool[] results = new bool[] {
                false, true, true, true
            };
            int t = 0;

            for (int i = 0; i < aabbs.Length; ++i) {
                if (Collisions.Intersects(aabbs[i], test) != resul
ts[t++]) {
                    LogError("Expected aabb " + i + " to " +
                        (results[t - 1] ? "intersect" : "not inter
sect") +
                    " the sphere");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            for (int i = 0; i < aabbs.Length; ++i) {
                GL.Color3(0f, 0f, 1f);
                if (Collisions.Intersects(test, aabbs[i])) {
```

```
                    GL.Color3(1f, 0f, 0f);
                }
                aabbs[i].Render();
            }

            GL.Color3(0f, 0f, 1f);
            test.Render();
        }
    }
}
```

Sphere Plane Intersection

This is a pretty simple intersection logic, like with the Sphere-AABB intersection, we've already written the basic checks to support it. To see if a sphere and plane intersect:

1. Find the closest point on the plane to the sphere
2. Make sure the distance of that point is <= than the sphere radius

That's it. Of course you might want to check length squared against radius squared to save on some performance.

The Algorithm

This is a super simple algorithm (about 3 lines), try implementing it without a code guide.

On Your Own

Add the following function to the `Collisions` class:

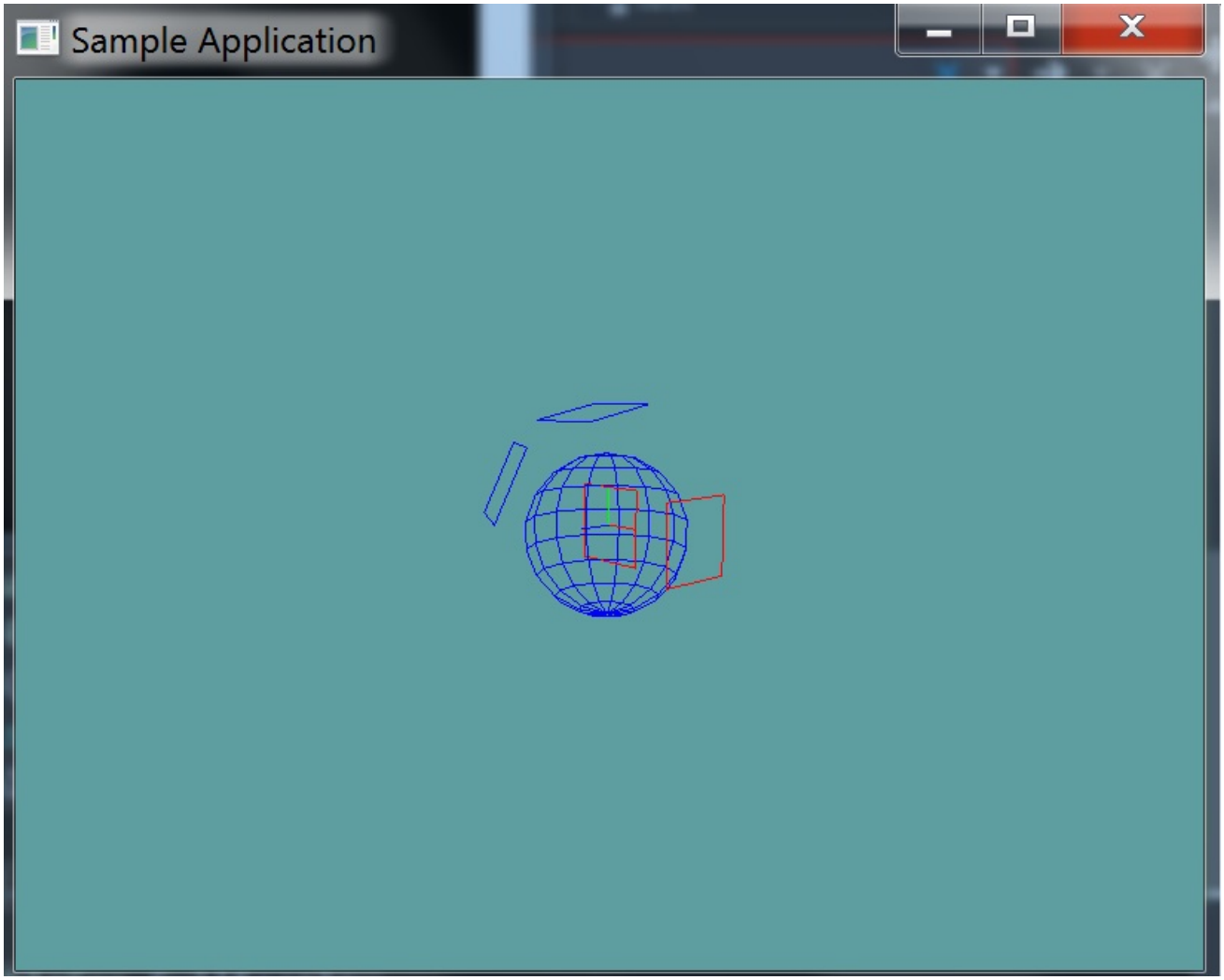
```
// TODO: Provide implementation
public static bool Intersects(Sphere sphere, Plane plane)

// Conveniance function
public static bool Intersects(Plane plane, Sphere sphere) {
    return Intersects(sphere, plane);
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will spit out minimal errors if your collision results are not as expected. It will render a blue sphere and several planes. Planes that intersect the sphere are rendered in red, planes that don't are rendered in blue.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class SpherePlaneIntersection : Application {
        Sphere test = new Sphere(new Point(1f, 0f, 1f), 2f);
```

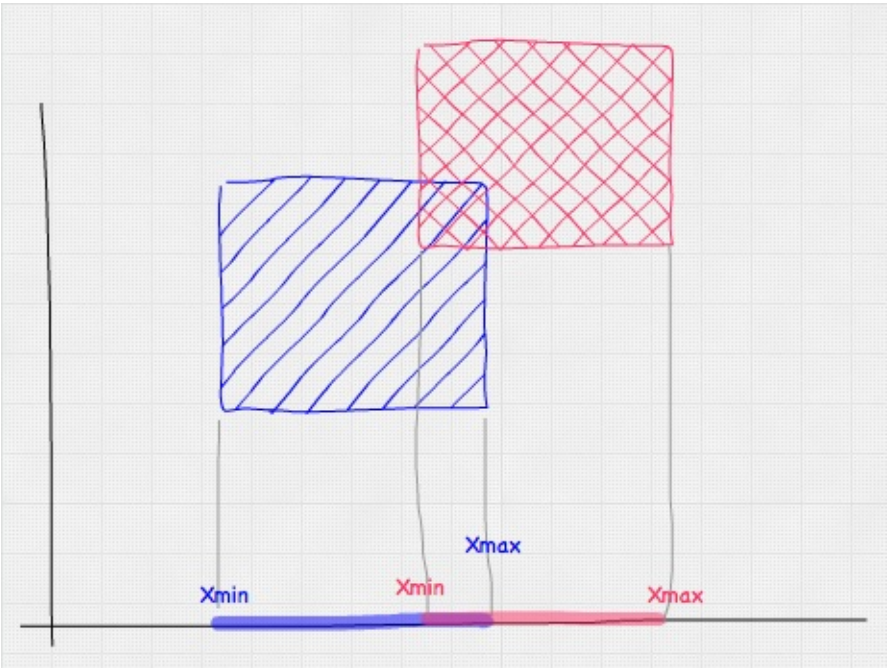
```
        Plane[] planes = new Plane[] {
            null, null, null, null, null // Size = 5
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.PointSize(5f);

            planes[0] = new Plane();
            planes[1] = new Plane(new Vector3(0f, 1f, 0.2f), 3f);
            planes[2] = new Plane(new Vector3(0f, 1f, 0f), -2f);
            planes[3] = new Plane(new Vector3(-1f, 1f, 2f), 3f);
            planes[4] = new Plane(new Vector3(1f, 0f, 0f), 3f);

            bool[] results = new bool[] {
                true, false, true, false, true
            };
            int t = 0;

            for (int i = 0; i < planes.Length; ++i) {
                if (Collisions.Intersects(planes[i], test) != resu
lts[t++]) {
                    LogError("Expected plane " + i + " to " +
                        (results[t - 1] ? "intersect" : "not inter
sect") +
                    " the sphere");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();
```

```
            GL.Disable(EnableCap.CullFace);
            for (int i = 0; i < planes.Length; ++i) {
                GL.Color3(0f, 0f, 1f);
                if (Collisions.Intersects(test, planes[i])) {
                    GL.Color3(1f, 0f, 0f);
                }
                planes[i].Render();
            }
            GL.Enable(EnableCap.CullFace);

            GL.Color3(0f, 0f, 1f);
            test.Render();
        }
    }
}
```

AABB v AABB intersection

One to the most useful intersection tests is testing an AABB against another AABB. In fact, this is probably the intersection test you will use the most!

Surprisingly, this is also one of the simplest collision tests to write, it's a basic overlap test. Becuase the AABB has 3 axis, we just have to check each axis for an overlap. Consider the following image:



It shows an overlap test on only the X axis. If either the X, Y or Z have any overlap we know we have an intersection. Representing the above test in code is as follows:

```
a.minX <= b.maxX && a.maxX >= b.minX
```

The Algorithm

Actually implementing the above test is just a matter of extrapolating the single axis test onto multiple axis. Very similar to the Point in AABB test.

```
bool Intersect(AABB a, AABB b) {
  return (a.minX <= b.maxX && a.maxX >= b.minX) &&
         (a.minY <= b.maxY && a.maxY >= b.minY) &&
         (a.minZ <= b.maxZ && a.maxZ >= b.minZ);
}
```

On Your Own

Add the following function to the `Collisions` class:

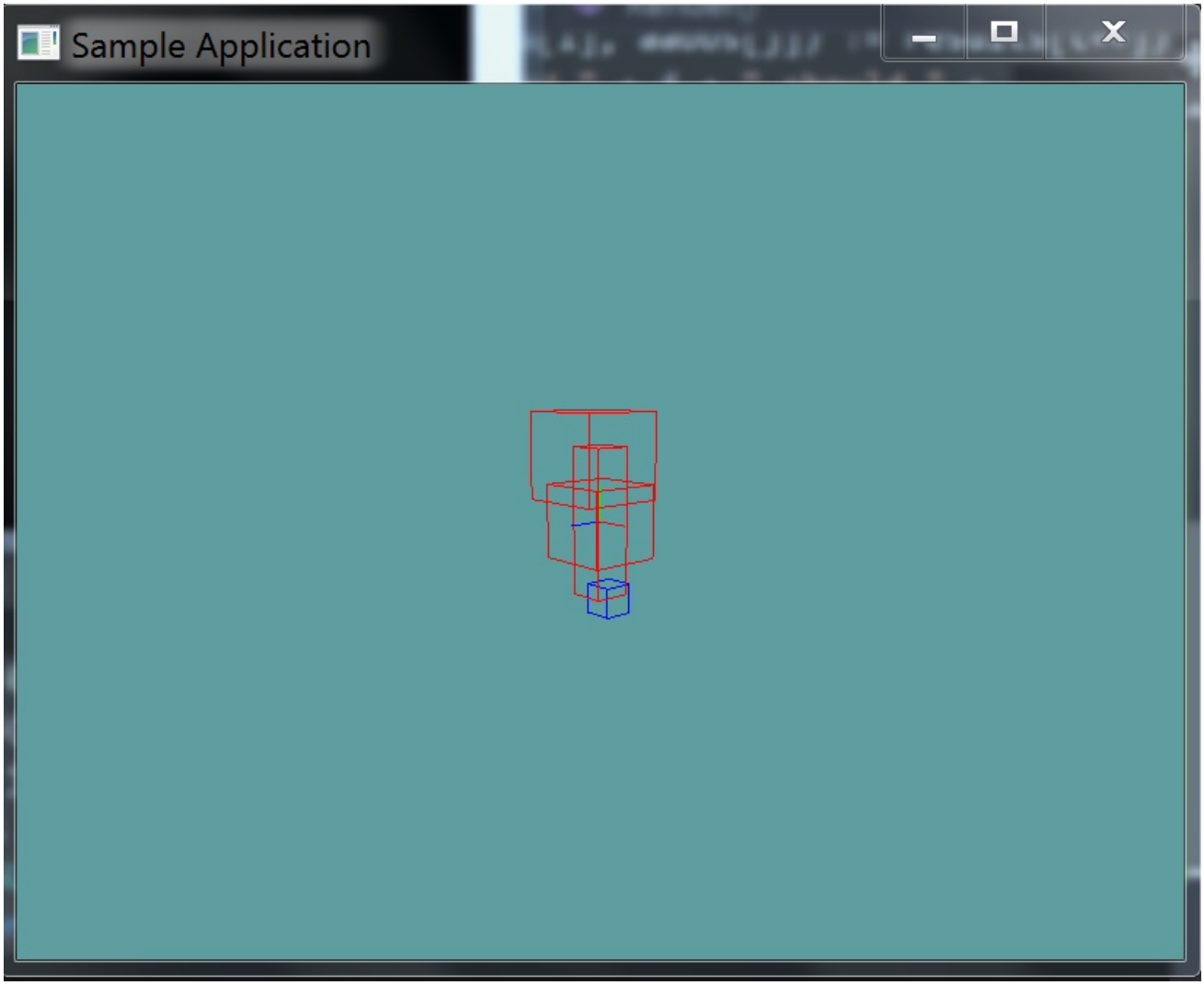
```
public static bool Intersects(AABB a, AABB b) {
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The constructor of the unit test will spit out errors if your results are not as expected. Boxes that intersect are rendered in red, those that don't are rendered in blue.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class AABBtoAABBIntersection : Application {
        AABB[] aabbs = new AABB[] {
            null, null, null, null // Size = 4
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
```

```
Line);

            aabbs[0] = new AABB(new Point(-0.5f, -2f, -0.5f), new
Point(0.5f, 2f, 0.5f));
            aabbs[1] = new AABB();
            aabbs[2] = new AABB(new Point(1f, 1f, 1f), new Point(3
f, 3f, 3f));
            aabbs[3] = new AABB(new Point(-3f, -3f, -3f), new Poin
t(-4f, -4f, -4f));

            bool[] results = new bool[] {
                true, true, false, false, true, true, true, false,
false,
                true, true, false, false, false, false, true,
            };
            int t = 0;

            for (int i = 0; i < aabbs.Length; ++i) {
                for (int j = 0; j < aabbs.Length; ++j) {
                    if (Collisions.Intersects(aabbs[i], aabbs[j])
!= results[t++]) {
                        LogError("aabb " + i + " and " + j + " sho
uld " +
                            (results[t - 1] ? "" : "not ") + "inte
rsect"
                        );
                    }
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            for (int i = 0; i < aabbs.Length; ++i) {
                GL.Color3(0f, 0f, 1f);
```

```
                    for (int j = 0; j < aabbs.Length; ++j) {
                        if (i != j && Collisions.Intersects(aabbs[i],
aabbs[j])) {
                            GL.Color3(1f, 0f, 0f);
                            break;
                        }
                    }
                    aabbs[i].Render();
                }
            }
        }
}
```

AABB Plane intersection

To test if an AABB and plane intersect, we first have to project each vertex of the AABB onto the plane's normal. This leaves us with all the vertices of the AABB on a line.

We then check the vertex that is furthest from the plane. If the vertex diagonally opposite that one is on the other side of the plane, we have an intersection.

The Algorithm

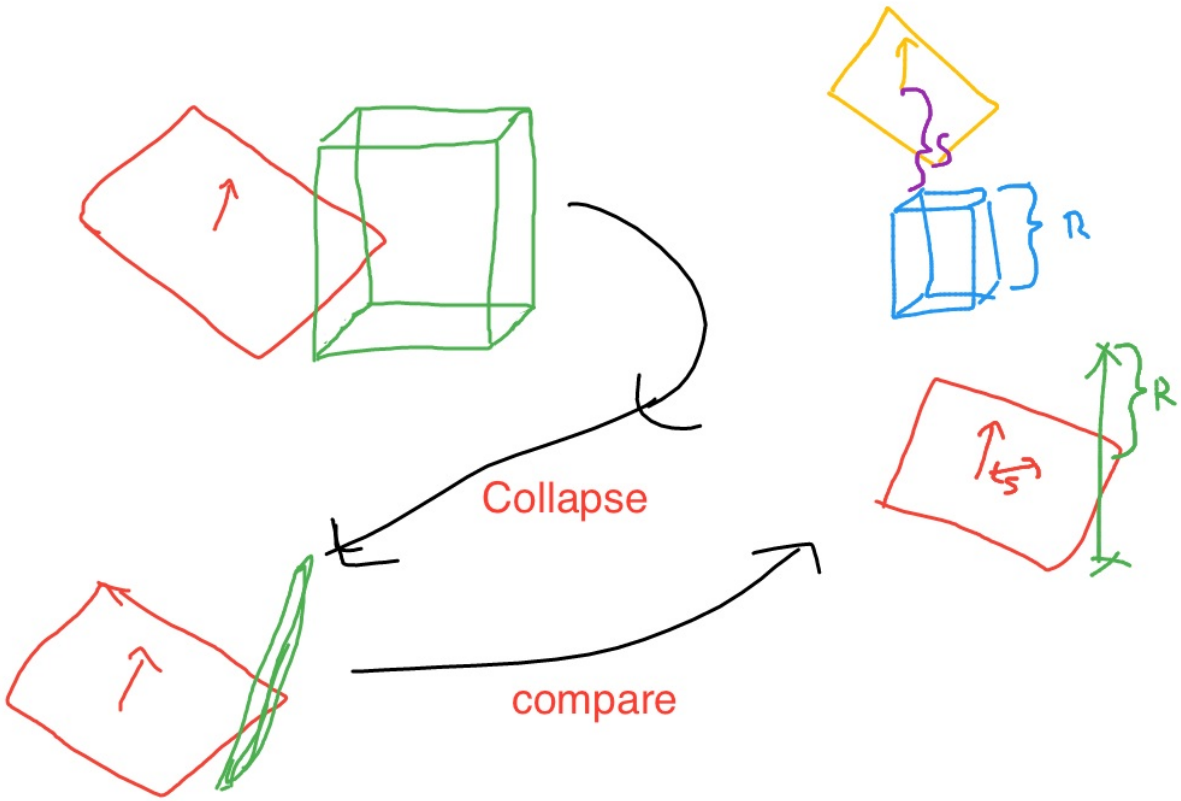
This test is equivalent to finding an AABB vertex most distant along the plane normal and making sure that vertex and the vertex diagonally opposite lie on opposite sides of the plane.

```
// Test if AABB b intersects plane p
int TestAABBPlane(AABB b, Plane p) {
    // Convert AABB to center-extents representation
    Point c = (b.max + b.min) * 0.5f; // Compute AABB center
    Point e = b.max - c; // Compute positive extents

    // Compute the projection interval radius of b onto L(t) = b.c
 + t * p.n
    float r = e[0]*Abs(p.n[0]) + e[1]*Abs(p.n[1]) + e[2]*Abs(p.n[2
]);

    // Compute distance of box center from plane
    float s = Dot(p.n, c) - p.d;

    // Intersection occurs when distance s falls within [-r,+r] in
terval
    return Abs(s) <= r;
}
```

So, what happens here, is we compute the center of the aabb (`c`) and it's extents (`e`). We then collapse extents `e` onto the plane normal (`p.n`).

At this point we have a line with center point `c` and size `r` in both directions.

We then compute `s` , which is the distance of the line from the plane. If the line is further away than half it's length, it does not intersect. If it's closer, it does.



Another method

Of course, this is not the only way to do the intersection test. You could always take the more logical brute force approach.

The brute force approach would be to loop trough all 8 points of the box and check them against the plane. There are 3 possible outcomes:

1) All points are in front of the plane, no collision 2) All points are behind the plane, no collision 3) Some points are in front of the plane, some are behind. COLLISION

That is to say, if you encounter any points that are not on the same side of the plane, a collision has occured. The hardest thing about the brute force approach is figuring out how to get all 4 sides of the AABB as vertices.

```
Vector3[] points = new Vector3[] {
    // Min side
    new Vector3(min.X, min.Y, min.Z),
    new Vector3(max.X, min.Y, min.Z),
    new Vector3(min.X, max.Y, min.Z),
    new Vector3(min.X, min.Y, max.Z),
    // Max side
    new Vector3(max.X, max.Y, max.Z),
    new Vector3(min.X, max.Y, max.Z),
    new Vector3(max.X, min.Y, max.Z),
    new Vector3(max.X, max.Y, min.Z)
};
```

On Your Own

Add the following function to the `Collisions` class, how you implement this function (brute force or fancy) is up to you:

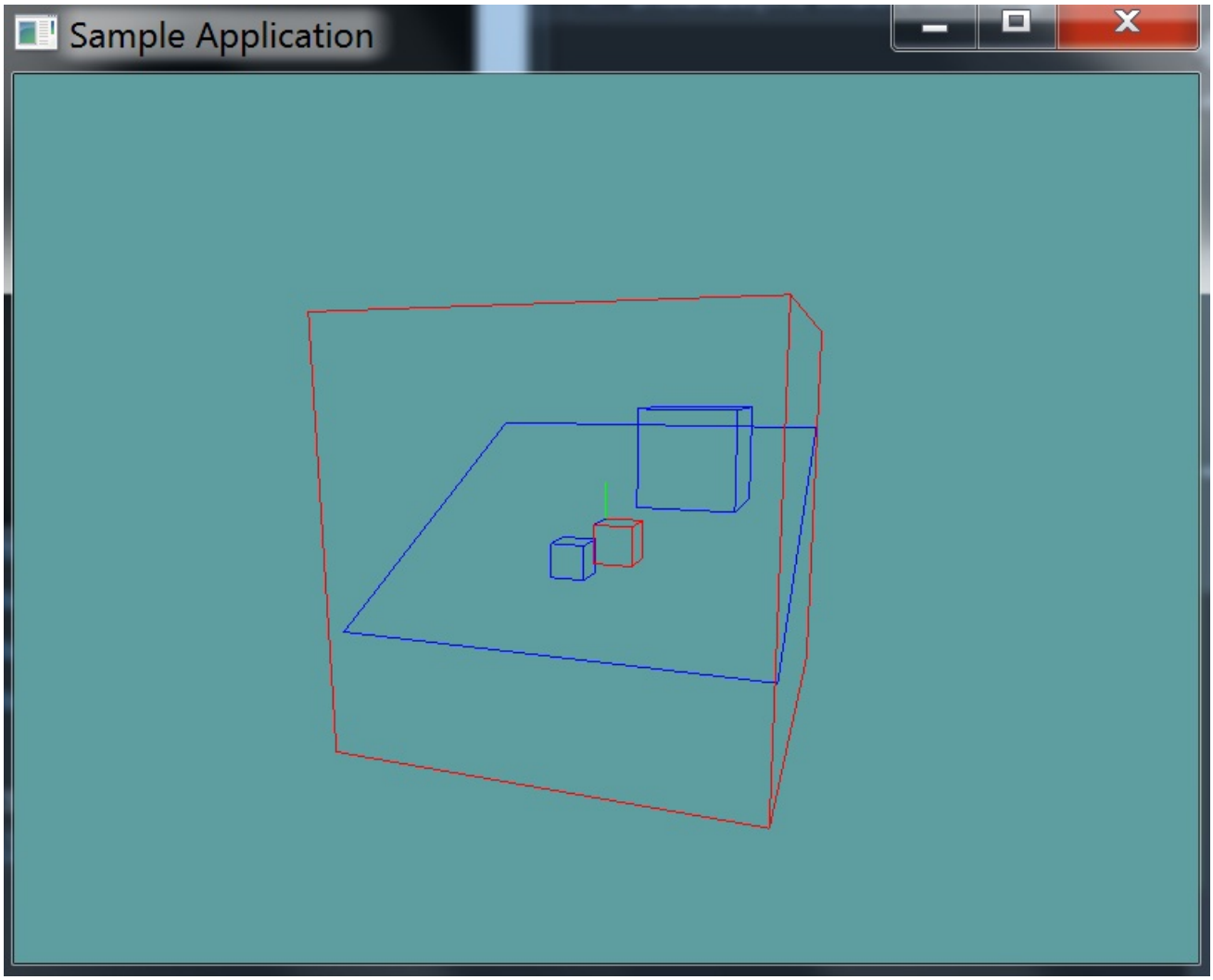
```
// TODO: Provide implementation for this
public static bool Intersects(AABB aabb, Plane plane)

// Conveniance function
public static bool Intersects(Plane plane, AABB aabb) {
    return Intersects(aabb, plane);
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The constructor of the unit test will spit out errors. There are 4 cubes, 1 in front of the plane, 1 behind the plane, 1 intersecting the plane and one that encompases (Again, just intersects becuse the plane is of infinate size) the plane. The two boxes that intersect the plane are renderd in red.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class AABBPlaneIntersection : Application {
```

```
        Plane test = new Plane();

        AABB[] aabbs = new AABB[] {
            null, null, null, null // Size = 4
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.PointSize(5f);

            test.Normal = new Vector3(0f, 1f, 0.5f);
            aabbs[0] = new AABB(new Point(-2f, -2f, -2f), new Poin
t(-1f, -1f, -1f));
            aabbs[1] = new AABB(new Point(2f, 1f, 2f), new Point(4
f, 3f, 4f));
            aabbs[2] = new AABB(new Point(1f, 0f, 1f), new Point(0
f, -1f, 0f));
            aabbs[3] = new AABB(new Point(5f, 5f, 5f), new Point(-5
f, -5f, -5f));

            bool[] results = new bool[] {
                false, false, true, true
            };
            int t = 0;

            for (int i = 0; i < aabbs.Length; ++i) {
                if (Collisions.Intersects(aabbs[i], test) != resul
ts[t++]) {
                    LogError("Expected aabb " + i + " to " +
                        (results[t - 1] ? "intersect" : "not inter
sect") +
                    " the plane");
                }
            }
```

```
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            for (int i = 0; i < aabbs.Length; ++i) {
                GL.Color3(0f, 0f, 1f);
                if (Collisions.Intersects(test, aabbs[i])) {
                    GL.Color3(1f, 0f, 0f);
                }
                aabbs[i].Render();
            }

            GL.Color3(0f, 0f, 1f);
            test.Render(5f);
        }
    }
}
```

Plane Plane intersection

We've already done some work with plane intersection, when we did the intersection for 3 planes onto a point. Luckly the intersection of 2 planes is a lot simpler.

The only time two planes don't intersect is if they are parallel. If the planes are not parallel they will intersect in a line (of infinate length)

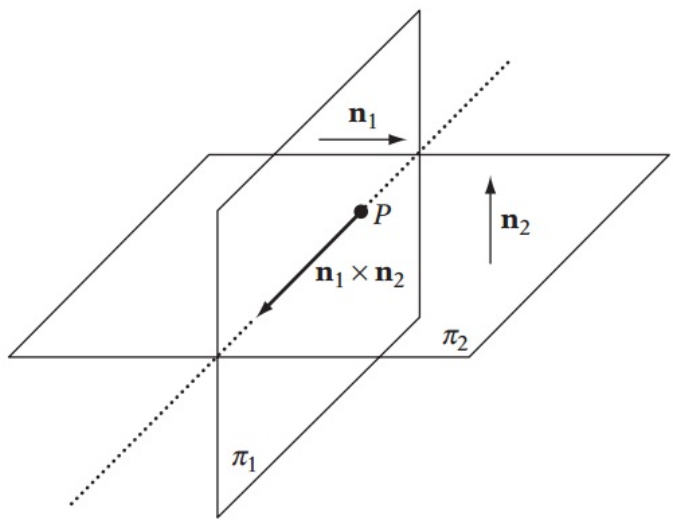


Figure 5.30 The intersection of two planes.

The line of intersection is fairly useless in game development, so we will not be covering it here. If you are interested, youtube is full of videos on how to find this line, and the book Real Time Collision Detection covers how to do it in the context of games. I have a PDF if you need it.

The Algorithm

The key piece of information above is that two planes never intersect if they are parallel. Only if they are parallel. Knowing this case, the problem becomes, how can we tell if two planes are parallel?

Dot Product to the rescue! We have the normal of each plane! If the normals point in the same direction, we have parallel planes!

```
bool PlanesIntersect(Plane p1, Plane p2) {
    // Compute direction of intersection line
    Vector d = Cross(p1.n, p2.n);

    // If the length(Squared) of d is zero, the planes are
    // parallel (and separated) or coincident,
    // so they're not considered intersecting
    return (Dot(d, d) > EPSILON); // EPSILON = 0.0001f
}
```

On Your Own

Add the following function to the `Collisions` class:

```
public static bool Intersects(Plane p1, Plane p2)
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

Visually, the unit test does not matter, this is because all of the planes intersect at least one other plane. Instead, the Initialize funciton will throw detailed error messages if needed.

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class PlanePlaneIntersection : Application {
        Plane[] planes = new Plane[] {
```

```
            null, null, null, null, null // Size = 5
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            //GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.PointSize(5f);

            planes[0] = new Plane();
            planes[1] = new Plane(new Vector3(2f, 0f, 1f), 0f);
            planes[2] = new Plane(new Vector3(0f, 1f, 0f), -2f);
            planes[3] = new Plane(new Vector3(-1f, 1f, 2f), 3f);
            planes[4] = new Plane(new Vector3(1f, 0f, 0.5f), 3f);

            bool[] results = new bool[] {
                false, true,  true, true,  true, true, false, true
,  true, false, true,
                true,  false, true, true,  true, true, true,  false
, true, true,  false,
                true, true,  false,
            };
            int t = 0;

            for (int i = 0; i < planes.Length; ++i) {
                for (int j = 0; j < planes.Length; ++j) {
                    if (Collisions.Intersects(planes[i], planes[j]
) != results[t++]) {
                        LogError("[" + (t - 1) + "] Expected plane
 " + i + " to " +
                            (results[t - 1] ? "intersect" : "not i
ntersect") +
                        " plane " + j);
                    }
                }
            }
```

```
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            for (int i = 0; i < planes.Length; ++i) {
                planes[i].Render();
            }
        }
    }
}
```

Rays and lines

Rays and lines are used to determine collisions with primitives over distances. Both methods are very useful when dealing with moving objects.

We will discuss the merrits of both methods in their own chapters.

Ray Casting

Ray casting is mainly used to query the world. Whenever you click on the screen and a 3D object is selected it happens because a ray was created out of your mouse position, and that was tested against the objects on screen.

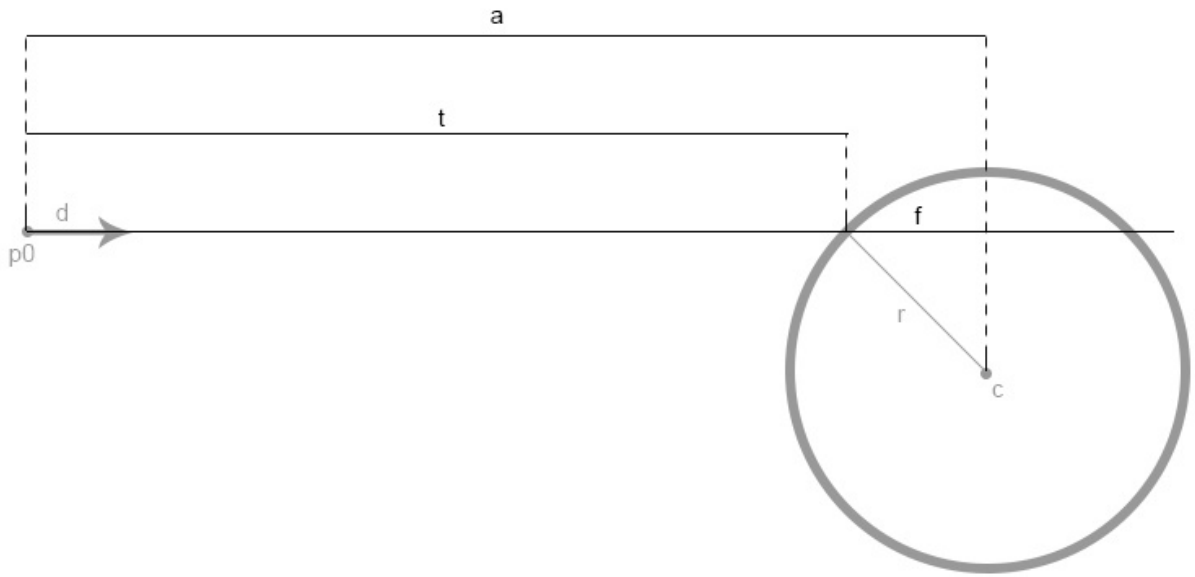
Rays are also used to determine if a collision between moving objects has happened.

Ray Against Sphere

Given the following image, we have :

- Ray **R**, defined by:
  - point **p0**
  - normal **d**
- Sphere **S**, defined by:
  - center point **c**
  - radius **r**

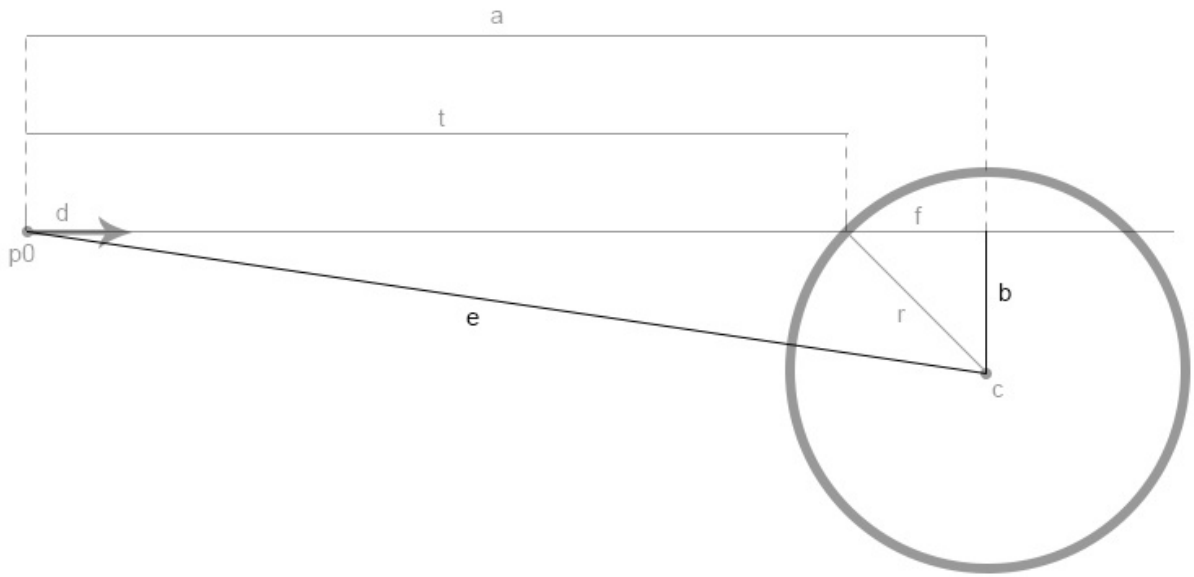
Given this information, we want to find out if ray **R** intersects sphere **S**. If the ray intersects the sphere, it's going to be along the normal. In the image below, **t** is the length at which the normal intersects the sphere:



An intersection happens at time **t**, along ray **R**. Another way to express this would be that the intersection happens at **R(t)**. To find **t**, we need to first find **a** and **f**. The formula for **t** is:

```
t = a - f
```

In order to find **a** and **f**, we need two more vectors, vector **e** and vector **b**. Vector **e** is the vector from **p0** to **c**. Vector **b** is one side of the right triangle formed by vectors **r**, **f** and **b**. The image below demonstrates this. You know r and f, you have to solve the right triangle for b.



In this image, we have two triangles we care about.

- Triangle **f**, **b**, **r**
- Triangle **a**, **e**, **b**

Given these two triangles, we can figure out the following

- Vector **e** is a vector going to **p0** to **c**
- Vector **a** is vector **e** projected onto vector **d**
- Vector **b** is given by the pathegroen therom
  - : Therom
  - : Triangle **a**, **e**, **b**
  - : Re-arranged
  - :final
- Vector **f** is given by the pathegroen therom
  - : Triangle **f**, **b**, **r**

  - : Re-arranged
- Now we can solve for **t**
  - : algorithm for **t**
  - : Expanded **f**
  - : Expanded **b**, final

With that, we have the value of t!

What happens when the ray doesn't hit the sphere? The expression inside the square root $sqrt(r^2 - e^2 + a^2)$ will be negative. Therefore, we have to make an early out test for that!

There is one more edge case, if e^2 is less than the squared radius of the sphere, the ray starts INSIDE the sphere! This is again a special case that we need to handle delicatley.

The Algorithm

In the above diagrams, it's not always clear what is a vector value and what is a scalar value! When a vector is used in a square root, we are talking about the length of the vector. The code below should give more context.

```
// This function will return the value of t
// if it returns negative, no collision!
float Raycast01(Ray ray, Sphere sphere) {
    Vector3 p0 = ray.Position;
    Vector3 d = ray.Normal;
    Vector3 c = sphere.Position;
    float r = sphere.Radius;

    Vector3 e = c - p0;
    // Using Length here would cause floating point error to creep
 in
    float Esq = Vector3.LengthSquared(e);
    float a = Vector3.Dot(e, d);
    float b = Sqrt(Esq - (a * a));
    float f = Sqrt((r * r) - (b * b));

    // No collision
    if (r * r - Esq + a * a < 0f) {
       return -1; // -1 is invalid.
    }
    // Ray is inside
    else if (Esq < r * r) {
        return a + f; // Just reverse direction
    }
    // else Normal intersection
    return = a - f;
}
```

On Your Own

Add the following function to the `Collisions` class:

```
// TODO: Implement this function
static bool Raycast(Ray ray, Sphere sphere, out float t)

// I've implemented these for you!

// Conveniance method, returns t without an out param
// If no collision happened, will return -1
static float Raycast(Ray ray, Sphere sphere) {
    float t = -1;
    if (!Raycast(ray, sphere, out t)) {
        return -1;
    }
    return t;
}

// Conveniance method, returns the point of intersection
// instead of p
static bool Raycast(Ray ray, Sphere sphere, out Point p) {
    float t = -1;
    bool result = Raycast(ray, sphere, out t);
    p = new Point(ray.Position.ToVector() + ray.Normal * t);
    return result;
}
```

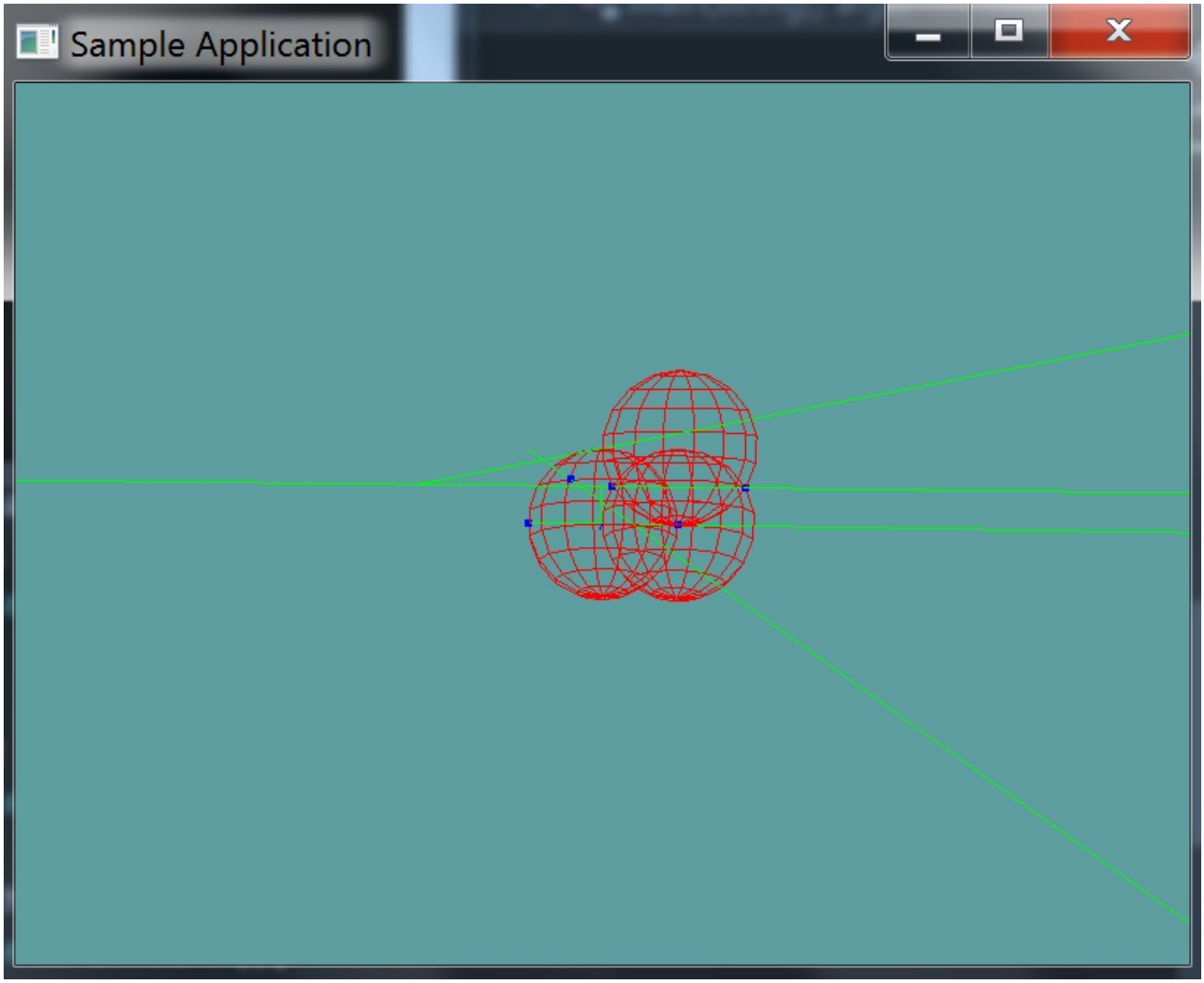
And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The unit test looks visually messy. 3 red spheres are rendered, and a bunch of green lines. Any collision point between a ray and a sphere is marked as blue.

Not every ray is tested against every line, that's why visually this one is impossible to call. The constructor however, will throw errors where needed.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class RaycastSphere : Application {
        public class Touple {
            public Ray ray;
            public Sphere sphere;
            public bool result;

            public Touple(float rayX, float rayY, float rayZ, float
 normX, float normY, float normZ,
                float sphereX, float sphereY, float sphereZ, float
```

```
 rad, bool res) {
                ray = new Ray(new Point(rayX, rayY, rayZ), new Vec
tor3(normX, normY, normZ));
                sphere = new Sphere(new Point(sphereX, sphereY, sp
hereZ), rad);
                result = res;
            }
        }

        Touple[] touples = new Touple[] {
            new Touple(-2, 1, 0, 2, 0, 0, 2, 0, 0, 2, true),
            new Touple(-2, 0, 0, 2, 0, 0, 2, 2, 0, 2, true),
            new Touple(-2, 0, 0, 2, 0, 0, 0, 0, 0, 2, true),
            new Touple(-2, 2, 0, 2, -1, 2, 0, 0, 0, 2, true),
            new Touple(2, 1, 0, 2, 0, 0, 2, 0, 0, 2, true),
            new Touple(-2, 1, 0, -1, 0, 0, 2, 0, 0, 2, false),
            new Touple(-5, 1, 0, 2, 0.4f, 0, 2, 0, 0, 2, false)
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            foreach(Touple touple in touples) {
                float t = 0f;
                if (Collisions.Raycast(touple.ray, touple.sphere,
out t) != touple.result) {
                    LogError("Expected ray: " + touple.ray + "\nTo
 " +
                        (touple.result ? "intersect" : "not inters
ect")
                    + " sphere: " + touple.sphere);
                }
            }
```

```
        }

        public override void Render() {
            //GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            base.Render();
            DrawOrigin();

            GL.Color3(1f, 0f, 0f);
            for (int i = 0; i < 3; ++i) {
                touples[i].sphere.Render();
            }

            GL.Color3(0f, 1f, 0f);
            foreach (Touple touple in touples) {
                touple.ray.Render();
                if (touple.result) {
                    Point p = new Point();
                    Collisions.Raycast(touple.ray, touple.sphere,
out p);
                    GL.Color3(0f, 0f, 1f);
                    p.Render();
                    GL.Color3(0f, 1f, 0f);
                }
            }
        }

        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Raycast AABB

Raycasting an AABB is often done trough an algorithm called **Cyrus-Beck clipping**. You can read more about it here and here. Because a ray that enters a box will intersect it twice, this algorithm gives both T1 and T2. We only care about the closer t value.

This algorithm works by clipping a ray against a plane. We know that an AABB has 6 sides, it's made out of 6 planes. It has 3 pairs of parallel planes (left and right, top and bottom, front and back).

When we break the box up into 3 pairs of parallel planes we actually break it down into orthagonal planes, defined by a single axis! That is to say:

```
Left plane = aabb.min.x
Right plane = aabb.max.x
```

Now then, a point along the ray is represented as:

```
point = ray.origin + t * ray.direction
```

Where t is some time along the ray. We can break this down into it's indevidual components:

```
point.x = ray.origin.x + t * ray.direction.x
point.y = ray.origin.y + t * ray.direction.y
point.z = ray.origin.z + t * ray.direction.z
```

Next, we can simply re-arrange this formula, to solve for t, rather than an unknown 3D point.

```
t = (point.x – ray.origin.x) / ray.direction.x
t = (point.y – ray.origin.y) / ray.direction.y
t = (point.z – ray.origin.z) / ray.direction.z
```

Finally, we substitute point, the unknown for a component of our perpendicular plane!

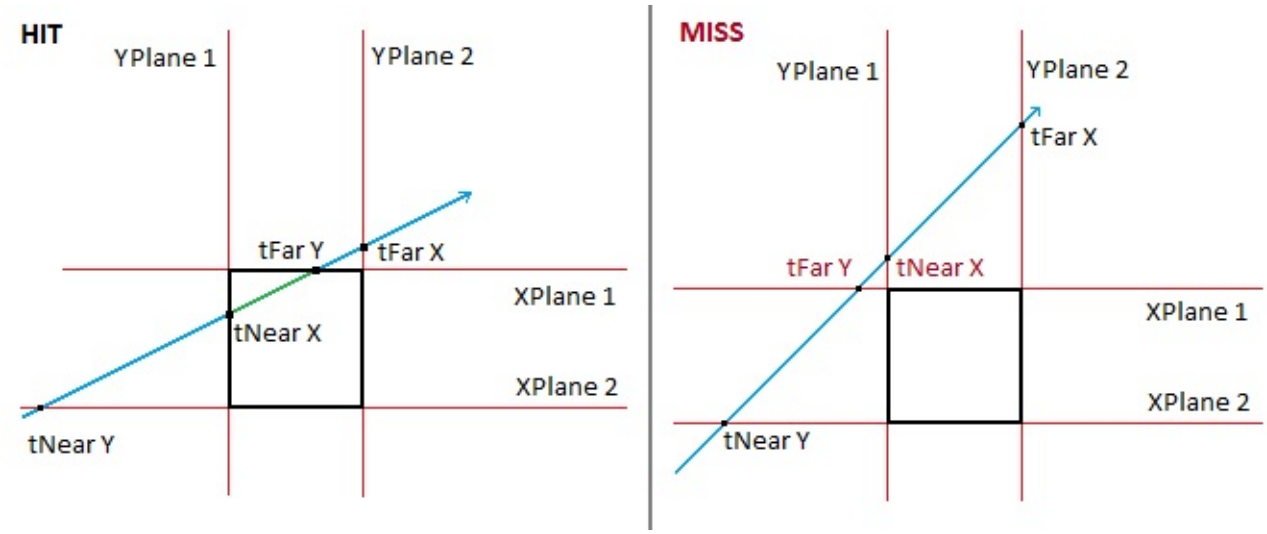
```
tMinX = (aabb.min.x – ray.origin.x) / ray.direction.x
tMaxX = (aabb.max.y – ray.origin.x) / ray.direction.x
// Same for y and z
```

At this point we have 6 min and max values, tMinX, tMinY, tMinZ, tMaxX, tMaxY, tMaxZ. How do we find just tMin and tMax? Find the MAXIMUM tMin and the MINIMUM tFar.

```
// The BIGGEST MIN value
tMin = Max( // Whats bigger [minX/minY] or MinZ
          Max( // What's bigger, minX or minY
            Min(tMinX, tMaxX), // Get the smaller X
            Min(tMinY, tMaxY) // Get teh smaller Y
          ),
          Min(tMinZ, tMaxZ) // Get the smaller Z
        );

// The SMALLEST MAX value
tMax = Min(
          Min( // Whats smaller [minX/minY] or MinZ
            Max(tMinX, tMaxX),  // Get the bigger X
            Max(tMinY, tMaxY)  // Get teh bigger Y
          ),
          Max(tMinZ, tMaxZ) // Get the bigger Z
        );
```

Thats it! If tMax is less than 0, then the ray is intersecting the AABB, but before the origin of the ray. So, it's not a collision! (t would be negative)

If tMin is less than tMax, then there is no collision. These images demonstrate why:



You want to return tMin as t, becuase it's smalelr, it's closer on our ray. Except if tMin is less than 0, in that case, you want to return tMax.

The Algorithm

```
// Given a ray and an aabb:
// return t at which the ray intersects the aabb.
// return -1 if there is no intersection
public static float Raycast(Ray ray, AABB aabb) {
    float t1 = (aabb.minX - ray.Position.X) / ray.Normal.X;
    float t2 = (aabb.maxX - ray.Position.X) / ray.Normal.X;
    float t3 = (aabb.minY - ray.Position.Y) / ray.Normal.Y;
    float t4 = (aabb.maxY - ray.Position.Y) / ray.Normal.Y;
    float t5 = (aabb.minZ - ray.Position.Z) / ray.Normal.Z;
    float t6 = (aabb.maxZ - ray.Position.Z) / ray.Normal.Z;

    float tmin = Max(Max(Min(t1, t2), Min(t3, t4)), Min(t5, t6));
    float tmax = Min(Min(Max(t1, t2), Max(t3, t4)), Max(t5, t6));

    // if tmax < 0, ray (line) is intersecting AABB, but whole AAB
B is behing us
    if (tmax < 0) {
        return -1;
    }

    // if tmin > tmax, ray doesn't intersect AABB
    if (tmin > tmax) {
        return -1
    }

    if (tmin < 0f) {
        return tmax;
    }
    return tmin;
}
```

On Your Own

Add the following function to the `Collisions` class:

```
// TODO: Implement ONLY THIS ONE method:
public static bool Raycast(Ray ray, AABB aabb, out float t)


// I've implemented the blow methods for you.
// Nothing to do past this point

public static float Raycast(Ray ray, AABB aabb) {
    float t = -1;
    if (!Raycast(ray, aabb, out t)) {
        return -1;
    }
    return t;
}

public static bool Raycast(Ray ray, AABB aabb, out Point p) {
    float t = -1;
    bool result = Raycast(ray, aabb, out t);
    p = new Point(ray.Position.ToVector() + ray.Normal * t);
    return result;
}
```

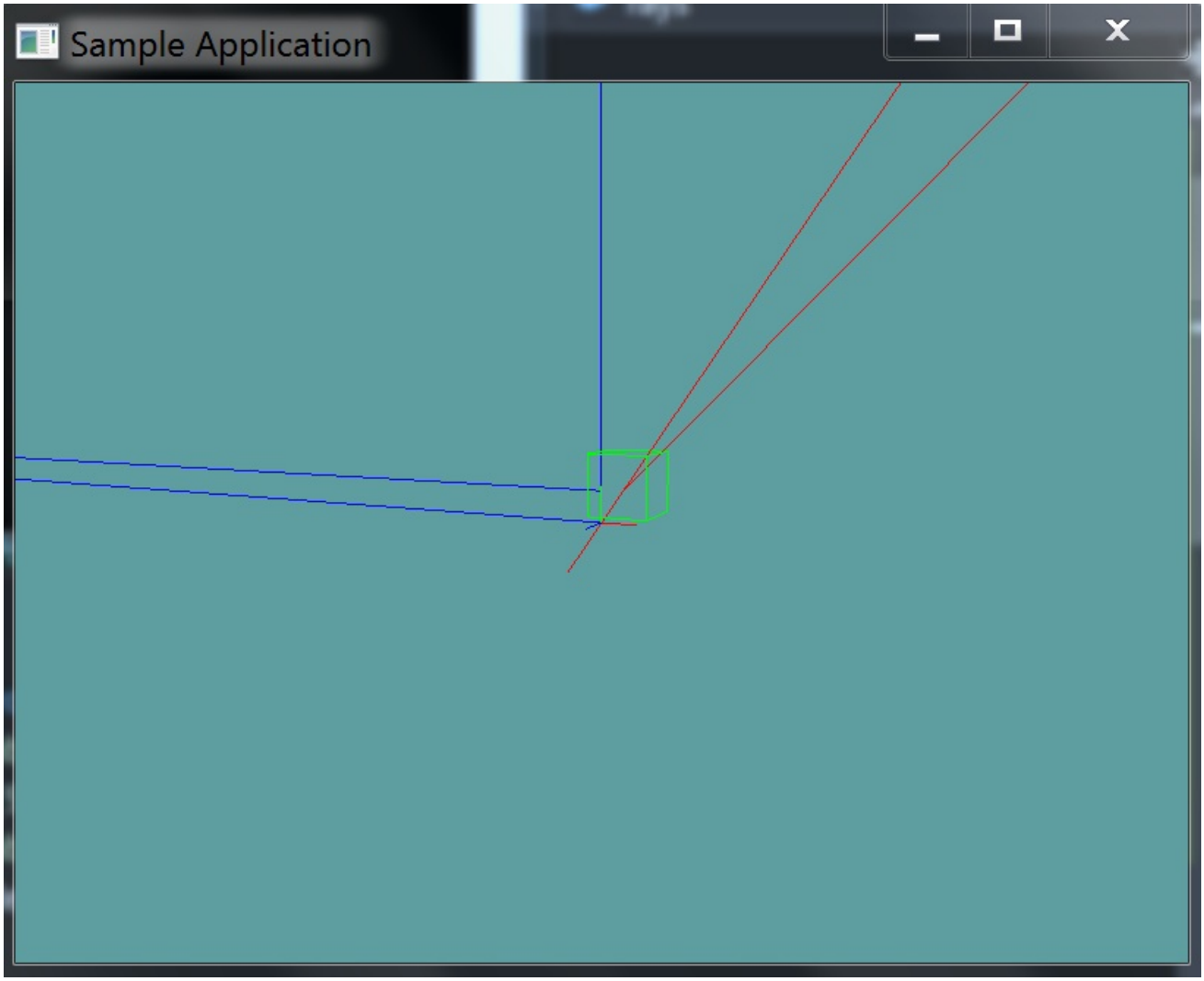
And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will render an AABB on screen, and several rays. If any rays intersect the AABB they will render red, otherwise blue.

The constructor of the unit test will spit out errors if the test results are not what are expected.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class RaycastAABB : Application {
        public AABB test = new AABB(new Point(0.5f, 0.5f, 0.5f), n
ew Point(2f, 2f, 2f));

        public Ray[] rays = new Ray[] {
            new Ray(new Point(-2, -2, -2), new Vector3(2, 2, 2)),
            new Ray(new Point(0f, 0f, 0f), new Vector3(0f, 1f, 0f)
),
            new Ray(new Point(0f, 0f, 0f), new Vector3(-1f, 0f, 0f
```

```
)),
            new Ray(new Point(1f, 1f, 1f), new Vector3(1f, 1f, 0f)
),
            new Ray(new Point(0.4f, 1f, 1f), new Vector3(-1f, 0f, 0
f)),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            bool[] results = new bool[] {
                true, false, false, true, false
            };

            float t;
            for (int i = 0; i < results.Length; ++i) {
                if (Collisions.Raycast(rays[i], test, out t) != re
sults[i]) {
                    LogError("Expected ray at index: " + i + " to "
 +
                        (results[i] ? "intersect" : "not intersect"
) +
                        " the aabb");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(0f, 1f, 0f);
            test.Render();
```

```
            float t;
            foreach(Ray ray in rays) {
                if (Collisions.Raycast(ray, test, out t)) {
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 0f, 1f);
                }
                ray.Render();
            }
        }

        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Ray Against Plane

Testing a ray against a plane is surprisingly easy. Remember, any point on the ray can be represented by a time t, such that:

```
point(t) = plane.position + plane.normal * t
^ the (t) above means point at time t
```

Also, recall we can test if a point is on a plane if it fits this equation:

```
DOT(point, plane.normal) = plane.distance
```

A ray and a plane collide if a point exists that satisfies both of these equations! Let's start out with the formula for point on plane:

```
DOT(point, plane.normal) = plane.distance
```

In this formula the point is unknown. But we know that the point must also be on the ray at some time t. So, let's substitute the unknown point for the point on ray formula.

```
DOT(ray.position + ray.normal * t, plane.normal) = plane.distance
```

At this point, the only unknown in the above equation is t. Let's start re-arranging the equation so t is on the left side by its-self. We can distribute the dot product operation:

```
DOT(ray.position, plane.normal) + DOT(ray.normal * t, plane.normal
) = plane.distance
```

Now, we can subtract the first dot product from each side.

```
DOT(ray.normal * t, plane.normal) = plane.distance - DOT(ray.posit
ion, plane.normal);
```

Finally, we can divide each side by ray.normal DOT plane.normal.

```
t = (plane.distance - DOT(ray.position, plane.normal) / DOT(ray.no
rmal, plane.normal)
```

That's it! We now have T! The above might be a bit confusing, let's take a look at it as just math. In the below formula **pos** is the position of the ray, **dir** is the normal of the ray. **norm** is the normal of the plane and **dist** is the distance of the plane. **t** is of course t. t**dir** means t * dir.

- First we have the formula for a plane
- $unknownPoint \cdot norm = dist$
-
- Next, we substitute the point on ray formula for the unknown point
- $(pos + t\mathbf{dir}) \cdot norm = dist$
-
- Distribute the dot product
- $pos \cdot norm + t\mathbf{dir} \cdot norm = dist$
-
- Subtract pos dot norm from both sides
- $t\mathbf{dir} \cdot norm = dist - pos \cdot nrom$
-
- Divide both sides by dir dot norm
- $t = \frac{dist - pos \cdot nrom}{dir \cdot norm}$

A few things to keep in mind.

If the ray is parallel to the plane $dir \cdot norm$ will be 0 and there is no intersection.

An intersection is only valid if the ray is in front of the plane.That is, if the direction of the ray is opposite of the planes normal. This is true if $dir \cdot norm < 0$.

If the value of t is out of range (that is, if t is negative), then there is no intersection.

The Algorithm

```
// Returns t if collision happened, -1 if it didnt
float Raycast(Ray ray, Plane plane) {
    float nd = Vector3.Dot(ray.Normal, plane.Normal);
    float pn = Vector3.Dot(ray.Position, plane.Normal);

    if (nd >= 0f) {
        return -1;
    }

    t = (plane.Distance - pn) / nd;

    if (t >= 0f) {
      return t;
    }
    return -1;
}
```

On Your Own

Add the following function to the `Collisions` class:

```
// TODO: Implement ONLY THIS ONE method:
public static bool Raycast(Ray ray, Plane plane, out float t)


// I've implemented the blow methods for you.
// Nothing to do past this point

// Conveniance method, returns t without an out param
// If no collision happened, will return -1
public static float Raycast(Ray ray, Plane plane) {
    float t = -1;
    if (!Raycast(ray, plane, out t)) {
        return -1;
    }
    return t;
}

// Conveniance method, returns the point of intersection
public static bool Raycast(Ray ray, Plane plane, out Point p) {
    float t = -1;
    bool result = Raycast(ray, plane, out t);
    p = new Point(ray.Position.ToVector() + ray.Normal * t);
    return result;
}
```

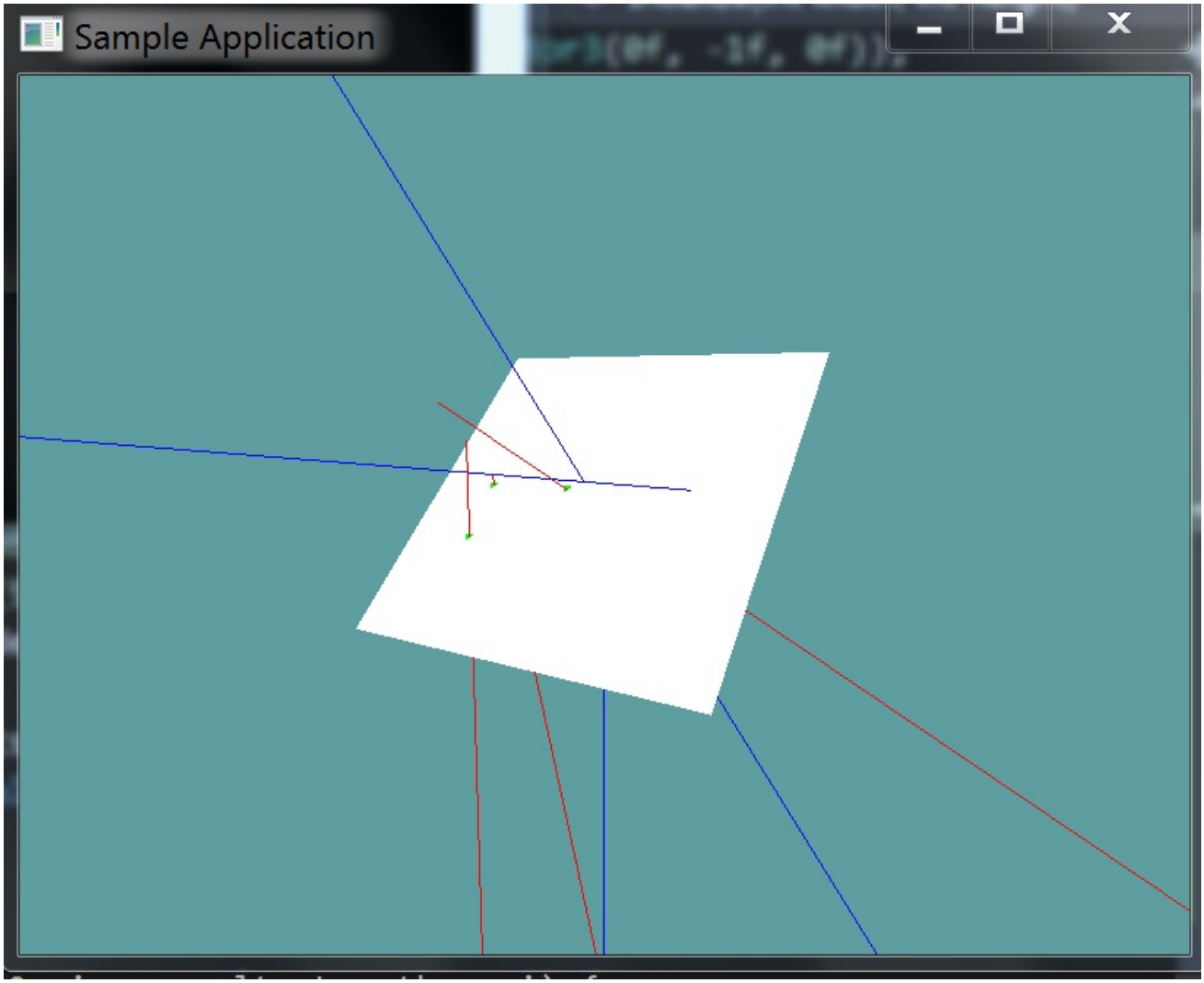
And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will render a plane on screen, and several rays. If any rays intersect the Plane they will render red, otherwise blue. The point of intersection is renderd as a green dot

The constructor of the unit test will spit out errors if the test results are not what are expected.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class RaycastPlane : Application {
        public Plane test = new Plane(new Vector3(1f, 1f, 0f), 1f)
;

        public Ray[] rays = new Ray[] {
            new Ray(new Point(0f, 0f, 0f), new Vector3(0f, -1f, 0f
```

```
)),
            new Ray(new Point(0.5f, 0.5f, 0f), new Vector3(-1f, -1
f, 0f)),
            new Ray(new Point(1f, 1f, 0f), new Vector3(1f, 1f, 0f)
),
            new Ray(new Point(1f, 1f, -3f), new Vector3(0f, 0f, 1f
)),
            new Ray(new Point(2f, 2f, 3f), new Vector3(0f, -1f, 0f
)),
            new Ray(new Point(3f, 3f, 3f), new Vector3(-3f, -3f, -3
f)),
            new Ray(new Point(1f, 1f, 3f), new Vector3(-2f, -3f, 1
f)),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Fill);

            bool[] results = new bool[] {
                false, false, false, false, true, true, true
            };

            float t;
            for (int i = 0; i < results.Length; ++i) {
                if (Collisions.Raycast(rays[i], test, out t) != re
sults[i]) {
                    LogError("Expected ray at index: " + i + " to "
 +
                        (results[i] ? "intersect" : "not intersect"
) +
                        " the plane");
                }
            }
        }
```

```
        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(1f, 1f, 1f);
            test.Render(5f);


            float t;
            foreach (Ray ray in rays) {
                if (Collisions.Raycast(ray, test, out t)) {
                    Point colPoint = new Point();
                    Collisions.Raycast(ray, test, out colPoint);
                    GL.Color3(0f, 1f, 0f);
                    colPoint.Render();
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 0f, 1f);
                }
                ray.Render();
            }
        }

        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Line Testing

When we talk about lines, especially in the context of line testing; we are actually talking about line segments. A line segment is a line that has a start point and an end point.

These tests are essentially capped ray-casts. Assume a line goes from p0 to p1. It has a length of L. If you do a ray cast from p0 to the direction of P1, and the returned t value is less than L, the collision is on the line segment. If t is less than 0 or greater than L, the collision is on the line, but outside the segment we are testing.

The most common approach line testing is to simply create wrapper functions around our existing ray-cast functions. However, there are often early out cases that exist in a line test, which don't exist in a ray cast.

For this reason, sometimes line tests are implemented as code of their own. But, given how much of the logic the two tests share, there is no reason to re-invent the wheel. For this reason, we will be going with the simpler approach of creating wrapper functions.

Line Against Sphere

In this chapter i will provide code for the wrapper functions of line against sphere. I'll provide all implementation as an example, the next few chapters i'll only provide the API and leave the implementation up to you.

This test will **test intersection**, it will **not test containment**. That is, if you have a line, and both start and end points are INSIDE the sphere, the result of this test will be false!

Implementation

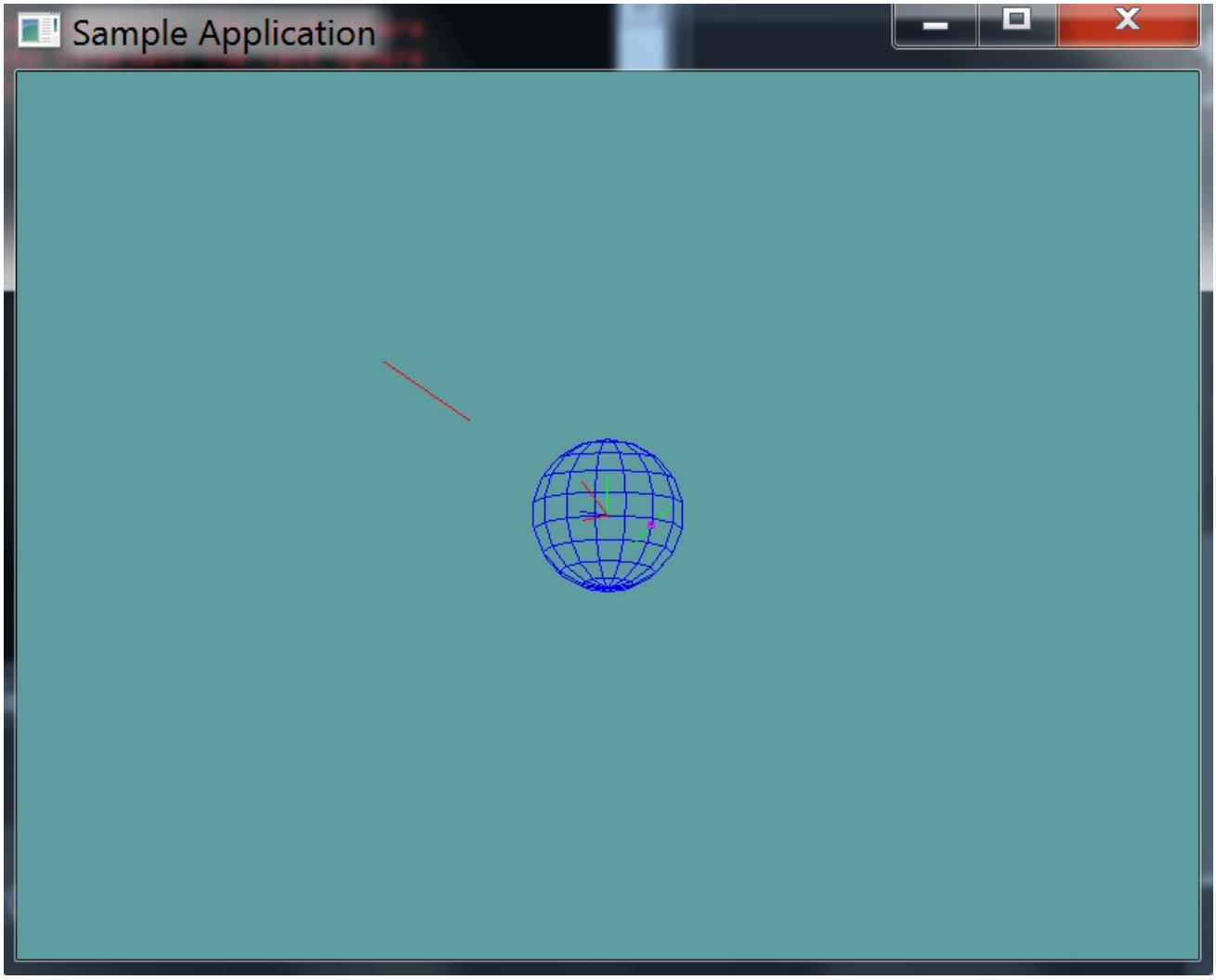
```
public static bool LineTest(Line line, Sphere sphere, out Point re
sult) {
    // First, create a ray from p0 to p1
    Ray ray = new Ray();
    ray.Position = line.start; // by reference, that's ok
    // Normal points from start to end, by value.
    // The Normal setter will automatically normalize this
    ray.Normal = (line.end.ToVector() - line.start.ToVector());

    // Not do the actual ray cast
    float t = -1;
    if (!Raycast(ray, sphere, out t)) {
        // If the raycast returns false, the point was never on th
e line!
        result = new Point(0f, 0f, 0f);
        return false;
    }

    // If t is less than 0, the point is behind the start point
    if (t < 0) {
        // By value! Call new to make a new point, don't want to a
ssigne a reference
```

```
        result = new Point(line.start.ToVector());
        return false;
    }
    // Optimize by checking length squared instead of length!
    // if t is greater than the length of the line, intersection i
s after start point
    else if (t * t > line.LengthSquared) {
        result = new Point(line.end.ToVector()); // Again, by valu
e!
        return false;
    }

    // If we made it here, the line intersected the sphere
    result = new Point(ray.Position.ToVector() + ray.Normal * t);
    return true;
}
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will print out errors if any are present in your code.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class LinetestSphere : Application {
        public Line[] lines = new Line[] {
            new Line(new Point(2.5f, 2.5f, 2.5f), new Point(4, 4, 4
)), // false
            new Line(new Point(1f, 1f, 0f), new Point(0f, 0f, 0f))
, // false
            new Line(new Point(-1f, -1f, 0f), new Point(-3f, 0f, 0
f)), // true
        };
```

```
        public Sphere sphere = new Sphere();

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.Enable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            sphere.Position = new Point(0f, 0f, 0f);
            sphere.Radius = 2;

            bool[] results = new bool[] { false, false, true };
            Point result = new Point();
            for (int i = 0; i < results.Length; ++i) {
                if (Collisions.LineTest(lines[i], sphere, out resu
lt) != results[i]) {
                    LogError("Line at index " + i + " was " +
                        (results[i] ? "expected" : "not expected")
 +
                        "to intersect the test SPHERE");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            Point result = new Point();
            foreach (Line line in lines) {
                if (Collisions.LineTest(line, sphere, out result))
 {
                    GL.Color3(1f, 0f, 1f);
                    result.Render();
                    GL.Color3(0f, 2f, 0f);
                }
```

```
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                line.Render();
            }

            GL.Color3(0f, 0f, 1f);
            sphere.Render();
        }

        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Line Against AABB

The code for Line V AABB is almost the same as the code for Line V Sphere. Both functions are wrappers around a ray cast.

This test will **test intersection**, it will **not test containment**. That is, if you have a line, and both start and end points are INSIDE the sphere, the result of this test will be false!

Implementation

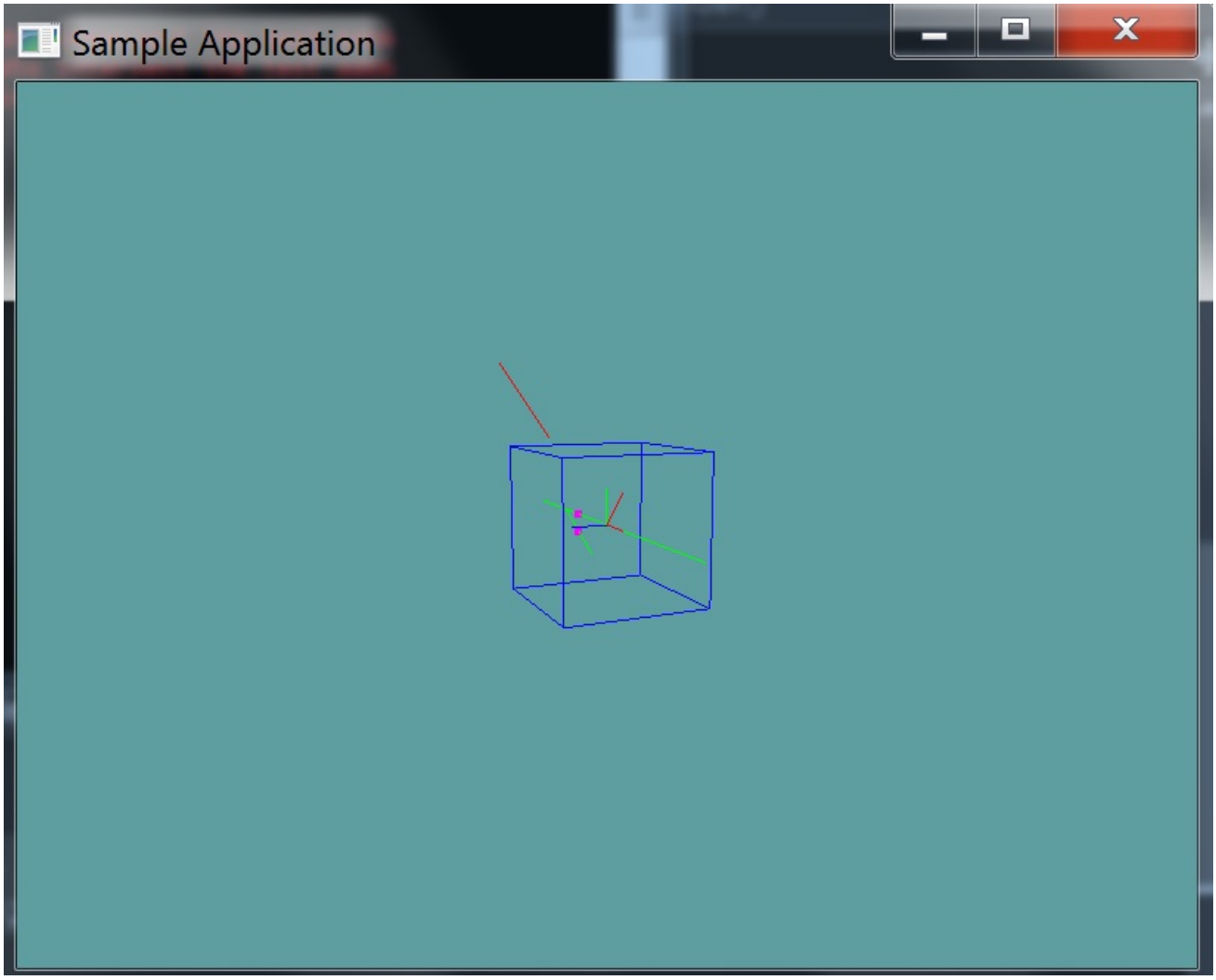
Provide an implementation for the following function

```
    public static bool LineTest(Line line, AABB aabb, out Point re
sult)
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will print out errors if any are present in your code.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class LinecastAABB : Application {
        public Line[] lines = new Line[] {
            new Line(new Point(2.5f, 2.5f, 2.5f), new Point(4, 4, 4
)), // false
            new Line(new Point(1f, 1f, 0f), new Point(0f, 0f, 0f))
, // false
            new Line(new Point(-1f, -1f, 0f), new Point(-3f, 0f, 0
f)), // true
            new Line(new Point(-5f, 0f, 0f), new Point(5f, 0f, 0f)
```

```
) // true
        };
        public AABB aabb = new AABB();

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            aabb.Max = new Point(2, 2, 2);
            aabb.Min = new Point(-2, -2, -2);

            bool[] results = new bool[] { false, false, true, true
 };
            Point result = new Point();
            for (int i = 0; i < results.Length; ++i) {
                if (Collisions.LineTest(lines[i], aabb, out result
) != results[i]) {
                    LogError("Line at index " + i + " was " +
                        (results[i] ? "expected" : "not expected")
 +
                        "to intersect the test aabb");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            Point result = new Point();
            foreach (Line line in lines) {
                if (Collisions.LineTest(line, aabb, out result)) {
                    GL.Color3(1f, 0f, 1f);
                    result.Render();
                    GL.Color3(0f, 2f, 0f);
```

```
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                line.Render();
            }

            GL.Color3(0f, 0f, 1f);
            aabb.Render();
        }

        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Line Against Plane

Again, this test is almost the same as the previous two. The code here should be really straight forward.

Implementation

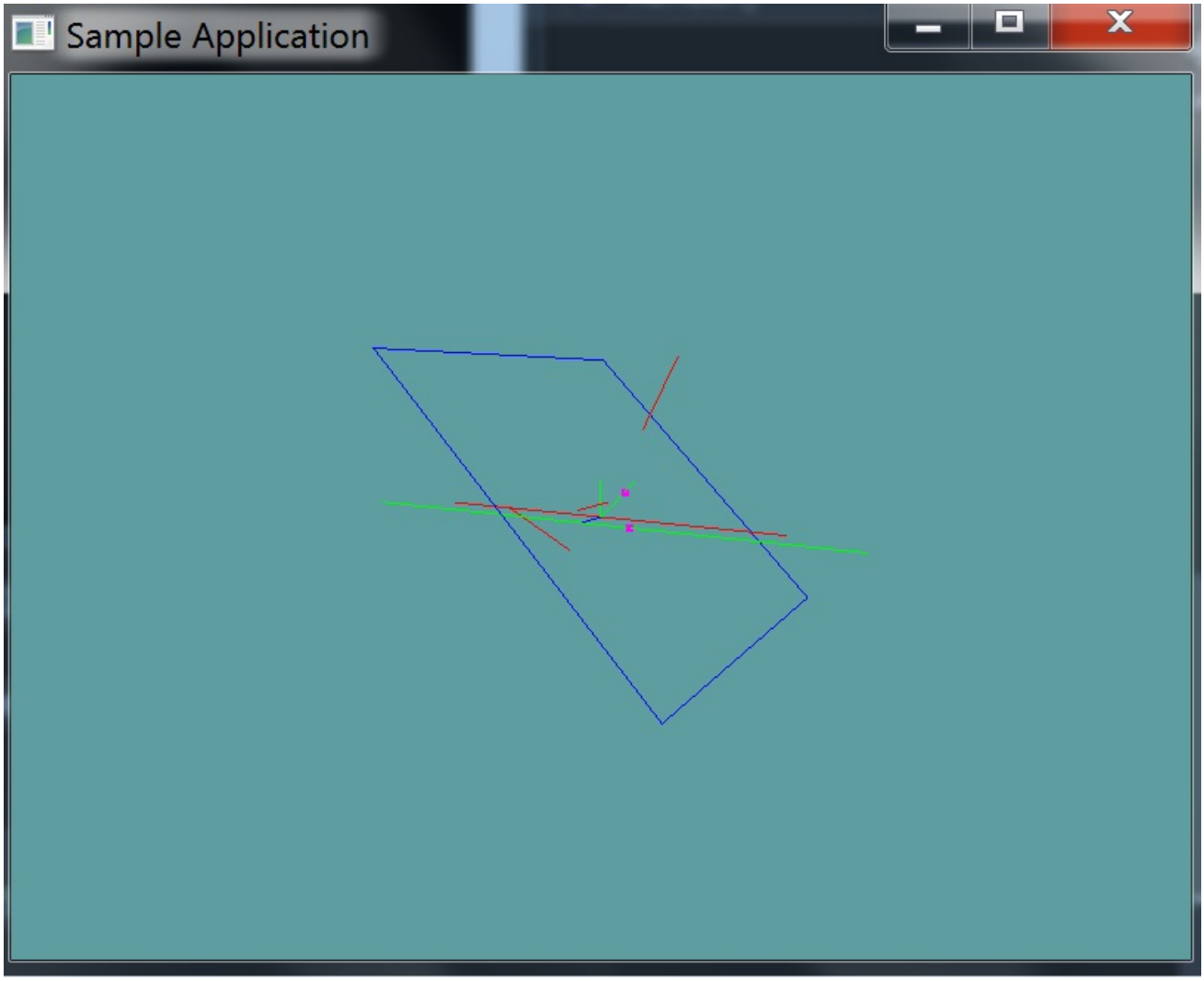
Provide an implementation for the following function

```
public static bool LineTest(Line line, Plane plane, out Point resu
lt) {
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will print out errors if any are present in your code.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class LinetestPlane : Application {
        public Line[] lines = new Line[] {
            new Line(new Point(2.5f, 2.5f, 2.5f), new Point(4, 4, 4
)), // false
            new Line(new Point(1f, 1f, 0f), new Point(0f, 0f, 0f))
, // false
            new Line(new Point(-1f, -1f, 0f), new Point(-3f, 0f, 0
f)), // false
            new Line(new Point(-5f, 0f, 0f), new Point(5f, 0f, 0f)
```

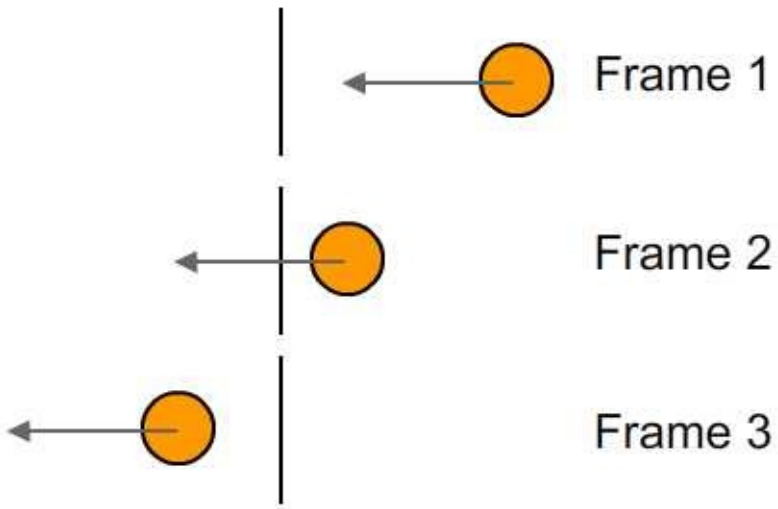
```
), // true
            new Line(new Point(7f, 0f, 1f), new Point(-7f, 0f, 1f)
), // true
            new Line(new Point(0.5f, 0.5f, 0.5f), new Point(0.5f,
0.5f, 2f)) // false
        };
        public Plane plane = new Plane();

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            plane.Normal = new Vector3(1, 1, 0);
            plane.Distance = 1;

            bool[] results = new bool[] { false, true, false, false
, true, false };
            Point result = new Point();
            for (int i = 0; i < results.Length; ++i) {
                if (Collisions.LineTest(lines[i], plane, out resul
t) != results[i]) {
                    LogError("Line at index " + i + " was " +
                        (results[i] ? "expected" : "not expected")
 +
                        "to intersect the test plane");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            Point result = new Point();
            foreach (Line line in lines) {
```

```
                if (Collisions.LineTest(line, plane, out result))
{
                    GL.Color3(1f, 0f, 1f);
                    result.Render();
                    GL.Color3(0f, 2f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                line.Render();
            }

            GL.Color3(0f, 0f, 1f);
            plane.Render(5f);
        }

        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Dynamic Intersections

Finding the collision between dynamic moving objects is a bit tougher than finding the collision between static objects. This is because of tunneling. Read this article for a bit more in depth information on the problem.

Let's take a look at the problem using this image:



The object is moving a consistent amount of space each frame. The problem is, it moves right past the plane! So a Sphere to Plane intersection (the way we know it) will not catch this collision!

Now, imagine turning that image 90 degrees to the left, the plane is the ground plane and the sphere is falling. It never hits the ground!

Special Case

The sphere tunneling test has a simple solution, but the more complex the shape, the harder the solution. **In this chapter** we are going to cover dynamic collisions of specific shapes that have trivial solutions.

This **will not** be as comprehensive a list of collisions as the dynamic ones. Because at some point you have to start getting crazy! But sometimes that is the solution. Detecting tunneling is a problem of logic, there is no standard algorithm.

Think about a moving AABB and a plane. You'd have to make a line from every vertex of the AABB in it's starting position to every vertex in it's end position, then you'd have to compare those 4 lines against the plane. If any of those 4 lines are intersecting the plane, but the aabb in it's final position is not, you have tunneling!

Knowing that you've tunneled will let you correct it. However, we are **not** going to cover the logic for tunel detection and correction of complex shapes. That's the kind of stuff you have to learn to figure out in the moment.

No Code

We are not going to write any code in this chapter. Collision resolution of dynamic objects is complex, it's often handled by a physics system.

Realistically, this is not a skill you are going to use, just understanding how it works is enough.

Time Slicing

Tunneling is inherantly a problem of time. If your delta time is lower, the less chance you have of tunneling! Becuase of this, some games solve the tunneling issue by breaking up their update loop by seperating collision detection / resolution from update steps:

```
void Update(float deltaTime) {
    foreach(GameObject go in world) {
      go.Update(deltaTime);
    }

    foreach(GameObject go in world) {
      go.ResolveCollisions(deltaTime);
    }

    foreach(GameObject go in world) {
      go.Render();
    }
}
```

Now you are probably thinking this doesn't solve the problem, and as presented above, it does not. The key is, by breaking the collision code out of update, you can slice in more collision frames.

Think of it like this, if your delta time is 0.66, you can update collisions once with a slice of 0.66, **twice** with a slice of 0.33, or **6 times every frame** with a slice of 0.11!

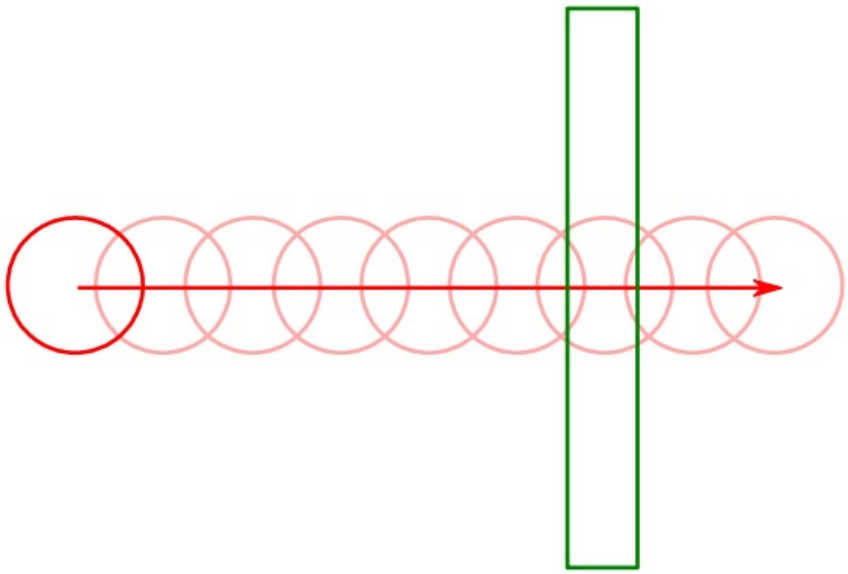
```
float sliceMax = 0.033f;
void Update(float deltaTime) {
    foreach(GameObject go in world) {
      go.Update(deltaTime);
    }

    float thisSlice = deltaTime;
    foreach(GameObject go in world) {
      while (thisSlice > 0) {
        go.ResolveCollisions(thisSlice);
        thisSlice -= sliceMax;
      }
    }

    foreach(GameObject go in world) {
      go.Render();
    }
}
```

Notice how the collision resolution will be called multiple times per frame now! Of course this is not an end-all solution, sometimes you still need to querry collisions during the update phase, which will throw this whole system out of whack.

Fun fact, this is how Bethesda games do physics, and the problem of querrying collisions outside the collision resolution method is why they have some famous physics bugs!

Linear Impact Search

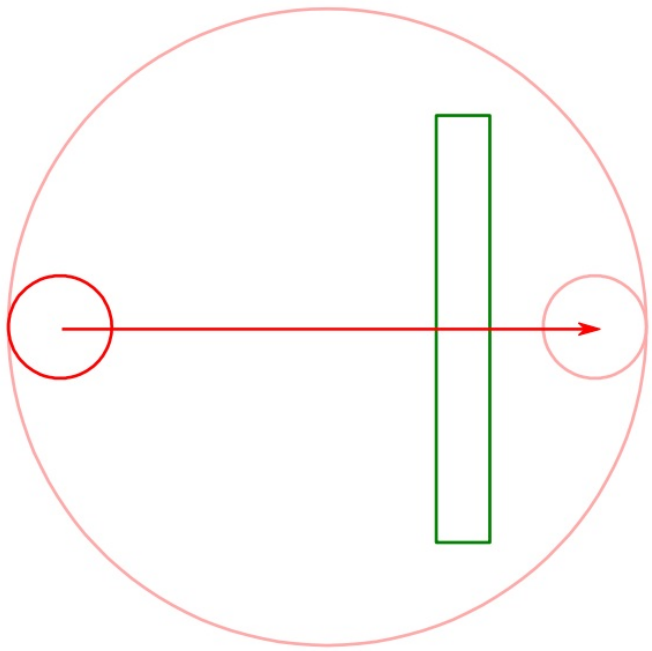
To solve the tunneling problem we can take intermediate steps into account. The brute force solution would be testing the object stepwise displaced by fractions of the full movement.



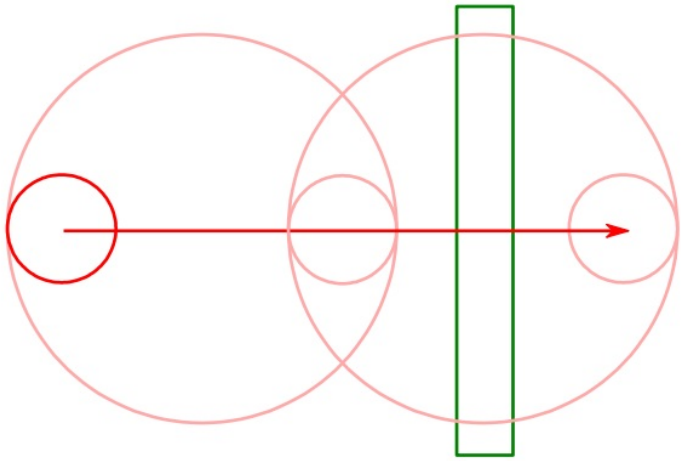
The number of tests rises proportionally with speed. This is no problem for slow objects. Whereas fast objects need many tests to avoid tunneling. Imagine a bullet fired at a wall. The high speed would result in a great number of collision tests. Now imagine a full battalion firing at that wall. The game's frame rate would start to crawl.

Binary Impact Search

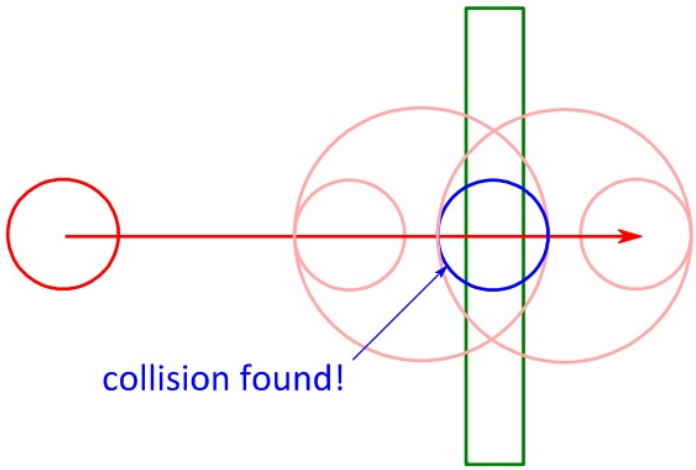
To get a better scaling algorithm we can use binary search. It starts this way:



We enclose the moving object's current position and moved position in a bounding circle. If the obstacle does not collide with this bounding circle we know there can't be a collision. Otherwise we subdivide the problem into two halves:



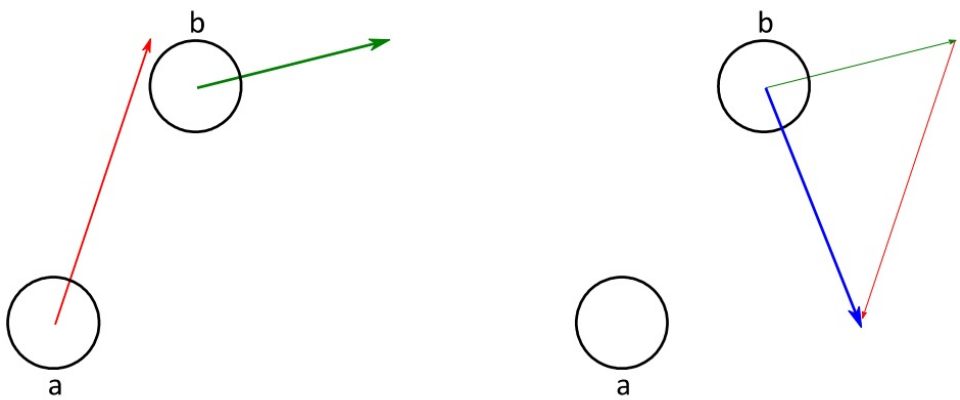
The first half represents the object covering the first half of the distance. As done before we encircle the start and end position with a bounding circle. If this bounding circle were to collide with the obstacle we would continue subdividing this movement segment. But it doesn't, so we test the second bounding circle, wrapping the halfway-through object and its destination position. This bounding circle collides with the obstacle so we subdivide the second half as done before:



This goes on until the subdivided distance gets close to zero. How close to zero is a matter of required accuracy. The moving shape's size gives a good factor for the minimum distance. This method is generic and works with all kinds of shapes. Tweaking the minimum distance gives us control over the trade-off between speed and accuracy. Longer distances result in faster collision detection, smaller distances yield more accurate results.

When Both Objects Move

... we can use a simple trick: we subtract the movement of object a from both objects' movements.



This way, object a remains stationary while object b does all the moving. Their motion relative to each other stays the same. Now we can apply the algorithms we have learned so far.

As yet we don't have a function moving_rectangle_circle_collide(). Instead of writing a full-blown binary search function we use the make-one-object-stationary trick from above:

```
Bool moving_rectangle_circle_collide(Rectangle a, Vector2D moveA,
Circle b) {
    Vector2D moveB = negate_vector(moveA);
    return moving_circle_rectangle_collide(b, moveB, a);
}
```

We just invert the movement of a and take it as movement for b. This way a becomes stationary and b moves. Now we can just use the standard test for one moving and one static colliding sphapes.

3D Models

The basic primitive shapes will only get you so far in games. At some point you might need to figure out collisions against 3D models. Tough, it's considered best practice to avoid using mesh colliders, sometimes (Like environment collision) you absoluteley need them.

Import Some Code

Before we begin implementing a 3D model, let's go ahead and add a way to load 3D models to the project. Pull in the OBJLoader from the OpenGL1X project. Make sure to place the class in the correct name space.

Once you have the class pulled in, go ahead and Download suzanne.obj. Here is some code to load up and display the model. Remember to set your working directory accordingly.

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJSample : Application {
        OBJLoader obj = null;

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            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 });

            obj = new OBJLoader("Assets/suzanne.obj");
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.PushMatrix();
            GL.Scale(3.0f, 3.0f, 3.0f);
            obj.Render();
            GL.PopMatrix();
        }
    }
}
```

The resulting window should look like this:

Sample Application

Triangle

If you don't remember how an obj is loaded / constructred, now would be a good time to review that before moving on. It's going to be important later in the chapter.

All 3D models are just a collection of triangles! As such, let's create a triangle primitive. A triangle is just a collection of 3 points. Be sure to add this class to your primitives folder, as a triangle is a primitive.

I'll implemented the entire triangle class below, it's so simple there is nothing else to do.

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

namespace CollisionDetectionSelector.Primitives {
    class Triangle {
        public Point p0 = new Point();
        public Point p1 = new Point();
        public Point p2 = new Point();

        public Triangle() {

        }

        public Triangle(Point _p0, Point _p1, Point _p2) {
            p0 = new Point(_p0.ToVector());
            p1 = new Point(_p1.ToVector());
            p2 = new Point(_p2.ToVector());
        }

        public Triangle(Vector3 _p0, Vector3 _p1, Vector3 _p2) {
            p0 = new Point(_p0);
            p1 = new Point(_p1);
```

```
            p2 = new Point(_p2);
        }

        public void Render() {
            GL.Begin(PrimitiveType.Triangles);

            GL.Vertex3(p0.X, p0.Y, p0.Z);
            GL.Vertex3(p1.X, p1.Y, p1.Z);
            GL.Vertex3(p2.X, p2.Y, p2.Z);

            GL.End();
        }

        public override string ToString() {
            return "p0: " + p0.ToString() + ", p1: " + p1.ToString
() + ", p2: " + p2.ToString();
        }
    }
}
```

Point in Triangle

There are a few ways to test if a point is inside a triangle. Most of them rely on the fact that a triangle is co-planar. That is, all 3 points of a triangle lie on the same plane. I'm going to describe three testing methods here, we will implement the third one.

Angle Test

A common way to check if a point is in a triangle is to find the vectors connecting the point to each of the triangle's three vertices and sum the angles between those vectors. If the sum of the angles is **2*pi** then the point is inside the triangle, otherwise it is not. It works, but it is very slow.

Same Side Test

The easyest way to test if a point is in a triangle is commonly refered to as the Same Side technique. I suggest reading that article, only the Same Side portion, not the barycentric one. The only problem is, the method presented is a 2D method. In addition to the 3 side tests you must also check if the test point is on the plane formed by the triangle.

Given the above paragraph and the extra condition i gave you you could build a 3D point in triangle test, but it would be an expensive one. Instead, we can do the test much cheaper.

Testing Normals

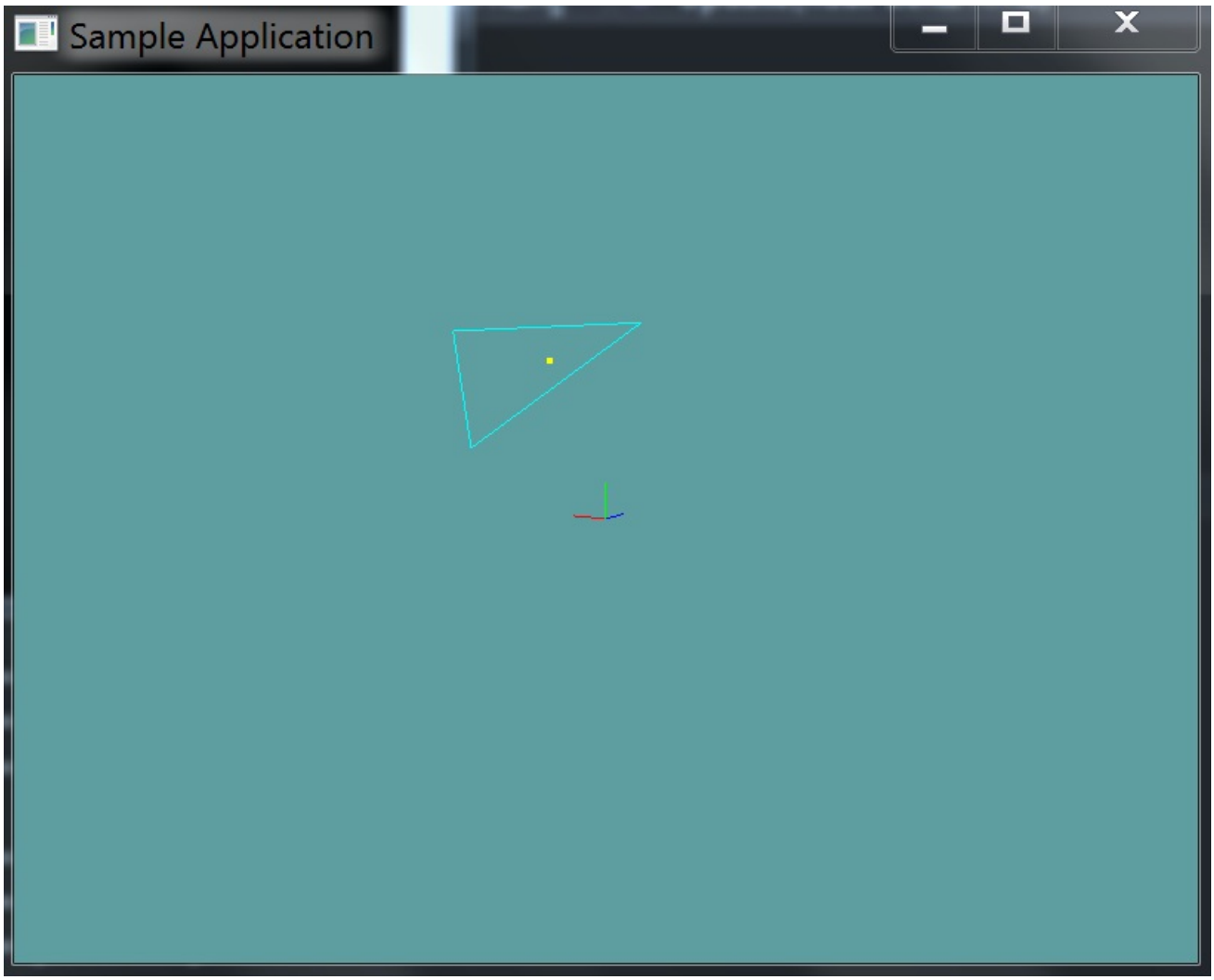
There is another intersection test we can perform, this one is easyer to read but harder to visualize. The first answer Here does a decent job of explaining the algorithm.

Given a triangle ABC and point P, we translate triangle ABC in a way that P lies on it's origin. The test now becomes checking if world origin is contained in the translated triangle (Still refered to as ABC).

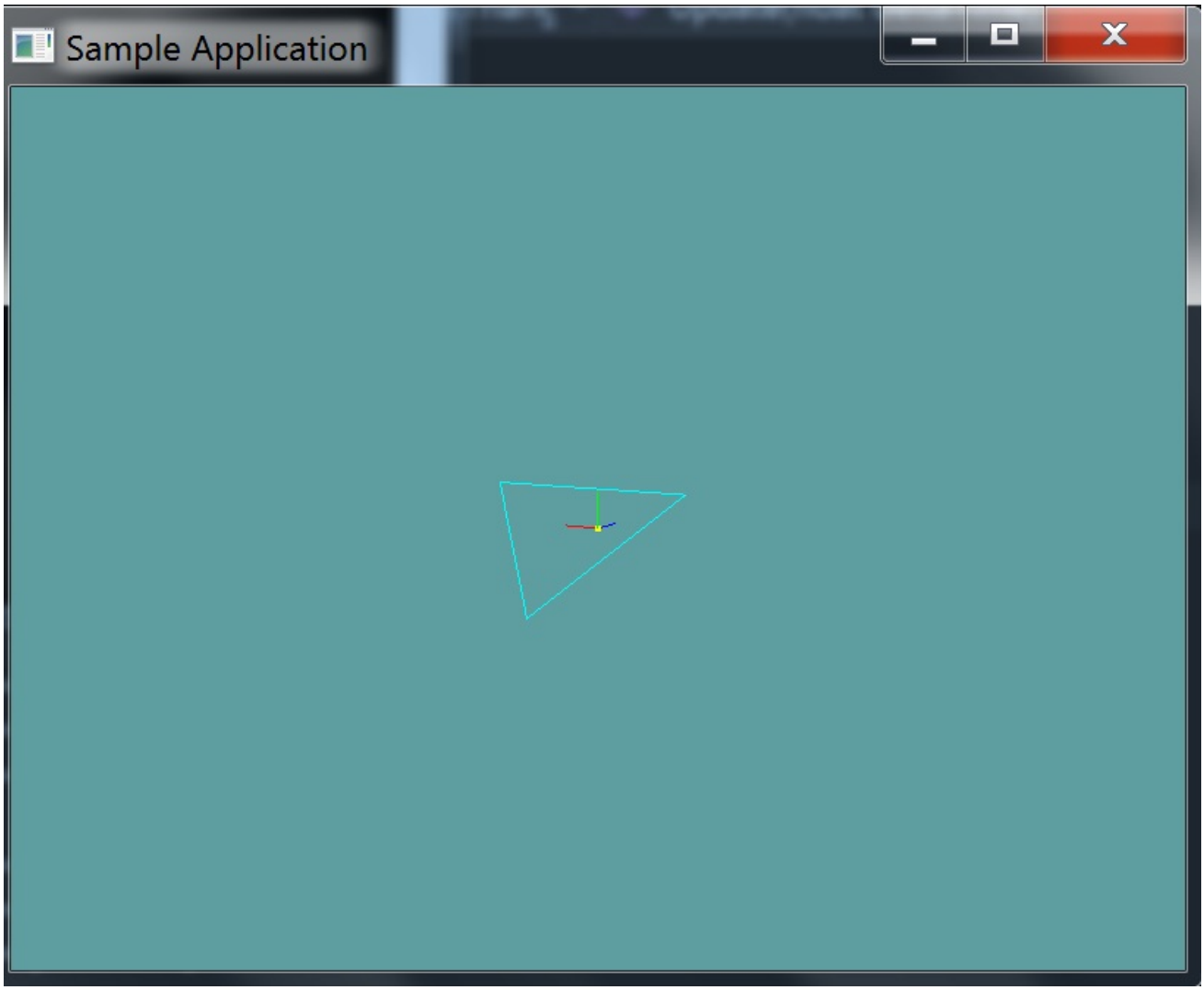
P lies in the triangle ABC only if PAB, PBC and PCA are all clockwise, or counterclockwise. Because P is at the origin, this is now just a matter of checking cross products.

- Move the triangle so that the point becomes the triangle's origin space
- Create 3 new triangles between the triangle and the point (Making a pyramid)
- Check to see if all sides of the pyramid point in the same direction (Relative to world origin)

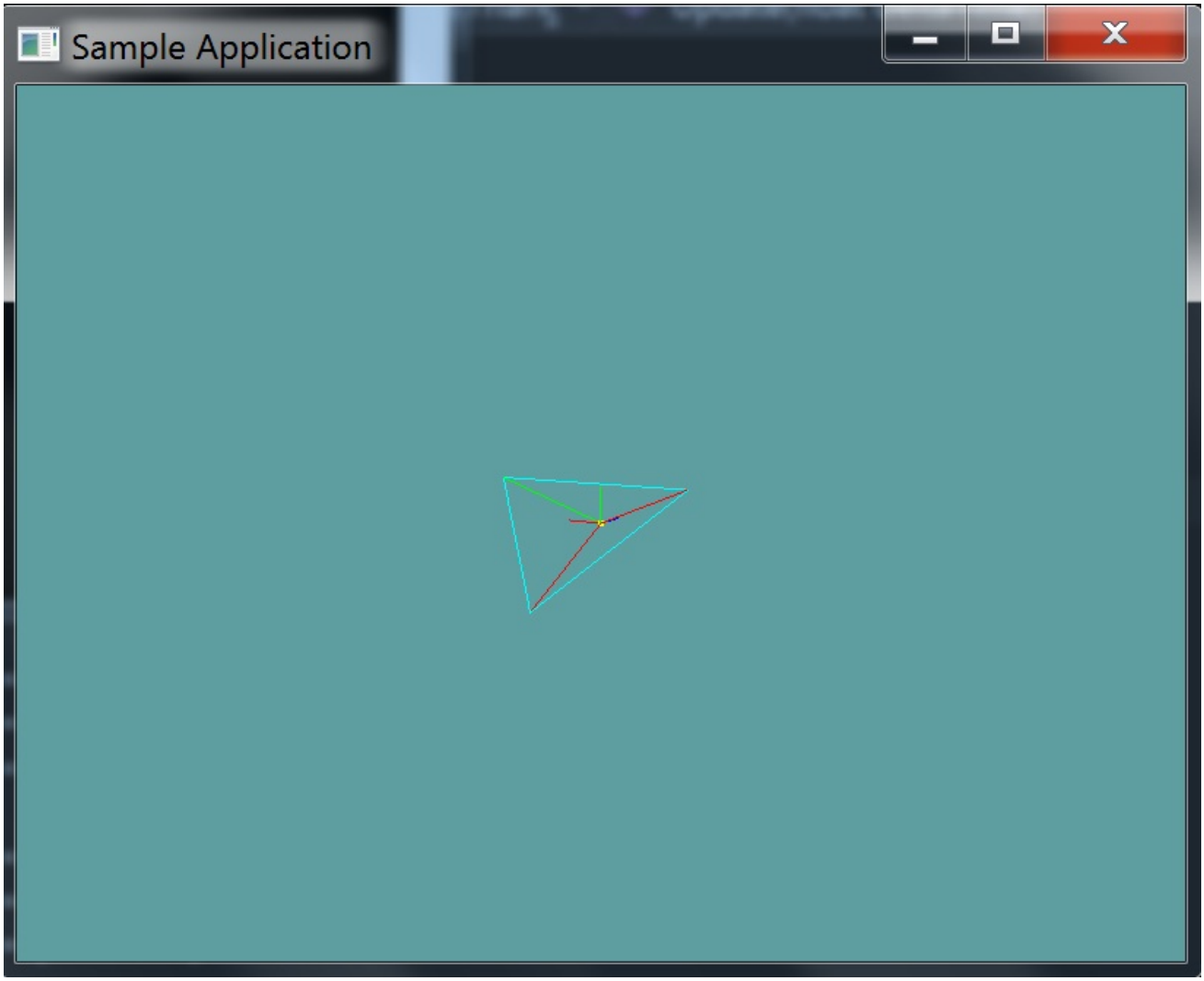
Let's see an example, we have a triangle and a point:



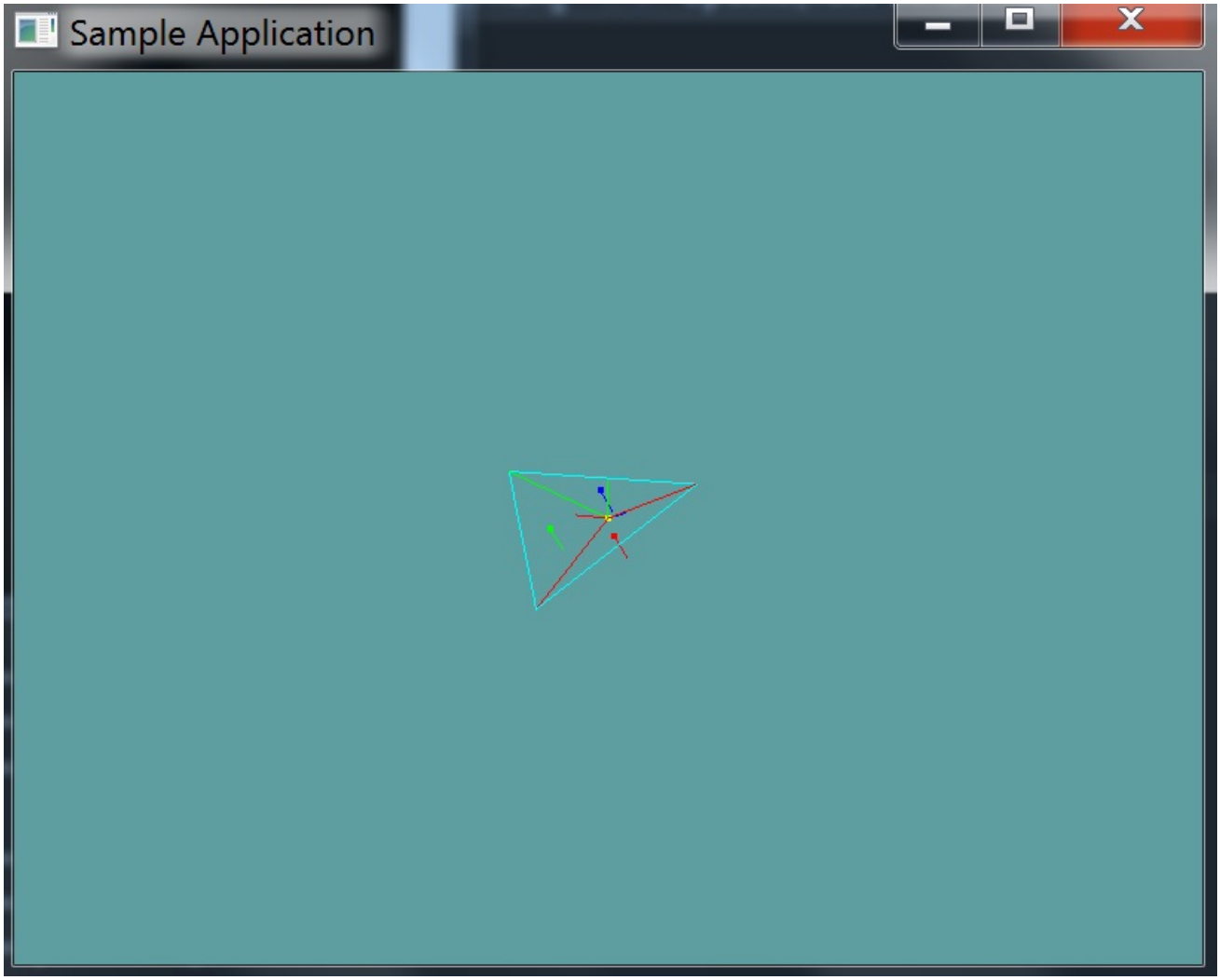
First, we translate the triangle so that the test point is it's origin:



Next we construct triangles PAB, PBC and PCA (You can only see the red and green ones, the blue one shares the same lines so it's not visible)

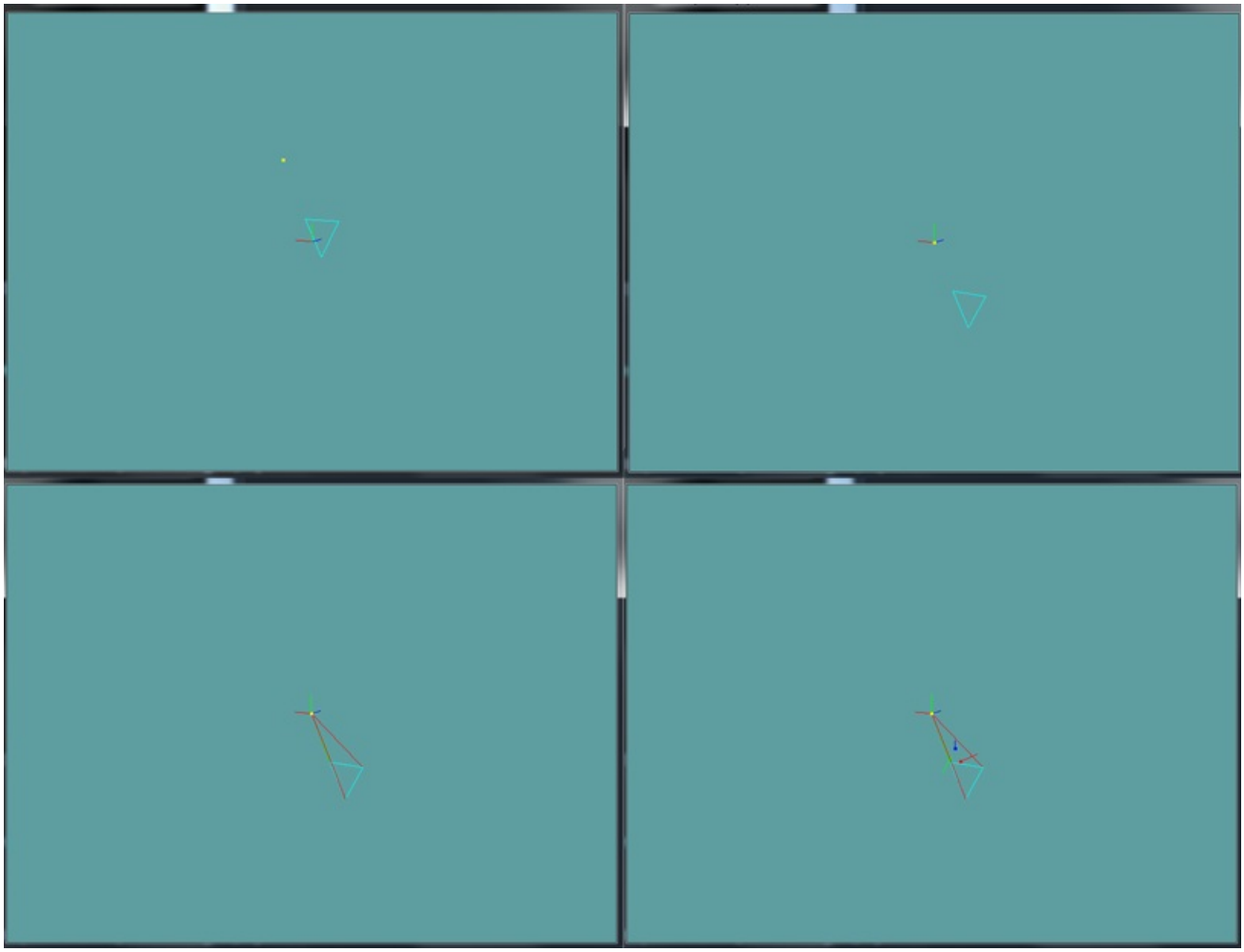


Now we check the normals of these 3 triangles



Because all of the normals are facing the same way, the triangles are wound the same. Because of this, the point is inside the triangle!

This is what the same steps look like when the point is not in a triangle:



Notice how the normals in the final image all face out!

The Algorithm

All of the above might sound a bit complicated, but its all really simple. Consider the following pseudo-code:

```
bool PointInTriangle(Point p, Triangle t) {
  // Lets define some local variables, we can change these
  // without affecting the references passed in
  Vector3 p = point;
  Vector3 a = t.p0;
  Vector3 b = t.p1;
  Vector3 c = t.p2;
```

```
    // Move the triangle so that the point becomes the
    // triangles origin
    a -= p;
    b -= p;
    c -=p;

    // The point should be moved too, so they are both
    // relative, but because we don't use p in the
    // equation anymore, we don't need it!
    // p -= p;

    // Compute the normal vectors for triangles:
    // u = normal of PBC
    // v = normal of PCA
    // w = normal of PAB

    Vector3 u = Cross(b, c);
    Vector3 v = Cross(c, a);
    Vector3 w = Cross(a, b);

    // Test to see if the normals are facing
    // the same direction, return false if not
    if (Dot(u, v) < 0f) {
        return false;
    }
    if (dot(u, w) < 0.0f) {
        return false;
    }

    // All normals facing the same way, return true
    return true;
}
```

Using the dot product to check if two normals point the same way is standard. The only thing that might be confusing here is how we use the cross products to get a normal.

On Your Own

Add the following function to the `Collisions` class:

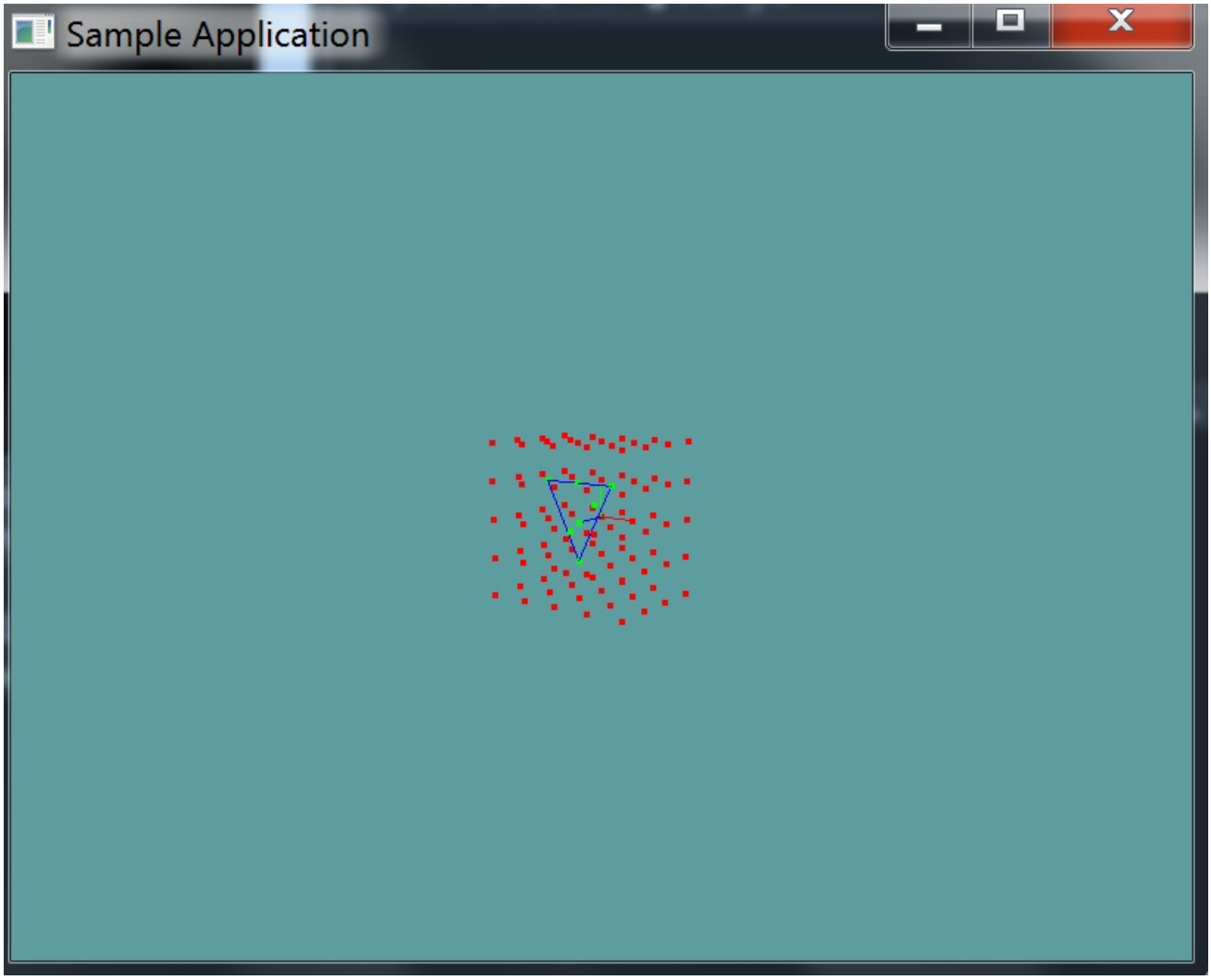
```
public static bool PointInTriangle(Triangle triangle, Point point)
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will render a grid of points and a triangle. If any point is outside the triangle, it's red. Any point inside the triangle is green. If you have any errors in your collision code, the constructor will alert you.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class PointInTriangle : Application {
        protected Point[] points = null;

        Triangle triangle = new Triangle(new Point(-1.0f, 1.0f, 1.0
f), new Point(0.0f, -1.0f, 1.0f), new Point(1.0f, 1.0f, 1.0f));

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);
```

```
            GL.Disable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            int i_min = -2;
            int i_max = 3;
            int j_min = -2;
            int j_max = 3;
            int k_min = -1;
            int k_max = 3;

            int indexer = 0;
            for (int i = i_min; i < i_max; ++i) {
                for (int j = j_min; j < j_max; ++j) {
                    for (int k = k_min; k < k_max; ++k) {
                        indexer++;
                    }
                }
            }

            points = new Point[indexer + 3];
            indexer = 0;

            for (int i = i_min; i < i_max; ++i) {
                for (int j = j_min; j < j_max; ++j) {
                    for (int k = k_min; k < k_max; ++k) {
                        points[indexer++] = new Point(i, j, k);
                    }
                }
            }

            points[indexer++] = new Point(0.5f, 0.5f, 1f);
            points[indexer++] = new Point(0.5f, -0.25f, 1f);
            points[indexer++] = new Point(-0.25f, -0.25f, 1f);


            int[] colliding = new int[] { 34, 46, 50, 54, 74, 100,
```

```
102 };
            int col_index = 0;

            for (int i = 0; i < points.Length; ++i) {
                if (col_index < colliding.Length && i == colliding
[col_index]) {
                    if (!Collisions.PointInTriangle(triangle, poin
ts[i])) {
                        LogError("point " + i + " SHOULD be collid
ing!");
                    }

                    col_index++;
                }
                else {
                    if (Collisions.PointInTriangle(triangle, point
s[i])) {
                        LogError("point " + i + " should NOT be co
lliding!");
                    }
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(0.0f, 0.0f, 1.0f);
            triangle.Render();

            foreach(Point point in points) {
                if (Collisions.PointInTriangle(triangle, point)) {
                    GL.Color3(0.0f, 1.0f, 0.0f);
                }
                else {
                    GL.Color3(1.0f, 0.0f, 0.0f);
```

```
                    }

                    point.Render();
                }
            }
        }
    }
```

Closest point to Triangle

Now that we can test if a point is inside a triangle or not, finding the closest point to a triangle becomes relativley simple. The process is as follows:

- Make a new plane out of the triangle
- Get the closest point on the test plane to your test point
    - This is your new test point
- Is the point inside the triangle?
    - If so, that point is the closest point
- Convert each edge of the triangle to a line
- Get the closest point on line
- Check the distance between the test point and all 3 closest points
- Return the point with the shortest distance.

Given that a triangle already consists of 3 points ABC (p0, p1 and p2), getting the 3 lines of AB, BC and CA becomes very simple. Especially when you consider that the line constructor takes two points.

Checking which point is closest is easy too, consider the following:

```
Point GetClosest(Point test, Point test1, Point test2, Point test2
) {
  // Get 3 vectors, going from the test point to all potential can
didates
  Vector3 diff1 = test - test1; // Vector from test point to poten
tially closest 1
  Vector3 diff2 = test - test2; // Vector from test point to poten
tially closest 2
  Vector3 diff3 = test - test3; // Vector from test point to poten
tially closest 3

  // Find out how far away each point is using the difference
  // vectors computed above
  float distSq1 = diff1.LengthSq();
  float distSq2 = diff2.LengthSq();
  float distSq3 = diff3.LengthSq();

  // Find the shortest one
  float min = Min(distSq1, distSq2);
  min = Min(max, distSq3);

  // Finally, return the result
  if (min == distSq1) {
    return test1;
  }
  else if (min == distSq2) {
    return test2;
  }
  return test3;
}
```

The Algorithm

Make sure you return a new point object, not a reference to the old point!

```
public static Point ClosestPoint(Triangle triangle, Point point) {
    Plane plane = new Plane(triangle.p0, triangle.p1, triangle.p2)
;
    point = ClosestPoint(plane, point);

    if (PointInTriangle(triangle, point)) {
        return new Point(point);
    }

    Line AB = new Line(triangle.p0, triangle.p1);
    Line BC = new Line(triangle.p1, triangle.p2);
    Line CA = new Line(triangle.p2, triangle.p0);

    Point c1 = ClosestPoint(AB, point);
    Point c2 = ClosestPoint(BC, point);
    Point c3 = ClosestPoint(CA, point);

    float mag1 = (point.ToVector() - c1.ToVector()).LengthSquared(
);
    float mag2 = (point.ToVector() - c2.ToVector()).LengthSquared(
);
    float mag3 = (point.ToVector() - c3.ToVector()).LengthSquared(
);

    float min = Math.Min(mag1, mag2);
    min = Math.Min(min, mag3);

    if (min == mag1) {
        return c1;
    }
    else if (min == mag2) {
        return c2;
    }
    return c3;
}
```

On Your Own

Add the following function to the `Collisions` class:

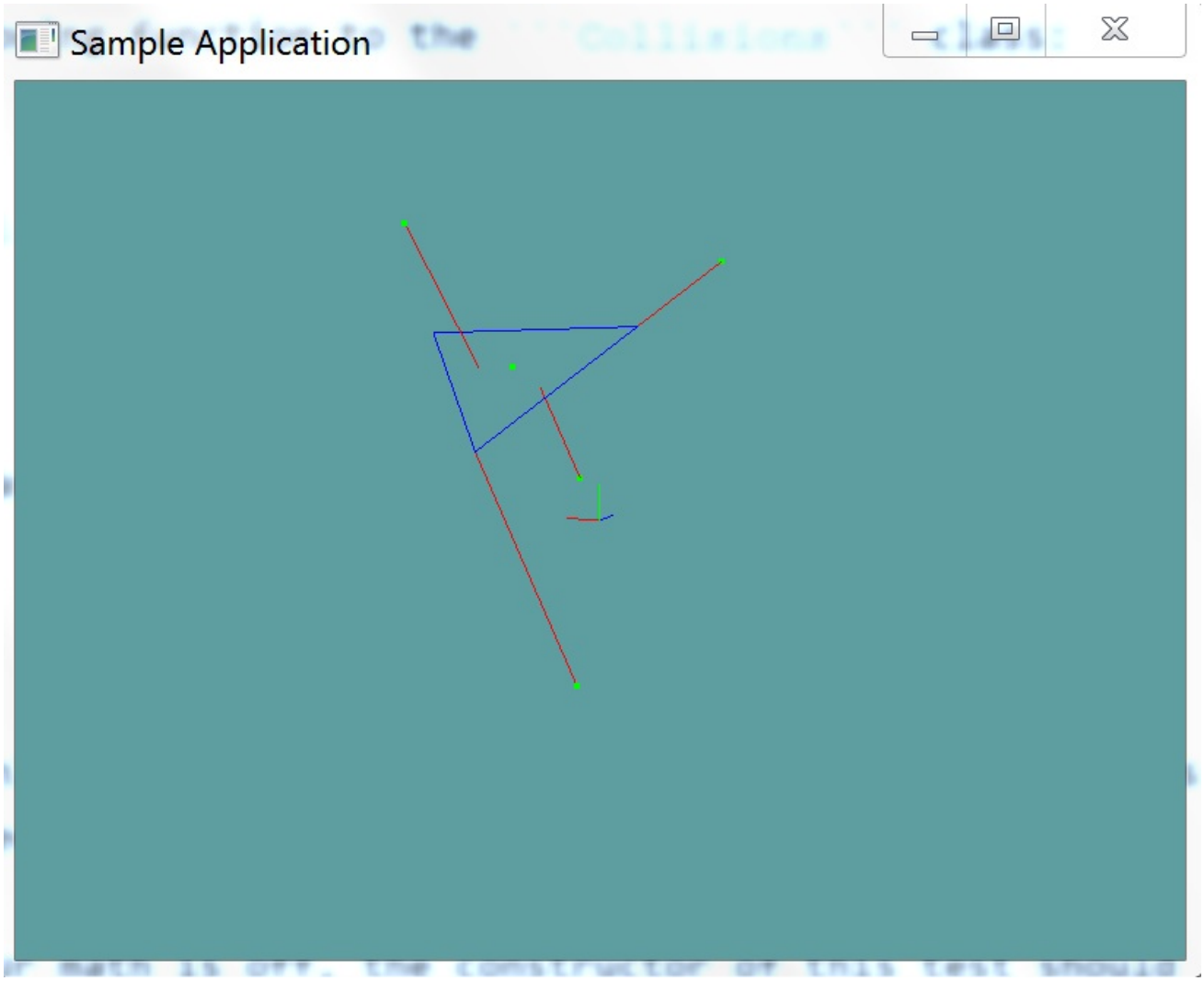
```
public static Point ClosestPoint(Triangle triangle, Point point)
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

If any of your math is off, the constructor of this test should throw errors. You should read trough the test, take note how i had to use an epsilon compare to compare two points.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class ClosestPointTriangle : Application {
        protected Point[] points = new Point[] {
            new Point(1f, 1f, 1f),
            new Point(-1f, -3f, -4f),
            new Point(2, 4, -1),
            new Point(-2.732051f, 6.732051f, 1.732051f),
            new Point(3, 7, -4)
        };
        Point[] tests = new Point[] {
```

```
            new Point(1, 3.5f, -1.5f),
            new Point(2, 2, -3),
            new Point(2, 4, -1),
            new Point(-1, 5, 0),
            new Point(3, 4, -0.9999995f),
        };

        Triangle triangle = new Triangle(new Point(-1.0f, 5.0f, 0.0
f), new Point(2.0f, 2.0f, -3.0f), new Point(5.0f, 5.0f, 0.0f));

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);
            GL.Disable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            for (int i = 0; i < points.Length; ++i) {
                Point c = Collisions.ClosestPoint(triangle, points
[i]);
                if (!PointCompare(c, tests[i])) {
                    LogError("Expected point " + i + " to be: " +
tests[i] + ", got: " + c);
                }
            }
        }

        protected bool PointCompare(Point p1, Point p2) {
            float epsilon = 0.00001f;

            float x = System.Math.Abs(p1.X - p2.X);
            float y = System.Math.Abs(p1.Y - p2.Y);
            float z = System.Math.Abs(p1.Z - p2.Z);

            if (x > epsilon) return false;
            if (y > epsilon) return false;
            if (z > epsilon) return false;
```

```
            return true;
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(0.0f, 0.0f, 1.0f);
            triangle.Render();

            GL.Color3(1.0f, 0.0f, 0.0f);
            GL.Begin(PrimitiveType.Lines);
            for (int i = 0; i < points.Length; ++i) {
                Point closest = Collisions.ClosestPoint(triangle,
points[i]);
                GL.Vertex3(points[i].X, points[i].Y, points[i].Z);
                GL.Vertex3(closest.X, closest.Y, closest.Z);
            }
            GL.End();

            GL.Color3(0.0f, 1.0f, 0.0f);
            foreach (Point point in points) {
                point.Render();
            }
        }
    }
}
```

Sphere triangle test

Testing for the intersection of a sphere and a trianlge is amazingly simple. Get the closest point on the triangle to the center of the sphere. Check the distance (squared) between the center of the sphere and this point. If the distance is less than the sphere's radius (squared), then there is a collision.

The Algorithm

This one is so simple, i'm not going to be providing sample code for it.

On Your Own

Add the following function to the `Collisions` class:

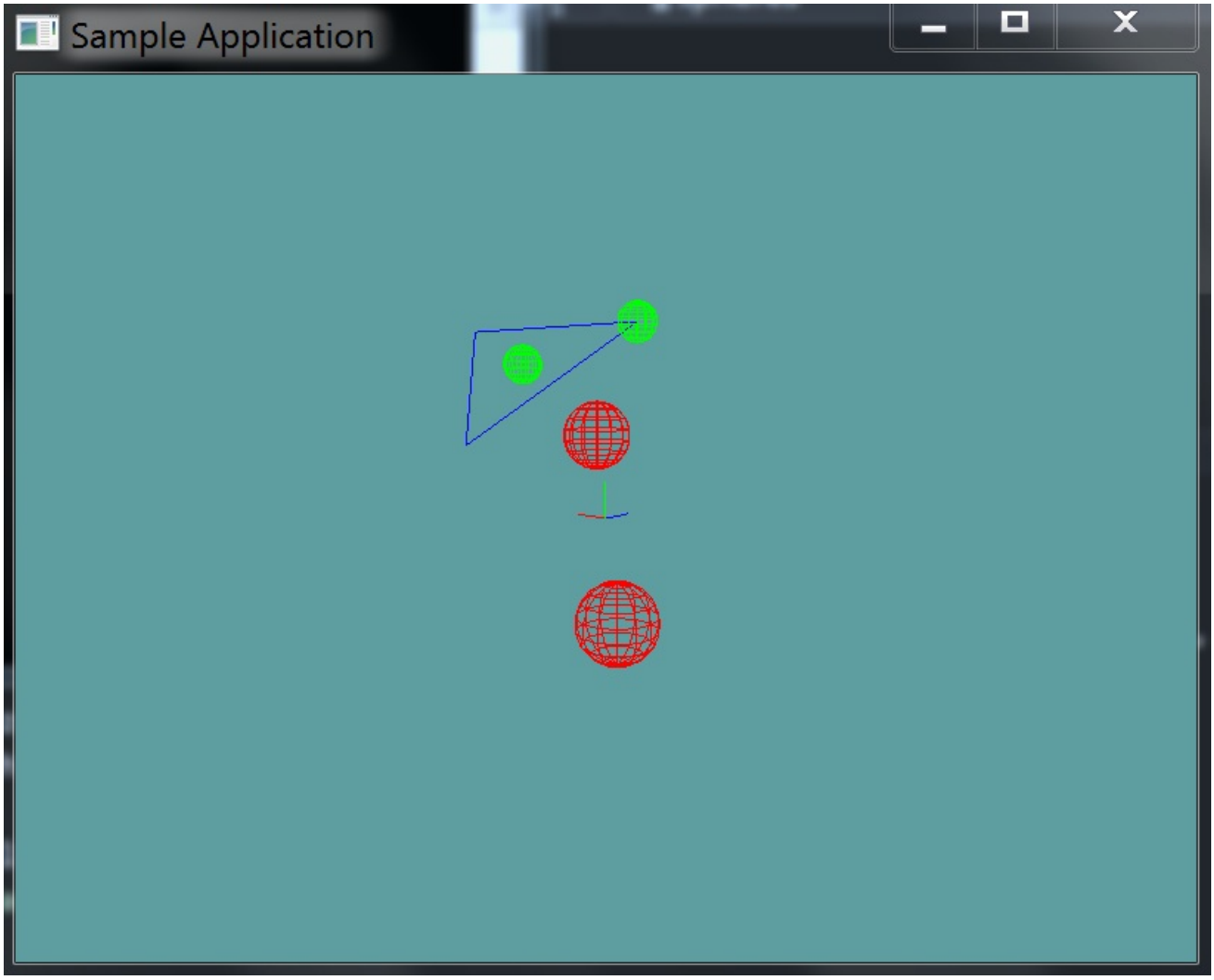
```
// TODO: Implement this function
public static bool Intersects(Triangle triangle, Sphere sphere)

// This is just a conveniance function
public static bool Intersects(Sphere sphere, Triangle triangle) {
    return Intersects(triangle, sphere);
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will print errors if your collision code is off. Any sphere that intersects the triangle is rendered in green. All other spheres are rendered in red.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class TriangleSphereIntersection : Application {
        Sphere[] spheres = new Sphere[] {
            new Sphere(new Point(2, 4, -1), 0.5f),
            new Sphere(new Point(-1.0f, 5.0f, 0.0f), 0.5f),
            new Sphere(new Point(2f, 2f, 2f), 1f),
            new Sphere(new Point(-2f, -2f, -2f), 1f)
        };

        Triangle triangle = new Triangle(new Point(-1.0f, 5.0f, 0.0
```

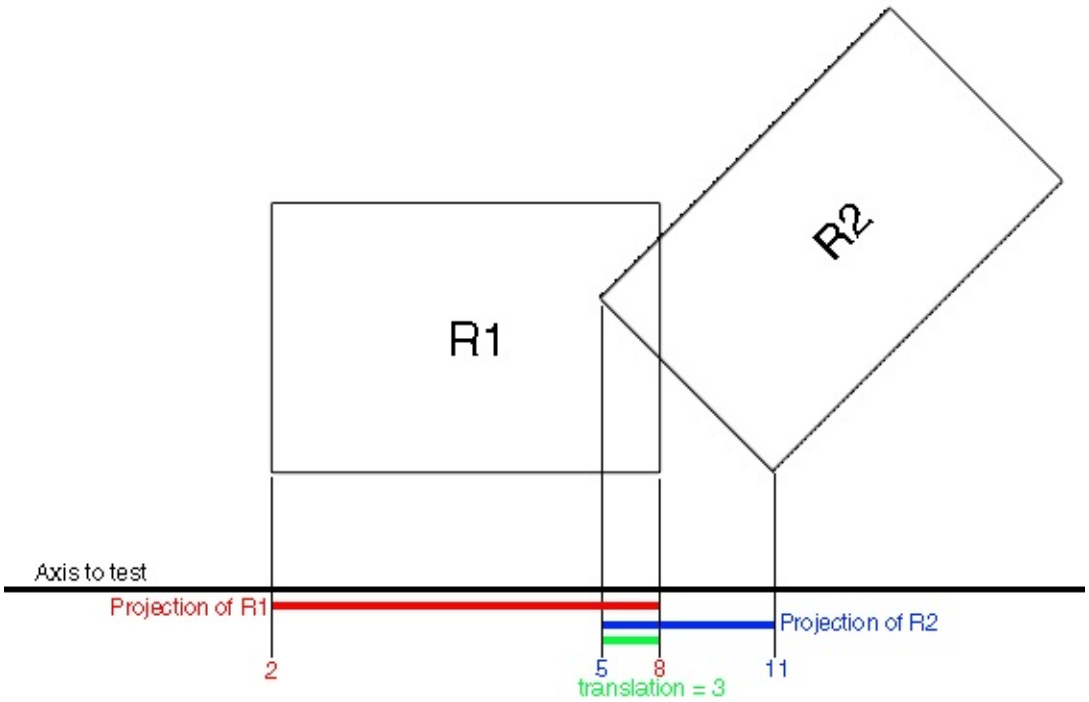
```
f), new Point(2.0f, 2.0f, -3.0f), new Point(5.0f, 5.0f, 0.0f));

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);
            GL.Disable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            bool[] expected = new bool[] { true, true, false, false
 };
            for (int i = 0; i < spheres.Length; ++i) {
                bool result = Collisions.Intersects(triangle, sphe
res[i]);
                if (result != expected[i]) {
                    LogError("Expected sphere " + i + " to " +
                        (expected[i] ? " intersect" : " NOT inters
ect") +
                        " the triangle");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(0.0f, 0.0f, 1.0f);
            triangle.Render();

            foreach(Sphere sphere in spheres) {
                if (Collisions.Intersects(triangle, sphere)) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
```

```
                    sphere.Render();
                }
            }
        }
    }
```

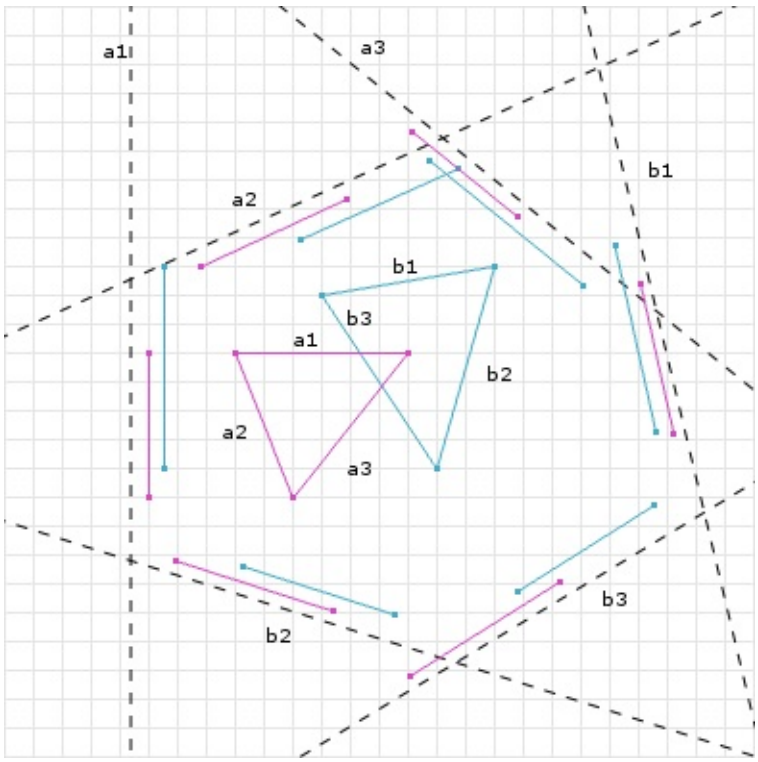
AABB Triangle intersection

Checking an AABB against a triangle is done using the **Seperating Axis Therom** or **SAT**. A great overview of the method can be found Here.

The theory behind the SAT is simple, for any two convex objects, if there exists an axis that seperates them, no collision occurs. Take the following image for example:



We have two boxes, and an axis to test them on. We project the two boxes onto the axis, the projections overlap, therefore the two boxes intersect on that axis. Here is a more complicated example of two triangles:



The triangles intersect on all projected axis. This means there is no axis of seperation, so there is an intersection.

In both of the above examples the primitives are projected onto an axis and the endpoints of their projected lines are tested for an intersection. There is a more optimal way to do this. We can translate both primitives so that one is at the center of the world.

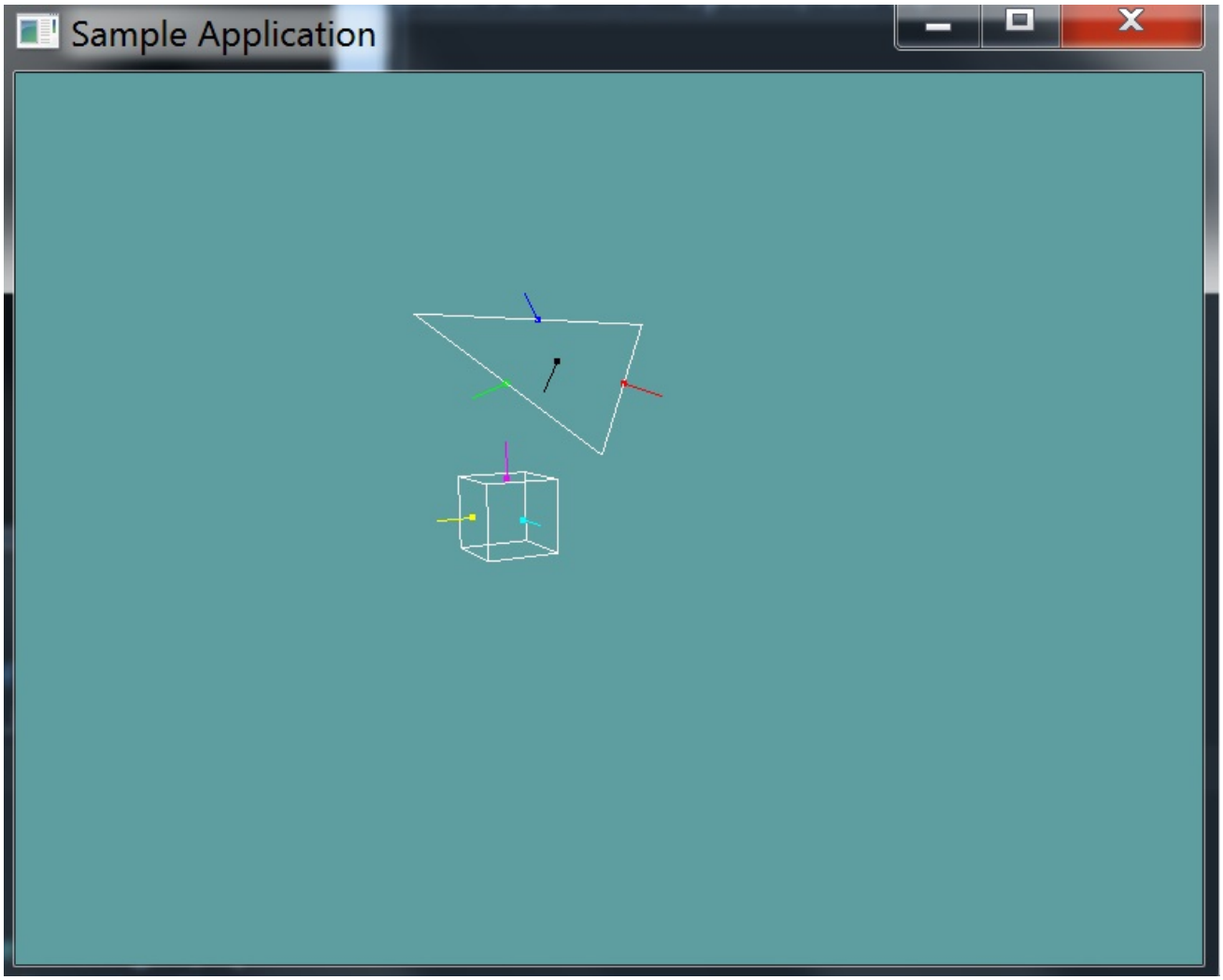
When one primitive is at the center of the world, the projection test just needs to compare relative lengths, not fixed projected points.

AABB Triangle specific

An AABB v Triangle intersection test has a total of 13 axis which could potentially seperate it. They are:

- 9 axis given by the cross profucts of the combined edges of both
- 3 face normals from the AABB
- 1 face normal from the triangle

Consider the following image:



If we want to get the axis seperating the blue triangle normal and the magenta AABB normal, we have to cross the two vectors.

```
Blue X Magenta = SAT0
```

We get 9 separating axis by getting the cross product of each of the colored (NOT THE BLACK) normals.

We get 3 separating axis from the AABB: the cyan, magenta and yellow normals are all an axis of seperation

We get 1 separating axis from the black triangle normal. It is an axis of it's own.

That's how we get the 9 seperating axis.

Projecting

As mentioned before, we would normally project both of the primitives onto the seperating axis and compare the projected lines. But we can optimize this!

By subtracting the center of the AABB from both the AABB and the triangle

```
triangle.p0 -= aabb.center;
triangle.p1 -= aabb.center
triangle.p2 -= aabb.center;

aabb.center -= aabb.center
```

We put the AABB at the center of the world. Now, all we care about is the half-length of the projected AABB. We know it's center is at (0, 0). We simply compare the triangles projected points against the projected half-length.

The Algorithm

As mentioned earlyer, in order for a triangle and an AABB to intersect, all 13 axis have to be NON SEPERATED. When we find the first seperating axis, it's safe to return false.

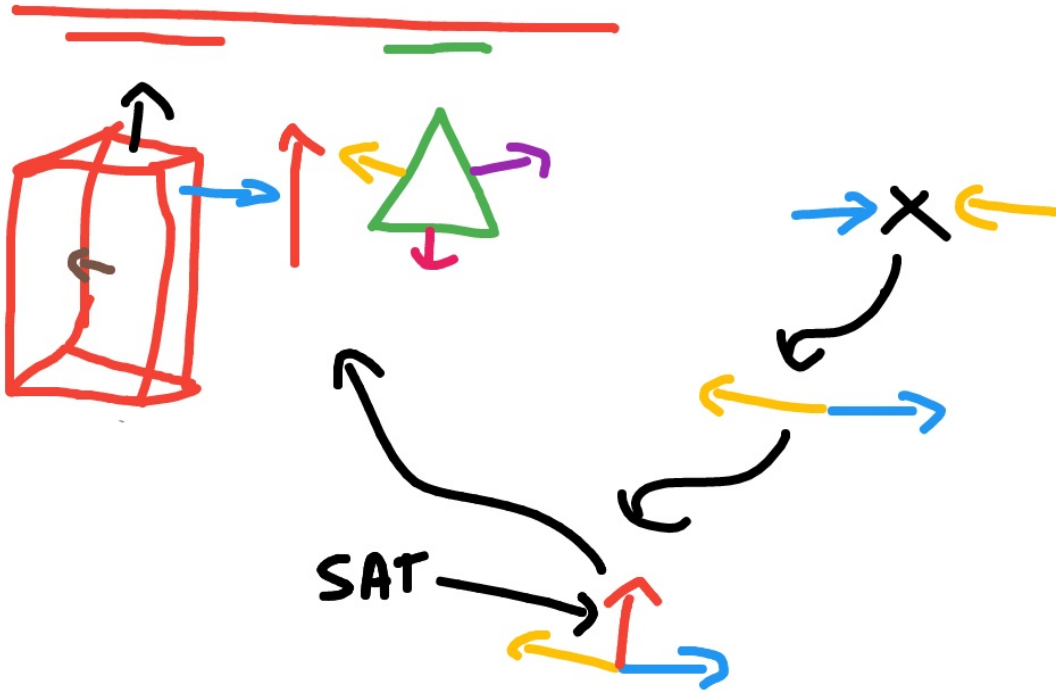
```
public static bool Intersects(Triangle triangle, AABB aabb) {
    // Get the triangle points as vectors
    Vector3 v0 = triangle.p0.ToVector();
    Vector3 v1 = triangle.p1.ToVector();
    Vector3 v2 = triangle.p2.ToVector();

    // Convert AABB to center-extents form
    Vector3 c = aabb.Center.ToVector();
    Vector3 e = aabb.Extents;
```

```
    // Translate the triangle as conceptually moving the AABB to o
rigin
    // This is the same as we did with the point in triangle test
    v0 -= c;
    v1 -= c;
    v2 -= c;

    // Compute the edge vectors of the triangle  (ABC)
    // That is, get the lines between the points as vectors
    Vector3 f0 = v1 - v0; // B - A
    Vector3 f1 = v2 - v1; // C - B
    Vector3 f2 = v0 - v2; // A - C

    // Compute the face normals of the AABB, because the AABB
    // is at center, and of course axis aligned, we know that
    // it's normals are the X, Y and Z axis.
    Vector3 u0 = new Vector3(1.0f, 0.0f, 0.0f);
    Vector3 u1 = new Vector3(0.0f, 1.0f, 0.0f);
    Vector3 u2 = new Vector3(0.0f, 0.0f, 1.0f);

    // There are a total of 13 axis to test!

    // We first test against 9 axis, these axis are given by
    // cross product combinations of the edges of the triangle
    // and the edges of the AABB. You need to get an axis testing
    // each of the 3 sides of the AABB against each of the 3 sides
    // of the triangle. The result is 9 axis of seperation
    // https://awwapp.com/b/umzoc8tiv/

    // Compute the 9 axis
    Vector3 axis_u0_f0 = Vector3.Cross(u0, f0);
    Vector3 axis_u0_f1 = Vector3.Cross(u0, f1);
    Vector3 axis_u0_f2 = Vector3.Cross(u0, f2);

    Vector3 axis_u1_f0 = Vector3.Cross(u1, f0);
    Vector3 axis_u1_f1 = Vector3.Cross(u1, f1);
```

```
    Vector3 axis_u1_f2 = Vector3.Cross(u2, f2);

    Vector3 axis_u2_f0 = Vector3.Cross(u2, f0);
    Vector3 axis_u2_f1 = Vector3.Cross(u2, f1);
    Vector3 axis_u2_f2 = Vector3.Cross(u2, f2);

    // Testing axis: axis_u0_f0
    // Project all 3 vertices of the triangle onto the Seperating
axis
    float p0 = Vector3.Dot(v0, axis_u0_f0);
    float p1 = Vector3.Dot(v1, axis_u0_f0);
    float p2 = Vector3.Dot(v2, axis_u0_f0);
    // Project the AABB onto the seperating axis
    // We don't care about the end points of the prjection
    // just the length of the half-size of the AABB
    // That is, we're only casting the extents onto the
    // seperating axis, not the AABB center. We don't
    // need to cast the center, because we know that the
    // aabb is at origin compared to the triangle!
    float r = e.X * Math.Abs(Vector3.Dot(u0, axis_u0_f0)) +
                e.Y * Math.Abs(Vector3.Dot(u1, axis_u0_f0)) +
                e.Z * Math.Abs(Vector3.Dot(u2, axis_u0_f0));
    // Now do the actual test, basically see if either of
    // the most extreme of the triangle points intersects r
    // You might need to write Min & Max functions that take 3 arg
uments
    if (Max(-Max(p0, p1, p2), Min(p0, p1, p2)) > r) {
        // This means BOTH of the points of the projected triangle
        // are outside the projected half-length of the AABB
        // Therefore the axis is seperating and we can exit
        return false;
    }

    // Repeat this test for the other 8 seperating axis
    // You may wish to make some kind of a helper function to keep
    // things readable
    TODO: 8 more SAT tests
```

```
    // Next, we have 3 face normals from the AABB
    // for these tests we are conceptually checking if the boundin
g box
    // of the triangle intersects the bounding box of the AABB
    // that is to say, the seperating axis for all tests are axis
aligned:
    // axis1: (1, 0, 0), axis2: (0, 1, 0), axis3 (0, 0, 1)
     TODO: 3 SAT tests
    // Do the SAT given the 3 primary axis of the AABB
    // You already have vectors for this: u0, u1 & u2

    // Finally, we have one last axis to test, the face normal of
the triangle
    // We can get the normal of the triangle by crossing the first
 two line segments
    Vector3 triangleNormal = Vector3.Cross(f0, f1);
    TODO: 1 SAT test

    // Passed testing for all 13 seperating axis that exist!
    return true;
}
```

This is the image references in the code as https://awwapp.com/b/umzoc8tiv/



On Your Own

Add the following function to the `Collisions` class:

```
public static bool Intersects(Triangle triangle, AABB aabb)

public static bool Intersects(AABB aabb, Triangle triangle) {
    return Intersects(triangle, aabb);
}
```

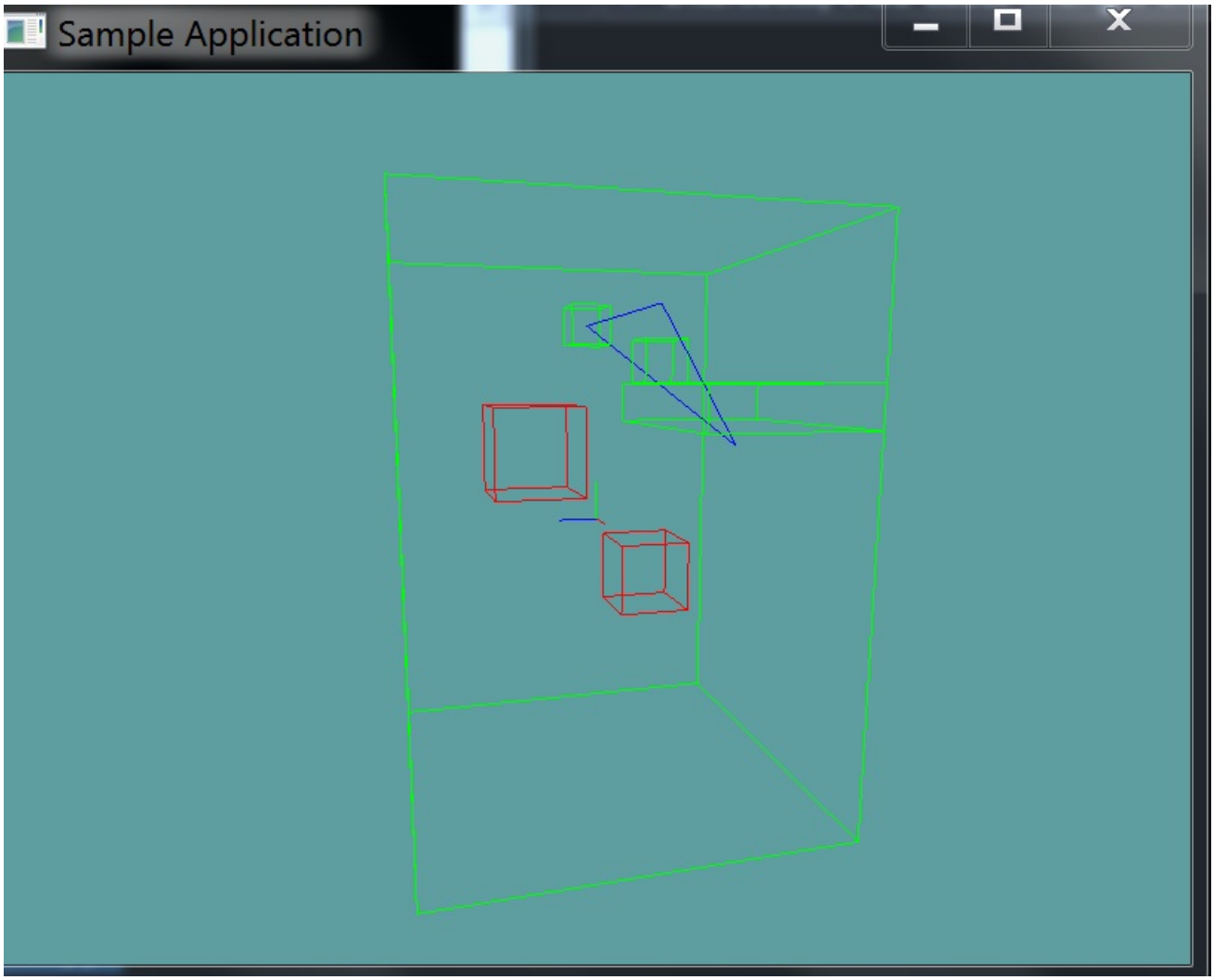
And provide an implementation for it! The code provided above should serve as a solid guide for doing this

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will render out a triangle and several AABB's. If an AABB is intersecting the triangle it is rendered in green, if it isn't it's rendered in red.

The constructor of the unit test will throw errors if your code is off.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class TriangleAABBIntersection : Application {
        AABB[] aabbs = new AABB[] {
            new AABB(new Point(2, 4, -1), new Vector3(0.5f, 0.5f,
0.5f)),
            new AABB(new Point(-1.0f, 5.0f, 0.0f), new Vector3(0.5
```

```
f, 0.5f, 0.5f)),
            new AABB(new Point(0.0f, 0.0f, 0.0f), new Vector3(5f, 7
f, 5f)),
            new AABB( new Point(2.0f, 3.0f, -3.0f), new Vector3(3f
, 0.5f, 2f)),


            new AABB(new Point(2f, 2f, 2f), new Vector3(1.0f, 1.0f
, 1.0f)),
            new AABB(new Point(-2f, -2f, -2f), new Vector3(1.0f, 1
.0f, 1.0f))
        };

        Triangle triangle = new Triangle(new Point(-1.0f, 5.0f, 0.0
f), new Point(2.0f, 2.0f, -3.0f), new Point(5.0f, 5.0f, 0.0f));

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);
            GL.Disable(EnableCap.CullFace);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            bool[] expected = new bool[] { true, true, true, true,
false, false };
            for (int i = 0; i < aabbs.Length; ++i) {
                bool result = Collisions.Intersects(triangle, aabb
s[i]);
                if (result != expected[i]) {
                    LogError("Expected aabb " + i + " to " +
                        (expected[i] ? " intersect" : " NOT inters
ect") +
                        " the triangle");
                }
            }
        }
```

```
        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(0.0f, 0.0f, 1.0f);
            triangle.Render();

            foreach (AABB aabb in aabbs) {
                if (Collisions.Intersects(triangle, aabb)) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                aabb.Render();
            }
        }
    }
}
```

Plane Triangle

Plane triangle intersections are fairly simple. There are two use cases

1. Use the dot-product to determine whether the triangle lies fully on one side of the plane and does not intersect the plane at all.
2. If there is an intersection, use a line-plane-intersection-algorithm for the two edges hitting the plane (algorithm on the same page)

Thats it. It's a lot like AABB-V-Plane, where we just tested if every point of the AABB was on the same side of the plane.

The Algorithm

This one is fairly simple, i'm not going to provide an implementation for it. You need to test each point of the triangle against the plane, to see which side the points are on.

You can use the `DistanceFromPlane` with each point, it will return a negative number, zero or a positive number.

- If all 3 distances are 0 (That is all 3 points are on the plane), there is a collision.
- If all 3 distances have the same sign (positive or negative) then there is no collision
- Otherwise there is a collision
    - This happens when at least one of the points is on the other side of the plan

On Your Own

Add the following function to the `Collisions` class:

```
public static bool Intersects(Triangle triangle, Plane plane)

public static bool Intersects(Plane plane, Triangle triangle) {
    return Intersects(triangle, plane);
}
```

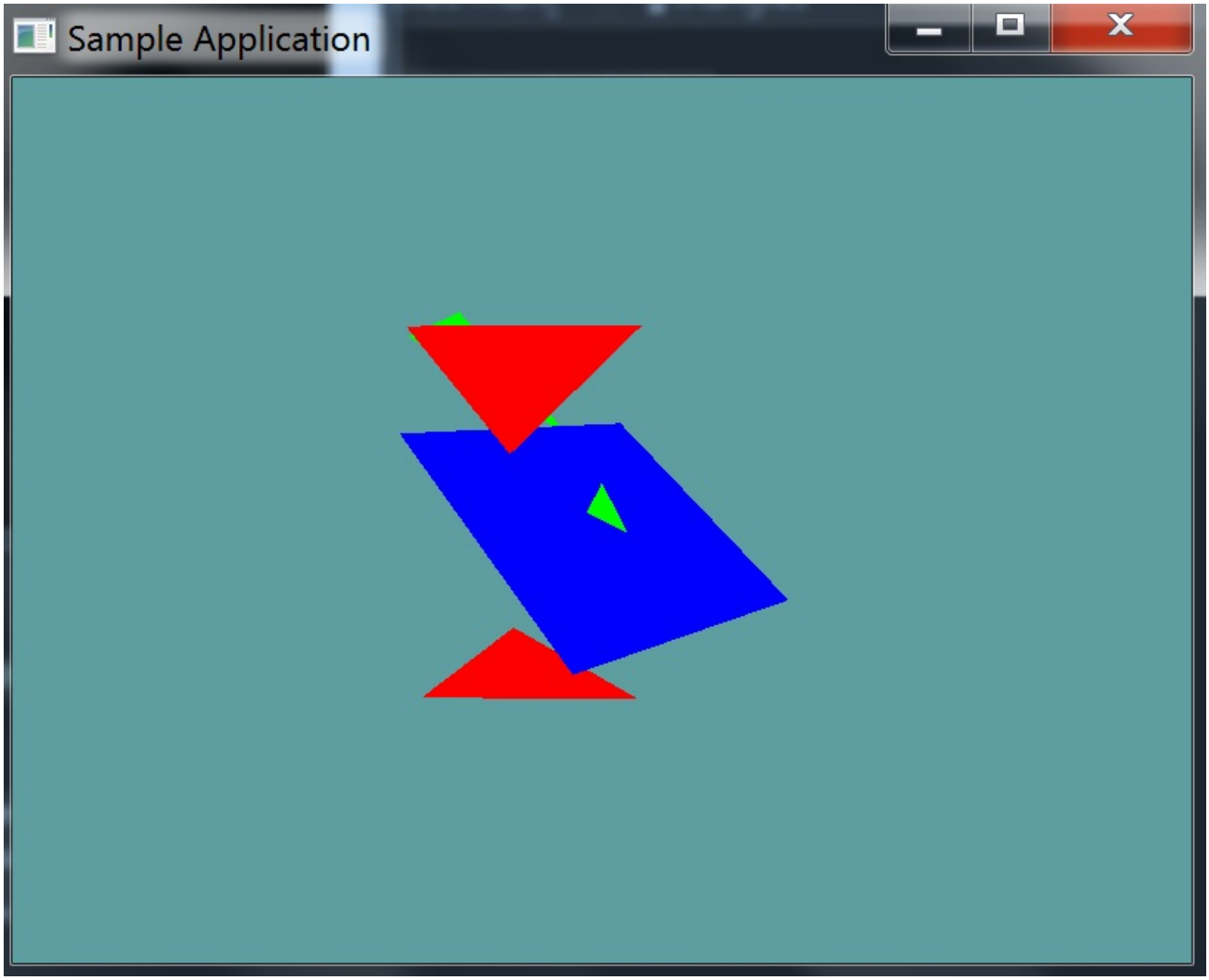
And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test draws a plane and some triangles. Triangles that intersect the plane are drawn in green. Triangles that don't are drawn in red.

The constructor will throw errors if any are present in your code.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class TrianglePlaneIntersection : Application {
        Plane plane = new Plane(new Point(5, 6, 7), new Point(6, 5
, 4), new Point(1, 2, 3));
        Triangle[] triangles = new Primitives.Triangle[] {
            new Triangle(new Point(-1.0f, 5.0f, 0.0f), new Point(2
.0f, 2.0f, -3.0f), new Point(5.0f, 5.0f, 0.0f)),
            new Triangle(new Point(-1, -1, 0), new Point(0, 1, 0),
new Point(1, -1, 0)),
            new Triangle(new Point(-1.0f, -5.0f, 0.0f), new Point(
```

```
2.0f, -2.0f, -3.0f), new Point(5.0f, -5.0f, 0.0f)),
            new Triangle(new Point(5, 6, 7), new Point(6, 5, 4), n
ew Point(1, 2, 3)),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);
            GL.Disable(EnableCap.CullFace);

            bool[] expected = new bool[] { false, true, false, true
 };
            for (int i = 0; i < triangles.Length; ++i) {
                bool result = Collisions.Intersects(plane, triangl
es[i]);
                if (result != expected[i]) {
                    LogError("Expected triangle " + i + " to " +
                        (expected[i] ? " intersect" : " NOT inters
ect") +
                        " the plane");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(0.0f, 0.0f, 1.0f);
            plane.Render(4);

            foreach(Triangle triangle in triangles) {
                if (Collisions.Intersects(triangle, plane)) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
```

```
                    }
                    triangle.Render();
                }
            }
        }
    }
```

Generic SAT Test

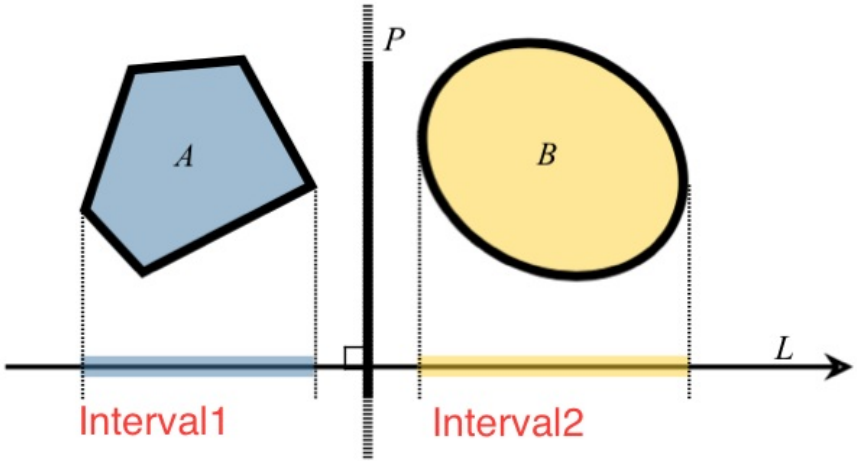
I originally wanted to leave this section until the end of the book, but triangle-triangle uses the most generic form of an SAT test, so this is a good place. There is no code to do on this page, it's purley a concept page!

Triangle AABB used an optimized version of the SAT, that was optimized for those primitives. But, there is a way to generalize the SAT test so that it will work with ANY primitive! See This presentation for details.

By now you should know what the SAT test does, so i'm going to skip the details of how a SAT test works.

Getting the interval

Projecting any arbitrary shape onto a plane results in a line segment on that plane. This line segment is what we call the interval.



```
class Interval {
    float min;
    float max;
}
```

We don't actually need to store an X / Y coordinate for the edges of the intervals. Because both intervals are on the same line, we just have to store the time t at which the interval starts and stops on the line.

It's possible to get the interval of any mesh (or primitive shape) if you know it's vertices. Getting an interval involves projecting each vector onto the given axis and storing the minimum and maximum values. The following code will do that.

```
Interval GetInterval(BasicShape shape, Axis axis) {
    // This assumes that the axis is normalized!

    Interval result = new Interval();
    result.min = Dot(axis, shape.GetVertex(0));
    result.max = result.min;

    for (int i = 1; i < shape.VertexCount(); ++i) {
      float value = Dot(axis, shape.GetVertex(i));
      result.min = Min(result.min, value);
      result.max = Max(result.max, value);
    }

    return result;
}
```

Comparing Intervals

Two objects intersect if there is no seperating axis between them. We have to test each axis. To test an axis you get the interval of the object on both axis, and check for overlap.

Once you have a way to get the interval of an object, you can perform an axis test like so:

```
bool TestAxis(BasicShape shape1, BasicShape shape2, Vector3 axis)
{
    // This assumes that axis s normalized!
    Interval i1 = GetInterval(shape1, axis);
    Interval i2 = GetInterval(shape2, axis);

    if (i1.max < i2.min || i2.max < i1.min) {
        // The intervals overlap on the given axis
        return true;
    }

    return false; // No collision found!
}
```

Test Axis

We now know how to get the interval of any arbitrary shape. But how do we know what axis to test? This comes from a math theory known as the `Minkowski` difference.

The theory states that to test intersection we need to test the face directions of object a, the face directions of object b and all of the edge directions of the two objects relative to each other.

We've done this with the AABB-Triangle test, where we used the 3 normals of the AABB, the 1 normal of the triangle and the cross products of all 3 faces of the triangle against the 3 faces of the aabb (9), for a total of 13 seperating axis.

It's worth noting, the AABB test is specialized by moving the triangle so that the AABB is at relative origin. Normal SAT tests don't do that. If we followed the generic outline given here, it would test 6 faces of the cube, bringing the total axis up to 25!

In any case, the following table should give some indication as to the complexity of the SAT test for various primitives

Object A	Object B	Face Dirs (A)	Face Dirs (B)	Edge Dirs (A x B)	Total
Segment	Triangle	0	0	1x3	4
Segment	OBB	0	3	1x3	6
AABB	AABB	3	0(3)	0(3x0)	3
OBB	OBB	3	3	3x3	15
Triangle	Triangle	1	1	3x3	11
Triangle	Obb	1	3	3x3	13

This test assumes that each object knows enough about its-self to know it's own face directions and edge directions. Lets see a basic example of thos. Given a basic triangle with points `p0`, `p1` and `p2` you first want to find the edges of the triangle:

```
Vector3 edge1 = p1 - p0; // B - A
Vector3 edge2 = p2 - p1; // C - B
Vector3 edge3 = p0 - p2; // A - C
```

These are the edges you will cross with the edges of the other primitive to get the bulk of the axis to test. Once you know the edges of a triangle, it's easy to find its face.

```
Vector3 faceNormal = Cross(edge1, edge2);
```

It's the responsibility of the object to know that information about its-self. An AABB for example will need to know that it has 3 face normals and 3 edges.

So, assuming that we know the edges and normals of an object, the most generic SAT algorithm that compares them all is:

```
bool ObjectsIntersect(BasicShape shape1, BasicShape shape2) {
    return !HasSeperatingAxis(shape1, shape2);
}

// This function returns true if A seperating axis is found
// NOT if the objects overlap
bool HasSeperatingAxis(BasicShape shape1, BasicShape shape2) {
    // First, test the face normals of object 1 as the seperating
axis
    for (int i = 0; i < shape1.FaceNormalCount(); ++i) {
        Vector3 testAxis = shape1.GetFaceNormal(i);

        if (TestAxis(shape1, shape2, testAxis)) {
            // Seperating axis found, early out
            return true;
        }
    }

    // Then, test the face normals of object 2 as the seperating a
xis
    for (int i = 0; i < shape2.FaceNormalCount(); ++i) {
        Vector3 testAxis = shape2.GetFaceNormal(i);

        if (TestAxis(shape1, shape2, testAxis)) {
            // Seperating axis found, early out
            return true;
        }
    }

    // Finally, check the normals obtained by getting the cross pr
oduct of each shapes edges.
    for (int i = 0; i < shape1.EdgeCount(); ++i) {
        for (int j = 0; j < shape2.EdgeCount(); ++j) {
            Vector3 testAxis = Cross(shape1.GetEdge(i), shape2.Get
Edge(j));

            if (TestAxis(shape1, shape2, testAxis)) {
```

```
                // Seperating axis found, early out
                return true;
            }
        }
    }

    // No seperating axis found, the objects do not intersect
    return false;
}
```

Fin

That's it. The above test can be used to test the intersection between ANY primitives. This of course assumes that all primitives have a base class called `BasicShape` which has the following virtual functions that each class overrides

```
class BasicShape {
    virtual int VertexCount()
    virtual Vector3 GetVertex(int i)

    virtual int FaceNormalCount()
    virtual Vector3 GetFaceNormal(int i)

    virtual int EdgeCount()
    virtual Vector3 GetEdge()
}
```

Of course, this is not how we implemented our collision system. The specific tests we do are much faster than the generic form SAT. But, we will implement the generic form SAT for triangle-triangle collision next, so i wanted to make sure you understood how it is implemented.

Triangle Triangle

As mentioned before, the triangle-triangle test is a generic form SAT test. As such we will need a few helper functions to get the intervals of the triangle and do the axis testing.

On Your Own

Add the following private static function to your collision class, it's going to be a helper function to get the interval of the triangle. You can return the interval however you like (Make a new class, use a vector 2, use out variables), whatever makes sense to you. I did mine with a Vector2.

```
Vector2 GetInterval(Triangle triangle, Vector3 axis) {
    // Vector2.X = min
    // Vector2.Y = max.
}
```

Next, we need to add another private static helper function, to test two triangles on an actual axis

```
bool TestAxis(Triangle triangle1, Triangle triangle2, Vector3 axis
) {
    // TODO: Test the intervals for overlap
}
```

Finally, add the public static function that tests 2 triangles for intersections, and provide an implementation for it!

You will need to test 11 axis, the face normal of triangle1, the face normal of triangle2, and the cross products of each of the 3 edges of the triangles. The previous section "Generic SAT" outlined how to get this information from a triangle.

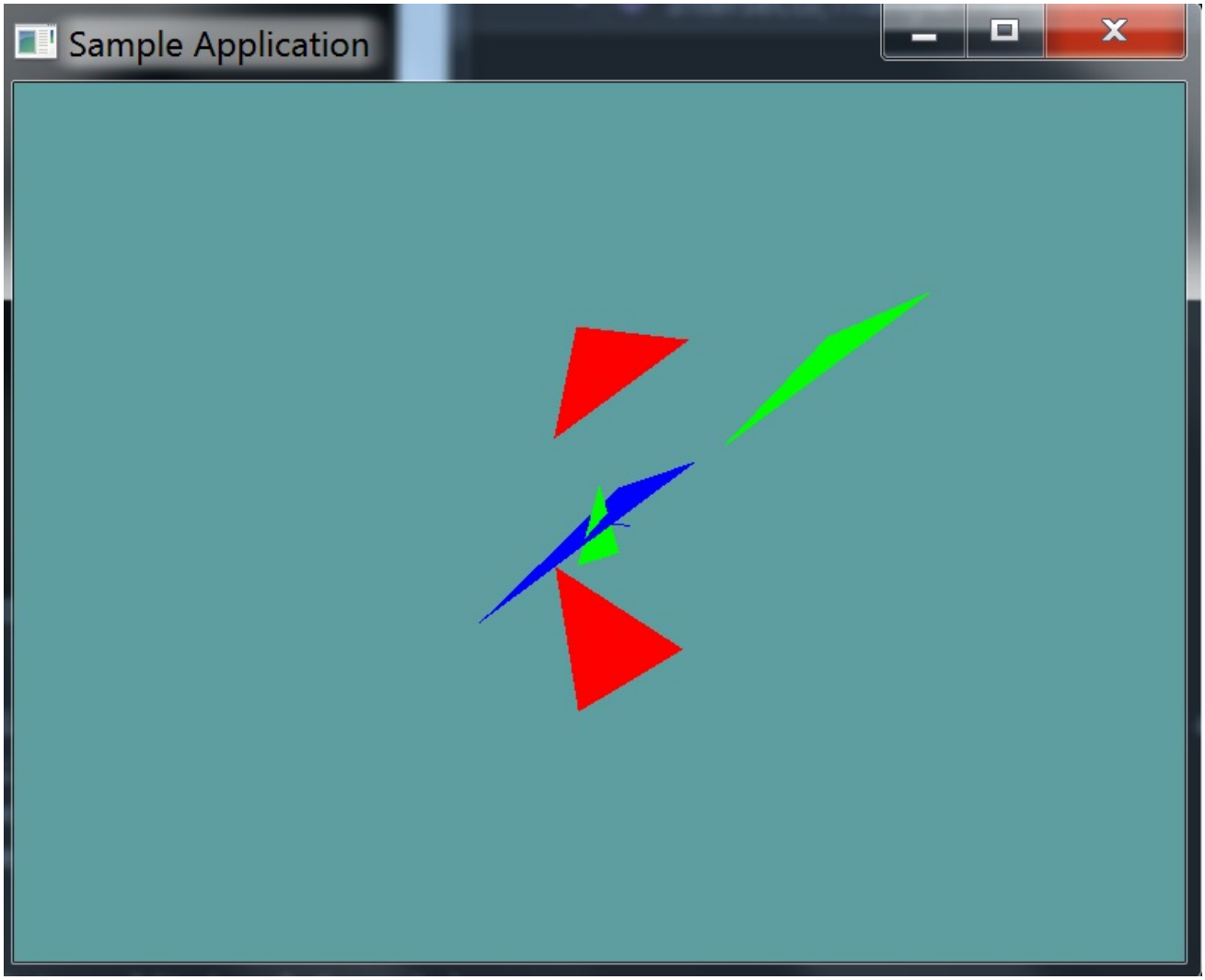
```
public static bool Intersects(Triangle triangle1, Triangle triangl
e2)
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test will check a few triangles against the blue one. Any triangle that intersects is green, ones that don't are red. The constructor does NOT throw any errors, this unit is visual only.

One thing you might notice is that one of the triangles gives us a false positive! This is becuase the SAT as we've implemented it is non-robust. Because both triangles are on the same plane, all of the cross products we do yield 0. And because they are all 0, they all intersect. We will fix this in the next section.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class TriangleTriangleIntersection : Application {
        Triangle test = new Triangle(new Point(5, 6, 7), new Point(
6, 5, 4), new Point(1, 2, 3));
        Triangle[] triangles = new Primitives.Triangle[] {
            new Triangle(new Point(-1.0f, 5.0f, 0.0f), new Point(2
.0f, 2.0f, -3.0f), new Point(5.0f, 5.0f, 0.0f)),
            new Triangle(new Point(-1, -1, 0), new Point(0, 1, 0),
new Point(1, -1, 0)),
            new Triangle(new Point(-1.0f, -5.0f, 0.0f), new Point(
```

```
2.0f, -2.0f, -3.0f), new Point(5.0f, -5.0f, 0.0f)),
            new Triangle(new Point(5, 6, 7), new Point(6, 5, 4), n
ew Point(1, 2, 3)),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(4f);
            GL.Disable(EnableCap.CullFace);

            Vector3 center = test.p0.ToVector() + test.p1.ToVector
() + test.p2.ToVector();
            center *= 1.0f / 3.0f;

            test.p0.X -= center.X;
            test.p0.Y -= center.Y;
            test.p0.Z -= center.Z;

            test.p1.X -= center.X;
            test.p1.Y -= center.Y;
            test.p1.Z -= center.Z;

            test.p2.X -= center.X;
            test.p2.Y -= center.Y;
            test.p2.Z -= center.Z;
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(0.0f, 0.0f, 1.0f);
            test.Render();

            foreach (Triangle triangle in triangles) {
                if (Collisions.Intersects(triangle, test)) {
                    GL.Color3(0f, 1f, 0f);
```

```
                    }
                    else {
                        GL.Color3(1f, 0f, 0f);
                    }
                    triangle.Render();
                }
            }
        }
    }
```

Robust SAT

The reason that our triangle - triangle collision has a false negative is because the two triangles are on the exact same plane. When this happens, the cross product of any of the vectors yields 0!

We can actually fix this issue. First, we have to know if the result of the cross product is 0. Remember, floating point errors tend to creep up, so we have to check against an epsilon. The easyest way to do this is to check the square magnitude of the vector

```
bool isZeroVector = crossVector.LengthSquared() < 0.0001f;
```

Constructing a new axis

The key here is, if we are given two parallel vectors, to create a new vector that is perpendicular to both. For this all we need is a new axis to test.

We have the function **TestAxis** defined as so

```
bool TestAxis(Triangle triangle1, Triangle triangle2, Vector3 axis
)
```

The key here is, instead of passing in the axis to test, we want to pass in the components that are used in the cross product to create the axis. The above would be rewritten like so

```
bool TestAxis(Triangle triangle1, Triangle triangle2, Vector3 A, V
ector3 B, Vector3 C, Vector3 D) {
    Vector3 axis = Vector3.Cross(A - B, C - D);

    if (axis.LengthSquared() < 0.0001f) {
        // Axis is zero, try other combination
        Vector3 n = Vector3.Cross(A - B, C - A);
        axis = Vector3.Cross(AB, n);

        if (axis.LengthSquared() < 0.0001f) {
            // Axis is still zero, not a seperating axis
            return false;
        }
    }

    // The rest of the function
}
```

With a triangle

We know a trianlge has the following properties:

```
Vector3 edge1 = p1 - p0; // Y - X
Vector3 edge2 = p2 - p1; // Z - Y
Vector3 edge3 = p0 - p2; // X - Z
Vector3 faceNormal = Cross(edge1, edge2); YX CROSS ZY
```

When you look at our triangle, everything is constructed out of 4 vectors. The face normal for example is constructed out of `YX CROSS ZY`, or `p1 - p0 CROSS p2 - p1`. So, for the axis that is the face normal, we would call the `TestAxis` method like so:

```
if (TestAxis(triangle1, triangle2, p1, p0, p2, p1)
```

Similarly, to construct the test axis out of the edges of the triangle, at some point the axis is found like this:

```
Vector3 axis = Cross(triangle1.edge1, triangle2.edge1)
```

This is repeated 9 times in all edge combinations. The above edge would be passed into the `TestAxis` function like this:

```
if (TestAxis(triangle1, triangle2, triangle1.p1, triangle1.p0, tri
angle2.p1, triangle2.p0)
```

Take note, all we're doing here is substituting.

This does mean that you can no longer hold a simple array of vectors called the testAxis and loop trough them. You now have to hold an array of 4 vectors, that was used to previously build the testAxis.

Implement this however you want, you can write it all out, or try to make a clever loop. Once you adjust your code to be a robust SAT test, the unit test for same plane triangles that has been failing until now will start to work.

Raycast Triangle

Raycsating against a triangle is a simple process

- Create a plane from the 3 points of the triangle
- Raycast against that plane
- Check if the raycast point is inside the triangle

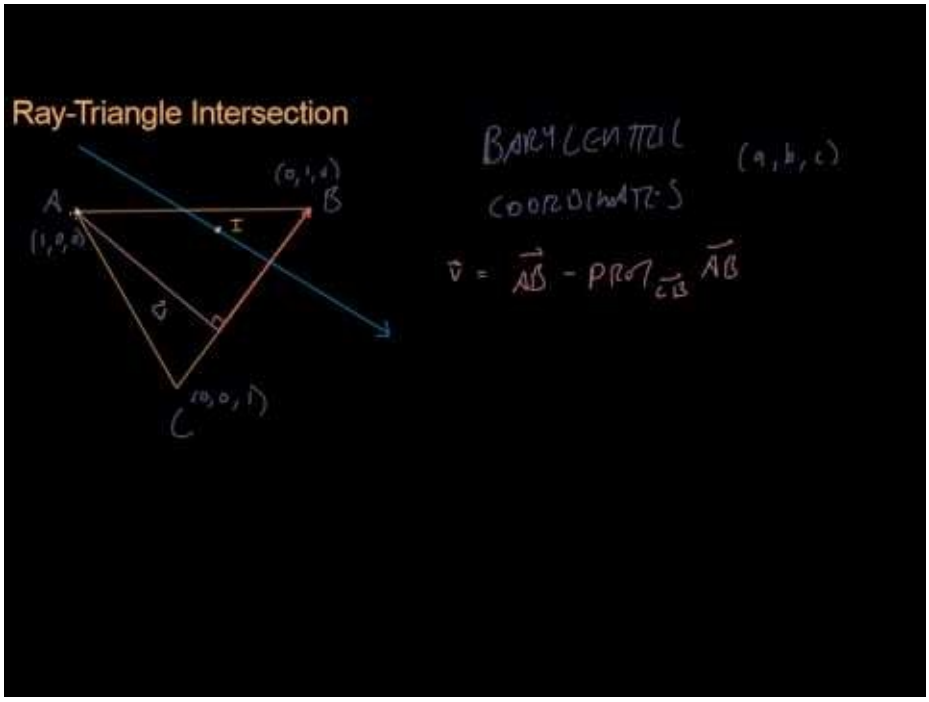
You already have all the fnuctions needed to implement this in a rudimentary way. However, our Point In Triangle test is not optimal! Also, we will have to make a special version of raycast plane!

Barycentric Point In Tirangle

Remember, when we did the point in triangle test, we did not use barycentric coordinates.

It's time to switch that. You can either implement barycentric coordinates straight up in the raycast triangle code, or you can change your point in triangle code and call that. If you change the point in triangle equation, make sure to run the unit test again to ensure everything still works.

Onto implementing barycentric coordinates! Watch this video:



Video link

Special Plane Raycast

Next up, the raycast plane function we have will not work. This is because that function takes the planes normal into consideration. I suggest making a new private function:

```
private static bool RaycastNoNormal(Ray ray, Plane plane, out float
 t) {
```

Inside the plane raycast code, we do a dot product of plane normal and ray normal, store the result in nd. We later test if the normals are pointing in the same direction by doing this bit:

```
if (nd >= 0f) {
    t = -1;
    return false;
}
```

Thats where the problem is, the only time we want the ray and the plane to not intercept is when they are parallel, that is when nd is 0. Remember, use an epsilon. This will take the direction of the plane and ray needing to be the same out of the equation.

On Your Own

Add the following function to the `Collisions` class:

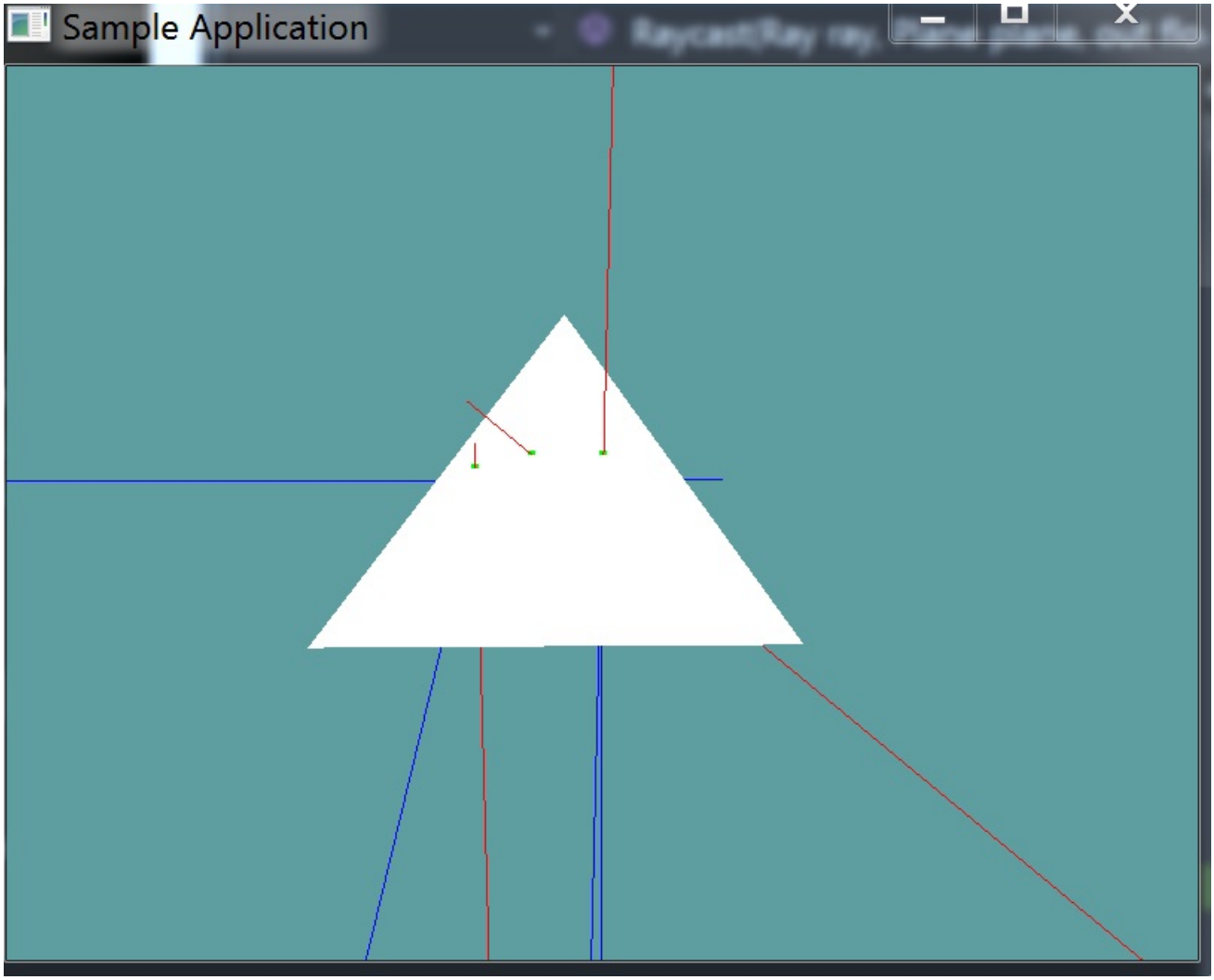
```
public static bool Raycast(Ray ray, Triangle triangle, out float t
) {
    // Implement your code here
}

// Conveniance method, returns t without an out param
// If no collision happened, will return -1
public static float Raycast(Ray ray, Triangle triangle) {
    float t = -1;
    if (!Raycast(ray, triangle, out t)) {
        return -1;
    }
    return t;
}

// Conveniance method, returns the point of intersection
public static bool Raycast(Ray ray, Triangle triangle, out Point p
) {
    float t = -1;
    bool result = Raycast(ray, triangle, out t);
    p = new Point(ray.Position.ToVector() + ray.Normal * t);
    return result;
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

A trianlge and a few rays are rendered. Any ray intersecting the triangle is red. All rays not intersecting the triangle are blue. Green dots are rendered at the points of intersection. The constructor of this unit test will spit out errors if it finds any




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
```

```
    class RaycastTriangle : Application {
        Triangle test = new Triangle(   new Point(5.207108f, -1.79
2894f, -3.949748f),
                                        new Point(5.207108f, -1.79
2894f, 5.949748f),
                                        new Point(-1.792894f, 5.20
7108f, 1));



        public Ray[] rays = new Ray[] {
            new Ray(new Point(0f, 0f, 0f), new Vector3(0f, -1f, 0f
)),
            new Ray(new Point(0.5f, 0.5f, 0f), new Vector3(-1f, -1
f, 0f)),
            new Ray(new Point(1f, 1f, 0f), new Vector3(1f, 1f, 0f)
),
            new Ray(new Point(1f, 1f, -3f), new Vector3(0f, 0f, 1f
)),
            new Ray(new Point(2f, 2f, 3f), new Vector3(0f, -1f, 0f
)),
            new Ray(new Point(3f, 3f, 3f), new Vector3(-3f, -3f, -3
f)),
            new Ray(new Point(1f, 1f, 3f), new Vector3(-2f, -3f, 1
f)),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Fill);

            bool[] results = new bool[] {
                false, false, true, false, true, true, false
            };
```

```
            float t;
            for (int i = 0; i < results.Length; ++i) {
                if (Collisions.Raycast(rays[i], test, out t) != re
sults[i]) {
                    LogError("Expected ray at index: " + i + " to "
 +
                        (results[i] ? "intersect" : "not intersect"
) +
                        " the triangle");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Color3(1f, 1f, 1f);
            test.Render();


            float t;
            foreach (Ray ray in rays) {
                if (Collisions.Raycast(ray, test, out t)) {
                    Point colPoint = new Point();
                    Collisions.Raycast(ray, test, out colPoint);
                    GL.Color3(0f, 1f, 0f);
                    colPoint.Render();
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 0f, 1f);
                }
                ray.Render();
            }
        }
```

```
        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Line Testing

Line testing is just a specialized form of ray casting. This test is done the same way all the other line tests are done, i'm not going to provide too much information on it.

On Your Own

Implement the following function

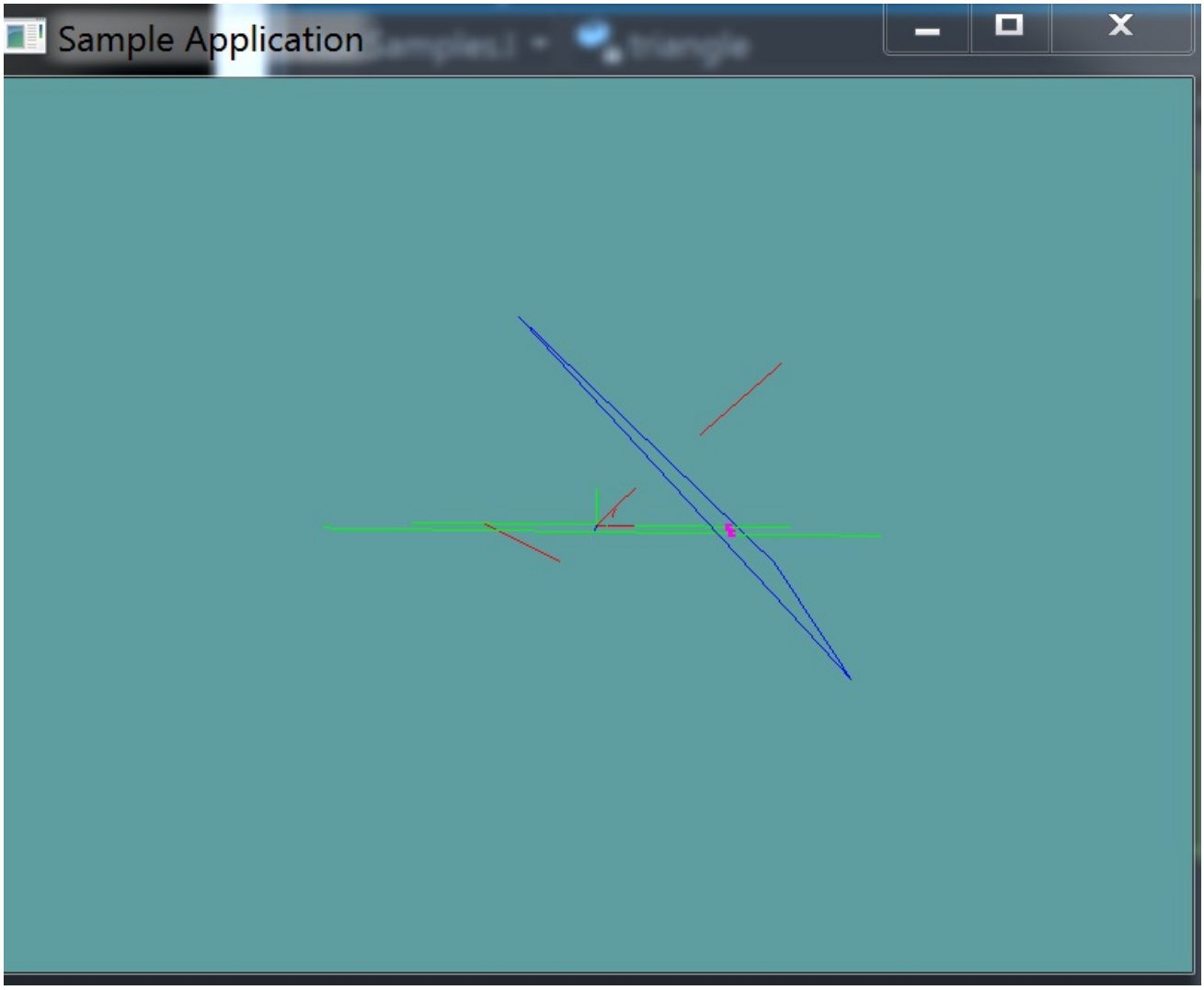
```
public static bool LineTest(Line line, Triangle triangle, out Poin
t result) {
```

It's just like all theother line test functions, i'm not going to go over it in detail.

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

A trianlge and a few lines are rendered. Any line intersecting the triangle is green. All lines not intersecting the triangle are red. Magenta dots are rendered at the points of intersection. The constructor of this unit test will spit out errors if it finds any




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector.Samples {
    class LinetestTriangle : Application {
        public Line[] lines = new Line[] {
            new Line(new Point(2.5f, 2.5f, 2.5f), new Point(4, 4, 4
)),
            new Line(new Point(1f, 1f, 0f), new Point(0f, 0f, 0f))
,
            new Line(new Point(-1f, -1f, 0f), new Point(-3f, 0f, 0
f)),
            new Line(new Point(-5f, 0f, 0f), new Point(5f, 0f, 0f)
```

```
),
            new Line(new Point(7f, 0f, 1f), new Point(-7f, 0f, 1f)
),
            new Line(new Point(0.5f, 0.5f, 0.5f), new Point(0.5f,
0.5f, 2f))
        };
        Triangle triangle = new Triangle(new Point(5.207108f, -1.7
92894f, -3.949748f),
                                         new Point(5.207108f, -1.79
2894f, 5.949748f),
                                         new Point(-1.792894f, 5.20
7108f, 1));


        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.PointSize(5f);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            bool[] results = new bool[] { false, false, false, true
, true, false };
            Point result = new Point();
            for (int i = 0; i < results.Length; ++i) {
                if (Collisions.LineTest(lines[i], triangle, out re
sult) != results[i]) {
                    LogError("Line at index " + i + " was " +
                        (results[i] ? "expected" : "not expected")
 +
                        "to intersect the test triangle");
                }
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();
```

```
            Point result = new Point();
            foreach (Line line in lines) {
                if (Collisions.LineTest(line, triangle, out result
)) {
                    GL.Color3(1f, 0f, 1f);
                    result.Render();
                    GL.Color3(0f, 2f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                line.Render();
            }

            GL.Color3(0f, 0f, 1f);
            triangle.Render();
        }

        private void Log(string s) {
            System.Console.WriteLine(s);
        }
    }
}
```

Containers

You've seen exactly how expensive triangle checks can be! In order to test for collision against a model we have to check every single triangle of the model against the primitive! In pseudo-code that might look something like this:

```
bool Intersects(OBJModel model, Sphere sphere) {
    foreach(Triangle triangle in model.triangles) {
        if (Intersects(triangle, sphere)) {
            return true;
        }
    }
    return false;
}
```

It's so much worse if you try to do 2 models! You have to check each triangle against all triangles in the other model. That means that 2 models with 100 triangles will need to do 10,000 tests! That's 110,000 Seperating Axis!

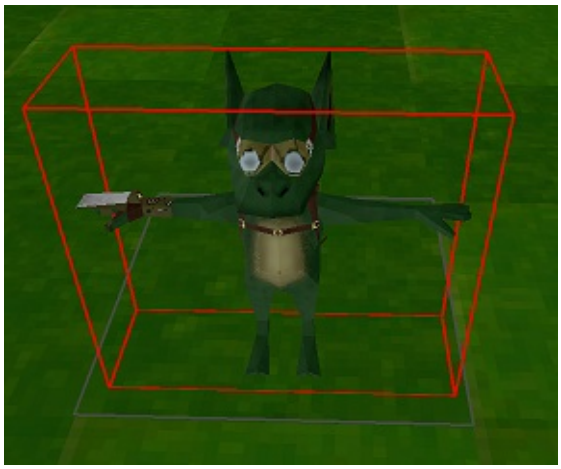
```
bool Intersects(ObjModel m1, ObjModel m2) {
    for (int i = 0; i < m1.triangles.Length; ++i) {
        for (int j = 0; j < m2.triangles.Length; ++j) {
            if (Intersects(m1.triangles[i], m2.triangles[j])) {
                return true;
            }
        }
    }
    return false;
}
```

In a model that consists of 100 triangles that might not be a big deal. But in modern games, the triangle count is often in the hundreds of thousands if not millions!

Doing the per triangle testing is known as **narrow-phase collision**.

Containers solve this problem by adding a bounding shape to the model. That way, when you test collisions, if the bounding shape of the model doesn't interact with the test, you don't waste precious resources.

An AABB around a model might look like this:

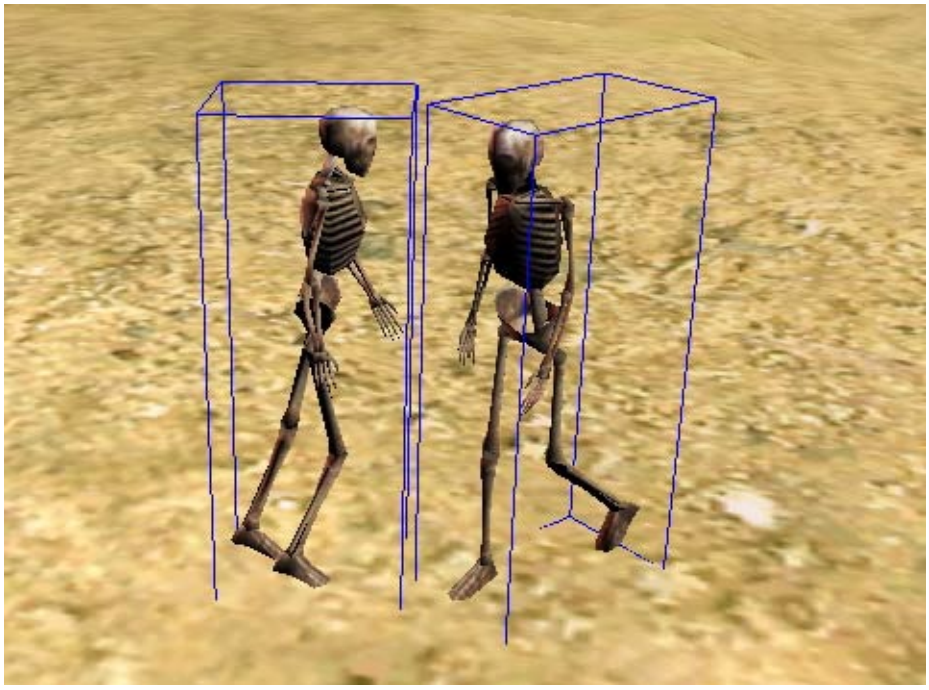


And the collision test could simplify into this:

```
bool Intersects(ObjModel m1, ObjModel m2) {
    // Early out if not touching
    if (!Intersects(m1.AABBContainer, m2.AABBContainer)) {
        return false;
    }

    for (int i = 0; i < m1.triangles.Length; ++i) {
        for (int j = 0; j < m2.triangles.Length; ++j) {
            if (Intersects(m1.triangles[i], m2.triangles[j])) {
                return true;
            }
        }
    }
    return false;
}
```

Adding a bounding volume to an object, and testing that before the actual triangle tests is known as **broad-phase collision**

Visually, that might look like this



That optimization will save so much processing power! With all that said, let's get started. In the next section we're going to add triangles to our loaded OBJ model.

Adding Triangles

It's time to start implementing some broad phase collision. The first thing we're going to do is add a list of triangles to our 3D model. This is going to be done at load time, as the models vertex array is filled in, we will also fill in a list of triangles.

We're also going to add a "DebugRender" function, that will loop trough all the triangles and render them. Because this is going to render each triangle indevidually, it's going to be much, much slower than the regular render function.

Constructing Triangles

First thing is first, add a triangle array as a member variable to the `ObjLoader` class. I called mine `collisionMesh`

```
protected Triangle[] collisionMesh = null;
```

Now, in the actual constructor, we have to fill this array in. The constructor has a local array, called `vertexData`, it's a bunch of floats. Once that array is filled in, it contains sequential triangles.

That is, the first triangle of the model is made out of the first 9 elements of the array, like so:

- new Triangle(
  - new Point(`vertexData[0]` , `vertexData[1]` , `vertexData[2]`),
  - new Point(`vertexData[3]` , `vertexData[4]` , `vertexData[5]`),
  - new Point(`vertexData[6]` , `vertexData[7]` , `vertexData[8]`),
-)

That means that you have as many triangles as the size of that array, divided by 9. In the case of the suzanne model, that's 968. You can allocate the array like so:

```
collisionMesh = new Triangle[vertexData.Count / 9];
```

On Your Own

Figure out a way to loop trough vertex data in a way that fills the `collisionMesh` array with proper data. Once you think you have that done, implement the debug renderer from below and run the unit test.

Debug Render

First, let's add a debug getter to check how many debug triangles we have

```
public int NumCollisionTriangles {
    get {
        return collisionMesh.Length;
    }
}
```

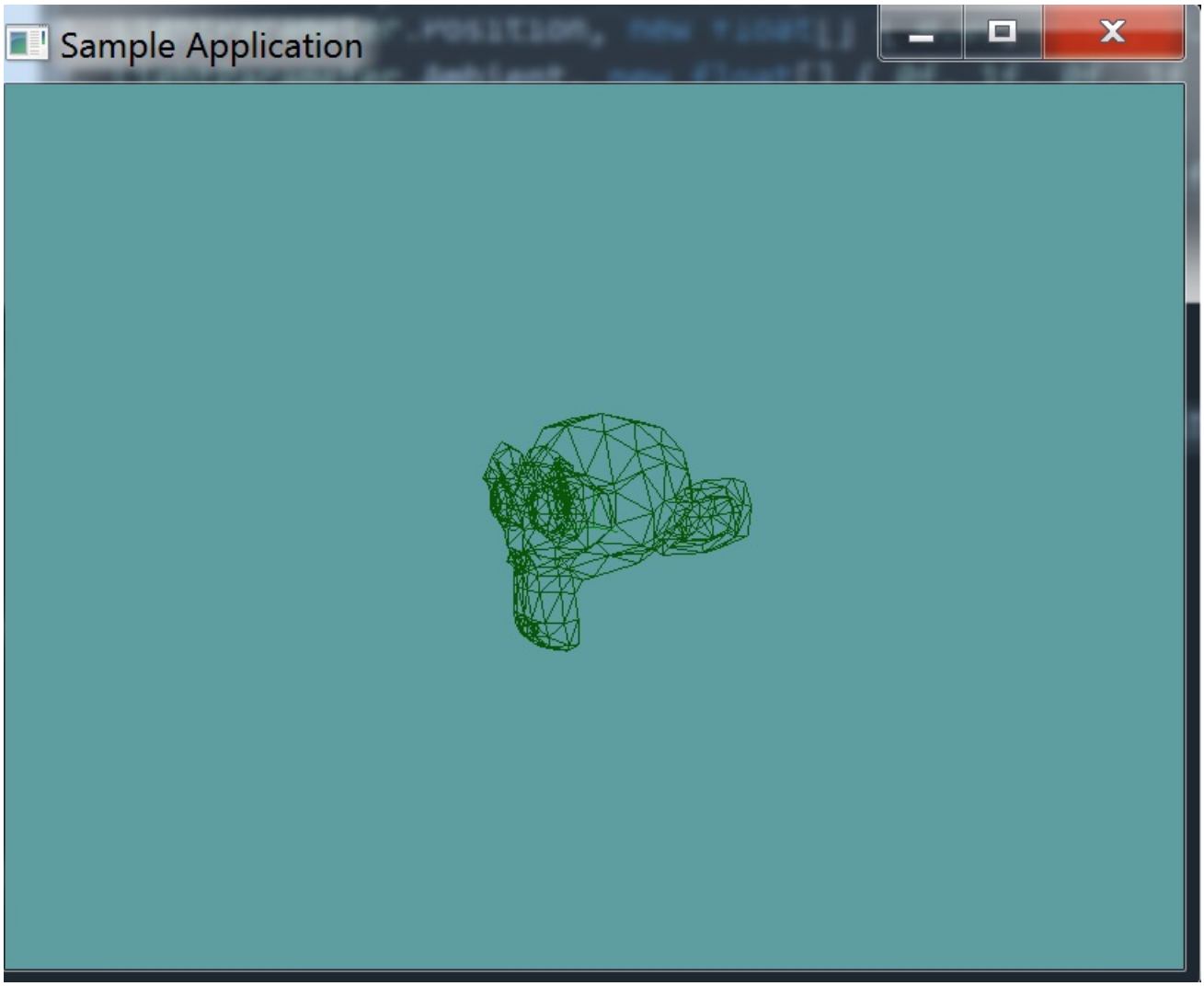
Next, a "debug render" method, this method will simply draw out all triangles:

```
 public void DebugRender() {
    foreach(Triangle trianlge in collisionMesh) {
        trianlge.Render();
    }
}
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

You should already have suzanne in your project from earlyer in this chapter. This section just added triangles to the OBJLoader code. The constructor only errors out if the triangle count is off. Otherwise the rest of the test is visual only. You should see susane in the test




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJDebugTriangles : Application {
        OBJLoader obj = null;
```

```
        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            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 });

            obj = new OBJLoader("Assets/suzanne.obj");

            if (obj.NumCollisionTriangles != 968) {
                LogError("Suzanne triangle count expected at 968,
got: " + obj.NumCollisionTriangles);
            }
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.PushMatrix();
            GL.Scale(3.0f, 3.0f, 3.0f);
            obj.DebugRender();
            GL.PopMatrix();
        }
    }
}
```

AABB

Our model now has a triangle soup. We can raycast against this, or do collision test, whatever works. But, like we discussed that's really not optimal! so, instead let's make a function that creates an Axis Aligned Bounding Box around the model.

Creating the AABB

We already have an array of triangles added to `OBJLoader.cs`, now lets add a new member variable to it for the AABB of the mesh.

```
protected AABB containerAABB = null;
```

We're going to have to populate this AABB in the constructor of the mesh. Again, the `vertexData` array is our friend.

First, assign `containerAABB` to be a new AABB object. Both the min and max point of this object should be the first vertex of `vertexData`. Remember, the first vertex is elements 0, 1 and 2.

```
containerAABB = new AABB(new Point(vertices[0], vertices[1], verti
ces[2]),
                new Point(vertices[0], vertices[1], vertices[2]));
```

Now loop trough vertexData one vertex at a time. You will have to adjust containerAABB one component at a time. If a vertex has a smaller y than min-y assign a new y. For eaxample:

```
// ...
if (vertexData[i + 0] < containerAABB.minX) {
    containerAABB.minX = vertexData[i + 0];
}
if (vertexData[i + 0] > containerAABB.maxX) {
    containerAABB.maxX = vertexData[i + 0];
}
if (vertexData[i + 1] < containerAABB.minY) {
    containerAABB.minY = vertexData[i + 0];
}
if (vertexData[i + 1] > containerAABB.maxY) {
    containerAABB.maxY = vertexData[i + 1];
}
if (vertexData[i + 2] < containerAABB.minZ) {
    containerAABB.minZ = vertexData[i + 0];
}
if (vertexData[i + 2] > containerAABB.maxZ) {
    containerAABB.maxZ = vertexData[i + 2];
}
// ...
```

Debug Render

Now, let's add a public getter for the aabb:

```
public AABB BoundingBox {
    get {
        return containerAABB;
    }
}
```

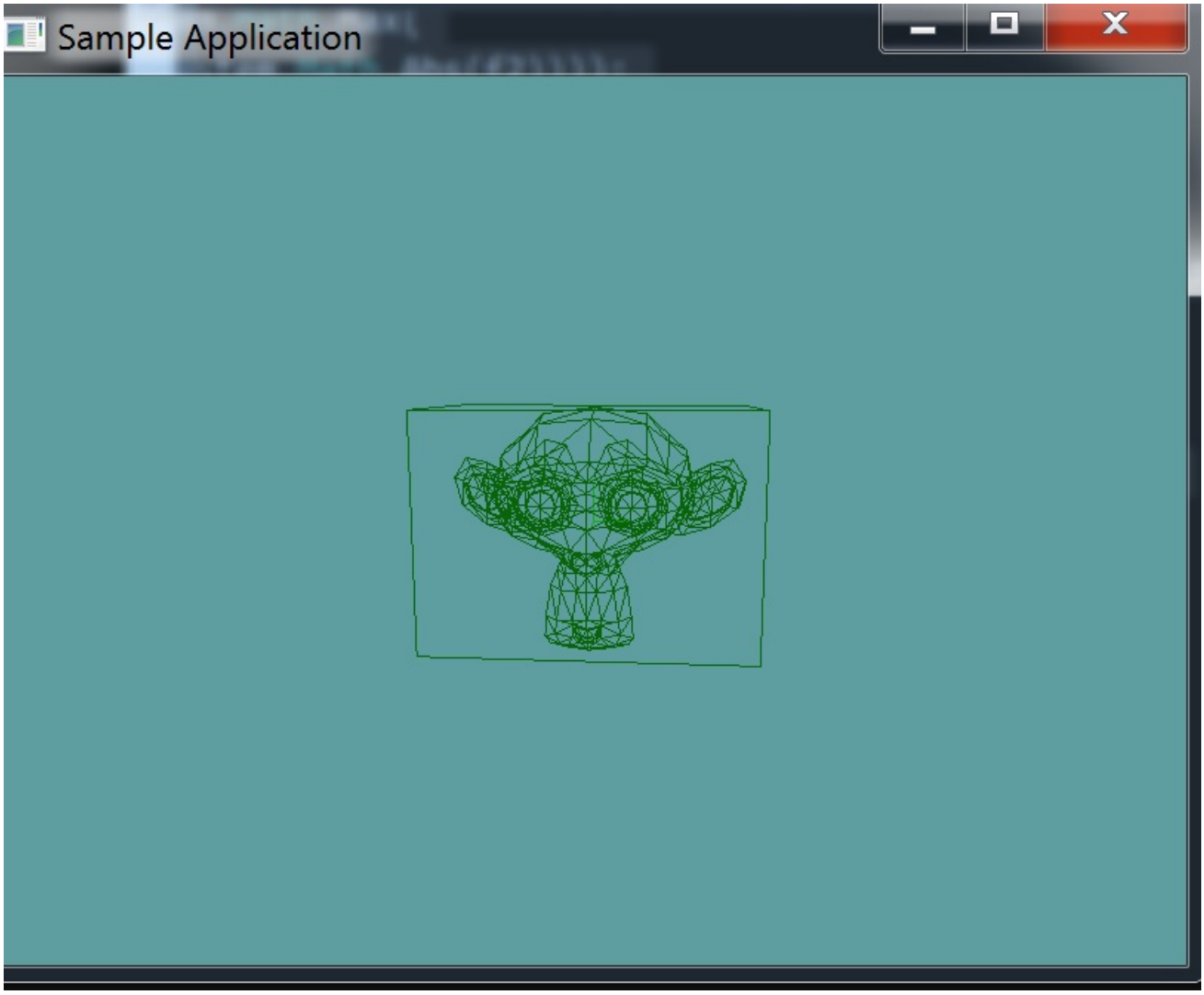
And modify the debug-render method to draw this box:

```
public void DebugRender() {
    containerAABB.Render();
    foreach (Triangle trianlge in collisionMesh) {
        trianlge.Render();
    }
}
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The constructor only errors out if the AABB or triangle count is off. Otherwise the rest of the test is visual only. You should see susane in the test




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJDebugTrianglesAABB : Application {
        OBJLoader obj = null;

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);
```

```
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            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 });

            obj = new OBJLoader("Assets/suzanne.obj");

            if (obj.NumCollisionTriangles != 968) {
                LogError("Suzanne triangle count expected at 968,
got: " + obj.NumCollisionTriangles);
            }
            AABB test = new AABB(new Point(-1.367188f, -0.984375f,
-0.851562f), new Point(1.367188f, 0.984375f, 0.851562f));
            if (!AlmostEqual(test, obj.BoundingBox)) {
                LogError("Expected bounting box:\n" + test.ToStrin
g() + "\n Got bounding box: \n" + obj.BoundingBox.ToString());
            }
        }

        private bool AlmostEqual(float f1, float f2) {
            return (System.Math.Abs(f1 - f2) <= float.Epsilon *
                System.Math.Max(1.0f, System.Math.Max(
                    System.Math.Abs(f1), System.Math.Abs(f2))));
        }

        private bool AlmostEqual(Point p1, Point p2) {
            return AlmostEqual(p1.X, p2.X) && AlmostEqual(p1.Y, p2
.Y) && AlmostEqual(p1.Z, p2.Z);
        }
```

```
        private bool AlmostEqual(AABB aabb1, AABB aabb2) {
            return AlmostEqual(aabb1.Min, aabb2.Min) && AlmostEqua
l(aabb1.Max, aabb2.Max);
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.PushMatrix();
            GL.Scale(3.0f, 3.0f, 3.0f);
            obj.DebugRender();
            GL.PopMatrix();
        }
    }
}
```

Sphere

In the last section we built an AABB around out model, this actually gives us everything we need to build a bounding sphere.

The center of the sphere is the same as the center of the AABB.

The radius of the sphere is either the distance between the center of the sphere and the min point or the max point. Take the length of both, and use the larger one.

Creating the sphere

Add a new sphere in the ObjLoader class:

```
protected Sphere containerSphere = null;
```

Now, in the constructor, after you have created the AABB, make the sphere object. Follow the instructions in the introduction paragraph.

Debug Render

Now, let's add a public getter for the sphere:

```
public Sphere BoundingSphere {
    get {
        return containerSphere;
    }
}
```

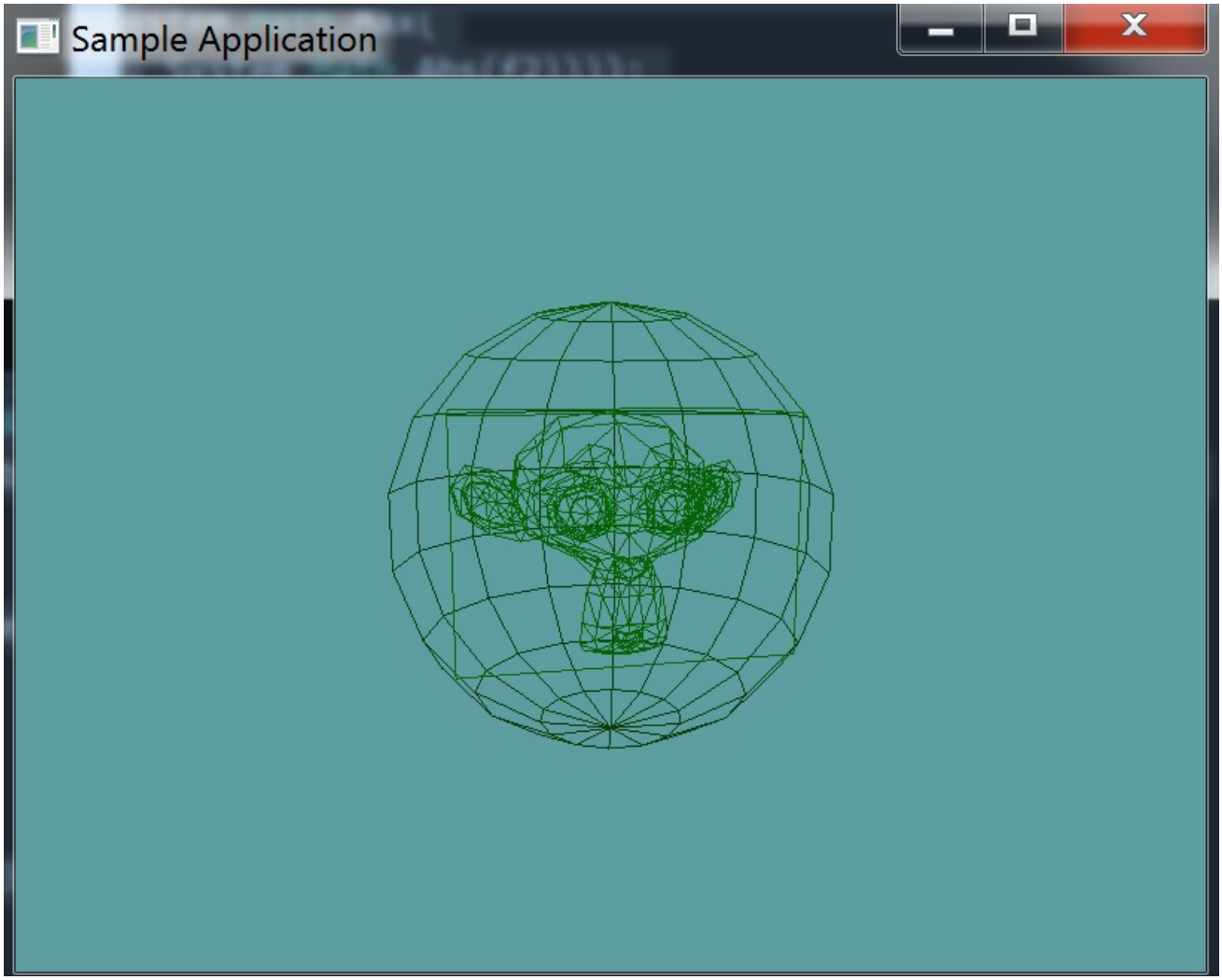
And modify the debug-render method to draw this box:

```
public void DebugRender() {
    containerAABB.Render();
    containerSphere.Render();
    foreach (Triangle trianlge in collisionMesh) {
        trianlge.Render();
    }
}
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

The constructor only errors out if the bounding sphere is off. Otherwise the rest of the test is visual only. You should see susane in the test




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJDebugTrianglesSphere : Application {
        OBJLoader obj = null;

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);
```

```
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            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 });

            obj = new OBJLoader("Assets/suzanne.obj");

            if (obj.NumCollisionTriangles != 968) {
                LogError("Suzanne triangle count expected at 968,
got: " + obj.NumCollisionTriangles);
            }
            AABB test = new AABB(new Point(-1.367188f, -0.984375f,
-0.851562f), new Point(1.367188f, 0.984375f, 0.851562f));
            if (!AlmostEqual(test, obj.BoundingBox)) {
                LogError("Expected bounting box:\n" + test.ToStrin
g() + "\n Got bounding box: \n" + obj.BoundingBox.ToString());
            }

            if (!AlmostEqual(obj.BoundingSphere.Radius, 1.887685f)
) {
                LogError("Expected bounding sphere radius to be: 1
.887685f, got: " + obj.BoundingSphere.Radius.ToString());
            }

            if (!AlmostEqual(obj.BoundingSphere.Position, new Poin
t(0f, 0f, 0f))) {
                LogError("Expected bounding sphere radius to be: (
0, 0, 0), got: " + obj.BoundingSphere.Position.ToString());
            }
        }
```

```
        private bool AlmostEqual(float f1, float f2) {
            return (System.Math.Abs(f1 - f2) <= 0.00001f *
                System.Math.Max(1.0f, System.Math.Max(
                    System.Math.Abs(f1), System.Math.Abs(f2))));
        }

        private bool AlmostEqual(Point p1, Point p2) {
            return AlmostEqual(p1.X, p2.X) && AlmostEqual(p1.Y, p2
.Y) && AlmostEqual(p1.Z, p2.Z);
        }

        private bool AlmostEqual(AABB aabb1, AABB aabb2) {
            return AlmostEqual(aabb1.Min, aabb2.Min) && AlmostEqua
l(aabb1.Max, aabb2.Max);
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.PushMatrix();
            GL.Scale(3.0f, 3.0f, 3.0f);
            obj.DebugRender();
            GL.PopMatrix();
        }
    }
}
```

Collision testing

In the last section of this document we added collision triangles to a model, a bounding box and a bounding sphere. Hopefully the debug render function we added will show the accuracy of the collision testing we do against the model at each level.

Before we dive into things, there is one getter that needs to be added to the ``ObjLoader``` class:

```
public Triangle[] CollisionMesh {
    get {
        return collisionMesh;
    }
}
```

Spaces

The collision tests against a model are going to be a bit different than those against primitives, this is because the model is located in it's own model space. Model space is always located at the origin. So long as we render a model like this:

```
public override void Render() {
    base.Render();
    DrawOrigin();

    obj.Render();
}
```

Everything will be fine, because the model is at origin. But as soon as we scale, rotate or translate the model:

```
public override void Render() {
    base.Render();
    DrawOrigin();

    GL.PushMatrix();
    GL.Translate(1f, 0f, 0f);
    GL.Scale(3.0f, 3.0f, 3.0f);
    obj.DebugRender();
    GL.PopMatrix();
}
```

Now the collisions wont work. Visually, the model looks different than how it's represented in memory. And all of the bounding primitives (triangles, AABB, Sphere) that we built around it, we built around the model in memory, not the visual model.

Luckily, we can solve this using matrix operations. The translation-rotation-scale of a model can be used to create a transform matrix. This matrix represents where in world space the model is.

If we take the inverse of that matrix, and multiply any point by it, the point will be transformed so that the model is at the origin of the points space.

This last bit can get a tad confusing, especially since we haven't done any matrix math in a while. If you need any help on any of it, give me a call on skype.

The OBJ Class

To make the concept of spaces less confusing, we're going to create a new class, the `OBJ` class. This class is going to simply hold a `OBJLoader` object and the transform matrix of the object. Let's implement the class as such:

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;

namespace CollisionDetectionSelector.Primitives {
    class OBJ {
        OBJLoader model = null;

        // VERY IMPORTANT THAT THESE HAVE DEFAULT VALUES
        protected Vector3 position = new Vector3(0f, 0f, 0f);
        protected Vector3 rotation = new Vector3(0f, 0f, 0f);
        protected Vector3 scale = new Vector3(1f, 1f, 1f);

        public OBJ(OBJLoader loader) {
            model = loader;
        }
    }
}
```

From this class, we can tell that an OBJ has a position, a rotation and a scale. It holds an OBJLoader, and can only be constructed if an OBJLoader is present. The whole point of this class was to hold a transform matrix. So far, we have all the components of that matrix, but not the matrix its-self. Let's add it.

The matrix is going to be a private variable, with a dirty flag. When getting the matrix trough a public accessor, if the dirty flag is true, the private matrix is re-calculated from position, rotation and scale, then the new correct matrix is returned

```
// Add this to the OBJ class
protected Matrix4 worldMatrix;
protected bool dirty = true; // MUST BE TRUE BY DEFAULT

public Matrix4 WorldMatrix {
    get {
        if (dirty) {
            Matrix4 translation = Matrix4.Translate(position);

            Matrix4 pitch = Matrix4.XRotation(rotation.X);
            Matrix4 yaw = Matrix4.YRotation(rotation.Y);
            Matrix4 roll = Matrix4.ZRotation(rotation.Z);
            Matrix4 orientation = roll * pitch * yaw;

            Matrix4 scaling = Matrix4.Scale(scale);

            worldMatrix = translation * orientation * scaling;
        }
        return worldMatrix;
    }
}
```

So, what is it that will set this matrix to be dirty? Ideally changing the position, rotation or scale of the object. Lets add getters and setters for these properties. Whenever one of them is set, we flip the dirty flag to true.

This way we can change all 3 properties in the intializer of the unit test (or your future game), and the world matrix will still only be re-calculated once.

```
// Add this to the OBJ class
public Vector3 Position {
    get { return position; }
    set {
        position = value;
        dirty = true;
    }
}

public Vector3 Rotation {
    get { return rotation; }
    set {
        rotation = value;
        dirty = true;
    }
}

public Vector3 Scale {
    get { return scale; }
    set {
        scale = value;
        dirty = true;
    }
}
```

Additionally, the OBJ class will be used to test collisions, so it needs to expose the collision related properties of the OBJLoader it contains, like so:

```
// Add this to the OBJ class
public AABB BoundingBox {
    get {
        return model.BoundingBox;
    }
}

public Sphere BoundingSphere {
    get {
        return model.BoundingSphere;
    }
}

public Triangle[] Mesh {
    get {
        return model.CollisionMesh;
    }
}
```

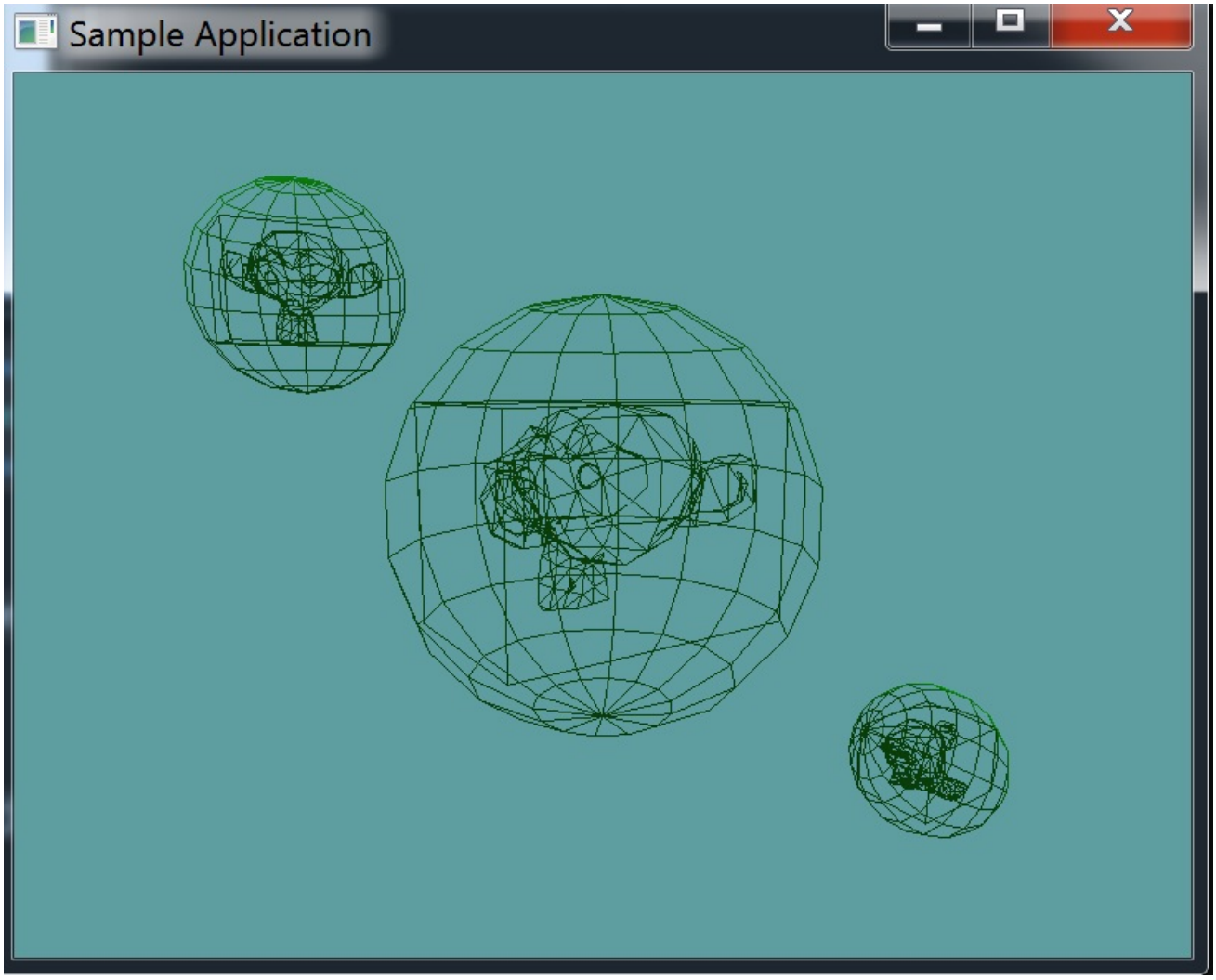
Finally, all primitives have a Render and a ToString function. It's important that we call the WorldMatrix getter when setting the matrix in the renderer, if we don't the dirty flag might not be notices and the matrix might not get re-calculated.

```
// Add this to the OBJ class
public void Render() {
    GL.PushMatrix();
    // IMPORTANT: Calling the getter, not raw accessing the array!
    GL.MultMatrix(WorldMatrix.OpenGL);
    model.Render();
    GL.PopMatrix();
}

public void DebugRender() {
    GL.PushMatrix();
    // IMPORTANT: Calling the getter, not raw accessing the array!
    GL.MultMatrix(WorldMatrix.OpenGL);
    model.DebugRender();
    GL.PopMatrix();
}

public override string ToString() {
    return "Triangle count: " + model.NumCollisionTriangles;
}
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test is visual only, make sure your project looks like the screenshot.




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJClassSample : Application {
        OBJLoader loader = null;
        OBJ[] objs = new OBJ[] { null, null, null };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
```

```
            GL.Enable(EnableCap.Light0);

            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);

            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 });

            loader = new OBJLoader("Assets/suzanne.obj");
            objs[0] = new OBJ(loader);
            objs[1] = new OBJ(loader);
            objs[2] = new OBJ(loader);

            objs[0].Scale = new Vector3(3.0f, 3.0f, 3.0f);

            objs[1].Position = new Vector3(6.0f, 6.0f, 6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);

            objs[2].Position = new Vector3(-6.0f, -6.0f, -6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);
            objs[2].Rotation = new Vector3(90.0f, 0.0f, 0.0f);
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            foreach(OBJ obj in objs) {
                obj.DebugRender();
            }
        }
```

```
        }
  }
```

Model Sphere Collision

The actual testing of collisions gets a bit complicated, the sphere is the perfect primitive to explore this with.

The first ting to know is that the model is in world-space, but it's collision primitives are in model space. That is, to test the AABB of the OBJ against a sphere accuratley, we must move the AABB into world space. We could do this by multiplying the min and max of the AABB by the `worldMatrix` of the OBJ.

The above solution is bad for a number of reasons. first, because all of the OBJ's share one `OBJLoader`, where would such a transformed AABB live? In the OBJ, or the OBJLoader? Sure you could make the getter of OBJ do the work, but that might get messy.

Second, and more importantly, this method would only be viable for broad-phase primitives. Once you get to narrow-phase collision, multiplying each triangle by the worldMatrix becomes implausable.

A much better solution is to leave the model at the origin of the world, and move the primtiive being tested (in this case the sphere) such that relative to it, the model is at origin.

We can achieve this by multiplying the position of the sphere by the inverse of the model's world matrix. This works, because if we where to multiply the models' workld position by the inverse of it's work matrix too, it would be at the origin of the world. In theory we translate both objects the same.

We also have to scale the radius of the center, we find the scalar by taking the largest scale component [0, 0], [1, 1] or [2, 2] of the inverse matrix and multiplying the radius by it. When we do the comparison for the largest component, we must compare absolute values, a negative scale is valid

The test

The test is simple, use the inverse world matrix of the model to create a new sphere whos center is translated. Then, do broad phase testing, if neither bounding sphere, nor bounding box intersect no intersection can happen. Finally, loop trough all the triangles, if one of them intersects and intersection did happen. If none of them do, no intersection happened.

I'm providing all the code needed for this one, there is no todo. Add the following test to `Collisions.cs`

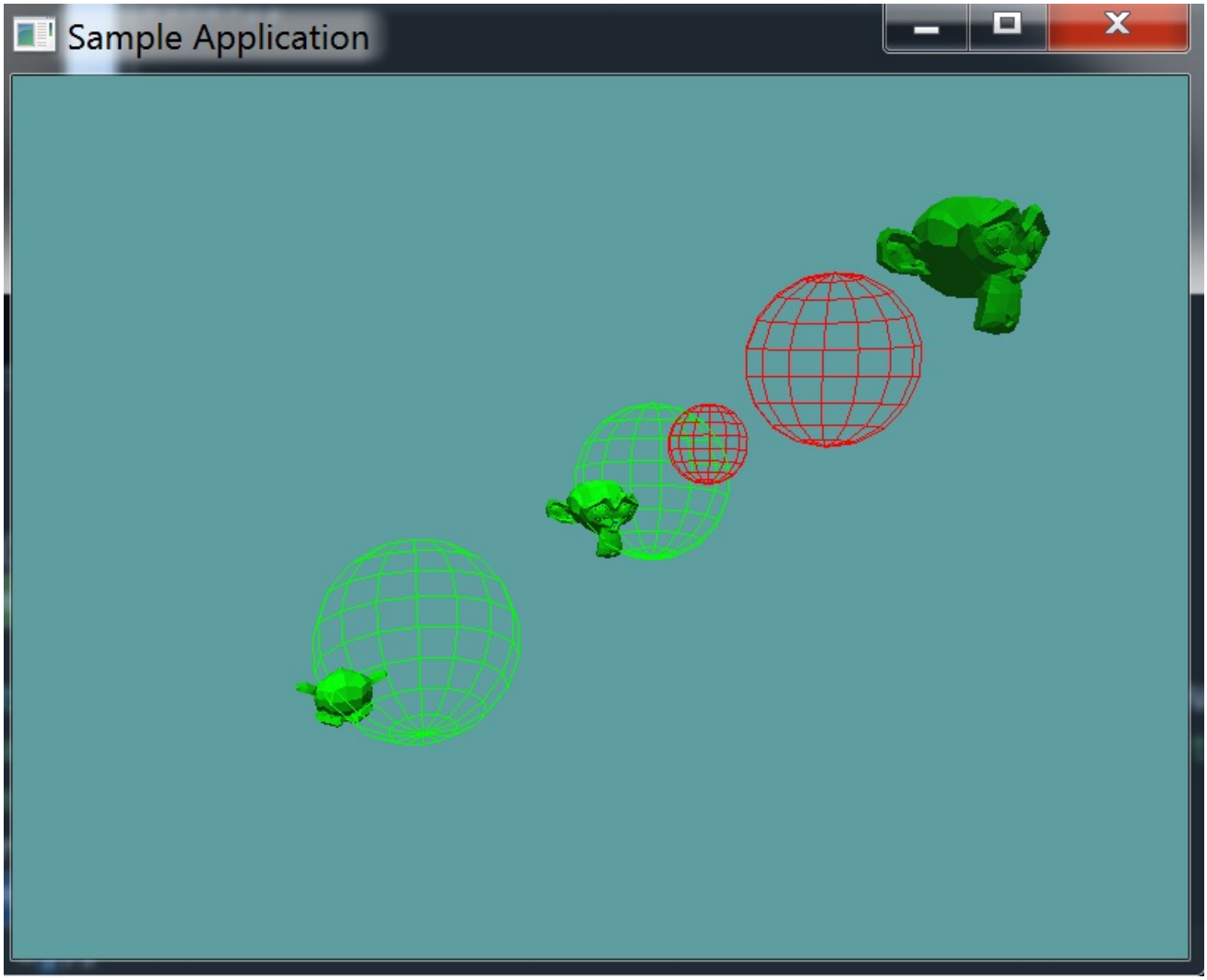
```
// Conveniance method
public static bool Intersects(OBJ model, Sphere sphere) {
    return Intersects(sphere, model);
}

public static bool Intersects(Sphere sphere, OBJ model) {
    Matrix4 inverseWorldMatrix = Matrix4.Inverse(model.WorldMatrix
);

    Vector3 newSpherePos = Matrix4.MultiplyPoint(inverseWorldMatri
x, sphere.Position.ToVector());
    // We have to scale the radius of the sphere! This is difficul
t. The new scalar is the old radius
    // multiplied by the largest scale component of the matrix
    float newSphereRad = sphere.Radius * Math.Abs(Math.Max(
            Math.Max(Math.Abs(inverseWorldMatrix[0, 0]), Math.Abs(
inverseWorldMatrix[1, 1])),
            Math.Abs(inverseWorldMatrix[2, 2])));

    // We make this new sphere because the old one is passed in
    // by reference. We don't want to actually modify
    // the reference!
    Sphere translatedSphere = new Sphere(newSpherePos, newSphereRa
d);

    // Broad-phase
    // If the bounding sphere does not intersect, nothing will
    if (!Intersects(model.BoundingSphere, translatedSphere)) {
```

```
        return false;
    }

    // Broad-phase
    // If the bounding box does not intersect, nothing will
    if (!Intersects(model.BoundingBox, translatedSphere)) {
        return false;
    }

    // Narrow-phase
    // At least one triangle must intersect!
    for(int i = 0, len = model.Mesh.Length; i < len; ++i) {
        if (Intersects(model.Mesh[i], translatedSphere)) {
            return true;
        }
    }

    // Narrow-phase
    // None of the triangles intersected, no intersection
    return false;
}
```

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test is visual only, make sure your project looks like the screenshot. Pay special attention to the sphere colors, green means there was a collision, red means there was not!




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJSphereIntersection : Application {
        OBJLoader loader = null;
        OBJ[] objs = new OBJ[] { null, null, null };

        Sphere[] tests = new Sphere[] {
            new Sphere(4f, 4f, 4f, 2f),
            new Sphere(-4f, -4f, -4f,3f),
            new Sphere(1f, 1f, 1f, 2f),
```

```
            new Sphere(2f, 2f, 2f, 1f),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            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 });

            loader = new OBJLoader("Assets/suzanne.obj");
            objs[0] = new OBJ(loader);
            objs[1] = new OBJ(loader);
            objs[2] = new OBJ(loader);

            objs[1].Position = new Vector3(6.0f, 6.0f, 6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);

            objs[2].Position = new Vector3(-6.0f, -6.0f, -6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);
            objs[2].Rotation = new Vector3(90.0f, 0.0f, 0.0f);
        }

        public override void Render() {
            GL.Disable(EnableCap.Lighting);
            base.Render();
            DrawOrigin();
            GL.Enable(EnableCap.Lighting);
```

```
            GL.Color3(0f, 0f, 1f);
            foreach (OBJ obj in objs) {
                obj.Render();
            }

            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.Disable(EnableCap.Lighting);
            foreach (Sphere test in tests) {
                bool intersection = false;
                foreach (OBJ obj in objs) {
                    if (Collisions.Intersects(test, obj)) {
                        intersection = true;
                    }
                }
                if (intersection) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                test.Render();
            }
            GL.Enable(EnableCap.Lighting);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Fill);
        }
    }
}
```

Model AABB Intersection

We take a similar approach to the model AABB collision that we took to the model sphere collision. First, you find the inverse world matrix of the model. Next, construct a new AABB that is translated by this inverse matrix. Then do the collision waterfall:

- If the new AABB does not collide with the model's bounding box, there is no collision
- If the new AABB does not collide with the models bounding sphere, there is no collision
- If the new AABB collides with one of the models triangles, there is a collision
- By default, there is no collision

On Your Own

Add the following function to the `Collisions` class:

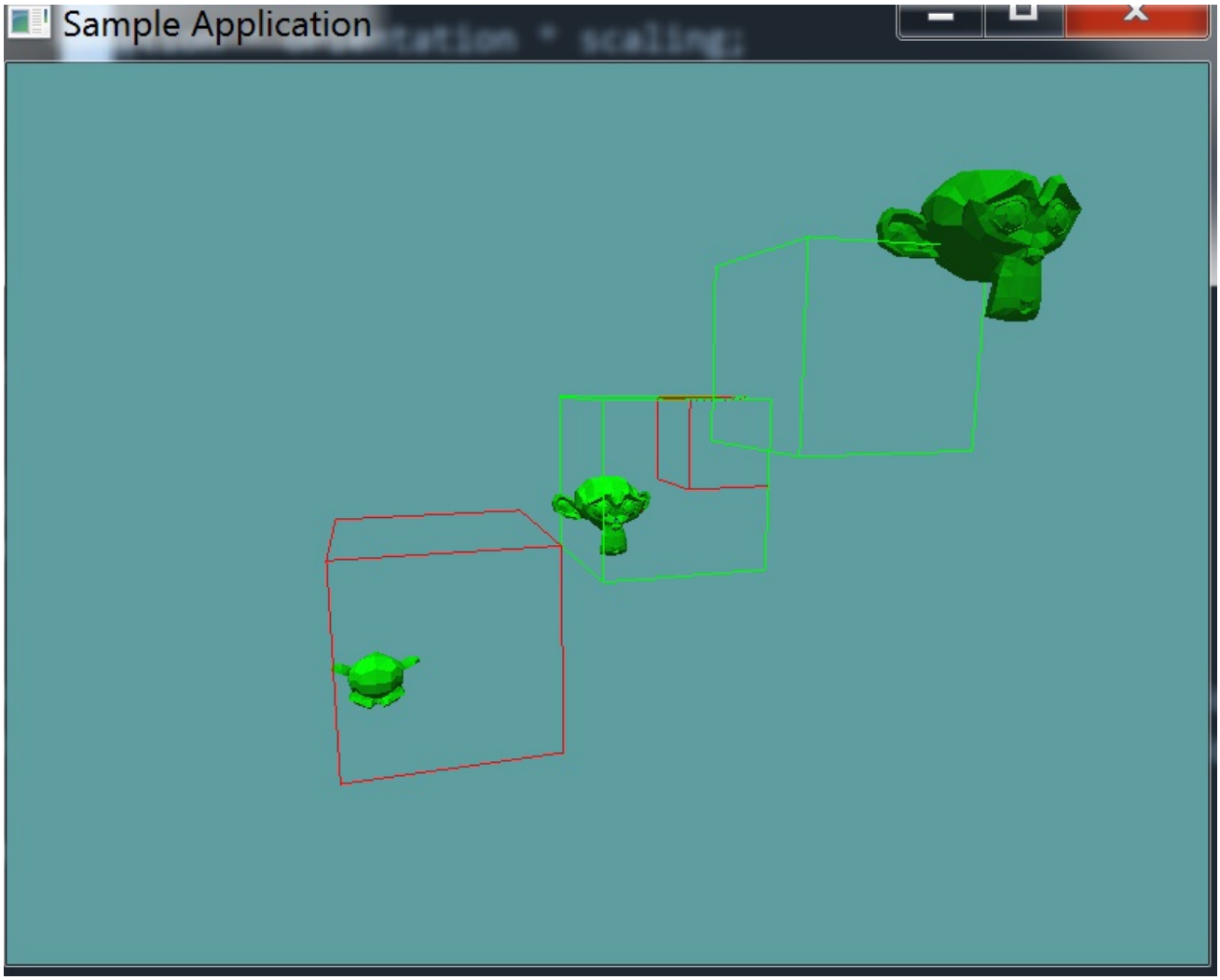
```
// Conveniance function
public static bool Intersects(OBJ model, AABB aabb) {
    return Intersects(aabb, model);
}

public static bool Intersects(AABB aabb, OBJ model) {
    // TODO: Provide an implementation for this
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test is visual only, make sure your project looks like the screenshot. Pay special attention to the aabb colors, green means there was a collision, red means there was not!



There is an error in the above image, the bottom left box is supposed to be red. IF yours is not, check your aabb, you probably need to call the `Fix` function. Despite it being visually correct, there is still a precision error.

Once you rotate an AABB, like we do in the collision test, it is no longer axis aligned. It becomes an oriented bounding box, or OBB. The fix for this bug is simple, instead of creating a new AABB that gets modified by the inverse world matrix, we need to create a new OBB.

The only issue is, we have not covered the implementation of an OBB yet, so for now, we are going to skip this and live with the error. OBB's will be covered in a future chapter.

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJAABBIntersection : Application {
        OBJLoader loader = null;
        OBJ[] objs = new OBJ[] { null, null, null };

        AABB[] tests = new AABB[] {
            new AABB(new Point(2f, 2f, 2f), new Point(6f, 6f, 6f))
,
            new AABB(new Point(-1f, -1f, -1f), new Point(-7f, -7f,
-7f)),
            new AABB(new Point(-1f, -1f, -1f), new Point(3f, 3f, 3
f)),
            new AABB(new Point(1f, 1f, 1f), new Point(3f,3f, 3f)),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            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 });

            loader = new OBJLoader("Assets/suzanne.obj");
            objs[0] = new OBJ(loader);
            objs[1] = new OBJ(loader);
            objs[2] = new OBJ(loader);

            objs[1].Position = new Vector3(6.0f, 6.0f, 6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);

            objs[2].Position = new Vector3(-6.0f, -6.0f, -6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);
            objs[2].Rotation = new Vector3(90.0f, 0.0f, 0.0f);
        }

        public override void Render() {
            GL.Disable(EnableCap.Lighting);
            base.Render();
            DrawOrigin();
            GL.Enable(EnableCap.Lighting);

            GL.Color3(0f, 0f, 1f);
            foreach (OBJ obj in objs) {
                obj.Render();
            }

            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Line);
            GL.Disable(EnableCap.Lighting);
            foreach (AABB test in tests) {
                bool intersection = false;
                foreach (OBJ obj in objs) {
                    if (Collisions.Intersects(test, obj)) {
                        intersection = true;
                    }
                }
                if (intersection) {
```

```
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                test.Render();
            }
            GL.Enable(EnableCap.Lighting);
            GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.
Fill);
        }
    }
}
```

Model Plane Collision

By now you can probably see the pattern of how model collisions are done. Model plane is the same way.

You must multiply the plane normal by the inverse world matrix of the model. This is a normal, not a point, so use the `Matrix4.MultiplyVector` method.

Scale the distance of the plane the same way we scaled the radius of the sphere.

On Your Own

Add the following function to the `Collisions` class:

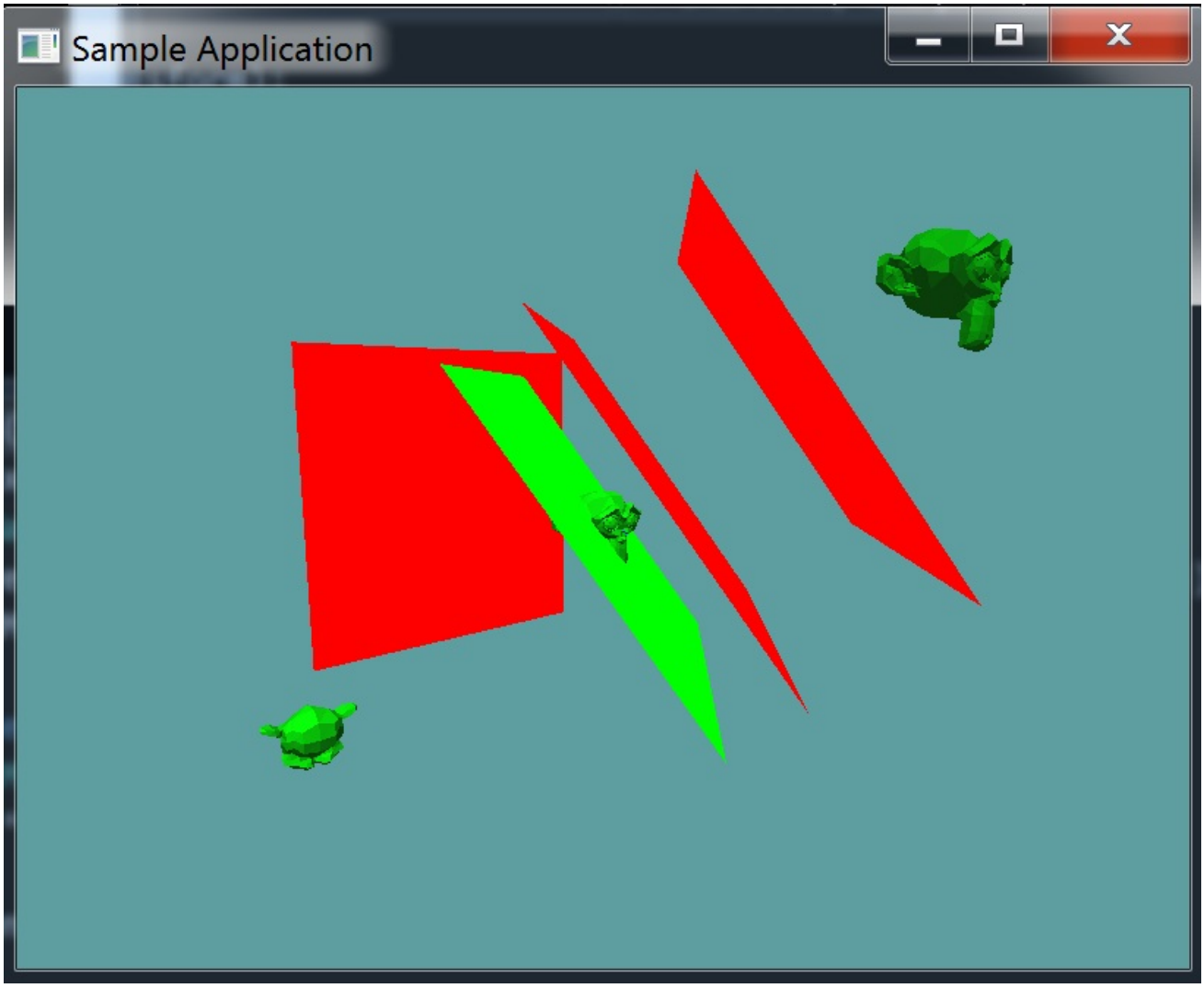
```
// conveniance method
public static bool Intersects(OBJ model, Plane plane) {
    return Intersects(plane, model);
}

public static bool Intersects(Plane plane, OBJ model) {
    // TODO
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test is visual only, make sure your project looks like the screenshot. Pay special attention to the plane colors, green means there was a collision, red means there was not!




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJPlaneIntersection : Application {
        OBJLoader loader = null;
        OBJ[] objs = new OBJ[] { null, null, null };

        Plane[] tests = new Plane[] {
            new Plane(new Vector3(1f, 1f, 1f), 2.0f),
            new Plane(new Vector3(1f, 1f, 1f), 0.0f),
            new Plane(new Vector3(-6f, -6f, -6f), -6.0f),
```

```
            new Plane(new Point(-4f, -8f, -8f), new Point(-5f, -4f
, -8f), new Point(-7f, -4f, -8f))
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            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 });

            loader = new OBJLoader("Assets/suzanne.obj");
            objs[0] = new OBJ(loader);
            objs[1] = new OBJ(loader);
            objs[2] = new OBJ(loader);

            objs[1].Position = new Vector3(6.0f, 6.0f, 6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);

            objs[2].Position = new Vector3(-6.0f, -6.0f, -6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);
            objs[2].Rotation = new Vector3(90.0f, 0.0f, 0.0f);
        }

        public override void Render() {
            GL.Disable(EnableCap.Lighting);
            base.Render();
            DrawOrigin();
            GL.Enable(EnableCap.Lighting);
```

```
            GL.Color3(0f, 0f, 1f);
            foreach (OBJ obj in objs) {
                obj.Render();
            }

            GL.Disable(EnableCap.Lighting);
            GL.Disable(EnableCap.CullFace);
            foreach (Plane test in tests) {
                bool intersection = false;
                foreach (OBJ obj in objs) {
                    if (Collisions.Intersects(test, obj)) {
                        intersection = true;
                    }
                }
                if (intersection) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                test.Render(5f);
            }
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
        }
    }
}
```

Triangle model collision

At this point you should have enough practice with the collision test methods to implement this one without any help. I'm only going to provide the unit test for this.

On Your Own

Add the following function to the `Collisions` class:

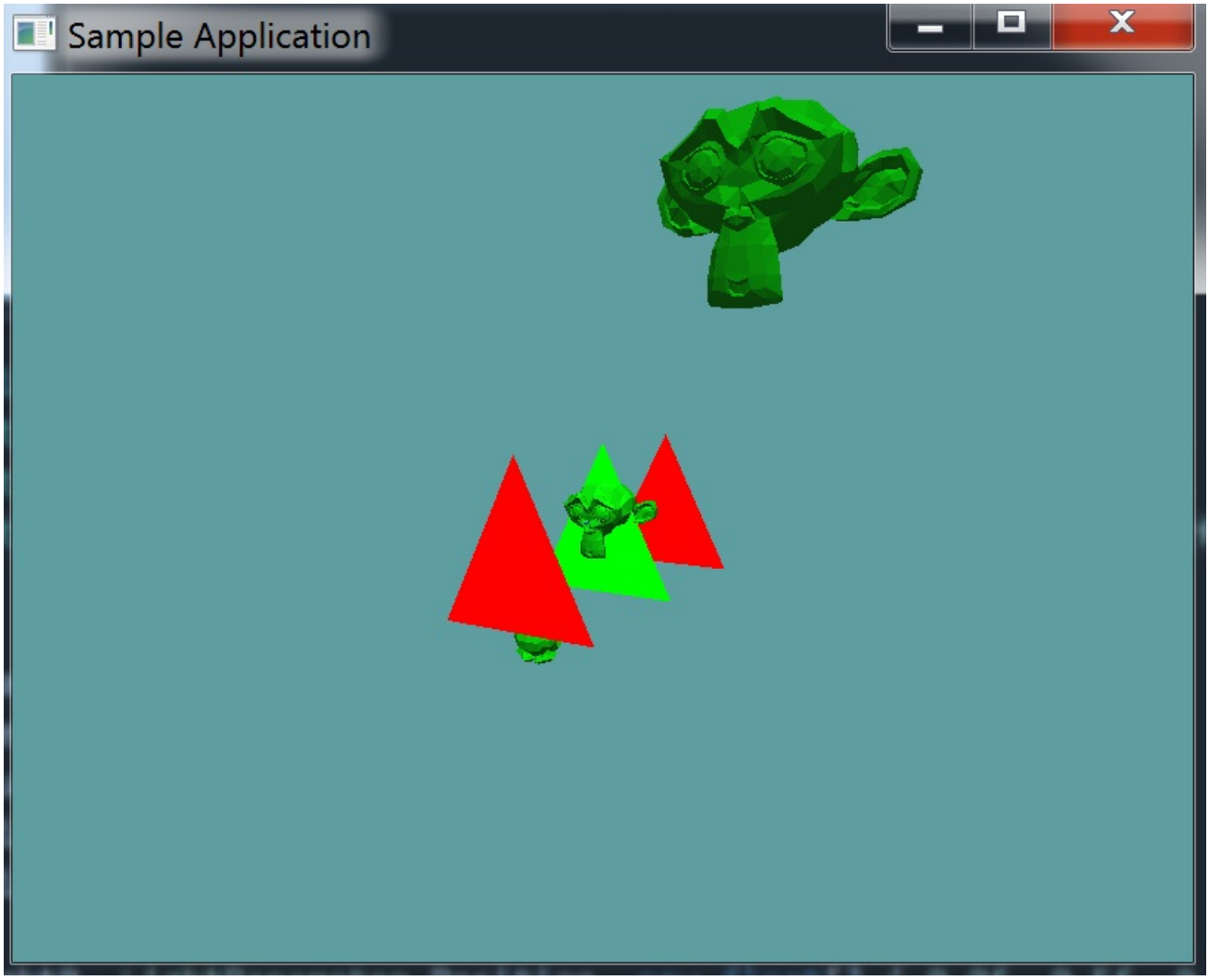
```
// Conveniance function
public static bool Intersects(OBJ model, Triangle triangle) {
    return Intersects(triangle, model);
}


public static bool Intersects(Triangle triangle, OBJ model) {
    // TODO: Implement this one!
}
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test is visual only, make sure your project looks like the screenshot. Pay special attention to the triangle colors, green means there was a collision, red means there was not!




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJTriangleIntersection : Application {
        OBJLoader loader = null;
        OBJ[] objs = new OBJ[] { null, null, null };

        Triangle[] tests = new Triangle[] {
            new Triangle(new Point(-2f, -2f, 0f), new Point(2f, -2
f, 0f), new Point(0f, 2f, 0f)),
            new Triangle(new Point(-2f, -2f, 4f), new Point(2f, -2
```

```
f, 4f), new Point(0f, 2f, 4f)),
            new Triangle(new Point(-2f, -2f, -4f), new Point(2f, -2
f, -4f), new Point(0f, 2f, -4f)),
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            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 });

            loader = new OBJLoader("Assets/suzanne.obj");
            objs[0] = new OBJ(loader);
            objs[1] = new OBJ(loader);
            objs[2] = new OBJ(loader);

            objs[1].Position = new Vector3(6.0f, 6.0f, 6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);

            objs[2].Position = new Vector3(-6.0f, -6.0f, -6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);
            objs[2].Rotation = new Vector3(90.0f, 0.0f, 0.0f);
        }

        public override void Render() {
            GL.Disable(EnableCap.Lighting);
            base.Render();
            DrawOrigin();
```

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

            GL.Color3(0f, 0f, 1f);
            foreach (OBJ obj in objs) {
                obj.Render();
            }

            GL.Disable(EnableCap.Lighting);
            GL.Disable(EnableCap.CullFace);
            foreach (Triangle test in tests) {
                bool intersection = false;
                foreach (OBJ obj in objs) {
                    if (Collisions.Intersects(test, obj)) {
                        intersection = true;
                    }
                }
                if (intersection) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                test.Render();
            }
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
        }
    }
}
```

Model Raycast

To raycast against a model is similar to what we've been doing so far. Make a new ray by multiplying the position of the ray (MultiplyPoint) and the normal of the ray (MultiplyVector) by the inverse matrix. Then, use this new ray to raycast against the models bounding sphere, bounding box and finally all it's triangles.

When you call the raycast against the models bounding sphere or box, pass the argument t that this method recieves as the out argument. Example:

```
public static bool Raycast(Ray ray, OBJ model, out float t) {
    // Code to create newRay
    if (!Raycast(newRay, model.BoundingSphere, out t)) {
        return false;
    }
    // rest of function
}
```

On Your Own

Add the following function to the `Collisions` class:

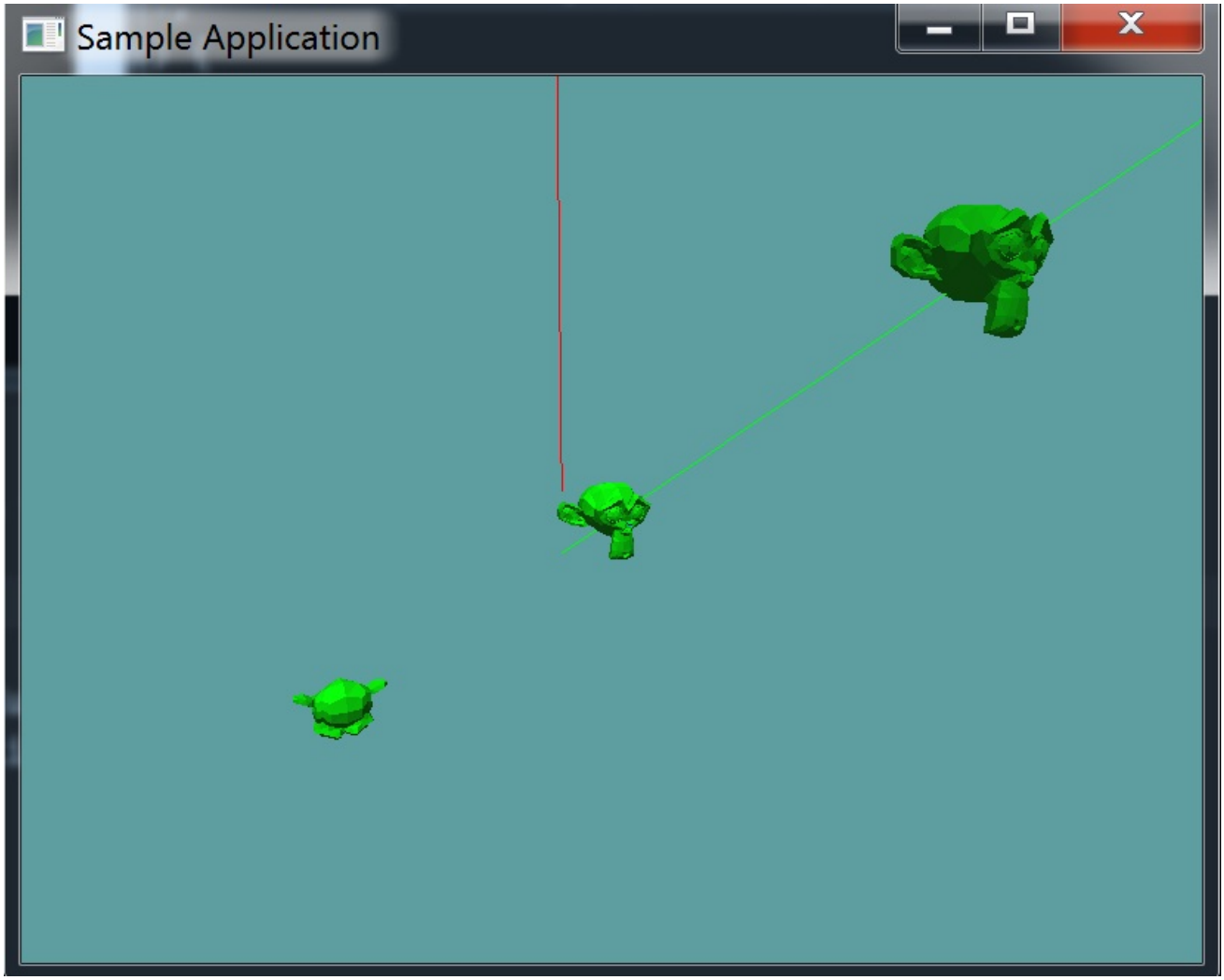
```
public static bool Raycast(Ray ray, OBJ model, out float t)
```

And provide an implementation for it!

Unit Test

You can Download the samples for this chapter to see if your result looks like the unit test.

This unit test is visual only, make sure your project looks like the screenshot. Pay special attention to the ray colors, green means there was a collision, red means there was not!




```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class OBJRaycast : Application {
        OBJLoader loader = null;
        OBJ[] objs = new OBJ[] { null, null, null };

        Ray[] tests = new Ray[] {
```

```
            new Ray(new Point(-1f, -1f, -1f), new Vector3(1f, 1f, 1
f)),
            new Ray(new Point(-2f, 1f, 1f), new Vector3(0f, 1f, 0f
))
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);

            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 });

            loader = new OBJLoader("Assets/suzanne.obj");
            objs[0] = new OBJ(loader);
            objs[1] = new OBJ(loader);
            objs[2] = new OBJ(loader);

            objs[1].Position = new Vector3(6.0f, 6.0f, 6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);

            objs[2].Position = new Vector3(-6.0f, -6.0f, -6.0f);
            objs[1].Scale = new Vector3(1.5f, 1.5f, 1.5f);
            objs[2].Rotation = new Vector3(90.0f, 0.0f, 0.0f);
        }

        public override void Render() {
            GL.Disable(EnableCap.Lighting);
            base.Render();
```

```
            DrawOrigin();
            GL.Enable(EnableCap.Lighting);

            GL.Color3(0f, 0f, 1f);
            foreach (OBJ obj in objs) {
                obj.Render();
            }

            GL.Disable(EnableCap.Lighting);
            GL.Disable(EnableCap.CullFace);
            float t = 0;
            foreach (Ray test in tests) {
                bool intersection = false;
                foreach (OBJ obj in objs) {
                    if (Collisions.Raycast(test, obj, out t)) {
                        intersection = true;
                    }
                }
                if (intersection) {
                    GL.Color3(0f, 1f, 0f);
                }
                else {
                    GL.Color3(1f, 0f, 0f);
                }
                test.Render();
            }
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
        }
    }
}
```

TODO: Review This

Raycast Model

Raycasting against an OBJ model is a VERY common operation. So common in fact, that the primitive implementation we have created so far is too slow.

One of the best examples of this is clamping a character to the world. The world (terrain) is usually represented as an OBJ mesh. Characters run around on top of this mesh. To clamp the character to the mesh we often use a raycast.

That is, we raycast from above the character down onto the terrain, wherever the raycast hits is where we clamp the character to. Having a terrain with say 6,000 triangles and a player character is no problem. Looping trough 6k triangles and doing a raycast against each of them once is acceptable. But when you have NPC's involved, looping trough the same ammount of characters 20 times slows the game to a crawl!

We can fix this problem by adding a new acceleration structure to the OBJ class, similar to the BoundingSphere and BoundingBox. This new acceleration structure is known as a bounding volume hierarchy, or BVH for short. (Side note, BVH is also the extension of an animation format, making googling the topic a little difficult)

Bounding Volume Hierarchy

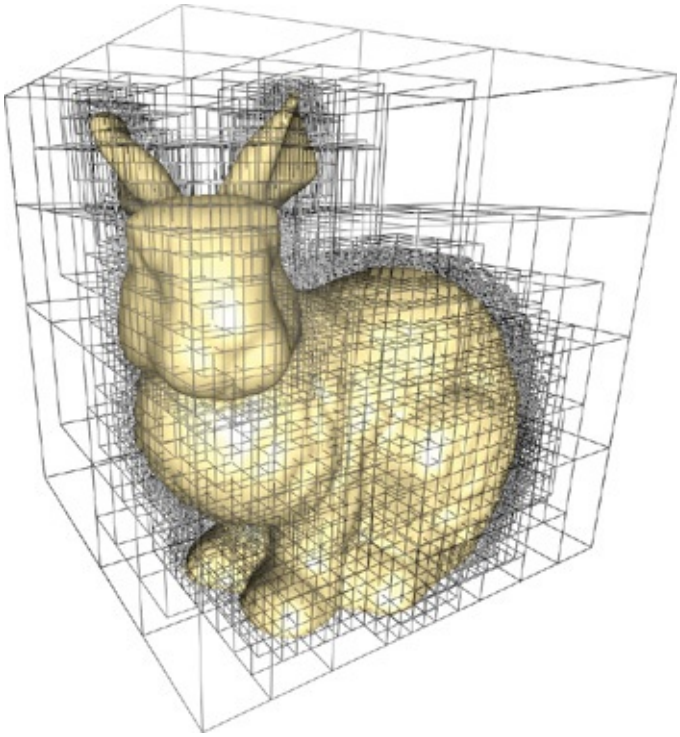
The idea of a BVH is simple, it's a tree of AABB's. We take the top level bounding box, and divide it into 4 smaller bounding boxes. We keep doing this until the tree is at a certain depth (Say 12). Now we have a highly subdivided model.

We loop trough each leaf node and check for collision against the triangles of the model. Whatever triangle collides with the leaf node is added to a list of triangles inside the leaf node.

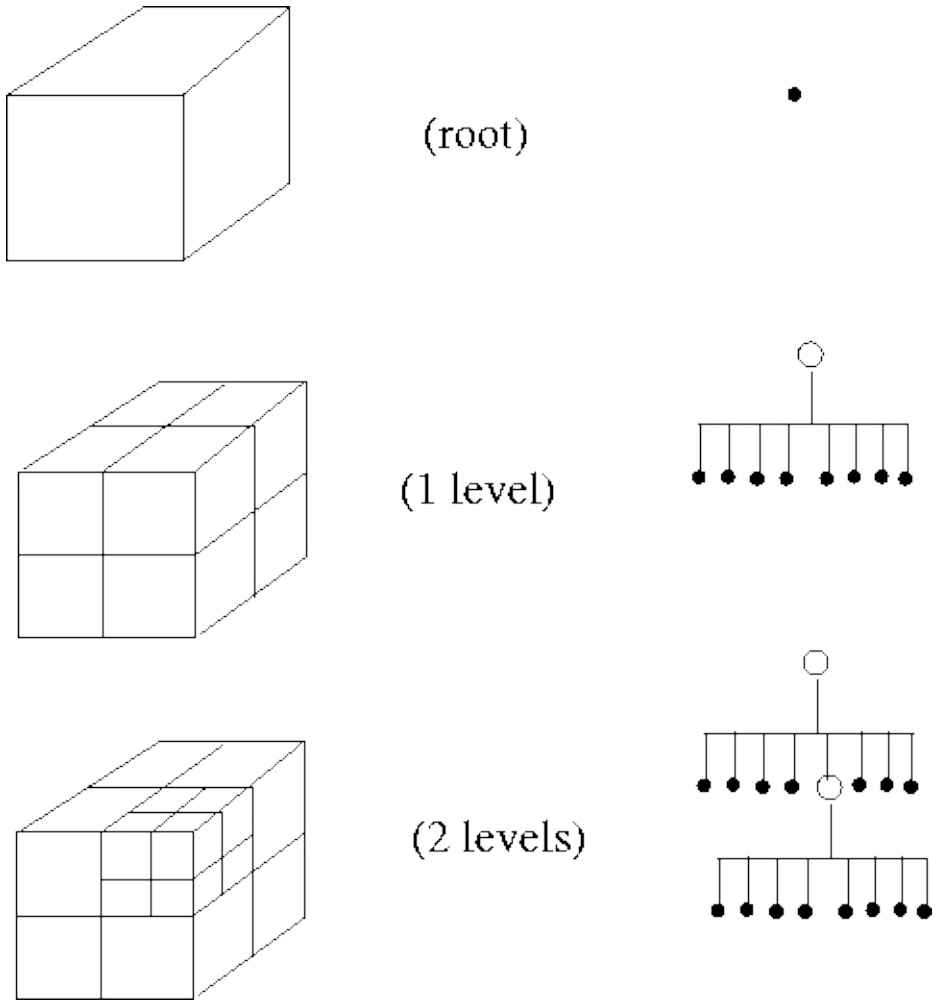
Now we have an AABB tree with a bunch of triangles in each leaf node. When you do a raycast you start at the top of the tree, if the ray intersects this AABB, then you check all 4 of it's childern. Whichever child the ray intersects, you check it's children. It's a recursive process until your ray hits a leaf node. Once your ray hit a leaf node, you check for collision with the triangles inside that leaf node.

Essentially, you just traded checking 6k triangles / raycast for checking maybe 20 AABB's and 30 Triangles. You can see how this will speed up raycasting against a mesh tremendusly!

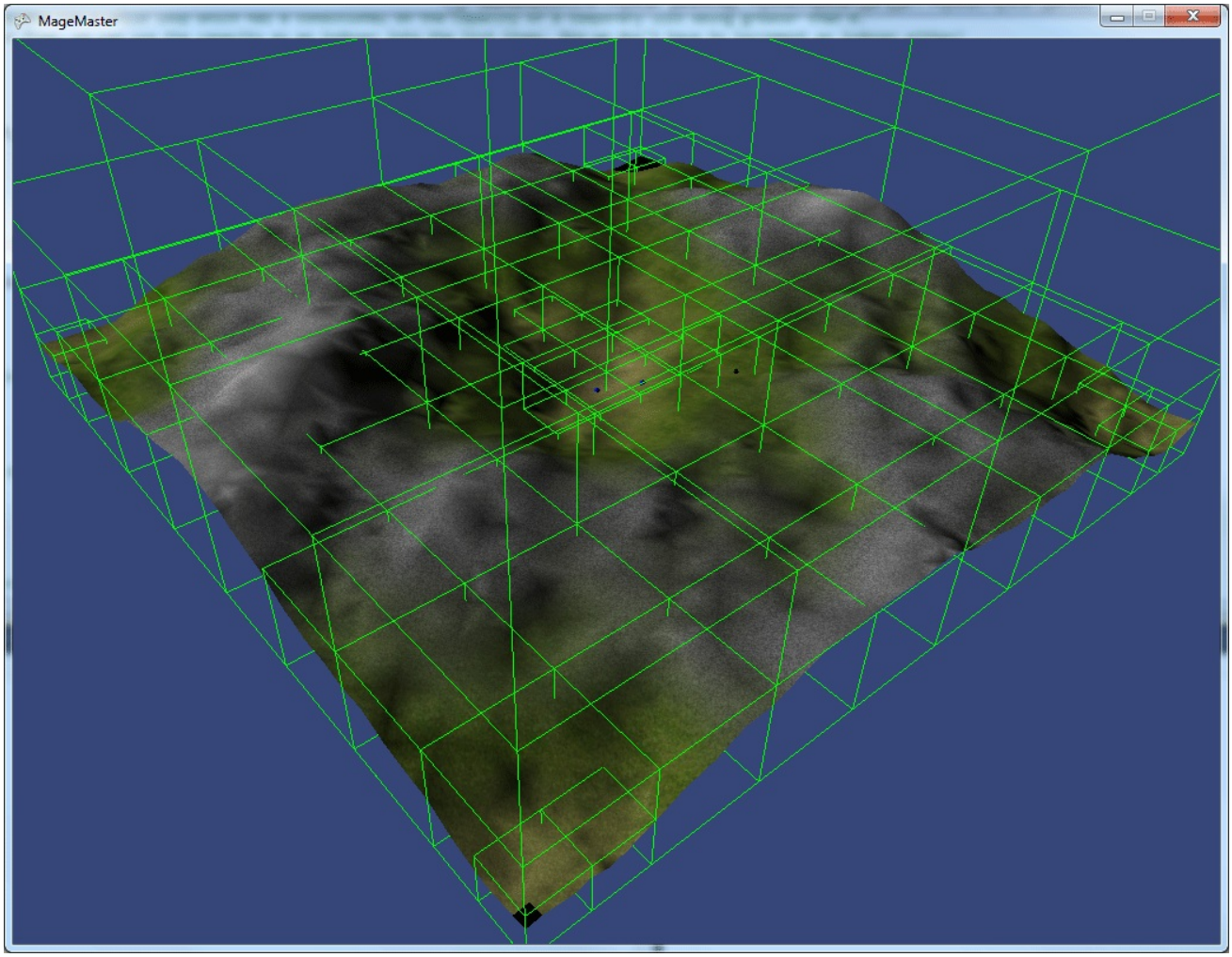
Here is an example of a BVH around a multi million triangle bunny model. This BVH is optimized in a manner that when an AABB does not contain any triangles, it is not split into child nodes. Therefore, this tree has leaf nodes with no triangles.



It might not be obvious in the above image, but this is how the ocrree is constructed trough sub-division:



And this is what your average mehs terrain looks like with the BVH acceleration structure rendering around it.



Optimize ALL OF IT

We're going to strictly use the BVH structure for raycasting, but professional engines use it for everything! Any time we loop trough all of the triangles of a model (Like OBJ Sphere intersection for example), we could replace that loop with a BHV query. This would improve performance considerably.

For now, the performance of our collision functions is fine, let's focus on fixing the only function that has performance issues, and that is raycast-mesh.

Bounding Volume Hierarchy

We're going to develop a BVH around our model class. The actual BVH will be a part of the OBJLoader class, but it's methods will be exposed trough the OBJ class as well.

I guess it's worth noting, every time i've built one of these BVH containers it came out different than the last one. This is why tree construction is a common interview question.

Because a bounding volume hierarchy is a tree, it is made up of nodes. The root of the tree will just be a regular node. So, before anything else, let's develop a class for the bvh node

BVHNode

For now we're going to divide the BVH tree at most 3 times, this is going to be a protected static variable.

A BVHNode is either a leaf node or not. If it is a leaf node, it will contain triangles and have no children. If it is not a leaf node, it will have children but no triangles. A node can have triangles or children, never both. We will expose this in a getter.

Finally, each BVH node is a box, an AABB box. As such, each node will need to have an AABB member. Also, when a new node is made, we assume it is going to be a leaf node.

```
public class BVHNode {
    protected static int maxDepth = 3;

    public List<BVHNode> Children = null;
    public List<Triangle> Triangles = null;
    public AABB AABB = null;

    public BVHNode(AABB aabb) {
        // Store AABB by value, not reference
        AABB = new AABB(aabb.Min, aabb.Max);
        // Assume this is leaf node by default
        Triangles = new List<Triangle>();
        Children = null;
    }

    public bool IsLeaf {
        get {
            return Children == null && Triangles != null;
        }
    }
}
```

Let's add a BVHNode object to the `OBJLoader` class. With it, we're going to add a recursive "RenderBVH" function, that will render a BVH and all of it's children!

```
protected BVHNode bvhRoot = null;

public BVHNode BvhRoot {
    get {
        return bvhRoot;
    }
}

// Non-Recursive public API
public void RenderBVH() {
    RenderBVH(bvhRoot);
}

// Recursive private API
private void RenderBVH(BVHNode node) {
    node.AABB.Render();

    if (node.Children != null) {
        foreach (BVHNode child in node.Children) {
            RenderBVH(child);
        }
    }
}
```

Even tough we've written functions to use the bvhRoot, we have not made the root yet. We need to create the bvh root in the constructor. This needs to happen after we know the bounding box of the object.

Remember, every node is considered a leaf node when constructed, the root node is no different. Because it's a leaf node, we're going to add all the triangles it contains to it. That just happens to be every triangle of the model.

There are two new lines of code in this excerpt from the `OBJLoader` constructor.

```
collisionMesh = new Triangle[vertexData.Count / 9];
int meshCounter = 0;
/*NEW*/bvhRoot = new BVHNode(containerAABB);

for (int i = 0; i < vertexData.Count; i += 9) {
    collisionMesh[meshCounter++] = new Triangle(
        new Point(vertexData[i + 0], vertexData[i + 1], vertexData
[i + 2]),
        new Point(vertexData[i + 3], vertexData[i + 4], vertexData
[i + 5]),
        new Point(vertexData[i + 6], vertexData[i + 7], vertexData
[i + 8])
    );

    /*NEW*/bvhRoot.Triangles.Add(collisionMesh[meshCounter - 1]);
}
```

And finally, let's add a way to render this trough the OBJ interface, to the `OBJ` class, add a "RenderBVH" function:

```
public void RenderBVH() {
    GL.PushMatrix();
    GL.MultMatrix(WorldMatrix.OpenGL);
    model.RenderBVH();
    GL.PopMatrix();
}
```

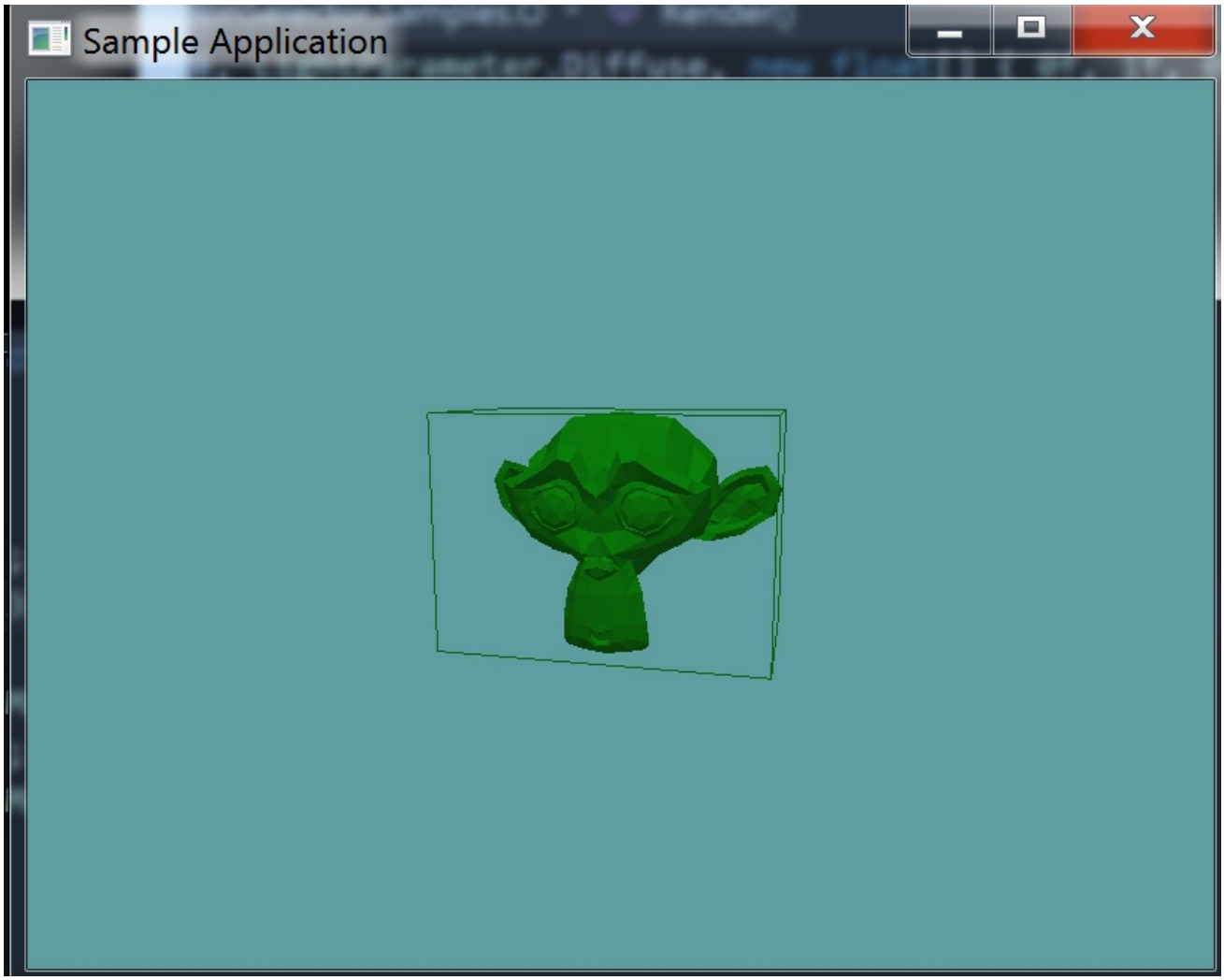
Test it!

Let's test our work so far! To test, we're going to change the existing `OBJSample` class. Even tough this class only uses the `OBJLoader` object, not the `OBJ` one, it should be enough for us to do some useful testing.

Change the render function of the `OBJSample` class, by adding a line mode render of the BVH

```
public override void Render() {
    base.Render();
    DrawOrigin();

    GL.PushMatrix();
    GL.Scale(3.0f, 3.0f, 3.0f);
    obj.Render();
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);
    obj.RenderBVH();
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);
    GL.PopMatrix();
}
```

Running your game, you should see this:



Not the most impressive piece of code in the world, but it's a start! Next up, we have to figure out a way to split that AABB!

Splitting

Splitting as you might imagine, is going to be a recursive process. We're actually going to implement 2 function, 1 public one that does not expose any recursion paramaters, and a private one that will. The paramater being passed in is recursion level.

The split function will first check that a node is a leaf node, as we can only split leaf nodes, non-leaf nodes are already split.

The leaf node is only going to be split if it has triangles. This makes sense, there is no need to split an empty node. Just keep blank space empty.

If both of the above conditions are met, we check to make sure that we are not below the recursion limit, and if not then the BVHNode is split into 8 smaller ones.

Splitting a node is simple. We get the 8 corners of the bounding box (Top Fron Left corner in code is called TFL for example). Then we make 8 new AABB's using each of the corners and the center point of the AABB as the min and max points.

Once a node has all of it's children, we loop torugh the triangles it contains, and test if the triangle fits inside one of the childern. If it does, it's added to the childs list of triangles. In this process, 1 triangle may belong to multiple children.

Once all triangles are assigned to children, we make the node a non-leaf node by clearing the triangle list it contains (Remember, the children have the triangles now) and then setting the list to null.

Finally, we loop trough every child and call the split function (this is the recursive part) on the child. This will continue the process until we've reached a certain depth.

This is the entire process in code:

```
// Public facing split function
public void Split() {
    Split(0);
```

```
}

// Private split function, recursivley calls its-self!
protected void Split(int depth) {
    // This node is currently a leaf, we can split it!
    if (Children == null && Triangles != null) {
        // Only split if the node actually contains triangles
        if (Triangles.Count > 0) {
            // Only split if we're below the maximum BVH depth
            if (depth < maxDepth) {
                Children = new List<BVHNode>();
                Vector3 center = AABB.Center.ToVector();
                Vector3 extent = AABB.Extents;

                Vector3 TFL = center + new Vector3(-extent.X, +ext
ent.Y, -extent.Z);
                Vector3 TFR = center + new Vector3(+extent.X, +ext
ent.Y, -extent.Z);
                Vector3 TBL = center + new Vector3(-extent.X, +ext
ent.Y, +extent.Z);
                Vector3 TBR = center + new Vector3(+extent.X, +ext
ent.Y, +extent.Z);
                Vector3 BFL = center + new Vector3(-extent.X, -ext
ent.Y, -extent.Z);
                Vector3 BFR = center + new Vector3(+extent.X, -ext
ent.Y, -extent.Z);
                Vector3 BBL = center + new Vector3(-extent.X, -ext
ent.Y, +extent.Z);
                Vector3 BBR = center + new Vector3(+extent.X, -ext
ent.Y, +extent.Z);

                Children.Add(new BVHNode(new AABB(new Point(TFL),
new Point(center))));
                Children.Add(new BVHNode(new AABB(new Point(TFR),
new Point(center))));
                Children.Add(new BVHNode(new AABB(new Point(TBL),
new Point(center))));
```

```
                Children.Add(new BVHNode(new AABB(new Point(TBR),
new Point(center))));
                Children.Add(new BVHNode(new AABB(new Point(BFL),
new Point(center))));
                Children.Add(new BVHNode(new AABB(new Point(BFR),
new Point(center))));
                Children.Add(new BVHNode(new AABB(new Point(BBL),
new Point(center))));
                Children.Add(new BVHNode(new AABB(new Point(BBR),
new Point(center))));
            }
        }
    }

    // IF this node was JUST split, it is neither a leaf or a
    // normal node, because it has both triangles and children!
    // Let's fix that and turn this into a non leaf node
    if (Triangles != null && Children != null) {
        foreach(Triangle triangle in Triangles) {
            foreach(BVHNode child in Children) {
                if (Collisions.Intersects(child.AABB, triangle)) {
                    child.Triangles.Add(triangle);
                }
            }
        }
        // Make sure this node is not a leaf
        Triangles.Clear();
        Triangles = null;
    }

    // Spliting is a recursive function. If this node is not a lea
f,
    // split all of it's leaf nodes (If possible)
    if (Children != null) {
        foreach(BVHNode child in Children) {
            child.Split(depth + 1);
        }
```

```
    }
}
```

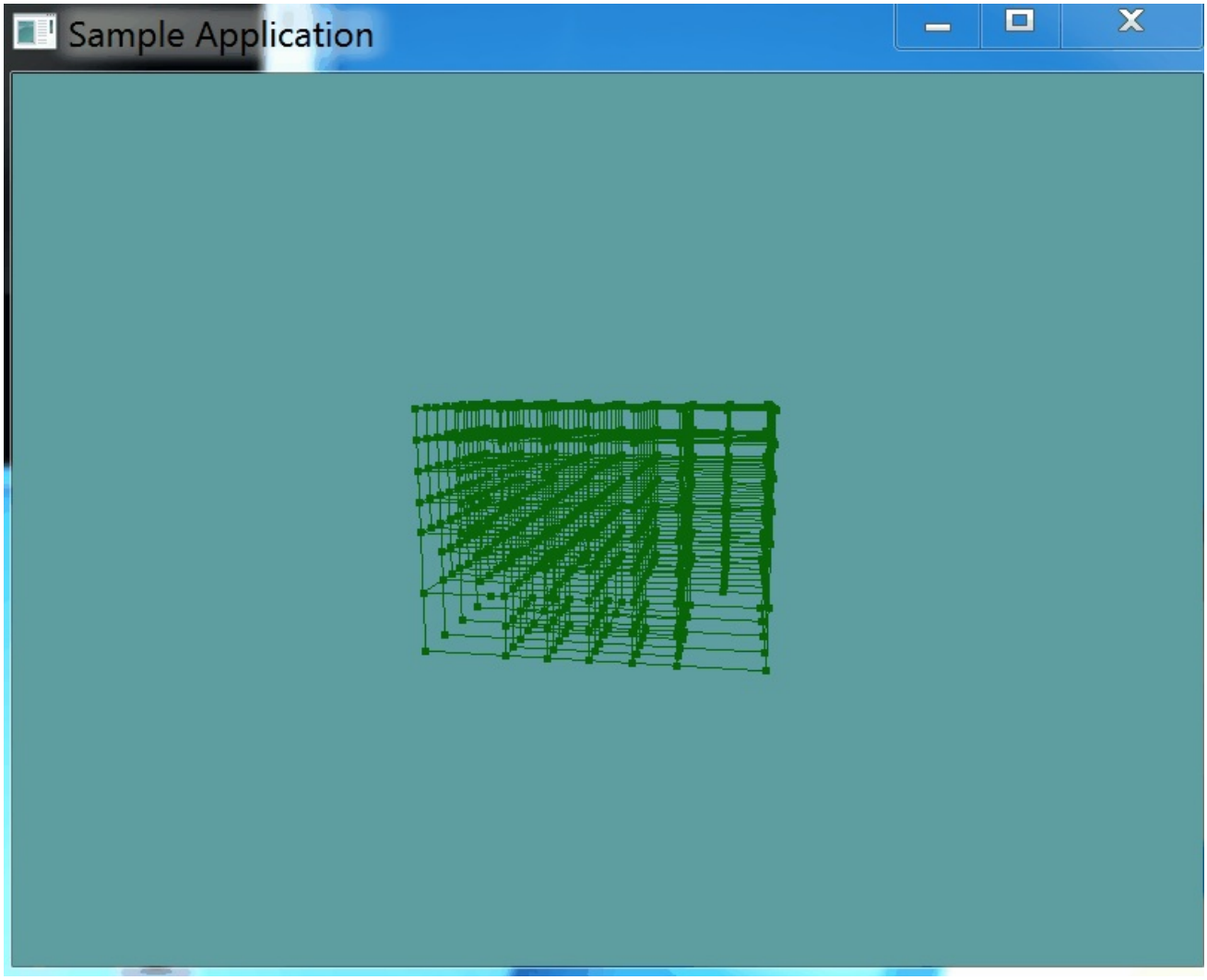
Test it!

IMPORTANT! Call that function in the `OBJLoader` constructor, right after the loop that adds all of the triangles to the BVH root node.

After that, go into the `OBJSample` class, and modify the render function so the actual model doesn't render, like so:

```
public override void Render() {
    base.Render();
    DrawOrigin();

    GL.PushMatrix();
    GL.Scale(3.0f, 3.0f, 3.0f);
    //obj.Render();
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);
    obj.RenderBVH();
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);
    GL.PopMatrix();
}
```

Running the game now, the result should look like this:



Even tough the model is not rendered, you can kind of see the outline of where the model is inside the BVH structure. Sorry for the strange gif loop.....

Shake

Right now, every node in our BVH has exactly 8 or 0 children. This structure is called an **Octree**. This octree structure is not optimal, 7 of those 8 children might not actually contain any triangles.

We can eliminate any empty leaf nodes, meaning a BVH node will have 0, 1, 2, 3, 4, 5, 6, 7 or 8 children. This modified structure is called a **sparse octree**. The process of removing empty leaf nodes from the tree is called shaking. You have to shake the tree, because programmers have a real sense of humor.

Count the triangles

The first piece of information we need to know about a BVH node in order to shake the tree is how many triangles the node contains. This isn't as simple as returning the count of the triangle list.

We need to know how many children the node, and all of it's children and all of their children contain. This means that getting the child count is going to be a recursive function.

```
public int TriangleCount {
    get {
        int totalCount = 0;
        if (Children != null) {
            foreach (BVHNode child in Children) {
                totalCount += child.TriangleCount;
            }
        }
        else if (Triangles != null) {
            totalCount += Triangles.Count;
        }

        return totalCount;
    }
}
```

Shake it!

Implementing the actual shake function is simple. First, we check if the node has any children. If it does not, we don't do anything.

If the node has children, we loop trough them BACKWARDS. Any child that has 0 triangles is removed.

After we've removed the 0 triangle children, we loop trough the remaining children and call shake on them. In this way, the shake function is recursive.

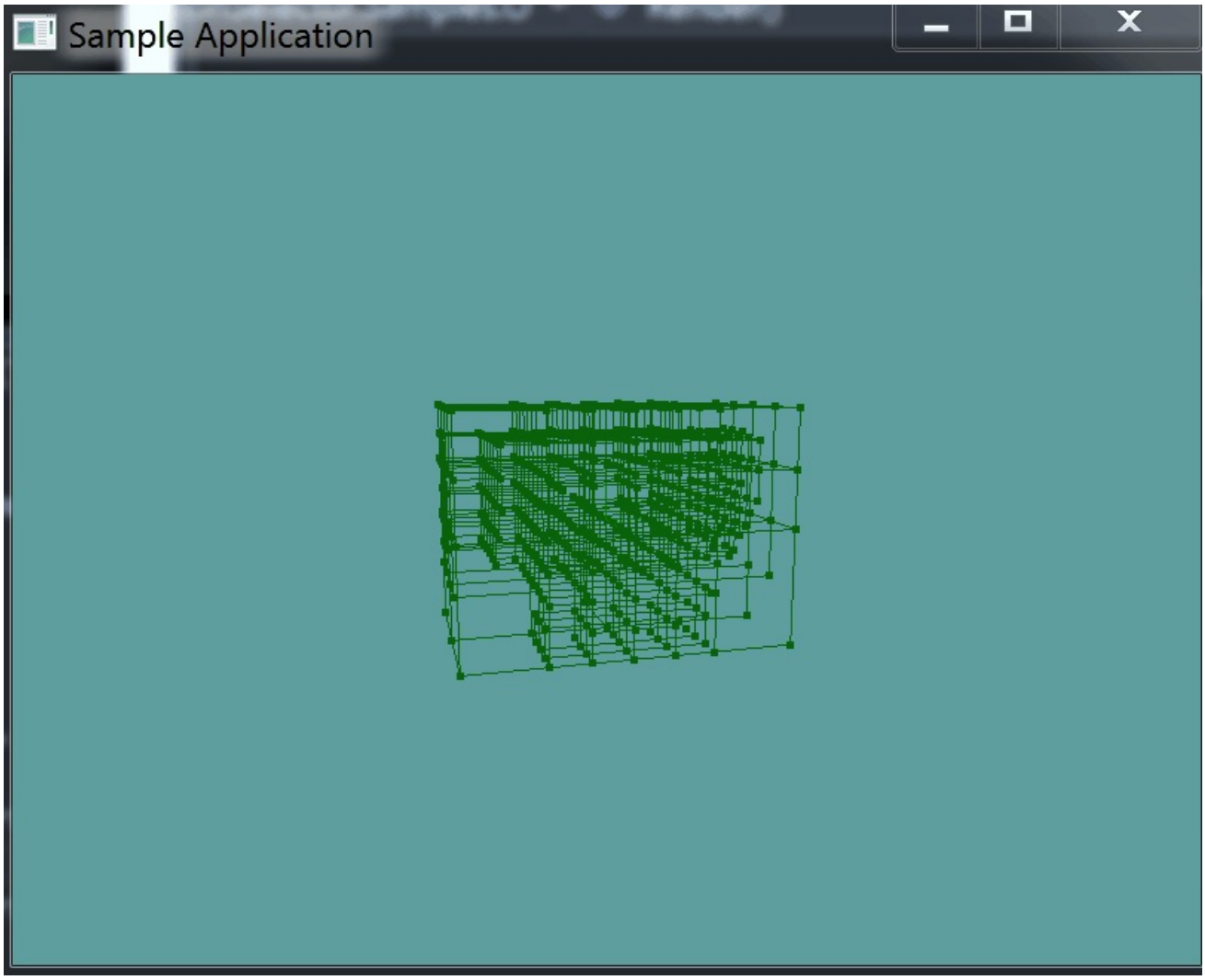
```
public void Shake() {
    if (Children != null) {
        for (int i = Children.Count - 1; i >= 0; --i) {
            if (Children[i].TriangleCount == 0) {
                Children.RemoveAt(i);
            }
        }
        foreach(BVHNode child in Children) {
            child.Shake();
        }
    }
}
```

Now, go ahead and call the Shake function in `ObjLoader`, right after the split function was called! After all the BVH functions have been added to the `OBJLoader` constructor, this is what the relevant section looks like for me:

```
bvhRoot = new BVHNode(containerAABB);

for (int i = 0; i < vertexData.Count; i += 9) {
    collisionMesh[meshCounter++] = new Triangle(
        new Point(vertexData[i + 0], vertexData[i + 1], vertexData
[i + 2]),
        new Point(vertexData[i + 3], vertexData[i + 4], vertexData
[i + 5]),
        new Point(vertexData[i + 6], vertexData[i + 7], vertexData
[i + 8])
    );

    bvhRoot.Triangles.Add(collisionMesh[meshCounter - 1]);
}

bvhRoot.Split();
bvhRoot.Shake();
```

Test it

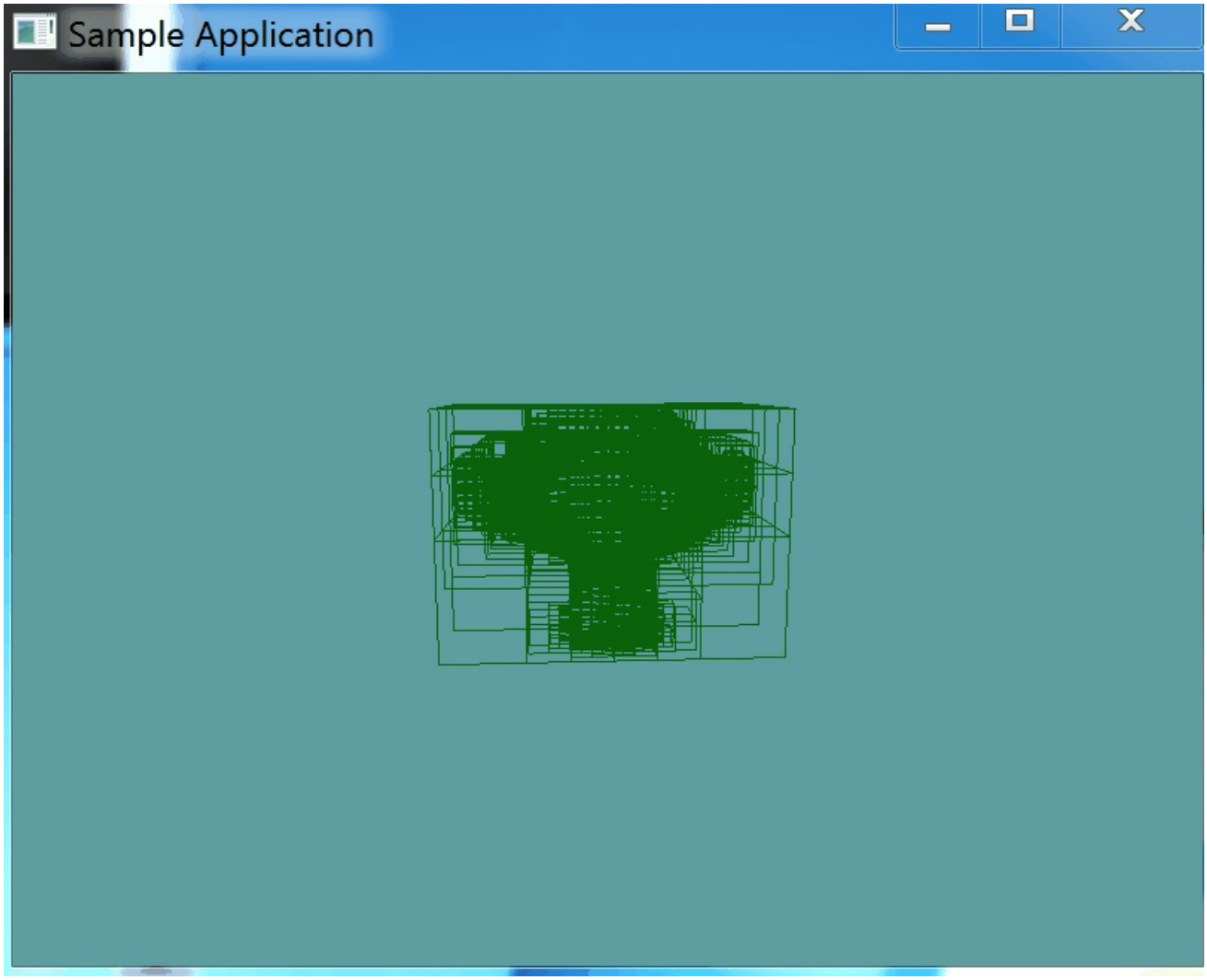
Running the game, the window should now display this:



You can clearly see how the tree got sparsed out. The shape of the mesh is even more visible now.

More detail

For our monkey model, a subdivision depth of 3 will suffice. The bigger a model, the deeper the subivision needs to be. The deeper the subdivision, the more expensive the BVH becomes. Just for fun, let's see what would happen if we turned the subdiv depth up from 3 to 5:



Now you can REALLY tell the shape of the model. Let's keep the max depth at 3 for now, 5 is more appropriate for things like terrain. I've NEVER seen a max depth greater than 8.

Raycast

Now, for the moment of truth, let's implement a raycast function! Well, we're not really going to implement a new one, as much as we're going to modify the existing `Raycast(Ray ray, OBJ model, out float t)` function.

Expose the BVH Tree

Go into the `OBJ` class, and expose the `BVHRoot` of the `OBJLoader` that's contained inside the `OBJ` class.

```
public BVHNode BVHRoot {
    get {
        return model.BvhRoot;
    }
}
```

Collisions

Just so we're on the same page, this is what my raycast function currently looks like:

```
public static bool Raycast(Ray ray, OBJ model, out float t) {
    Matrix4 inverseWorldMatrix = Matrix4.Inverse(model.WorldMatrix
);
    Ray newRay = new Ray(ray.Position, ray.Normal);
    newRay.Position = new Point(Matrix4.MultiplyPoint(inverseWorld
Matrix, ray.Position.ToVector()));
    newRay.Normal = Matrix4.MultiplyVector(inverseWorldMatrix, ray
.Normal);

    if (!Raycast(newRay, model.BoundingSphere, out t)) {
        return false;
    }
    if (!Raycast(newRay, model.BoundingBox, out t)) {
        return false;
    }

    foreach(Triangle triangle in model.Mesh) {
        if (Raycast(newRay, triangle, out t)) {
            return true;
        }
    }

    return false;
}
```

Now, what we want to do is replace that triangle raycast loop with a raycast against a BVH node. That's going to be a new function.

Raycast BVHNode

Raycasting against a BVH node is a recursive process. First, we check if the ray being passed in hits the AABB of the node. If it does not, we just return.

Next, we loop trough the children of the node (if it's not a leaf node, and has children). We call the raycast function on each of it's children, this is the recursive part. If any of those result in true, we return true.

Finally, if the node was a leaf, we loop trough all of it's triangles. If any of the triangles hit, return true.

By default we return false.

```
public static bool Raycast(Ray ray, BVHNode node, out float t) {
    if (!Raycast(ray, node.AABB, out t)) {
        return false;
    }

    if (node.Children != null) {
        foreach(BVHNode child in node.Children) {
            if (Raycast(ray, child, out t)) {
                return true;
            }
        }
    }

    if (node.Triangles != null) {
        foreach (Triangle triangle in node.Triangles) {
            if (Raycast(ray, triangle, out t)) {
                return true;
            }
        }
    }

    return false;
}
```

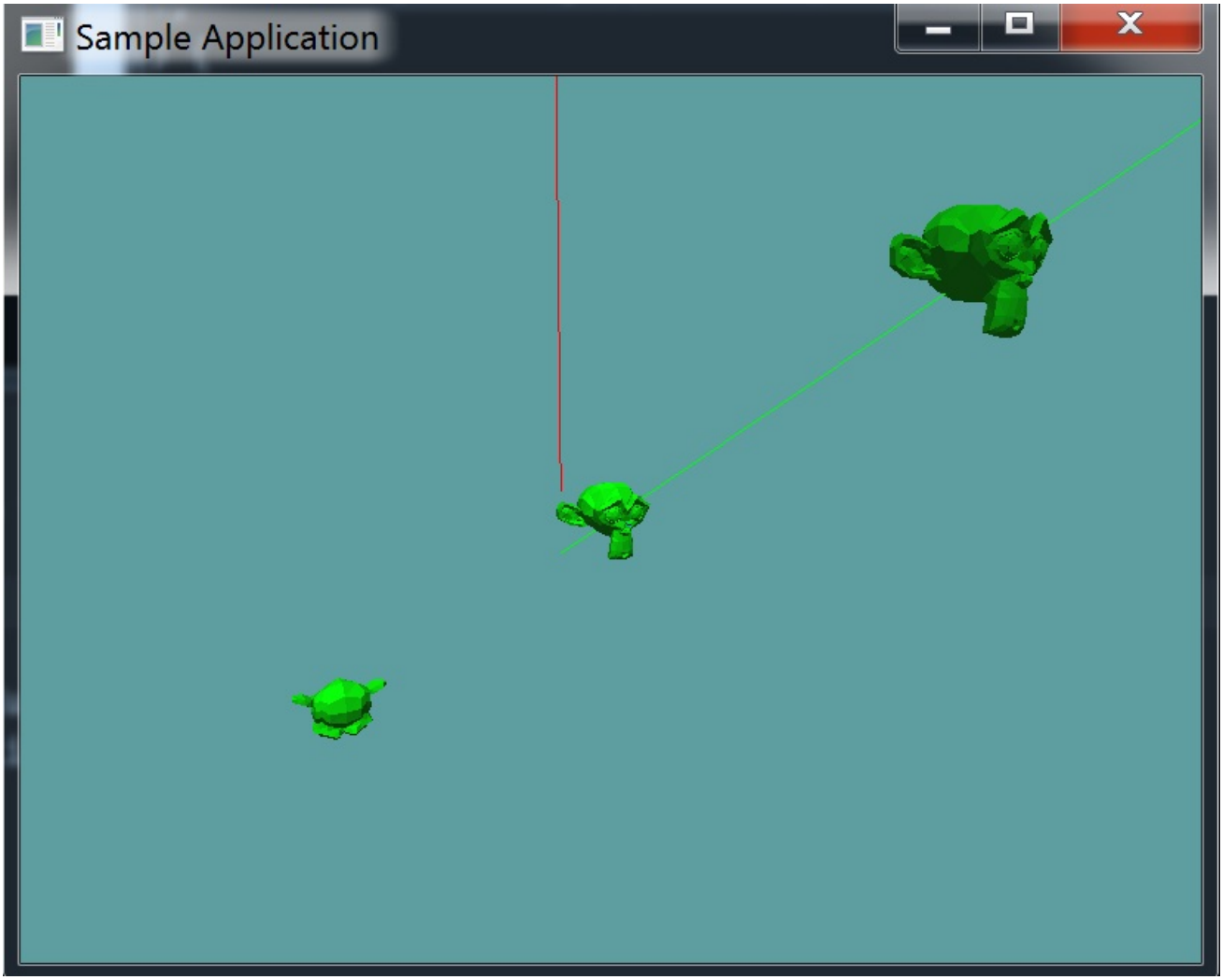
Raycast OBJ

Now we can go back to the raycast obj function, and eliminate its triangle loop. Make sure to pass `newRay` into the Raycast-BVHNode function

```
public static bool Raycast(Ray ray, OBJ model, out float t) {
    Matrix4 inverseWorldMatrix = Matrix4.Inverse(model.WorldMatrix
);
    Ray newRay = new Ray(ray.Position, ray.Normal);
    newRay.Position = new Point(Matrix4.MultiplyPoint(inverseWorld
Matrix, ray.Position.ToVector()));
    newRay.Normal = Matrix4.MultiplyVector(inverseWorldMatrix, ray
.Normal);

    if (!Raycast(newRay, model.BoundingSphere, out t)) {
        return false;
    }
    if (!Raycast(newRay, model.BoundingBox, out t)) {
        return false;
    }

    return Raycast(newRay, model.BVHRoot, out t);
}
```

Test it!

Now you can test that we didn't break anything! Just load up and run the existing `OBJRaycast` unit test. With any luck, nothing has changed! It should look like this:



Even tough visually nothing has changed, the actual raycast test is much faster now, we probably saved about 5 to 10 milliseconds / raycast. Which in game time is HUGE!

Scene Management

We've covered scenes before with other games, but they are important, so we are going to re-visit them. A basic scene is just a hierarchy of transform matrices. We have two options to implement this.

We can either implement a component system, where we have a base game object that gets components such as meshes and scripts attached.

Or we can hard-code what belong into a scene.

The component system is preferable, but hard-coding is much faster. Back in the Doom days, hard-coding used to be the standard, not so much today.

Non-the-less we are going to hard code our scene, mainly because we're aiming for fast development time, and this isn't code that's going to ship.

Transform hiearchy

Our scene is going to consist of only OBJ objects. They have something to display, and they already have a sense of "position". We're going to change the OBJ class into a tree. Essentially, this will give us a transform hierarchy.

Changine the OBJ class into nodes

Having a tree is all about having child and parent nodes. The first thing we're going to do is add a Parent reference, and a list of children to the OBJ class.

```
public OBJ Parent = null;
public List<OBJ> Children = new List<OBJ>();
```

TODO: Now, adjust the Render, RenderBVH and RenderDebug functions to loop trough all children of the OBJ and call the appropriate Render function. Do this loop after the PopMatrix, each child has it's own world matrix, and will set it for its-self.

Also, we're going to have OBJ's with a null model, be sure to add null checks. If an OBJ's internal model variable is null it will not render, but all of its children still should.

Transform hierarchy

We're also going to make the OBJ's form a transform hierarchy. So, if you move one of the parent obj's all of it's children will move. In order to do this, the first thing we need to do is clear up what is in world-space and what is in object space.

The **WorldMatrix** getter is obviously in world space. It is an absolute position, relative to world 0, regardless of the parent of the node.

The **Position**, **Rotation** and **Scale** getters / setters are going to be in local space. That is, they are relative to the node's parent. If the parent's position is `0, 1, 2` and the node's local position is `1, 0, 1` then the world position of the node will be

```
1 1 3
```

Because we've now introducted the concept of an "Empty" OBJ IE: Once without a mesh, let's add a getter to check if an OBJ is empty or not

```
public bool IsEmpty {
    get {
        return model == null;
    }
}
```

Because we work in world and local space now, when the transform of a parent node changes, all children must be updated. We almost have a method of doing this already, each object has a dirty flag.

All we have to do is make the dirty flag recursive. If a node is marked as dirty, all it's children will need to be marked dirty as well.

Change

```
protected bool dirty = true;
```

To

```
protected bool dirtySelf = true;

protected bool dirty {
    get {
        return dirtySelf;
    }
    set {
        dirtySelf = value;
        if (value) {
            foreach(OBJ child in Children) {
                child.dirty = true;
            }
        }
    }
}
```

With the use of a protected getter we made the dirty flag recursive. That is, whenever an OBJ is marked as dirty, all it's children, and all of that nodes children will be marked dirty as well. This will force every child of the node to re-calculate it's own world matrix, the next time render is called.

Applying the transform

The last thing we need to do here is make sure the world matrix takes the parents world matrix into account. We can achieve this, by multiplying the world matrix by the parent nodes world matrix

```
public Matrix4 WorldMatrix {
    get {
        if (dirty) {
            Matrix4 translation = Matrix4.Translate(position);

            Matrix4 pitch = Matrix4.XRotation(rotation.X);
            Matrix4 yaw = Matrix4.YRotation(rotation.Y);
            Matrix4 roll = Matrix4.ZRotation(rotation.Z);
            Matrix4 orientation = roll * pitch * yaw;

            Matrix4 scaling = Matrix4.Scale(scale);

            worldMatrix = translation * orientation * scaling;

            if (Parent != null) {
                worldMatrix = Parent.WorldMatrix * worldMatrix;
            }

            // DO NOT FORGET THIS! The first version
            // I wrote, i forgot to clear the dirty flag
            // and my game stopped running!
            dirty = false;
        }
        return worldMatrix;
    }
}
```

As you can see, not much has actually changed, we just added another matrix multiplication. Important to note, we are multiplying by the parent's world matrix getter `Parent.WorldMatrix`, using a capital W!

Scene

Currently we have everything set up to create a transform hierarchy, but not a scene. A transform hierarchy is just a part of a scene. The scene provides a single interface for managing this crazy hierarchy. Let's make a new `Scene` class and implement a basic scene that will render it's transform hierarchy and perform a ray cast on it.

```
using OpenTK.Graphics.OpenGL;
using System.Collections.Generic;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector {
    class Scene {
        public OBJ RootObject = new OBJ(null);

        public void Render() {
            RootObject.Render();
        }

        public OBJ Raycast(Ray ray, out float t) {
            return RecursiveRaycast(RootObject, ray, out t);
        }

        public OBJ Raycast(Ray ray) {
            float t = 0.0f;
            return RecursiveRaycast(RootObject, ray, out t);
        }
}
```

We have a null object at the root of the scne. This is conveniant because now every single object will form a tree. The rendering function is fairly straight forward, Because `OBJ.Render` is already recursive, so we just call it on the root object.

Raycast however, we are not so lucky with. Implement the

```
protected OBJ RecursiveRaycast(OBJ current, Ray ray, out float t)
{
```

helper function on your own. It should perform a ray cast against the current Obj being passed in. If the result of that ray cast is true, return the `OBJ`. If the result of that ray cast is false, recursively call this function on all children of the `OBJ`. If any of the children hit something, return the `OBJ` which was hit. If nothing was hit, return `null`;

We're going to have to modify the raycast obj function, in Collisions.cs, this one:

```
  public static bool Raycast(Ray ray, OBJ model, out float t) {
```

Change the function so it first checks if the model is empty `model.IsEmpty`, if it is, set t to -1 and return false by default.

The Point

Now that we have a scene class, we no longer care about the indevidual OBJ objects! Right now we're hard coding the new objects, but ideally you'd want to make the scene hierarchy dynamically, by reading in some text file.

That's the whole point of the scene. We no longer care about indevidual objects, we just perform operations on the scene its-self! And the scene knows about objects, and how to call our existing code that's written for them

Test it out!

It's time to test out your work. Use the following unit test code:

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
```

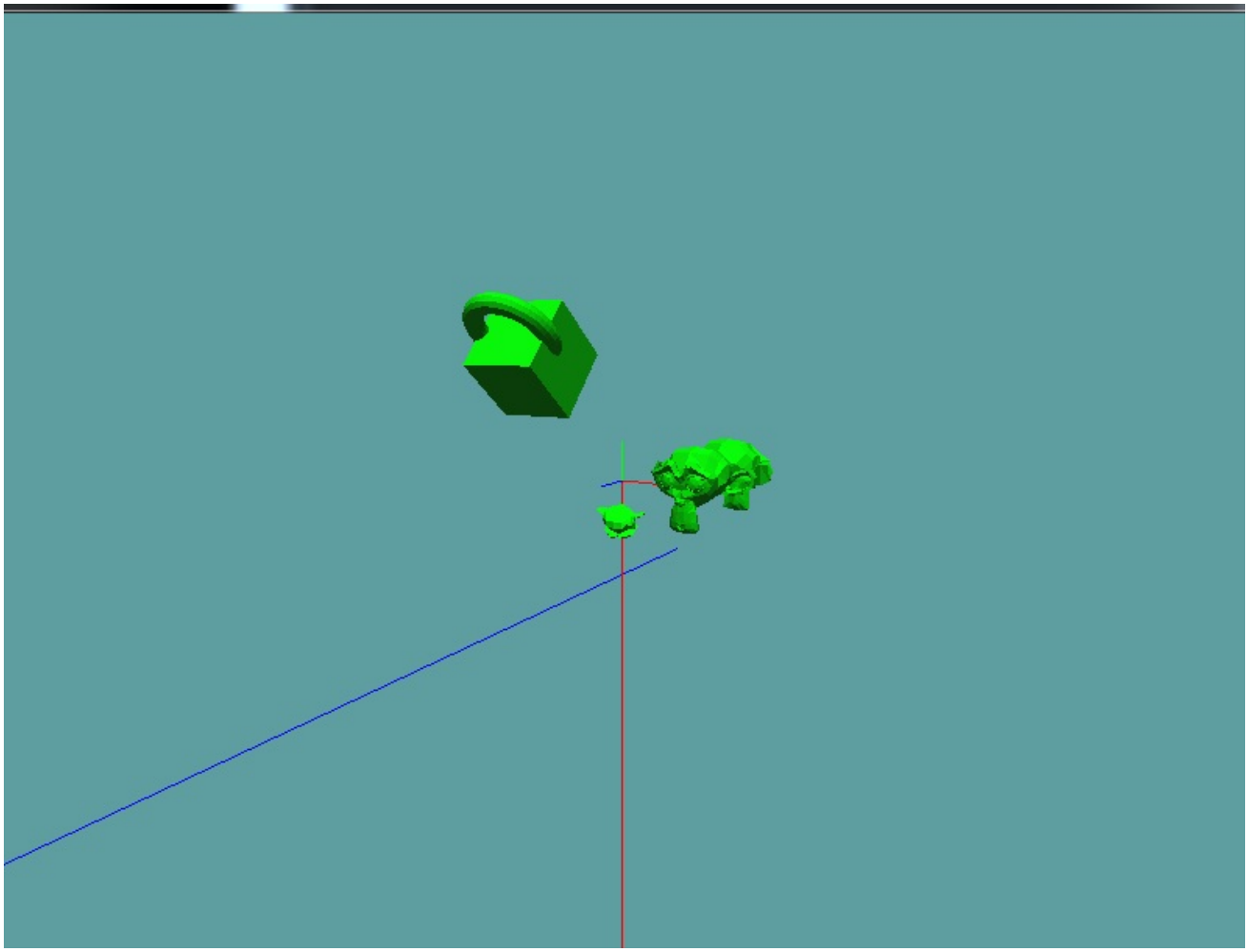
```
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class SceneSample01 : Application {
        Scene scene = new Scene();
        OBJ cubeNode = null;

        OBJLoader suzane = null;
        OBJLoader cube = null;
        OBJLoader torus = null;

        Ray[] rays = new Ray[] {
            new Ray(new Point(0.0f, 0.0f, 0.0f), new Vector3(0.0f,
-1.0f, 0.0f)),
            new Ray(new Point(-1f, -3f, -5f), new Vector3(0f, 0f, 1
f))
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);
            GL.PointSize(5f);

            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 });

            suzane = new OBJLoader("Assets/suzanne.obj");
            cube = new OBJLoader("Assets/cube.obj");
```

```
            torus = new OBJLoader("Assets/torus.obj");

            OBJ node = new OBJ(suzane);
            node.Parent = scene.RootObject;
            node.Parent.Children.Add(node);
            node.Position = new Vector3(2.0f, 0.0f, 0.0f);

            node = new OBJ(suzane);
            node.Parent = scene.RootObject.Children[0]; // suzane
reference
            node.Parent.Children.Add(node);
            node.Position = new Vector3(-2.0f, -1.0f, 0.0f);
            node.Scale = new Vector3(0.5f, 0.5f, 0.5f);
            node.Rotation = new Vector3(90.0f, 0f, 0f);

            node = new OBJ(suzane);
            node.Parent = scene.RootObject.Children[0]; // suzane
reference
            node.Parent.Children.Add(node);
            node.Position = new Vector3(0.0f, 0.0f, -2.0f);
            node.Rotation = new Vector3(0f, 180f, 0f);

            node = new OBJ(cube);
            node.Parent = scene.RootObject;
            node.Parent.Children.Add(node);
            node.Position = new Vector3(-2.0f, 3.0f, 1.0f);
            node.Rotation = new Vector3(45.0f, 0.0f, 0.0f);
            cubeNode = node;

            node = new OBJ(torus);
            node.Parent = cubeNode;
            node.Parent.Children.Add(node);
            node.Position = new Vector3(0.0f, 0.0f, -1.0f);

            bool[]  b_res = new bool[] { true, false };
            float[] t_res = new float[] { 2.730469f, 0f } ;
```

```
            float t = 0.0f;
            for (int i = 0; i < rays.Length; ++i) {
                bool result = scene.Raycast(rays[i], out t) != null
;
                if (result != b_res[i]) {
                    System.Console.WriteLine("ray " + i + ", expec
ted: " + b_res[i].ToString() + ", got: " + result.ToString());
                }
                if (!CMP(t, t_res[i])) {
                    System.Console.WriteLine("ray " + i + "t, expe
cted: " + t_res[i].ToString() + ", got: " + t.ToString());
                }
            }
        }

        bool CMP(float x, float y) {
            return System.Math.Abs(x - y) < 0.00001f;
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Enable(EnableCap.Lighting);
            scene.Render();
            GL.Disable(EnableCap.Lighting);

            foreach (Ray r in rays) {
                if (scene.Raycast(r) != null) {
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 0f, 1f);
                }
                r.Render();
            }
        }
```

```
        public override void Update(float deltaTime) {
            base.Update(deltaTime);
            Vector3 cubeRotation = cubeNode.Rotation;
            cubeRotation.X += 45.0f * deltaTime;
            cubeNode.Rotation = cubeRotation;
        }
    }
}
```

Your final scene should look like this, with the cube and torus rotating:



Octree

An octree is a tree in which every node is either a leaf or has 8 children. It's very similar to the BVH acceleration structure we implemented for model raycasting. Like the BVH, the octree is also an acceleration structure.

Unlike the BVH structure, Octree's are not usually sparse. That means we don't shake the tree. Every node has 8 children or no children. There are many octree variations out there, we are going to implement the simplest one.

There are schemes where an object belongs to multiple octre nodes, schemes where octree node boundries are fuzzy, and some really crazy implementations i can't follow. What makes our implementaiton simple is that each object will belong to exactly 1 octree node. This is not the most efficient octree implementation, but it's the basic one.

Acceleration structure

We're going to explore how the octree can be used as an acceleration structure to speed up raycasting into our scene. The idea is almost the same as the BVH we created earlyer.

We do a raycast, if the ray hits one of the octree nodes. If the ray hits a node, we raycast against all models inside the node. If any of those models where hit by the ray, we return the model. If not, we recursivley call raycast on all children nodes of the octree node.

Inserting

We insert objects into the tree based on either the objects Bounding Box or the objects Bounding Sphere. Usually you want to use the bounding box, but if you do that there is additional logic required for rotated objects.

That is, whenever an object rotates, you have to re-calculate it's bounding box. I prefer saving that performance, and sacraficing a little bit of performance in the raycast accuracy instead. So our implementation will use the bounding sphere.

Updating

Whenever an object changes position (The "Is Dirty" flag is set) there is a chance that it migrated into a new node of the octree. Should this happen, we have to remove the object from the node it currently belongs to, and try to insert it into the parent of that node. If that insertion fails, we recursivley walk the tree and keep trying to insert into a new parent until we reach the top of the tree.

Other Uses

An octree is obviously not only used for raycasting. Normally, all scene collision operations go trough the octree. Most functions in the scene should never touch the object hierarchy directly, they should be operating on the octree.

We will see this in action in the next chapter, where we are going to utilize the octree to perform Frustum Culling. For now, let's get started and actually implement the octree.

Octree Node

A tree is made up of many nodes, the octree is no different. There is no specific `Octree` class, just an `OctreeNode` within our scene that servers as the root of the tree.

Create Node

Let's start off with the definition of a node. A node contains an Axis Aligned Bounding Box, this is the area of the node. A reference to it's parent (We are in a tree after all), and a list of it's children. The node also contains a list of objects it can potentially hold.

If a node has children, it will not have contents. If a node has contents, it will not have children. This makes determining what is and isn't a leaf node fairly simple. Leaf nodes have no children, therefore `Children` will be null.

The constructor will, by default make each node a leaf node. It isn't until later, when we split the node that it can become a non-leaf.

```
using System;
using System.Collections.Generic;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;

namespace CollisionDetectionSelector {
    class OctreeNode {
        public AABB Bounds = null;
        public OctreeNode Parent = null;
        public List<OctreeNode> Children = null;
        public List<OBJ> Contents = null;

        public OctreeNode(Point position, float halfSize, OctreeNo
de parent) {
            Bounds = new AABB(
                new Point(position.X - halfSize, position.Y - half
Size, position.Z - halfSize),
                new Point(position.X + halfSize, position.Y + half
Size, position.Z + halfSize)
            );
            Parent = parent;
            Contents = new List<OBJ>();
            // Keep children null, this is a leaf node until the "
Split" function is called
        }
    }
}
```

Split Node

The split node function will invalidate the current octree. Therefore it MUST be called BEFORE any objects are inserted into the tree. Much like the BVH version, our split method is going to be recursive.

Unlike the BVH however, the user will have control over how many times a node gets split! The function takes a single integer, the number of times to split the node. This integer is decremented by 1 for each recursive call.

```
public void Split(int level) {
    // Bounds.Extens.x is the same as y and z, all nodes are square

    // We want each child node to be half as big as this node
    float splitSize = Bounds.Extents.X / 2.0f;

    // Also, the constructor of the Octree node takes in the CENTER

    // position of the new node, therefore we want to create new
    // nodes from the center of this node + or - 1/4 the size of
    // this node. This is similar to BVH, but different in size.
    Vector3[] childPattern = new Vector3[] {
        new Vector3(+1f, -1f, -1f), // Right, Top, Front
        new Vector3(+1f, -1f, +1f), // Right, Top, Back
        new Vector3(+1f, +1f, -1f), // Right, Bottom, Front
        new Vector3(+1f, +1f, +1f), // Right, Bottom, Back
        new Vector3(-1f, -1f, -1f), // Left, Top, Front
        new Vector3(-1f, -1f, +1f), // Left, Top, Back
        new Vector3(-1f, +1f, -1f), // Left, Bottom, Front
        new Vector3(-1f, +1f, +1f), // Left, Bottom, Back
    };

    // Make the actual child nodes
    Children = new List<OctreeNode>();
    Contents = null; // We are no longer a leaf node!
    foreach (Vector3 offset in childPattern) {
        // Remember to account for the center of the current node
        Point position = new Point(Bounds.Center.ToVector() + offs
et * splitSize);
        OctreeNode child = new OctreeNode(position, splitSize, this
);
        Children.Add(child);
    }
```

```
    // If we have not reached the max split depth yet,
    // go ahead and recursivley call this method for all children
    if (level > 1) {
        foreach(OctreeNode child in Children) {
            child.Split(level - 1);
        }
    }
}
```

Insert Object

Inserting an object is fairly straight forward. The object will only insert if the Bounds of the current node intersect it. If that's not the case, none of the children nodes will intersect the object either!

^ The above statement only holds true if the ABB-OBJ test is implemented as a SAT test. Ours is not, this is why we are going to be testing the bounding sphere, instead of the OBJ its-self. That way, our Octree insertion is a series of Sphere-AABB tests.

The one catch here is that `OBJ.BoundingSphere` is local space, not world space! So, we need to fix that. This is a simple translation we've already made when handling the BVH code before.

```
public bool Insert(OBJ obj) {
    // Remember, the bounding sphere is model space, we need it to
    // be world space! Multiply the position and scale the radius
    Sphere worldSpaceSphere = new Sphere();
    worldSpaceSphere.Position = new Point(
        Matrix4.MultiplyPoint(obj.WorldMatrix, obj.BoundingSphere.
Position.ToVector())
    ); // End new point
    float scale = Math.Max(obj.WorldMatrix[0, 0], Math.Max(obj.Wor
ldMatrix[1, 1], obj.WorldMatrix[2, 2]));
    worldSpaceSphere.Radius = obj.BoundingSphere.Radius * scale;

    if (Collisions.Intersects(worldSpaceSphere, Bounds)) {
        // We knoe the obj intersects this node, if it's a leaf
        // add it to the object list, if not, recurse!
        if (Children != null) {
            foreach(OctreeNode child in Children) {
                child.Insert(obj);
            }
        }
        else {
            Contents.Add(obj);
        }
        return true;
    }
    return false;
}
```

Remove Object

Removing an object is super simple. If this node is a leaf node, search trough the objects it contains, and remove the object if it is in the list. If not, recursivley call the function for every node

```
public void Remove(OBJ obj) {
    if (Children != null) {
        foreach(OctreeNode child in Children) {
            child.Remove(obj);
        }
    }
    else {
        Contents.Remove(obj);
    }
}
```

Update Object

Updating an octree is a very hot topic of research. Everyone wants to have the fastest update possible. When an object moves, rotates or scales, we have to update what octree nodes it belongs to.

While there are a LOT of algorithms for this out there, we're going to implement a very basic one! Any time an object rotates or moves we simply remove it from the tree and re-insert it. This is the brute force method, it's not efficient, but it is reliable

```
public bool Update(OBJ obj) {
    Remove(obj);
    return Insert(obj);
}
```

This update method actually has a bit of a flaw. Right now, when an object is updated it is removed from the octree and then added back in. Really, this should be recursive! All of the object's children, and it's children need to be removed and inserted too! We can easily turn this function into a recursive one:

```
public bool Update(OBJ obj) {
    if (obj.Children != null) {
        foreach (OBJ child in obj.Children) {
            Update(child);
        }
    }
    Remove(obj);
    return Insert(obj);
}
```

Debug Render

It's useful to be able to visualize what we just implemented! For that reason, i say we add a `DebugRender` function. This function is simple, it just recursivley draws the bounding rectangle of each node.

```
public void DebugRender() {
    Bounds.Render();
    if (Children != null) {
        foreach (OctreeNode node in Children) {
            node.DebugRender();
        }
    }
}
```

After having finished this article, i decided it would be visually great to have the bounding spheres of objects also render so i could approximate which OctreeNode's the object is stored in. You don't have to do this, but it's a good exercise, if you want to see how i did it, the code is here

Scene Integration

Now that we have an octree node, all we have to do is add the root node to the scene, and we have an octree! I'm going to be modifying the existing scene class for this.

First, add the octree root to the scene. Next, add an initalize function. This function will take in the overall size of the octree (The half-size of it's root node). This function is responsible for creating the tree and splitting it. A split size of 3 will do for now.

```
public OctreeNode Octree = null;

public void Initialize(float octreeSize) {
    Octree = new OctreeNode(new Point(0, 0, 0), octreeSize, null);
    Octree.Split(3);
}
```

In larger AAA games, it's not uncommon to see the default octree be of **size 5000**, with a **split** level of **8**.

Next up, let's modify the render function of the scene to render the debug version of the octree

```
public void Render() {
    RootObject.Render();

    GL.Disable(EnableCap.Lighting);
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);
    Octree.DebugRender();
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);
    GL.Enable(EnableCap.Lighting);
}
```

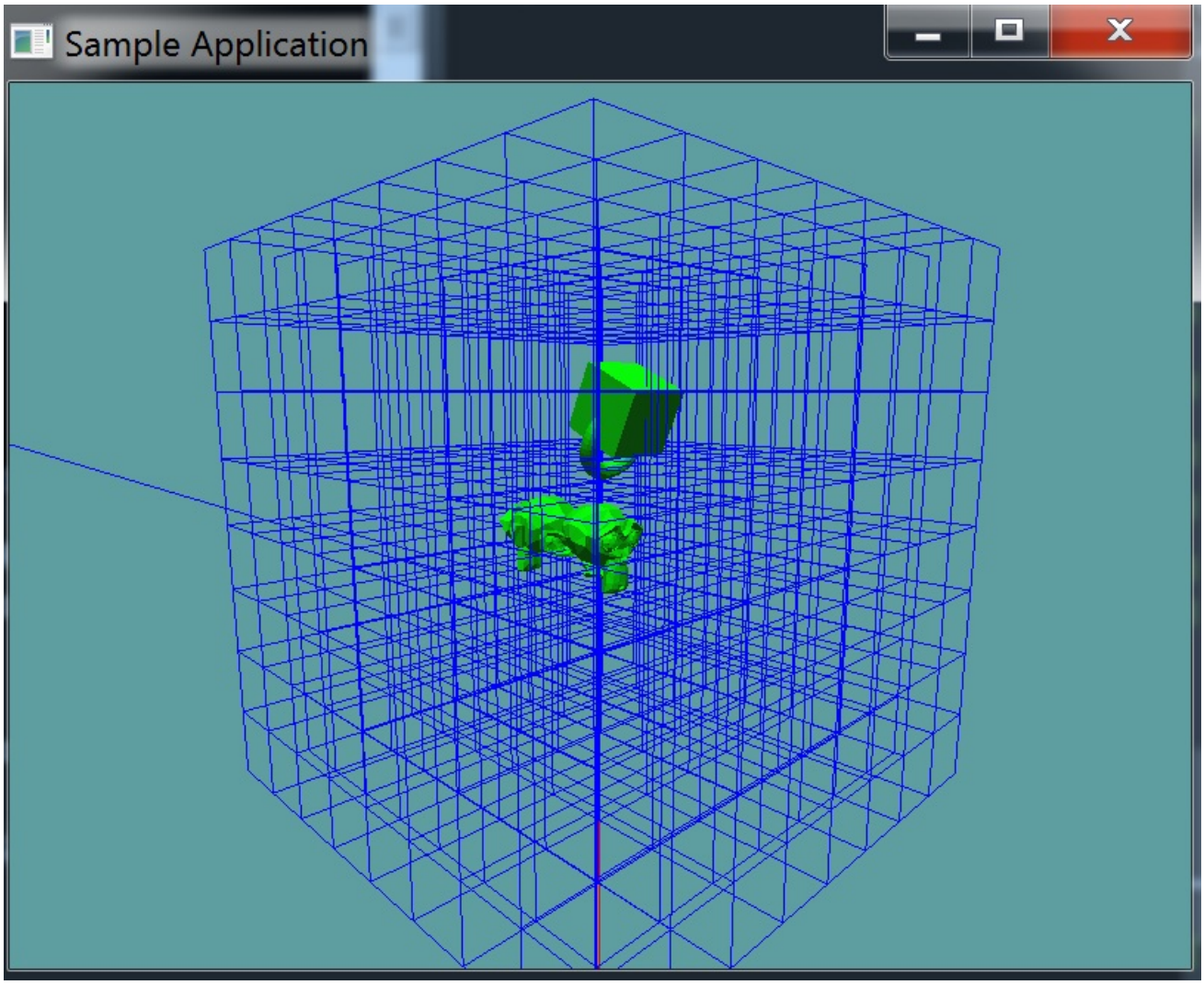
And that's it for us! In an actual game engine you would not expose the `RootObject` or `Octree`. Instead you'd have a function like `AddObject` that would create a new object and add it to both the root and octree. For our example however, this is not needed, whoever owns the scene will also update the octree.

Unit Testing

The code for this unit test is **almost** identical to the last one. The only real difference is in the `Initialize` and `Update` functions. Whenever we create a new node in the `Initialize` function, it is added to the octree. And, when we rotate the cube node insize the `Update` function we update the octree.

The rendering looks different, the octree we created will be visible. This has nothing to do with the unit test however, as we simply added the code to render the octree to the scenes render function, which is called from the unit test.

The current project should look like this:



That blue line sticking out the back is NOT a rendering artifact. It's a ray that has hit an object.

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class SceneSample02 : Application {
        Scene scene = new Scene();
        OBJ cubeNode = null;

        OBJLoader suzane = null;
        OBJLoader cube = null;
        OBJLoader torus = null;

        Ray[] rays = new Ray[] {
            new Ray(new Point(0.0f, 0.0f, 0.0f), new Vector3(0.0f,
-1.0f, 0.0f)),
            new Ray(new Point(-1f, -3f, -5f), new Vector3(0f, 0f, 1
f))
        };

        public override void Intialize(int width, int height) {
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.CullFace);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);
            GL.PointSize(5f);

            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 });

            scene.Initialize(7f);

            suzane = new OBJLoader("Assets/suzanne.obj");
            cube = new OBJLoader("Assets/cube.obj");
            torus = new OBJLoader("Assets/torus.obj");

            OBJ node = new OBJ(suzane);
            node.Parent = scene.RootObject;
            node.Parent.Children.Add(node);
            node.Position = new Vector3(2.0f, 0.0f, 0.0f);
            /* NEW */ scene.Octree.Insert(node);

            node = new OBJ(suzane);
            node.Parent = scene.RootObject.Children[0]; // suzane
reference
            node.Parent.Children.Add(node);
            node.Position = new Vector3(-2.0f, -1.0f, 0.0f);
            node.Scale = new Vector3(0.5f, 0.5f, 0.5f);
            node.Rotation = new Vector3(90.0f, 0f, 0f);
            /* NEW */ scene.Octree.Insert(node);

            node = new OBJ(suzane);
            node.Parent = scene.RootObject.Children[0]; // suzane
reference
            node.Parent.Children.Add(node);
            node.Position = new Vector3(0.0f, 0.0f, -2.0f);
            node.Rotation = new Vector3(0f, 180f, 0f);
            /* NEW */ scene.Octree.Insert(node);

            node = new OBJ(cube);
            node.Parent = scene.RootObject;
            node.Parent.Children.Add(node);
            node.Position = new Vector3(-2.0f, 3.0f, 1.0f);
            node.Rotation = new Vector3(45.0f, 0.0f, 0.0f);
```

```
            cubeNode = node;
            /* NEW */ scene.Octree.Insert(node);

            node = new OBJ(torus);
            node.Parent = cubeNode;
            node.Parent.Children.Add(node);
            node.Position = new Vector3(0.0f, 0.0f, -1.0f);
            /* NEW */ scene.Octree.Insert(node);

            bool[]  b_res = new bool[] { true, false };
            float[] t_res = new float[] { 2.730469f, 0f } ;

            float t = 0.0f;
            for (int i = 0; i < rays.Length; ++i) {
                bool result = scene.Raycast(rays[i], out t) != null
;
                if (result != b_res[i]) {
                    System.Console.WriteLine("ray " + i + ", expec
ted: " + b_res[i].ToString() + ", got: " + result.ToString());
                }
                if (!CMP(t, t_res[i])) {
                    System.Console.WriteLine("ray " + i + "t, expe
cted: " + t_res[i].ToString() + ", got: " + t.ToString());
                }
            }
        }

        bool CMP(float x, float y) {
            return System.Math.Abs(x - y) < 0.00001f;
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Enable(EnableCap.Lighting);
            scene.Render();
```

```
            GL.Disable(EnableCap.Lighting);

            foreach (Ray r in rays) {
                if (scene.Raycast(r) != null) {
                    GL.Color3(1f, 0f, 0f);
                }
                else {
                    GL.Color3(0f, 0f, 1f);
                }
                r.Render();
            }
        }

        public override void Update(float deltaTime) {
            base.Update(deltaTime);
            Vector3 cubeRotation = cubeNode.Rotation;
            cubeRotation.X += 45.0f * deltaTime;
            cubeNode.Rotation = cubeRotation;

            /* NEW */ scene.Octree.Update(cubeNode);
        }
    }
}
```

Raycasting

This is where we see the acceleration power of the octree! Like with the BVH, we're going to re-direct raycasting against a scene to actually raycast against the octree. Also like BVH, this is the only intersection test we're going to accelerate. But do keep in mind, in a AAA engine, all intersections would happen trough the octree, not just raycasting.

Implementation

Inside the `OctreeNode` class, we're going to implement a recursive raycast method. The implementation of this method is fairly straight forward.

First, we're going to take care of the recursive part. If the node has children, we loop trough all it's children. We check the ray against each of the AABB's of the child nodes. If the ray does not hit the bounds of the child node, it can't hit anything in the child. If the ray does intersect with the bounds of the child node, we recursivley call this same `Raycast` function on the child node.

The recursive part only applies to non-leaf nodes. After that, let's handle the leaf node scenario. If a node is a leaf node, we already know it passed the ray-bounds test, so we just loop trough all the OBJ contents of the node. If the ray hits any of the OBJ's contained within this node, we return said OBJ.

By default, return null!

```
public OBJ Raycast(Ray ray, out float t) {
    // This is NOT a leaf node, see which child can be raycast aga
inst!
    if (Children != null) {
        foreach(OctreeNode child in Children) {
            // Does the ray intersect the AABB of the node in ques
tion?
            if (Collisions.Raycast(ray, child.Bounds, out t)) {
                // If so, recursivley call the raycast method!
                OBJ result = child.Raycast(ray, out t);
                // If we hit something, return it!
                if (result != null) {
                    return result;
                }
            }
        }
    }
    // This IS a leaf node! See if any of the contents are hit by
the ray.
    // If we made it this far, then the Bounds already intersect t
he ray,
    // no bounds check for the OctreeNode is needed!
    else if (Contents != null) {
        // Loop trough all children
        foreach(OBJ content in Contents) {
            // Return the first child that was hit!
            if (Collisions.Raycast(ray, content, out t)) {
                return content;
            }
        }
    }

    t = 0f;
    return null;
}
```

Changing the scene

Now that we have a raycast method implemented in the `OctreeNode` class, we no longer need the implementation in the `Scene` class.

Go ahead and remove the `RecursiveRaycast` method from the `Scene` class. Change the two `Raycast` methods inside the `Scene` class, to just call the `Raycast` method of the octree, like so:

```
public OBJ Raycast(Ray ray, out float t) {
    //return RecursiveRaycast(RootObject, ray, out t);
    return Octree.Raycast(ray, out t);
}

public OBJ Raycast(Ray ray) {
    float t = 0.0f;
    //return RecursiveRaycast(RootObject, ray, out t);
    return Octree.Raycast(ray, out t);
}
```

This is actually how a scene works. It's a transform hierarchy and an octree. All of the methods inside the scene are just facads for the hierarchy and octree.

Run the last example, and nothing has changed. We still have a blue (hit detected) and red (hit not detected) ray in the scene. But, the raycast code is now much faster! It no longer loops trough all possible objects in the scene, rather it iterates over a well orgonized tree.

This might not make a huge performance difference right now, but when you get into the 10,000 object range it's really noticable!

Get Fancy

Just for the sake of fancy-ness i want to modify the octree so we can visualize exactly which nodes get traversed!

First, add a new debug bool to `OctreeNode` to let us know if the node has been visited or not:

```
protected bool debugVisited = false;
```

Next, inside the `Racyast` function of `OctreeNode` flip the new debug flag to true

```
public OBJ Raycast(Ray ray, out float t) {
    debugVisited = true;

    // This is NOT a leaf node, see which child can be raycast aga
inst!
```

Then, modify the `DebugRender` function of the `OCtreeNode` class to render the visited nodes as green:

```
 public void DebugRender() {
    if (debugVisited) {
        GL.Color3(0f, 1f, 0f);
    }
    else {
        GL.Color3(0f, 0f, 1f);

    }
    Bounds.Render();

    if (Contents != null) {
```

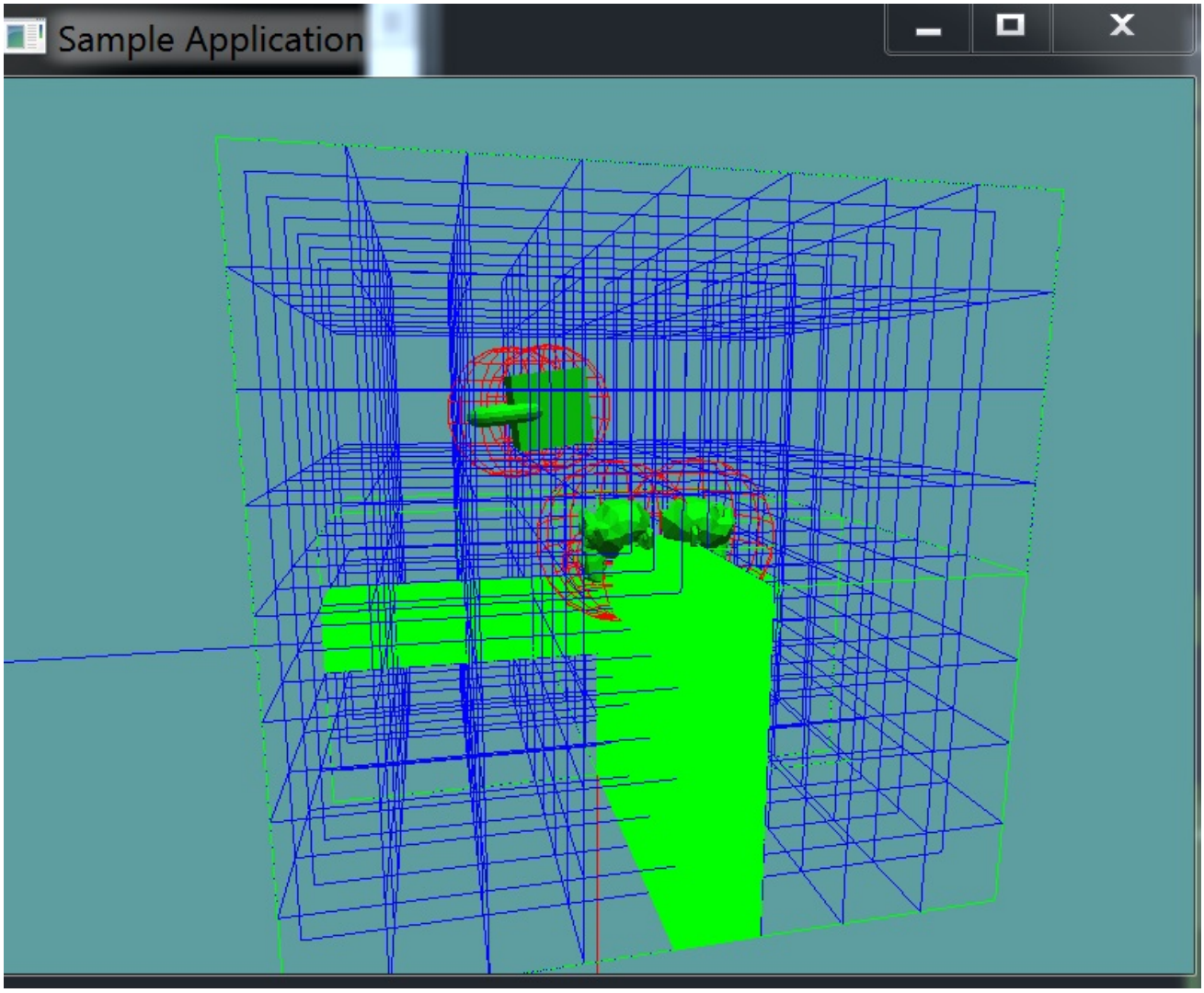
Changing the outline will not be enough. It's going to flicker, because child AABB's are drawn where the parent AABB's are present. I want to get fancy and add a new debug render method that renders only the legit visited nodes:

```
public void DebugRenderOnlyVisitedNodes() {
    if (!debugVisited) {
        return;
    }
    if (Children != null) {
        foreach (OctreeNode node in Children) {
            node.DebugRenderOnlyVisitedNodes();
        }
    }
    else {
        GL.Color3(0f, 1f, 0f);
        Bounds.Render();
    }
}
```

Finally, let's modify the `Render` method of the `Scene` class to render the visited nodes as solid boxes:

```
public void Render() {
    RootObject.Render();
    GL.Disable(EnableCap.Lighting);
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);
    Octree.DebugRender();
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);
    /* NEW */ Octree.DebugRenderOnlyVisitedNodes();
    GL.Enable(EnableCap.Lighting);
}
```

That's it! Running the last unit test should now look like this:

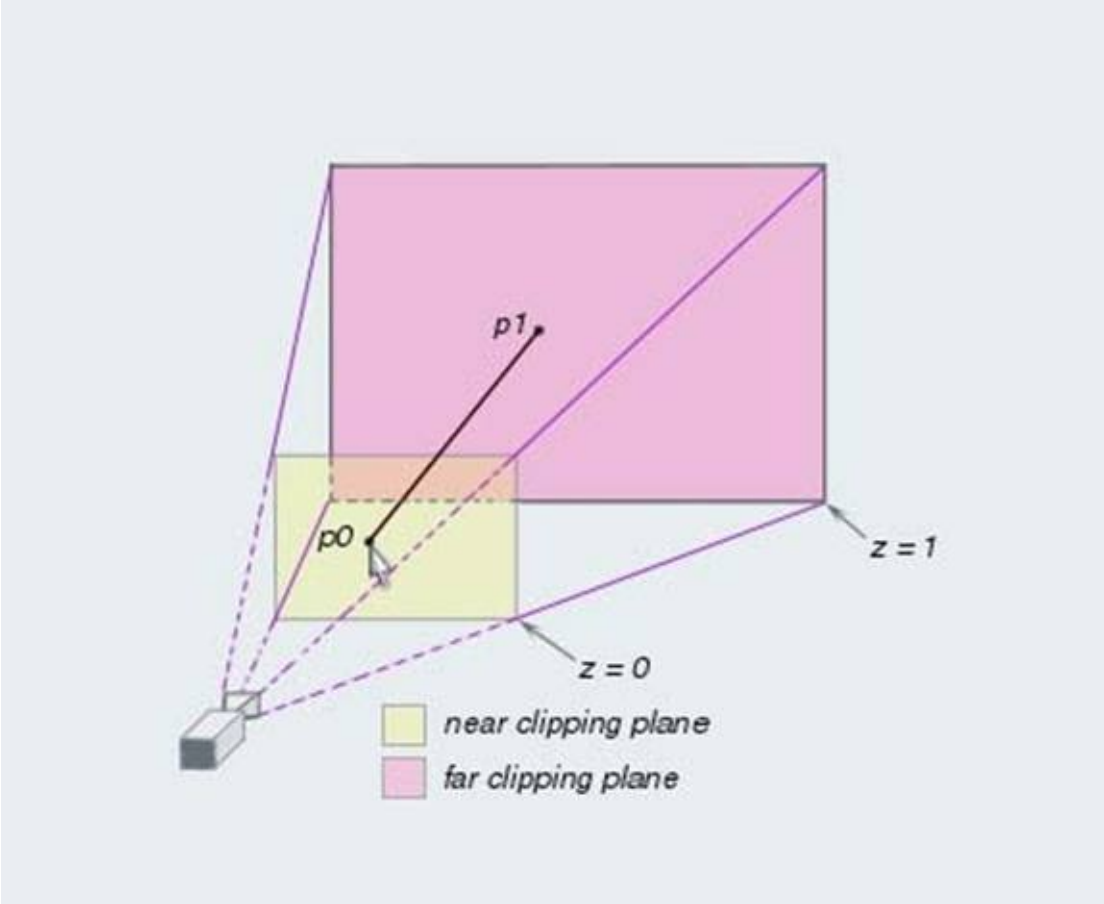


For this demo it might not make a big difference, but for a larger demo it would! Also, this is a very small Octree. Usually one octree node is large enough to contain 3 to 5 objects! The size of our entire octree is usually the size of a single node on a larger scale project.

Picking

Object picking is one of those problems that's fairly easy to solve, but for some reason no one ever knows how. What is picking? Click on a 3D scene, and the object you click on gets selected. So how to we accompolish this?

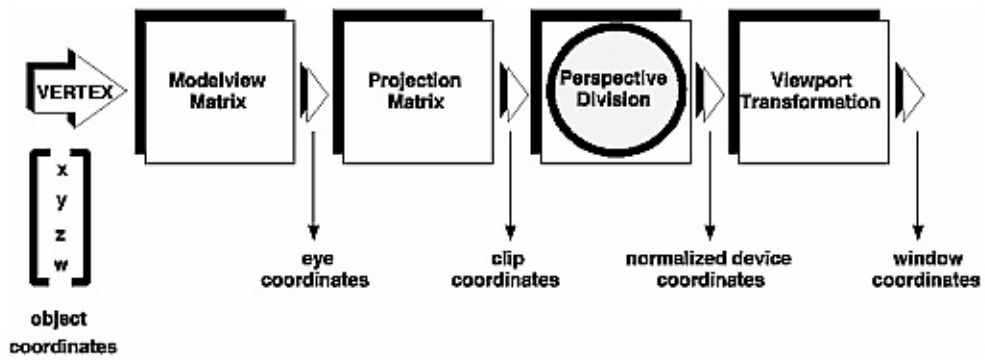
The user clicks on the screen. We find the pixel they selected on the near and far clip planes of the view frustum. We use these two points to construct a ray, then we do a raycast into the world using this ray. Whatever object collides with the ray (and has the lowest t value) is the object the user clicked on!



The entire point of the OpenGL graphics pipeline is to take a world-space point and transform it to screen space coordinates. This is done using a variety of matrix multiplications. For picking we have to do the reverse! We have to take a screen

space point and apply the reverse of the graphics pipeline to get two world space points.

Going from world space to screen space (What openGL does) is called **Projection**. Going from screen space to world space (What we need to do) is **Un-Projection**. Let's take a look at the projection pipeline:



Un-project

It's time to implement an "Unproject" function. I'm adding this to my **Matrix4.cs** file, as it is a matrix function.

The unproject function takes 4 arguments. The first one is a `Vector3` that is the screen space position of the mouse. Of course, screen space mouse only has an X and a Y coordiante! What is Z used for? Z represents WHERE in space we want to get the unprojected point. A Z of 0 is the near clip plane of the camera. A Z of 1 is the far clip plane of the camera.

Next up, we take the current view and projection matrices. These are self explanatory. Finally, we take a 4 component float array that is the screen viewport. It holds the x, y, width and height of the viewport, relative to the game window. We need this to be able to normalize the mouse screen coords.

The actual unproject pipeline works like this:

1. Normalize screen space X and Y
    - Simply divide the offset X and Y by screen Width and height
2. Transform the normalized vector into NDC Space

    - NDC space is -1 to 1, not 0 to 1 like normalized space.
3. Transform the NDC space vector into eye / view space
    - Multiply by the inverse of the projection matrix.
    - This will leave the inverse perspectiv divide in the W component
    - Remember, eye space is the world as if the camera was at it's center
4. Transform the eye space vector into world space
    - Multiply by the inverse of the modelView matrix
    - This will leave the point in proper world space
    - The W component from step 3 remains unchanged
5. Compensate for perspective division
    - Diide the X, Y and Z components by the W component acquired in step 3

I'll provide the basic implementation here

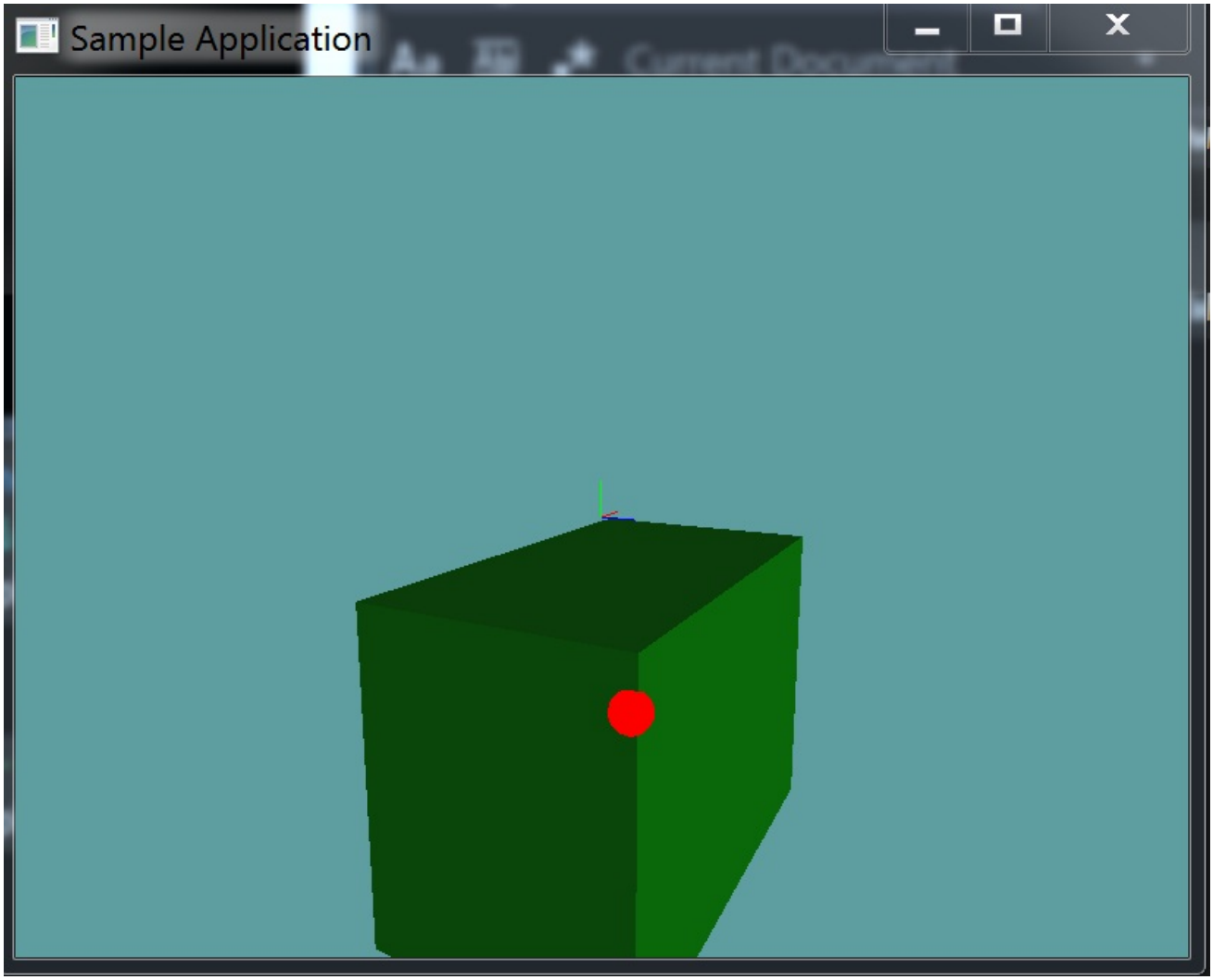
```
public static Vector3 UnProject(Vector3 windowCoords, Matrix4 mode
lView, Matrix4 projection, float[] viewPort) {
    // First, convert from window coordinates to NDC coordinates
    Vector4 ndcCoords = new Vector4(windowCoords.X, windowCoords.Y
, windowCoords.Z, 1.0f);
    ndcCoords.X = (ndcCoords.X - viewPort[0]) / viewPort[2]; // Ra
nge 0 to 1: (windowX - viewX) / viewWidth
    ndcCoords.Y = (ndcCoords.Y - viewPort[1]) / viewPort[3]; // Ra
nge 0 to 1: (windowY - viewY) / viewHeight
    // Remember, NDC ranges from -1 to 1, not 0 to 1
    ndcCoords.X = ndcCoords.X * 2f - 1f; // Range: -1 to 1
    ndcCoords.Y = 1f - ndcCoords.Y * 2f; // Range: -1 to 1 - Flipp
ed!
    ndcCoords.Z = ndcCoords.Z * 2f - 1f; // Range: -1 to 1

    // Next, from NDC space to eye / view space.
    // We get here by multiplying the inverse of the projection ma
trix
    // by NDC coords. Note, this leaves a scalar in the W componen
t!
    Vector4 eyeCoords = Matrix4.Inverse(projection) * ndcCoords;
```

```
    // Next, from eye space to world space.
    // Remember, eye space assumes the camera is at the center of
the world,
    // this is not the case, let's move the actual point into worl
d space
    Vector4 worldCoords = Matrix4.Inverse(modelView) * eyeCoords;

    // Finally, undo the perspective divide!
    // When we multiplied by the inverse of the projection matrix,
 that
    // multiplication left the inverse of the perspective divide i
n the
    // W component of the resulting vector. This could be 0
    if (Math.Abs(0.0f - worldCoords.W) > 0.00001f) {
        // This is the same as dividing every component by W
        worldCoords *= 1.0f / worldCoords.W;
    }

    // Now we have a proper 4D vector with a W of 1 (or 0)
    return new Vector3(worldCoords.X, worldCoords.Y, worldCoords.Z
);
}
```

Unit test

We're going to implement a unit test that will show you how to unproject a point. Unlike previous unit tests, we will talk about excerpts of code in this one after the test is written.

Move your mouse around the screen, when it is over the cube being rendered, a red sphere should render in world space representing WHERE on the cube your mouse is:



Before we implement the unit test, i want to change the `Render` function of the `Scene` class to take a bool. This bool will determine if debug geometry is rendered or not.

```
public void Render(bool debug) {
    RootObject.Render();
    if (debug) {
        GL.Disable(EnableCap.Lighting);
        GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line
);
        Octree.DebugRender();
        GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill
);
        Octree.DebugRenderOnlyVisitedNodes();
        GL.Enable(EnableCap.Lighting);
    }
}
```

Now, onto the actual unit!

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class SceneSample04 : Application {
        Scene scene = new Scene();

        OBJLoader cube = null;

        Sphere debugSphere = new Sphere(new Point(), 0.5f);
        Ray debugRay = new Ray();
        AABB debugBox = null;

        Matrix4 modelView = new Matrix4();
        Matrix4 projection = new Matrix4();
        float[] viewport = new float[4];

        public override void Intialize(int width, int height) {
```

```
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);
            GL.PointSize(5f);

            GL.Light(LightName.Light0, LightParameter.Position, new
 float[] { 0.5f, -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 });

            scene.Initialize(7f);

            cube = new OBJLoader("Assets/cube.obj");

            // Because the debug AABB we are actually using for pi
cking has no lighting
            // let's actually render an OBJ in the exact position
of it!
            scene.RootObject.Children.Add(new OBJ(cube));
            scene.RootObject.Children[0].Parent = scene.RootObject
;
            scene.RootObject.Children[0].Position = new Vector3(3f
, -7f, -1f);
            scene.RootObject.Children[0].Scale = new Vector3(7f, 5
f, 4f);

            // Gotta set the World space points of the debug AABB
            // to the same thing as the visual node!
            Matrix4 world = scene.RootObject.Children[0].WorldMatr
ix;
            debugBox = scene.RootObject.Children[0].BoundingBox; /
/ TEMP
            Vector3 newMin = Matrix4.MultiplyPoint(world, debugBox
```

```
.Min.ToVector());
            Vector3 newMax = Matrix4.MultiplyPoint(world, debugBox
.Max.ToVector());
            // Construct the final collision box
            debugBox = new AABB(new Point(newMin), new Point(newMa
x));
        }

        public override void Render() {
            base.Render();
            DrawOrigin();

            GL.Enable(EnableCap.Lighting);
            scene.Render(false);
            GL.Disable(EnableCap.Lighting);

            float[] rawModelView = new float[16];
            GL.GetFloat(GetPName.ModelviewMatrix, rawModelView);

            float[] rawProjection = new float[16];
            GL.GetFloat(GetPName.ProjectionMatrix, rawProjection);

            GL.GetFloat(GetPName.Viewport, viewport);

            modelView = Matrix4.Transpose(new Matrix4(rawModelView
));
            projection = Matrix4.Transpose(new Matrix4(rawProjecti
on));

            GL.Color3(1f, 0f, 0f);
            debugSphere.Render();
        }

        public override void Update(float deltaTime) {
            // Don't rotate the scene
            //base.Update(deltaTime);
```

```
            float[] viewport = new float[] { 0f, 0f, Window.Width,
 Window.Height };

            Vector3 near = Matrix4.UnProject(new Vector3(Window.Mo
use.X, Window.Mouse.Y, 0.0f), modelView, projection, viewport);
            Vector3 far = Matrix4.UnProject(new Vector3(Window.Mou
se.X, Window.Mouse.Y, 1.0f), modelView, projection, viewport);

            debugRay = new Ray(near, Vector3.Normalize(far - near)
);
            Point p = new Point();
            if (Collisions.Raycast(debugRay, debugBox, out p)) {
                debugSphere.Position = p;
            }
            else {
                // Off-screen
                debugSphere.Position = new Point(-5000, -5000, -50
00);
            }
        }
    }
}
```

Things to note

In the above example we store the Projection matrix, ModelView matrix and Viewport as member variables:

```
Matrix4 modelView = new Matrix4();
Matrix4 projection = new Matrix4();
float[] viewport = new float[4];
```

This is because we use them in the `Update` function, but we can only retrieve their values in the `Render` function. This is how we get data from one function into the other. Let's take a quick look at how they are retrieved:

```
float[] rawModelView = new float[16];
GL.GetFloat(GetPName.ModelviewMatrix, rawModelView);

float[] rawProjection = new float[16];
GL.GetFloat(GetPName.ProjectionMatrix, rawProjection);

GL.GetFloat(GetPName.Viewport, viewport);

modelView = Matrix4.Transpose(new Matrix4(rawModelView));
projection = Matrix4.Transpose(new Matrix4(rawProjection));
```

The `GL.GetFloat` method can be used to retrieve indevidual floating point values or entire arrays! The matrices are pretty self explanatory. Remember, our matrices are row major, openGL's matrices are column major. We therefore need to transpose any returned matrix.

The viewport might seem like a bit of black magic, considering we don't set it that often, but in reality it's set every time the game is resized. We just so happened to implement a helper function in the base `Application` class to do it for us. Just to avoid repetative typing. That's the retrieved viewport.

Finally, it's worth noting, we use an overload of raycast that returns a point, not t. If we just had t we could still find the point like so:

```
Point hitPoint = null;
float t = -1;
bool result = Raycast(ray, aabb, out t);
if (raycast) {
    hitPoint = new Point(ray.Position + ray.Normal * t);
}
else { // Not hit, ofscreen?
    hitPoint = new Point(-5000, -5000, -5000);
}
```

The rest of the code should be self-explanatory. The only thing that might be confusing in there is why we don't render `debugBox`, it's because `AABB`'s that we render have no normal. For the sample to look good we need geometry with normals. So, i just added a box to the root of the scene that's at the same position as the AABB we are raycasting against.

Note, this raycast did not go trough the Octree, adding support for octree-based picking is a fairly trivial task.

Other structures

Scenes can be orgonized into a veriety of structures. Octree's are a good general purpose sturcture, but many games use more specific structures that are more optimized for the type of game they are. For example, Doom3 doesn't have an octree, it uses portals. That's because the entire game is indoors, with tight corners and actual doors. Let's explore some alternatives on a high level.

PVS

PVS or Potential Visibility Set is a method that was very popular in the early days of PC-games, most of the Quake games used it. Some indie games today still use it, for it's ease of implementation.

Building a PVS is pretty simple, you divide the world into chunks using arbitrary cutting planes. Then, you place the camera at random points within this chunk and spin it around. Whatever other chunks where visible are added to the visibility set of this chunk.

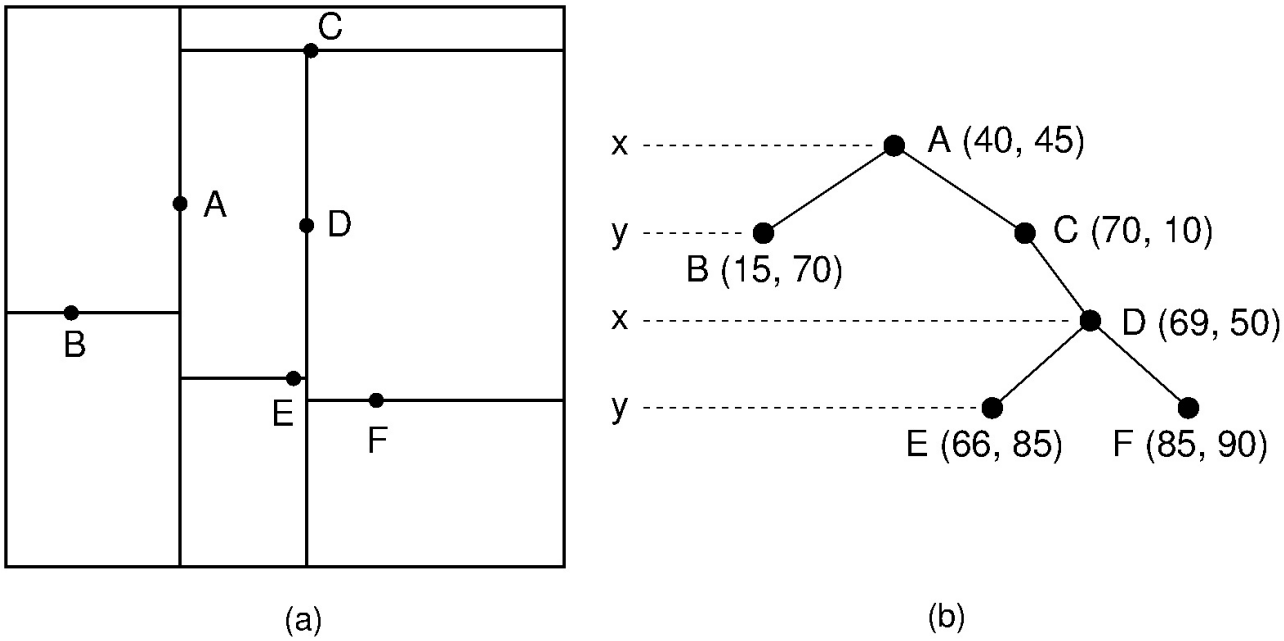
Once you have built up all chunks, and PVS sets for each chunk at runtime it's just a matter of checking which chunk the camera is in, and then rendering all chunks within the PVS of the current one.

Kd-Tree

A kd-tree, short for k-dimensional tree is often used for large outdoor games (Like grand theft auto or far cry).

In practice, a kd-tree works very similar to the BVH structure we created, just on a larger scene scale, like the Octree. The kd-tree is shaken, so the number of children is not exact. There is one major difference between the kd-tree and things we implemented. Unlike our things, the KD-tree is not evenly split.

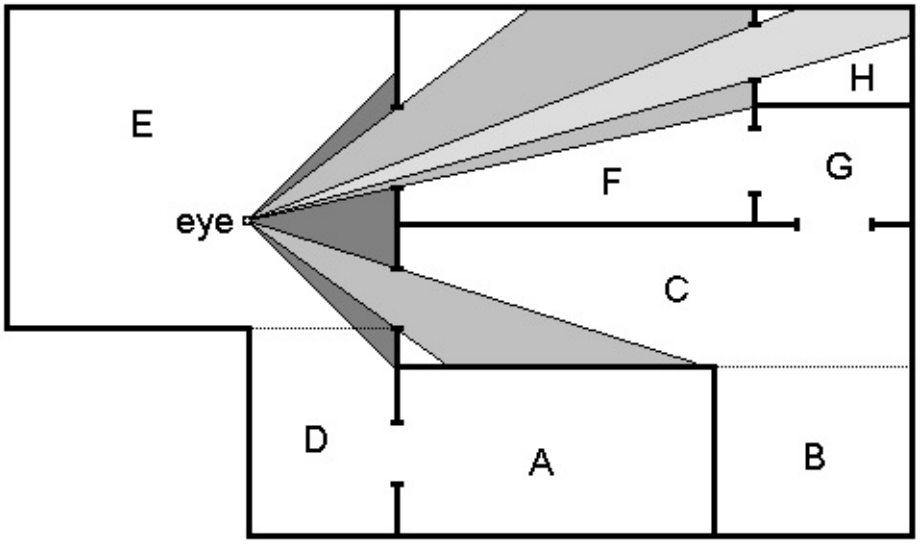
Instead of getting split down the middle, like an octree you split a kd tree according to some heuristic. Usually, polygon count. This leaves the tree divided heavily where it counts, and not very much where it doesn't.



Portals

Portals are mainly used to cull indoor scenes. The tought is, if you are indoors, you can see the room you are in, and any rooms connected with a door. A portal can be more than just a door tough. Assume you have a curvy tunnel in a cave. You can't see the end of the cave from the start because it curves. The most curvy point of that cave makes for a good protal.

Portals are a natural evolution of the PVS algorithm.


E
eye
H
F
G
C
D
A
B

Camera

So far, we've set the view matrix of every example scene manually. Now, it's time to implement a camera! We're going to implement a mouse controlled editor camera. Our controls are going to be:

- Left click + Drag - Move the camera around
- Right click + Drag - Rotate the camera around
- Mouse wheel - Zoom the camera in and out

Getting Ready

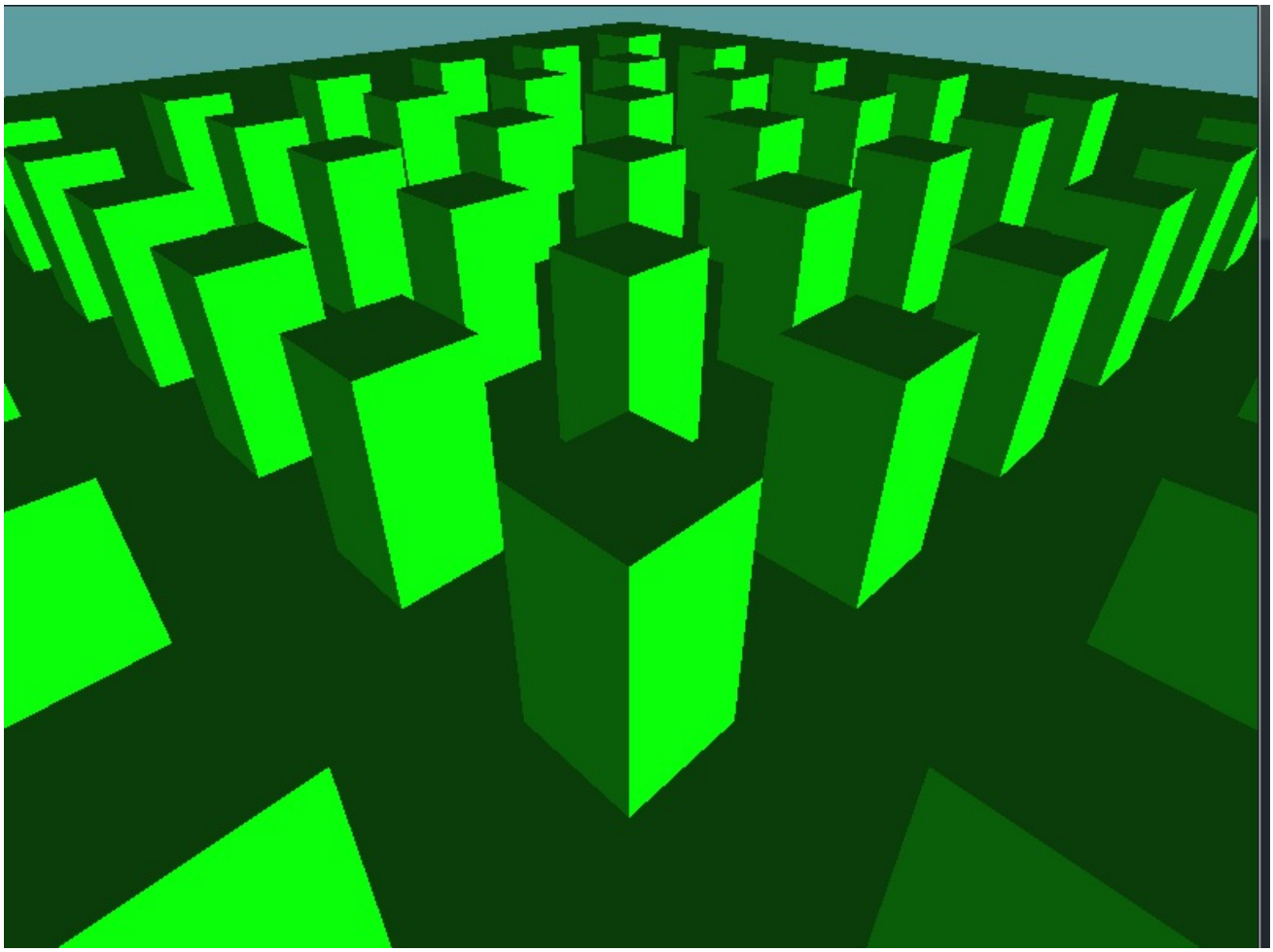
Let's set up a quick scene before we do anything

```
using OpenTK.Graphics.OpenGL;
using Math_Implementation;
using CollisionDetectionSelector.Primitives;
using CollisionDetectionSelector;

namespace CollisionDetectionSelector.Samples {
    class CameraSample : Application {
        Scene scene = new Scene();
        OBJLoader cube = null;

        void AddCubeToSceneRoot(Vector3 position, Vector3 scale) {
            scene.RootObject.Children.Add(new OBJ(cube));
            int count = scene.RootObject.Children.Count - 1;
            scene.RootObject.Children[count].Parent = scene.RootOb
ject;
            scene.RootObject.Children[count].Position = position;
            scene.RootObject.Children[count].Scale = scale;
        }

        public override void Intialize(int width, int height) {
```

```
            GL.Enable(EnableCap.DepthTest);
            GL.Enable(EnableCap.Lighting);
            GL.Enable(EnableCap.Light0);
            GL.PointSize(5f);

            GL.Light(LightName.Light0, LightParameter.Position, new
 float[] { 0.5f, -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 });

            scene.Initialize(7f);
            cube = new OBJLoader("Assets/cube.obj");

            AddCubeToSceneRoot(new Vector3(0.0f, 0.0f, 0.0f), new
Vector3(50.0f, 1.0f, 50.0f));
            for (int i = 0; i < 10; ++i) {
                for (int j = 0; j < 10; ++j) {
                    AddCubeToSceneRoot(new Vector3(25 - i * 5, 0.0
f, 25 - j * 5), new Vector3(1.0f, 5.0f, 1.0f));
                }
            }
        }

        public override void Render() {
            Matrix4 viewMat = Matrix4.LookAt(new Vector3(10, 10, 10
), new Vector3(0, 0, 0), new Vector3(0, 1, 0));
            GL.LoadMatrix(viewMat.OpenGL);
            DrawOrigin();

            GL.Enable(EnableCap.Lighting);
            scene.Render(false);
            GL.Disable(EnableCap.Lighting);
        }
```

```
    }
}
```

This is a simple scene with a bunch of cubes, should look like this when you run the app



The Algorithm

Now it's time to implement the basic camera class. Make a new **Camera.cs** file. I'll provide the skeleton here, and there will be a TODO after the code.

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

```
namespace CollisionDetectionSelector.Primitives {
    class Camera {
        // Members to hold the position & orientation of the camera

        protected Vector3 position = new Vector3(0f, 0f, 0f);
        protected Vector3 forward = new Vector3(0f, 0f, 1f);
        protected Vector3 right = new Vector3(1f, 0f, 0f);
        protected Vector3 up = new Vector3(0f, 1f, 0f);

        // Cached matrices to avoid work every frame
        protected bool worldDirty = true;
        protected bool viewDirty = true;
        protected Matrix4 cachedWorld = new Matrix4();
        protected Matrix4 cachedView = new Matrix4();

        public Matrix4 Translation {
            get {
                // TODO: Return a translation matrix!
            }
        }

        public Matrix4 Orientation {
            get {
                // TODO: Return an orientation matrix
                // based on forward, right and up!
            }
        }

        // The World matrix is a combination of the translation an
d orientation matrices
        public Matrix4 WorldMatrix
        {
            get
            {
                if (worldDirty) {
                    // Remember, POST MULTIPLICATION!
```

```
                    // That means rotate first, translate second!
                    cachedWorld = Translation * Orientation;
                }
                worldDirty = false;
                return cachedWorld;
            }
        }

        // The view is just the inverse of the world matrix.
        public Matrix4 ViewMatrix
        {
            get
            {
                if (viewDirty) {
                    cachedView = Matrix4.Inverse(WorldMatrix);
                }
                viewDirty = false;
                return cachedView;
            }
        }

        // Read Only Position
        public Point Position {
            get {
                return new Point(position.X, position.Y, position.
Z);
            }
        }

        // Read only forward
        public Vector3 Forward {
            get {
                return new Vector3(forward.X, forward.Y, forward.Z
);
            }
        }
```

```
        // Read only right
        public Vector3 Right {
            get {
                return new Vector3(right.X, right.Y, right.Z);
            }
        }

        // Read only up
        public Vector3 Up {
            get {
                return new Vector3(up.X, up.Y, up.Z);
            }
        }

        // Does the same thing that Matrix4.LookAt does
        public void LookAt(Vector3 camPosition, Vector3 camTarget,
 Vector3 camUp) {
            worldDirty = true;
            viewDirty = true;

            forward = Vector3.Normalize(camTarget - camPosition);
            right = Vector3.Normalize(Vector3.Cross(forward, camUp
));
            up = Vector3.Cross(right, forward);

            // Set to a copy, not a reference!
            position = new Vector3(camPosition.X, camPosition.Y, c
amPosition.Z);
        }
    }
}
```

There are two TODO's in this code. First, you have to create a translation matrix based on the position of the camera. This should be fairly simple.

Next, you must return a new orientation matrix. This one might be a little harder. You should be hand-filling in the elements of the matrix. Remember, the upper 3x3 matrix is the rotation matrix. Within the upper 3x3 matrix, each column is a basis vector. The X axis for example is the right vector. Also, becuase this is OpenGL, and we are in a right-handed coordinate system, you have to negate the forward basis when constructing the matrix.

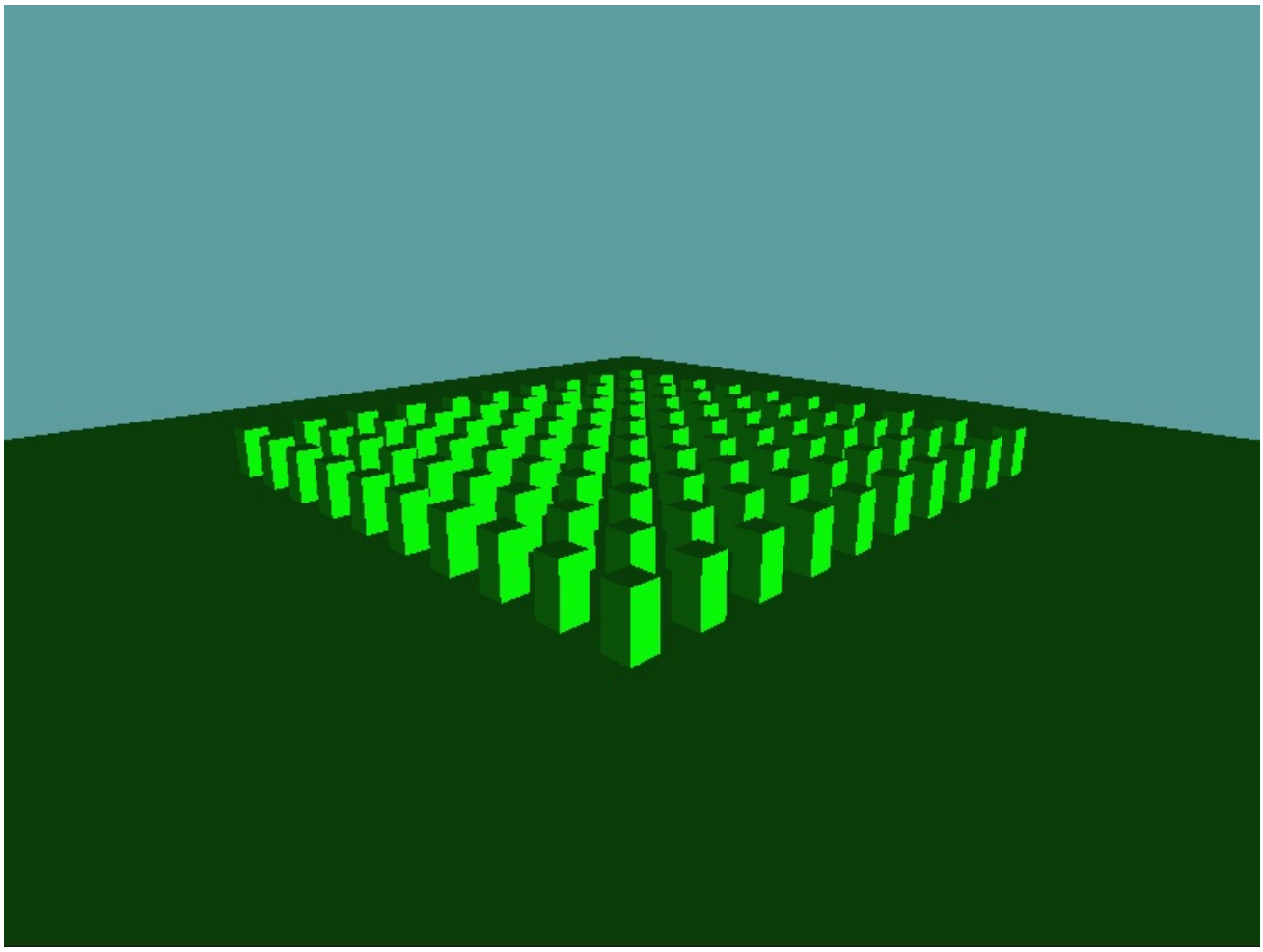
Add the following unit test code to the end of the scene class (`CameraSample`) Initialize function

```
// Set the camera look at!
            camera.LookAt(new Vector3(50.0f, 20.0f, 50.0f), new Ve
ctor3(0.0f, 0.0f, 0.0f), new Vector3(0.0f, 1.0f, 0.0f));

            // Unit tests!
            Matrix4 expectedTrans = new Matrix4();
            expectedTrans[0, 3] = expectedTrans[2, 3] = 50f;
            expectedTrans[1, 3] = 20f;
            if (expectedTrans != camera.Translation) {
                System.Console.WriteLine("Error! Translation matri
x is wrong!");
            }

            Matrix4 expectedRot = new Matrix4();
            expectedRot[0, 0] = 0.7071068f;
            expectedRot[2, 0] = -expectedRot[0, 0];
            expectedRot[1, 1] = 0.9622504f;
            expectedRot[2, 2] = expectedRot[0, 2] = 0.6804138f;
            expectedRot[0, 1] = expectedRot[2, 1] = -0.1924501f;
            expectedRot[1, 2] = 0.2721655f;
            if (expectedRot != camera.Orientation) {
                System.Console.WriteLine("Error! Rotation matrix i
s wrong!");
            }

            Matrix4 camView = camera.ViewMatrix;
            Matrix4 matView = Matrix4.LookAt(new Vector3(50, 20, 50
), new Vector3(0, 0, 0), new Vector3(0, 1, 0));
            if (camView != matView) {
                System.Console.WriteLine("ERROR! expected camera a
nd matrix view's to be the same!");
            }
```

And change the render function to use the cameras matrix:

```
        public override void Render() {
            GL.LoadMatrix(camera.ViewMatrix.OpenGL);
            DrawOrigin();

            GL.Enable(EnableCap.Lighting);
            scene.Render(false);
            GL.Disable(EnableCap.Lighting);
        }
```

If no errors are thrown, your scene should have a slightly different perspective and look like this:



Interaction

Finally, lets make the camera interactive! There are 3 key motions that the camera will need to do. The left mouse button will **Pan**, the middle mouse wheel will **Zoom** and the right mouse button will **Pivot**. With that estabolished, add the following functions to the **Camera** class.

```
public void Pan(float horizontal, float vertical) {
    worldDirty = true;
    viewDirty = true;
    // TODO: Implement
}

public void Zoom(float value) {
    worldDirty = true;
    viewDirty = true;
    // TODO: Implement
}

public void Pivot(float yaw, float pitch) {
    worldDirty = true;
    viewDirty = true;
    // TODO: Implement
}
```

Before we implement anything, let's add the code to `CameraSample` that calls these methods. That way you can test one method as a time as you are implementing each. Add the following `Update` method to the `CameraSample` scene:

```
public override void Update(float deltaTime) {
    float xDelta = (float)Window.Mouse.XDelta / (float)Window.Widt
h;
    float yDelta = (float)Window.Mouse.YDelta / (float)Window.Heig
ht;
    float zoom = (float)Window.Mouse.WheelDelta;

    if (Window.Mouse[OpenTK.Input.MouseButton.Left]) {
        if (xDelta != 0.0f || yDelta != 0.0f) {
            float xPan = xDelta * deltaTime * -900.0f;
            float yPan = yDelta * deltaTime * 900.0f;
            camera.Pan(xPan, yPan);
        }
    }

    if (zoom != 0) {
        zoom = zoom < 0 ? -1f : zoom;
        zoom = zoom > 0 ? 1f : zoom;
        camera.Zoom(zoom * deltaTime * 50.0f);
    }

    if (Window.Mouse[OpenTK.Input.MouseButton.Right]) {
        if (xDelta != 0.0f || yDelta != 0.0f) {
            float xPan = xDelta * deltaTime * -900.0f;
            float yPan = yDelta * deltaTime * 900.0f;
            camera.Pivot(xPan, yPan);
        }
    }
}
```

Implement Pan

Pan is simple. You have to modify the position vector. Basically, any horizontal movement will modify the horizontal vector along the right normal by that amount. Any vertical movement should modify the position vector along the up normal by that

amount. If this is confusing (I can't think of a better way to explain it) talk to me.

Implement zoom

Zoom works the same way pan does. But, instead of moving on the right or up normals, zoom works along the forward normal

Implement Pivot

Pivot is by far the hardest camera transform we have to do. But, we can make it happen. The key is, there are actually two pivots happening. One about the yaw and one about the pitch. We can actually model these.

- First, multiply the forward vector by an anlge axis.
  - The anlge is the yaw, the axis is the up vector
- Next, multiply the forward vector by another angle axis
  - The angle is the pitch, the axis is the right vector
- Finally, normalize the 3 direction vectors
  - Similar to how the `LookAt` function does this.
  - Major difference is you already have forward, you just need to normalize it

Debug info

Before moving on, to implement a camera frustum i want to add some debug info to the scene. Nameley, i want to know how many objects are rendered. I'm actually going to provide all the code for this, just follow along. This is actually pretty simple, we start with the most basic thing that's being rendered.

In **OBJ.cs**, change the `Render` method of the `OBJ` class from this (taken from your github):

```
public void Render() {
    GL.PushMatrix();
    //always getter
    GL.MultMatrix(WorldMatrix.OpenGL);
    if (model != null) {
        model.Render();
    }
    GL.PopMatrix();
    ChildrenRender(true, false, false);
}
```

to this

```
public int Render() {
    int result = 0;

    GL.PushMatrix();
    GL.MultMatrix(WorldMatrix.OpenGL);
    if (model != null) {
        model.Render();
        result += 1;
    }
    GL.PopMatrix();

    result += ChildrenRender(true, false, false);
    return result;
}
```

All this changes is the Render function now keeps track of every time model.Render has been caleed. Of course, this means we now also have to change `ChildrenRender` to:

```
private int ChildrenRender(bool normal,bool bvh, bool debug) {
    int result = 0;
    if (Children != null) {
        foreach(OBJ child in Children) {
            if (normal) {
                result += child.Render();
            }
            else if (bvh) {
                child.RenderBVH();
            }
            else if (debug) {
                child.DebugRender();
            }
            if(child.Children != null) {
                result += child.ChildrenRender(normal, bvh, debug)
;
            }
        }
    }
    return result;
}
```

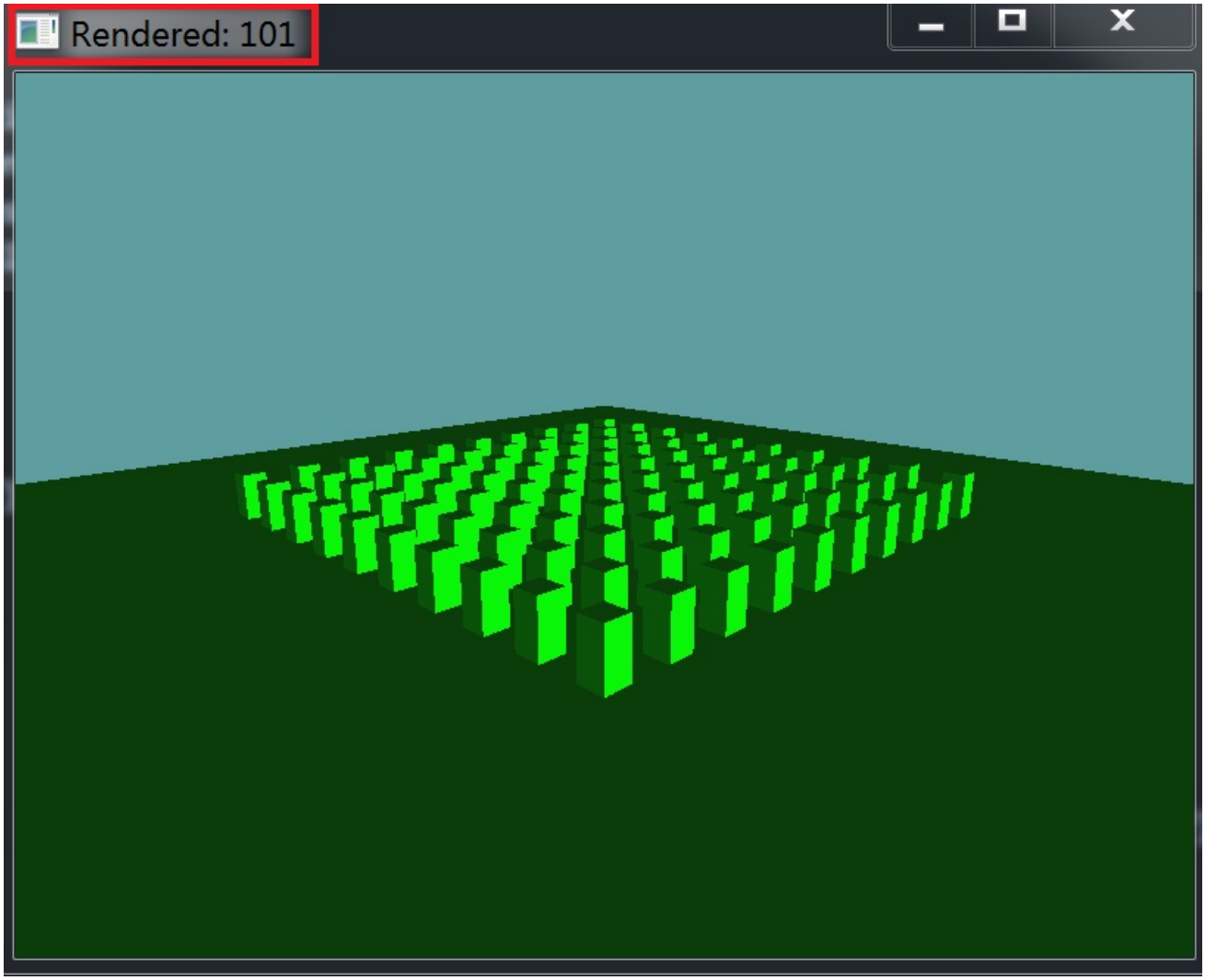
Now we have to go one more up the call stack and change the `Render` function of **Scene.cs** to also return how many objects where rendered:

```
public int Render() {
    int result = RootObject.Render();

    GL.Disable(EnableCap.Lighting);
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Line);
    Octree.DebugRender();
    GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill);

    Octree.DebugRenderOnlyVisitedNodes();

    GL.Enable(EnableCap.Lighting);
    return result;
}
```

Finally, we change the **Render** function of `CameraSample.cs` to display how many objects are rendered in the window title.

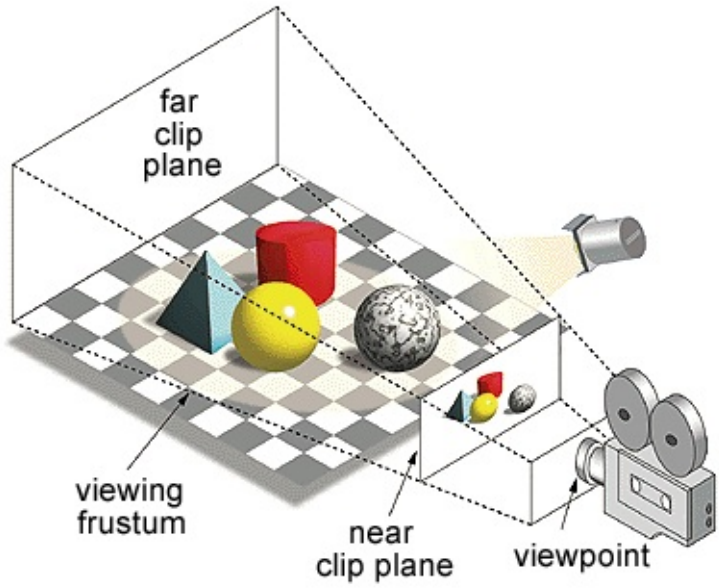
```
public override void Render() {
    GL.LoadMatrix(camera.ViewMatrix.OpenGL);
    DrawOrigin();

    GL.Enable(EnableCap.Lighting);
    int numRendered = scene.Render(false);
    Window.Title = "Rendered: " + numRendered;
    GL.Disable(EnableCap.Lighting);
}
```

If all is well, the number of rendered objects should be a constant 101.


Rendered: 101

Frustum

The frustum structure is actually an acceleration structure. It's a primitive shape (Like an AABB or any other primitive). A frustum structure is usually built out of a view matrix as a series of 6 planes.



The Algorithm

If this all sounds a bit familiar, it should. We have actually done this exact same thing when we made a frustum in OpenGL.

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.

Our camera class already has the ViewMatrix, but not the projection matrix! Let's fix that. Add the following getter to your `Camera` class. This getter will simply query OpenGL for the current projection matrix. This is similar to what we did in the picking example.

```
public Matrix4 ProjectionMatrix {
    get {
        float[] rawProjection = new float[16];
        GL.GetFloat(GetPName.ProjectionMatrix, rawProjection);
        return Matrix4.Transpose(new Matrix4(rawProjection));
    }
}
```

Now that we have the projection matrix, we can resume extracting the planes

- **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 =  ProjectionMatrix * ViewMatrix;

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;
```

Fixing some things

Originally, the `Frustum` getter was a **On Your Own** todo. However, i noticed a pretty big error with the way we did it previously on our OpenGL Implementation. So, i'm just going to provide the code for it and explain what was wrong. Add the following code to the `Camera` class:

```
private static Plane FromNumbers(Vector4 numbers) {
            Vector3 abc = new Vector3(numbers.X, numbers.Y, number
s.Z);
    float mag = abc.Length();
```

```
    abc.Normalize();

    Plane p = new Plane();
    p.Normal = abc;
    p.Distance = numbers.W / mag;
    return p;
}

public Plane[] Frustum {
    get {
        Plane[] frustum = new Plane[6];

        Matrix4 vp = ProjectionMatrix * ViewMatrix;

        Vector4 row1 = new Vector4(vp[0, 0], vp[0, 1], vp[0, 2], v
p[0, 3]);
        Vector4 row2 = new Vector4(vp[1, 0], vp[1, 1], vp[1, 2], v
p[1, 3]);
        Vector4 row3 = new Vector4(vp[2, 0], vp[2, 1], vp[2, 2], v
p[2, 3]);
        Vector4 row4 = new Vector4(vp[3, 0], vp[3, 1], vp[3, 2], v
p[3, 3]);

        frustum[0] = FromNumbers(row4 + row1);
        frustum[1] = FromNumbers(row4 - row1);
        frustum[2] = FromNumbers(row4 + row2);
        frustum[3] = FromNumbers(row4 - row2);
        frustum[4] = FromNumbers(row4 + row3);
        frustum[5] = FromNumbers(row4 - row3);

        return frustum;
    }
}
```

PReviously, the `FromNumbers` function was busted. It was not setting the plane normal to a normalized number. Also, when normalizing, you MUST divide the W component by the length of the XYZ components. PReviously, that was not being

done either. The above version is correct! In order for the below unit test to pass, you have to populate your frustum as follows:

- frustum[0] = left plane
- frustum[1] = right plane
- frustum[2] = bottom plane
- frustum[3] = top plane
- frustum[4] = near plane
- frustum[5] = far plane

Unit Test

Add the following unit test at the bottom of the `Initialize` function of `CameraSample`

```
Plane[] frustum = new Plane[6];
frustum[0] = new Plane(new Vector3(0.2381448f, -0.2721655f, -1.598
973f), 44.82524f);
frustum[1] = new Plane(new Vector3(-1.598973f, -0.2721655f, 0.2381
448f), 44.82524f);
frustum[2] = new Plane(new Vector3(-1.013747f, 1.394501f, -1.013747
f), 36.74234f);
frustum[3] = new Plane(new Vector3(-0.3470805f, -1.938832f, -0.347
0805f), 36.74234f);
frustum[4] = new Plane(new Vector3(-1.360841f, -0.5443366f, -1.360
841f), 73.4747f);
frustum[5] = new Plane(new Vector3((float)1.364946E-05, (float)5.4
53825E-06, (float)1.364946E-05), 923.8687f);

for (int i = 0; i < 6; ++i) {
    if (camera.Frustum[i].Normal != frustum[i].Normal) {
        if (Vector3.Normalize(camera.Frustum[i].Normal) == Vector3
.Normalize(frustum[i].Normal)) {
            System.Console.WriteLine("\tLooks like the normal of y
our frustum plane is NOT NORMALIZED!");
        }
```

```
        else {
            System.Console.WriteLine("Detected error in frustum, p
lane " + i);
        }
        System.Console.WriteLine("\tExpected: " + camera.Frustum[i
].Normal);
        System.Console.WriteLine("\tGot: " + frustum[i].Normal);
    }
    if (System.Math.Abs(camera.Frustum[i].Distance - frustum[i].Di
stance) > 0.0001f) {
        System.Console.WriteLine("Wrong distance for plane: " + i)
;
        System.Console.WriteLine("\tExpected: " + camera.Frustum[i
].Distance);
        System.Console.WriteLine("\tGot: " + frustum[i].Distance);
    }
}
```

All tests should pass before moving on to the next section.

Sidenote, This Article is generally considered the go-to resource on frsutum culling.

Point In Frustum

Perhaps the easyest frustum test to perform is the Point In Frustum test. Actually, we've implemented this functionality before. The only difference is, now we will formally include it in our collision library.

The Algorithm

The algorithm here is simple, it's essentially just a half-space test. A half space test is very similar to the `DistanceFromPlane` function we wrote, except instead of adding plane.Distance at the end, we subtract it. Once you can do a half-space test, doing the actual intersection is easy, loop trough all of the frustum planes, and do a half space test between the point and each plane. If any of the half space tests are less than 0, return false immediateley. IF they are all 0 or greater you can return true by default.

On Your Own

Add the following function to the `Collisions` class:

```
public static float HalfSpaceTest(Point point, Plane plane) {
    // TODO: Implement this!
}

public static bool Intersects(Plane[] frustum, Point point) {
    // TODO: Implement this!
}

public static bool Intersects(Point point, Plane[] frustum) {
    return Intersects(frustum, point);
}
```

And provide an implementation for it!

Unit Test

Add the following unit test at the end of the `Initialize` function in `CameraSample`.

```
if (!Collisions.Intersects(new Point(), camera.Frustum)) {
    System.Console.WriteLine("Error with point in frustum! 0");
}
if (Collisions.Intersects(new Point(-5000,-5000,-5000), camera.Fru
stum)) {
    System.Console.WriteLine("Error with point in frustum! 1");
}
if (!Collisions.Intersects(new Point(2, 30, 4), camera.Frustum)) {
    System.Console.WriteLine("Error with point in frustum! 2");
}
```

Sphere In Frustum

Doing the sphere in frustum is pretty easy. We already have a `HalfSpaceTest` function in our collision class. We want to check the distance of the sphere's space to every plane of the frustum. If the distance between the sphere and any frustum plane is less than the negative radius of the sphere (That is to say, its further behind the plane than the radius of the sphere), no collision has happened

The Algorithm

```
for each plane of the frustum
  space = HalfSpaceTest(sphere.center, plane);
  // If the distance is < radius, the sphere is outside
  if (space < -radius)
    return false; // We are outside!
return true; // We're in
```

On Your Own

Add the following function to the `Collisions` class:

```
public static bool Intersects(Plane[] frustum, Sphere sphere) {
    // TODO: Implement this
}

public static bool Intersects(Sphere sphere, Plane[] frustum) {
    return Intersects(frustum, sphere);
}
```

And provide an implementation for it!

Unit Test

Add the following unit test to the end of the `Initialize` function of `CameraSample`

```
if (!Collisions.Intersects(new Sphere(), camera.Frustum)) {
    System.Console.WriteLine("Error with sphere in frustum! 0");
}
if (Collisions.Intersects(new Sphere(-5000, -5000, -5000, 1f), cam
era.Frustum)) {
    System.Console.WriteLine("Error with sphere in frustum! 1");
}
if (!Collisions.Intersects(new Sphere(2, 30, 4, 4f), camera.Frustu
m)) {
    System.Console.WriteLine("Error with sphere in frustum! 2");
}
```

AABB In Frustum

Testing if an AABB is within a frustum is a bit of a brute force method. We first have to get every vertex of the AABB, this part is simple enough. We alrady did this when we split the BVH tree. We just need the 8 points.

Once we have the 8 points it's a nested loop.

For every plane of the frustum, we check all 8 corners of the AABB. If there is any plane where all 8 corners of the AABB are outside of the plane (Have a negative half-space test), then the AABB and frustum do not intersect!

By default, if everything passes we assume an intersection to be true

The Algorithm

```
foreach Plane plane in frustum {
    int inCount = 8;
    foreach Point corner in corners{
        if HalfSpaceTest(corner, plane) < 0.0f {
            inCount -= 1;
        }
    }
    if inCount <= 0
        return false;
}
return true; // by default
```

On Your Own

Add the following function to the `Collisions` class:

```
public static bool Intersects(Plane[] frustum, AABB aabb) {
    // TODO
}

public static bool Intersects(AABB aabb, Plane[] frustum) {
    return Intersects(frustum, aabb);
}
```

And provide an implementation for it!

Unit Test

Add the following unit test to the bottom of the `Initialize` function in `CameraSample`

```
if (!Collisions.Intersects(new AABB(), camera.Frustum)) {
    System.Console.WriteLine("Error with aabb in frustum! 0");
}
if (Collisions.Intersects(new AABB(new Point(-650, -650, -650), new
 Point(-600, -600, -600)), camera.Frustum)) {
    System.Console.WriteLine("Error with aabb in frustum! 1");
}
if (!Collisions.Intersects(new AABB(new Point(2, 30, 4), new Point(
25, 35, 45)), camera.Frustum)) {
    System.Console.WriteLine("Error with aabb in frustum! 2");
}
```

Culling

We can now test robust enough intersections to perform vibility culling on a scene level! Whoo!!!! To achieve this, we're going to have to move the `Rendering` responsibility out of the `OBJ` class and into the `OctreeNode` class.

OBJ - Render Flag

There is a slight problem in this shift of responsibility. One OBJ can belong to multiple `OctreeNode` objects. To make sure that only the first node that contains the `OBJ` actually renders it, we're going to add a new flag to the `OBJ` class.

```
protected bool wasRendered = false;
```

OBJ - Reset Flag

Because we added a Render Flag, let's add a way to clear that flag. Still working with the `OBJ` class, add the following method:

```
public void ResetRenderFlag() {
    wasRendered = false;
}
```

OBJ - Non Recursive Render

The OBJ class should no longer render its-self recursivley. This is becuase the `OctreeNode` is going to be rendering only `OBJ` objects that are in view. We are changing our renderer from hierarchial traversal to spacial traversal. We're going to add a new function to `OBJ` to render only this object, and return true only if this is the first time the new render function was called, and something was rendered.

Additionally, this function will take in a frustum as an argument and do a narrow-phase visibility check. That is, if the `OctreeNode` calls this function, we know the node is visible. But that doens't mean this object is within the camera frustum. So, we do a narrow phase test between the frustum and the objects bounding sphere.

```
public bool NonRecursiveRender(Plane[] frustum) {
    // Is there anything to render?
    if (model == null) {
        return false;
    }

    // Already rendered once!
    if (wasRendered) {
        return false;
    }


    // Is the bounds of this obejct within the view?
    Sphere bounds = new Sphere();
    bounds.Position = new Point(
        Matrix4.MultiplyPoint(WorldMatrix, model.BoundingSphere.Po
sition.ToVector())
    );
    float scalar = System.Math.Abs(System.Math.Max(
        System.Math.Max(
            System.Math.Abs(WorldMatrix[0, 0]), System.Math.Abs(Wo
rldMatrix[1, 1])),
            System.Math.Abs(WorldMatrix[2, 2])));
    bounds.Radius = model.BoundingSphere.Radius * scalar;


    if (!Collisions.Intersects(frustum, bounds)) {
        return false;
    }

    // Cool, we can render!
    wasRendered = true;
```

```
    GL.PushMatrix();
    GL.MultMatrix(WorldMatrix.OpenGL);
    model.Render();
    GL.PopMatrix();
    return true;
}
```

OctreeNode - Render

Add a new method to `OctreeNode`. Call this method `Render`. It will have the following signature:

```
public int Render(Plane[] frustum) {
    int total = 0;
    // TODO: Render logic for this node

    // TODO: Recurse trough all children
    return total;
}
```

The key here is, you can only render an OctreeNode IF you have a frustum. This happens because we will be culling the node against the frustum.

First up, the total is 0. Before doing anything, check if the `Bounds` of the Node are intersecting with the frustum (Frsutum V AABB). If not, just return 0.

From here on out, if we didn't return we know we have a visible node.

Next, if the `Contents` of the node are not null, go ahead and call the `NonRecursiveRender` function on each of the contents. for every `NonRecursiveRender` called, increment `total` by 1, if the drawing took place.

Lastly, if `Children` is not null, loop trough each of the child `OctreeNode` objects, and recursively call their `Render` functions, passing in the same frustum. Be sure to add the result of that function to the total variable.

OctreeNode - ResetRenderFlag

Once a frame is done, we need a way to mark all objects as non-rendered. For this reason we are going to add a `ResetRenderFlag` function to `OctreeNode`. This function is simple, if the node has contents (OBJ objects), it calls `ResetRenderFlag` on each of them. If the node has children, this `ResetRenderFlag` function recursivley calls its-self on each child.

```
public void ResetRenderFlag() {
    if (Contents != null) {
        foreach (OBJ content in Contents) {
            content.ResetRenderFlag();
        }
    }
    if (Children != null) {
        foreach (OctreeNode child in Children) {
            child.ResetRenderFlag();
        }
    }
}
```

Unit Test

There is no real unit test, we're just going to make culling work in the `CameraSample`.

The `Scene` class already contains an `Octree`, but it's WAY too small. Find where the scene is Initialized and change it from `scene.Initialize(7f);` to a bigger Octree `scene.Initialize(70f);`.

Now, just because we have an Octree, does not mean anything is in it. Change the `AddCubeToSceneRoot` function so that it adds cubes not only into the hierarchy, but also the octree.

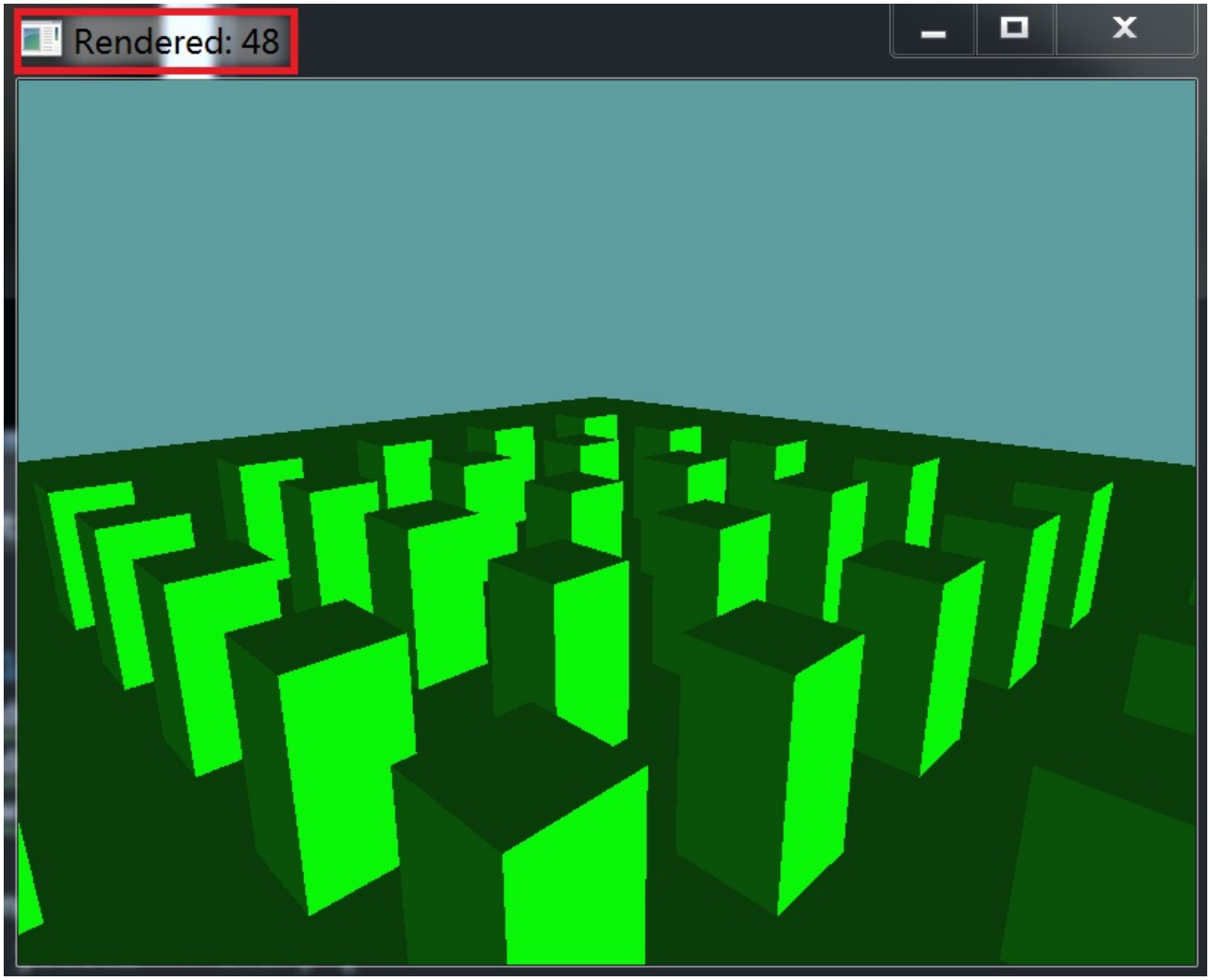
```
void AddCubeToSceneRoot(Vector3 position, Vector3 scale) {
    scene.RootObject.Children.Add(new OBJ(cube));
    int count = scene.RootObject.Children.Count - 1;
    scene.RootObject.Children[count].Parent = scene.RootObject;
    scene.RootObject.Children[count].Position = position;
    scene.RootObject.Children[count].Scale = scale;

    // Record object with spacial partitioning tree
    scene.Octree.Insert(scene.RootObject.Children[count]);
}
```

Lastly, we need to change the `Render` function so that it renders the octree, instead of the root node, and that it clears render flags after the scene was drawn:

```
public override void Render() {
    GL.LoadMatrix(camera.ViewMatrix.OpenGL);
    DrawOrigin();

    GL.Enable(EnableCap.Lighting);
    int numRendered = scene.Octree.Render(camera.Frustum);
    scene.Octree.ResetRenderFlag();
    Window.Title = "Rendered: " + numRendered;
    GL.Disable(EnableCap.Lighting);
}
```

Try it out

Running the game now, as you move the camera trough the world, you should see considerably less objects being drawn.


Rendered: 48

Further Research

We've covered the bare minimum of game collisions so far. With what we did cover, you should be able to create a basic game. For example, your character can walk on uneven terrain now, if you where to cats a ray from the top of the character down, and get the intersection point with the terrain mesh, you would know to plane the characters feet at that point.

The basiscs are enough to make some simple games, but at some point more advanced knowledge might be needed. I've listed out several other common collision shapes.

The real key here is, when working with a game engine, the work is often done for you. What you need to know is which functions exist and what they do, then you can figure out how to do it in engine.

For example, unity uses PhysX for both it's physics and collision detection. It has a Raycast method that already does the raycast against an optimized Octree for you. If you want to pick objects in unity, the engine provides a Unproject function. You just have to call the built in Unproject and Raycast functions the same way you called your own functions.

The challenge from here on out is going to be figuring out how to achieve what you want to do in unity. What part of the math is implemented by unity for you, and what part you have to make from scratch. Unity will have most things already made tough, so long as you know how to use htem you should be fine.

OBB

An OBB, or Oriented Bounding box is essentially an AABB with a rotation. An OBB structure often looks like this:

```
class OBB {
    Vector3 Extents;
    Point Position;
    Matrix3 Orientation;
}
```

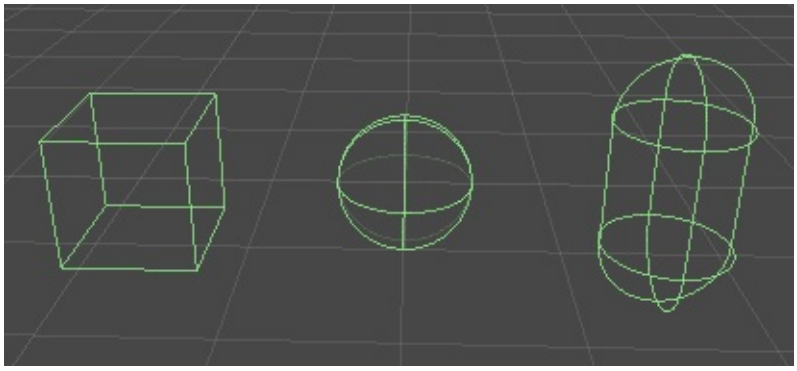
We've actually written most OBB collision tests already.

To test against collision, you take whatever shape the OBB is being tested against, and bring it into the local space of the OBB. You do this by multiplying the shape with the inverse of the OBB's Rotation Matrix.

Once the Other object is local to the OBB, in the OBB's view it's not an AABB. So, you just do an AABB to whatever test.

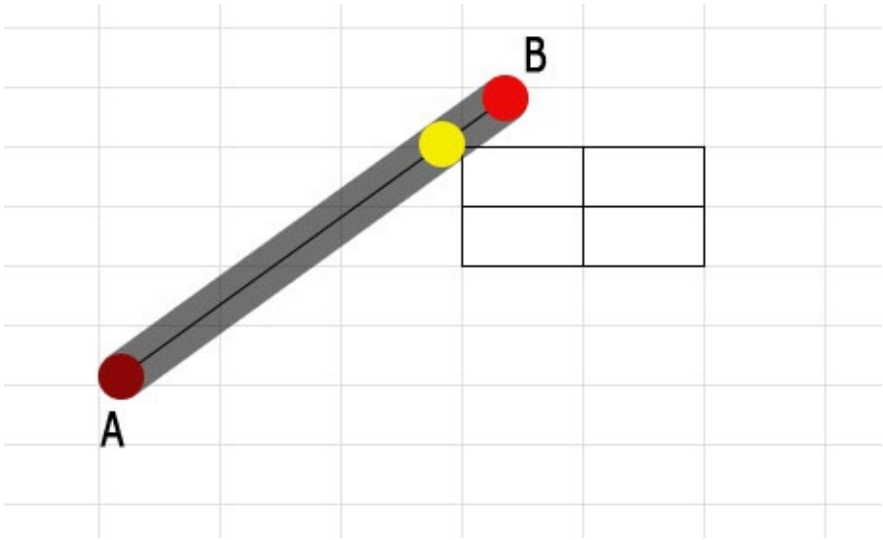
The one exception to this is of course OBB-OBB collisions. The best you can do is turn that into an OBB v AABB test. Once you have an OBB and an AABB, you test for collision with a seperating axis therom test.

Capsule



The basic collider types for any game are usually AABB, Sphere and Capsule. We use capsules because they can closly fit a human body.

So, what do you need to know about a capsule? It's basically a line with an infinate number of circles on it!



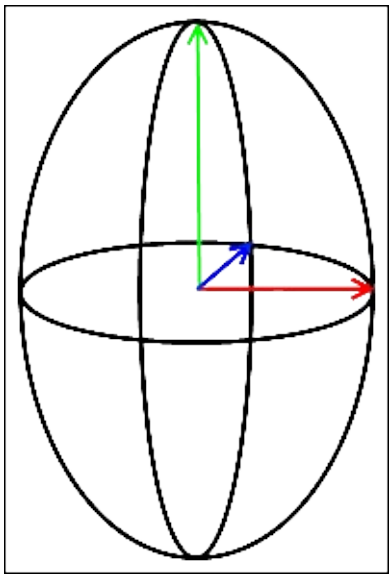
A capsule has a start point, end point and a radius.

```
class Capsule {
  Point start;
  Point end;
  float radius;
}
```

To test collision against a capsule, you must find the closest point to it's line segment to whatever it is you are testing, then construct a spehre at that point. Then it becomes a whatever / sphere test.

Ellipsoid

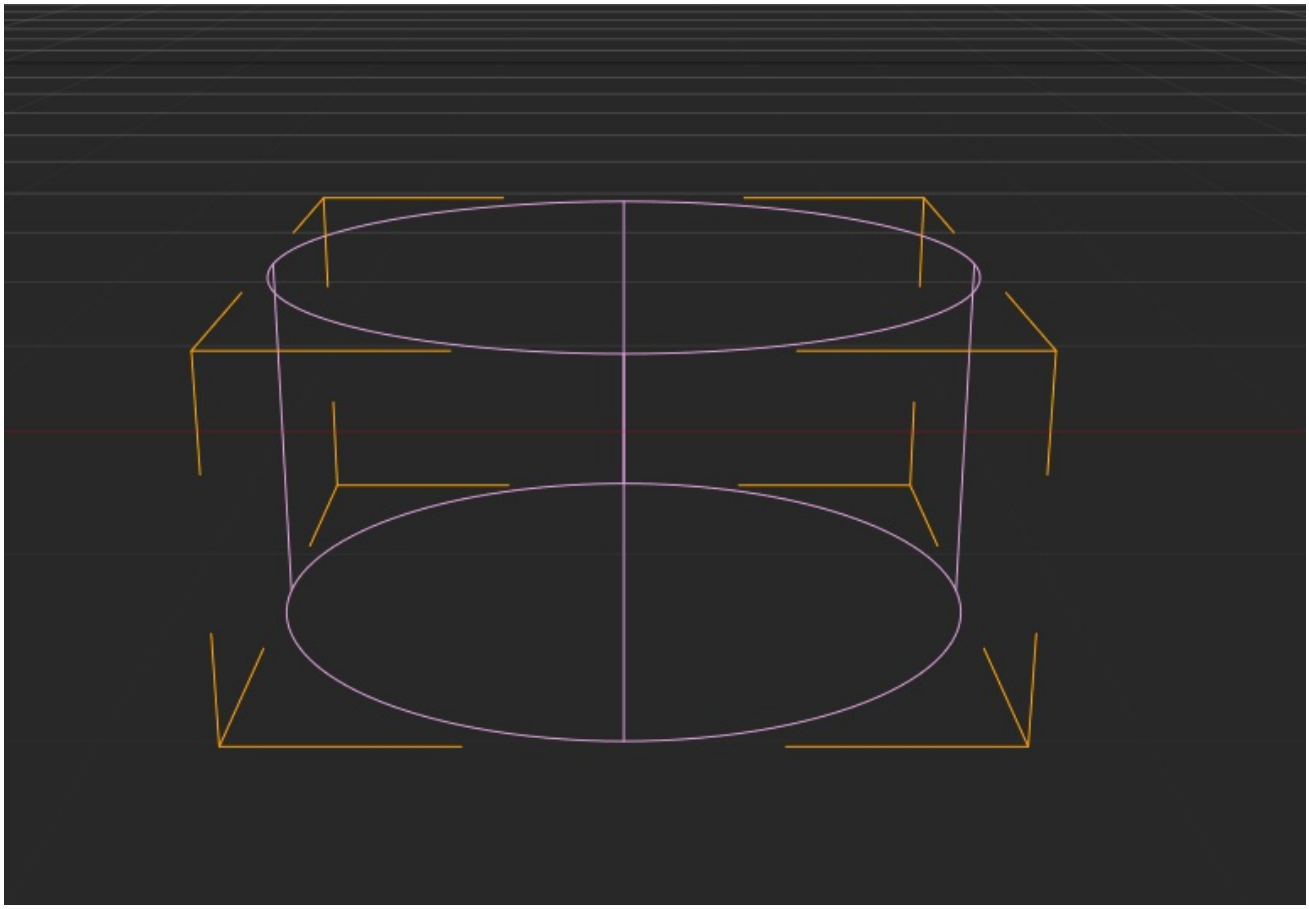
An ellipsoid is sometimes used in place of a capsule. It's not that great. An elipsoid is basically a sphere with non-uniform scale.



If cou can, avoid using this!

Cylinder

A cylinder is like a capsule, but without the caps.



There is usually nothing that this collider is too great for. If you need a cylinder shaped collider, you are better off just using a cylinder shaped mesh.

Physics Engines

Collisions in games are usually handled by physics engines. Mobile games sometimes forgo the physics engine in favor of more efficient hand-coded collisions, but with how powerful mobile devices are getting this is becomming less and less of a concern. Let's explore a few engines

Farseer

Farseer is one of the few engines specifically targetted at C#. As such, it was actually written in C#! This is their official description: Farseer Physics Engine is a collision detection system with realistic physics responses. This means you can create a game or robotic simulation easily using the engine and the associated tools. Everything from a simple hobby game to a complex robotic simulation is possible with Farseer Physics Engine

JigLib

The other engine written specifically for C#! It's a port of a non C# engine, but it performs quet well

PhysX

Unity, and most AAA games that ship use this as their physics engine. Written and maintained by NVidia, this engine has amazig performance as it can run mainly on the GPU.

Bullet

THE OPEN SOURCE ONE! Bullet is the Open Source Physics Engine of choice. It's been used in millions of student and professional projects. It's open source nature means someone has pretty much ported it to every platform.

ODE

The Open Dynamics Engine is bullets main open source competitor. I think the code is better, but it's under a very aweful, GPL License. The license basically means this can only be used in other open source projects. So, the library does not get much love.

MathGeoLib

This one is not actually a physics engine. It's just a bunch of hand-coded collision detection code. I often turn to this library when i'm stuck on a hard problem.