Chapter 1: Introduction to Computer Graphics and Vertex-Based Rendering

This chapter begins your study of graphics by starting with a discussion of the fundamental constructs of graphics and then developing a straightforward but inefficient implementation of a 3D rendering engine.

Section 1.1: Basic Concepts in Graphics

Computer graphics is the study of software and hardware related to deciding at each step of a program what color each pixel should be. Technically any program that produces any output to the display device, or to a printer, is a graphics program, but we are most interested in applications that specifically involve producing an image, often in real-time, of a scene. The typical application involves *modeling* a scene which is then *rendered*. The model or scene is simply a computer-generated simulation of a virtual reality of some sort. Rendering is the process of computing the color of each pixel in the window that is displaying a image of the scene.

In one common scenario, physical reality provides the scene, and a camera takes the light from that scene and uses it to determine the color of each position on the photograph. If it is a camera using chemical film, each chunk of chemical on the film reacts to the light that hits it by changing in some way. If it is a digital camera, each sensor determines its color from the light hitting it.

This is not the scenario we will pursue. Instead, we will use a computer-generated, or virtual, reality to produce the scene, and each pixel in a window will be set to a color depending on the scene, as determined by the rendering computation.

So, we have two fundamental issues to deal with: how can a scene be modeled, and how can the information from the model be used to compute the color of each pixel in the grid.

The coordinates in three dimensional space of the points in the model are referred to as *world coordinates*. They are real numbers (floating point numbers in a program) and can be specified and computed with to a very high degree of precision.

The hardware has a pixel grid, with something on the order of a thousand by thousand grid of individual spots that can be lit up in some color.

> This is a common technique—right now you're reading a document that was printed with a grid of rather small black or white dots, and then duplicated by a machine that scanned the printout to produce a similar grid in its internal memory and then printed those dots. And, your brain is perceiving those dots through your eyes, each of which has a sort of grid of receptors set up to detect incoming light and send that information to the brain. Indeed, all of physical reality, at least with a somewhat simple view of physics, ultimately breaks down into tiny component particles.
>
> The pixel grid is like a sheet of graph paper, but note that the pixels are not the squares, but rather are little blobs (circles or squares, say, depending on the display device)

drawn with their centers at where the lines cross. Thus, a pixel has a mathematically exact representation given by its row and column coordinates in the pixel grid, with that exact point being in the center of the pixel. The fundamental inaccuracy in computer graphics (for example, that straight lines look staircasey) comes from the fact that a whole volume of space in world coordinates maps to a single pixel grid location.

We will do all our modeling and rendering work using floating point numbers, and only at the end of the entire process map to the rather coarse pixel grid.

We have been talking about "color," so it should be defined carefully. The eye has a lens which takes light from the scene—the part of the physical world currently in sight—and aims it onto the back of the eye, which has a large number of different kinds of receptors that are sensitive to red, green, blue light.

There are also receptors that are sensitive just to intensity, of whatever frequency, allowing for monochrome vision. Apparently the peripheral vision of a human is not in color, and dogs have monochrome vision which is more effective in low light conditions than human color vision.

Each little receptor can detect how much light of its favorite color is hitting it, and sends that information to the brain, which forms a picture.

A CRT (cathode ray tube) display device has a red gun, a green gun, and a blue gun, and all of these scan the entire pixel grid many times (say 60) a second, with each gun aiming at each pixel in succession and firing with some intensity, from 0 to 255, at that position, which then lights up with a color depending on the three color intensities it is hit with.

So, there are $256^3 = 16777216$ different colors, and that's it—in principle all any application can do is simply to set the color, in some reasonable way, of each pixel in the window each time it renders an image of a model.

This is incredibly cool—when doing graphics programming, it's like you've been given a box of crayons with 16 million-some colors that are always one pixel sharp!

Example: navigating the color cube

On the web site you will find a number of Java source files in a folder named `colorcube`. Download them, compile them, and run this program.

This program shows one way to produce a pixel grid in a Java program. It uses a `Frame`, but draws to a `Canvas` so that the $(0, 0)$ location is not hidden behind the title bar the way it is in a `Frame`. This program also shows how event handling is used to produce an interactive program. Study the code and note how this all works—the basic principle is that inputs from the user produce events which go into the operating system queue to be processed in turn. When an object registers itself as a listener for a certain type of event, its corresponding method gets called whenever an event of that type is processed. The event object includes detailed information about the event, such as the key that was pressed for a key listener.

Study the appropriate `keyPressed` method to see what keys you can press to interact with this program. Run the program, hit those keys, and admire the effect—with this program

you can move about in color space, hitting any color you wish. Each point in color space corresponds to an ordered triple, (r, g, b), where r is the intensity of red light, g is the intensity of green light, and b is the intensity of blue light. All three of these are integers in the range 0 to 255, inclusive.

In addition to the issues of how to produce a computer-generated scene, and how to render an image of that scene in the pixel grid, a key issue is how fast this can be done.

In real-time graphics applications, modeling and rendering need to happen very quickly. The prime example of this is immersive computer games. But, certainly some very important applications of computer graphics can sacrifice real-time modeling and rendering for higher quality images. For example, when making a computer-generated movie, hours of computer time might be spent in rendering each frame, which only displays for about 1/30th of a second.

In this first part of the course, we will not worry about efficiency at all. Specifically, if the choice is between an algorithm that takes more memory space and/or takes more time to execute, or one that takes more effort to implement, we will pick the one that is easier to implement. We do this not entirely out of laziness, but mostly to avoid obscuring the real issues involved in modeling and rendering.

We will develop the concepts and mathematical tools for a very simple way to model a scene, and in parallel create, in Java, a basic rendering engine. This will provide valuable insight into basic computer graphics classes, as well as making our later study of OpenGL much more effective.

Section 1.2: An Idealized Approach to Modeling and Rendering

Imagine a physical scene. It consists of a bunch of pieces of stuff, each with some physical properties that determine the color of light it sends out in various directions, given light sources in the scene. This is actually very complex physically, so at this point we will just simplify and pretend that each piece of stuff has a color. Later we'll add lighting and material properties to our modeling and rendering scheme.

So, we have a gazillion pieces of stuff in the scene. Each piece of stuff has a position in space, which can be represented in some coordinate system. We'll use a standard right-handed Cartesion coordinate system, with the x axis being a line in space, the y axis perpendicular to it, with the point where they intersect being the origin, with coordinates $(0, 0, 0)$, and the z axis passing through that point and perpendicular to both the x and the y axes.

Each axis is a number line, and if you look down the z axis with the positive end toward you, you see the positive x axis going off to the right and the positive y axis going upward.

This coordinate system is the "world" coordinates, and each piece of stuff in the scene has a location given by its x, y, and z coordinates.

Now, how does viewing work? First, the eye is at some location in this same coordinate system. Right now, your left eye is somewhere in the room. We will call this point in space the *eyepoint*.

But, that's not all there is to it, because you are "looking in some direction." So, we'll put a huge sheet of glass in the scene, which is known as the *view plane*, and say that you are looking through the view plane. This is where the image of the scene will be formed, as follows. For each piece of stuff in the scene, draw a line from the eye to it, and color the point where this line crosses the view plane the color of the piece of stuff.

This way of approaching viewing is known as perspective projection, and it is pretty much how real viewing works.

Of course, the line between the piece of stuff in the scene and the eye may not hit the view plane, in which case we don't need to do anything with it.

This approach to viewing is highly idealized, and doesn't even mention the pixel grid, but it is a good starting point for what we will actually do, so we will now develop the mathematical tools needed to do this approach, and later turn to a more practical approach.

Representing a Point in Space

We will represent a point in space as a column vector, such as $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$, where a is the x coordinate, b is the y coordinate, and c is the z coordinate of the point.

In mathematical notation, we will often represent a point P, say, by

$$P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}.$$

▷ Begin developing a Java class `SGPoint` that implements point objects. All the classes in the on-going project will have an `SG` prefix, which stands for `Simple Graphics`, but is really just to relieve us of the burden of thinking of names for things that don't match any standard Sun Java classes.

Representing a Line in Space

This is probably the most important mathematical tool in the course! They are all crucial at times, but this one comes up in lots of different situations. Given two different points in space, say P and Q, we can represent the unique line L that passes through them using a parametric representation, namely

$$L(\lambda) = P + \lambda(Q - P).$$

▷ Verify that this makes sense by computing $L(0)$ and $L(1)$.

Write down this representation for the points $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ and $\begin{bmatrix} -1 \\ 0 \\ 4 \end{bmatrix}$.

Write down the the midpoint of the two given points, and the points that are a quarter of the way from each one to the other.

We call the quantity $Q - P$ a "direction vector" of the line. You can also represent a line that passes through P with direction vector d by

$$L(\lambda) = P + \lambda d.$$

▷ You've spent a lot of time in previous math classes learning how to find the equation of a line through two points. Compare that approach (in 2D) to the parametric approach. Why was your previous work so focused on the non-parametric approach?

Points vs. Vectors

A point in space is a basic concept, and so is a vector, and they are closely related, but at times it is important to distinguish between them. A point is a location is space, while a vector represents a direction and length of movement from one point to another.

For example, even though it makes algebraic sense to form a quantity such as $P+Q$, using ordinary matrix operations, this actually makes no sense if P and Q are points. You can subtract points to get the vector between them, and you can add two vectors, and add a point and a vector, but adding two points makes no sense.

We won't worry much about this now, but later we'll see how OpenGL uses a technique known as "homogeneous coordinates" to manage this carefully.

Representing a Plane in Space

We have seen how to represent the line from the eyepoint to a piece of stuff in the scene, but our idealized rendering scheme involves setting up a view plane, so we need to know how to represent planes in space.

A plane in 3D space is defined by a single point on it, say P, and a vector N, known as a "normal vector for the plane." The plane is all points Q such that the vector from P to Q, namely $Q - P$, is perpendicular to N.

This leads to a natural digression: how can you tell if two vectors are perpendicular? We will return to this topic later, but for now, just take on faith that for any two vectors A and B in space, the so-called "dot product" of A and B is defined by

$$A^T B = ||A||\,||B||\cos(\alpha),$$

where for any vector X,

$$||X|| = \sqrt{X^T X},$$

which is known as the length or norm of X, and α is the angle between A and B.

▷ Try out this definition a little. Pick two points in space and compute the various quantities involved.

This fact tells us, among many other useful things, that the angle between two vectors A and B is 90 (or 270 degrees, depending on how you rotate) if and only if $A^T B = 0$.

So, the plane through P with normal vector N is the set of all points Q such that

$$N^T(Q - P) = 0.$$

▷ Play with this definition a lot, as follows. First, pick P and N and compute the equation of the plane. Note that the equation can be rewritten as $N^TQ = N^TP$. If we use a different point on the plane, and a different length normal vector, is the resulting represenation the same? Compare this equation of a plane to the more traditional $ax + by + cz = d$ form.

Finding a Plane Containing Three Given Points

Before, we saw how to get a representation of a line given either two points, or a point and a direction.

So far we have seen how to get a representation of a plane given a point and a normal, but it is more common to want to find the unique plane through three given points.

So, given points A, B, and C, we can first form the two vectors $B - A$ and $C - A$ that lie in the plane, and hence are both perpendicular to any normal vector for the plane. So, to find the normal vector, we can use the cross product (which we'll look at more later) operation to compute

$$N = (B - A) \otimes (C - A).$$

The cross product of the vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ into the vector $\begin{bmatrix} d \\ e \\ f \end{bmatrix}$ is defined as

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \otimes \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} bf - ce \\ cd - af \\ ae - bd \end{bmatrix}.$$

▷ Verify that this formula does produce a vector whose dot products with each of the two given ones are zero.

Try this out with two numeric vectors.

Compute the cross product of the positive x axis into the positive y axis, and verify that you get the positive z axis.

Prove that $A \otimes B = -B \otimes A$.

Putting It All Together

The preceding discussion concludes the math tools needed to do our idealized rendering. The following Test Question Type summarizes what you need to know (at a minimum!) from the preceding material

---

Test Question Type 1

Given an eyepoint E, a point in space (representing the location of a colorful piece of stuff) P, and three points A, B, and C on the view plane, compute the point where the line from E to P hits the view plane.

▷ Render, by hand, a very simple scene consisting of three points "in the room" by using the algorithm of TQT1, where points "in the room" are picked for for E, A, B, and C. Verify that the rendered scene appears reasonable. Note that the room has an x-z coordinate grid on the floor.

▷ Implement these ideas in Java, figuring out what classes are needed, and giving them all the prefix `SG`. Test the engine by a program that asks for the eyepoint and viewplane and then takes any number of input points representing points in the scene and produces the rendered points.

Use a simple view plane so that the results can be sanity-checked.

Precise Mathematical Description of Idealized Rendering

In case it is not clear from the previous work, and by way of review, here is a precise description of the idealized rendering process developed so far.

Let a view plane have a normal vector N and a point on it A. Then any point R in 3D space is on the plane if and only if $N^T(R - A) = 0$.

N may have been computed as $(B - A) \otimes (C - A)$, given three points A, B, and C known to be on the view plane.

Let E be the eye point. Consider a point P that is to be "rendered," that is, to find where the line from E to P hits the view plane.

All points on the line segment in space going from E to P have the form $E + \lambda(P - E)$ for some $\lambda \in [0, 1]$.

So, we want λ such that

$$N^T(E + \lambda(P - E) - A) = 0,$$

and by basic matrix algebra,

$$N^T(E + \lambda(P - E) - A) = N^T E + \lambda N^T(P - E) - N^T A = N^T(E - A) + \lambda N^T(P - E) = 0,$$

so

$$\lambda = \frac{N^T(A - E)}{N^T(P - E)}.$$

▷ Does this expression for λ always give a value in $(0, 1)$? Can it be zero? Negative? Undefined? Relate each of these to the corresponding positions of E and P relative to the view plane.

▷ Check the algebraic steps above carefully. Note that matrix algebra is the same as "regular algebra" except that you can only commute multiplication (like $AB = BA$) when one of the quantities is a scalar (a number as opposed to a matrix). Note that dot product quantities such as $A^T B$ are scalars. You also need to know, at places in the course, that the transpose operator has the properties

$$(A + B)^T = A^T + B^T,$$

$$(AB)^T = B^T A^T,$$

and

$$A^{T^T} = A.$$

Finally, to get the point that P renders to, we simply substitute the value for λ into the form of the point on the line, obtaining

$$E + \frac{N^T(A-E)}{N^T(P-E)}(P-E).$$

Section 1.3: Rendering Model Points to the Pixel Grid

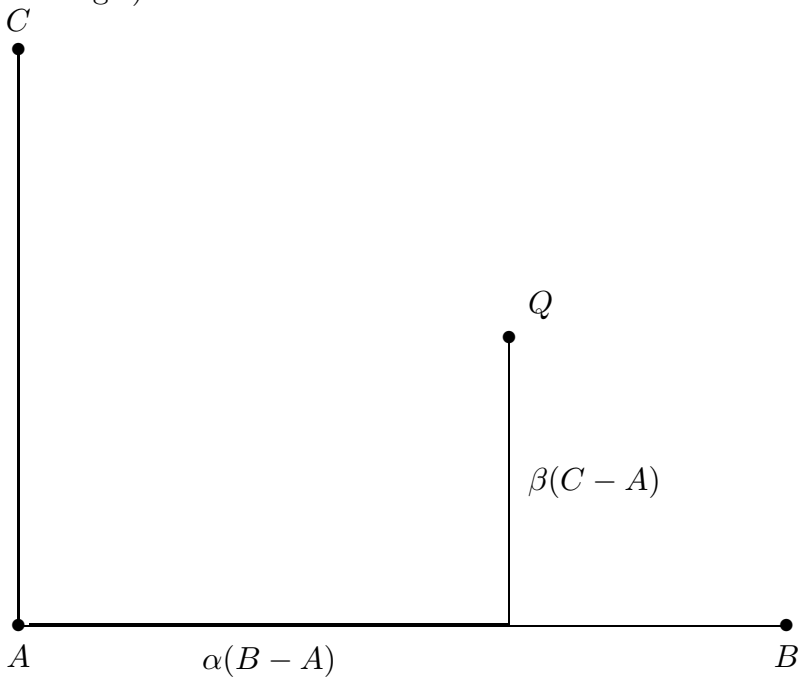
The preceding discussion covered a lot of the basics of rendering a model point P onto a view plane, but ignored the fact that we actually want to end up coloring pixels in a pixel grid that corresponds to points on the view plane.

To do this, we have to establish a *view rectangle* on the view plane and map the points on the view rectangle to the pixel grid.

To establish a view rectangle on the view plane, let's just say that A is the origin, at the lower left corner of the view rectangle, and use the segment from A to B as the bottom edge of the view rectangle, and use the segment from A to C as the left edge of the view rectangle. Equivalently, the segment from A to B lies along the positive x axis of the coordinate system we are establishing, and the segment from A to C lies along the positive y axis.

For this to make sense, we must have $(B - A)^T(C - A) = 0$, which is the same as saying that these two lines in space are perpendicular to each other.

Now, consider any point Q on the plane (we're thinking of the point that P renders to, of course). Here is a picture of the situation (where Q is shown as lieing within the view rectangle):



With this setup, it turns out that we can figure out the "coordinates" of Q in the new view rectangle coordinate system quite easily, as follows. We express the vector $Q - A$ as the sum of two vectors, one being a multiple of $B - A$, and the other being a multiple of $C - A$, namely let

$$Q - A = \alpha(B - A) + \beta(C - A)$$

Then, we just have to figure out what α and β must be. The trick is to multiply this equation on both sides by the same quantity. First, we use $(B-A)^T$, obtaining

$$(B-A)^T(Q-A) = (B-A)^T(\alpha(B-A)+\beta(C-A)) =$$

$$\alpha(B-A)^T(B-A)+\beta(B-A)^T(C-A) =$$

$$\alpha(B-A)^T(B-A),$$

so

$$\alpha = \frac{(B-A)^T(Q-A)}{(B-A)^T(B-A)}.$$

Similarly, by multiplying both sides by $(C-A)^T$, we can obtain

$$\beta = \frac{(C-A)^T(Q-A)}{(C-A)^T(C-A)}.$$

▷ Check this algebra, and note carefully how the perpendicularity of the edges $B-A$ and $C-A$ comes in.

The point of the previous discussion is that we can express any point Q on the view plane in terms of scalar multiples (α and β) of the coordinate vectors $B-A$ and $C-A$. Thus, these values α and β tell us what fraction of the way from the left edge to the right edge, and from the bottom edge to the top edge, the point lies. And, if these values are outside $[0,1]$, they tell us that Q does not lie in the view rectangle at all.

Mapping to the Pixel Grid

Now consider the pixel grid, say as viewed by the Java AWT class `Graphics`. It is a coordinate system with x values going from 0 to some integer value `w` (for "width," but note that there are really $w+1$ pixels in this range), and with y values going from 0 to `h`. But, the top-most row of pixels has $y=0$, with the y value increasing as you go downward, until you reach the lowest row of pixels that has $y=h$.

Now, we need to map each point (α,β), in the view rectangle system just described, to a pixel point (x,y). This is very easy to do, because we can just use a separate linear function for each map (in the x and in the y direction).

To find a linear map, we just need its value at two arguments. Here, starting with the x direction, note that we want 0 to map to 0 and 1 to map to w. So, we want the function $x(\alpha)$ that has $x(0)=0$ and $x(1)=w$, or, we want the equation of the line passing through the points $(0,0)$ and $(1,w)$.

Using basic algebra, it is easy to obtain

$$x(\alpha) = w\alpha.$$

▷ Verify that this function has the desired two values. Derive the equation of this line using basic algebra (find the slope, etc.).

To find the map for the y direction, we note that we want $y(0) = h$ and $y(1) = 0$, and easily obtain the map

$$y(\beta) = h(1 - \beta).$$

▷ Verify that this function has the desired two values. Derive the equation of this line using basic algebra.

This actually completes our work on the mathematics of our rendering process. Here is a summary of the entire process.

Summary of Idealized Rendering

Choose an eye point E.

Choose a view plane passing through the origin A and two other points B and C, where $B - A$ and $C - A$ are perpendicular and act, respectively, as the positive x and positive y coordinate axes on the view rectangle coordinate system.

Compute $N = (B - A) \otimes (C - A)$ as the normal to the plane.

Use a pixel grid in a Java window with x coordinates ranging from 0 to w inclusive, and y coordinates ranging from 0 to h inclusive (but oriented opposite to the usual coordinate system).

For each piece of stuff at location P in space, with color K:

Compute the point Q on the view plane where the line from E to P hits it by

$$Q = E + \frac{N^T(A - E)}{N^T(P - E)}(P - E).$$

Determine the coordinates of Q in the view rectangle coordinate system, namely (α, β) by

$$\alpha = \frac{(B - A)^T(Q - A)}{(B - A)^T(B - A)}$$

and

$$\beta = \frac{(C - A)^T(Q - A)}{(C - A)^T(C - A)}.$$

Compute the pixel grid location (x, y) corresponding to this point in the view rectangle coordinate system by

$$x = w\alpha$$

and

$$y = h(1 - \beta).$$

Color the pixel closest to (x, y) K.

Test Question Type 2

Given points A, B, and C on a view plane, with $(B-A)^T(C-A)=0$, a point Q on the plane, and pixel grid dimensions w and h, determine the point on the pixel grid that Q maps to.

▷ Run and study the code in the folder `feb7` at the course web site, starting with `App1.java`. Note what happens when the cube is moved in the positive z direction. Pay special attention to how the `ofTheWay` method works.

▷ Develop `App2` that lets the user move the eye point and view plane interactively as follows.

We currently require the application to provide the points A, B, and C that define the view rectangle, but it is easy to improve on this. Create a constructor for the renderer that takes the eye point E, a point in the scene F to "focus on," a desired distance d from E to the view plane, and an "up" direction U (which is a vector). Given this information, we can compute the required information for the view rectangle, as follows.

The normal N is taken as $F-E$, and then the vector pointing from F to the right along the view plane is $X=N\otimes U$, and the vector pointing from F upward along the view plane is $Y=X\otimes N$.

Now, to obtain the required three points on the plane, we first have to find the point G that is on the view plane, in the center of the view rectangle. This point is simply $G=E+\frac{d}{||F-E||}(F-E)$, where the "norm" or length of a vector V is defined as $||V||=\sqrt{V^TV}$. Then, we obtain $A=G-\frac{w}{2||X||}X-\frac{h}{2||Y||}Y$, $B=A+\frac{w}{||X||}X$, and $C=A+\frac{h}{||Y||}Y$.

Verify all this, and implement it in `App2`, which is copied from `App1` and modified as follows. Use the same keys as in `App1` for interaction, but leave the cube fixed in place and move either the eye point or the focus point F. Use the `<space>` key to toggle which is controlled. Interactively change the distance from the eye point to the view plane by hitting the keys `=` and `-`.

Exercise 1 Target due date: Wednesday, February 16.

Modify `App2.java` to `Ex1.java` so that the modeled scene includes a collection of rectangular boxes—edges only—scattered around (crudely simulating a city with a number of buildings). You must have enough buildings to give the impression of several "streets" with buildings lined up along them. Put the bases of all the buildings on the $y = 0$ plane. Your buildings must be of different dimensions. You will want to have a method that models a "box" given its desired location, dimensions, and color. Give each box a uniform color (as opposed to the "color cube" approach used in `App1`).

Turn in your file `Ex1.java` (the others will be copies of what we do in class, so I'll already have them) by email.

Section 1.4: Vertex-Based Modeling

We have been viewing the "scene" as consisting of a large number of bits of stuff, each with its own position in space and color, and we have successfully rendered such a scene by finding the point where the line from the eye point to the bit of stuff hits the view rectangle, and mapping that point to the corresponding point on the pixel grid.

This is all fine, as far as it goes (except for efficiency issues, of course, which we have intentionally been neglecting), but one problem is that there is simply too much "stuff" in a scene for this approach to really work—it is not at all clear how to decide how many points to render in order to tidily fill the pixel grid. Doing too few model points will leave unpleasant holes in the rendered scene displayed in the pixel grid, and doing too many model points will simply be a huge waste of time.

The standard solution to this problem is to pick relatively few, meaningful points on the surface of an object that is being modeled, and use polygons through these points to approximate the surface of the object. These points are known as *vertices*, and this approach is known as vertex-based modeling.

For example, to model a rectangular box, 8 vertices can be used, with groups of four vertices determining the polygonal faces of the box.

As another example, to model a sphere, we need to pick a fairly large number of vertices on its surface and use the polygons they form to approximate the surface.

▷ Think about doing this on a sphere by taking some lines of latitude and longitude and using as the vertices the points where they cross. What type of polygons are formed? Repeat by starting with the 6 vertices of a regular octahedron on a sphere and refining the mesh by converting each triangle into three triangles by taking the center point as the new vertex. Repeat again starting with an octahedron but this time refine each triangle into 4 smaller triangles by using midpoints of the sides as the new vertices.

The key point for rendering a vertex-based model is to note that all the vertices are rendered, just like we have been doing, but after obtaining the corresponding pixel grid points for the vertices of a polygon, we then need to *rasterize* the polygon, which is the

process of filling in, with the correct interpolated color, all the pixel grid points that are in the interior of the polygon.

Note that this is a radically different strategy from what we were previously suggesting. In our idealized approach, we imagined breaking the model object into a large number of points and rendering all those points. With the vertex-based approach, only a few vertices need to be rendered, and then all the other points are colored in by scanning the pixel grid. This involves far less computation, and ensures that all the points inside the polygon are colored in.

Scan-Line Algorithm for Filling a Convex Polygon

To rasterize a polygon with integer coordinates for its vertices, a so-called scan-line algorithm proceeds by examining every horizontal line with a y level that intersects the polygon, determining the first and last x values for which the line hits the polygon, and filling in the pixels along that horizontal line by interpolating.

To be more precise, but not work too hard, let's focus on triangles, because they are simpler, and have a crucial property that polygons with more than 3 vertices do not: they are guaranteed to be convex (a convex polygon is one for which if you take any two points in its interior—including the border—then the entire line segment between them lies in the region of the polygon).

Let the vertices be (x_1, y_1), (x_2, y_2), and (x_3, y_3). Note that these are the pixel grid coordinates for the points obtained by rendering the original three vertices in 3D space.

> Repeat for all integer values of y going from the minimum of y_1, y_2, and y_3 to the maximum of y_1, y_2, and y_3:
>
> > Find all intersections of the horizontal scan line at height y with the three edges of the polygon, and let x_l and x_r be the smallest and largest x values, respectively, of intersection points.
> >
> > As part of doing this, express $(x_l, y) = (1 - \lambda_l)A + \lambda_l B$, where A and B are endpoints of the edge that the scan line hits the farthest left, and similarly express $(x_r, y) = (1 - \lambda_r)C + \lambda_r D$, where C and D are endpoints of the edge that the scan line hits the farthest right.
> >
> > Determine the colors of the pixels at (x_l, y) and (x_r, y) by interpolating in each of the colors (red, green, and blue). For example, the redness of the pixel at (x_l, y) will be $(1 - \lambda_l)r_A + \lambda_l r_B$, where r_A is the redness of point A and r_B is the redness of point B. The other five quantities are computed similarly.
> >
> > For each integer x going from x_l to x_r, compute $\lambda = (x - x_l)/(x_r - x_l)$, and interpolate the color at (x, y) as being λ of the way from the color at (x_l, y) to the color at (x_r, y).

- ▷ Draw some sketches of triangles in the pixel grid and make sense out of the algorithm for those sketches. Try some triangles that might mess up the algorithm, like say one with a horizontal or vertical edge.
- ▷ Note that except in one case, finding the points A, B, C, and D can be done quite nicely by checking incrementally how they change.
- ▷ Draw a non-convex polygon for which this algorithm fails.

Test Question Type 3

Given any appropriate situation, be able to linearly interpolate from one point to another, using either position or color.

As practice for this, note how the form

$$(1 - \lambda)A + \lambda B$$

is used to interpolate in various ways in the scan-line algorithm.

Clipping

The approach we have developed to this point has a major inefficiency related to the issue of *clipping*. In general, clipping refers to the idea of not rendering parts of the scene that end up being out of sight. Our current approach renders the three vertices of a triangle, any of which may or may not map into the pixel grid, and then rasterizes that triangle, obtaining integer coordinates and colors for each "pixel" in the triangle, even if those coordinates put it out of the pixel grid. A more efficient approach would be to find the intersection of the triangle with the pixel grid, and then rasterize the resulting polygon. We will not follow this approach, just because it is a little more difficult to program, and we are allowing ourselves to be inefficient.

- ▷ Note how inefficient our approach could be, for example when the entire triangle is out of sight.
- ▷ Draw some examples of triangles intersecting rectangles (the pixel grid). Note that the intersection can be quite a bit more complex than a triangle.

Hidden Surface Removal

Our previous approach to modeling and rendering actually had a major flaw that we have not mentioned, namely that two bits of stuff could map to the same pixel, in which case the one drawn later would determine the color of the pixel, which is wrong in the case that the bit of stuff drawn first is actually in front of the one drawn second, along the line from the eye point to the points in space.

In the past in computer graphics, when memory was at a premium, a number of elaborate algorithms for handling this problem of "hidden surface removal" were developed, but all we, with no concerns about time or memory efficiency, have to do is keep a *depth* value

stored for each pixel, which is the "depth" of the bit of stuff that was most recently drawn at that location. Then, when a new bit of stuff wants to color a pixel, the depth of the bit of stuff has to be less than the depth currently stored, or else it is ignored.

Depth

What we want for "depth" of a bit of stuff is the distance from the eye point, because that is clearly what determines which of two bits of stuff is closer and hence should get to determine the color. With our earlier approach, that meaning of depth would work fine, but with vertex-based modeling and rendering, we need to work a little harder.

The problem is that even if we had the distances from the eye point for two vertices, when we rasterize the line segment between their rendered points, we use linear interpolation, and distance from the eye point does not follow linear interpolation.

- ▷ Consider vertices A and B, and a typical point $P(\lambda) = (1-\lambda)A+\lambda B$ along the segment between them. Algebraically compute $||P(\lambda) - E||$ far enough to be convinced that this quantity is *not* a linear function of λ that could easily be interpolated. As simpler evidence of this, draw two vertices that are the same distance from an eye point and note that the points in between them have a distance from the eye point that varies non-linearly.

So, instead of trying to monitor the distance from the eye point of various bits of stuff, we will use their perpendicular distance from the view plane, because this quantity does vary linearly.

Computing the Distance from the View Plane

Let Q be any point in space, and let a view plane be defined by normal vector N and a point A on it. The "distance from Q to the plane" is defined as the distance from Q to the point R on the plane that is the closest point to Q on the plane. Clearly for some α,

$$Q = R + \alpha N.$$

Thus,

$$Q - A = R - A + \alpha N,$$

so

$$N^T(Q - A) = N^T(R - A + \alpha N) = N^T(R - A) + \alpha N^T N = \alpha N^T N,$$

since R is on the plane. Thus,

$$\alpha = \frac{N^T(Q - A)}{N^T N},$$

and the distance from Q to the plane is

$$||\alpha N|| = |\alpha|||N||,$$

so we now know how to compute the depth of a vertex in space, defined as the shortest distance from the vertex to the plane.