Graphics with OpenGL Documentation

Release 0.1

Jose Salvatierra

Nov 13, 2017

Contents

Please contribute to any section you feel like you can add to via a pull request at https://github.com/jslvtr/OpenGL-Notes.

Contents:

CHAPTER 1

Introduction and OpenGL

Contents:

1.1 Introduction to this document

1.1.1 What are graphics?

Simply **visual images or designs** on a medium, with the purpose to inform, illustrate, or entertain. **Computer graphics** are these on a computer screen.

1.1.2 What is covered in this document?

This document is mostly taken from Iain Martin's Graphics module from The University of Dundee.

It covers OpenGL 4.x, but it is not specifically an "OpenGL module". It is simply a Graphics module, which contains some OpenGL and theory of graphics.

There are a lot of programming examples, and some mathematics is necessary.

1.1.3 Recommended resources

Other resources that you may want to look at are:

- "The OpenGL Programming Guide" (latest edition, 8th at time of writing)
- "OpenGL SuperBible" (6th edition at time of writing)
- Books on OpenGL shaders, GLSL ("OpenGL Shading Language")
- Anton's OpenGL 4 Tutorials

Online resources

- Lighthouse tutorials
- OpenGL resources
- GLFW

1.1.4 Mathematics

Although we use libraries and built-in functions where possible, and seldom have to implement our own, it is useful to know some of the mathematics behind it. It's not hard, promise!

- Vectors (dot product, cross product)
- Matrices (and operations with them)
- Simple geometry (lines, planes, 3D geometry like spheres)
- Trigonometry
- Fractals and Noise functions (towards the end)

1.1.5 Why OpenGL?

The first vendor-independent API for development of graphics applications.

There is no need to license it to use it.

It was designed to use the graphics card where possible to improve performance.

Originally based on a state machine, procedural model; thus it can be used with a wide variety of programming languages (using Python in this document).

Primitives

OpenGL is platform independent, but does utilize hardware natively. We define our models using OpenGL primitives (vectors and matrices, mostly). OpenGL passes this to the hardware.

GLSL ("OpenGL Shading Language") allows (normally small chunks of) code to run in the graphics hardware. This code executes **substantially** faster than if it were executed in the CPU, as would happen in *normal* programming.

1.1.6 OpenGL versions

At time of writing, OpenGL 4.5 is the latest version. **Mac OS X 10.11 only supports OpenGL 4.1**.

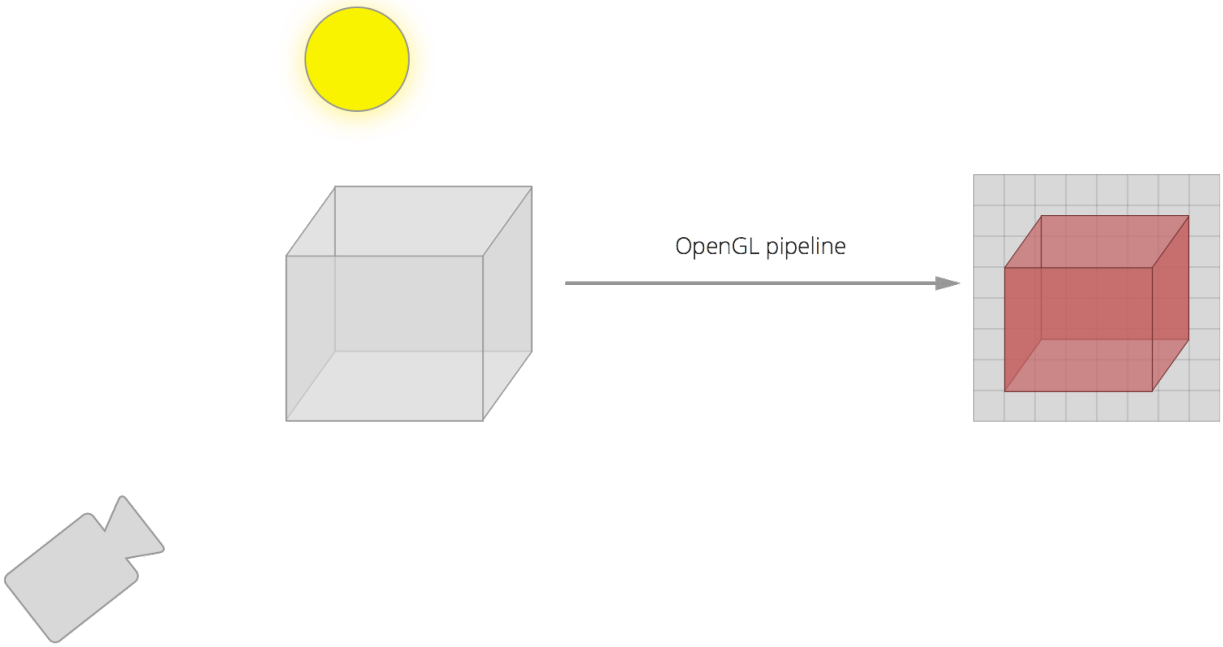
1. **OpenGL 1.x**
    - Based on a state machine and procedural model
2. **OpenGL 2.x**
    - Includes shading language (vertex and fragment)
    - Numerous stacked extensions
3. **OpenGL 3.3 onwards**
    - Forked into compatibility and core directions

- Geometry shaders (which were actually introduced in 3.2)

4. **OpenGL 4.x**
    - Updates to shading language (tessellation shaders)
    - Support for embedded devices

1.2 The OpenGL Pipeline

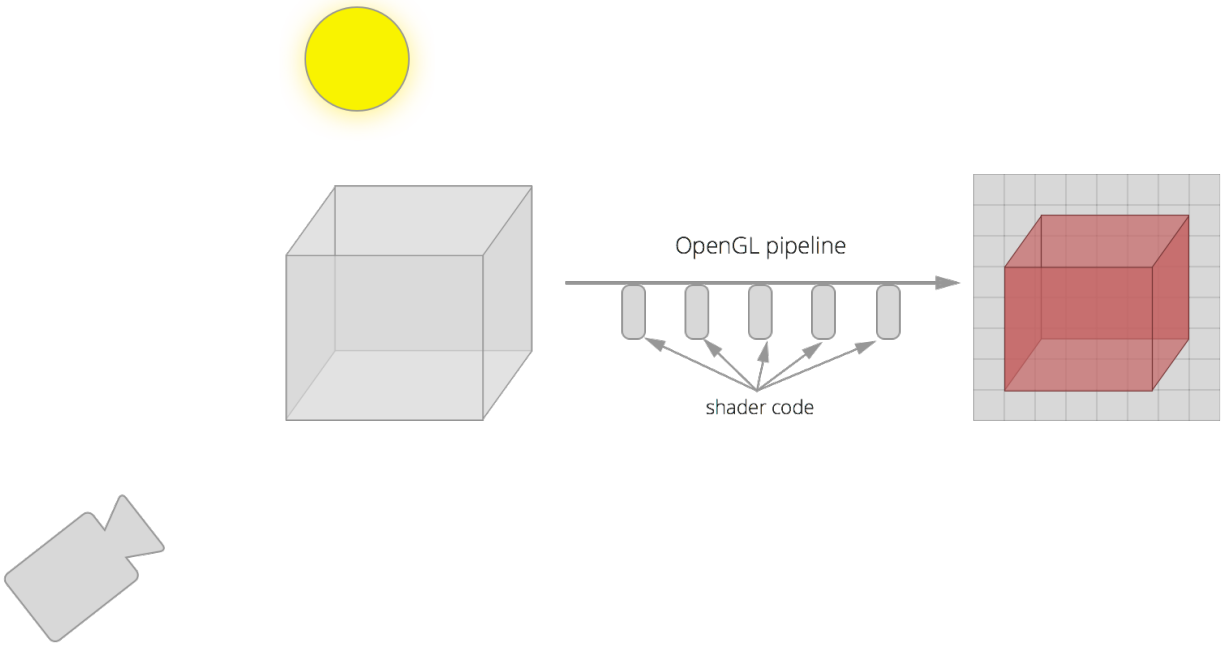
1.2.1 The deprecated model



In the original OpenGL model, we would define the state of the program (light source, vertices, colours, and camera position, amongst others). Then, we would send it to OpenGL, which would run with that state, and output an image.

Fairly simple, but lacking power.

1.2.2 The new model



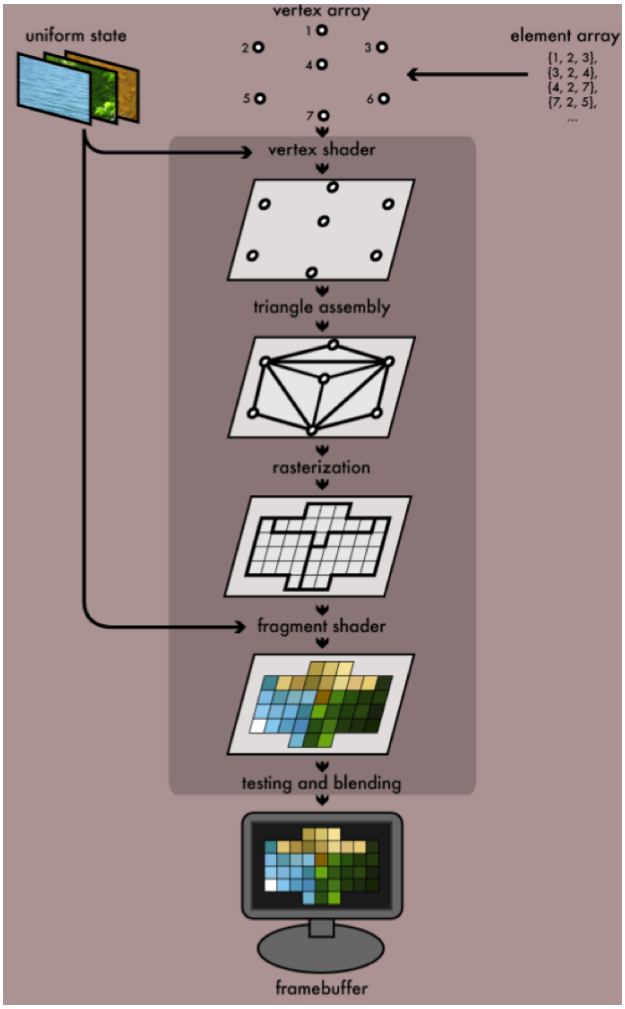
In the new model, the pipeline not only runs with the state defined before running it, but also we can give OpenGL shader code that will run at various stages of the pipeline.

For example, we can tell OpenGL some transformations to do to every vertex of our shape. We can also tell OpenGL some transformations to do to every pixel of our shape (for example, very useful to apply lighting effects).

The output is also an image.

1.2.3 The pipeline itself

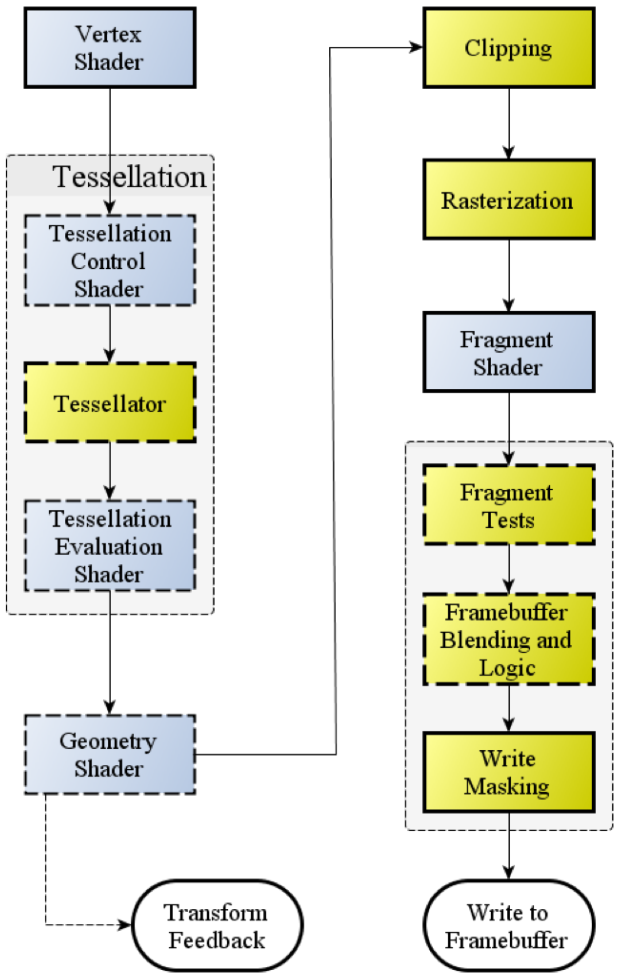
The figure below shows what happens in the pipeline (we can write shaders for each of the steps).



(see http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html)

In more detail

The figure below shows the different shaders and the pipeline itself in more detail.



(see http://www.opengl.org/wiki_132/images/RenderingPipeline.png)

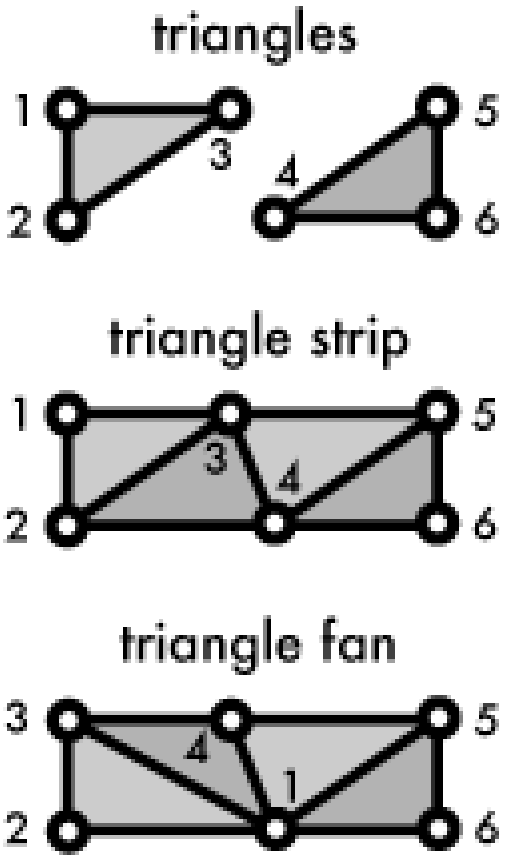
The blue boxes are where we can write shaders and pass them to the OpenGL pipeline. The yello boxes are the ones we cannot change, but happen in the pipeline.

1.2.4 Graphics journey

So we start with a number of **vertices** (in a vertex buffer) and related data (like textures: images we apply on to the planes that make up the shapes to make them look like things, such as rocks or trees).

We also have a **vertex shader**. At minimum, this **calculates the projected position of the vertex in screen space**. But it can also do other things, like generate colour or texture coordinates. The vertex shader runs for every vertex that we pass through the pipeline.

Then comes **primitive assembly**, where we create triangles from vertices. Here we can use a number of ways of assembling, such as `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN` (see below).



Then we **rasterize**, where we output pixel fragments for each triangle.

Finally, the **fragment shader** can process the pixel fragments and do things with them, like output colour and depth values. The output of this gets drawn to a framebuffer. Then we can put this in a window or used in other ways. The fragment shader runs for every pixel fragment!

So this is what happens (gif)

Vertex shader

One matrix is used for each transformation. This is each of rotating, translating, or defining views and projections (they are essentially moving the model to make it look like we are looking at it from a specific angle).

Imagine we want to translate (move) and rotate our model. We could define this with two matrices, and then we could pass these two matrices to the vertex shader, to apply them to each vertex in the model.

However, this would mean that the shader would have to calculate the final transform (translate + rotate) for each vertex. It would be more optimal to calculate the final transform in our application, and then pass that transform–in the form of a matrix–to the shader. That way it only runs once (albeit in the slower CPU, rather than the GPU).

We would also pass our shaders the projection and view matrices, although we could also combine these in the application. I find it more readable to not combine them, and so I end up with `projection`, `view`, and `model` transformation matrices.

A simple shader to execute the transforms would look like this:

```
attribute vec4 position;
uniform mat4 model, projection, view;

void main()
{
    gl_Position = projection * view * model * position;
}
```

The variable `gl_Position` is a standard name variable which is the position of the vertex at the end of the transformation, before it is passed through to the next step of the pipeline.

Optional shaders

After the vertex shaders come the optional tessellation and geometry shaders.

Tessellation shaders can add extra detail to patches of polygons.

Geometry shaders can modify, add, or remove vertices.

Primitive assembly

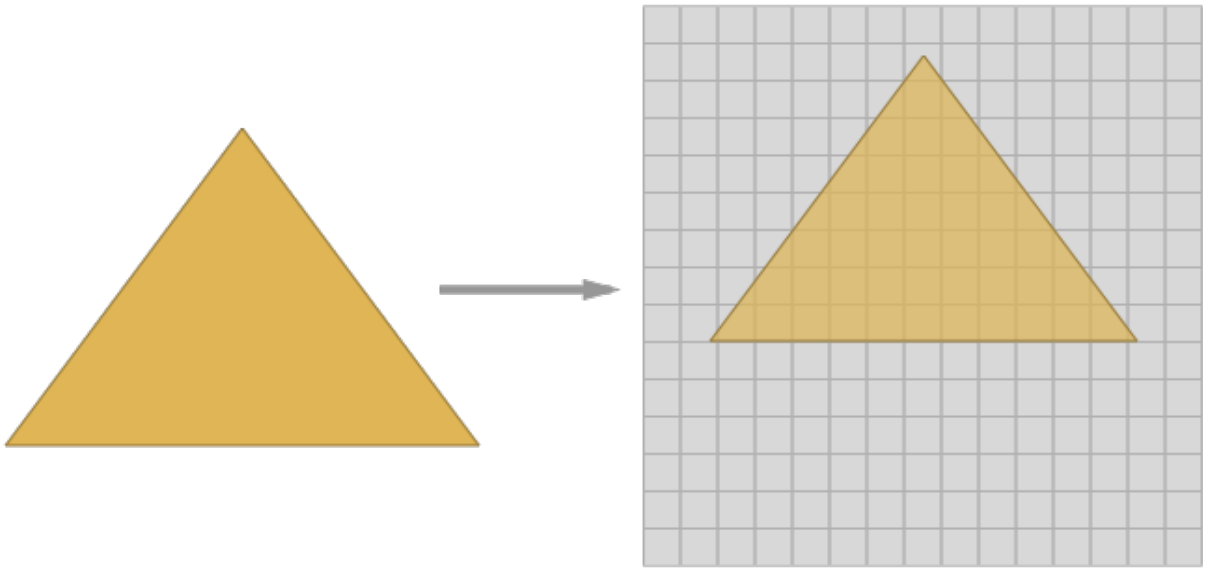
Afterwards the pipeline assembles the shapes using primitives. For example, a rectangle may be assembled using `GL_TRIANGLES` as two triangles. It may be assembled using `GL_POINTS` as four points, one on each vertex.

Clipping

Clipping is the act of discarding vertices that are outside the viewing area, so as to decrease load trying to draw things that aren't going to be visible.

Rasterization

Here the pipeline samples at the pixel level, producing pixel fragments where shapes don't occupy a full pixel. For example, if the edge of a rectangle cuts through a pixel.



Fragment shader

These shaders run for each pixel fragment that has been through the pipeline. It obtains colour and depth as interpolated values from the vertex shader.

A simple fragment shader which would just assign the interpolated colour would look something like this:

```
varying vec4 v_color;
void main()
{
    gl_FragColor = v_color;
}
```

Once again the variable `gl_FragColor` is a standard variable name that should be used as output of the fragment shader.

1.3 Development environment

OpenGL does not ship with a windowing API. This means it cannot create windows for the programs to display on. We must use a windowing API to create the windows (and allow for things like fullscreen). In this document I recommend using GLFW, but there are a number of others which you can explore:

- freeglut
- OpenTK
- Windows Forms
- Gaming environments like SDL, pygame, and others

1.3.1 Python

Throughout this document we will be looking at Python code. I will be running the examples on a Mac (which only supports OpenGL 4.1), but everything should be platform-independent.

For a Python project, you only need Python installed (Python 2.7 recommended), and an IDE (PyCharm recommended). The `requirements.txt` (read more) file for OpenGL programming in Python is:

```
PyOpenGL>=3.0
PyOpenGL_accelerate
numpy>=1.5
glfw==1.0.1
```

1.3.2 Go

<Waiting for PR...>

1.4 Drawing

I recommend reading this before continuing.

There are a number of steps in drawing:

- *VAO & VBO*
- *Binding objects*
- *Vertex attributes*
- *Uniforms*
- *Vertex and fragment shaders*
  - *Vertex*
  - *Fragment*
- *Other shaders*
  - *Geometry*
  - *Tessellation*

1.4.1 VAO & VBO

When drawing we start with a Vertex Array Object (VAO) and Vertex Buffer Objects (VBOs).

Good information regarding VAO and VBO can be found here.

The gist of it is that Vertex Array Object is a really bad name for what that VAO is. It may be better to call it a Vertex Buffer Array, as that tells us more of what it does.

Essentially the VAO holds what data we are going to be sending the OpenGL pipeline. So, usually we need to send the pipeline data like vertices and colours we are drawing, but potentially more stuff as well like textures or normal maps.

Those attributes, like vertices, colours, textures, and more are stored in the Vertex Buffer Object (VBO).

So it goes in this order:

1. Generate Vertex Array Object
2. Bind Vertex Array Object
3. Generate Vertex Buffer Object
4. Bind Vertex Buffer Object

Steps 1 and 2 are as follows:

```
glGenVertexArrays(1, &vaoID[0]); // Create our Vertex Array Object
glBindVertexArray(vaoID[0]); // Bind our Vertex Array Object so we can use it
```

1.4.2 Binding objects

Steps 3 and 4 above are the 'Binding Objects' part:

```
glGenBuffers(1, vboID); // Generate our Vertex Buffer Object
glBindBuffer(GL_ARRAY_BUFFER, vboID[0]); // Bind our Vertex Buffer Object
glBufferData(GL_ARRAY_BUFFER, 18 * sizeof(GLfloat), vertices, GL_STATIC_DRAW); // Set␣
↪the size and data of our VBO and set it to STATIC_DRAW
```

So once steps 1 and 2 have been executed the VAO is bound, which means it is the current VAO we are modifying. All VBOs we create and pass data to (steps 3 and 4) will go into that VAO.

Once we have done step 4, we need to fill our VBO with the vertex data. When creating VBO's, you can set what type of VBO it is, and in this case we are going for `GL_STATIC_DRAW`, this tells OpenGL that we do not intend on changing the data in any way, and we just want to be able to render it.

1.4.3 Vertex attributes

When we have bound the buffer, we still need to tell OpenGL what it is that we have bound, and we have to tell it which variable in our Vertex Shader it should be assigned to.

What we do is the following:

```
loc = glGetAttribLocation(program, "position")
glEnableVertexAttribArray(loc)
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, 0)
```

That gets the index of the `position` variable in our Vertex Shader. Then, we enable the `loc` index of our currently bound VAO. Finally, we state that the element in position `loc` of our VAO is:

- Of size 3 (e.g. each vertex has 3 elements to it);
- Each element of the vertex is of type `GL_FLOAT`;
- We do not want OpenGL to normalize the values for us;
- The stride is 0; and
- The offset is 0.

We want to use stride and offset if we want to send multiple different pieces of data in the same buffer.

For example, we would store both position and colour in the same buffer, like so:

```
[(-1, 1, 1, 1), (0.235, 0.677, 0.9), (1, -1, -1, 1), (0.113, 0.199, 0.53)]
```

Imagine every even element (`0` and `2`) are position, whereas the others are colour.

We could then pass in the values like so:

```
glBindBuffer(GL_ARRAY_BUFFER, self.buffer)  # Bind the buffer that contains both data

loc = glGetAttribLocation(program, "position")
glEnableVertexAttribArray(loc)
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 1, 0)  # Give stride of 1 so it␣
↪skips one element for every element it puts in

loc = glGetAttribLocation(program, "colour")
glEnableVertexAttribArray(loc)
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 1, 1)  # As above, but starting at␣
↪index 1, makes it get the odd elements only
```

1.4.4 Uniforms

Uniforms are just variables that don't change during the OpenGL rendering.

We can pass them in to our program to define custom state.

For example, we could use an uniform `vec3 colour` variable if we wanted to only use that one colour for all our objects that get drawn using the shader.

We can pass uniform values to our shaders like so:

```
color_mode_id = glGetUniformLocation(compiled_shader_program, "colormode")
glUniform1i(color_mode_id, 1)
```

There are many `glUniform<xx>` functions.

The number specifies how large the variable is (only accepts value 1 unless the letter `v` is at the end). If we pass `1` then it is just a number, anything else is a vertex of size `x`. The letter at the end specifies the type. `i` is an integer, `f` is a float, `v` is a vector.

1.4.5 Vertex and fragment shaders

Vertex

The inputs to the vertex shader are vertices and their related attributes.

The outputs of the vertex shader are:

- Clip-space vertex positions;
- Texture coordinates;
- Point sizes;
- Colours and fog coordinates; and
- Potentially other custom vertex attributes as well.

At minimum, they must return the **clip-space vertex positions**!

Fragment

These calculate the colours of the individual pixel fragments.

Gets the input from the rasterizer (which fills in the polygons being sent through the graphics pipeline).

Typically used for lighting effects, shadows, bump mapping, and colour toning.

Inputs:

- Pixel fragments and all their attributes.

Outputs:

- Pixel fragment colours.

1.4.6 Other shaders

Geometry

- Can add and remove vertices from a mesh
- Can be used to generate geometry procedurally
- Can add detail to existing meshes
- Output is then sent to the rasteriser.

Tessellation

- Geometry split into "patches". Tessellation shader inputs and outputs patches.
- Can be used to increase resolution of a geometry by adding patches or splitting a patch into multiple patches.

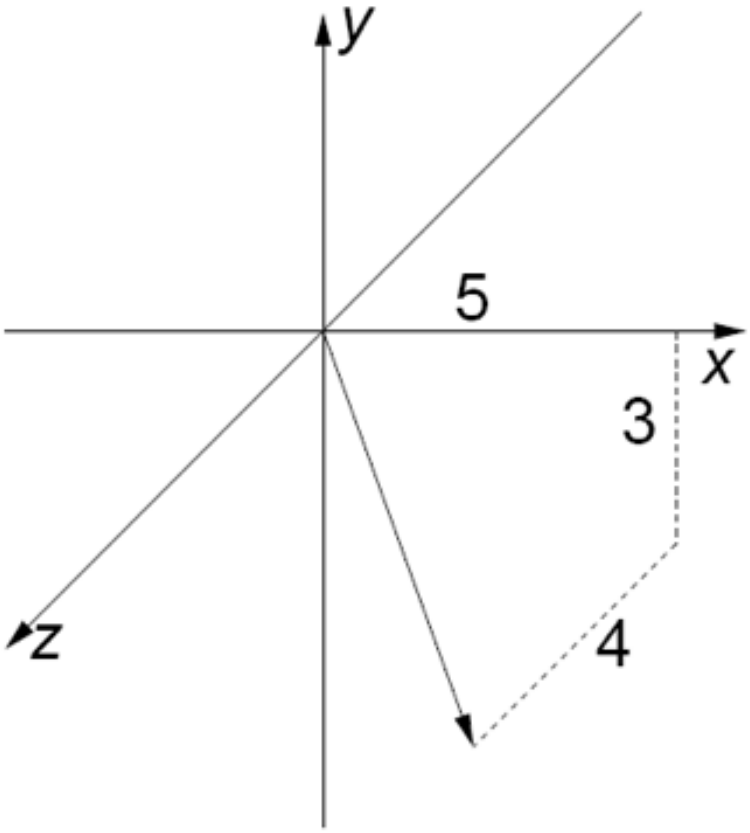
CHAPTER 2

Transforms and 3D

Contents:

2.1 Vectors

Vectors are simply 2, 3, or 4 dimension elements. In OpenGL we often use them to specify position, direction, velocity, colour, or a lot more.



$$\begin{bmatrix} 5 \\ -3 \\ 4 \end{bmatrix}$$

The axes as shown in the image above are the default OpenGL axes. However, it is possible to change them, inverting the Z axes.

2.1.1 Vector operations

To add or subtract vectors, simply each of the dimensions together.

The **length** of a vector is given by the equation $|V| = \sqrt{x^2 + y^2 + z^2}$.

To scale a vector (multiply a vector by a scalar), just multiple each of the dimensions by the scalar.

To normalise a vector, divide the vector by its length, like so

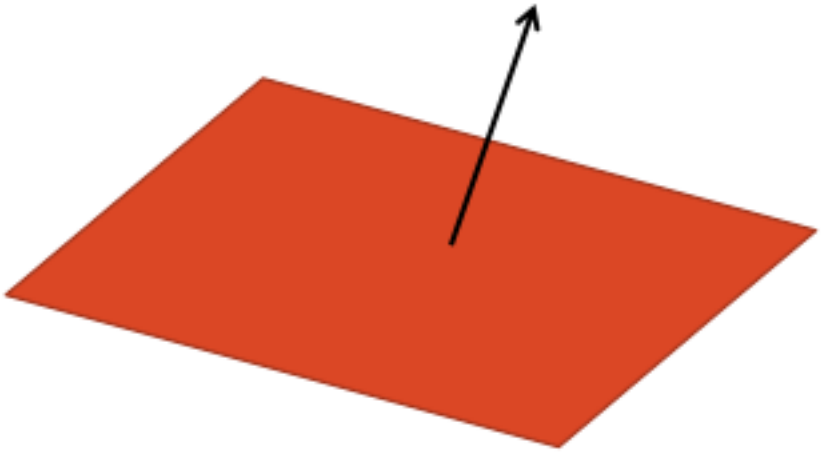
$$U = \frac{V}{|V|}$$

2.1.2 Normal vectors

Normal vectors can be normalised, but they are not the same thing!

A normal vector specifies the direction which a surface is looking towards.

For example, in a cube you would want the light to shine on the outside of the cube, and not on the inside. What to do is define the normals for each of the vertices of the cube to be facing outwards from the cube. If you made them face inwards, then the cube would be lit on the inside, and would look black on the outside.



The image above shows the normal vector for a plane.

Remember that the normal vector is the direction the plane is looking towards. If we change the normal vector without changing the rotation of the plane, the lighting would change, but it may not look physically realistic.

2.1.3 Dot Product of Two Vectors

To calculate the dot product of two vectors, watch the video here.

The dot product can be calculated as: a.b = |a|×|b| × cos(θ)$Orasthis : a.b = a_x \times b_x + a_y \times b_ySotherefore itcanbeuseful, if weknowthevectors, wecanrearrangetheequationstocalculatetheanglebetweenthetwovectors.$

So, therefore:

A more comprehensive explanation of why that works is here.

2.1.4 Cross Product of Two Vectors

To calculate the cross product of two vectors, watch the video here.

The cross product can be calculated as: a= |a|×|b| × sin() × $nWhere$:

|a| is the magnitude (length) of vector a

|b| is the magnitude (length) of vector b

is the angle between a and b

n‘$istheunitvectoratrightanglestoboth : math :$ ‘a and b

So we can re-organise that equation to calculate n.

Thus the cross product can be used along the dot product to calculate a unit vector perpendicular to a plane.

What this is useful for, is to calculate normal vectors of a plane, when we only have points directly on the plane to work with. This will be extremely useful to calculate lighting later on.

A more comprehensive explanation of why that works is here.

2.1.5 Homogeneous Coordinates

Homogeneous coordinates are just 3D vectors that instead of 3 dimensions have 4 dimensions. Usually the 4-th coordinate is `1`.

This is used for things like translations (we will see soon), and to define whether a vector is simply a direction (`w == 0`) or a position (`w != 0`).

Thus the vector `(x, y, z, w)` corresponds in 3D to `(x/w, y/w, z/w)`.

2.2 Transformations

Every object has a matrix associated with it, so that we can easily apply transformations to it and pass it to the shader to determine the correct position of the object.

Before applying any transformations, an object must have the identity matrix associated with it.

This matrix does not apply any transformations. When a matrix is multiplied by the identity matrix, the result is the original matrix. The identity matrix looks like this: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ Multiplying the identity matrix by another matrix results in the matrix being returned: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 2 & 0 & 1 \\ 4 & 2 & 2 & 0 \\ 4 & 1 & 3 & 0 \\ 0 & 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 2 & 0 & 1 \\ 4 & 2 & 2 & 0 \\ 4 & 1 & 3 & 0 \\ 0 & 2 & 2 & 2 \end{bmatrix}$

2.2.1 Simple transformations

Translation

A translation matrix looks like this: $\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$ We can multiply a position vector by a translation matrix and see that we end with a translated vector: $\begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 5 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*3+5*1 \\ 1*5+1*1 \\ 1*2+1*1 \\ 1 \end{bmatrix} = \begin{bmatrix} 8 \\ 6 \\ 3 \\ 1 \end{bmatrix}$

Rotation

A rotation matrix rotates a vertex around a line by θ degrees.

A matrix to rotate a vertex around the `x` axis: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ A matrix to rotate a vertex around the `y` axis: $\begin{bmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ A matrix to rotate a vertex around the `z` axis: $\begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Scaling

A scaling matrix will multiply the vertices to expand or reduce the size of a polygon. The matrix below, if applied to a group of vertices making a polygon, would expand the polygon by a factor of 3. $\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

2.2.2 Combining transformations

We can combine transformations, such as scaling an object and then rotating it. Remember that the order in which we apply transformations does matter.

For example, if we have an object at position `(0, 0, 0)` (the origin), and we rotate it around the origin, it will rotate in place. Then we can move the rotated object elsewhere.

However, if we first move the object and then rotate it around the origin, the object won't rotate around itself, it will in fact move.

If we want to combine transformations, **we much apply transformations in reverse order to which we want them to happen**.

If we wanted to first rotate and object and then scale it, this would be the order to do it in:

$$scale * rotate * objectmatrix$$

So the matrices would look something like this (scaling by 3 and rotating by 45°): $\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(45) & -sin(45) & 0 \\ 0 & sin(45) & cos(45) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ And this would give you the model matrix for your object: $\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 2.12 & 0 & 0 \\ 0 & 0 & 2.12 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

2.3 Viewing

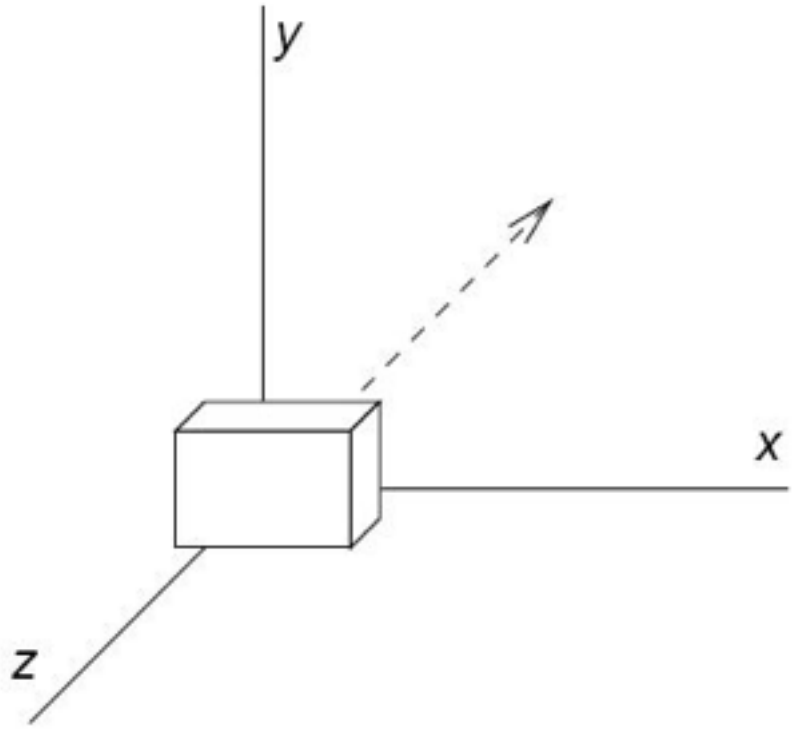
Once we have constructed the 3D model, we need to define the viewpoint. To use an analogy of a camera, we need to choose:

- Where to put the camera;
- Which way up to hold the camera; and
- Where to point the camera to.

The defaults are:

- Camera at the origin `(0, 0, 0)`;
- Pointing down the Z axis; and
- With the up position being the Y axis.

Something like this:



If we turn the camera upside moving, so it is help with the "up" being down the Y axis, then everything would appear upside down.

2.3.1 Moving the camera

Moving the camera is a bit of a funny one.

We do not move the camera in OpenGL. We move everything else, but the camera "stays in place".

So if we wanted to move the camera **up**, we would move everything that we can see **down**. If we want the camera to zoom in on something, we move that something **closer**. If we want the camera to move away from something, we move that something **away**.

2.3.2 Perspective

However, moving something closer to us or away from us doesn't make it bigger or smaller, at least not in computer world.

What happens is that things that are closer to us will block things that are farther away, if they are in front of each other.

There is a way to make things far away smaller, though, which is simply by shrinking them as they get farther away.

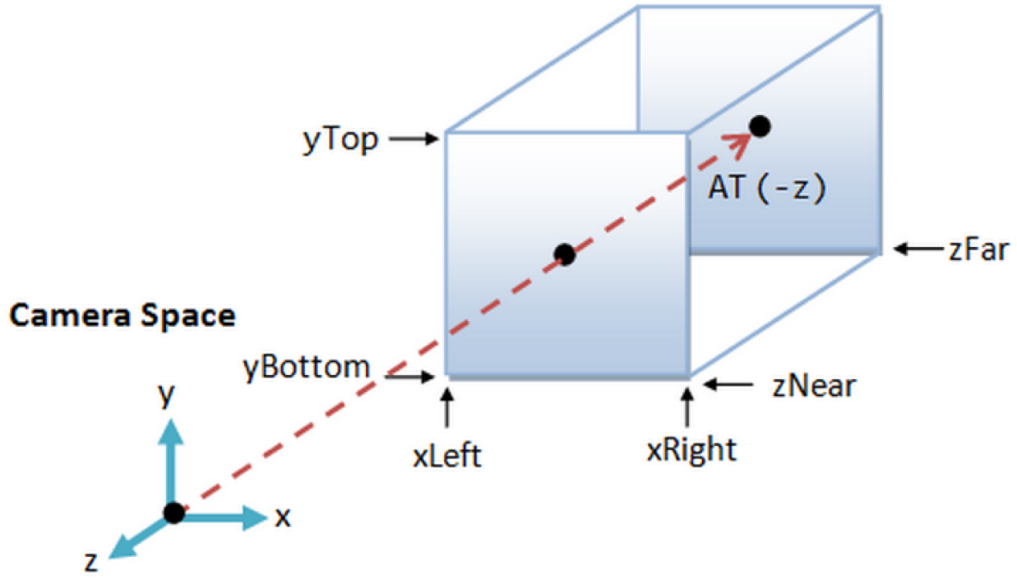
We can do that using a different projection:

- Parallel or Orthographic (things don't get smaller as they move farther away); or

- Perspective (things get smaller as they move farther away).

Orthographic Projection

In Orthographic Projection, we imagine we are drawing things in a cube. Things outside the cube don't get drawn (**clipping**), but things that are closer to the camera or far away from the camera are the same size.

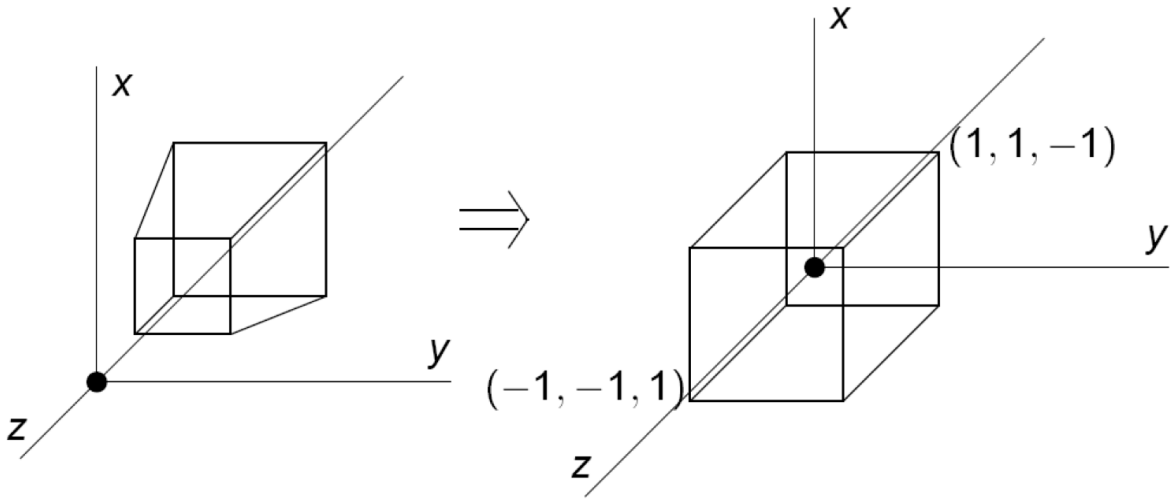


Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

Perspective Projection

In Perspective Projection, we imagine we are drawing things on a frustum instead of a cube. Then, we convert the frustum to a cube once we have finished drawing.

What this achieves is that things that are further away from the camera get shrunk, whereas things that are closer to the camera get enlarged. Things that are outside the frustum get clipped.



2.3.3 Clipping

Finally, after the projection has been set and the transformation applied, we can do clipping and remove vertices that are outside the viewing volume.

2.3.4 Viewport

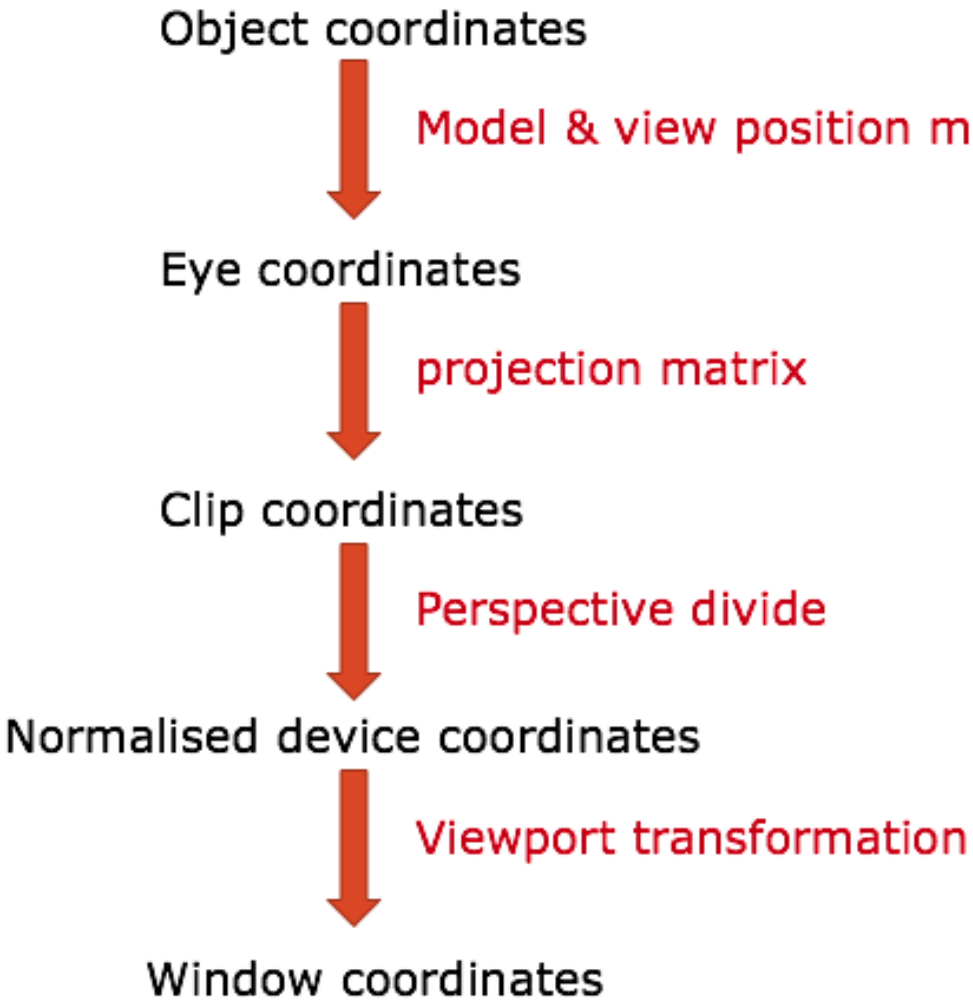
And after, we have to transform the image into an actual screen image, which is drawn in the window.

It is important that the viewport has the same aspect ratio as the viewing volume, so this conversion does not distort the image.

2.3.5 To Summarise

1. We define our object coordinates and apply transformations to them;
2. We apply the view transformation;
3. We apply the perspective transformations;
4. We clip the coordinates;
5. We apply the perspective divide, which converts coordinates into ordinary 3-d coordinates in the range of `(-1, 1)`;
6. We calculate normalised device coordinates which converts frustum into cube and ensures z-buffer culling[1];
7. We apply the viewport transformation to convert view into window coordinates.

[1] z-buffer culling is the removal of vertices that are behind other vertices, and so cannot be seen.


Object coordinates
Model & view position matrix
Eye coordinates
projection matrix
Clip coordinates
Perspective divide
Normalised device coordinates
Viewport transformation
Window coordinates

CHAPTER 3

Lighting

Contents:

3.1 Lighting

3.1.1 Phong Lighting Model

Lighting is based on the Phong model, which states that for every object, we can see light reflected from it in three ways:

- Ambient light;
- Diffuse reflection; and
- Specular reflection.

Ambient light

Usually light gets reflected from surfaces and walls, so entire scenes have a small amount of light even when they are not pointing towards the light source.

We can model that (unrealistically) using ambient light: giving the entire scene a small amount of light that is applied to all objects.

This is not realistic, but does a reasonably good job and does not entail a performance loss.

Diffuse reflection

This is light that gets reflected in all directions when a light ray hits a surface.

When the surface is parallel to the ray of light, it does not get hit, and so no diffusion can take place.

When the surface is perpendicular to the ray, maximum diffusion takes place. This follows **Lambert's Law**[1]: $R_d = I_d K_d cos(\theta) = I_d K_d (N.L) Where$:

I_d is the intensity of the incoming light ray.

θ is the angle between the light and the surface normal.

K_d is the diffusion constant, the "non-glossyness" of the surface.

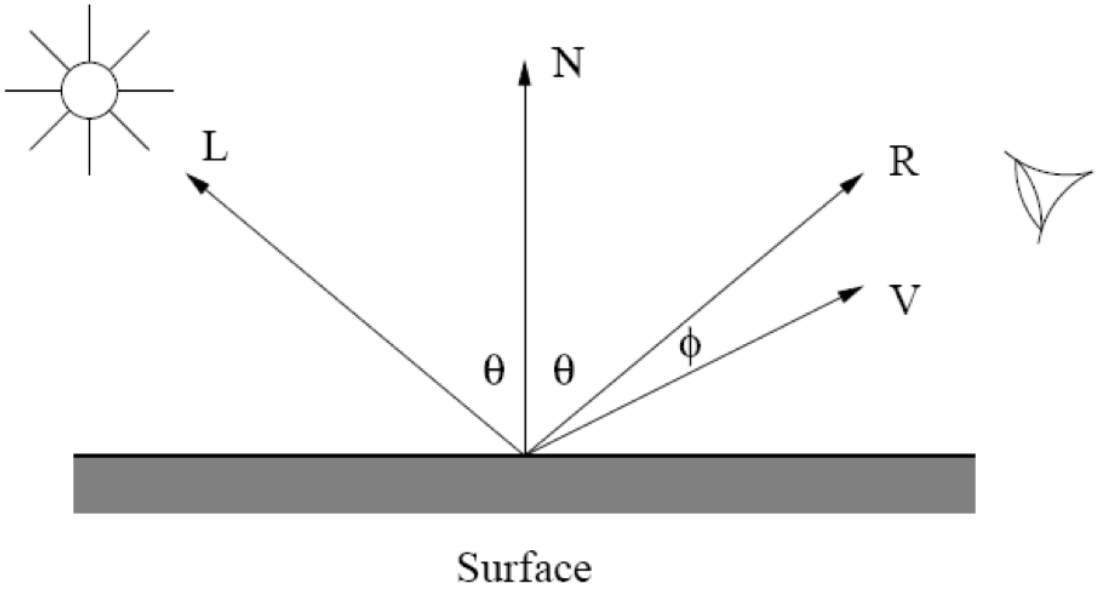
N is the surface normal vector.

L is the direction of the light coming from the light source.

Specular reflection

A **Specular Highlight** is the reflection we see when the angle of incidence equals the angle of reflection, and our eye is in position at that angle to see the reflection.

For a perfectly glossy surface, the specular highlight can only be seen for a small θ, as in below:



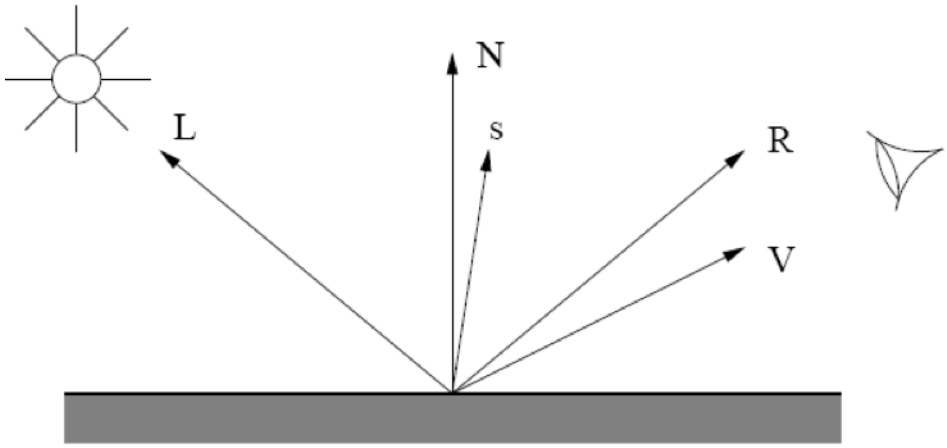
However, for a non-glossy surface, there is a range of ϕ where the specular highlight can be seen.

The equation is as follows: $R_s = I_s K_s cos^n(\theta) Phong suggested a large value of n$ for a large range of ϕ, and a small value of n for a small range.

Blinn-Phong approximation

The Blinn-Phong approximation is used in OpenGL as a means to simulate the equation but made simpler.

[1] Lambert's Law states that the intensity of the diffused light is proportional to the cosine of the angle between two vectors, the surface normal and the direction of the light source.



Instead of using the angle between R and V, we calculate S which is between L and V. Then, we use the angle between S and N.

Emitted light

An object can emmit light, and this simply means that we program it to be bright, as if it were a source of light.

Remember that making an object bright does not mean it will be making other objects bright. A source of light and an object that looks lit can be different things in OpenGL.

Attenuation

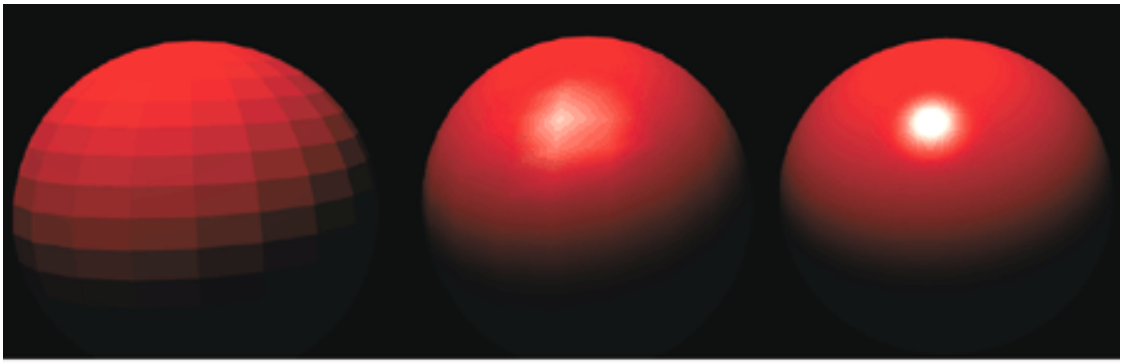
The intensity of the light must decrease the further away from the light we are, unless the light is directional, and not positional.

Although light intensity can be calculated with a number of formulas (e.g. the square of the distance from the light), we can use a more complex formula when programming graphics: $1 \frac{}{k_c + k_l d + k_q d^2 Where k_c}$, k_l, and k_q are constants, and d is the distance from the light for any given position.

3.2 Shading

Shading is applying lighting effects in OpenGL code. The three ways below are in increasing realism, but also each take a higher performance hit.

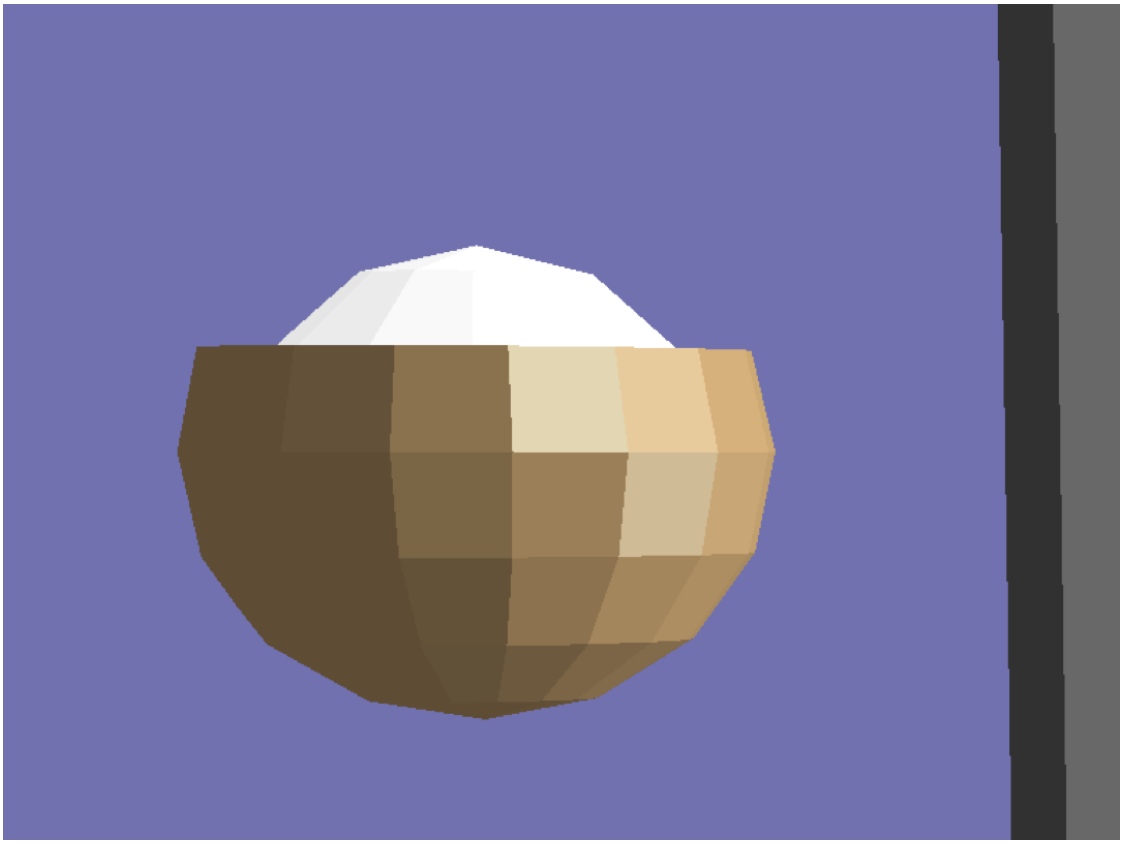
In summary, it's something like this:



3.2.1 Flat Shading

Define a normal for each plane (a.k.a. polygon) that makes up your shape. Then, calculate lighting for each plane.

This will give you **flat shading**, as shown in the image below.



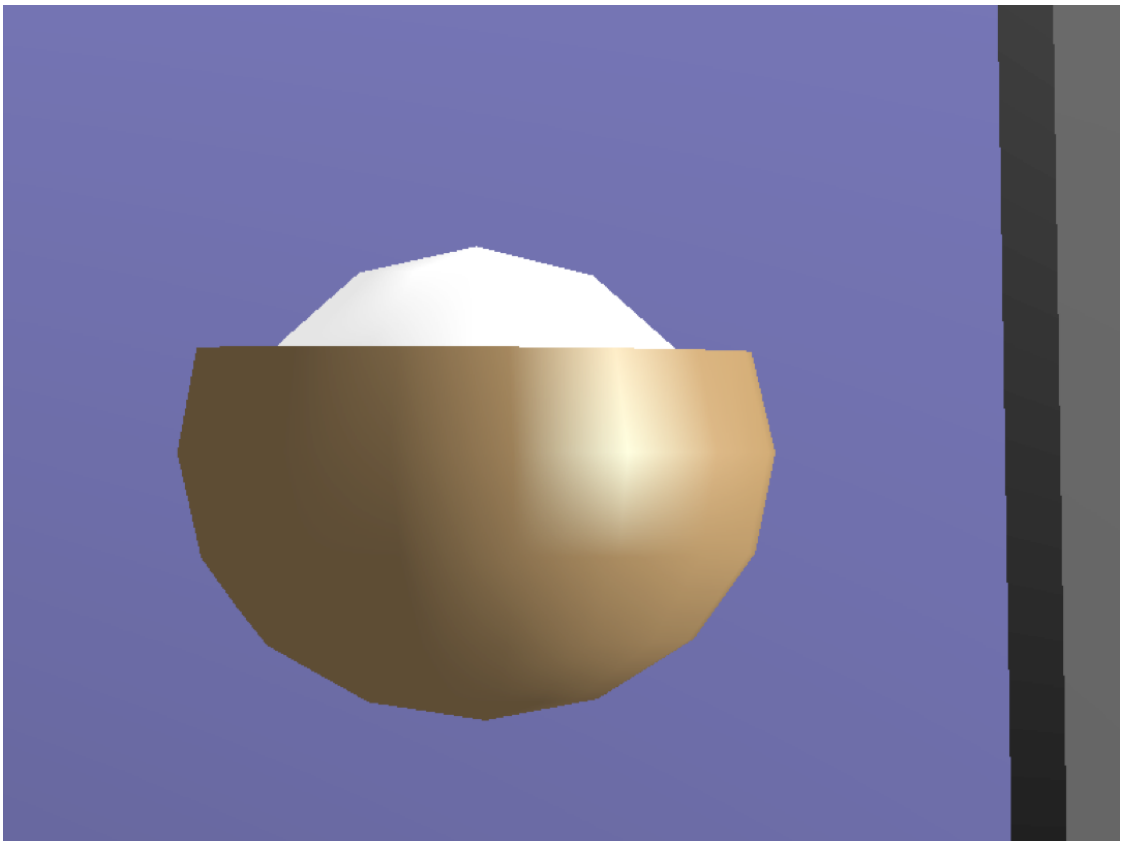
3.2.2 Gouraud Shading

Define normals at each vertex and use them to calculate lighting

Because the lighting gets calculated at a vertex level, you will see specular highlights appear and disappear

quickly. However, it is cheaper to do than Phong shading (although Phong is acceptably cheap with modern hardware).

Below, we can see how the specular highlight follows the vertices as opposed to being truly smooth.



So, in order to implement Gouraud shading, the following steps could be followed:

In application 1. Define vertex normal and colours (material properties) 2. Define light source position and colour (and optional intensity) 3. Calculate NormalMatrix transform (or can do in vertex shader)

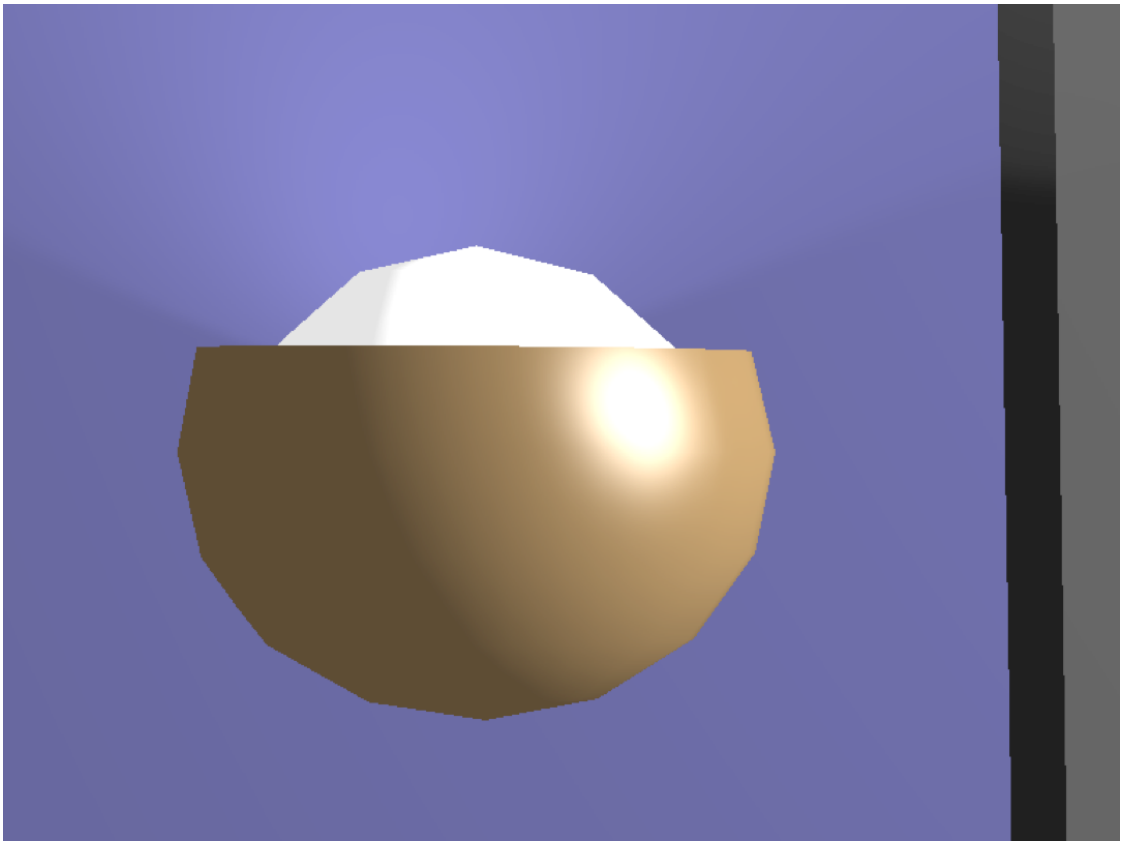
In vertex shader 1. Calculate transformed vertex position 2. Calculate transformed normal 3. Calculate light direction to vertex 4. Calculate diffuse component 5. Calculate specular component 6. Calculate ambient component 7. Calculate vertex colour (from lighting values and vertex material colour)

3.2.3 Phong Shading

In Phong shading, the normals are defined at vertex level but lighting is not calculated at vertex level.

Instead, the normals are interpolated between vertices, and lighting is calculated at a per-pixel-fragment level.

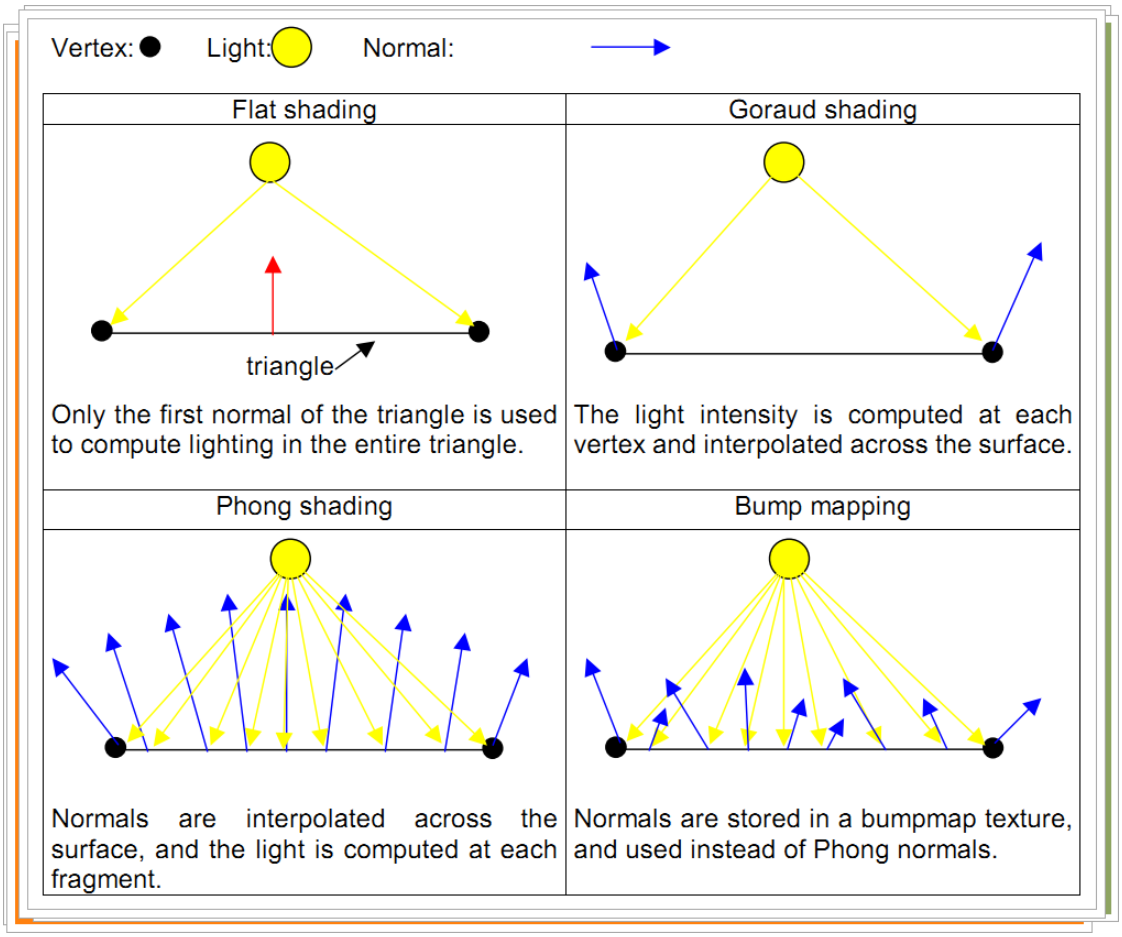
This means more calculations, and thus it more computationally expensive, but the results are substantially more realistic, and it is more likely to capture specular highlights.



3.2.4 Comparison of shading and normals

Please see original post here (in Chinese).

The image below summarises the three main shading models, and also includes bump mapping (which we will look at later on).


Vertex:
Light:
Normal:
Flat shading
triangle
Only the first normal of the triangle is used to compute lighting in the entire triangle.
Goraud shading
The light intensity is computed at each vertex and interpolated across the surface.
Phong shading
Normals are interpolated across the surface, and the light is computed at each fragment.
Bump mapping
Normals are stored in a bumpmap texture, and used instead of Phong normals.

CHAPTER 4

Texture and coloring

Contents:

4.1 Texturing

Texturing or texture mapping is extremely important for realism.

Simply it is attaching a picture to a polygon. It can give the impression of materials, like wood or granite.

Each point of the image is called a "texel" in analogy to pixels, and the image coordinates range from `(0, 0)` (bottom-left) to `(1, 1)` (top-right).

4.1.1 Multi-texturing

Can bind textures to different OpenGL texture elements. Each also needs a separate texture sampler, for example:

```
glActiveTexture(GL_TEXTURE0)  # This is for the first texture element
glBindTexture(GL_TEXTURE_2D, textureID0)
glBindSampler(GL_TEXTURE0, textureSamplerID0)

glActiveTexture(GL_TEXTURE1)  # This is for the second texture element
glBindTexture(GL_TEXTURE_2D, textureID1)  # And here we bind to it the␣
↪texture with id textureID1
glBindSampler(GL_TEXTURE1, textureSamplerID1)
```

Then, the fragment shader code could look something like this:

```
in vec4 fcolour;
in vec2 ftexcoord;
out vec4 outputColor;

layout(binding = 0) uniform sampler2D tex1;
```

```
layout(binding = 1) uniform sampler2D tex2;

void main()
{
 vec4 texcolour1 = texture(tex1, ftexcoord);
 vec4 texcolour2 = texture(tex2, ftexcoord);
 outputColor = fcolour * (texcolour1 * texcolour2);
}
```

4.1.2 Anti-aliasing textures (mipmapping)

Mipmapping is creating or providing the same texture at different resolutions, for use when the texture is rendered at a distance.

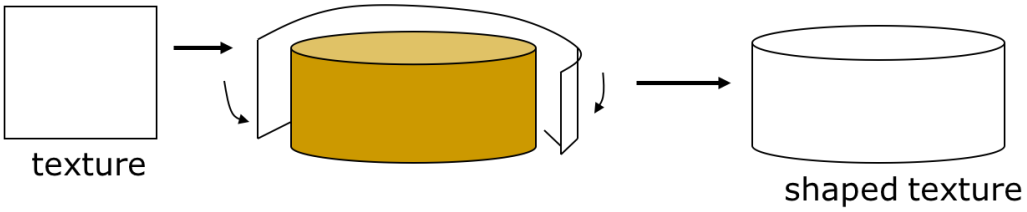
For example, the same texture could be provided (or calculated) at 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, and 1x1.

4.1.3 Mapping textures to surfaces

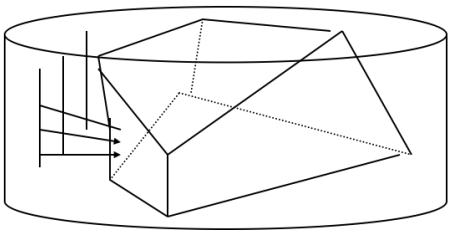
Distortion often results, especially if mapping to a sphere, for example.

For simple polygons, mapping the texture can be very simple: just define texture coordinates for each vertex.

For more complicated shapes things can get tricky. Something we can do is shrink-wrap mapping:



Then, 'shrink-wrapped' onto object



4.2 Blending, Aliasing, and Fog

4.2.1 Blending

When blending is enabled, the colour of surfaces can be combined, if the surface in front is not completely opaque.

This can be calculated using the alpha value of the colour. Instead of RGB, we can use RGBA, where the `A` is the opacity of the colour, ranging from 0 (transparent) to 1 (opaque).

We need to enable it in OpenGL:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
```

Remember the result also depends on which object is in front and which one is behind.

It usually is best to draw objects far away first, as that will result in more realism.

4.2.2 Fog

Normally, objects far away would not be as sharp and clear as objects close by. We can simulate this with "fog".

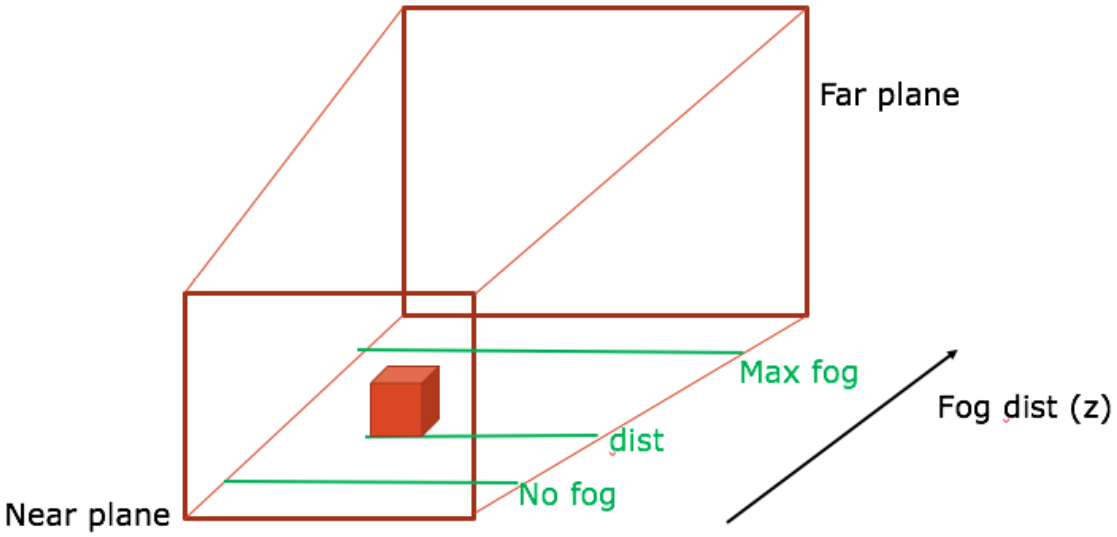
Fog is just an adjustement of the colour of a pixel based on the distance from the camera.

It can also give an impression of depth.

It can be generally implemented as follows:

1. Define the clear colour to be your fog colour
2. Pass a distance value to the fragment shader in some way
3. Define fog parameters
   - Min distance
   - Max distance
   - Fog colour – make the same as the clear colour
   - Fog relationship, e.g. linear or exponential
4. Calculate fog factor from depth distance (in eye coordinates)
5. Ensure the fog factor is in the range 0 (no fog) to 1 (max fog)
   - `clamp()` function is useful
6. Mix the lit, shaded colour with the fog colour
   - `mix()` function is useful

```
float fog_factor = (fog_maxdist - dist) /
                   (fog_maxdist - fog_mindist);
```



In the fragment shader, we could have code like follows:

```
// Fog parameters, could make them uniforms and pass them into the fragment
↪shader
float fog_maxdist = 8.0;
float fog_mindist = 0.1;
vec4  fog_colour = vec4(0.4, 0.4, 0.4, 1.0);

// Calculate fog
float dist = length(fposition.xyz);
float fog_factor = (fog_maxdist - dist) /
                   (fog_maxdist - fog_mindist);
fog_factor = clamp(fog_factor, 0.0, 1.0);

outputColor = mix(fog_colour, shadedColor, fog_factor);
```

The fog blending factor is usually a simple equation, which could be one of the following, where d is the density of the fog.

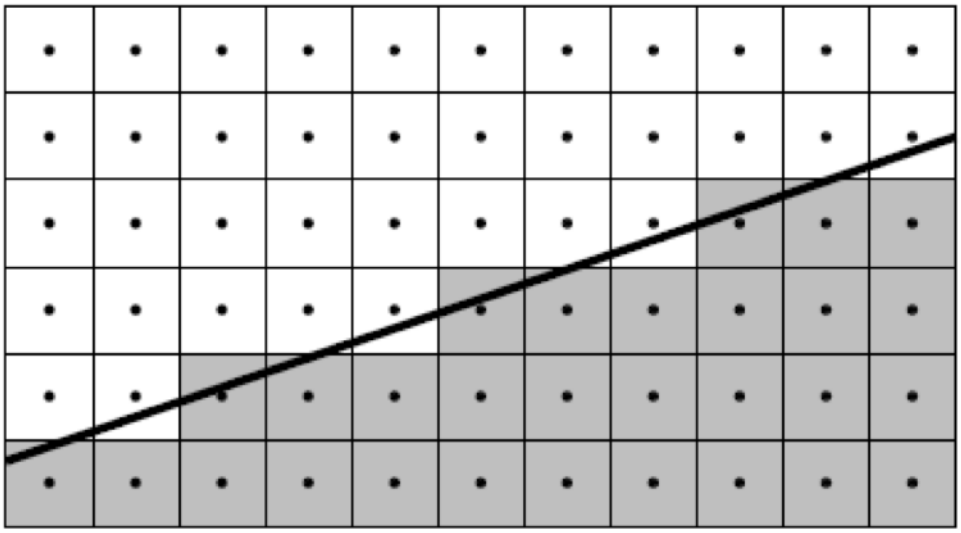
Below, fog density increases rapidly with distance: f = e(-d$^{z)Below,fogislessdenseatshortrange,butincreasesevenmorerapidlywithdistance:f=e(-d_z)^2Below,foggreateratshortdistanceandincre}{(end-start)}$

4.2.3 Aliasing

> General term for artefacts in computer graphics images, caused by problems with regular sampling of the image model.

Antialiasing are measures to counteract that.

Here's what aliasing looks like:



Antialiasing

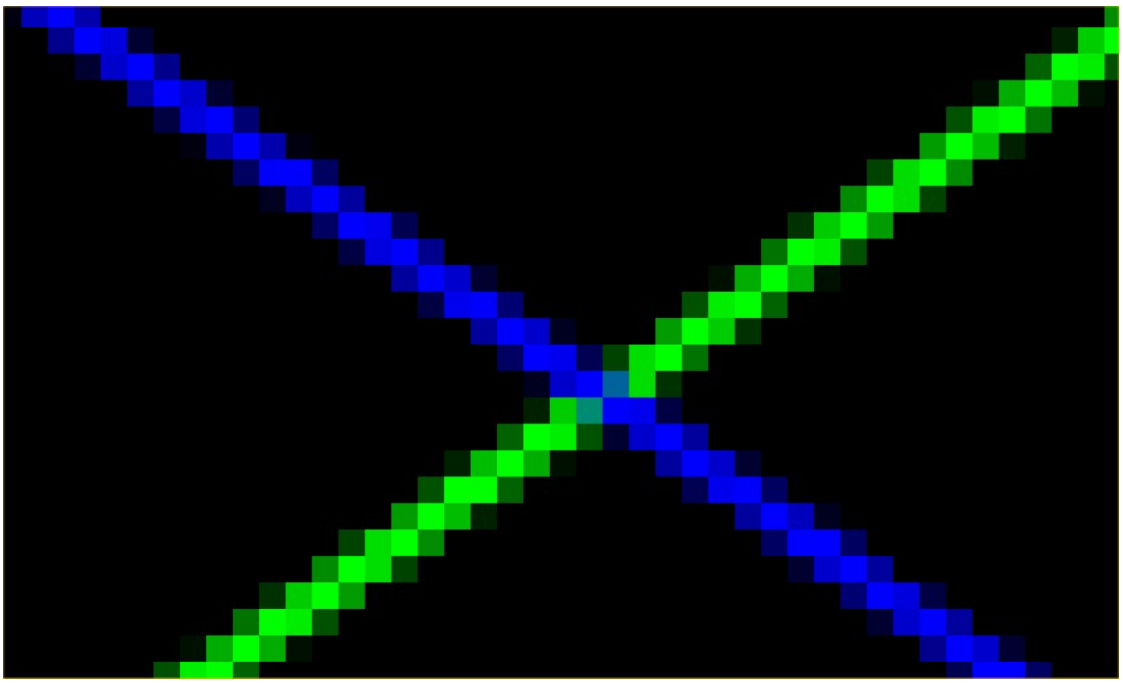
Drawing lines: colour the pixel if its centre lies on the line.

The obvious problem with this is that if the line isn't exactly over the center of a pixel, we don't draw the pixel at all.

So what to do is calculate how much of the line is over a pixel, and then colour the pixel in an amount equivalent to the percentage covered multiplied by the colour. This is done by **blending**, actually.

The coverage value (from 0.0 to 1.0) is multiplied by the alpha value of the colour, and then the pixel is coloured with that alpha value.

When the line is not lying fully over the pixel, the pixel becomes slightly more transparent.



Lines

For lines, it is easy enough in OpenGL. We just have to pass a hint to the OpenGL pipeline to do anti-aliasing of lines:

```
glEnable(GL_LINE_SMOOTH);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
```

Super-Sampling

Involves generating a virtual image at a higher resolution (e.g. 3x or 5x the resolution). Then we can calculate the colour of pixels in the original image as the average between the pixels that would correspond to that position in the extra-resolution image.

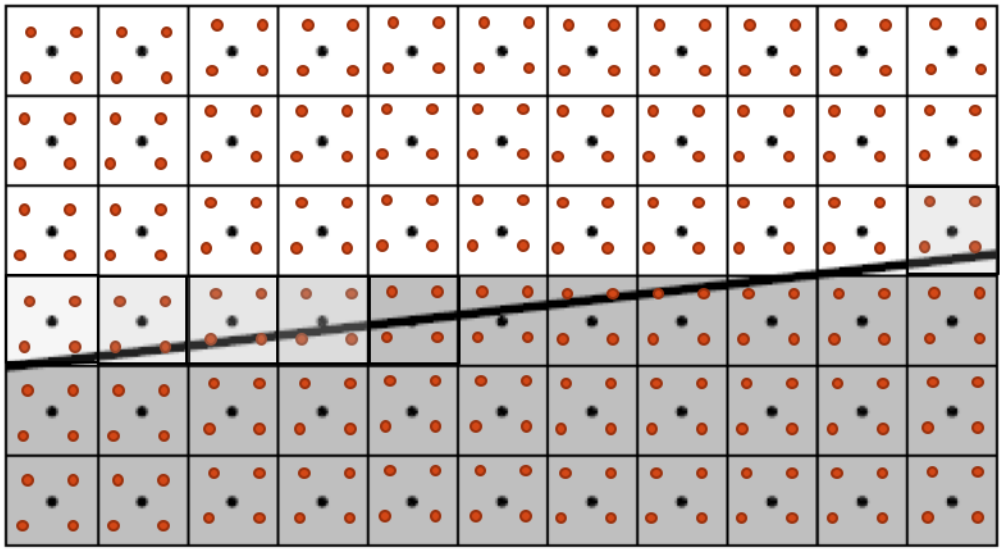
It entails a performance hit, so it is no longer very popular.

Multi-Sampling

Multi-sampling is a good general solution to aliasing, but it comes with some caveats:

- Performance hit;
- Disables line smoothing;
- Not available under all circumstances; and
- Effect will vary with the implementation.

Just sample each pixel multiple times at different positions, and calculate the colour based on that:



Simply send the `GLFW_SAMPLES` window hint when creating the window:

```
glfw.init()

glfw.window_hint(glfw.SAMPLES, 4)
```

Then, enable `GL_MULTISAMPLING`:

```
glEnable(GL_MULTISAMPLING)
```

CHAPTER 5

Complex objects

Contents:

5.1 Importing 3D objects

We can use a package like the following:

- Blender
- 3DS Max
- Autodesk Maya
- Houdini

We will also need a way to load the objects these packages create:

- Assimp
- Load Blender .obj files (essentially text files)
- Lib3ds

Assimp relies on D3D9, so it is not recommended unless you are using windows. Reading in the text files by creating your own object loader is recommended.

5.1.1 .obj files

The .obj file exported from Blender will look something like this:

```
# Blender
o ModelName
v 0.5 0 1
v 1.0 0 1
v 0.6777772 0.5 1
f 1 2 0
```

```
f 2 0 1
f 1 0 2
```

A line starting with `o` defines the model name, one starting with `v` defines a vertex position, and one starting with `f` defines a triangle face from three vertices.

5.2 Procedural Generation

Procedular generation is just writing code to define vertices and polygons, instead of loading them or creating them manually.

We have to remember to:

1. Define points (vertices);
2. Define how to draw them (elements + primitives); and
3. Define normals.
4. (optionally) add texture coordinates and textures.

Random numbers are very useful for procedural generation, but remember that computers cannot really generate random numbers.

Instead, they generate pseudo-random numbers by using a formula and an initial seed.

If you provide the same seed to a random generator, you will always come up with the same "random" numbers.

5.2.1 Self-similar objects

If a portion of the object is looked at more closely, it looks exactly like the original object.

For example, a snowflake or a rocky mountain.

L-systems

An L-system is a simple way to represent self-similar objects: F[+F][-F+F]F[+F]

- The F means move one unit forward.
- The $+$ means turn clockwise.
- The $-$ means turn counter-clockwise.
- The $[$ means start a new branch.
- The $]$ means end a branch.

So L-systems work by iterations. Each iteration replaces an F with the complete L-system definition.

So the first iteration would look like this: F[+F][-F+F]F[+F] And the second iteration would look like this: F[+F][-F+F]F[+F][+F[+F][-F+F]F[+F]][-F[+F][-F+F]F[+F]+F[+F][-F+F]F[+F]]F[+F][-F+F]F[+F][+F[+F][-F+F]F[+F]]

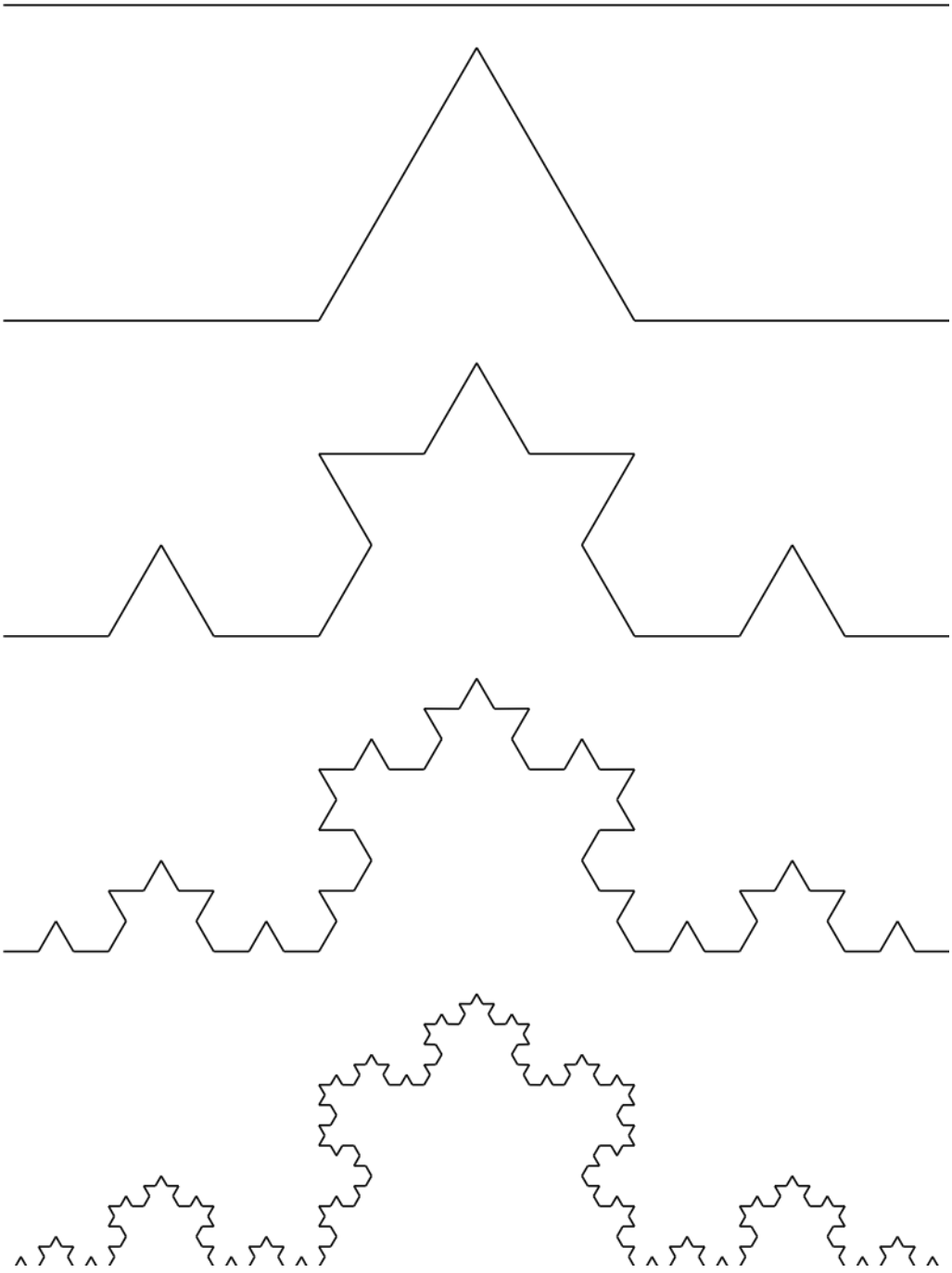
Fractals

A fractal entails generating geometrical objects by combining simpler gemetries, like lines, circles, triangles, etc...

Koch Curve

The Koch curve is just a set of four lines.

Each iteration, the mid-point of each line is displaced.



5.2.2 Skylines and landscapes

Heightfield or DEM

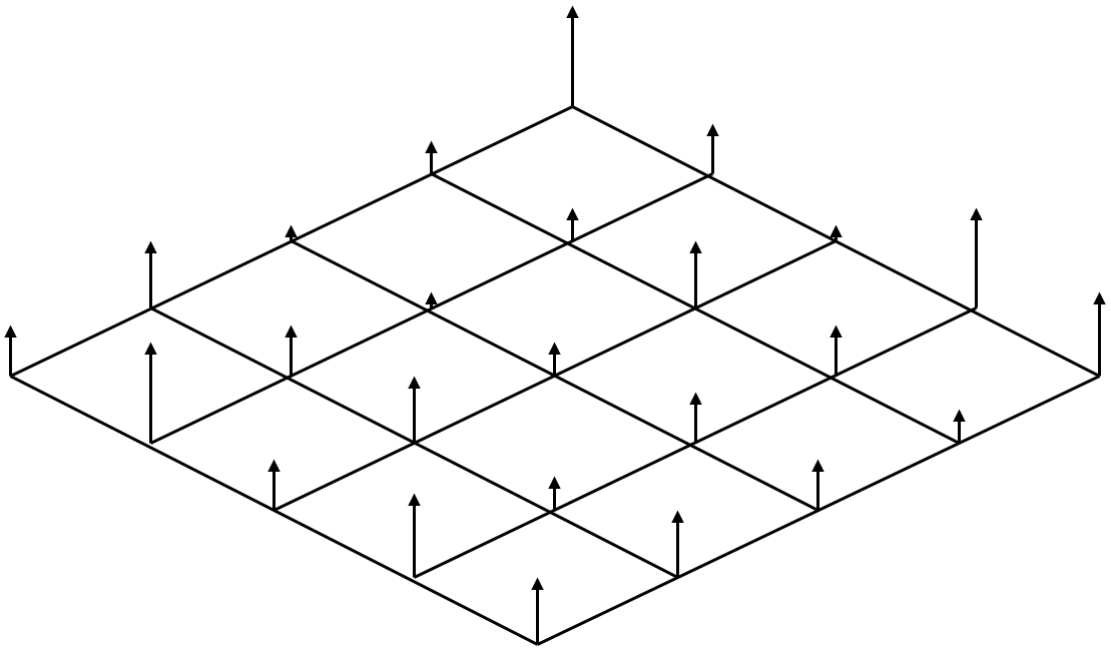
A heightfield is just an array of arrays of height (`Y`) values.

It can look something like this:

```
[
  [5.6, 3.4, 3.7, 3.5, 4.8],   # z == 0
  [3.5, 3.9, 4.9, 7.9, 7.5],   # z == 1
  [2.3, 3.0, 4.0, 3.7, 3.2],   # z == 2
  [3.5, 3.9, 4.9, 7.9, 7.5],   # z == 3
  [3.2, 3.9, 5.3, 4.7, 8.1],   # z == 4
]
```

For each of the elements of the array, `x` is increasing (from `0` to `4`).

Eventually, we end up with a terrain, that may look something like this:



Then we can draw the triangles. And calculate the normals.

In order to achieve a more random-looking terrain, there are a few functions we can use.

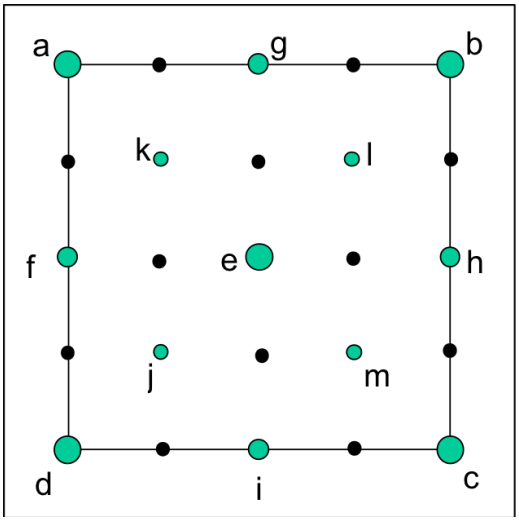
Random mid-point displacement

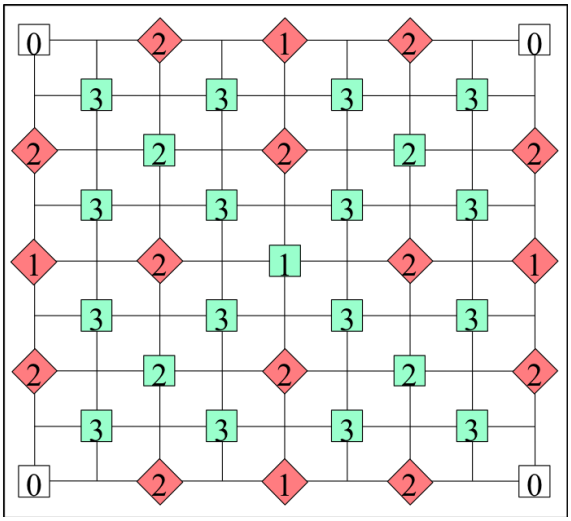
Set values for the first and last elements of the heightmap, and then the middle element will be randomised between the two beside it.

So on the first iteration, there would be 3 elements.

On the second iteration, there would be 5 elements, then 7, and so on.

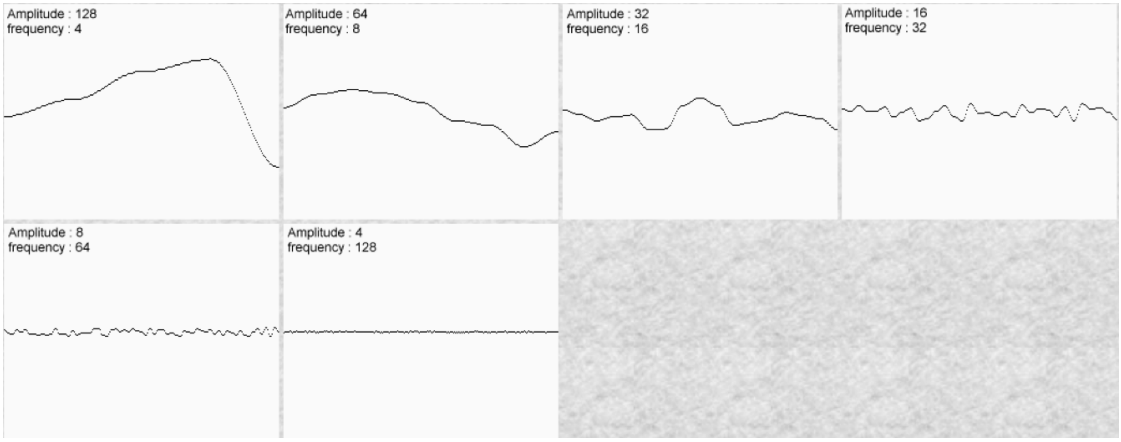
The order in which elements on a 3D height-map would be calculated would be as follows:





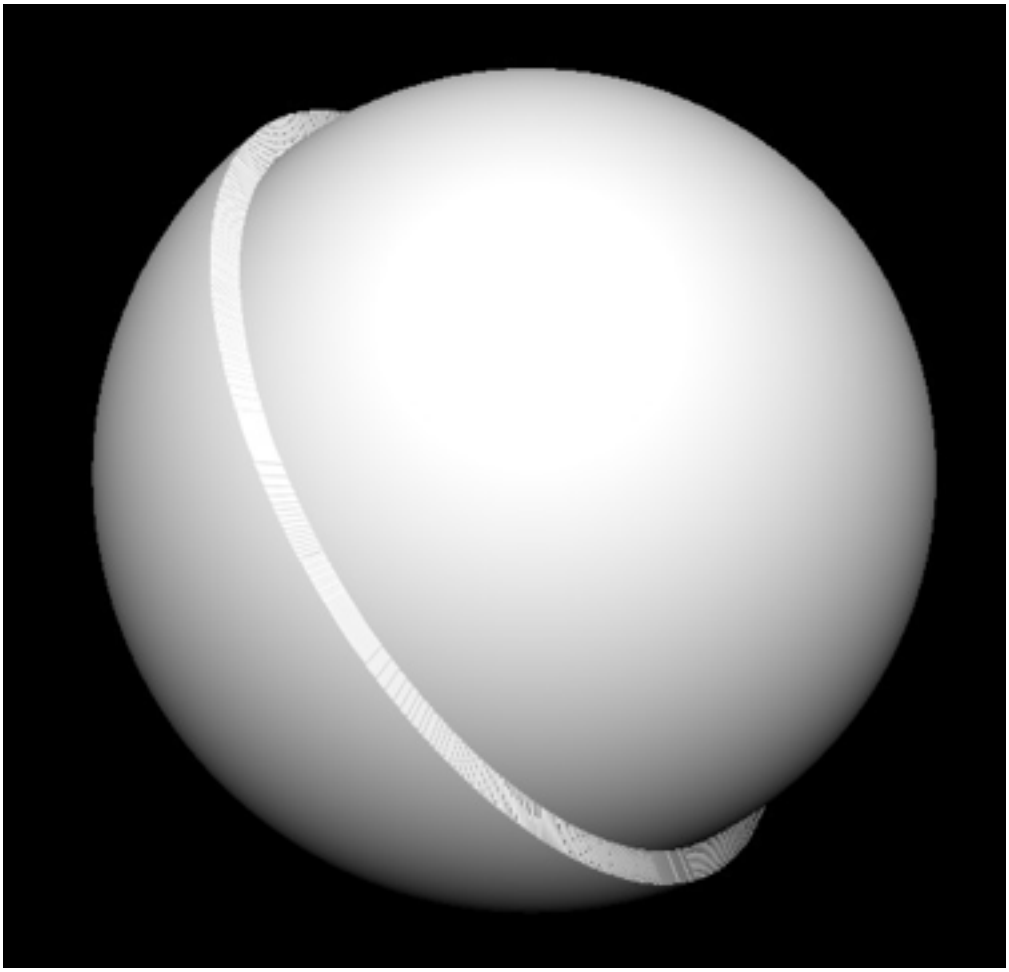
Noise functions

Perlin noise is generated by adding multiple noise functions together, at increasing frequency and decreasing amplitude:



A "better" noise function is **Simplex noise**.

Poisson faulting adds detail to a sphere by making it grow at random positions, like so:



Doing that many times (without making the growing too large) will result in added detail for e.g. asteroids or planets. However, Poisson faulting is slow at $O(N^3)$.

CHAPTER 6

Noise particles and normal mapping

Contents:

6.1 Particles

Many effects can be simulated by particles: smoke, fire, rain, snow, etc...

Each of the particles could have simple behaviour (like just moving down, or to the side), or they could have more complex movement, like gravity or collisions with other particles.

The best way to implement particle systems in OpenGL is by using instancing:

1. Send a particle object to the pipeline; and
2. Draw different instances of it with one command.

6.1.1 GL_POINTS for particles

It is efficient to use `GL_POINTS` to draw your particles, as opposed to other primitives like complex polygons, because it is more efficient as they have no geometry.

It is also easy to modify the final shape of the particle by discarding pixel fragments at the fragment shader stage.

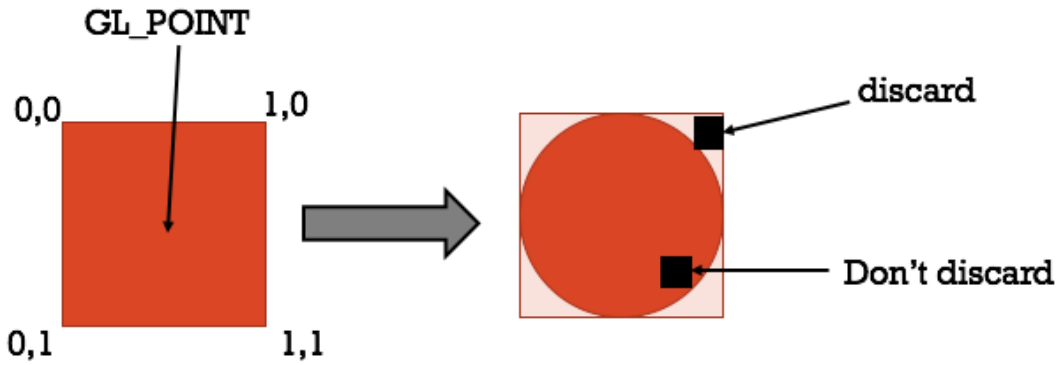
You can control the point size in the shader like so:

```
gl_PointSize = (1.0 - pos2.z / pos2.w) * size;
```

It is easy to discard pixel fragments in the fragment shader like so (example below to make a circle from a `GL_POINT`):

```
vec2 temp = gl_PointCoord - vec2(0.5);
float f = dot(temp, temp);
if (f>0.25) discard;
```

`gl_PointCoord` is a variable which contains **where within a point primitive the current fragment is located**, and only works for `GL_POINTS`.



6.2 Bump and Normal Maps

6.2.1 Bump map

A bump map is a heightmap (a file of gray-scale values, where the height is depicted by a darker gray) used to modify surface height by modifying the normal.

It's about the fine-grained detail of the object rather than the colour.

However it is not very used anymore.

6.2.2 Normal map

Has a similar effect to bump mapping, but the file uses RGB colours instead of a gray-scale. It is generally considered superior to bump-mapping.

It is a complex topic, but a tutorial is available here.

6.2.3 Displacement mapping

This is, in contrast to the two above, not a "fake" technique.

This technique is used to actually modify the geometry of the shape to which it is applied, and this can be done in the tessellation shaders.

It is used to add high-resolution mesh detail to a low-resolution mesh.

CHAPTER 7

Shadow Casting

Shadows consist of umbra (area completely cut off from light source) and penumbra (light source partially obscured).

7.1 Simple Method

Simple method for working out shadows:

1. Draw each object with normal lighting and colour; then
2. Draw each object again but projected into a plane, and in black/gray.

So the algorithm could look something like this:

1. Define the light source position and the normal to the plane you want to cast the shadow on
2. Create a mat4 shadow matrix
3. Calculate the dot product of the light ray and the object normal, to find out the projected plane.
4. Define your object's transformations (store this value or use a stack)
5. Transform the model matrix by the shadowMatrix (multiply)
6. Enable blending and a dark colour (e.g. ambient only)
7. Draw the object
8. Define the object's unshadowed model matrix (e.g. push stack)
9. Disable blending and enable the object's colour
10. Draw the object

7.2 Shadow Z-buffer method

This is a Z-buffer calculated using the light source as the viewpoint.

For any visible point, a transformation maps the point to the equivalent point in coordinates based on the light source.

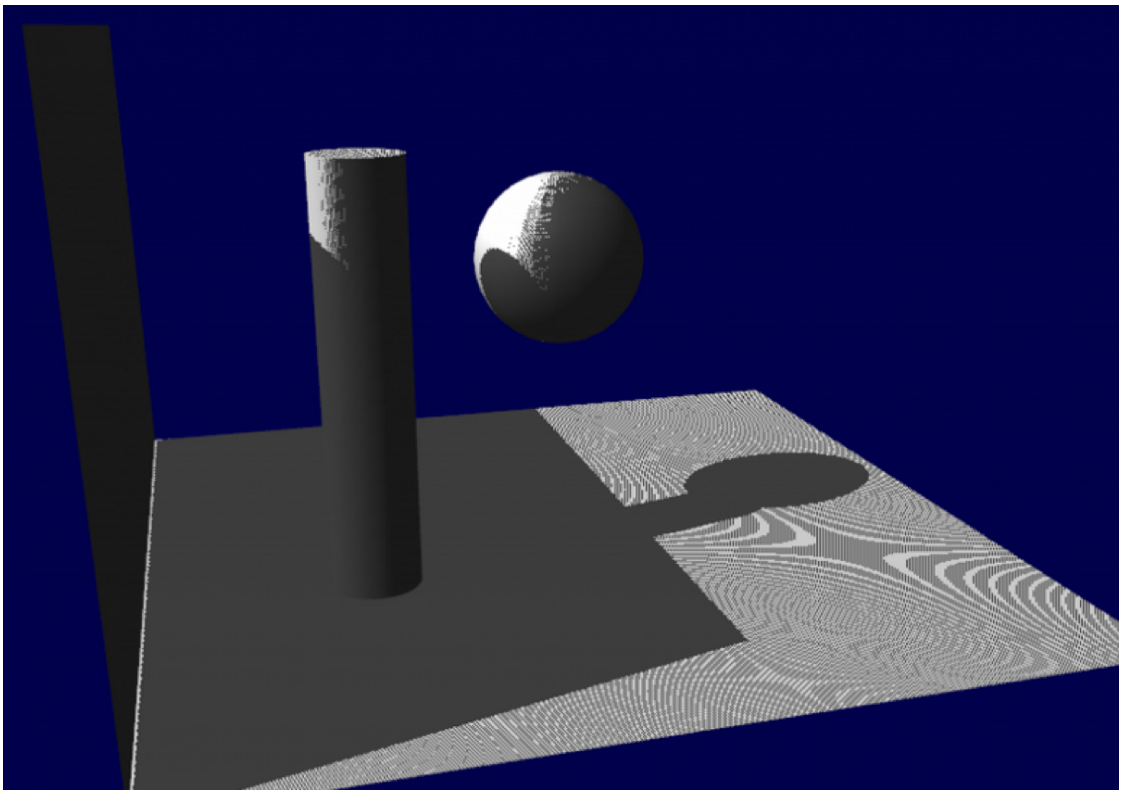
However, when the original objects are rendered, the normal Z-buffer is used. The Shadow Z-buffer is only used to render the shadows.

> The depth of the point is then compared to the depth stored in the appropriate pixel of the shadow Z-buffer; if the current point has greater depth, it is in shadow and should be modified accordingly, otherwise it is not in shadow.

In the algorithm, we would render the scene in two passes:

1. Render the depth view (what the light can 'see').
   - This includes generating a texture of 'depth values'.
   - But we don't actually have to calculate colour from the fragment shader, only depth values.
   - This can be done by enabling front-face culling.
2. Render the scene from the camera position, passing the depth texture to the fragment shader.
   - Remember to disable culling, or enable back-face culling.
   - If the fragment is in shadow (calculated by using depth buffer from previous pass), then include no diffuse or specular component.

There can be problems, like below:



That is called "shadow acne", and is caused by some sections self-shadowing. It can be solved by a polygon offset to push depth values away from the viewer, essentially making the depth test more conservative.

7.3 Fixed vs Dynamic Shadows

Fixed:

- No per-frame cost; (good)
- No need to tune; (good)
- No artefacts; (good)
- Area light source possible; (good)
- Lengthy pre-computation phase (worse for area light); (bad)
- No support for dynamic scenes; and (bad)
- Restricted to mesh resolution. (bad)

Dynamic:

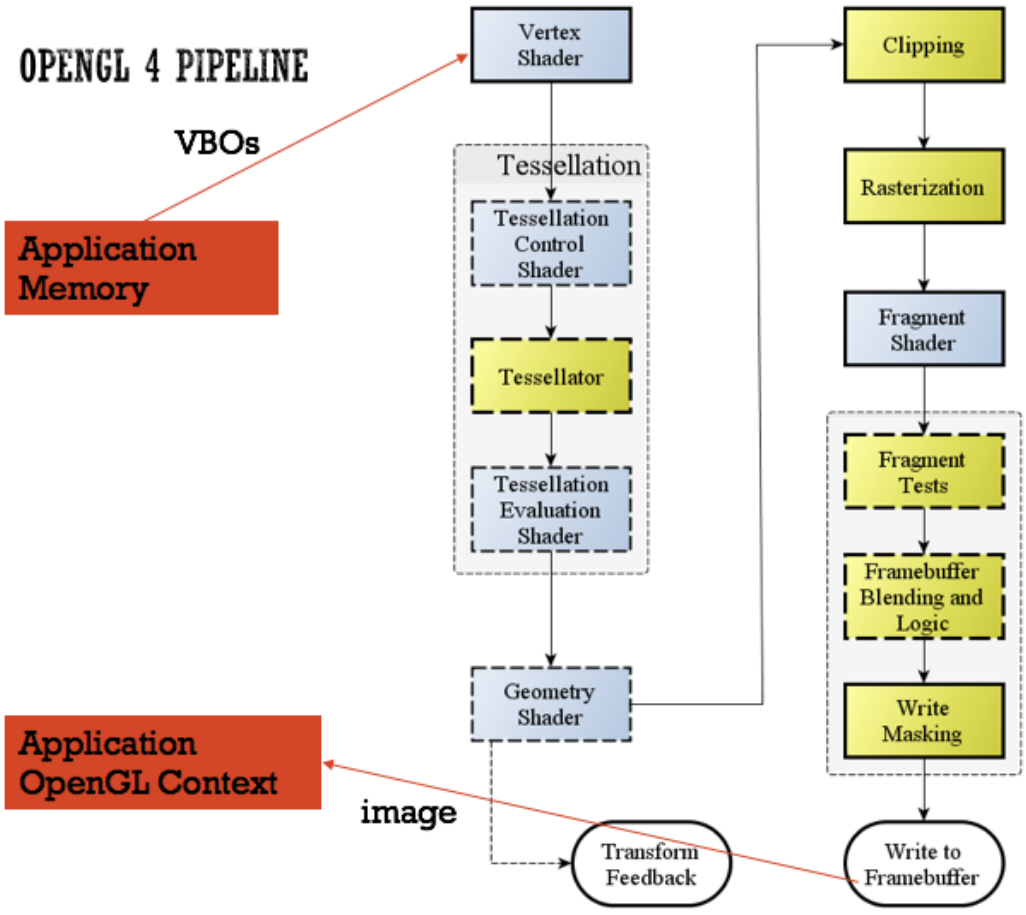
- Dynamic scene support;
- (Potentially) higher quality;
- No lengthy pre-computation;
- Slower rendering per-frame;
- Need to tune;
- Potential for artefacts; and
- Directional light source approximation (no penumbrae).

CHAPTER 8

Geometry and Tessellation Shaders

Remember to look at the *The OpenGL Pipeline*.

But, just in case, here is the final diagram of the OpenGL pipeline in version 4 and greater:



Most of the elements in the pipeline have already been described:

- *Vertex shader*;
- *Primitive assembly*;
- *Clipping*;
- *Rasterization*; and
- *Fragment shader*.

8.1 Tessellation Shaders

There are two shaders: `Control` and `Evaluation`.

They both operate on a new type of primitive: `GL_PATCHES`. A patch is just a list of vertices which preserves their order of specification.

They will give errors if patches are not passed to them.

8.1.1 Tessellation Control Shader

They do the following:

- Generate output patch vertices to be passed to Evaluation Shader; and
- Update per-vertex or per-patch attributes as requires.

Commonly this is a pass-through shader as if often not needed.

8.1.2 Tessellation Evaluation Shader

They do the following:

- Executed once for each tessellation coordinate that the primitive generator emits; and
- Sets the position of the vertices derived from the tessellation coordinates.

8.2 Geometry Shaders

Geometry shaders execute once for each primitive (e.g. point, line, or triangle), and they have access to all vertices in the primitive.

They can:

- Add or remove vertices from a mesh; - Cull triangles based on some visibility criteria; or - Change triangles to points or lines or viceversa; or - Shrink triangles; or - Emit vertices as components of particle animations.
- Generate geometry procedurally; and
- Add (limited) detail to existing meshes.

This is the last shader to run before the rasteriser.

Although they can add some detail to existing meshes, they are not ideal for general-purpose detail-adding algorithms, because they only have access to surrounding vertices, and not entire polygons.

Main functionality is provided by `EmitVertex()` and `EndPrimitive()`.

This is what a pass-through geometry shader would look like:

```
#version 400
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main()
{
   for (int i = 0; i<gl_in.length(); i++)
   {
      gl_Position = gl_in[i].gl_Position;
      EmitVertex();
   }

   EndPrimitive();
}
```

8.3 Compute Shaders

These are like a stripped-down version of CUDA, and can be used for non-graphics processing that uses the graphics card.

This is interesting because graphics cards tend to be faster and have extra dedicated memory.

But they can mix in with graphics programs (so some of the application computation can be done in the compute shader, while other shaders are also doing their stuff in the graphics pipeline).

CHAPTER 9

Indices and tables

- genindex
- modindex
- search