MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.709 Computer Graphics

Lecture 01 - Introduction

Daniel Playne
d.p.playne@massey.ac.nz

Course Information

- ***Lecture:*** Tuesday, MS Vision Lab, 09:00-11:00
- ***Assessments:*** Submit on Stream (nominal deadlines)
  - Assignment 1, 10%, 20th March
  - Assignment 2, 20%, 24th April
  - Assignment 3, 30%, 22nd May
  - Assignment 4, 40%, 20th June
- ***Stream:*** lecture notes, examples, additional reading

Course Information

- ***Textbook:*** there is no set (required) textbook.
- ***Suggested Books:***

A. Gerdelan, *Anton's OpenGL 4 Tutorials*, Amazon Digital Services.

T. Moller, E. Haines and N. Hoffman, *Real-Time Rendering*, CRC Press.

D. Wolff, *OpenGL 4.0 Shading Language Cookbook*, Packt Publishing.

G. Sellers, R. S. Wright, Jr. and N. Haemel, *OpenGL Superbible*, Addison-Wesley.

Course Content

What is this course about?

- Computer Graphics
- Real-time 3D Rendering Methods and Techniques
- Coordinate Spaces and Transforms
- 3D Object Representation
- Shading and Lighting Models and Effects
- OpenGL - Open Graphics Library

3D Graphics

Computer screens display images as a 2D grid of pixels, so how can they display a 3D image?

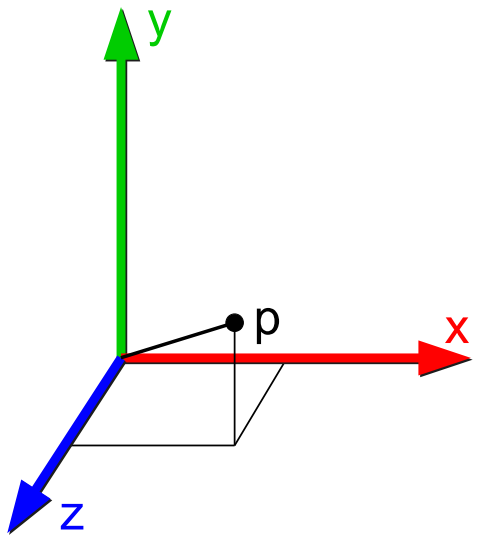
Rendering

Rendering is process of creating a ***two-dimensional image*** of a ***three-dimensional scene*** given a particular view in three-dimensions.

The view of the scene is defined by the three-dimensional position the scene is viewed from as well as a viewing direction and a ***viewing frustrum*** (field of view).

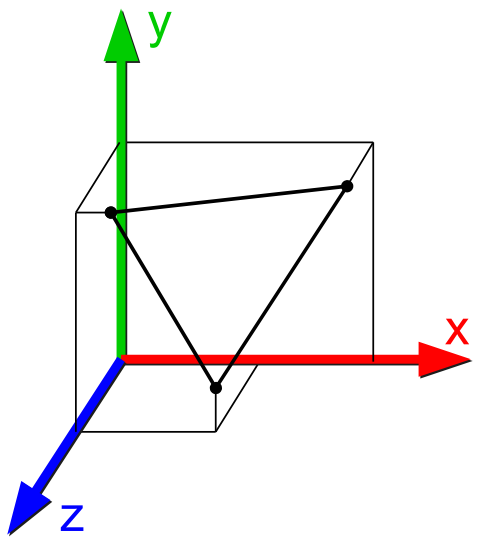
3D Coordinates

Represent points in 3D cartesian coordinates - a point p can be represented by a coordinate in each dimension (x_p, y_p, z_p).



Triangle

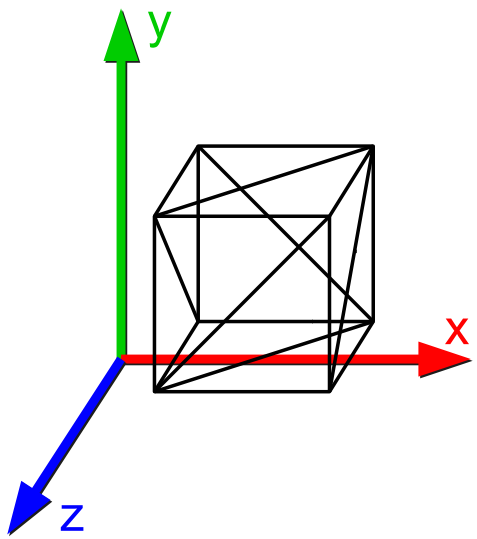
Points or ***vertexes*** are connected by ***edges*** which can be used to define ***faces***. In most of computer graphics, these faces are usually triangles.



Why triangles? All points on a triangle must lie on the same plane.

Polygon Modelling

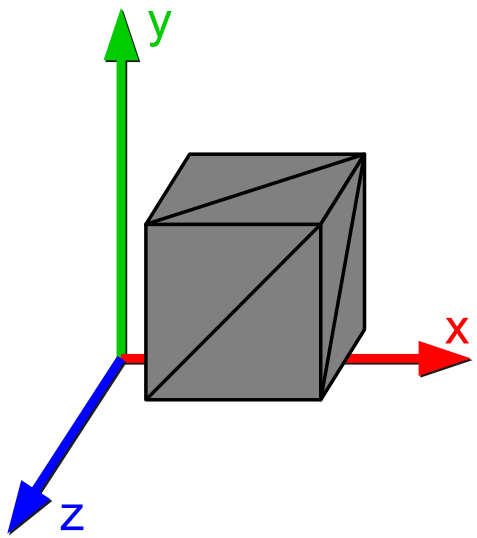
Faces can be used to define the surface of a 3D object, this is called ***polygonal modelling***. In this technique, any kind of polygons can be used (but again they're usually triangles).



Generally the faces should entirely enclose the volume of the object.

Rasterisation

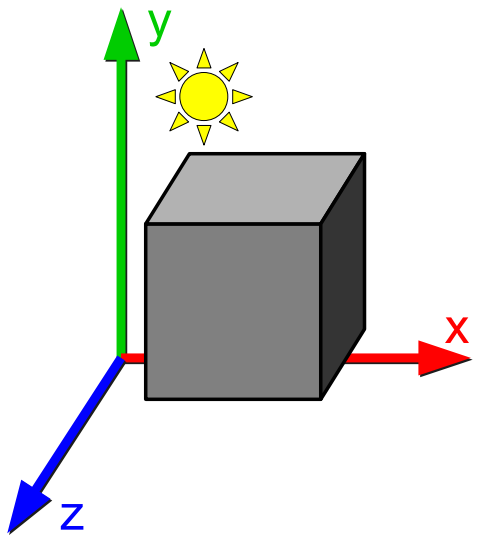
When a polygon is rendered, the faces are (usually) filled in and drawn in the correct depth order.



A polygon is ‘filled in’ by rasterisation, finding the pixels that it encompasses.

Shading

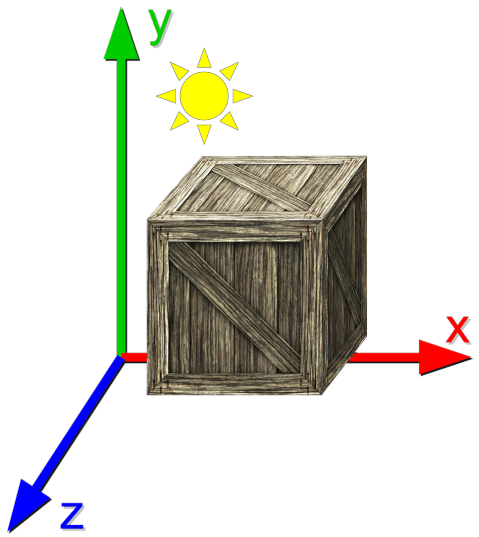
The shading of the polygon is determined by computing a lighting model. This is an important part of creating the illusion of depth.



There are a lot of different lighting models that can compute the lighting at different stages.

Texturing

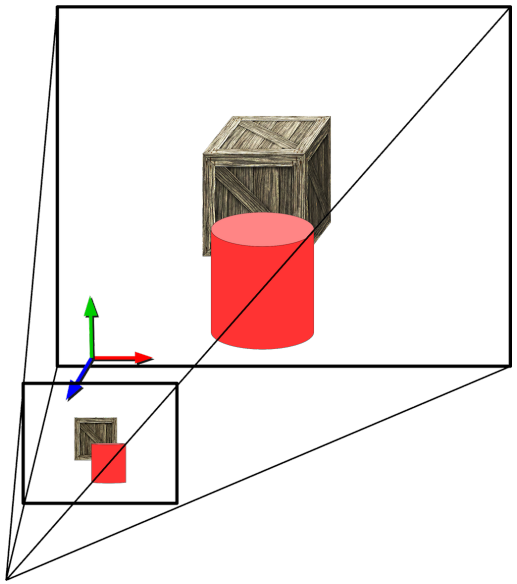
Textures (images) may also be applied to the surface of a polygon model to create a more realistic appearance.



Vertexes are mapped to a 2D texture by the use of UV coordinates.

Projection

The three-dimensional objects can be projected onto a two-dimensional plane that represents the computer screen. This step creates the illusion of perspective and vanishing points.



It should be noted that these steps are not performed in this order. In fact the perspective projection generally occurs before any lighting or texturing calculations are performed.

Rendering Pipeline

A ***three-dimensional scene*** consists of a description of all of the 3D objects, surfaces, materials, textures, lights, visual effects and more.

The ***graphics rendering pipeline*** (or just pipeline) has the job of taking the description of a scene and step-by-step turning it into the two-dimensional image that can be finally displayed on the screen.

Z-Buffering

In this course we will be working with real-time graphics and will be using the Z-buffering method for rendering.

This method essentially draws every triangle in the scene and keeps track of the depth of each pixel. The final colour of the pixel drawn on the screen comes from the triangle that was closest to the viewer.

Ray-Tracing

Ray-tracing is another popular rendering method that traces light from the eye to the surface of objects and lights and can render high-quality scenes with reflective and translucent surfaces and is widely used for CGI in movies and TV.

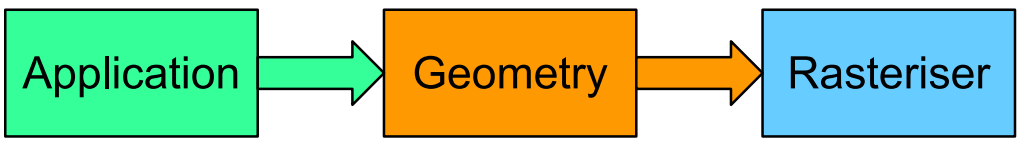
However, it is more computationally expensive (and variable) which makes it unsuitable for real-time graphics.

Ray-Tracing

NVIDIA GeForce RTX uses real-time ray-tracing to produce improved effects (especially important for reflections).

Rendering Pipeline

The rendering pipeline we will be using in this course can be roughly broken into three conceptual stages - ***application***, ***geometry*** and ***rasteriser***.



Each of these stages will usually consist of a number of different sub-stages that may sometimes be performed in parallel.

Application Stage

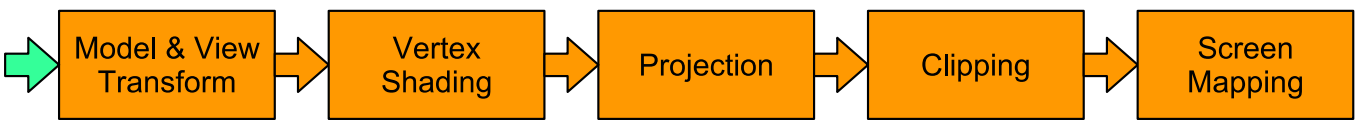
The ***application stage*** of the pipeline is entirely controlled by the programmer and is usually performed entirely on the CPU.

In this stage the programmer can configure the scene, operations such as:

- Updating the position of moving objects
- Detecting collisions between objects
- Updating animations
- Responding to user input
- and more..

Geometry Stage

The ***geometry stage*** of the pipeline is responsible for most of the operations that occur *per-polygon* or *per-vertex*. The main function of the geometry stage is to apply the various transformations to the vertexes in the scenes and convert them into screen coordinates.



The geometry stage can be conceptually split into the different sub-stages shown here.

Model & View Transform

Each 3D object or model usually has its own coordinate system called ***model space*** where the vertexes and polygons are defined.

To position an object appropriately within the scene it must be converted to ***world space*** which may involve moving, rotating and scaling the object according to the ***model transform*** associated with the object.

Model & View Transform

The object is also transformed by the ***view transform*** which represents the view of the scene - this transform converts ***world space*** into ***eye*** or ***camera space***. Coordinates in ***eye space*** represent the position of an object relative to the viewer.

The ***view*** can be thought of as a virtual camera that moves around the scene to observe it from different positions and angles.

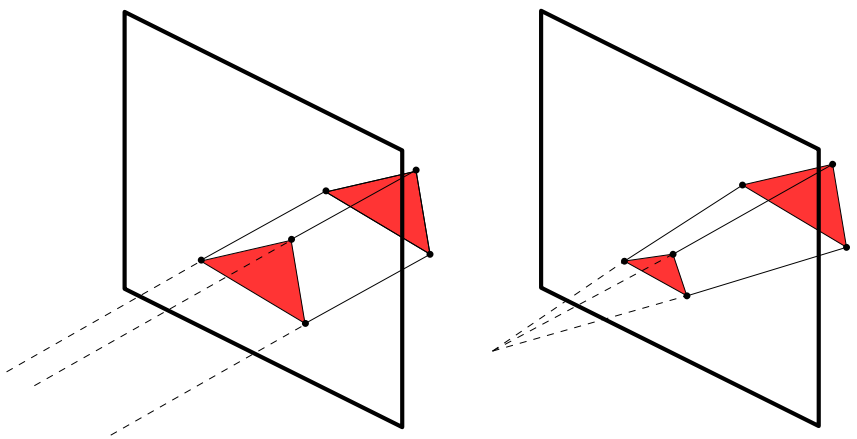
Vertex Shading

Producing a convincing rendering requires not only the position of the objects to be calculated but also their lighting, shading and texturing.

There are a variety of methods that can be used to perform this computation but most typically require (at least some) computation to be performed per-vertex in the geometry stage.

Projection

The projection transforms the ***view volume*** (the volume that the view can 'see') into a cube $(-1, -1, -1)$ to $(1, 1, 1)$.



The two most commonly used projection methods are ***orthographic*** (left) and ***perspective*** projection (right).

Clipping

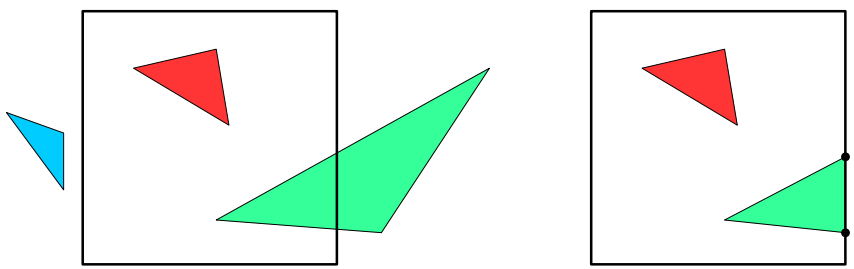
Only primitives that are actually visible inside the view volume need to be passed onto the rasteriser stage. If an object is entirely outside the view volume, it cannot be seen and rendering it would be a waste of time.

Primitives that lie entirely within the view volume are passed onto the next stage without modification. If a a primitive is only partially inside the view volume it will be clipped to fit.

Unlike most geometry stages, clipping is usually performed automatically and is not programmable.

Clipping

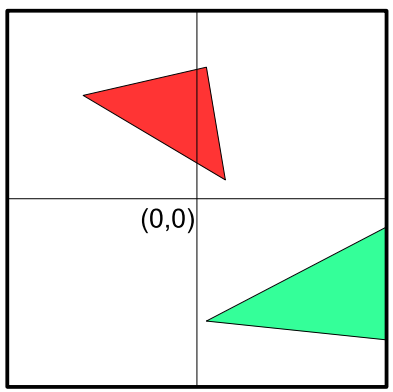
If a line has a vertex outside the view volume, the vertex will be replaced by a new vertex at the intersection between the line and the volume.

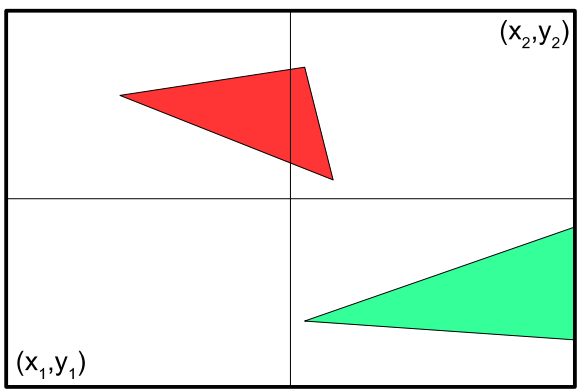


The advantage of performing view transformation and projection before clipping is that the clipping process is always performed on a simple cube.

Screen Mapping

The clipped primitives inside the view are transformed to ***screen coordinates***, this converts the x and y coordinates from $[-1, 1]$ to the coordinates of the screen $[x_1, x_2]$ and $[y_1, y_2]$.





Screen Coordinates

The is some confusion on how floating-point coordinates are converted to integers. OpenGL uses the scheme that the left edge of the leftmost pixel is represented by the value 0.0 while the centre is 0.5. This means that the conversion between a floating point coordinate f and the equivalent integer coordinate i are:

$$\begin{aligned} i &= \texttt{floor}(f) \\ f &= i + 0.5 \end{aligned}$$

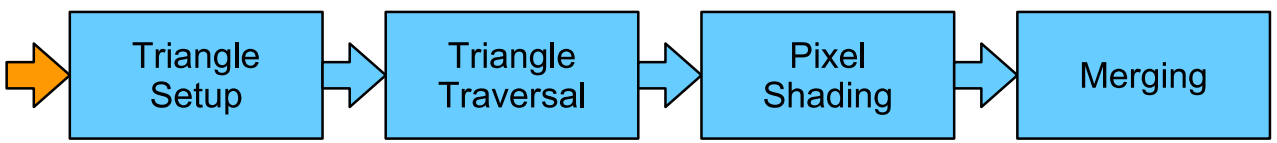
Screen Coordinates

There is also some confusion on whether the integer coordinate $(0, 0)$ represents the top-left or the bottom-left pixel. OpenGL uses the convention that the bottom-left pixel has the coordinate $(0, 0)$.

In DirectX the coordinate $(0, 0)$ represents the top-left pixel. Neither approach is inherently right or wrong but you generally follow the convention of the API you are using.

Rasteriser Stage

The rasteriser takes the transformed and projected primitives (and associated data) from the geometry stage and computes the colour for each pixel covered by the object.



The primitives are converted to a set of pixels with associated shading information through a process called ***rasterisation*** or ***scan conversion***.

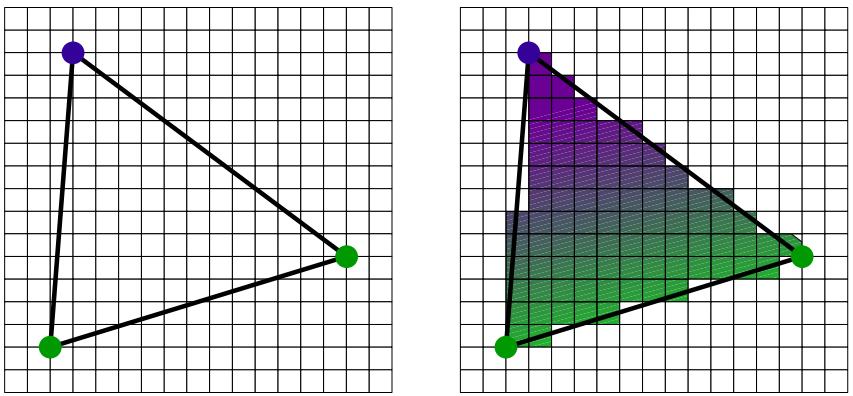
Triangle Setup & Traversal

In the ***triangle setup*** and ***traversal*** stage, the triangles have the necessary differentials computed and a ***fragment*** is generated for each pixel that has its centre (or sample) covered by the triangle. Each fragment will have the appropriate data interpolated from the vertexes that make up the triangle.

Like the clipping stage, this process is usually performed automatically and is not programmed by the user.

Pixel Shading

The per-pixel (or rather per-fragment) computations are performed in the ***pixel shading*** stage. This is one of the main programmable stages of the graphics pipeline where many of the lighting, shading, texturing calculations take place.



This stage takes the interpolated data from the previous step and computes a colour for the fragment or pixel.

Merging

As multiple primitives may overlap the same pixel, the colours produced by the different fragments must be combined together to form the final colour of the pixel.

This is performed by keeping a ***colour buffer*** which contains the current colour of the pixel and a ***z-buffer*** (also called the ***depth-buffer***) which stores the current depth of each pixel. Both of these buffers are the same size with a single entry for each pixel on the screen.

Merging

Each time a fragment is drawn, the z-value of the fragment is compared to the current value in the ***z-buffer***. If the fragment has a smaller z-value, then the fragment is in front of the current entry and will overwrite the ***colour-buffer*** and ***z-buffer***.

If the fragment has a larger z value then it is behind the current entry and the ***colour-buffer*** and ***z-buffer*** will be left unchanged. OpenGL normally maintains two sets of buffers, one that is currently being rendered to and one that is currently being displayed on the screen.

Merging

The ***z-buffer*** algorithm is simple and popular as it works with any primitives (provided a z-value can be computed) and allows them to be drawn in any order.

However, if there are semi-transparent objects in the scene then these must be drawn after the opaque objects and in order from back to front. Fast rendering methods for transparency (and translucency) are one of the major challenges for real-time computer graphics.

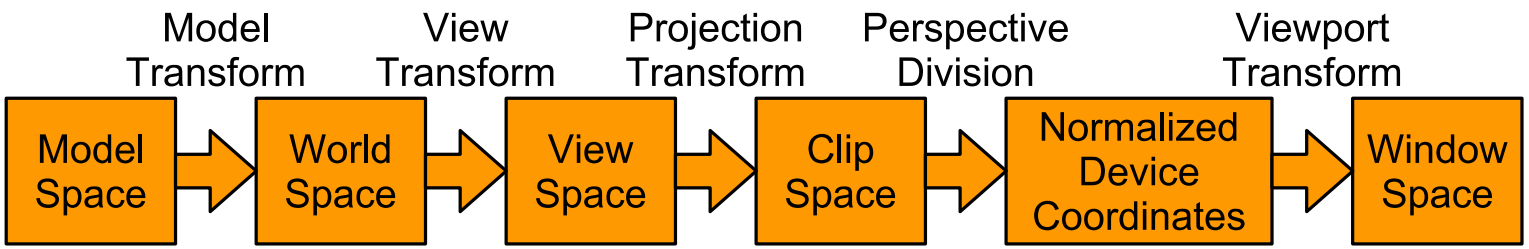
Merging

The ***merging*** stage may also make use of a ***stencil-buffer*** that can be used to control the rendering into both the ***colour***- and ***z-buffers***.

Fragments are only rendered to the output if they are within the area defined in the ***stencil-buffer***, essentially it defines a clipping region which can be a helpful tool to control rendering.

Coordinate Spaces

There are a number of different coordinate spaces used throughout the rendering pipeline. Transforms, comparisons and other calculations must be performed in the appropriate coordinate space if the desired effect is to be achieved.



These coordinate spaces will be covered in more detail as we progress through the course, but it is a good idea to try to get an overview of them.

Coordinate Spaces

Coordinate Space	Description
Model Space	Position relative to local origin.
World Space	Position relative to global origin.
View Space	Position relative to viewer.
Clip Space	Position after projection into non-linear homogenous coordinates.
Normalised Device Coordinates NDC	Vertex coordinates after clip coordinates are divided by w component.
Window Space	Position in pixels, relative to window origin.

OpenGL

In this course we will be using OpenGL or Open Graphics Library as our graphics API.

- OpenGL 1.0 released in 1992 (current version is 4.6)
- Originally developed by Silicon Graphics and now maintained by OpenGL Architecture Review Board (part of Kronos Group).
- Divided into ***Core*** and ***Compatibility*** profile since version 3.2 (we will be using ***Core*** profile).
- OpenGL Shading Language ***GLSL*** - C-style language for implementing shaders.
- Portable and not tied to any one company or OS.

Third-Party Libraries

In addition to OpenGL (and a C++ compiler of course) we will be using some additional third-party libraries:

- ***GLFW*** - Graphics Library Framework (http://www.glfw.org) for managing windows and handling input/output.
- ***GLEW*** - OpenGL Extension Wrangler Library (http://glew.sourceforge.net) for managing OpenGL extension loading.
- and more as we progress through the course.

Create a Window

```
1  // Initialise GLFW
2  if (!glfwInit()) {
3     // Error – Abort
4     return 1;
5  }
6
7  // Create Window
8  GLFWwindow *window = createWindow(600, 600, "Example 01", 3, 2);
9
10 // Check Window
11 if (window == NULL) {
12    // Print Error Message
13    std::cerr << "Error: create window or context failed." << std::endl;
14
15    // Return Error
16    return 1;
17 }
18
19 // Initialise GLEW
20 if(glewInit() != GLEW_OK) {
21    // Print Error Message
22    std::cerr << "Error: could not initialise GLEW." << std::endl;
23
24    // Return Error
25    return 1;
26 }
27 ...
```

Create a Window

```
1  // Create a GLFW Window
2  GLFWwindow* createWindow(int width, int height, const char *title,
3                           int major = 3, int minor = 2,
4                           GLFWmonitor *monitor = NULL,
5                           GLFWwindow *share = NULL) {
6    // Request an OpenGL context with specific features
7     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, major);
8     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, minor);
9
10    // If Version is 3 or higher
11    if (major >= 3) {
12       // Request Forward Compatibility
13       glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
14       // If version is 3.2 or higher
15       if (major > 3 || minor >= 2) {
16          // Request Core Profile
17          glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
18       }
19    }
20    ...
```

Create a Window

```
1     ...
2     // Create GLFW Window
3     GLFWwindow *window = glfwCreateWindow(width, height, title, monitor, share);
4
5     // Check Returned Window
6     if (window == NULL) {
7        return NULL;
8     }
9
10    // Set Context
11    glfwMakeContextCurrent(window);
12
13    // Set window callback functions
14    glfwSetFramebufferSizeCallback(window, onFramebufferSize);
15    glfwSetWindowCloseCallback(window, onWindowClose);
16
17    // Return GLFW window
18    return window;
19 }
```

Simple Render Loop

```
1     // Main Render loop
2     while (!glfwWindowShouldClose(window)) {
3        // Make the context of the given window current on the calling thread
4        glfwMakeContextCurrent(window);
5
6        // Set clear (background) colour to green
7        glClearColor(0.f, 0.5f, 0.f, 0.f);
8
9        // Clear Screen
10       glClear(GL_COLOR_BUFFER_BIT);
11
12       // Swap the back and front buffers
13       glfwSwapBuffers(window);
14
15       // Poll window events
16       glfwPollEvents();
17    }
18
19    // Stop receiving events for the window and free resources; this must be
20    // called from the main thread and should not be invoked from a callback
21    glfwDestroyWindow(window);
22
23    // Terminate GLFW
24    glfwTerminate();
```

Setting up your machine

To compile and run this example program (and all subsequent examples and assignments) you will need to make sure your machine is set up correctly.

The exact paths and flags you need will depend on how you configure your machine. To help get started, here is a suggested set up and makefiles for the program.

Mac OSX

- Compiler - clang/llvm compiler from XCode tools.
- OpenGL - OpenGL framework should already be installed.
- GLFW - installed with macports (installed in `/opt/local`)
- GLEW - unnecessary.

Makefile

```
1  HEADERS = -I/opt/local/include/
2  LIBS = -L/opt/local/lib/ -lglfw -framework OpenGL
3  CC = g++ -O3 -Wall -W
4
5  example01: example01.o
6     ${CC} example01.o -o example01 ${LIBS}
7
8  example01.o: example01.cpp
9     ${CC} -c example01.cpp -o example01.o ${HEADERS}
10
11 clean:
12    rm *.o
13    rm example01
```

Windows

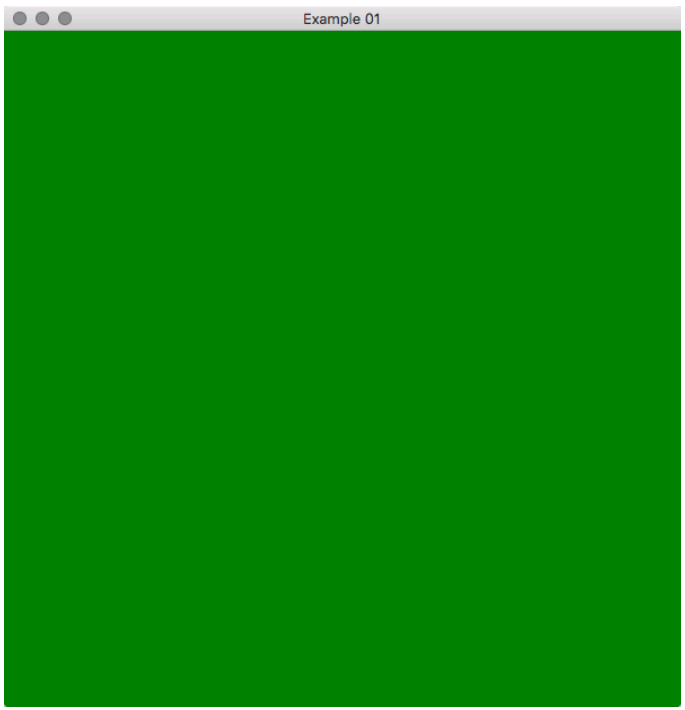
- Compiler - MinGW g++.
- OpenGL - OpenGL installed by graphics drivers.
- GLFW - pre-compiled binaries available (in `./GLFW`)
- GLEW - pre-compiled binaries available (in `./GLEW`)

Makefile

```
HEADERS =  -I./GLFW/include/ -I./GLEW/include/
LIBS = -L./GLFW/lib/ -L./GLEW/lib/ -lglew32 -lglfw3 -lopengl32 -lgdi32
CC = g++ -O3 -Wall -W

example01: example01.o
    ${CC} example01.o -o example01 ${LIBS}

example01.o: example01.cpp
    ${CC} -c example01.cpp -o example01.o ${HEADERS}

clean:
    del *.o
    del example01
```

Test Program

This program should create a single window entirely filled with a green background (the window appearance will vary slightly depending on the operating system).



This Week:

Tasks for this week:

- Set up the compiler and libraries on your machine.
- Download `example01` and read through the code.
- ***Compile and run the example***.