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 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 there is an entire chapter on scientific and mathematical applications of computer graphics. This makes the notes usable 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.

Mathematics background needed

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 P0=(X0,Y0,Z0) and the second point be P1=(X1,Y1,Z1). Let’s call P0 the origin and P1
the unit point. Any point P=(X,Y,Z) on the line can be calculated by

\[ X = X_0 + t \times (X_1 - X_0) = (1-t) \times X_0 + t \times X_1 \]

for a single value of a real variable \( t \). The same calculation on \( Y \) or \( Z \) would also work, and would
yield the same value of \( t \). Thus in vector terms, the point \( P \) can be calculated by the vector equation
Thus any line can be determined by a single parameter, and so is called a 1-dimensional object.

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. This gives us the classical 2D coordinate system, often called the Cartesian coordinate system. 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.

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, with each point represented by an ordered triple of real numbers \((x,y,z)\). 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. The three lines correspond to three unit 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. 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 0.1 below.

Coordinate systems can be 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. 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.
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 P0 and P1, we can identify any point between them as

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

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 are A = (X1,Y1,Z1) and B = (X2,Y2,Z2), the dot product is computed as

\[ A \cdot B = X1 \cdot X2 + Y1 \cdot Y2 + Z1 \cdot Z2. \]

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 include 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:

\[ \begin{vmatrix} i & j & k \\ a & b & c \\ u & v & w \end{vmatrix} \]

Note that the cross product is not commutative; the order of the vectors is important, because 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).

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 with vertices P0, P1, and P2 in order counterclockwise from the “front” side of the triangle, this can be computed by creating two difference vectors \( V_1 = P_1 \cdot P_0 \) and \( V_2 = P_2 \cdot P_1 \), and computing the cross product as \( V_1 \times V_2 \).
to yield a vector normal to the plane of the triangle. As we shall see immediately below, this normal vector, and any point on the triangle, allow us to generate the equation of the plane that contains the triangle.

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.

Now 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.

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 0.2, we consider two rectangles with boundaries and points named as shown. In this first case, 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

\[
\begin{align*}
X : \text{XMIN} & : : \text{XMAX} : : \text{XMIN} = u : L : : R : L \\
Y : \text{YMIN} & : : \text{YMAX} : : \text{YMIN} = v : B : : T : B
\end{align*}
\]

from which we can derive the following equations:

\[
\begin{align*}
(x - \text{XMIN}) / (\text{XMAX} - \text{XMIN}) &= (u - L) / (R - L) \\
(y - \text{YMIN}) / (\text{YMAX} - \text{YMIN}) &= (v - B) / (T - B)
\end{align*}
\]

and finally the equations can be solved for the variables of either point in terms of the other:

\[
\begin{align*}
x &= \text{XMIN} + (u - L) \cdot (\text{XMAX} - \text{XMIN}) / (R - L) \\
y &= \text{YMIN} + (v - B) \cdot (\text{YMAX} - \text{YMIN}) / (T - B)
\end{align*}
\]

or the dual equations that solve for \((u,v)\) in terms of \((x,y)\).
Figure 0.2: correspondences between points in two rectangles

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 \( y' = Y_{\text{MAX}} - y \) 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 0.3. 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:

\[
\text{center} = \frac{P_0 + P_1 + P_2}{3} \\
\text{radius} = \text{distance (center, P0)}
\]

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 \( Q_0 + t*(Q_1 - Q_0) \); for \( t \) between 0 and 1 you get the segment, but the entire line is given by considering all values of \( t \). 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 above, 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. If the value does lie between 0 and 1, we calculate the point and compute the distance to the center of the triangle. The line can only have a chance to meet the triangle if this distance is less than the radius of the circle.
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. 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 point 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.

**Interpolations**

When we talked about the parametric form for a line segment above, and created a correspondence between the unit line segment and an arbitrary line segment, we were really interpolating between them. This can be seen much more generally in terms of corresponding any one line segment with another. If we have two line segments with endpoints \( P_0 \) and \( P_1 \) for the first and \( Q_0 \) and \( Q_1 \) for the second, we may find a correspondence between points \( P \) and \( Q \) in the two segments by solving the similarity equation

\[
\frac{P - P_0}{P_1 - P_0} = \frac{Q - Q_0}{Q_1 - Q_0}
\]

for either \( P \) or \( Q \) as you wish. In a similar way, we may create correspondences between rectangular regions in 2-space and in 3-space if we consider the points \( P_0 \) and \( P_1 \) (respectively \( Q_0 \) and \( Q_1 \)) to be the lower left and upper right corners of the two spaces, respectively. These equations then become vector equations and can be solved componentwise to get the coordinates of the points \( P \) or \( Q \) in the spaces.

In general, however, we probably want interpolation to mean determining a curve made up of intermediate points that approximate the space between a set of points in the order the points are given. Interpolating a pair of points \( P_0 \) and \( P_1 \) is simply creating the line segment between them, and we know that this can be expressed in terms of the parametric form of the segment:

\[
(1-t)*P_0 + t*P_1, \text{ for } t \text{ in } [0,1.]
\]

Finding a way to interpolate three points \( P_0 \), \( P_1 \), and \( P_2 \) is more interesting, because one can imagine many ways to do this. However, extending the concept of the parametric line we could consider a quadratic interpolation in \( t \) as:

\[
(1-t)^2*P_0 + t*(1-t)*P_1 + t^2*P_2, \text{ for } t \text{ in } [0,1.]
\]

This gives a smooth quadratic function in \( t \) that has value \( P_0 \) if \( t=0 \) and value \( P_1 \) if \( t=1 \), and that is the average of the three points if \( t=.5 \).

This use of the simple polynomials \( (1-t) \) and \( t \) is suggestive of a general approach in which we would use components which are products of these polynomials and take their coefficients from
the geometry we want to interpolate. If we follow this pattern, interpolating four points \( \text{P0}, \text{P1}, \text{P2}, \text{P3} \) would look like:

\[
(1-t)^3 \cdot \text{P0} + t(1-t)^2 \cdot \text{P1} + t^2(1-t) \cdot \text{P2} + t^3 \cdot \text{P3}, \quad \text{for } t \in [0,1].
\]

and in fact, this is an expression of the standard Bézier spline function to interpolate four control points, and the polynomials \((1-t)^3, \ t(1-t)^2, \ t^2(1-t), \ t^3\) are called the Bernstein basis for the spline. We will not carry this idea further, but it can sometimes be a handy tool even though OpenGL provides evaluators that can make spline computations easier and more efficient. Note that if the points we are interpolating lie in 3D space, each of these techniques does provide a 3D curve, that is, a function from a line segment to 3D space.

**Polar, cylindrical, and spherical coordinates**

Above we emphasized Cartesian (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 them directly, when you want to use them you will need to translate them to 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:

\[
X = R \cdot \cos(\Theta), \quad Y = R \cdot \sin(\Theta) \quad \text{or, inversely,}
\]

\[
R = \sqrt{X^2 + Y^2}, \quad \Theta = \arccos(X/R) \quad \text{where } \Theta \text{ is the value that matches the signs of } X \text{ and } Y
\]

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. This is not often used, but would make a useful extension of a 2D polar model to 3D space. Figure 0.4 shows the structure of the 2D and 3D spaces with polar and cylindrical coordinates, respectively.

![Figure 0.4: polar coordinates (left) and cylindrical coordinates (right)](image)

Spherical coordinates represent 3D points in terms much like the latitude and longitude on the surface of the earth. 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 a point and a unit sphere centered at that point, with the sphere having latitude (angle north or south from the...
equatorial plane) and longitude (angle from a particular half-plane through the diameter of the sphere perpendicular to the equatorial plane). For any point \( P \) in the space, the line segment from the point to the center of the sphere is determined by a length \( R \) of the segment, the latitude \( \Phi \) of the point where the segment (possibly extended through \( P \) if the segment is too short) meets the sphere, and the longitude \( \Theta \) of the point where the segment meets the sphere. It is straightforward to convert spherical coordinates to 3D Cartesian coordinates. Noting the relationship between spherical and rectangular coordinates below, we see the following conversion equations from polar to rectangular coordinates. The conversion the other direction is left to the student.

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

Spherical coordinates and the conversion to rectangular coordinates are shown in Figure 0.5. These can be very useful when you want to control motion to achieve smooth changes in angles or distances around a point.

![Figure 0.5: spherical coordinates (left); the conversion from spherical to rectangular coordinates (right)](image)

**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.

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.

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, \((1-t)P_0 + tP_1\), 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 tesselate a polygon — divide it into convex polygons — automatically, but this is a complex operation that we do not cover in these notes.

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 this later in terms of visual communications and science applications.

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 modules. 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. (Yes, Virginia, the term “side effects” does apply here.) So the skeleton of a typical GLUT-based OpenGL program would look something like this:

```c
// include section
#include "glut.h" // alternately <GL/glut.h> for Windows
// other includes as needed
```
There are some things we need to mention here 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 to the following functions specified in the main function:

```c
void display(void)
void reshape(int, int)
void idle(void)
```

The task of the function `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. So 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 an environment, with some functions manipulating the data and the graphical functions using that data (that 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 an emphasis on documentation in student projects, which most of us may decide 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.

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 initial width and height of the window, but GLUT allows a window to be moved or resized very flexibly without the
programmer having to manage this directly. Surely this 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 invokes the display function again 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 0.6 below. Note that `main()` does not call the function `display()` directly; instead `main()` calls the event handling function `glutMainLoop()` which in turn calls `display()` and then waits for events to be posted to the system event queue.

![Figure 0.6: the event loop for the idle event](image)

So we see how the program continues to apply the activity of the `idle()` function as time progressed, leading to an image that would change over time — that is, to an animated image.

Now we have an idea of what a program can look like, we can move on to discuss how we take fundamental geometry and create the image in the `display()` function with the environment that we define in other functions. That is the subject of further chapters below in these notes.