Getting Started

These notes are intended for an introductory course in computer graphics with a few features that are not found in most beginning courses:

- The focus is on computer graphics programming with the OpenGL graphics API, and many of the algorithms and techniques that are used in computer graphics are covered only at the level they are needed to understand questions of graphics programming. This differs from most computer graphics textbooks that place a great deal of emphasis on understanding these algorithms and techniques. We recognize the importance of these for persons who want to develop a deep knowledge of the subject and suggest that a second graphics course built on the ideas of these notes can provide that knowledge. Moreover, we believe that students who become used to working with these concepts at a programming level will be equipped to work with these algorithms and techniques more fluently than students who meet them with no previous background.

- We focus on 3D graphics to the almost complete exclusion of 2D techniques. It has been traditional to start with 2D graphics and move up to 3D because some of the algorithms and techniques have been easier to grasp at the 2D level, but without that concern it seems easier simply to start with 3D and discuss 2D as a special case.

- Because we focus on graphics programming rather than algorithms and techniques, we have fewer instances of data structures and other computer science techniques. This means that these notes can be used for a computer graphics course that can be taken earlier in a student’s computer science studies than the traditional graphics course. Our basic premise is that this course should be quite accessible to a student with a sound background in programming a sequential imperative language, particularly C.

- These notes include an emphasis on the scene graph as a fundamental tool in organizing the modeling needed to create a graphics scene. The concept of scene graph allows the student to design the transformations, geometry, and appearance of a number of complex components in a way that they can be implemented quite readily in code, even if the graphics API itself does not support the scene graph directly. This is particularly important for hierarchical modeling, but it provides a unified design approach to modeling and has some very useful applications for placing the eye point in the scene and for managing motion and animation.

- These notes include an emphasis on visual communication and interaction through computer graphics that is usually missing from textbooks, though we expect that most instructors include this somehow in their courses. We believe that a systematic discussion of this subject will help prepare students for more effective use of computer graphics in their future professional lives, whether this is in technical areas in computing or is in areas where there are significant applications of computer graphics.

- Many, if not most, of the examples in these notes are taken from sources in the sciences, and they include two chapters on scientific and mathematical applications of computer graphics. This makes the notes useable for courses that include science students as well as making graphics students aware of the breadth of areas in the sciences where graphics can be used.

This set of emphases makes these notes appropriate for courses in computer science programs that want to develop ties with other programs on campus, particularly programs that want to provide science students with a background that will support development of computational science or scientific visualization work.

What is a graphics API?
The short answer is than an API is an Application Programming Interface — a set of tools that allow a programmer to work in an application area. Thus a graphics API is a set of tools that allow a programmer to write applications that use computer graphics. These materials are intended to introduce you to the OpenGL graphics API and to give you a number of examples that will help you understand the capabilities that OpenGL provides and will allow you to learn how to integrate graphics programming into your other work.
Overview of these notes

In these notes we describe some general principles in computer graphics, emphasizing 3D graphics and interactive graphical techniques, and show how OpenGL provides the graphics programming tools that implement these principles. We do not spend time describing in depth the way the techniques are implemented or the algorithms behind the techniques; these will be provided by the lectures if the instructor believes it necessary. Instead, we focus on giving some concepts behind the graphics and on using a graphics API (application programming interface) to carry out graphics operations and create images.

These notes will give beginning computer graphics students a good introduction to the range of functionality available in a modern computer graphics API. They are based on the OpenGL API, but we have organized the general outline so that they could be adapted to fit another API as these are developed.

The key concept in these notes, and in the computer graphics programming course, is the use of computer graphics to communicate information to an audience. We usually assume that the information under discussion comes from the sciences, and include a significant amount of material on models in the sciences and how they can be presented visually through computer graphics. It is tempting to use the word “visualization” somewhere in the title of this document, but we would reserve that word for material that is fully focused on the science with only a sidelight on the graphics; because we reverse that emphasis, the role of visualization is in the application of the graphics.

We have tried to match the sequence of these modules to the sequence we would expect to be used in an introductory course, and in some cases, the presentation of one module will depend on the student knowing the content of an earlier module. However, in other cases it will not be critical that earlier modules have been covered. It should be pretty obvious if other modules are assumed, and we may make that assumption explicit in some modules.

What is Computer Graphics?

We view computer graphics as the art and science of creating synthetic images by programming the geometry and appearance of the contents of the images, and by displaying the results of that programming on appropriate display devices that support graphical output. The programming may be done (and in these notes, is assumed to be done) with the support of a graphics API that does most of the detailed work of rendering the scene that the programming defines.

The work of the programmer is to develop representations for the geometric entities that are to make up the images, to assemble these entities into an appropriate geometric space where they can have the proper relationships with each other as needed for the image, to define and present the look of each of the entities as part of that scene, to specify how the scene is to be viewed, and to specify how the scene as viewed is to be displayed on the graphic device. These processes are supported by the 3D graphics pipeline, as described below, which will be one of our primary tools in understanding how graphics processes work.

In addition to the work mentioned so far, there are two other important parts of the task for the programmer. Because a static image does not present as much information as a moving image, the programmer may want to design some motion into the scene, that is, may want to define some animation for the image. And because a user may want to have the opportunity to control the nature of the image or the way the image is seen, the programmer may want to design ways for the user to interact with the scene as it is presented.
All of these topics will be covered in the notes, using the OpenGL graphics API as the basis for implementing the actual graphics programming.

*The 3D Graphics Pipeline*

The 3D computer graphics pipeline is simply a process for converting coordinates from what is most convenient for the application programmer into what is most convenient for the display hardware. We will explore the details of the steps for the pipeline in the chapters below, but here we outline the pipeline to help you understand how it operates. The pipeline is diagrammed in Figure 0.9, and we will start to sketch the various stages in the pipeline here, with more detail given in subsequent chapters.

![Figure 0.9: The graphics pipeline’s stages and mappings](image)

**3D model coordinate systems**

The application programmer starts by defining a particular object about a local origin, somewhere in or around the object. This is what would naturally happen if the object was exported from a CAD system or was defined by a mathematical function. Modeling something about its local origin involves defining it in terms of *model coordinates*, a coordinate system that is used specifically to define a particular graphical object. Note that the modeling coordinate system may be different for every part of a scene. If the object uses its own coordinates as it is defined, it must be placed in the 3D world space by using appropriate transformations.

Transformations are functions that move objects while preserving their geometric properties. The transformations that are available to us in a graphics system are rotations, translations, and scaling. Rotations hold the origin of a coordinate system fixed and move all the other points by a fixed angle around the origin, translations add a fixed value to each of the coordinates of each point in a scene, and scaling multiplies each coordinate of a point by a fixed value. These will be discussed in much more detail in the chapter on modeling below.
3D world coordinate system

After a graphics object is defined in its own modeling coordinate system, the object is transformed to where it belongs in the scene. This is called the model transformation, and the single coordinate system that describes the position of every object in the scene is called the world coordinate system. In practice, graphics programmers use a relatively small set of simple, built-in transformations and build up the model transformations through a sequence of these simple transformations. Because each transformation works on the geometry it sees, we see the effect of the associative law for functions; in a piece of code represented by metacode such as

```c
transformOne(...);
transformTwo(...);
transformThree(...);
geometry(...);
```

we see that transformThree is applied to the original geometry, transformTwo to the results of that transformation, and transformOne to the results of the second transformation. Letting $t_1$, $t_2$, and $t_3$ be the three transformations, respectively, we see by the application of the associative law for function application that

$$t_1(t_2(t_3(\text{geometry}))) = (t_1*t_2*t_3)(\text{geometry})$$

This shows us that in a product of transformations, applied by multiplying on the left, the transformation nearest the geometry is applied first, and that this principle extends across multiple transformations. This will be very important in the overall understanding of the overall order in which we operate on scenes, as we describe at the end of this section.

The model transformation for an object in a scene can change over time to create motion in a scene. For example, in a rigid-body animation, an object can be moved through the scene just by changing its model transformation between frames. This change can be made through standard built-in facilities in most graphics APIs, including OpenGL; we will discuss how this is done later.

3D eye coordinate system

Once the 3D world has been created, an application programmer would like the freedom to be able to view it from any location. But graphics viewing models typically require a specific orientation and/or position for the eye at this stage. For example, the system might require that the eye position be at the origin, looking in –Z (or sometimes +Z). So the next step in the pipeline is the viewing transformation, in which the coordinate system for the scene is changed to satisfy this requirement. The result is the 3D eye coordinate system. One can think of this process as grabbing the arbitrary eye location and all the 3D world objects and sliding them around together so that the eye ends up at the proper place and looking in the proper direction. The relative positions between the eye and the other objects have not been changed; all the parts of the scene are simply anchored in a different spot in 3D space. This is just a transformation, although it can be asked for in a variety of ways depending on the graphics API. Because the viewing transformation transforms the entire world space in order to move the eye to the standard position and orientation, we can consider the viewing transformation to be the inverse of whatever transformation placed the eye point in the position and orientation defined for the view. We will take advantage of this observation in the modeling chapter when we consider how to place the eye in the scene’s geometry.

At this point, we are ready to clip the object against the 3D viewing volume. The viewing volume is the 3D volume that is determined by the projection to be used (see below) and that declares what portion of the 3D universe the viewer wants to be able to see. This happens by defining how for the scene should be visible to the left, right, bottom, top, near, and far. Any portions of the scene that are outside the defined viewing volume are clipped and discarded. All portions that are inside are retained and passed along to the projection step. In Figure 0.10, note how the front of the
The 3D eye coordinate system still must be converted into a 2D coordinate system before it can be placed on a graphic device, so the next stage of the pipeline performs this operation, called a projection. Before the actual projection is done, we must think about what we will actually see in the graphic device. Imagine your eye placed somewhere in the scene, looking in a particular direction. You do not see the entire scene; you only see what lies in front of your eye and within your field of view. This space is called the viewing volume for your scene, and it includes a bit more than the eye point, direction, and field of view; it also includes a front plane, with the concept that you cannot see anything closer than this plane, and a back plane, with the concept that you cannot see anything farther than that plane.

There are two kinds of projections commonly used in computer graphics. One maps all the points in the eye space to the viewing plane by simply ignoring the value of the z-coordinate, and as a result all points on a line parallel to the direction of the eye are mapped to the same point on the viewing plane. Such a projection is called a parallel projection. The other projection acts as if the eye were a single point and each point in the scene is mapped, along a line from the eye to that point, to a point on a plane in front of the eye, which is the classical technique of artists when drawing with perspective. Such a projection is called a perspective projection. And just as there are parallel and perspective projections, there are parallel (also called orthographic) and perspective viewing volumes. In a parallel projection, objects stay the same size as they get farther away. In a perspective projection, objects get smaller as they get farther away. Perspective projections tend to look more realistic, while parallel projections tend to make objects easier to line up. Each projection will display the geometry within the region of 3-space that is bounded by the right, left, top, bottom, back, and front planes described above. The region that is visible with each projection is often called its view volume. As seen in Figure 0.11 below, the viewing volume of a parallel projection is a rectangular region (here shown as a solid), while the viewing volume of a perspective projection has the shape of a pyramid that is truncated at the top. This kind of shape is sometimes called a frustum (also shown here as a solid).
Figure 0.11: Parallel and Perspective Viewing Volumes, with Eyeballs

Figure 0.12 presents a scene with both parallel and perspective projections; in this example, you will have to look carefully to see the differences!

Figure 0.12: the same scene as presented by a parallel projection (left) and by a perspective projection (right)

2D screen coordinates

The final step in the pipeline is to change units so that the object is in a coordinate system appropriate for the display device. Because the screen is a digital device, this requires that the real numbers in the 2D eye coordinate system be converted to integer numbers that represent screen coordinate. This is done with a proportional mapping followed by a truncation of the coordinate values. It is called the window-to-viewport mapping, and the new coordinate space is referred to as screen coordinates, or display coordinates. When this step is done, the entire scene is now represented by integer screen coordinates and can be drawn on the 2D display device.

Note that this entire pipeline process converts vertices, or geometry, from one form to another by means of several different transformations. These transformations ensure that the vertex geometry of the scene is consistent among the different representations as the scene is developed, but
computer graphics also assumes that the topology of the scene stays the same. For instance, if two points are connected by a line in 3D model space, then those converted points are assumed to likewise be connected by a line in 2D screen space. Thus the geometric relationships (points, lines, polygons, ...) that were specified in the original model space are all maintained until we get to screen space, and are only actually implemented there.

**Overall viewing process**

Let’s look at the overall operations on the geometry you define for a scene as the graphics system works on that scene and eventually displays it to your user. Referring again to Figure 0.8 and omitting the clipping and window-to-viewport process, we see that we start with geometry, apply the modeling transformation(s), apply the viewing transformation, and apply the projection to the screen. This can be expressed in terms of function composition as the sequence

$$\text{projection(viewing(transformation(geometry)))}$$

or, as we noted above with the associative law for functions and writing function composition as multiplication,

$$(\text{projection} * \text{viewing} * \text{transformation})(\text{geometry}).$$

In the same way we saw that the operations nearest the geometry were performed before operations further from the geometry, then, we will want to define the projection first, the viewing next, and the transformations last before we define the geometry they are to operate on. We will see this sequence as a key factor in the way we structure a scene through the scene graph in the modeling chapter later in these notes.

**Different implementation, same result**

*Warning!* This discussion has shown the concept of how a vertex travels through the graphics pipeline. There are several ways of implementing this travel, any of which will produce a correct display. Do not be disturbed if you find out a graphics system does not manage the overall graphics pipeline process exactly as shown here. The basic principles and stages of the operation are still the same.

For example, OpenGL combines the modeling and viewing transformations into a single transformation known as the *modelview matrix*. This will force us to take a little different approach to the modeling and viewing process that integrates these two steps. Also, graphics hardware systems typically perform a window-to-normalized-coordinates operation prior to clipping so that hardware can be optimized around a particular coordinate system. In this case, everything else stays the same except that the final step would be normalized-coordinate-to-viewport mapping.

In many cases, we simply will not be concerned about the details of how the stages are carried out. Our goal will be to represent the geometry correctly at the modeling and world coordinate stages, to specify the eye position appropriately so the transformation to eye coordinates will be correct, and to define our window and projections correctly so the transformations down to 2D and to screen space will be correct. Other details will be left to a more advanced graphics course.

**Summary of viewing advantages**

One of the classic questions beginners have about viewing a computer graphics image is whether to use perspective or orthographic projections. Each of these has its strengths and its weaknesses. As a quick guide to start with, here are some thoughts on the two approaches:

*Orthographic* projections are at their best when:

- Items in the scene need to be checked to see if they line up or are the same size
- Lines need to be checked to see if they are parallel
• We do not care that distance is handled unrealistically
• We are not trying to move through the scene

*Perspective* projections are at their best when:
• Realism counts
• We want to move through the scene and have a view like a human viewer would have
• We do not care that it is difficult to measure or align things

In fact, when you have some experience with each, and when you know the expectations of the audience for which you’re preparing your images, you will find that the choice is quite natural and will have no problem knowing which is better for a given image.

*A basic OpenGL program*

Our example programs that use OpenGL have some strong similarities. Each is based on the GLUT utility toolkit that usually accompanies OpenGL systems, so all the sample codes have this fundamental similarity. (If your version of OpenGL does not include GLUT, its source code is available online; check the page at http://www.reality.sgi.com/opengl/glut3/glut3.h and you can find out where to get it. You will need to download the code, compile it, and install it in your system.) Similarly, when we get to the section on event handling, we will use the MUI (micro user interface) toolkit, although this is not yet developed or included in this first draft release.

Like most worthwhile APIs, OpenGL is complex and offers you many different ways to express a solution to a graphical problem in code. Our examples use a rather limited approach that works well for interactive programs, because we believe strongly that graphics and interaction should be learned together. When you want to focus on making highly realistic graphics, of the sort that takes a long time to create a single image, then you can readily give up the notion of interactive work.

So what is the typical structure of a program that would use OpenGL to make interactive images? We will display this example in C, as we will with all our examples in these notes. OpenGL is not really compatible with the concept of object-oriented programming because it maintains an extensive set of state information that cannot be encapsulated in graphics classes. Indeed, as you will see when you look at the example programs, many functions such as event callbacks cannot even deal with parameters and must work with global variables, so the usual practice is to create a global application environment through global variables and use these variables instead of parameters to pass information in and out of functions. (Typically, OpenGL programs use side effects — passing information through external variables instead of through parameters — because graphics environments are complex and parameter lists can become unmanageable.) So the skeleton of a typical GLUT-based OpenGL program would look something like this:

```c
// include section
#include <GL/glut.h> // alternately "glut.h" for Macintosh
// other includes as needed

// typedef section
// as needed

// global data section
// as needed

// function template section
void doMyInit(void);
```
void display(void);
void reshape(int,int);
void idle(void);
// others as defined

// initialization function
void doMyInit(void) {
    set up basic OpenGL parameters and environment
    set up projection transformation (ortho or perspective)
}

// reshape function
void reshape(int w, int h) { 
    set up projection transformation with new window 
    dimensions w and h
    post redisplay
}

// display function
void display(void){
    set up viewing transformation as described in later chapters
    define whatever transformations, appearance, and geometry you need
    post redisplay
}

// idle function
void idle(void) {
    update anything that changes from one step of the program to another
    post redisplay
}

// other graphics and application functions
// as needed
// main function -- set up the system and then turn it over to events
void main(int argc, char** argv) {
    // initialize system through GLUT and your own initialization
    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(windW,windH);
    glutInitWindowPosition(topLeftX,topLeftY);
    glutCreateWindow("A Sample Program");
    doMyInit();

    // define callbacks for events
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);

    // go into main event loop
    glutMainLoop();
}

The viewing transformation is specified in OpenGL with the gluLookAt() call:

    gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz);

The parameters for this transformation include the coordinates of eye position (ex, ey, ez), the coordinates of the point at which the eye is looking (lx, ly, lz), and the coordinates of a
vector that defines the “up” direction for the view \((ux, uy, uz)\). This would most often be called from the display() function above and is discussed in more detail in the chapter below on viewing.

Projections are specified fairly easily in the OpenGL system. An orthographic (or parallel) projection is defined with the function call:

\[
glOrtho( \text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far} );
\]

where \text{left} and \text{right} are the x-coordinates of the left and right sides of the orthographic view volume, \text{bottom} and \text{top} are the y-coordinates of the bottom and top of the view volume, and \text{near} and \text{far} are the z-coordinates of the front and back of the view volume. A perspective projection is defined with the function call:

\[
glFrustum( \text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far} );
\]
or:

\[
\text{gluPerspective( fovy, aspect, near, far );}
\]

In the \text{glFrustum(...)} call, the values \text{left}, \text{right}, \text{bottom}, and \text{top} are the coordinates of the left, right, bottom, and top clipping planes as they intersect the near plane; the other coordinate of all these four clipping planes is the eye point. In the \text{gluPerspective(...)} call, the first parameter is the field of view in degrees, the second is the aspect ratio for the window, and the near and far parameters are as above. In this projection, it is assumed that your eye is at the origin so there is no need to specify the other four clipping planes; they are determined by the field of view and the aspect ratio.

In OpenGL, the modeling transformation and viewing transformation are merged into a single modelview transformation, which we will discuss in much more detail in the modeling chapter below. This means that we cannot manage the viewing transformation separately from the rest of the transformations we must use to do the detailed modeling of our scene.

There are some specific things about this code that we need to mention here and that we will explain in much more detail later, such as callbacks and events. But for now, we can simply view the main event loop as passing control at the appropriate time to the following functions specified in the main function:

\[
\begin{align*}
\text{void doMyInit(void)} \\
\text{void display(void)} \\
\text{void reshape(int,int)} \\
\text{void idle(void)}
\end{align*}
\]

The task of the function \text{doMyInit()} is to set up the environment for the program so that the scene’s fundamental environment is set up. This is a good place to compute values for arrays that define the geometry, to define specific named colors, and the like. At the end of this function you should set up the initial projection specifications.

The task of the function \text{display()} is to do everything needed to create the image. This can involve manipulating a significant amount of data, but the function does not allow any parameters. Here is the first place where the data for graphics problems must be managed through global variables. As we noted above, we treat the global data as a programmer-created environment, with some functions manipulating the data and the graphical functions using that data (the graphics environment) to define and present the display. In most cases, the global data is changed only through well-documented side effects, so this use of the data is reasonably clean. (Note that this argues strongly for a great deal of emphasis on documentation in your projects, which most people believe is not a bad thing.) Of course, some functions can create or receive control parameters, and it is up to you to decide whether these parameters should be managed globally or locally, but even in this case the declarations are likely to be global because of the wide number of functions that
may use them. You will also find that your graphics API maintains its own environment, called its system state, and that some of your functions will also manipulate that environment, so it is important to consider the overall environment effect of your work.

The task of the function `reshape(int, int)` is to respond to user manipulation of the window in which the graphics are displayed. The two parameters are the width and height of the window in screen space (or in pixels) as it is resized by the user’s manipulation, and should be used to reset the projection information for the scene. GLUT interacts with the window manager of the system and allows a window to be moved or resized very flexibly without the programmer having to manage any system-dependent operations directly. Surely this kind of system independence is one of the very good reasons to use the GLUT toolkit!

The task of the function `idle()` is to respond to the “idle” event — the event that nothing has happened. This function defines what the program is to do without any user activity, and is the way we can get animation in our programs. Without going into detail that should wait for our general discussion of events, the process is that the `idle()` function makes any desired changes in the global environment, and then requests that the program make a new display (with these changes) by invoking the function `glutPostRedisplay()` that simply requests the display function when the system can next do it by posting a “redisplay” event to the system.

The execution sequence of a simple program with no other events would then look something like is shown in Figure 0.13. Note that `main()` does not call the `display()` function directly; instead `main()` calls the event handling function `glutMainLoop()` which in turn makes the first call to `display()` and then waits for events to be posted to the system event queue. We will describe event handling in more detail in a later chapter.

![Figure 0.13: the event loop for the idle event](image)

So we see that in the absence of any other event activity, the program will continues to apply the activity of the `idle()` function as time progresses, leading to an image that changes over time — that is, to an animated image.

Now that we have an idea of the graphics pipeline and know what a program can look like, we can move on to discuss how we specify the viewing and projection environment, how we define the fundamental geometry for our image, and how we create the image in the `display()` function with the environment that we define through the viewing and projection.

**Mathematics background needed for graphics**

The primary mathematical background needed for computer graphics programming is 3D analytic geometry. It is unusual to see a course with this title, however, so most students pick up bits and pieces of mathematics background that fill this in. One of the common sources of the background is introductory physics; another is multivariate calculus. Neither of these is a common requirement for computer graphics, however, so here we will outline the general concepts we will use in these notes.
Coordinate systems and points

The set of real numbers — often thought of as the set of all possible distances — is a mathematical abstraction that is effectively modeled as a Euclidean straight line with two uniquely-identified points. One point is identified with the number 0.0 (we write all real numbers with decimals, to meet the expectations of programming languages), called the origin, and the other is identified with the number 1.0, which we call the unit point. The direction of the line from 0.0 to 1.0 is called the positive direction; the opposite direction of the line is called the negative direction. These directions identify the parts of the lines associated with positive and negative numbers, respectively.

In this model, any real number is identified with the unique point on the line that is

- at the distance from the origin which is that number times the distance from 0.0 to 1.0, and
- in the direction of the number’s sign.

We have heard that a line is determined by two points; let’s see how that can work. Let the first point be $P_0 = (X_0, Y_0, Z_0)$ and the second point be $P_1 = (X_1, Y_1, Z_1)$. Let’s call $P_0$ the origin and $P_1$ the unit point. Points on the segment are obtained by starting at the “first” point $P_0$ offset by a fraction of the difference vector $P_1 - P_0$. Then any point $P = (X, Y, Z)$ on the line can be expressed in vector terms by

$$P = P_0 + t\cdot(P_1 - P_0) = (1-t)\cdot P_0 + t\cdot P_1$$

for a single value of a real variable $t$. This computation is actually done on a per-coordinate basis, with one equation each for $X$, $Y$, and $Z$ as follows:

$$X = X_0 + t\cdot(X_1 - X_0) = (1-t)\cdot X_0 + t\cdot X_1$$
$$Y = Y_0 + t\cdot(Y_1 - Y_0) = (1-t)\cdot Y_0 + t\cdot Y_1$$
$$Z = Z_0 + t\cdot(Z_1 - Z_0) = (1-t)\cdot Z_0 + t\cdot Z_1$$

Thus any line segment can be determined by a single parameter, and so is called a 1–dimensional object. This is illustrated in Figure 0.1 below that represents a way to calculate the coordinates of the points along a line segment by incrementing the value of $t$ from 0 to 1.

![Figure 0.1: a parametric line segment with some values of the parameter](image)

This representation for a line segment (or an entire line, if you place no restrictions on the value of $t$) also allows you to compute intersections involving lines. The reverse concept is also useful, so if you have a known point on the line, you can calculate the value of the parameter $t$ that would produce that point. For example, if a line intersects an object at a point $Q$, a vector calculation of the form $P_0 + t\cdot(P_1 - P_0) = Q$ would allow you to find the parameter $t$ that gives the intersection point on the line. This calculation is not usually applied for points, however, because the calculation for any of the single variables $X$, $Y$, or $Z$ would yield the same value of $t$. This is often the basis for geometric computations such as the intersection of a line and a plane.

If we have two straight lines that are perpendicular to each other and meet in a point, we can define that point to be the origin for both lines, and choose two points the same distance from the origin on each line as the unit points. A distance unit is defined to be used by each of the two lines, and the points at this distance from the intersection point are marked, one to the right of the intersection and one above it. This gives us the classical 2D coordinate system, often called the Cartesian...
coordinate system. The vectors from the intersection point to the right-hand point (resp. the point above the intersection) are called the X- and Y-direction vectors and are indicated by $\mathbf{i}$ and $\mathbf{j}$ respectively. Points in this system are represented by an ordered pair of real numbers, $(X, Y)$, and this is probably the most familiar coordinate system to most people. These points may also be represented by a vector $<X, Y>$ from the origin to the point, and this vector may be expressed in terms of the direction vectors as $X\mathbf{i} + Y\mathbf{j}$.

In 2D Cartesian coordinates, any two lines that are not parallel will meet in a point. The lines make four angles when they meet, and the acute angle is called the angle between the lines. If two line segments begin at the same point, they make a single angle that is called the angle between the line segments. These angles are measured with the usual trigonometric functions, and we assume that the reader will have a modest familiarity with trigonometry. Some of the reasons for this assumption can be found in the discussions below on polar and spherical coordinates, and in the description of the dot product and cross product. We will discuss more about the trigonometric aspects of graphics when we get to that point in modeling or lighting.

The 3D world in which we will do most of our computer graphics work is based on 3D Cartesian coordinates that extend the ideas of 2D coordinates above. This is usually presented in terms of three lines that meet at a single point, which is identified as the origin for all three lines and is called the origin, that have their unit points the same distance from that point, and that are mutually perpendicular. Each point represented by an ordered triple of real numbers $(x, y, z)$. The three lines correspond to three unit direction vectors, each from the origin to the unit point of its respective line; these are named $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ for the X-, Y-, and Z-axis, respectively, and are called the canonical basis for the space, and the point can be represented as $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$. Any triple of numbers is identified with the point in the space that lies an appropriate distance from the two-axis planes. This is all illustrated in Figure 0.2 below.

3D coordinate systems can be either right-handed or left-handed: the third axis can be the cross product of the first two axes, or it can be the negative of that cross product, respectively. (We will talk about cross products a little later in this chapter.) The “handed-ness” comes from a simple technique: if you hold your hand in space with your fingers along the first axis and curl your fingers towards the second axis, your thumb will point in a direction perpendicular to the first two axes. If you do this with the right hand, the thumb points in the direction of the third axis in a right-handed system. If you do it with the left hand, the thumb points in the direction of the third axis in a left-handed system.

Some computer graphics systems use right-handed coordinates, and this is probably the most natural coordinate system for most uses. For example, this is the coordinate system that naturally fits electromagnetic theory, because the relationship between a moving current in a wire and the magnetic field it generates is a right-hand coordinate relationship. The modeling in Open GL is based on a right-hand coordinate system.

![Figure 0.2: right-hand coordinate system with origin (left) and with a point identified by its coordinates; left-hand coordinate system (right)](image_url)
On the other hand, there are other places where a left-handed coordinate system is natural. If you consider a space with a standard X-Y plane as the front of the space and define Z as the distance back from that plane, then the values of Z naturally increase as you move back into the space. This is a left-hand relationship.

**Line segments and curves**

In standard Euclidean geometry, two points determine a line as we noted above. In fact, in the same way we talked about any line having unique origin identified with 0.0 and unit point identified with 1.0, a line segment — the points on the line between these two particular points — can be identified as the points corresponding to values between 0 and 1. It is done by much the same process as we used to illustrate the 1-dimensional nature of a line above. That is, just as in the discussion of lines above, if the two points are \( P_0 \) and \( P_1 \), we can identify any point between them as \( P = (1-t)P_0 + tP_1 \) for a unique value of \( t \) between 0 and 1. This is called the parametric form for a line segment.

The line segment gives us an example of determining a continuous set of points by functions from the interval [0,1] to 3-space. In general, if we consider any set of functions \( x(t) \), \( y(t) \), and \( z(t) \) that are defined on [0,1] and are continuous, the set of points they generate is called a curve in 3-space. There are some very useful examples of such curves, which can display the locations of a moving point in space, the positions from which you will view a scene in a fly-through, or the behavior of a function of two variables if the values two variables lie on a curve in 2-space.

**Dot and cross products**

There are two computations that we will need to understand, and sometimes to perform, in developing the geometry for our graphic images. The first is the **dot product** of two vectors. This produces a single real value that represents the projection of one vector on the other and can be thought of as the product of the lengths of the two vectors times the cosine of the angle between them. The computation is quite simple: it is simply the sum of the componentwise products of the vectors. If the two vectors A and B are

\[
A = (X_1,Y_1,Z_1) \\
B = (X_2,Y_2,Z_2),
\]

the dot product is computed as

\[
A \cdot B = X_1X_2+Y_1Y_2+Z_1Z_2.
\]

The second computation is the **cross product** of two vectors. This produces a vector that is perpendicular to each of the original vectors and whose length is the product of the two vector lengths times the sine of the angle between them. Thus if two vectors are parallel, the cross product is zero; if they are orthogonal, the cross product has length equal to the product of the two lengths; if they are both unit vectors, the cross product is the sine of the included angle. The computation of the cross product can be expressed as the determinant of a matrix whose first row is the three standard unit vectors, whose second row is the first vector of the product, and whose third row is the second vector of the product. Denoting the unit direction vectors in the X, Y, and Z directions as \( i \), \( j \), and \( k \), we can express the cross product of two vectors \( \langle a,b,c \rangle \) and \( \langle u,v,w \rangle \) in terms of a determinant:

\[
\langle a,b,c \rangle \times \langle u,v,w \rangle = \text{det} \begin{vmatrix} i & j & k \\ a & b & c \\ u & v & w \end{vmatrix}
\]
The cross product has a “handedness” property, because if you align the fingers of your right hand with the direction of the first vector and curl your fingers towards the second vector, your right thumb will point in the direction of the cross product. Thus the order of the vectors is important; if you reverse the order, you reverse the sign of the product (recall that interchanging the second and third rows of the determinant above will change its sign), so the cross product operation is not commutative. As a simple example, with \( \mathbf{i}, \mathbf{j}, \) and \( \mathbf{k} \) as above, we see that \( \mathbf{i} \times \mathbf{j} = \mathbf{k} \) but that \( \mathbf{j} \times \mathbf{i} = -\mathbf{k} \). In general, as you can see in Figure 0.3, if you think of the three direction vectors as being wrapped around as if they were visible from the first octant of 3-space, the product of any two is the third direction vector if the letters are in counterclockwise order, and the negative of the third if the order is clockwise. Note also that the cross product of two collinear vectors (one of the vectors is a constant multiple of the other) will always be zero, so the geometric interpretation of the cross product does not apply in this case.

![Figure 0.3: the direction vectors in order](image)

The cross product can be very useful when you need to define a vector perpendicular to two given vectors; the most common application of this is defining a normal vector to a polygon by computing the cross product of two edge vectors. For a triangle as shown in Figure 0.4 below with vertices \( A, B, \) and \( C \) in order counterclockwise from the “front” side of the triangle, this can be computed by creating two difference vectors \( P = C - B \) and \( Q = A - C \), and computing the cross product as \( P \times P \) to yield a vector \( N \) normal to the plane of the triangle. As we shall see in the next section, this normal vector, and any point on the triangle, allow us to generate the equation of the plane that contains the triangle. When we need to use this normal for lighting, we will need to normalize it, or make it a unit vector, but that can easily be done by calculating its length and dividing each component by that length.

![Figure 0.4: the normal to a triangle as the cross product of two edges](image)

Planes and half-spaces

We saw above that a line could be defined in terms of a single parameter, so it is often called a one-dimensional space. A plane, on the other hand, is a two-dimensional space, determined by two
parameters. If we have any two non-parallel lines that meet in a single point, we recall that they
determine a plane that can be thought of as all combinations ... where each of the two lines
contributes one of the components. In more general terms, let’s consider the vector \( \mathbf{N} = \langle A, B, C \rangle \)
defined as the cross product of the two vectors determined by the two lines. Then \( \mathbf{N} \) is
perpendicular to each of the two vectors and hence to any line in the plane. In fact, this can be
taken as defining the plane: the plane is defined by all lines through the fixed point perpendicular
to \( \mathbf{N} \). If we take a fixed point in the plane, \((U, V, W)\), and a variable point in the plane, \((x, y, z)\),
we can write the perpendicular relationship as
\[
\langle A, B, C \rangle \cdot \langle x-U, y-V, z-W \rangle = 0.
\]
When we expand this dot product we see
\[
A(x-U) + b(y-V) + C(z-W) = Ax + By + Cz + (-AU-BV-CW) = 0.
\]
This allows us to give an equation for the plane:
\[
Ax + By + Cz + D = 0.
\]
Thus the coefficients of the variables in the plane equation exactly match the components of the
vector normal to the plane — a very useful fact from time to time.

Any line divides a plane into two parts. If we know the equation of the line in the traditional form
\[
ax + by + c = 0,
\]
then we can determine whether a point lies on, above, or below the line by evaluating the function
\[
f(x, y) = ax + by + c
\]
determining whether the result is zero, positive, or negative. In a similar way, the equation for the plane as defined above does more than just identify the plane; it
allows us to determine on which side of the plane any point lies. If we create a function of three
variables from the plane equation
\[
f(x, y, z) = Ax + By + Cz + D,
\]
then the plane consists of all points where \( f(x, y, z) = 0 \). All points \((x, y, z)\) with
\( f(x, y, z) > 0 \) lie on one side of the plane, called the positive half-space for the plane, while all
points with \( f(x, y, z) < 0 \) lie on the other, called the negative half-space for the plane. We will
find that OpenGL uses the four coordinates \( A, B, C, D \) to identify a plane and uses the half-space
concept to choose displayable points when the plane is used for clipping.

Polygons and convexity

Most graphics systems, including OpenGL, are based on modeling and rendering based on
polygons and polyhedra. A **polygon** is a plane region bounded by a sequence of directed line
segments with the property that the end of one segment is the same as the start of the next segment,
and the end of the last line segment is the start of the first segment. A **polyhedron** is a region of
3–space that is bounded by a set of polygons. Because polyhedra are composed of polygons, we
will focus on modeling with polygons, and this will be a large part of the basis for the modeling
chapter below.

The reason for modeling based on polygons is that many of the fundamental algorithms of graphics
have been designed for polygon operations. In particular, many of these algorithms operate by
interpolating values across the polygon; you will see this below in depth buffering, shading, and
other areas. In order to interpolate across a polygon, the polygon must be **convex**. Informally, a
polygon is complex if it has no indentations; formally, a polygon is complex if for any two points
in the polygon (either the interior or the boundary), the line segment between them lies entirely
within the polygon.

Because a polygon or polyhedron bounds a region of space, we can talk about the interior or
exterior of the figure. In a convex polygon or polyhedron, this is straightforward because the
figure is defined by its bounding planes or lines, and we can simply determine which side of each
is “inside” the figure. For a non-convex figure this is less simple, so we look to convex figures for
a starting point and notice that if a point is inside the figure, any ray from an interior point (line
extending in only one direction from the point) must exit the figure in precisely one point, while if a point is outside the figure, if the ray hits the polygon it must both enter and exit, and so crosses the boundary of the figure in either 0 or 2 points. We extend this idea to general polygons by saying that a point is inside the polygon if a ray from the point crosses the boundary of the polygon an odd number of times, and is outside the polygon if a ray from the point crosses the boundary of the polygon an even number of times. This is illustrated in Figure 0.5. In this figure, points A, D, E, and G are outside the polygons and points B, D, and F are inside. Note carefully the case of point G; our definition of inside and outside might not be intuitive in some cases.

![Figure 0.5: Interior and exterior points of a convex polygon (left) and two general polygons (center and right)](image)

Another way to think about convexity is in terms of linear combinations of points. We can define a convex sum of points $P_0, P_1, \ldots, P_n$ as a sum $\sum c_i P_i$ where each of the coefficients $c_i$ is non-negative and the sum of the coefficients is exactly 1. If we recall the parametric definition of a line segment, $(1-t) \cdot P_0 + t \cdot P_1$, we note that this is a convex sum. So a polygon is convex if certain convex sums of points in the polygon must also lie in the polygon. However, it is straightforward to see that this can be generalized to say that all convex sums of points in a convex polygon must lie in the polygon, so this gives us an alternate definition of convex polygons that can sometimes be useful.

As we suggested above, most graphics systems, and certainly OpenGL, require that all polygons be convex in order to render them correctly. If you have a polygon that is not convex, you may always subdivide it into triangles or other convex polygons and work with them instead of the original polygon. As an alternative, OpenGL provides a facility to tessellate a polygon — divide it into convex polygons — automatically, but this is a complex operation that we do not cover in these notes.

### Corresponding points in rectangles

There are a number of places where we will want to understand the relation between points in two corresponding rectangular spaces. In the simplest case, we have the rectangle through which a scene is viewed as one space, and the rectangle on the screen where the viewed scene is presented as another. In a more complex case, we have the position on the screen where the cursor is when a mouse button is pressed, and the point that corresponds to this in the viewing space. In a third setting we have points in the world space and points in a texture space. These are all particular cases of a correspondence of points in two rectangular spaces.

In Figure 0.6 below, we consider two rectangles with boundaries and points named as shown. In this example, we assume that the lower left corner of each rectangle has the smallest values of the X and Y coordinates in the rectangle. With the names of the figures, we have the proportions

$$
X : X_{MIN} :: X_{MAX} : X_{MIN} = u : L :: R : L \\
Y : Y_{MIN} :: Y_{MAX} : Y_{MIN} = v : B :: T : B
$$

from which we can derive the equations:
\[
\frac{x - XMIN}{XMAX - XMIN} = \frac{u - L}{R - L}
\]
\[
\frac{y - YMIN}{YMAX - YMIN} = \frac{v - B}{T - B}
\]

and finally these two equations can be solved for the variables of either point in terms of the other:
\[
x = XMIN + (u - L) \times \frac{XMAX - XMIN}{R - L}
\]
\[
y = YMIN + (v - B) \times \frac{YMAX - YMIN}{T - B}
\]
or the dual equations that solve for \((u,v)\) in terms of \((x,y)\).

![Figure 0.6: correspondences between points in two rectangles](image)

In cases that involve the screen coordinates of a point in a window, there is an additional issue because the upper left, not the lower left, corner of the rectangle contains the smallest values, and the largest value of \(Y\), \(YMAX\), is at the bottom of the rectangle. In this case, however, we can make a simple change of variable as \(Y' = YMAX - Y\) and we see that using the \(Y'\) values instead of \(Y\) will put us back into the situation described in the figure. We can also see that the question of rectangles in 2D extends easily into rectangular spaces in 3D, and we leave that to the student.

**Line intersections**

There are times when we need to know whether two objects meet in order to understand the logic of a particular scene. Calculating whether there are intersections is relatively straightforward and we outline it here. The fundamental question is whether a line segment that is part of one object meets a triangle that is part of the second object.

There are two levels at which we might be able to determine whether an intersection occurs. The first is to see whether the line containing the segment can even come close enough to meet the triangle, and the second is whether the segment actually meets the triangle. The reason for this two-stage question is that most of the time there will be few segments that could even come close to intersecting, so we will ask the first question because it is fastest and will only ask the second question when the first indicates it can be useful.

To consider the question of whether a line can come close enough to meet the triangle, look at the situation outlined on the left in Figure 0.6. We first compute the incenter of the triangle and then define the bounding circle to lie in the plane of the triangle, to have that point as center, and to have as its radius the distance from that point to each vertex:

\[
\text{center} = \frac{P0 + P1 + P2}{3}
\]
\[
\text{radius} = \text{distance} \ (\text{center}, \ P0)
\]

Then we can compute the point where the line meets the plane of the triangle. Let the line segment be given by the parametric equation \(Q0 + t \times (Q1 - Q0)\); the entire line is given by considering all values of \(t\), and if you consider only values of \(t\) between 0 and 1 you get the segment. Next compute the cross product of two edges of the triangle in order and call the result \(N\). Then the equation of the plane is given by the processes described earlier in this chapter, and when the parametric equation for the line meets the plane we can solve for a single value of \(t\). If that value of \(t\) does not lie between 0 and 1 we can immediately conclude that there is no possible intersection because the line segment does not meet the plane at all. If the value does lie between 0
and 1, we calculate the point where the line segment meets the plane and compute the distance from that point to the center of the triangle. The line cannot meet the triangle unless this distance is less than the radius of the circle.

![Figure 0.7: a line and the bounding circle (left) and a line and triangle (right)](image)

Once we know that the line is close enough to have a potential intersection, we move on to look at the exact computation of whether the point where the line meets the plane is inside the triangle, as shown in the right-hand part of Figure 0.7. We note that a point on the inside of the triangle is characterized by being to the left of the oriented edge for each edge of the triangle. This is further characterized by the cross product of the edge vector and the vector from the vertex to the point; if this cross product has the same orientation as the normal vector to the triangle for each vertex, then the point is inside the triangle. If the intersection of the line segment and the triangle’s plane is $Q$, this means that we must have $N \cdot ((Q-P_0) \times (P_1-P_0)) > 0$ for the first edge, and similar relations for each subsequent edge.

**Polar, cylindrical, and spherical coordinates**

Up to this point we have emphasized Cartesian, or rectangular, coordinates for describing 2D and 3D geometry, but there are times when other kinds of coordinate systems are most useful. The coordinate systems we discuss here are based on angles, not distances, in at least one of their terms. Because OpenGL does not handle these coordinate systems directly, when you want to use them you will need to translate points between these forms and rectangular coordinates.

In 2D coordinates, we can identify any point $(X, Y)$ with the line segment from the origin to that point. This identification allows us to write the point in terms of the angle $\Theta$ the line segment makes with the positive X-axis and the distance $R$ of the point from the origin as:

$$X = R \cos(\Theta), \quad Y = R \sin(\Theta)$$

or, inversely,

$$R = \sqrt{X^2 + Y^2}, \quad \Theta = \arccos(X/R)$$

where $\Theta$ is the value that matches the signs of $X$ and $Y$.

This representation $(R, \Theta)$ is known as the *polar form* for the point, and the use of the polar form for all points is called the *polar coordinates* for 2D space.

There are two alternatives to Cartesian coordinates for 3D space. **Cylindrical coordinates** add a third linear dimension to 2D polar coordinates, giving the angle between the X-Z plane and the plane through the Z-axis and the point, along with the distance from the Z-axis and the Z-value of the point. Points in cylindrical coordinates are represented as $(R, \Theta, Z)$ with $R$ and $\Theta$ as above and with the Z-value as in rectangular coordinates. Figure 0.8 shows the structure of the 2D and 3D spaces with polar and cylindrical coordinates, respectively.
Cylindrical coordinates are a useful extension of a 2D polar coordinate model to 3D space. They not particularly common in graphics modeling, but can be very helpful when appropriate. For example, if you have a planar object that has to remain upright with respect to a vertical direction, but the object has to rotate to face the viewer in a scene as the viewer moves around, then it would be appropriate to model the object’s rotation using cylindrical coordinates. An example of such an object is a billboard, as discussed later in the chapter on high-efficiency graphics techniques.

Spherical coordinates represent 3D points in terms much like the latitude and longitude on the surface of the earth. The latitude of a point is the angle from the equator to the point, and ranges from 90° south to 90° north. The longitude of a point is the angle from the “prime meridian” to the point, where the prime meridian is determined by the half-plane that runs from the center of the earth through the Greenwich Observatory just east of London, England. The latitude and longitude valued uniquely determine any point on the surface of the earth, and any point in space can be represented relative to the earth by determining what point on the earth’s surface meets a line from the center of the earth to the point, and then identifying the point by the latitude and longitude of the point on the earth’s surface and the distance to the point from the center of the earth. Spherical coordinates are based on the same principle: given a point and a unit sphere centered at that point, with the sphere having a polar axis, determine the coordinates of a point \( P \) in space by the latitude \( \Phi \) (angle north or south from the equatorial plane) and longitude \( \Theta \) (angle from a particular half-plane through the diameter of the sphere perpendicular to the equatorial plane) of the point where the half-line from the center of the sphere, and determine the distance from the center to that point. Then the spherical coordinates of \( P \) are \( (R, \Theta, \Phi) \).

Spherical coordinates can be very useful when you want to control motion to achieve smooth changes in angles or distances around a point. They can also be useful if you have an object in space that must constantly show the same face to the viewer as the viewer moves around; again, this is another kind of billboard application and will be described later in these notes.

It is straightforward to convert spherical coordinates to 3D Cartesian coordinates. Noting the relationship between spherical and rectangular coordinates shown in Figure 0.9 below, and noting that this figure shows the Z-coordinate as the vertical axis, we see the following conversion equations from polar to rectangular coordinates.

\[
\begin{align*}
x &= R \cos(\Phi) \sin(\Theta) \\
y &= R \cos(\Phi) \cos(\Theta) \\
z &= R \sin(\Phi)
\end{align*}
\]

Converting from rectangular to spherical coordinates is not much more difficult. Again referring to Figure 0.9, we see that \( R \) is the diagonal of a rectangle and that the angles can be described in terms of the trigonometric functions based on the sides. So we have the equations

- \( R = \sqrt{x^2 + y^2 + z^2} \)
- \( \Theta = \cos^{-1}\left(\frac{z}{R}\right) \)
- \( \Phi = \cos^{-1}\left(\frac{y}{R \sin(\Theta)}\right) \)
\[
R = \sqrt{X^2 + Y^2 + Z^2} \\
\Phi = \text{Arcsin}(Z/R) \\
\Theta = \text{arctan}(X/\sqrt{X^2 + Y^2})
\]

Note that the inverse trigonometric function is the principle value for the longitude (\(\Phi\)), and the angle for the latitude (\(\Theta\)) is chosen between 0° and 360° so that the sine and cosine of \(\Theta\) match the algebraic sign (+ or −) of the X and Y coordinates.

Figure 0.9: spherical coordinates (left); the conversion from spherical to rectangular coordinates (right)

Figure 0.9 shows a sphere showing latitude and longitude lines and containing an inscribed rectangular coordinate system, as well as the figure needed to make the conversion between spherical and rectangular coordinates.

Higher dimensions?

While our perceptions and experience are limited to three dimensions, there is no such limit to the kind of information we may want to display with our graphics system. Of course, we cannot deal with these higher dimensions directly, so we will have other techniques to display higher-dimensional information. There are some techniques for developing three-dimensional information by projecting or combining higher-dimensional data, and some techniques for adding extra non-spatial information to 3D information in order to represent higher dimensions. We will discuss some ideas for higher-dimensional representations in later chapters in terms of visual communications and science applications.