Lesson 0:

1 INTRODUCTION

Dear Student

Greetings to you and welcome to the Computer Graphics module. We hope that you find it interesting and exciting.

COS3712 is a final year undergraduate (i.e. third year) semester module. We are obliged to offer a course that is comparable in standard to what other universities in the world offer in CG (Computer Graphics) at this level. As the School of Computing, we opted for a CG programming course using WebGL, which is based on OpenGL a state-of-the-art and widely used public domain graphics API, (application programming interface).

COS3712 is not primarily a programming module. The theory of CG, including its mathematical foundations (linear algebra), is the primary focus. A certain amount of practical programming is included in COS3712 to illustrate how the theory is applied.

The prescribed book that we use covers some of the linear algebra applicable to CG in Chapter 4, as well as in two appendices at the end of the textbook. Additional notes on linear algebra are also available as an additional resource on myunisa.

2 PURPOSE AND OUTCOMES

2.1 Purpose

Students who successfully complete this module will be equipped with knowledge of the fundamental principles and techniques of modern Computer Graphics. They will be able to use these ideas, methods and tools to write and implement graphics applications of medium complexity.

This module presents a top-down, programming-oriented approach to computer graphics with an emphasis on applications programming. Areas covered include Application Programming Interfaces (API's), three dimensional graphics, interactive graphics, as well all aspects of the computer graphics pipeline

2.2 Outcomes

For this module, there are several outcomes that we hope you will be able to accomplish by the end of the course:

Specific outcome 1: Demonstrate an understanding of graphics systems, models, architectures and API's.

Specific outcome 2: Demonstrate an understanding of the basic principles involved in programming two-dimensional graphics applications.

Specific outcome 3: Demonstrate an understanding of how to develop user input and interaction in computer graphics applications.

Specific outcome 4: Demonstrate an understanding of how geometric objects and transformations can be used in 3-Dimensional Computer graphics.

Specific outcome 5: Demonstrate an understanding of 3-dimensional programming techniques and concepts, including viewing, lighting, shading and discrete techniques.

Specific outcome 6: Demonstrate an understanding of the four major tasks of the graphics pipeline including modelling, geometry processing, rasterization and fragment processing.

Lesson 1

Computer graphics is concerned with all aspects of producing pictures or images using a computer.

1.1 Applications of Computer Graphics

The development of computer graphics has been driven both by the needs of the user community and by advances in hardware and software.

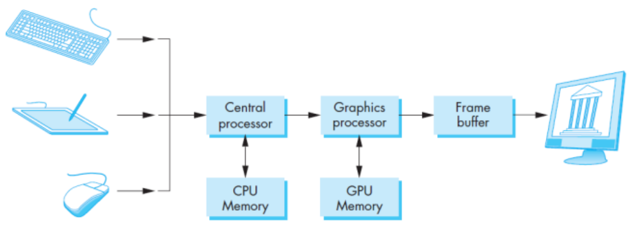
The application of computer graphics can be divided into four, possibly overlapping, areas:

1. Display of information: Some examples include:
    - Maps are used to display celestial and geographical information.
    - Statistical plots are generated to aid the viewer in determining information in a set of data.
    - Medical imaging technologies, such as CT, MRI, ultrasound, and PET.
2. Design: Professions such as engineering and architecture use computer-aided design (CAD) tools to create interactive technical drawings.
3. Simulation and animation: Graphical flight simulators have proved both to increase safety and to reduce training expenses. Computer graphics are also used for animation in the television, motion-pictures, and advertising industries. Virtual reality (VR) technology allows the viewer to act as part of a computer-generated scene.
4. User interfaces: Our interaction with computers has become dominated by a visual paradigm that includes windows, icons, menus, and a pointing device, such as a mouse.

1.2 A Graphics System

The six major components of a basic graphics system are:

1. Input devices;
2. Central Processing Unit (CPU);
3. Graphics Processing Unit (GPU);
4. Memory;
5. Frame buffer;
6. Output devices.



In a graphics program, we can obtain the measure (what the device returns) of an input device in three distinct modes:

1. Request mode: The measure of the device is not returned to the program until the device is triggered. A trigger of a device is a physical input on the device with which the user can signal the computer.
2. Sample-mode: As soon as the function call of a function that expects device input is encountered, the measure is returned.
3. Event-mode: Each time that a device is triggered, an event is generated and the device measure, including the identifier for that device, is placed in an event queue. Periodically the queue is polled, and for each (if any) event in the queue, the program can look at the event's type and then decide what to do.

Both request- and sample-mode input APIs require that the user identify which device is to provide the input, and, are thus not sufficient for modern computing environments.

Raster based graphics system: The image we see on the output device is an array – the raster – of pixels produced by the graphics system. Each pixel corresponds to a unique location in the image. Collectively, the pixels are stored in a part of memory called the frame buffer. Resolution refers to the number of pixels in the frame buffer, and it determines the detail that you can see in the image. The depth, or precision of the frame buffer, defined as the number of bits used per pixel, determines properties such as how many colours can be represented on a given system. In full-colour (also known as true-colour or RGB-colour) systems, there is at least 24 bit per pixel.

The frame buffer is actually a collection of buffers: Colour buffers hold the coloured pixels that are displayed; depth buffers hold information needed for creating images from three-dimensional data; and other special purpose buffers, such as accumulation buffers, etc..

Rasterization (or scan conversion) is the process of converting geometric entities to pixel colours and locations in the frame buffer.

Non-interlaced display: The pixels are displayed row by row, or scan line by scan line at the refresh rate. I.e. all the rows are refreshed.

Interlaced display: Odd rows and even rows are refreshed alternatively.

1.3 Images: Physical and Synthetic

Computer-generated images are synthetic or artificial, in the sense that the objects being imaged do not exist physically.

Two basic entities must be part of any image-formation process, be it mathematical or physical:

- Object: It exists in space independent of any image-formation process and of any viewer. We define a synthetic object by specifying the positions in space, called vertices, of various geometric primitives (points, lines, and polygons) that, when put together, approximate the object. CAD systems make it easy for a user to build synthetic objects.
- Viewer: It is the viewer that forms the image of our objects. Viewers placed at different positions, will see different ***images*** of the same ***object***.

Projection (image formation): The process by which the specification of an object is combined with the specification of a viewer to produce a two-dimensional image.

Visible light has wavelengths in the range of 350 to 780 nanometres (nm). Distinct frequencies within this range are visible as distinct colours. Wavelengths in the middle of the range, around 520 nm, are seen as green; those near 450 nm are seen as blue; and those near 650 nm are seen as red.

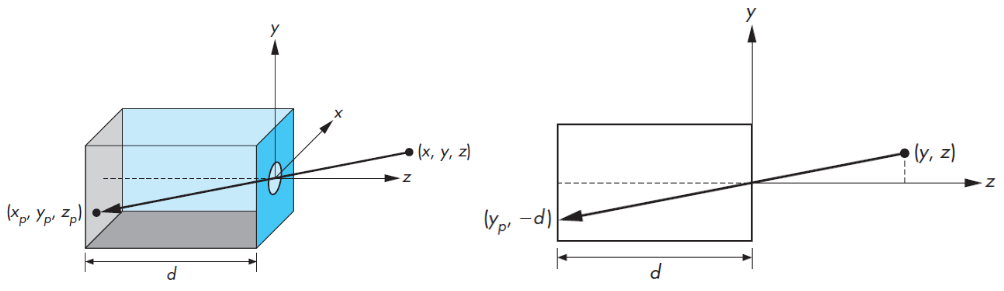
Light sources can emit light either as a set of discrete frequencies or continuously. A particular light source is characterized by the intensity of light that it emits at each frequency and by that light's directionality. An ideal point source emits energy from a single location at one or more frequencies equally in all directions.

Light from the source strikes various surfaces of an object; the details of the interaction between light and the surfaces of the object determine how much light is reflected, and hence the colour(s) of the object as perceived by the viewer.

1.4 Imaging Systems

A pinhole camera is a box with a small hole in the centre of one side of the box. The hole must be small enough to ensure that only a single ray of light, emanating from a point, can enter it. The film is placed inside the box, at a distance d from the pinhole. If we orient the camera along the z-axis, with the pinhole at the origin of the coordinate system, the projection of the point (x, y, z) is

$$(x_p, y_p, z_p) = \left(-\frac{xd}{z}, -\frac{yd}{z}, -d\right)$$



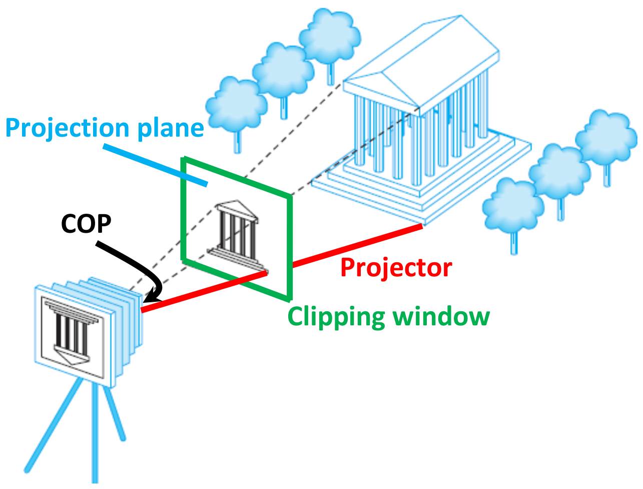
Infinite depth of field: Every point within the field of view is in focus.

1.5 The Synthetic Camera Model

Synthetic-camera model: A paradigm in which we look at the creation of a computer-generated image as being similar to forming an image using an optical system, such as a camera.

To create artificial images, we need to identify a few basic principles:

- The specification of the objects is independent of the specification of the viewer.
- We can compute the image of an object using simple geometric calculations. We find the image of a point on an object on the virtual image plane, called the projection plane, by drawing a line, called a projector, from the point to the centre of the lens, called the centre of projection (COP).The image of the point is located where the projector passes through the projection plane.
- The size of the image is limited. Objects outside the field of view should not appear in the resulting image. We place a clipping rectangle, or clipping window, in the projection plane.


Projection plane
COP
Projector
Clipping window

1.6 The Programmer's Interface

The interface between an application program and a graphics system can be specified through a set of functions that resides in a graphics library. These specifications are called the application programming interface (API). The application programmer sees only the API and is thus shielded from the details of both the hardware and the software implementation of the graphics library.

If we are to follow the synthetic-camera model, we need functions in the API to specify the following:

- Objects: The geometry of an object is usually defined by sets of vertices.
- A viewer: We can define a viewer or camera by specifying a number of parameters, including: position, orientation, focal length, and the size of the projection plane.
- Light sources: Light sources are defined by their location, strength, colour, and directionality.
- Material properties: Material properties are characteristics, or attributes, of the objects, and such properties are specified through a series of function calls at the time that each object is defined.

Wire-frame image: Only the edges of polygons (outline) are rendered using line segments.

The modelling-rendering paradigm: The modelling of the scene is separated from the production of the image, or the rendering of the scene. Thus, we might implement the modeller and the renderer with different software and hardware. This paradigm has become popular as a method for generating computer games and images over the Internet. Models, including the geometric objects, lights, cameras, and material properties, are placed in a data structure called a scene graph that is passed to a renderer or game engine.

1.7 Graphics Architectures

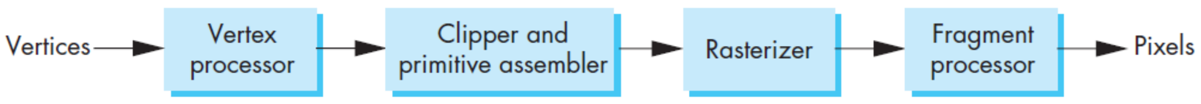
Early graphics architectures: Early graphics systems used general-purpose computers that could process only a single instruction at a time. Its display included the necessary circuitry to generate a line segment connecting two points. The job of the host computer was to run the application program that computes and sends endpoint data of the line segments in the image to the display at a rate high enough to avoid flicker on the display.

Display processors: The earliest attempts to build special-purpose graphics systems were concerned primarily with relieving the general-purpose computer from the task of refreshing the display continuously. The instructions to generate the image could be assembled once in the host and sent to the display processor where they were stored in the display processor's local memory. Thus, the host is freed for other tasks while the display processor continuously refreshes the display.

Pipeline architecture: A pipeline comprises a series of interconnected components, each optimized to perform a specific, or set, of operations on data moving through the pipeline. A pipeline can provide significant performance increase when the same sequence of concurrent operations must be performed on many, or large, data sets. That is exactly what we do in computer graphics, where large sets of vertices and pixels must be processed. Latency is the time it takes a single data item to pass through the pipeline. Throughput is the rate at which data flows through the pipeline.

The graphics pipeline: We start with a (possibly enormous) set of vertices which defines the geometry of the scene. We must process all these vertices in a similar manner to form an image in the frame buffer. There are four major steps in the imaging process:

1. Vertex processing: Each vertex is processed independently. The two major functions of this block are to carry out coordinate transformations and to compute a colour for each vertex. Per-vertex lighting calculations can be performed in this box.
2. Clipping and primitive assembly: Sets of vertices are assembled into primitives, such as line segments and polygons, before clipping can take place. In the synthetic camera model, a clipping volume represents the field of view of an optical system. The projections of objects in this volume appear in the image; those that are outside do not (clipped out); and those that straddle the edges of the clipping volume are partly visible. Clipping must be done on a primitive-by-primitive basis rather than on a vertex-by-vertex basis. The output of this stage is a set of primitives whose projections should appear in the image.
3. Rasterization: The primitives that emerge from the clipper are still represented in terms of their vertices and must be converted to pixels in the frame buffer. The output of the rasterizer is a set of fragments for each primitive. A fragment can be thought of as a potential pixel that carries with it information, including its colour, location, and depth.
4. Fragment processing: It takes the fragments generated by the rasterizer and updates the pixels in the frame buffer. Hidden-surface removal, texture mapping, bump mapping, and alpha blending can be applied here. Per-fragment lighting calculations can also be performed in this box.



1.8 Programmable Pipelines

For many years, although the application program could set many parameters, the basic operations available within the pipeline were fixed (fixed-function pipeline). Recently, both the vertex processor and the fragment processor are programmable by the application program. The main advantage of this is that many of the techniques that formerly could not be done in real time, because they were not part of the fixed-function pipeline, can now be done in real time. Vertex programs can alter the location or colour of each vertex as it flows through the pipeline: allowing for a variety of light-material models or creating new projections. Fragment programs allow us to uses textures in new ways (bump-mapping) and to implement other parts of the pipeline, such as lighting, on a per-fragment basis. These vertex- and fragment-programs, are commonly known as shaders, or shader programs.

1.9 Performance Characteristics

The overall performance of a graphics system is characterized by how fast we can move geometric entities through the pipeline and by how many pixels per second we can alter in the frame buffer. Consequently, the fastest graphics workstations are characterized by geometric pipelines at the front end and parallel bit processors at the back end. Physically based techniques, such as ray tracing and radiosity, can create photorealistic images with great fidelity, but usually not in real time.

Lesson 2: Graphics Programming

2.1 The Sierpinski Gasket

Immediate mode graphics: As vertices are generated by the application, they are sent directly to the graphics processor for rendering on the display. One consequence of immediate mode is that there is no memory of the geometric data. Thus, if we want to redisplay the scene, we would have to go through the entire creation and display process again (and every time a redisplay is required).

Retained mode graphics: We compute all the geometric data first and store it in some data structure. We then display the scene by sending all the stored data to the graphics processor at once. This approach avoids the overhead of sending small amounts of data to the graphics processor for each vertex we generate, but at the cost of having to store all the data. Because the data are stored, we can redisplay the scene, by resending the stored data without having to regenerating it.

Current GPUs allow us to store the generated data directly on the GPU, thus avoiding the bottleneck caused by transferring the data from the CPU to the GPU each time we wish to redisplay the scene.

2.2 Programming Two-Dimensional Applications

Two-dimensional systems are regarded as a special case of three-dimensional systems. Mathematically, we view the two-dimensional plane, or a simple two-dimensional curved surface, as a subspace of a three-dimensional space. We can represent the two-dimensional point $p = (x, y)$ as $p = (x, y, 0)$ in the three-dimensional world. In OpenGL, vertices specified as two- or three-dimensional entities are internally represented in the same manner.

In OpenGL terms:

- A vertex is a position in space; we use two-, three- and four-dimensional spaces in computer graphics. We use vertices to specify the atomic geometric primitives that are recognized by our graphics system.
- A point is the simplest geometric primitive, and is usually specified by a single vertex.

Clip coordinate system: Can be visualized as a cube centred at the origin whose diagonal goes from (-1, -1, -1) to (1, 1, 1). Objects outside this cube will be eliminated, or clipped, and cannot appear on the display. The vertex shader uses transformations to convert geometric data specified in some coordinate system to a representation in clip coordinates and outputs this information to the rasterizer.

2.3 The OpenGL Application Programming Interface

A graphics system performs multiple tasks to produce output and handle user input. An API for interfacing with this system can contain hundreds of individual functions. These functions can be divided into seven major groups:

1. Primitive functions: Define the low-level objects or atomic entities that our system can display. OpenGL supports only points, line segments, and triangles.
2. Attribute functions: Govern the way that a primitive appears on the display.

3. Viewing functions: Allow us to specify various views. OpenGL does not provide any viewing functions, but relies on the use of transformations in the shaders to provide the desired view.
4. Transformation functions: Allow us to carry out transformations of objects, such as rotation, translation, and scaling. In OpenGL, we carry out transformations by forming transformation matrices in our applications, and then applying then either in the application or in the shaders.
5. Input functions: Deals with input devices.
6. Control functions: Enable us to communicate with the window system, to initialize our programs, and to deal with any errors that take place during the execution of our programs.
7. Query functions: Allow us to obtain information about the operating environment, camera parameters, values in the frame buffer, etc.

We can think of the entire graphics system as a state machine. Applications provide input that change the state of the machine or cause the machine to produce a visible output. From the perspective of the API, graphics functions are of two types: those that specify primitives that flow through a pipeline inside the state machine and those that either change the state inside the machine or return state information. One important consequence of the state machine view is that most parameters are persistent; their values remain unchanged until we explicitly change them.

OpenGL functions are in a single library called GL. Shaders are written in the OpenGL Shading Language (GLSL), which has a separate specification from OpenGL, although the functions to interface the shaders with the application are part of the OpenGL API. To interface with the window system and to get input from external devices into our programs, we need to use some other library (GLX for X Window System, wgl for Windows, agl for Macintosh, GLUT (OpenGL Utility Toolkit) is a simple cross-platform library). The OpenGL Extension Wrangler (GLEW) library is used with cross-platform libraries, such a GLUT, to removes operating system dependencies. OpenGL makes heavy use of defined constants to increase code readability and avoid the use of magic numbers. Functions that transfer data to the shaders have the following notation:

```
glSomeFunction*();
```

where the `*` can be interpreted as either *`nt`* or *`nt`*`v`, where *`n`* signifies the number of dimensions (1, 2, 3, 4, or Matrix); *`t`* denotes the data type, such as integer (`i`), float (`f`), or double (`d`); and `v`, if present, indicates that the variables are specified through a pointer to an array, rather than through an argument list.

The units used to specify vertex positions in the application program are referred to as vertex coordinates, object coordinates, or world coordinates, and can be arbitrarily chosen by the programmer to suite the application. Units on the display device are called window coordinates, screen coordinates, physical-device coordinates or just device coordinates. At some point, the values in vertex coordinates must be mapped to window coordinates, but this is automatically done by the graphics system as part of the rendering process. The user needs to specify only a few parameters. This allows for device-independent graphics; freeing application programmers from worrying about the details of input and output devices.

2.4 Primitives an Attributes

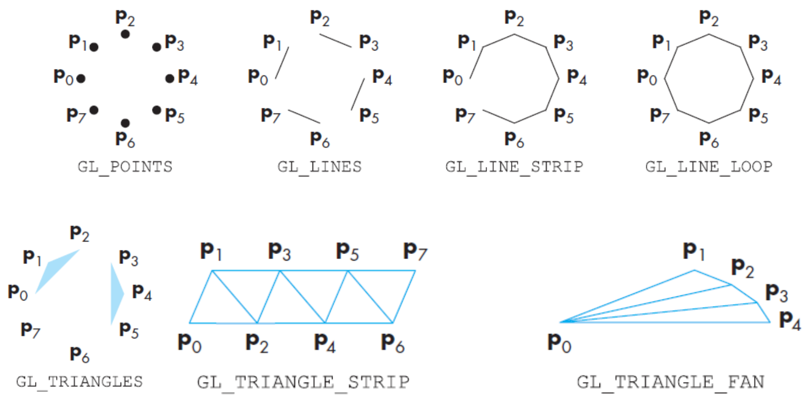
To ensure that a filled polygon is render correctly, it must have a well-defined interior. A polygon has a well-defined interior if it satisfies the following three properties:

- Simple: In two dimensions, as long as no two edges of a polygon cross each other, we have a simple polygon.
- Convex: An object is convex if all points on the line segment between any two points inside the object, or on its boundary, are inside the object (i.e. the line segment never intersects the edges of the object). Convex objects include triangles, tetrahedral, rectangles, parallelepipeds, circles, and spheres.
- Flat (planar). All the vertices that specify the polygon lie in the same plane.

As long as the three vertices of a triangle are not collinear, its interior is well defined and the triangle is simple, flat, and convex. Consequently, triangles are easy to render, and for these reasons triangles are the only fillable geometric entity that OpenGL recognizes.

We can separate primitives into two classes: geometric primitives and image, or raster, primitives. The basic OpenGL geometric primitives are specified by sets of vertices. All OpenGL geometric primitives are variants of points, line segments, and triangular polygons.

- Points (`GL_POINTS`): A point can be displayed as a single pixel or a small group of pixels. Use `glPointSize()` to set the current point size (in pixels).
- Lines: Use `glLineWidth()` the set the current line width (in pixels).
    - Line segments (`GL_LINES`): Successive pairs of vertices are interpreted as the endpoints of individual line segments.
    - Line strip or polyline (`GL_LINE_STRIP`): Successive vertices are connected.
    - Line loop (`GL_LINE_LOOP`): Successive vertices are connected, and a line segment is drawn from the final vertex to the first, thus creating a closed path.
- Polygons (triangles): Use `glPolygonMode()` to tell the renderer to generate only the edges or just points for the vertices, instead of fill (the default).
    - Triangles (`GL_TRIANGLES`): Each successive group of three vertices specifies a new triangle.
    - Triangle strip (`GL_TRIANGLE_STRIP`): Each additional vertex is combined with the previous two vertices to define a new triangle.
    - Triangle fan (`GL_TRIANGLE_FAN`): It is based on one fixed point. The next two points determine the first triangle, and subsequent triangles are formed from one new point, the previous point, and the first (fixed) point.



Triangulation is the process of approximating a general geometric object by subdividing it into a set of triangles. Every set of vertices can be triangulated. Triangulation is a special case of the more general problem of tessellation, which divides a polygon into a polygonal mesh, not all of which need be triangles.

Text in computer graphics is problematic. There are two forms of text:

- Stroke text: stroke text is constructed as are other geometric objects. We use vertices to specify line segments or curves that outline each character. The advantage of stroke text is that it can be defined to have all the detail of any other object and it can be manipulated by standard transformations and viewed like any other graphical primitive.
- Raster text: Characters are defined as rectangles of bits called bit blocks. Each block defines a single character by the pattern of 0 and1 bits in the block. A raster character can be placed in the frame buffer rapidly by a bit-block-transfer (bitblt) operation. Increasing the size of raster text characters cause it to appear blocky.

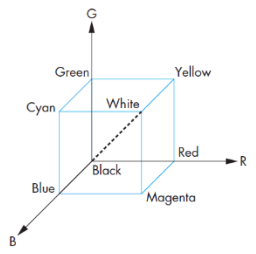
OpenGL does not have a text primitive.

Attributes are properties that describe how an object should be rendered. Available attributes depend on the type of object. For example, line segments can have colour, thickness, and pattern (solid, dashed, or dotted).

2.5 Colour

Additive colour model: The three primary colours (Red, Green, Blue) add together to give the perceived colour. (CRT monitors and projectors are examples of additive colour systems).With additive colour, primaries add light to an initially black display, yielding the desired colour.

Subtractive colour model: Here we start with a white surface, such as a sheet of paper. Coloured pigments remove colour components from light that is striking the surface. If we assume that white light hits the surface, a particular point will appear red if all components of the incoming light are absorbed by the surface except for wavelengths in the red part of the spectrum, which is reflected. In subtractive systems, the primaries are usually the complementary colours: cyan, magenta, and yellow (CMY). Industrial printers are examples of subtractive colour systems.



Colour cube: We can view a colour as a point in a colour solid. We draw the solid using a coordinate system corresponding to the three primaries. The distance along a coordinate axis represents the amount of the corresponding primary in the colour. We can represent any colour that we can produce with this set of primaries as a point in the cube.

In a RGB system, each pixel might consist of 24 bits (3 bytes): 1 byte for each of red, green, and blue. The specification of RGB colours is based on the colour cube. Thus, specify colour components as numbers between 0.0 and 1.0, where 1.0 denotes the maximum (or saturated value) of the corresponding primary and 0.0 denotes a zero value of that primary. RGBA is an extension of the RGB model, where the fourth colour (A, or alpha) is treated by OpenGL as either an opacity or transparency value. Transparency and opacity are complements of each other: an opaque object lets no light through, while a transparent object passes all light. Opacity values range from 0.0 (fully transparent) to 1.0 (fully opaque). Alpha blending is disabled by default.

Indexed colour: Early graphics systems had frame buffers that were limited in depth: for example, each pixel was only 8 bits deep. Instead of subdividing a pixel's bits into groups, and treat them as RGB values (which will result in a very restricted set of colours), with Indexed colours the limited-depth pixel is interpreted as an integer value which index into a colour-lookup table. The user program can fill the entries (rows) of the table with the desired colours. A Problem with indexed colours is that when we work with dynamic images that must be shaded, usually we need more colours than are provided by colour-index mode. Historically, colour-index mode was important because it required less memory for the frame buffer; however, the cost and density of memory is no longer an issue.

2.6 Viewing

A fundamental concept that emerges from the synthetic-camera model is that the specification of the objects in our scene is completely independent of our specification of the camera.

The simplest and OpenGL's default view is the orthographic projection. All projectors are parallel, and the centre of projection is replaced by a direction of projection. Furthermore, all projectors are perpendicular (orthogonal) to the projection plane. The orthographic projection takes a point (x, y, z) and projects it into the point $(x, y, 0)$.

2.7 Control Functions

A window, or display window, is an operating-system managed rectangular area of the screen in which we can display our images. A window has a height and width. Because the window displays the contents of the frame buffer, positions in the window are measured in window or screen coordinates, where the units are pixels. Note that references to positions in a window are usually relative to the top-left corner (which is regarded as position (0, 0)).

The aspect ratio of a rectangle is the ratio of the rectangle's width to its height. If the aspect ratio of the viewing (clipping) rectangle, specified by camera parameters, is not the same as the aspect ratio of the window, objects appear distorted on the screen.

A viewport is a rectangular area of the display window in which our images are rendered. By default, it is the entire window, but it can be set to any smaller size in pixels via the function

```
	void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)
```

where (`x, y`) is the lower-left corner of the viewport (measured relative to the lower-left corner of the window) and `w` and `h` give the width and height, respectively. For a given window, we can adjust the height and width of the viewport to match the aspect ratio of the clipping rectangle, thus preventing any object distortion in the image.

Events are changes that are detected by the operating system and include such actions as a user pressing a key on the keyboard, the user clicking a mouse button or moving the mouse. When an event occurs, it is placed in an event-queue. The event queue can be examined by an application program or by the operating system. We can associate callback functions with specific types of events. Event processing gives us interactive control in our programs. With GLUT, we can execute the function `glutMainLoop()` to begin an event-processing loop. All our programs must have at least a display callback function which is invoked when the application program or the operating system determines that the graphics in a window need to be redrawn.

2.8 The Gasket Program

A vertex-array object (VAO) allows us to bundle data associated with a vertex array. Use of multiple vertex-array objects will make it easy to switch among different vertex arrays. You can have at most one current (bound) VAO at a time.

A buffer object allows us to store data directly on the GPU.

Include `glFlush();` at the end of the display callback function to ensure that all the data are rendered as soon as possible.

Every application, no matter how simple, must provide both a vertex- and a fragment-shader (there are no default shaders). Each shader is a complete C-line program with `main()` as its entry point. The vertex shader is executed for each vertex that is passed through the pipeline. In general, a vertex shader will transform the representation of a vertex location from whatever coordinate system in which it is specified to a representation in clip coordinates for the rasterizer. The fragment shader is executed for each fragment generated by the rasterizer. At a minimum, each execution of the fragment shader must output a colour for the fragment.

2.9 Polygons and Recursion

Recursive subdivision is a powerful technique that can be used to subdivide a polygon into a set of smaller polygons.

2.10 The Three-Dimensional Gasket

By default, primitives are drawn in the order in which they appear in the array buffer. As a primitive is being rendered, all its fragments are unconditionally placed into the frame buffer covering any previously drawn objects, even if those objects are actually located closer to the viewer. Algorithms for ordering objects so that they are drawn correctly are called visible-surface algorithms or hidden-surface-removal algorithms, depending on how we look at the problem. The hidden-surface-removal algorithm supported by OpenGL is called the z-buffer algorithm. The z-buffer is one of the buffers that make up the frame buffer. You use `glEnable(GL_DEPT_TEST)` to enable the z-buffer algorithm.

2.11 Adding Interaction

Instead of calling the display callback function directly, rather invoke the `glutPostRedisplay()` function which sets an internal flag indicating that the display needs to be redrawn. At the end of each event-loop iteration, if the flag is set, the display callback is invoked and the flag is unset. This method prevents the display from being redrawn multiple times in a single pass through the event loop.
Two types of events are associated with the pointing device (mouse):

- Mouse events occur when one of the mouse buttons is either depressed (mouse down event) or released (mouse up event).
- Move events are generated when the mouse is moved with one of the buttons depressed. If the mouse is moved without a button being held down, this event is called a passive move event.

Reshape events occur when the user resizes the window, usually by dragging a corner of the window to a new location. This is an example of a window event. Unlike most other callbacks, there is a default reshape callback that simply changes the viewport to the new window size.

Keyboard events can be generated when the mouse is in the window and one of the keys is depressed or released.

The idle callback is invoked when there are no other events. A typical use of the idle callback is to continue to generate graphical primitives through a display function while nothing else is happening. Another is to produce an animated display.

An application program operates asynchronously from the automatic display of the contents of the frame buffer, and can cause changes to the frame buffer at any time. Hence, a redisplay of the frame buffer can occur while its contents are still being altered by the application and the user will see only a partially drawn display. This distortion can be severe, especially if objects are constantly moving around in the scene. A common solution is double-buffering. The hardware has two frame buffers: one, called the front buffer, is the one that is displayed, the other, called the back buffer, is then available for constructing what we would like to display next. Once the drawing is complete, we swap the front and back buffers. We then clear the new back buffer and can start drawing into it. With double-buffering we use `glutSwapBuffers()` instead of `glFlush()` at the end of the display callback function.

GLUT provides pop-up menus that we can use with the mouse to create sophisticated interactive applications. GLUT also supports hierarchical (cascading) menu entries. Using menus involves taking a few simple steps:

- Define callback function(s) that specify the actions corresponding to each entry in the menu.
- Create a menu; register its callback; and add menu entries and/or submenus. This step must be repeated for each submenu, and once for the top-level menu.
- Attach the top-level menu to a particular mouse button.

Lesson 3: Geometric Objects and Transformations

3.1 Scalars, Points, and Vectors

See tutorial letter 103 for a summary on linear algebra.

3.2 Three-Dimensional Primitives

A full range of three-dimensional objects cannot be supported on existing graphics systems, except by approximate methods. Three features characterize three-dimensional objects that fit well with existing graphics hardware and software:

1. The objects are described by their surfaces and can be thought of as being hollow. Since a surface is a two- rather than a three-dimensional entity, we need only two-dimensional primitives to model three-dimensional objects.
2. The objects can be specified through a set of vertices in three dimensions.
3. The objects either are composed of, or can be approximated by simple, flat, and convex polygons. Most graphics systems are optimized for the processing of points, line segments, and triangles. Even if our modelling system provides curved objects, we assume that a triangle mesh approximation is used for implementation.

3.3 Coordinate Systems and Frames

In three dimensions, the representation of a point and a vector is the same:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Homogeneous coordinates avoid this difficulty by using a four-dimensional representation for both points and vectors in three dimensions: The fourth component of a vector is set to 0:

$$\begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix},$$

whereas the fourth component of a point is set to 1:

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

Some advantages of using homogeneous coordinates include:

- All affine (line-preserving) transformations can be represented as matrix multiplications.
- We can carry out operations on points and vectors using their homogeneous-coordinate representations and ordinary matrix algebra.
- The uniform representation of all affine transformations makes carrying out successive transformations (concatenation) far easier than in three-dimensional space.
- Although we have to work in four dimensions to solve three-dimensional problems when we use homogeneous-coordinate representations, less arithmetic work is involved.
- Modern hardware implements homogeneous-coordinate operations directly, using parallelism to achieve high-speed calculations.

3.4 Frames in OpenGL

In versions of OpenGL with a fixed-function pipeline and immediate-mode rendering, six frames were specified in the pipeline. With programmable shaders, we have a great deal of flexibility to add additional frames or avoid some traditional frames. The following is the usual order in which the frames occur in the pipeline:

1. Object (or model) coordinates: In most applications, we tend to specify or use an object with a convenient size, orientation, and location in its own frame called the model or object frame.
2. World (or application) coordinates: A scene may comprise many objects. The application program generally applies a sequence of transformations to each object to size, orient, and position it within a frame that is appropriate for the particular application. This application frame is called the world frame, and the values are in world coordinates.
3. Eye (or camera) coordinates: Virtually all graphics systems use a frame whose origin is the centre of the camera's "lens" and whose axes are aligned with the sides of the camera. This frame is called the camera frame or eye frame.
4. Clip coordinates: Once objects are in eye coordinates, OpenGL must check whether they lie within the view volume. If an object does not, it is clipped from the scene prior to rasterization. OpenGL can carry out this process most efficiently if it first carries out a projection transformation that brings all potentially visible objects into a cube centred at the origin in clip coordinates.
5. Normalized device coordinates: At this stage, vertices are still represented in homogeneous coordinates. The division by the w (fourth) component, called perspective division, yields three-dimensional representations in normalized device coordinates.
6. Window coordinates: The final transformation takes a position in normalized device coordinates and, taking into account the viewport, creates a three-dimensional representation in window coordinates. Window coordinates are measured in units of pixels on the display but retain depth information.
7. Screen coordinates: If we remove the depth coordinate, we are working with two-dimensional screen coordinates.

Because there is an affine transformation that corresponds to each change of frame, there are 4×4 matrices that represent the transformation from model coordinates to world coordinates and from world coordinates to eye coordinates. These transformations usually are concatenated together into the model-view transformation, which is specified by the model-view matrix.

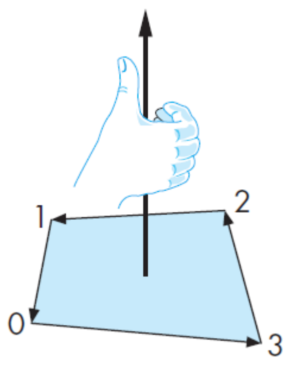
3.5 Matrix and Vector Classes

The authors of the book provide two header files, namely mat.h and vec.h, that contain the definitions of mat2, mat3, mat4, vec2, vec3, and vec4 types. These classes mirror those available in the GLSL language. Matrix classes are for 2×2, 3×3, and 4×4 matrices whereas the vector types are for 2-, 3- and 4-element arrays. Standard matrix and vector operations are also implemented.

3.6 Modelling a Coloured Cube

We describe geometric objects through a set of vertex specifications. The data specifying the location of the vertices (geometry) can be stored as a simple list or array - the vertex list.

We have to be careful about the order in which we specify our vertices when we are defining a three-dimensional polygon. The order is important because each polygon has two sides. Our graphics systems can display either or both of them. We call a face outward facing if the vertices are traversed in a counter-clockwise order when the face is viewed from the outside. This method is also known as the right-hand rule because if you orient the fingers of your right hand in the direction the vertices are traversed, the thumb points outward. By specifying front and back carefully, we will be able to eliminate (or cull) faces that are not visible or to use different attributes to display front and back faces.



3.7 Affine Transformations

A transformation is a function that takes a point (or vector) and maps it into another point (or vector). When we work with homogeneous coordinates, any affine transformation can be represented by a 4×4 matrix that can be applied to a point or vector by pre-multiplication:

$$\mathbf{q} = \mathbf{Tp}.$$

All affine transformations preserve lines. Common affine transformations include rotation, translation, scaling, shearing, or any combination of these.

3.8 Translation, Rotation, and Scaling

Translation is an operation that displaces points by a fixed distance in a given direction.

Rotation is an operation that rotates points by a fixed angle about a point or line. In a right-handed system, when we draw the x- and y-axes in the standard way, the positive z-axis comes out of the "page". If we look down the positive z-axis towards the origin, the positive direction of rotation (positive angle of rotation) is counter-clockwise. This definition applies to both the x- and y-axes as well.

Rotation and translation are known as rigid-body transformations. No combination of rotations and translations can alter the shape or volume of an object; they can alter only the object's location and orientation.

Scaling is an affine non-rigid-body transformation by which we can make an object bigger or smaller. Scaling has a fixed point: a point that is unaffected by the transformation. A negative scaling factor gives us reflection about the fixed point, in the specific scaling direction.

- Uniform scaling: The scaling factor in all directions is identical. The shape of the scaled object is preserved.
- Non-uniform scaling: The scaling factor of each direction need not be identical. The shape of the scaled object is distorted.

3.9 Transformations in Homogenous Coordinates

Translation matrix:

$$T(\alpha_x, \alpha_y, \alpha_z) = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Inverse translation matrix:

$$T^{-1}(\alpha_x, \alpha_y, \alpha_z) = T(-\alpha_x, -\alpha_y, -\alpha_z).$$

Scaling matrix (fixed point at origin):

$$S(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Inverse scaling matrix (fixed point at origin):

$$S^{-1}(\beta_x, \beta_y, \beta_z) = S\left(\frac{1}{\beta_x}, \frac{1}{\beta_y}, \frac{1}{\beta_z}\right).$$

Rotation about the x-axis by an angle θ:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the y-axis by an angle θ:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the z-axis by an angle θ:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Suppose that we let R denote any of our three rotation matrices, the inverse rotation matrix is:

$$R^{-1}(\theta) = R(-\theta) = R^T(\theta)$$

We can construct any desired rotation matrix, with a fixed point at the origin as a product of individual rotations about the three axes:

$$R = R_z R_y R_x.$$

Using the fact that the transpose of a product is the product of the transposes in the reverse order, we see that for any rotation matrix,

$$R^{-1} = R^T.$$

3.10 Concatenation of Transformations

We can create transformation matrices for more complex affine transformations by multiplying together, or concatenating, sequences of the basic transformation matrices. This strategy is preferable to attempting to define an arbitrary transformation directly.

To rotate an object about an arbitrary fixed point, say $\mathbf{p}_f$, we first translate the object such that $\mathbf{p}_f$ coincides with the origin: $T(-\mathbf{p}_f)$; we then apply the rotation: $R(\theta)$; and finally move the object back such that $\mathbf{p}_f$ is again at its original position: $T(\mathbf{p}_f)$. Thus, concatenating the matrices together, we obtain the single matrix:

$$M = T(\mathbf{p}_f)R(\theta)T(-\mathbf{p}_f).$$

Notice the “reverse” order in which the matrices are multiplied. Matrix multiplication, in general, is not a commutative operation, thus, the order in which we apply transformations is critical!

Objects are usually defined in their own frames, with the origin at the centre of mass and the sides aligned with the model frame axes. To place an instance of such an object in a scene, we apply an affine transformation – the instance transformation – to the prototype to obtain the desired size, orientation, and location. The instance transformation is constructed in the following order: first, we scale the object to the desired size; then we orient it with a rotation matrix; finally, we translate it to the desired location. Hence, the instance transformation is of the form

$$M = TRS.$$

To rotate an object by an angle θ about an arbitrary axis, we carry out at most two rotations to align the axis of rotation with, say the z-axis; then rotate by θ about the z-axis; and finally we undo the two rotations that did the aligning. Thus, our final rotation matrix will be of the form:

$$R = R_x^{-1}R_y^{-1}R_z(\theta)R_yR_x.$$

3.11 Transformation Matrices in OpenGL

In a modern implementation of OpenGL, the application programmer not only can choose which frames to use (model-, world-, and eye-frame), but also where to carry out the transformations between frames (application or vertex shader). Although very few state variables are predefined in OpenGL, once we specify various attributes and matrices, they effectively define the state of the system and hence how vertices are processed. The two transformations we will use most often are:

1. Model-view transformation: The model-view matrix is an affine transformation matrix that brings representations of geometric objects from application or model frames to the camera frame.
2. Projection transformation. The projection matrix is usually not affine and is responsible for carrying out both the desired projection and also changes the representation to clip coordinates.

In my opinion, the remainder of this section (except for the example) applies to older versions of OpenGL.

3.12 Spinning of the Cube

In a given application, a variable may change in a variety of ways. When we send vertex attributes to a shader, these attributes can be different for each vertex in a primitive. We may also want parameters that will remain the same for all vertices during a draw. Such variables are called uniform qualified variables.

Lesson 4: Viewing

4.1 Classical and Computer viewing

Projectors meet at the centre of projection (COP). The COP corresponds to the centre of the lens in the camera or in the eye, and in a computer-graphics system, it is the origin of the camera frame for perspective views. The projection surface is a plane, and the projectors are straight lines. If we move the COP to infinity, the projectors become parallel and the COP can be replaced by a direction of projection (DOP). Views with a finite COP are called perspective views; views with a COP at infinity (i.e. a DOP) are called parallel view. The class of projections produced by parallel and perspective systems is known as planar geometric projections because the projection surface is a plane and the projectors are lines. Both perspective and parallel projections preserve lines; they do not, in general, preserve angles.

Classical views:

- Parallel projections:
  - Orthographic projection: In all orthographic (or orthogonal) views, the projectors are perpendicular to the projection plane. In a multi-view orthographic projection, we make multiple projections, in each case with the projection plane parallel to one of the principal faces of the object. The importance of this type of view is that it preserves both distances and angles. It is well suited for working drawings.
  - Axonometric projections: In axonometric views, the projectors are still orthogonal to the projection plane, but the projection plane can have any orientation with respect to the object.
    - Isometric view: The projection plane is placed symmetrically with respect to the three principal faces that meet at a corner of a rectangular object.
    - Diametric view: The projection place is placed symmetrically with respect to two of the principal faces of a rectangular object.
    - Trimetric view: The projection plane can have any orientation with respect to the object (the general case).

    Although parallel lines are preserved in the image, angles are not. Axonometric views are used extensively in architecture and mechanical design.
  - Oblique projections: It is the most general parallel view. We obtain an oblique projection by allowing the projectors to make an arbitrary angle with the projection plane. Angles in planes parallel to the projection plane are preserved.
- Perspective projections: All perspective views are characterized by diminution of size: the farther an object is moved from the viewer, the smaller its image becomes. We cannot make measurements from a perspective view. Hence, perspective views are used by applications where it is important to achieve natural-looking images. The classical perspective views are usually known as one- , two-, and three-point perspective. The one, two, and three prefixes refer to the number of vanishing points (points at which lines of perspective meet).

Three-point

Two-point

One-point

4.2 Viewing with a Computer

In computer graphics, the desired view can be achieved by applying a sequence of transformations on each object in the scene. Every transformation is equivalent to a change of frames. Of the frames that are used in OpenGL, three are important in the viewing process: the object frame, the camera frame, and the clip coordinate frame.

Viewing can be divided into two fundamental operations:

- First, we must position and orient the camera. This transformation is performed by the model-view transformation.
- The second step is the application of the projection transformation (parallel or perspective). Objects within the specified clipping volume are placed into the same cube in clip coordinates.

Hidden-surface removal occurs after the fragment shader. Consequently, although an object might be blocked from the camera by other objects, even with hidden-surface removal enabled, the rasterizer will still generate fragments for blocked objects within the clipping volume.

4.3 Positioning of the Camera

At any given time, the model-view matrix encapsulates the relationship between the camera frame and the object frame. As its name suggests, the model-view transformation is the concatenation of two transformations:

- A modelling transformation that takes instances of objects in object coordinates and brings them into the world frame. For each object in a scene, we construct a transformation matrix, called the instance transformation, that will scale, orient, and translate the object to the desired location in the scene. This matrix can be derived by concatenating a series of basic transformation matrices.
- The viewing transformation transforms world coordinates to camera coordinates. We consider three ways to construct this transformation matrix.

The first method for constructing the view transformation is by concatenating a carefully selected series of affine transformations. We can think of the camera as being fixed at the origin, pointing down the negative z-axis. Thus, we transform (translate, rotate, etc.) the scene relative to the camera frame. For example, if we want to move farther away from an object located directly in front of the camera, we move the scene down the negative z-axis (i.e. translate by a negative z-value).

In the second approach, we specify the camera frame with respect to the world frame and construct the matrix, called the view-orientation matrix, that will take us from world coordinates to camera coordinates. In order to define the camera frame, we require three parameters to be specified:

- View-Reference Point (VRP): Specifies the location of the COP, given in world coordinates.
- View-Plane Normal (VPN): Also known as n, specifies the normal to the projection plane.
- View-up vector (VUP): Specifies what direction is up from the camera's perspective. This vector need not be perpendicular to n.

We project the VUP vector onto the view plane to obtain the up-direction vector, v, which is orthogonal to n. We then use the cross product (v$\times$ n) to obtain a third orthogonal direction u. This new orthogonal coordinate system usually is referred to as either the viewing-coordinate system or the u-v-n system.With the addition of the VRP, we have the desired camera frame.

The third method, called the look-at function, is similar to our second approach: it differs only in the way we specify the VPN. We specify a point, $\mathbf{e}$, called the eye point, which has exactly the same meaning as the VRP described above. Next, we define a point, $\mathbf{a}$, called the at point, at which the camera is pointing. Together, these points determine the VPN ($vpn = \mathbf{a} - \mathbf{e}$). The specification of VUP and the derivation of the camera frame is the same as above.

The second part of the viewing process, often called the normalization transformation, involves specifying and applying a specific projection matrix (parallel or perspective).

4.4 Parallel Projections

Projectors are parallel and point in a direction of projection (DOP).

Projection is a technique that takes the specification of points in three dimensions and maps them to points on a two-dimensional projection surface. Such a transformation is not invertible, because all points along a projector map into the same points on the projection surface.

Orthogonal or orthographic projections are a special case of parallel projections, in which the projectors are perpendicular to the projection plane.

Projection normalization is a technique that converts all projections into simple orthogonal projections by distorting the objects such that the orthogonal projection of the distorted objects is the same as the desired projection of the original objects. This is done by applying a matrix called the normalization matrix, also known as the projection matrix. Conceptually, the normalization matrix should be defined such that it transforms (distorts) the specified view volume to coincide exactly with the canonical (default) view volume. Consequently, vertices are transformed such that vertices within the specified view volume are transformed to vertices within the canonical view volume, and vertices outside the specified view volume are transformed to vertices outside the canonical view volume. The canonical view volume is the cube defined by the planes

$$x = \pm 1,$$
$$y = \pm 1,$$
$$z = \pm 1.$$

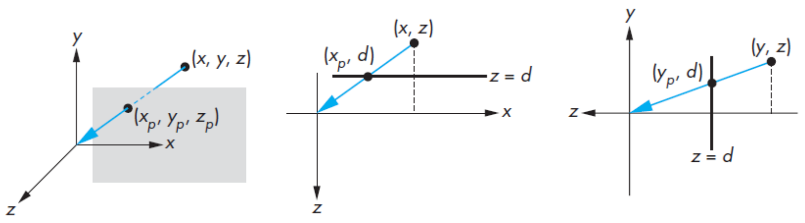
Two advantages of employing projection normalization are:

- Both perspective and parallel views can be supported by the same pipeline;
- The clipping process is simplified because the sides of the canonical view volume are aligned with the coordinate axes.

The shape of the viewing volume for an orthogonal projection is a right-parallelepiped. Thus, the projection normalization process for an orthographical projection requires two steps:

1. Perform a translation to move the centre of the specified view volume to the centre of the canonical view volume (the origin).
2. Scale the sides of the specified view volume such that they have a length of 2.

4.5 Perspective Projections



A point in space (x, y, z) is projected along a projector into the point (x_p, y_p, z_p). All projectors pass through the COP (origin), and, because the projection plane is perpendicular to the z-axis,

$$z_p = d.$$

Because the camera is pointing in the negative z-direction, d is negative. From the top view shown in the figure above, we see that two similar triangles are formed. Hence

$$\frac{x_p}{d} = \frac{x}{z}$$
$$\Rightarrow x_p = \frac{x}{z/d}$$

Using the side view:

$$\frac{y_p}{d} = \frac{y}{z}$$
$$\Rightarrow y_p = \frac{y}{z/d}$$

The division by z describes nonuniform foreshortening: The images of objects farther from the centre of projection are reduced in size(diminution) compared to the images of objects closer to the COP. Although this perspective transformation preserves lines, it is not affine. It is also irreversible: we cannot recover a point from its projection.

We can extend our use of homogeneous coordinates to handle projections. When we introduced homogeneous coordinates, we represented a point in three dimensions (x, y, z) by the point $(x, y, z, 1)$ in four dimensions. Suppose that, instead, we replace (x, y, z) by the four-dimensional point

$$\begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}.$$

As long as $w \neq 0$, we can recover the three-dimensional point from its four-dimensional representation by dividing the first three components by w; a process known as perspective division. By allowing w to change, we can represent a larger class of transformations, including perspective projections. Consider the matrix

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}.$$

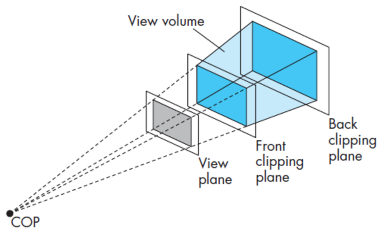
The matrix M transforms the point $[x, y, z, 1]^T$ to the point $[x, y, z, z/d]^T$. By performing perspective division (i.e. divide the first three components by the fourth), we obtain

$$\begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}.$$

Hence, matrix M can be used to perform a simple perspective projection. We apply the projection matrix after the model-view matrix, but remember that we must perform a perspective division at the end.

4.6 Perspective Projections with OpenGL

The shape of the view volume for a perspective projection is a frustum (truncated pyramid).



`Frustum()` and `Perspective()` are two APIs that can be used to specify a perspective projection matrix.

4.7 Perspective-Projection Matrices

A perspective-normalization transformation converts a perspective projection to an orthogonal projection.

4.8 Hidden-Surface Removal

The graphics system must be careful about which surfaces it displays in a three-dimensional scene. Algorithms that remove those surfaces that should not be visible to the viewer are called hidden-surface-removal algorithms, and algorithms that determine which surfaces are visible to the viewer are called visible-surface algorithms.

Hidden-surface-removal algorithms can be divided into two broad classes:

- Object-space algorithms attempt to order the surfaces of the objects in the scene such that rendering surfaces in a particular order provides the correct image. This class of algorithms does not work well with pipeline architectures in which objects are passed down the pipeline in an arbitrary order. The graphics system must have all the objects available so it can sort them into the correct back-to-front order.
- Image-space algorithms work as part of the projection process and seek to determine the relationship among object points on each projector.

The z-buffer algorithm is an image-space algorithm that fits in well with the rendering pipeline. As primitives are rasterized, we keep track of the distance from the COP or the projection plane to the closest point on each projector that has already been rendered. We update this information as successive primitives are projected and filled. Ultimately, we display only the closest point on each projector. The algorithm requires a depth buffer, or z-buffer, to store the necessary depth information as primitives are rasterized. The z-buffer forms part of the frame buffer and has the same spatial resolution as the colour buffer.

Major advantages of this algorithm are that its complexity is proportional to the number of fragments generated by the rasterizer and that it can be implemented with a small number of additional calculations over what we have to do to project and display polygons without hidden-surface removal.

Culling: for a convex object, such as the cube, faces whose normals point away from the viewer are never visible and can be eliminated or culled before the rasterization process commences.

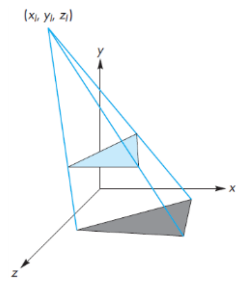
4.9 Displaying Meshes

A mesh is a set of polygons that share vertices and edges. We use meshes to display, for example, height data. Height data determine a surface, such as terrain, through either a function that gives the heights above a reference value, such as elevations above sea level, or through samples taken at various points on the surface. For the sake of efficiency and simplicity, we should strive to organize this data in a way that will allow us to drawn it using a combination of triangle strips and triangle fans.

4.10 Projections and Shadows

Shadows are important components of realistic images and give many visual clues to the spatial relationships among the objects in a scene. Shadows require a light source to be present. If the only light source is at the centre of projection, a model known as “flashlight in the eye”, then there are no visible shadows, because any shadows are behind visible objects. To add physically correct shadows, we must understand the interaction between light and material properties.

Consider a simple shadow that falls on the surface $y = 0$. Not only is this shadow a flat polygon, called a shadow polygon, but it is also the projection of the original polygon onto this surface. More specifically, the shadow polygon is the projection of the polygon onto the surface with the centre of projection at the light source. It is possible to compute the vertices of the shadow polygon by means of a suitable projection matrix.



Lesson 5: Lighting and Shading

Local lighting models, as opposed to global lighting models, allow us to compute the shade to assign to a point on a surface, independent of any other surfaces in the scene. The calculations depend only on the material properties assigned to the surface, the local geometry of the surface, and the locations and properties of the light sources. This model is well suited for a fast pipeline graphics architecture.

5.1 Light and Matter

From a physical perspective, a surface can either emit light by self-emission, as a light bulb does, or reflect light from other surfaces that illuminate it. Some surfaces may both reflect light and emit light at the same time. When we look at a point on an object, the colour that we see is determined by multiple interactions among light sources and reflective surfaces: we see the colour of the light reflected from the surface toward our eyes.

Interactions between light and materials can be classified into three groups:

1. Specular surfaces appear shiny because most of the light that is reflected or scattered is in a narrow range of angles close to the angle of reflection. With a perfectly specular surface, an incoming light ray may be partially absorbed, but all reflected light from a given angle emerges at a single angle, obeying the rule that the angle of incidence is equal to the angle of reflection.
2. Diffuse surfaces are characterized by reflected light being scattered in all directions. Perfectly diffuse surfaces scatter light equally in all directions.
3. Translucent surfaces allow some light to penetrate the surface and to emerge from another location on the object. This process of refraction characterizes glass and water.

5.2 Light Sources

We describe a light source through a three-component intensity, or luminance, function:

$$I = \begin{bmatrix} I_{\mathrm{r}} \\ I_{\mathrm{g}} \\ I_{\mathrm{b}} \end{bmatrix}.$$

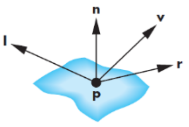
Each component represents the intensity of the independent red, green, and blue components.

We consider four basic types of sources:

- Ambient light: In many rooms, such as class rooms, the lights have been designed and positioned to provide uniform illumination throughout the room. Ambient illumination is characterized by an intensity, I_{a}, that is identical at every point in the scene.
- Point sources: An ideal point source emits light equally in all directions. The intensity of illumination received from a point source is proportional to the inverse square of the distance between the source and surface, called the distance term.
- Spotlights: Spotlights are characterized by a narrow range of angles (a cone) through which light is emitted. More realistic spotlights are characterized by the distribution of light within the cone – usually with most of the light concentrated in the centre of the cone.
- Distant light source: All rays are parallel and we replace the location (point) of the light source with the direction (vector) of the light.

5.3 The Phong Reflection Model

The Phong model uses the four vectors shown in the diagram to calculate a colour for an arbitrary point $\boldsymbol{p}$ on a surface. The vector $\boldsymbol{n}$ is the normal at $\boldsymbol{p}$; the vector $\boldsymbol{v}$ is in the direction from $\boldsymbol{p}$ to the viewer or COP; the vector $\boldsymbol{l}$ is in the direction of a line from $\boldsymbol{p}$ to an arbitrary point on the light source; finally, the vector $\boldsymbol{r}$ is in the direction that a perfectly reflected ray from $\boldsymbol{l}$ would take.. Note that $\boldsymbol{r}$ is determined by $\boldsymbol{n}$ and $\boldsymbol{l}$.



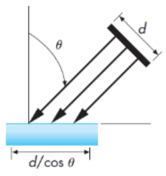
The Phong model supports the three types of material-light interactions: ambient, diffuse, and specular. For each light source we can have separate ambient, diffuse, and specular components for each of the three primary colours. Thus we need nine coefficients to characterize these terms at any point $\boldsymbol{p}$. We can place these coefficients in a 3×3 illumination matrix for the *i*th light source:

$$L_i = \begin{bmatrix} L_{i\mathrm{ra}} & L_{i\mathrm{ga}} & L_{i\mathrm{ba}} \\ L_{i\mathrm{rd}} & L_{i\mathrm{gd}} & L_{i\mathrm{bd}} \\ L_{i\mathrm{rs}} & L_{i\mathrm{gs}} & L_{i\mathrm{bs}} \end{bmatrix}$$

The intensity of ambient light I_a is the same at every point on the surface. Some of this light is absorbed and some is reflected. The amount reflected is given by the ambient reflection coefficient, k_a $(0 \leq k_\mathrm{a} \leq 1)$. Thus: $I_\mathrm{a} = k_\mathrm{a} L_\mathrm{a}$

A perfectly diffuse reflector scatters the light that it reflects equally in all directions. Hence, such a surface appears the same to all viewers (i.e. neither $\boldsymbol{v}$ nor $\boldsymbol{r}$ need be considered). The amount of light reflected depends both on the material – because some of the incoming light is absorbed – and on the position of the light source relative to the surface. Diffuse surfaces, sometimes called Lambertian surfaces, can be modelled mathematically with Lambert's law. According to Lambert's law, we see only the vertical component of the incoming light. Lambert's law states that

$$R_d \propto \cos\theta$$



where θ is the angle between the normal at the point of interest $\boldsymbol{n}$ and the direction of the light source $\boldsymbol{l}$. If both $\boldsymbol{l}$ and $\boldsymbol{n}$ are unit-length vectors, then
If we add in a reflection coefficient k_d $(0 \leq k_\mathrm{d} \leq 1)$ representing the fraction of incoming diffuse light that is reflected, we have the diffuse reflection term:

$$I_\mathrm{d} = k_\mathrm{d} L_\mathrm{d} (\boldsymbol{l} \cdot \boldsymbol{n}).$$

Specular reflection adds a highlight that we see reflected from shiny objects.The amount of light that the viewer sees depends on the angle ϕ between $\boldsymbol{r}$, the direction of a perfect reflector and $\boldsymbol{v}$, the direction of the viewer. The Phong model uses the equation

$$I_s = k_s L_s \cos^\alpha \phi$$

The coefficient k_s $(0 \leq k_s \leq 1)$ is the fraction of the incoming specular light that is reflected. The exponent α is a shininess coefficient. As α is increased, the reflected light is concentrated in a narrower region centred on the angle of a perfect reflector. Values in the range 100 to 500 correspond to most metallic surfaces. If $\boldsymbol{r}$ and $\boldsymbol{v}$ are normalized, then

$$I_s = k_s L_s (\boldsymbol{r} \cdot \boldsymbol{v})^\alpha$$

The Phong model, including the distance term, is written

$$I = \frac{1}{a + bd + cd^2}\left(k_\mathrm{d}L_\mathrm{d}\max(\boldsymbol{l}\cdot\boldsymbol{n}, 0) + k_sL_s\max((\boldsymbol{r}\cdot\boldsymbol{v})^\alpha, 0)\right) + k_\mathrm{a}L_\mathrm{a}.$$

This formula is computed for each light source and for each primary.

If we use the Phong model with specular reflections, the dot product $\boldsymbol{r}\cdot\boldsymbol{v}$ sould be recalculated at every point on the surface. An approximation to this involves the unit vector halfway between the viewer vector and the light-source vector:

$$\boldsymbol{h} = \frac{\boldsymbol{l} + \boldsymbol{v}}{|\boldsymbol{l} + \boldsymbol{v}|}.$$

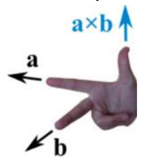
If we replace $\boldsymbol{r}\cdot\boldsymbol{v}$ with $\boldsymbol{n}\cdot\boldsymbol{h}$, we avoid calculating $\boldsymbol{r}$. When we use the halfway vector in the calculation of the specular term, we are using the Blim-Phong, or modified Phong, lighting model.

5.4 Computation of Vectors

If we are given three noncolinear points – p_0, p_1, p_2 – we can calculate the normal to the plane in which they lie as follows

$$\boldsymbol{n} = (p_2 - p_0) \times (p_1 - p_0).$$

The order in which we cross-multiply vectors determines the direction of the resulting vector (see illustration).



At every point (x, y, z) on the surface of a sphere centred at the origin, we have that

$$\boldsymbol{n} = (x, y, z).$$

To calculate $\boldsymbol{r}$, we first normalize both $\boldsymbol{l}$ and $\boldsymbol{n}$, and then use the following equation

$$\boldsymbol{r} = 2(\boldsymbol{l}\cdot\boldsymbol{n})\boldsymbol{n} - \boldsymbol{l}.$$

GLSL provides a function, `reflect()`, which we can use in our shaders to compute $\boldsymbol{r}$.

5.5 Polygonal Shading

A polygonal mesh comprises many flat polygons, each of which has a well-defined normal. We consider three ways to shade these polygons:

- Flat shading (or constant shading): The shading calculation is carried out only once for each polygon, and each point on the polygon is assigned the same shade. Flat shading will show differences in shading among adjacent polygons. We will see stripes, known as Mach bands, along the edges.
- Gouraud shading (or smooth shading): The lighting calculation is done at each vertex using the material properties and the vectors $\boldsymbol{n}$, $\boldsymbol{v}$, and $\boldsymbol{l}$. Thus, each vertex will have its own colour that the rasterizer can use to interpolate a shade for each fragment. We define the normal at a vertex to be the normalized average of the normals of the polygons that share the vertex. We implement Gouraud shading either in the application or in the vertex shader.
- Phong shading: Instead of interpolating vertex intensities (colours), we interpolate normals across each polygon. We can thus make an independent lighting calculation for each fragment. We implement Phong shading in the fragment shader.

5.6 Approximation of a Sphere by Recursive Subdivision

The sphere is not a primitive type supported by OpenGL. By a process known as recursive subdivision, we can generate approximations to a sphere using triangles. Recursive subdivision is a powerful technique for generating approximations to curves and surfaces to any desired level of accuracy.

5.7 Specifying Lighting Parameters

For every light source, we must specify its colour and either its location (for a point source or spotlight) or its direction (for a distant source).The colour of a source will have three components – ambient, diffuse, and specular – that we can specify. For positional light sources, we may also want to account for the attenuation of light received due to its distance from the source. We can do this by using the distance –attenuation model

$$f(d) = \frac{1}{a + bd + cd^2},$$

which contains constant, linear, and quadratic terms.

Material properties should match up directly with the supported light sources and with the chosen reflection model. We specify ambient, diffuse, and specular reflectivity coefficients (k_a, k_d, k_s) for each primary colour via three colours using either RGB or RGBA colours. Note that often the diffuse and specular reflectivity coefficients are the same. For the specular component, we also need to specify its shininess coefficient.

We also want to allow for scenes in which a light source is within the view volume and thus might be visible. We can create such effects by including an emissive component that models self-luminous sources. This term is unaffected by any of the light sources, and it does not affect any other surfaces. It simply adds a fixed colour to the surface.

5.8 Implementing a Lighting Model

Because light from multiple sources is additive, we can repeat our lighting calculation for each source and add up the individual contributors.

We have three choices as to where we do lighting calculations: in the application, in the vertex shader, or in the fragment shader. But for the sake of efficiency, we will almost always want to do lighting calculations in the shaders.

Light sources are special types of geometric object and have geometric attributes, such as position, just like polygons and points. Hence, light sources can be affected by transformations.

To implement lighting in the shader, we must carry out three steps:

1. Chose a lighting model. Do we use the Blim-Phong or some other model? Do we include distance attenuation? Do we want two-sided lighting?
2. Write the shader to implement the model.
3. Finally, we have to transfer the necessary data to the shader. Some data can be transferred using uniform variables, and other data can be transferred as vertex attributes.

On page 318: The calculation view_direction = v – origin; should be view_direction = origin – v;

5.9 Shading of the Sphere Model

The smoother the shading, the fewer polygons we need to model a sphere (or any curved surface). To obtain the smoothest possible display we can get with relatively few triangles; is to use the actual normals of the sphere for each vertex in the approximation. For a sphere centred at the origin, the normal at a point **p** is simply **p**.

5.10 Per-Fragment Lighting

By doing the lighting calculations on a per-fragment basis, as opposed to a per-vertex basis, we can obtain highly smooth and realistic looking shadings. With a fragment shader, we can do an independent lighting calculation for each fragment. The fragment shader needs to get the interpolated values of the normal vector, the light source position, and the eye position from the rasterizer.

Programmable shaders make it possible to not only incorporate more realistic lighting models in real time but also to create interesting non-photorealistic effects. Two such examples are the use of only a few colours and emphasizing the edges in objects.

5.11 Global Illumination

Since each object is shaded independently, there are limitations imposed by the local lighting models. Shadows, reflections, and blockage of light are global effects and require a global lighting model, but these models are incompatible with the pipeline architecture. Rendering strategies, including ray tracing and radiosity, can handle global effects. Ray tracing starts with the synthetic-camera model but determines for each projector that strikes a polygon if that point is indeed illuminated by one or more sources before computing the local shading at each point. A radiosity renderer is based upon energy considerations. A ray tracer is best suited to a scene consisting of highly reflective surfaces, whereas a radiosity renderer is best suited for a scene in which all the surfaces are perfectly diffuse.

Lesson 6: From Vertices to Fragments

Clipping involves eliminating objects that lie outside the viewing volume and thus cannot be visible in the image. Rasterization produces fragments from the remaining objects. Hidden-surface removal determines which fragments correspond to objects that are visible, namely, those that are in the view volume and are not blocked from view by other objects closer to the camera.

6.1 Basic Implementation Strategies

At a high level, we can consider the graphics system as a black box whose inputs are the vertices and states defined in the program – geometric objects, attributes, camera specifications – and whose output is an array of coloured pixels in the frame buffer. Within this black box, we must do many tasks, including transformations, clipping shading, hidden-surface removal, and rasterization of the primitives that can appear on the display. Every geometric object must be passed through this system, and we must assign a colour to every pixel in the colour buffer that is displayed.

In the object-oriented approach, we loop over the objects. A pipeline renderer fits this description Vertices flow through a sequence of modules that transform them, colours them, and determines whether they are visible. Data (vertices) flow ***forward*** through the system. Because we are doing the same operations on every primitive, the hardware to build an object-based system is fast and relatively inexpensive. Because each geometric primitive is processed independently, the main limitation of object-oriented implementations is that they cannot handle most global calculations.

Image-oriented approaches loop over pixels, or rows of pixels called scan-lines, that constitute the frame buffer. For each pixel, we work backward, trying to determine which geometric primitives can contribute to its colour. The main disadvantage of this approach is that all the geometric data must be available at all times during the rendering process.

6.2 Four Major Tasks

There are four major tasks that any graphics system must perform to render a geometric entity:

1. Modelling: The results of the modelling process are sets of vertices that specify a group of geometric objects supported by the rest of the system. Because the modeller knows the specifics of the application, it can often use a good heuristic to eliminate many, if not most, primitives before they are sent on through the standard viewing process.
2. Geometry processing: The goals of the geometry processor are to determine which geometric objects can appear on the display and to assign shades or colours to the vertices of these objects. Four processes are required: projection, primitive assembly, clipping, and shading. The first step is to apply the model-view transformation. The second step is to transform vertices using the projection transformation. Vertices are now represented in clip coordinates. Before clipping can take place, vertices must be grouped into primitives. After clipping takes place, the remaining vertices are still in four-dimensional homogeneous coordinates. Perspective division converts them to three-dimensional representations in normalized device coordinates. Per-vertex shading is also performed during this stage.
3. Rasterization (scan conversion): The rasterizer starts with vertices in normalized device coordinates and outputs fragments whose locations are in units of the display - window coordinates. Although only the x and y values of the vertices are needed to determine

which pixels in the frame buffer can be affected by the primitive, we still need to retain depth information for hidden-surface removal. For line segments, rasterization determines which fragments should be used to approximate a line segment between the projected vertices. For polygons, rasterization determines which pixels lie inside the two-dimensional polygon determined by the projected vertices. The colours that we assign to these fragments can be determined by the vertex attributes or obtained by interpolating the shades at the vertices that were computed. Screen coordinates refer to the two-dimensional system that is the same as window coordinates but lacks the depth coordinate.

4. Fragment processing: Per-fragment shading, texture-mapping, bump-mapping, alpha blending, antialiasing, and hidden-surface removal all take place in this stage.

6.3 Clipping

Clipping is the process of determining which primitives, or parts of primitives, should be eliminated because they lie outside the viewing volume. Clipping is done before the perspective division that is necessary if the w component of a clipped vertex is not equal to 1.The portions of all primitives that can possibly be displayed lie within the cube

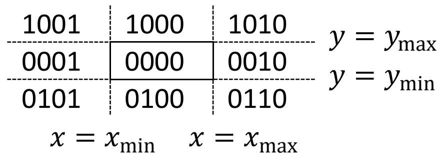
$$\begin{gathered} w \geq x \geq -w, \\ w \geq y \geq -w, \\ w \geq z \geq -w. \end{gathered}$$

Note that projection has been carried out only partially at this stage: perspective division and the final orthographic projection must still be performed.

6.4 Line-Segment Clipping

A clipper decides which primitives, or parts of primitives, can possibly appear on the display and be passed on to the rasterizer. Primitives that fit within the specified view volume pass through the clipper (are accepted); those that fall outside are eliminated (or rejected or culled); and those that are only partially within the view volume must be clipped such that any part lying outside the volume is removed.

Cohen-Sutherland clipping: The algorithm starts by extending the sides of the clipping rectangle to infinity, thus breaking up space into the nine regions shown in the diagram below.

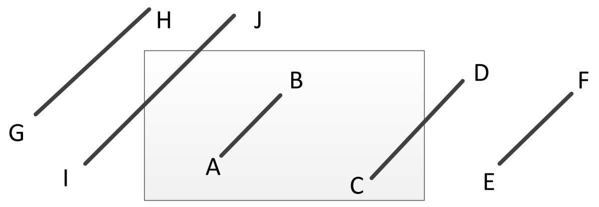


Each region is assigned a unique 4-bit binary number called an outcode, $b_0 b_1 b_2 b_3$, as follows. Suppose that (x, y) is a point in the region; then

$$b_0 = \begin{cases} 1 & \text{if } y > y_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Likewise, b_1 is 1 if $y < y_{\min}$, and b_2 and b_3 are determined by the relationship between x and the left and right sides of the clipping window. The resulting codes are indicated in the diagram above. For each endpoint of a line segment, we first compute the endpoint's outcode. Consider a line segment whose outcodes are given by $o_1 = outcode(x_1, y_1)$ and $o_2 = outcode(x_2, y_2)$. There are four cases to consider:

1. ($o_1 = o_2 = 0$). Both endpoints are inside the clipping window, as is true for segment AB in the figure below. The entire line segment can be sent on to be rasterized.
2. ($o_1 \neq 0, o_2 = 0$; or vice versa). One endpoint is inside the clipping window; one is outside (see segment CD in the figure below). The line segment must be shortened. The nonzero outcode indicates which edge or edges of the window are crossed by the segment. One or two intersections must be computed. Note that after one intersection is computed, we can compute the outcode of the point of intersection to determine whether another intersection calculation is required.
3. ($o_1 \,\&\, o_2 \neq 0$). By taking the bitwise AND of the outcodes, we determine whether or not the two endpoints lie on the same outside side of the window. If so, the line segment can be discarded (see segment EF in the figure below).
4. ($o_1 \,\&\, o_2 = 0$). Both endpoints are outside, but they are on the outside of different edges of the window. As we can see from segments GH and IJ in the figure below, we cannot tell from just the outcodes whether the segment can be discarded or must be shortened. The best we can do is to intersect with one of the sides of the window and to check the outcode of the resulting point.



Thus, with this algorithm we do intersection calculations only when they are needed, as in the second case, or where the outcodes did not contain enough information, as in the fourth case. The Cohen-Sutherland algorithm works best when there are many line segments but few are actually displayed (line segments lie fully outside one or two of the extended sides of the clipping rectangle). This algorithm can be extended to three dimensions. The main disadvantage of the algorithm is that it must be used recursively.

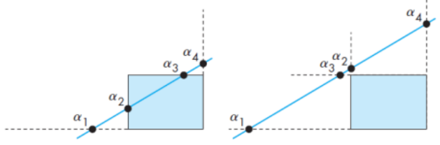
Liang-Barsky Clipping: Suppose that we have a line segment defined by the two endpoints $p_1 = [x_1, y_1]^T$ and $p_2 = [x_2, y_2]^T$. We can parametrically express this line in either matrix form:

$$p(\alpha) = (1 - \alpha)p_1 + \alpha p_2,$$

or as two scalar equations:

$$x(\alpha) = (1 - \alpha)x_1 + \alpha x_2,$$
$$y(\alpha) = (1 - \alpha)y_1 + \alpha y_2.$$

As the parameter α varies from 0 to 1, we move along the segment from p_1 to p_2; negative values of α yield points on the line on the other side of p_1 from p_2; values of $\alpha > 1$ gives points on the line past p_2. Consider a line segment and the line of which it is part, as shown in the figure below



As long as the line is not parallel to a side of the window (if it is, we can handle that situation with ease), there are four points where the line intersects the extended sides of the window. These points correspond to the four values of the parameter: α_1 (bottom), α_2 (left), α_3 (top) and α_4 (right). We can order these values and determine which correspond to intersections (if any) that we need for clipping. For the first example, $0 < \alpha_1 < \alpha_2 < \alpha_3 < \alpha_4 < 1$. Hence, all four intersections are inside the original line segment, with the two innermost (α_2 and α_3) determining the clipped line segment.The case in the second example also has the four intersections between the endpoints of the line segment, but notice that the order for this case is $0 < \alpha_1 < \alpha_3 < \alpha_2 < \alpha_4 < 1$. The line intersects both the top and the bottom of the window before it intersects either the left or the right; thus, the entire line segment must be rejected. Other cases of the ordering of the points of intersection can be argued in a similar way.

The efficiency of this approach, compared to that of the Cohen-Sutherland algorithm, is that we avoid multiple shortening of line segments and the related re-executions of the clipping algorithm.

6.8 Rasterization

Pixels have attributes that are colours in the colour buffer.

Fragments are potentially pixels. Each fragment has a colour attribute and a location in screen coordinates that corresponds to a location in the colour buffer. Fragments also carry depth information that can be used for hidden-surface removal.

The DDA algorithm (Digital Differential Analyser): Suppose that we have a line segment defined by the endpoints (x_1, y_1) and (x_2, y_2). Because we are working in a colour buffer, we assume that these are all integer values. The slope of his line is given by

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}.$$

This algorithm is based on writing a pixel for each value of `ix` in `write_pixel` as x goes from x_1 to x_2. . For any change in x equal to Δx, the corresponding changes in y must be

$$\Delta y = m\Delta x.$$

As we move from x_1 to x_2, we increase x by 1 in each iteration; thus, we must increase y by

$$\Delta y = m.$$

This algorithm in pseudo code is:

```
for(ix = x1; ix <= x2;ix++){
      y += m;
      write_pixel(x, round(y), line_colour);
}
```

For large slopes, the separation between fragments can be large, generating an unacceptable approximation to the line segment. To alleviate this problem, for slopes greater than 1, we swap the roles of x and y.

6.9 Bresenham's Algorithm

The DDA algorithm requires a floating-point addition for each pixel generated. The Bresenham algorithm avoids all floating point calculations and has become the standard algorithm used in hardware and software rasterizers. The calculation of each successive pixel in the colour buffer requires only an addition and a sign test.

6.10 Polygon Rasterization

One of the major advantages that the first raster systems brought to users was the ability to display filled polygons.

Flat simple polygons have well-defined interiors. If they are also convex, they are guaranteed to be rendered correctly by OpenGL.

Inside-outside testing: Conceptually, the process of filling the inside of a polygon with a colour or pattern is equivalent to deciding which points in the plane of the polygon are interior (inside) points.

- The crossing (or odd-even) test is the most widely used test for making inside-outside decisions. Suppose that **p** is a point inside a polygon. Any ray emanating from **p** and going off to infinity must cross an odd number of edges. Any ray emanating from a point outside the polygon and entering the polygon crosses an even number of edges before reaching infinity. Usually, we replace rays through points with scan-lines, and we count the crossing of polygon edges to determine inside and outside.
- The winding test considers the polygon as a knot being wrapped around a point or a line. We start by traversing the edges of the polygon from any starting vertex and going around the edge in a particular direction until we reach the starting point. Next we consider an arbitrary point. The winding number for this point is the number of times it is encircled by the edges of the polygon. We count clockwise encirclements as positive and counter-clockwise encirclements as negative. A point is inside the polygon if its winding number is not zero.

Since OpenGL can only render triangles; to render a more complex (non-flat or concave) polygon, we can apply a tessellation algorithm to this polygon to subdivide it into triangles. A goo tessellation should not produce triangles that are long and thin; it should, if possible, produce sets of triangles that can use supported features, such as triangle strips and triangle fans.

Flood fill: This algorithm works directly with pixels in the frame buffer. We first rasterize a polygon's edges into the frame buffer using Bresenham's algorithm. If we can find an initial point (x, y) that lie inside these edges (called a seed point), we can look at its neighbouring pixels recursively, colouring them with the fill colour only if they are not coloured already. We can obtain a number of variants of flood fill by removing the recursion. One way to do so is to work one scan-line at a time (called scan-line fill).

6.11 Hidden Surface Removal

Hidden-surface removal (or visible-surface determination) is done to discover what part, if any, of each object in the view volume is visible to the viewer or is obscured from the viewer by other objects.

Culling: For situations where we cannot see back faces, such as scenes composed of convex polyhedral, reduce the work required for hidden-surface removal by eliminating all back-facing polygons before we apply any other hidden-surface-removal algorithms. A polygon is facing forward if and only if $n \cdot v \geq 0$, where v is in the direction of the viewer and n is the normal to the front face of the polygon. Usually, culling is performed after the transformation to normalized device coordinates (perspective division).

The z-buffer is a buffer which usually has the same spatial resolution as the colour buffer, and before each scene rendering, each of its elements is initialized to a depth corresponding to the maximum distance away from the centre of projection. At any time during rasterization and fragment processing, each location in the z-buffer contains the distance along the ray corresponding to the location of the closest polygon found so far. Rasterization is done polygon by polygon using some rasterization algorithm. For each fragment on the polygon corresponding to the intersection of the polygon with a ray through a pixel, we compute the depth from the centre of projection. The method compares this depth to the value in the z-buffer corresponding to this fragment. If this depth is greater than the depth in the z-buffer, this fragment is discarded. If the depth is less than the depth in the z-buffer, update the depth in the z-buffer and place the shade computed for this fragment at the corresponding location in the colour buffer. The z-buffer algorithm is the most widely used hidden-surface-removal algorithm. It has the advantages of being easy to implement, in either hardware or software, and of being compatible with pipeline architectures, where it can execute at the speed at which fragments are passing through the pipeline.

Suppose that we have already computed the z-extents of each polygon. The next step of depth sort is to order all the polygons by how far away from the viewer their maximum z-value is. If no two polygons' z-extents overlap, we can paint the polygons back to front and we are done. However, if the z-extents of two polygons overlap, we still may be able to find an order to paint (render) the polygons individually and yield the correct image. The depth-sort algorithm runs a number of increasingly more difficult tests, attempting to find such an ordering. Two troublesome situations remain: If three or more polygons overlap cyclically, or if a polygon can pierce another polygon, there is no correct order for painting without having to subdivide some polygons. The main idea behind this class of algorithms is that if one object obscures part of another then the first object is painted after the object that it obscures. The painter's algorithm is an example of such a method.

6.12 Antialiasing

Rasterized line segments and edges of polygons can appear jagged. Aliasing errors are caused by three related problems with the discrete nature of the frame buffer:

1. If we have an $n \times m$ frame buffer, the number of pixels is fixed, and we can generate only certain patterns to approximate a line segment.
2. Pixel locations are fixed on a uniform grid; regardless of where we would like to place pixels, we cannot place them at other than evenly spaced locations.
3. Pixels have a fixed size and shape.

The scan-conversion algorithm forces us, for lines of slope less than 1, to choose exactly one pixel value for each value of x. If, instead, we shade each box by the percentage of the ideal line that crosses it, we get a smoother looking rendering. This technique is known as antialiasing by area averaging. If polygons share a pixel, and each polygon has a different colour, the colour assigned to the pixel is the one associated with the polygon closest to the viewer. We could obtain a much more accurate image if we could assign a colour based on an area-weighted average of the colours of these polygons. Such algorithms can be implemented with fragment shaders on hardware with floating point frame buffers. These algorithms are collectively known as spatial-domain aliasing.

Lesson 7: Discrete Techniques

7.4 Mapping Methods

Mapping algorithms can be thought of as either modifying the shading algorithm based on a two-dimensional array, the map, or as modifying the shading by using the map to alter surface parameters, such as material properties and normals. There are three major techniques:

- Texture mapping: Uses an image (texture) to influence the colour of a fragment. The texture image can either be a digitized image or generated by a procedural texture-generation method.
- Bump mapping: Distorts the normal vectors during the shading process to make the surface appear to have small variations in shape, such as the bumps on a real orange.
- Environment mapping (or reflection mapping): Allow us to create images that have the appearance of reflected materials without our having to trace reflected rays. An image of the environment is painted onto the surface as that surface is being rendered.

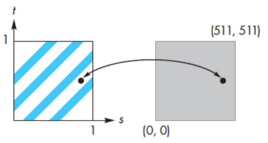
All three methods:

- Alter the shading of individual fragments as part of fragment processing;
- Rely on the map being stored as a one-, two-, or three-dimensional digital image;
- Keep the geometric complexity low while creating the illusion of complex geometry; and
- Are subject to aliasing errors.

7.5 Texture Mapping

Conceptually, texture mapping is the process of mapping a small area of the texture pattern to the area of the geometric surface, corresponding to a pixel in the final image.

In most applications, textures start out as two-dimensional images that might be formed by application programs or scanned in from a photograph, but, regardless of their origin, they are eventually brought into processor memory as arrays. We call the elements of these arrays texels. We can think of this array as a continuous rectangular two-dimensional texture pattern $T(s, t)$. The independent variables s and t are known as texture coordinates and vary over the interval [0, 1].



7.6 Texture Mapping in OpenGL

OpenGL's texture maps rely on its pipeline architecture. There are actually two parallel pipelines: the geometric pipeline and the pixel pipeline. For texture mapping, the pixel pipeline merges with fragment processing after rasterization. Texture mapping is done as part of fragment processing.

Texture mapping requires interaction among the application program, the vertex shader, and the fragment shader. There are three basic steps:

1. We must form a texture image and place it in texture memory on the GPU;
2. We must assign texture coordinates to each fragment;
3. We must apply the texture to each fragment.

Texture objects allow the application program to define objects that consist of the texture array and the various texture parameters that control its application to surfaces. Many of the complexities of how we can apply the texture are inside the texture object and thus will allow us to use very simple fragment shaders. To create a texture object:

1. Get some unused texture identifiers by calling `glGenTextures()`;
2. Bind the texture object (`glBindTexture()`) to make it the current texture object;
3. Use texture functions to specify the texture image and its parameters, which become part of the current texture object.

The key element in applying a texture in the fragment shader is the mapping between the location of a fragment and the corresponding location within the texture image where we will get the texture colour for that fragment. We specify texture coordinates as a vertex attribute in the application. We then pass these coordinates to the vertex shader and let the rasterizer interpolate the vertex texture coordinates to fragment texture coordinates. The key to putting everything together is a variable called a sampler which most often appears only in a fragment shader. A sampler variable provides access to a texture object, including all its parameters.

Aliasing of textures is a major problem. When we map texture coordinates to the array of texels, we rarely get a point that corresponds to the centre of a texel. There are two basic strategies:

- Point sampling: We use the value of the texel that is closest to the texture coordinate output by the rasterizer. This strategy is the one most subject to visible aliasing errors.
- Linear filtering: We use a weighted average of a group of texels in the neighbourhood of the texel determined by point sampling. This results in smoother texturing.

The size of the pixel that we are trying to colour on the screen may be smaller or larger than one texel (i.e. the resolution of the texture image does not match the resolution of the area on the screen to which the texture is mapped). If the texel is larger than one pixel (the resolution of the texture image is less than that of the area on the screen), we call it magnification; if the texel is smaller than one pixel (the resolution of the texture image is greater than that of the area on the screen), it is called minification.

Mipmaps can be used to deal with the minification problem. For objects that project to an area of screen space that is small compared with the size of the texel array, we do not need the resolution of the original texel array. OpenGL allows us to create a series of texture arrays, called the mipmap hierarchy, at reduced sizes. OpenGL will automatically use the appropriate sized mipmap from the mipmap hierarchy.

7.8 Environment Maps

Highly reflective surfaces are characterized by specular reflections that mirror the environment. Environment mapping or reflection mapping is a variant of texture mapping that can give approximate results that are visually acceptable. We can achieve this effect by using a two-step rendering pass:

1. In the first pass, we render the scene without the reflecting object, say a mirror, with the camera placed at the centre of the mirror pointed in the direction of the normal of the mirror. . Thus, we obtain an image of the objects in the environment as “seen” by the mirror.
2. We can then use this image to obtain the shades (texture values) to place on the mirror for the second, normal, rendering with the mirror placed back in the scene.

There are two difficulties with this approach:

1. The images that we obtain in the first pass are not quite correct, because they have been formed without one of the objects – the mirror – in the environment.
2. The mapping issue: onto what surface should we project the scene in the first pass, and where should we place the camera?

The classic approach to solve the second difficulty is to project the environment onto a sphere centred at the centre of projection. OpenGL supports a variation of this method called sphere mapping. The application program supplies a circular image that is the orthographic projection of the sphere onto which the environment has been mapped. The advantage of this method is that the mapping from the reflection vector to two-dimensional texture coordinates on this circle is simple and can be implemented in either hardware or software. The difficult part is obtaining the required circular image.
Another approach is to compute six projections, corresponding to the six sides of a cube, using six virtual cameras located at the centre of the box, each pointing in a different direction. Once we computed the six images, we can specify a cube map in OpenGL with six function calls, one for each face of a cube centred at the origin. The advantage of this approach is that we can compute the environment map using the standard projections that are supported by the graphics system.

These techniques are examples of multi-pass rendering (or multi-rendering) techniques, where, in order to compute a single image, we compute multiple images, each using the rendering pipeline.

7.10 Bump Mapping

Bump mapping is a texture-mapping technique that can give the appearance of great complexity in an image without increasing the geometric complexity. The technique of bump mapping varies the apparent shape of the surface by perturbing the normal vectors as the surface is rendered; the colours that are generated by shading then show a variation in the surface properties. Unlike simple texture mapping, bump mapping will show changes in shading as the light source or object moves, making the object appear to have variations in surface smoothness. Bump mapping cannot be done in real time without programmable shaders. In the case of bump mapping, the texture map, called a normal map, stores displacement (height) values.

7.11 Compositing Techniques

OpenGL provides a mechanism through alpha (α) blending, that can, among other effects, create images with translucent objects.The alpha channel is the fourth colour in RGBA (or RGBα) colour mode. Like the other colours, the application program can control the value of A (or α) for each pixel. However, in RGBA mode, if blending is enabled, the value of α controls how the RGB values are written into the frame buffer. Because fragments from multiple objects can contribute to the colour of the same pixel, we say that these objects are blended or composited together.

The opacity of a surface is a measure of how much light penetrates through that surface. An opacity of 1 ($\alpha = 1$) corresponds to a completely opaque surface that blocks all light incident on it. A surface with an opacity of 0 is transparent; all light passes through it. The transparency or translucency of a surface with opacity α is given y $1 - \alpha$.

To use blending in OpenGL is straightforward:

1. Enable blending by `glEnable(GL_BLENDING);`
2. Set up the desired source and destination factors by `glBlendFunc(source_factor, destination_factor);`
3. The application program must use RGBA colours.

The major difficulty with compositing is that for most choices of the blending factors, the order in which we render the polygons affects the final image. Consequently, unlike most OpenGL programs where the user does not have to worry about the order in which polygons are rasterized, to get a desired effect we must now control this order within the application. In applications where handling of translucency must be done in a consistent and realistic manner, we often must sort the polygons from front to back within the application. Then depending on the application, we can do a front-to-back or back-to-front rendering using OpenGL's blending functionality.

A more subtle but visibly apparent problem occurs when we combine opaque and translucent objects in a scene. In a scene containing both opaque and translucent polygons, any polygon (or part of a polygon) behind an opaque polygon should not be rendered, but polygons (or parts of polygons) behind translucent polygons should be composited. If all polygons are rendered with the standard z-buffer algorithm, compositing will not be performed correctly, particularly if a translucent polygon is rendered first, and an opaque behind it is rendered later. However, if we make the z-buffer read-only when rendering translucent polygons, we can prevent the depth information from being updated when rendering translucent objects. In other words, if the depth information allows a pixel to be rendered, it is blended (composited) with the pixel already stored there. If the pixel is part of an opaque polygon, the depth data is updated, but if it is a translucent pixel, the depth data is not updated.

One of the major uses of the α channel is for antialiasing. When rendering a line, instead of colouring an entire pixel with the colour of the line if it passes through it, the amount of contribution of the line to the pixel is stored in the pixels alpha value. This value is then used to calculate the intensity of the colour (specified by the RGB values), and avoids the sharp contrasts and steps of aliasing.

Rather than antialiasing individual lines and polygons, we can antialias the entire scene using a technique called multisampling. In this mode, every pixel in the frame buffer contains a number of samples. Each sample is capable of storing a colour, depth, and other values. When a scene is rendered, it is as if the scene is rendered at an enhanced resolution. However, when the image must be displayed in the frame buffer, all of the samples for each pixel are combined to produce the final pixel colour.