CMSC 427: Computer Graphics[1]
Fall 2000
Dave Mount

Lecture 1: Course Introduction

(Thursday, Aug 31, 2000)
Read: Chapter 1 in Hill.

Computer Graphics: Computer graphics is concerned with producing images and animations (or sequences of images) using a computer. This includes the hardware and software systems used to make these images. The task of producing photo-realistic images is an extremely complex one, but this is a field that is in great demand because of the nearly limitless variety of applications. The field of computer graphics has grown enormously over the past 10–20 years, and many software systems have been developed for generating computer graphics of various sorts. This can include systems for producing 3-dimensional models of the scene to be drawn, the rendering software for drawing the images, and the associated user-interface software and hardware. Our focus in this course will not be on how to use these systems to produce these images (you can take courses in the art department for this) but in understanding how these systems are constructed, and the underlying mathematics, physics, algorithms, and data structures needed in the construction of these systems.

The field of computer graphics dates back to the early 1960's with Ivan Sutherland, one of the pioneers of the field. This began with the development of the (by current standards) very simple software for performing the necessary mathematical transformations to produce simple line-drawings of 2- and 3-dimensional scenes. As time went on, and the capacity and speed of computer technology improved, successively greater degrees of realism were achievable. Today it is possible to produce images that are practically indistinguishable from photographic images (or at least that create a pretty convincing illusion of reality).

Overview: Given the state of current technology, it would be possible to design an entire university major to cover everything (important) that is known about computer graphics. In this introductory course, we will attempt to cover only the meerest *fundamentals* upon which the field is based. Nonetheless, with these fundamentals, you will have a remarkably good insight into how many of the modern video games and Hollywood animations are produced. This is true since even very sophisticated graphics stem from the same basic elements that simple graphics do. They just involve much more complex light and physical modeling, and more sophisticated rendering techniques.

In this course we will deal primarily with the task of producing a single image from a 2- or 3-dimensional scene model. This is really a very limited aspect of computer graphics. For example, it ignores the role of computer graphics in tasks such as visualizing things that cannot be described as such scenes. This includes rendering of technical drawings including engineering charts and architectural blueprints, and also scientific visualization such as mathematical functions, ocean temperatures, wind velocities, and so on. We will also ignore many of the issues in producing animations. We will produce simple animations (by producing lots of single images), but issues that are particular to animation, such as motion blur, morphing and blending, temporal anti-aliasing, will not be covered. They are the topic of a more advanced course in graphics.

Let us begin by considering the process of drawing (or *rendering*) a single image of a 3-dimensional scene. This is crudely illustrated in the figure below. The process begins by producing a mathematical model of the object to be rendered. Such a model should describe not only the shape of the object but its color, its surface finish (shiny, matte, transparent, fuzzy, scaly, rocky). Producing realistic models is extremely complex, but luckily it is not our main concern. We will leave this to the artists and modelers. The scene model should also include information about the location and characteristics of the light sources (their color, brightness), and the atmospheric nature of the medium through which the light travels (is it foggy or clear). In addition we will need to know the location of the viewer. We can think of the viewer as holding a "synthetic camera", through which the image is to be photographed. We need to know the characteristics of this camera (its focal length, for example).

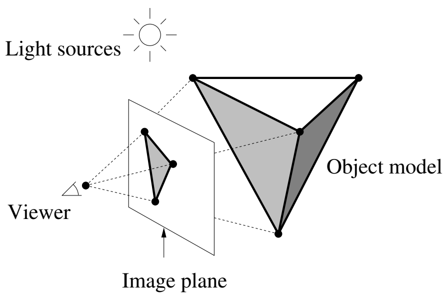


Figure 1: A typical rendering situation.

Based on all of this information, we need to perform a number of steps to produce our desired image.

Projection: Project the scene from 3-dimensional space onto the 2-dimensional image plane in our synthetic camera.

Color and shading: For each point in our image we need to determine its color, which is a function of the object's surface color, its texture, the relative positions of light sources, and (in more complex illumination models) the indirect reflection of light off of other surfaces in the scene.

Hidden surface removal: Elements that are closer to the camera obscure more distant ones. We need to determine which surfaces are visible and which are not.

Rasterization: Once we know what colors to draw for each point in the image, the final step is that of mapping these colors onto our display device.

The Course in a Nutshell: The process that we have just described involves a number of steps, from modeling to rasterization. The topics that we cover this semester will consider many of these issues.

Basics:

Graphics Programming: OpenGL, graphics primitives, color, viewing, event-driven I/O, GL toolkit, frame buffers.

Geometric Programming: Review of linear algebra, affine geometry, (points, vectors, affine transformations), homogeneous coordinates, change of coordinate systems.

Implementation Issues: Rasterization, clipping.

Modeling:

Model types: Polyhedral models, hierarchical models, fractals and fractal dimension.

Curves and Surfaces: Representations of curves and surfaces, interpolation, Hermite, Bezier, B-spline curves and surfaces, NURBS.

Surface finish: Texture-, bump-, and reflection-mapping.

Projection:

3-d transformations and perspective: Scaling, rotation, translation, orthogonal and perspective transformations, 3-d clipping.

Hidden surface removal: Back-face culling, z-buffer method, depth-sort.

Issues in Realism:

Light and shading: Diffuse and specular reflection, the Phong and Gouraud shading models.

Ray tracing: Ray-tracing model, reflective and transparent objects, shadows.

Color: Gamma-correction, halftoning, and color models.

Lecture 2: Graphics Systems and Models

(Tuesday, Sep. 5, 2000)
Read: Chapter 1 in Hill.

Elements of Pictures: Computer graphics is all about producing pictures (realistic or stylistic) by computer. Before discussing how to do this, let us first consider the elements that make up images and the devices that produce them. How are graphical images represented? There are four basic types that make up virtually of computer generated pictures: *polylines*, *filled regions*, *text*, and *raster images*.

Polylines: A polyline (or more properly a *polygonal curve* is a finite sequence of line segments joined end to end. These line segments are called *edges*, and the endpoints of the line segments are called *vertices*. A single line segment is a special case. (An infinite line, which stretches to infinity on both sides, is not usually considered to be a polyline.) A polyline is *closed* if it ends where it starts. It is *simple* if it does not self-intersect. Self-intersections include such things as two edge crossing one another, a vertex intersecting in the interior of an edge, or more than two edges sharing a common vertex.

A polyline in the plane can be represented simply as a sequence of the (x, y) coordinates of its vertices. This is sufficient to encode the geometry of a polyline. In contrast, the way in which the polyline is rendered is determined by a set of properties call *graphical attributes*. These include elements such as *color*, *line width*, and *line style* (solid, dotted, dashed), how consecutive segments are *joined* (rounded, mitered or beveled; see the book for further explanation).

Many graphics systems support common special cases of curves such as circles, ellipses, circular arcs, and Bezier and B-splines. We should probably include *curves* as a generalization of polylines. Most graphics drawing systems implement curves by breaking them up into a large number of very small polylines, so this distinction is not very important.

Filled regions: Any simple, closed polyline in the plane defines a region consisting of an inside and outside. (An utterly obvious fact from topology, which is notoriously hard to prove.) We can fill any such region with a color or repeating pattern. In some instances the bounding polyline itself is also drawn and others the polyline is not drawn.

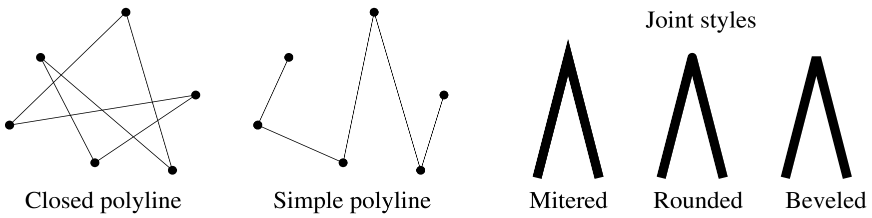


Figure 2: Polylines.

A polyline with embedded "holes" also naturally defines a region that can be filled. In fact this can be generalized by nesting holes within holes (alternating color with the background color). Even if a polyline is not simple, it is possible to generalize the notion of interior. Given any point, shoot a ray to infinity. If it crosses the boundary an odd number of times it is colored. If it crosses an even number of times, then it is given the background color.

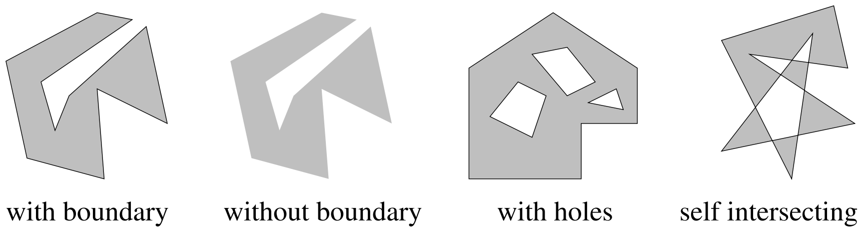


Figure 3: Filled regions.

Text: Although we do not normally think of text as a graphical output, it occurs frequently within graphical images such as engineering diagrams. Text can be thought of as a sequence of characters in some *font*. As with polylines there are numerous attributes which affect how the text appears. This includes the *size* (which is usually measured in *points*, a unit of measure equal to 1/72-inch), *weight* (medium, bold), *slant* (italic, roman), and *color*.

Raster Images: Raster images are what most of us think of when we think of a computer generated image. Such an image is a 2-dimensional array of square (or generally rectangular) cells called *pixels* (short for "picture elements"). Such images are sometimes called *pixel maps*.

The simplest example is an image made up of black and white pixels, each represented by a single bit (0 for black and 1 for white). This is called a *bitmap*. For gray-scale (or *monochrome*) raster images raster images, each pixel is represented by assigning it a numerical value over some range (e.g., from 0 to 255, ranging from black to white). There are many possible ways of encoding color images. We will discuss these further below.

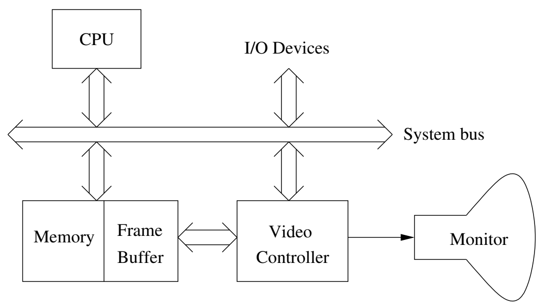
Graphics Devices: The standard interactive graphics device today is called a *raster display*. As with a television, the display consists of a two-dimensional array of pixels. There are two common types of raster displays.

Video displays: consist of a screen with a phosphor coating, that allows each pixel to be illuminated momentarily when struck by an electron beam. A pixel is either illuminated (white) or not (black). The level of intensity can be varied to achieve arbitrary gray values.

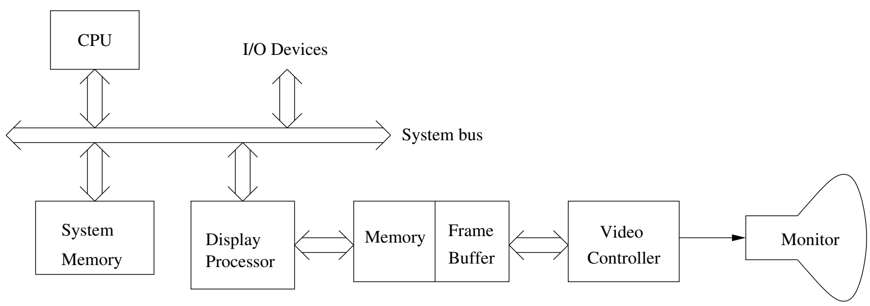
Because the phosphor only holds its color briefly, the image is repeatedly rescanned, at a rate of at least 30 times per second.

Liquid crystal displays (LCD's): use an electronic field to alter polarization of crystalline molecules in each pixel. The light shining through the pixel is already polarized in some direction. By changing the polarization of the pixel, it is possible to vary the amount of light which shines through, thus controlling its intensity.

Irrespective of the display hardware, the computer program stores the image in a two-dimensional array in RAM of pixel values (called a *frame buffer*). The display hardware produces the image line-by-line (called *raster lines*). A hardware device called a *video controller* constantly reads the frame buffer and produces the image on the display. The frame buffer is not a device. It is simply a chunk of RAM memory that has been allocated for this purpose. A program modifies the display by writing into the frame buffer, and thus instantly altering the image that is displayed. An example of this type of configuration is shown below.



Simple Raster Graphics System



Raster Graphics with Display Processor

Figure 4: Raster display architectures.

Complex graphics systems achieve great speed by providing separate hardware support, in the form of a *display processor* (more commonly known as a *graphics accelerator* or *graphics card* to PC users). This relieves the computer's main processor from much of the mundane repetitive effort involved in maintaining the frame buffer. A typical display processor will provide assistance for a number of operations including the following:

Transformations: Rotations and scalings used for moving objects and the viewer's location.

Clipping: Removing elements that lie outside the viewing window.

Projection: Applying the appropriate perspective transformations.

Texture mapping: Coloring objects by "painting" textures onto their surface.

Hidden-surface elimination: Determines which of the various objects that project to the same pixel is closest to the viewer and hence is displayed.

An example of this architecture is shown in the figure above. These operations are often *pipelined*, where each processor on the pipeline performs its task and passes the results to the next phase. Given the increasing demands on a top quality graphics accelerator, they have become quite complex. The following figure shows the architecture of existing accelerator. (Don't worry about understanding the various elements just now.)

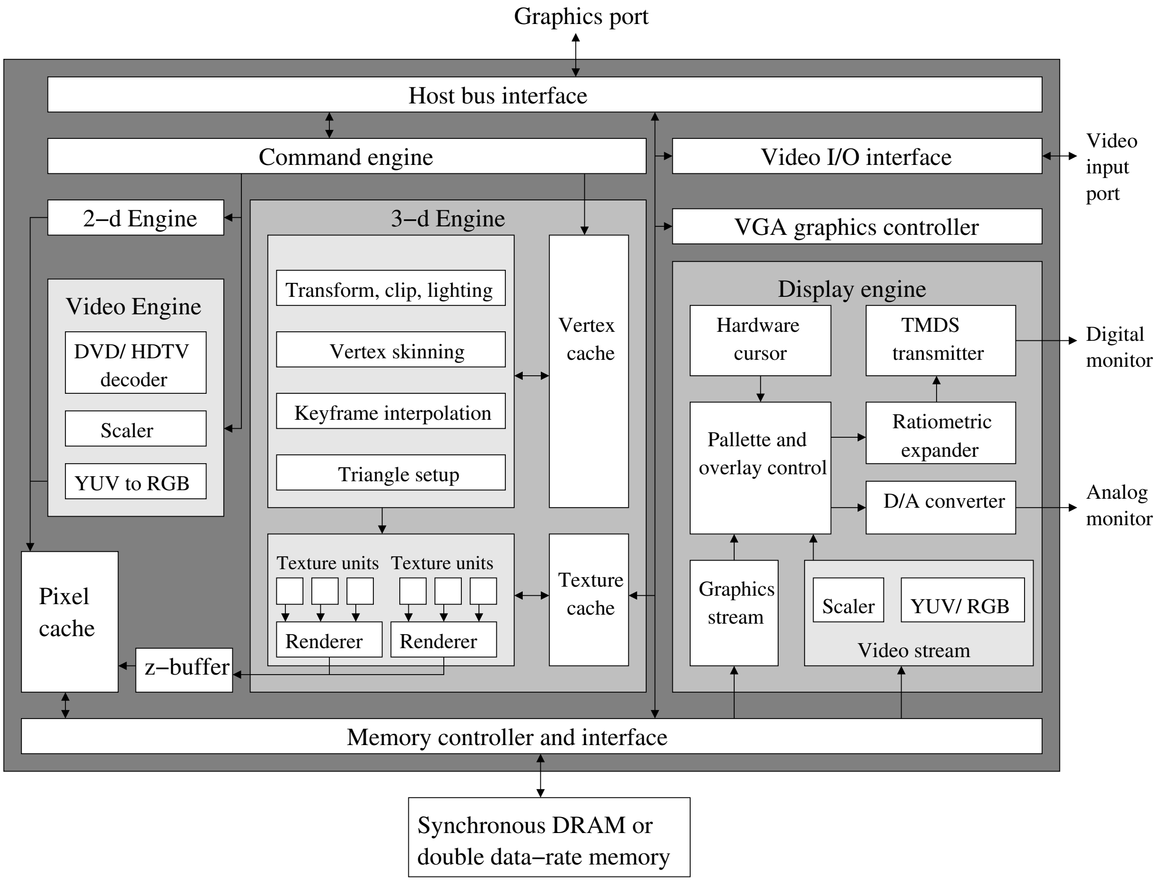


Figure 5: Graphics accelerator architecture.

Color: The method chosen for representing color depends on the characteristics of the graphics output device (e.g., whether it is *additive* as are video displays or *subtractive* as are printers). It also depends on the number of bits per pixel that are provided, called the *pixel depth*. For example, the most method used currently in video and color LCD displays is a *24-bit RGB* representation. Each pixel is represented as a mixture of red, green and blue components, and each of these three colors is represented as a 8-bit quantity (0 for black and 255 for the brightest color).

In many graphics systems it is common to add a fourth component, sometimes called *alpha*, denoted A. This component is used to achieve various special effects, most commonly in describing how opaque a color is. We will discuss its use later in the semester. For now we will ignore it.

In some instances 24-bits may be unacceptably large. For example, when downloading images from the web, 24-bits of information for each pixel may be more than what is needed. A common alternative is to used a *color map*, also called a *color look-up-table* (LUT). (This is the method used in most gif files, for example.) In a typical instance, each pixel is represented by an 8-bit quantity in the range from 0 to 255. This number is an index to a 256-element array, each of whose entries is a 234-bit RGB value. To represent the image, we store both the LUT and the image itself. The 256 different colors are usually chosen so as to produce the best possible reproduction of the image. For example, if the image is mostly blue and red, the LUT will contain many more blue and red shades than others.

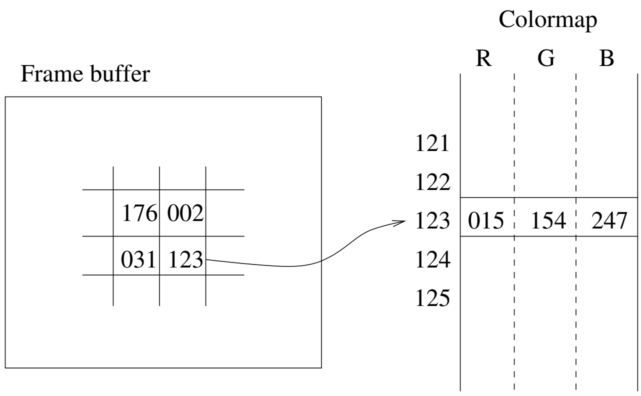


Figure 6: Color-mapped color.

Although this is fine for most nongraphics applications (e.g., special highlighting for icons and fonts), it is woefully inadequate for displaying realistic images with lots of colors and shading. So how is it that you can display high quality images on, say netscape. The answer involves a fair amount of clever trickery to fool the eye into seeing many shades of colors where only a small number of distinct colors exist. This process is called *digital halftoning*. Examples of this that you are familiar with are photographs in newspapers. Many small black dots are used to approximate shades of gray.

Lecture 3: Drawing in OpenGL: Glut

(Thursday, Sep 7, 2000)
Read: Chapter 2 in Hill.

The OpenGL API: Today we will begin discussion of using OpenGL, and its related libraries, Glu (OpenGL utility library) and Glut (an OpenGL utility toolkit). OpenGL is designed to be a machine-independent graphics library, but one that can take advantage of the structure of typical hardware accelerators for computer graphics.

The Main Program: Before discussing how to actually draw shapes, we will begin with the basic elements of how to create a window. OpenGL was intentionally designed to be independent of any specific window system. It is the Glut toolkit which provides the necessary tools for requesting the windows be created and providing interaction with I/O devices. Let us begin by considering a typical main program, and then dissecting its various elements. This creates a window that is 300 pixels wide and 400 pixels high, located in the upper left corner of the display.

Typical OpenGL/Glut Main Program

```
int main(int argc, char** argv)                 // program arguments
{
    glutInit(&argc, argv);                      // initialize glut and gl
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);// double buffering and RGB color
    glutInitWindowSize(300, 400);               // initial window size
    glutInitWindowPosition(0, 0);               // initial window position
    glutCreateWindow(argv[0]);                  // create window

    ...initialize callbacks here (described below)...

    myInit();                                   // your own initializations
    glutMainLoop();                             // turn control over to glut
    return 0;                                   // we don't really return here
}
```

Initialization: The arguments given to the main program (argc and argv) are the command-line arguments supplied to the program. We pass these into the main initialization procedure, `glutInit()`. This procedure must be called before any others. It processes (and removes) command-line arguments that may be of interest to Glut and the window system and does general initialization of Glut and OpenGL.

Display Mode: The next procedure, `glutInitDisplayMode()`, performs initializations informing OpenGL how to set up the frame buffer. As we mentioned in class, the frame buffer stores the color information for the pixels as well as other information (e.g. depth for hidden surface removal). The system needs to know how we are representing colors of our general needs in order to determine the depth (number of bits) to assign for each pixel in the frame buffer.

Display Mode	Meaning
`GLUT_RGB`	Use RGB colors
`GLUT_RGBA`	Use RGB plus α (for transparency)
`GLUT_INDEX`	Use colormapped colors (not recommended)
`GLUT_DOUBLE`	Use double buffering (recommended)
`GLUT_SINGLE`	Use single buffering (not recommended)
`GLUT_DEPTH`	Use depth-buffer (for hidden surface elim.)

Figure 7: Arguments to `glutInitDisplayMode()`.

Its argument is a logical-or (using the operator "|") of a number of possible options. First off, we need to tell the system how colors will be represented. There are three methods, of which two are fairly commonly used: `GLUT_RGB` or `GLUT_RGBA`. The first uses standard RGB colors, and is the default. The second requests RGBA coloring. In this color system there is a fourth component (A or α), which indicates the opaqueness of the color (1 = fully opaque, 0 = fully transparent). This is useful in creating transparent effects. We will discuss this later this semester.

The next option specifies single- or double-buffering: `GLUT_SINGLE` or `GLUT_DOUBLE`, respectively. Remember that whatever is written to the frame buffer is immediately transfered to the display. When you write into the frame buffer, you usually set it to some background color and then draw in the new contents. Unless the drawing happens very fast, the user will be annoyed by this continuous blanking out and redrawing of the image. In double-buffering, you maintain two buffers. The *front buffer* is the one which is displayed. You draw to the other one, called the *back buffer*. Then to update the image, you simply swap the two buffers. The swapping process is very fast, and appears to happen essentially instantaneously. Double-buffering requires twice the buffer space, but is almost always preferred with interactive graphics.

One other option that we will need later with 3-dimensional graphics will be hidden surface removal. This fastest and easiest (but most space-consuming) way to do this is with a depth buffer, which records not only the color of each pixel, but its distance from the viewer. This is made possible with the option `GLUT_DEPTH`. For this program it is not needed, and so has been omitted.

Window setup: The command `glutInitWindowSize(int width, int height)` sets the desired window size in pixels. The command `glutInitWindowPosition(int x, int y)` sets its position, where the (x, y) coordinates indicate the upper left corner of the window, and where $(0, 0)$ is the upper left corner of the display. Note that these are considered *suggestions* to the window manager on how to set up your window. Depending on its layout policies, and the size of the display, it may not honor your requests.

The command, `glutCreateWindow(char *title)`, creates a window and gives it the specified title. The title can be any string. We pass in `argv[0]`, which contains the name of your program.

Callbacks: Virtually all interactive graphics programs are *event driven*. Unlike traditional programs that read from a standard input file, a graphics program must be prepared at any time for input from any number of sources (mouse or keyboard for example).

In OpenGL this is done through the use of *callbacks*. The graphics program instructs the system to invoke a particular procedure whenever, say, the mouse button is clicked. The graphics program *registers* for the various events in which it is interested. This involves telling the window system that when an event of a particular type occurs, please call a particular procedure whose name you provide.

There is one event which any application program must list for, called a *display event*. This is signaled when the window is first displayed, or whenever an obscuring window has moved away, thus revealing portions of a hidden window. Other events include mouse clicks, motion of the mouse (without clicking), keyboard hits. Note that you are only signaled about events that happen to your window. (For example, entering text into another program's dialogue box will not generate a keyboard event for your program.)

In an animation, the user may not be providing any input at all. In these cases the program can register for either a *timer event* or an *idle event*. An idle event is generated every time the system has nothing better to do. This can generate a huge number of events. A better approach is to request a timer event. In a timer event you request that your program go to sleep for some period of time and that it be "awakened" by an event some time later. In `glutTimerFunc()` the first argument gives the sleep time as an integer in milliseconds and the last argument is an integer identifier, which is passed into the callback function.

For example, to specify the display function and mouse callbacks, you create procedures for these events, and initialize them as follows.

Typical Callback Setup

```
int main(int argc, char** argv)
{
    ...
    glutDisplayFunc(myDraw);                // set up the callbacks
```

Event	Callback request	Callback prototype (all return `void`)
(Re)display	`glutDisplayFunc`	`myDisplay()`
(Re)size window	`glutReshapeFunc`	`myResize(int w, int h)`
Mouse button	`glutMouseFunc`	`myMouse(int b, int s, int x, int y)`
Mouse motion	`glutPassiveMotionFunc`	`myMotion(int x, int y)`
Keyboard key	`glutKeyboardFunc`	`myKeyboard(unsigned char c, int x, int y)`
Timer event	`glutTimerFunc`	`myTimer(int id)`
Idle event	`glutIdleFunc`	`myIdle()`

Figure 8: Common callbacks and the associated registration functions.

```
    glutReshapeFunc(myResize);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
    glutTimerFunc(20, myTimeOut, 0);
    ...
}
```

Callback Contents: What does a typical callback function do? This depends entirely on the application that you are designing. Here are some examples of things that a typical callback might contain. Note that the timer callback and the reshape callback both invoke `glutPostRedisplay()`. This procedure informs OpenGL that the state of the scene has changed and should be redrawn (by calling your drawing procedure). This might be requested in other callbacks as well.

Typical Callback Code

```
void myDraw(void) { ...insert scene redrawing code here ...  }

void myResize(int w, int h) {
    windowWidth = w; windowHeight = h;  // save window width and height

    ...update the projection transformation...

    glutPostRedisplay();                // request redisplay
}

void myMouse(int b, int s, int x, int y) {
    switch (b) {
        case GLUT_LEFT_BUTTON:
            if (s == GLUT_DOWN) {
                ...left button pushed down...
            }
            else if (s == GLUT_UP) {
                ...left button released...
            }
            break;
        ...
    }
}

void myKeyboard(unsigned char c, int x, int y) {
   switch (c) {
```

```
        case 'q': exit(0);              // 'q' means quit
                  break;
        ...
      }
    }

    void myTimeOut(int id) {
        ...advance the state of animation incrementally...

        glutPostRedisplay();             // request redisplay
        glutTimerFunc(20, myTimeOut, 0);    // request next timer event
    }
```

Lecture 4: Drawing in OpenGL: Drawing and Viewports

(Tuesday, Sep 12, 2000)
Read: Chapter 2 in Hill.

Basic Drawing: We have shown how to create a window, how to get user input, but we have not discussed how to get graphics to appear in the window. Before being able to draw a scene, OpenGL needs to know the following information: what are the *objects* to be drawn, how is the image to be *projected* onto the window, and how *lighting* and *shading* are to be performed. To begin with, we will consider a very the simple case. There are only 2-dimensional objects, no lighting or shading, and relatively little user interaction.

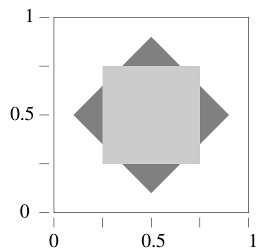


Figure 9: Drawing produced by the simple display function.

We will consider a simple drawing routine for the picture shown in the figure. We assume that our idealized drawing region is a unit square over the real interval $[0, 1]$. Unlike Glut's convention, here the coordinates will be floating point values and the origin will be in the lower left corner. This is strange, but it seems to be an unfortunate fact of life in graphics. Virtually all window systems work with the origin in the upper left (and Glut followed this convention), and most graphics systems use OpenGL's lower right convention.

The display callback function is shown below. To draw the image we will first erase whatever is in the image, then do our drawing, and finally swap buffers (so that what we have drawn becomes visible). This function first draws a red diamond and then (on top of this) it draws a blue rectangle. Let us assume double buffering is being performed, and so the last thing to do is invoke `glutSwapBuffers()` to make everything visible.

Simple Display Function

```
    void simpleDisplay()                        // display function
    {
```

```
        glClear(GL_COLOR_BUFFER_BIT);                  // clear the window

        glColor3f(1.0, 0.0, 0.0);                      // set color to red
        glBegin(GL_POLYGON);                           // draw a red diamond
            glVertex2f(0.90, 0.50);
            glVertex2f(0.50, 0.90);
            glVertex2f(0.10, 0.50);
            glVertex2f(0.50, 0.10);
        glEnd();

        glColor3f(0.0, 0.0, 1.0);                      // set color to blue
        glRectf(0.25, 0.25, 0.75, 0.75);               // draw a blue rectangle

        glutSwapBuffers();                             // swap buffers
    }
```

Clearing the Window: The command `glClear()` clears the window, by overwriting it with the background color. This is set by the call

`glClearColor(GLfloat R, GLfloat G, GLfloat B, GLfloat A).`

The type `GLfloat` is OpenGL's redefinition of the standard `float`. To be correct, you should use the approved OpenGL types (e.g. `GLfloat`, `GLdouble`, `GLint`) rather than the obvious counterparts (`float`, `double`, and `int`). Usually these are the same, but not always. For example on the CSD machines `GLint` is defined to be `long`.

Recall that the A value is set to 1 for opaque colors. Thus to set the background color to black, we would use `glClearColor(0.0, 0.0, 0.0, 1.0)`, and to set it to blue use `glClearColor(0.0, 0.0, 1.0, 1.0)`, (Note: For debugging purposes, it is often a good idea to use an uncommon color, since black is the color that most often arises when a bug is present.) Since the background color is usually independent of drawing, the function `glClearColor()` is typically set in one of your initialization procedures.

Clearing the window involves resetting information within the frame buffer. As we mentioned before, the frame buffer may store different types of information (color and depth information, for example). Typically when the window is cleared, we want to clear everything. But sometimes, in order to achieve special effects, we may clear one part of the buffer without clearing others. So the `glClear()` command allows the user to select what is to be cleared. In this case we only have color in the depth buffer, which is selected by the option `(GL_COLOR_BUFFER_BIT`. If we had a depth buffer to be cleared it as well we could do this by "or"ing together: `(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`.

Drawing Attributes: The OpenGL drawing commands describe the geometry of the object that you want to draw. More specifically, all OpenGL is based on drawing objects with straight sides, so it suffices to specify the *vertices* of the object to be drawn. The manner in which the object is displayed is determined by various *drawing attributes* (color, point size, line width, etc.).

For example, the command `glColor3f()` sets the drawing color. The arguments are three `GLfloat`'s, giving the R, G, and B components of the color. In this case, RGB = $(1, 0, 0)$ means pure red. Once set, the attribute applies to all subsequently defined objects, until it is set to some other value. Thus, we could set the color, draw three polygons with the color, then change it, and draw five polygons with the new color.

This call illustrates a common feature of many OpenGL commands, namely flexibility in argument types. The suffix "3f" means that three floating point arguments (actually `GLfloat`'s) will be given. For example, `glColor3d()` takes three `double` (or `GLdouble`) arguments, `glColor3ui()` takes three `unsigned int` arguments, and so on. For floats and doubles, the arguments range from 0 (no intensity) to 1 (full intensity). For integer types (byte, short, int, long) the input is assumed to be in the range from 0 (no intensity) to its maximum possible value (full intensity). Thus, for integer arguments `MAXINT` would be full intensity.

If we were using RGBA color rather than RGB color, we could use `glColor4f()` instead. (For opaque colors set $A = 1.0$.) In some cases it is convenient to store your colors in an arrays with three elements. The suffix "v" means that the argument is a vector. For example `glColor3fv()` expects a single argument, a vector containing three `float`'s.

Drawing commands: OpenGL supports drawing of a number of different types of objects. The simplest is `glRectf()`, which draws a filled rectangle. All the others are complex objects consisting of a (generally) unpredictable number of elements. This is handled in OpenGL by the constructs `glBegin(`*mode*`)` and `glEnd()`. Between these two commands a list of vertices is given, which defines the object. The sort of object to be defined is determined by the *mode* argument of the `glBegin()` command. Some of the possible modes are illustrated in the figure below. For details on the semantics of the drawing methods, see the reference manuals.

Note that in the case of `GL_POLYGON` only *convex polygons* (internal angles less than 180 degrees) are supported. You must subdivide nonconvex polygons into convex pieces.

```
glBegin(mode);
    glVertex(v0); glVertex(v1); ...
glEnd();
```

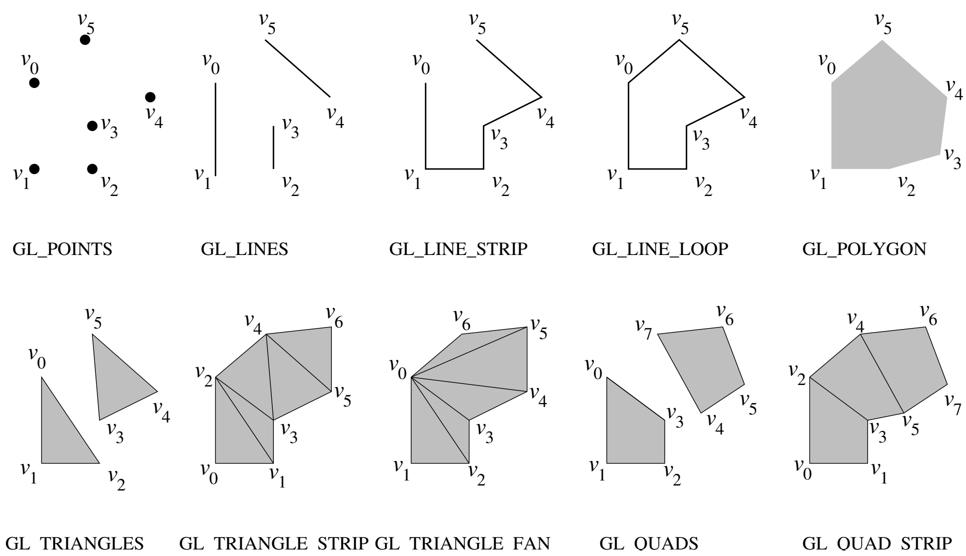


Figure 10: Some OpenGL object definition modes.

In the example above we only defined the x- and y-coordinates of the vertices. How does OpenGL know whether our object is 2-dimensional or 3-dimensional? The answer is that it does not know. OpenGL represents all vertices as 3-dimensional coordinates internally. (This may seem wasteful, but remember that OpenGL is designed primarily for 3-d graphics.) If you do not specify the z-coordinate, then it simply sets the z-coordinate to zero. By the way, `glRectf()` always draws its rectangle on the $z = 0$ plane.

Between any `glBegin()...glEnd()` pair, there is a restricted set of OpenGL commands that may be given. This includes `glVertex()` and also other command attribute commands, such as `glColor3f()`. At first it may seem a bit strange that you can assign different colors to the different vertices of an object, but this turns out to be a useful feature. Depending on the shading model, it allows you to produce shapes whose color blends from one color to another.

There are a number of drawing attributes other than color. For example, for points it is possible adjust their size (with `glPointSize()`). For lines, it is possible to adjust their width (with `glLineWidth()`), and create dashed or dotted lines (with `glLineStipple()`). It is also possible to pattern or stipple polygons (with `glPolygonStipple()`). When we discuss 3-dimensional graphics we will discuss many more properties that are used in shading and hidden surface removal.

After drawing the diamond, we change the color to blue, and then invoke `glRectf()` to draw a rectangle. This procedure takes four arguments, the (x, y) coordinates of any two opposite corners of the rectangle, in this case $(0.25, 0.25)$ and $(0.75, 0.75)$. (There are also versions of this command that takes double or int arguments, and vector arguments as well.) We could have drawn the rectangle by drawing a `GL_POLYGON`, but this form is easier to use.

Projection Transformation: In the simple drawing procedure, we said that we were assuming that the idealized drawing area was a unit square over the interval $[0, 1]$ with the origin in the lower left corner. The transformation that maps the idealized drawing region (in 2- or 3-dimensions) to the window is called the *projection*.

Generally, specifying 3-dimensional perspective projections is a tricky business, which we will discuss in detail later this semester. But for this simple 2-dimensional example, it is relatively simple. There is a transformation matrix called the *projection matrix*, which OpenGL maintains internally. (In the next lecture we will discuss OpenGL's rather complex transformation mechanism in greater detail. In the mean time some of this may seem a bit arcane.) Since matrices are often cumbersome to work with, OpenGL (actually Glu) provides a number of relatively simple and natural ways of defining this matrix. For our 2-dimensional example, we will do this by simply informing OpenGL of the rectangular region of two dimensional space that makes up our idealized drawing region. This is handled by the command

`gluOrtho2d(left, right, bottom, top)`.

First note that the prefix is "glu" and not "gl". Also, note that the "2d" designator stands for "2-dimensional", as does not indicate the argument types. There is a more general command, `glOrtho()`, for doing 3-dimensional orthogonal projections. All arguments are of type `GLdouble`. The arguments specify the x-coordinates (*left* and *right*) and the y-coordinates (*bottom* and *top*) of the rectangle into which we will be drawing. Any drawing that we do outside of this region will automatically be clipped away by OpenGL. The code to set the projection is given below.

Setting a Two-Dimensional Projection

```
glMatrixMode(GL_PROJECTION);                // set projection matrix
glLoadIdentity();                           // initialize to identity
gluOrtho2D(0.0, 1.0, 0.0, 1.0);             // map unit square to viewport
```

The first command tells OpenGL that we are modifying the projection transformation. (OpenGL maintains three different types of transformations, as we will see later.) Most of the commands that manipulate these matrices do so by multiplying some matrix times the current matrix. Thus, we initialize the current matrix to the identity, which is done by `glLoadIdentity()`. This code usually appears in some initialization procedure or possibly in the reshape callback.

Viewports: OpenGL does not assume that you are mapping your graphics to the entire window. Often you want to subdivide your window into a set of smaller subwindows and draw separate pictures in each window. The subwindow into which the current graphics are being drawn is called a *viewport.* The viewport is typically the entire display window, but it may generally be any rectangular subregion.

This quantity depends on the dimensions of our window. Thus, every time the window is resized (and this includes when the window is created originally) we should readjust the viewport. For example, the reshape callback would contain the following call if they entire window is the viewport.

Setting the Viewport in the Reshape Callback

```
void myReshape(int winWidth, int winHeight)       // reshape window
{
    ...
    glViewport (0, 0, winWidth, winHeight);       // reset viewport
    ...
}
```

The general form is

`glViewport(GLint x, GLint y, GLsizei width, GLsizei height),`

where (x, y) are the pixel coordinates of the lower-left corner of the viewport, and *width* and *height* are the width and height of the viewport in pixels.

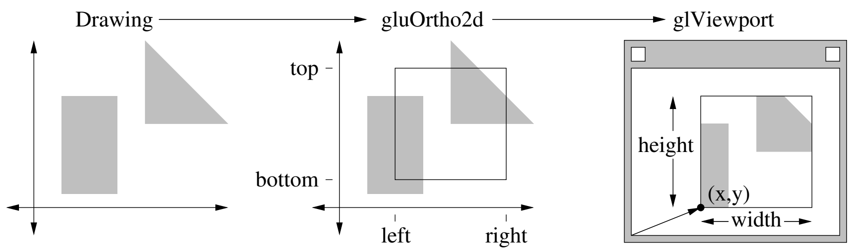


Figure 11: Projection and viewport transformations.

Lecture 5: Drawing in OpenGL: Transformations

(Thursday, Sep 13, 2000)

Read: Chapter 3 in Hill. Today's material is not really covered in the text (except for 3-d in Chapter 5).

More about Drawing: So far we have discussed how to draw simple 2-dimensional objects using OpenGL. Suppose that we want to draw more complex scenes. For example, we want to draw objects that move and rotate or to change the projection. We could do this by computing (ourselves) the coordinates of the transformed vertices, but OpenGL provides some tools to handle this automatically. Today we consider how this is done in 2-space. This will form a foundation for the more complex transformations, which will be needed for 3-dimensional viewing.

Transformations: Linear transformations are at the very heart of computer graphics. They arise in various ways.

(1) Moving rigid objects as part of an animation.

(2) Change of coordinate systems, which are used when objects stored relative to one reference frame are to be accessed in a different reference frame. One important case of this is that of mapping objects stored in a standard coordinate system to a coordinate system that is associated with the camera (or viewer).

(3) Transformations used to project objects from the idealized drawing window to the viewport, and mapping the viewport to the graphics display window.

(4) Transformations that indicate how textures are to be wrapped around objects, as part of texture mapping.

OpenGL maintains three sets of matrices for performing various transformation operations. These are:

Modelview matrix: Used for transforming objects in the scene and for changing the coordinates into a form that is easier for OpenGL to deal with. (Used for tasks (1) and (2) above).

Projection matrix: Handles parallel and perspective projections. (Used for task (3).)

Texture matrix: This is used in specifying how textures are mapped onto objects. (Used for task (4).)

We will discuss the texture matrix later in the semester, when we talk about texture mapping. There is one more transformation that is not handled by these matrices. This is the transformation that maps the viewport to the display. It is set by `glViewport()`.

For each matrix type, OpenGL maintains a *stack* of matrices. The *current matrix*, that is the one on the top of the stack, is the one that is applied at any given time. The stack mechanism allows you to save the current matrix (by pushing it on the stack) and restoring it later (by popping the stack). We will discuss the entire process of implementing affine and projection transformations later in the semester. For now, we'll give just basic information on OpenGL's approach to handling matrices and transformations.

OpenGL has a number of commands for handling matrices. In order to know which matrix (modelview, projection, or texture) to which an operation applies, you can set the current *matrix mode*. This is done with

`glMatrixMode(`*mode*`);`

where *mode* is either `GL_MODELVIEW`, `GL_PROJECTION`, or `GL_TEXTURE`. The initial mode is `GL_MODELVIEW`. Since this is the most common mode, the common convention is to assume that you are always in modelview mode, and if you want to modify either of the other two matrices, you first change the mode to the desired mode (projection or texture), perform whatever operations you want, and then change the mode back to modelview.

Once the matrix mode is set, you can perform various operations to the stack. (OpenGL has a somewhat funny way of handling its the stack. Note that most operations below (except `glPushMatrix()`) destroy the current matrix at the top of the stack.)

`glLoadIdentity()`: Sets the current matrix to the identity matrix.

`glLoadMatrix*(M)`: Loads (copies) a given matrix over the current matrix. The '*' refers to the coordinate type, which is either 'f' or 'd' for `GLfloat` or `GLdouble`.

glMultMatrix*(M): Multiplies the current matrix by a given matrix and replaces the current matrix with this result.

glPushMatrix(): Pushes a copy of the current matrix on top the stack.

glPopMatrix(): Pops the current matrix off the stack.

We will discuss how matrices like M are presented to OpenGL later in the semester. There are a number of other matrix operations, which we will also discuss later.

Applying transformations: How do you apply a transformation to a point? The answer is that it happens automatically. In particular, every vertex or geometric object generated by a call to `glVertex()` is passed through a series of matrices, as shown in the figure below. Thus, transformation behave much like drawing attributes: you set them, do some drawing, alter them, do more drawing, etc.

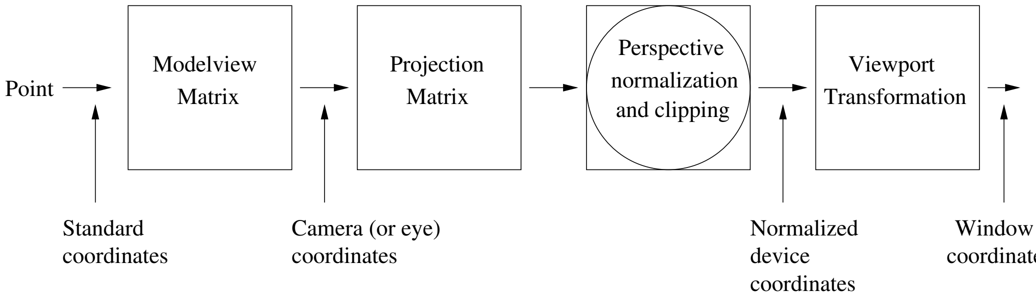


Figure 12: Transformation pipeline.

Modelview Transformations: The modelview matrix is useful for applying transformations to objects, which would otherwise require you to perform your own linear algebra. Suppose that rather than drawing a rectangle that is aligned with the coordinate axes, you want to draw a rectangle that is rotated by 10 degrees (counterclockwise) and centered at some point (x, y). You could compute the coordinates of the vertices yourself (using the built-in library routines for trigonometric functions), but OpenGL provides a way of doing this transformation more easily.

Suppose that we are drawing within the unit square, $0 \leq x, y \leq 1$. Suppose we have a 0.2×0.2 sized rectangle to be drawn centered at location (x, y). We could draw an unrotated rectangle with the following command:

```
glRectf(x - 0.1, y - 0.1, x + 0.1, y + 0.1);
```

Now let's draw a rotated rectangle. Let us assume that the matrix mode is modelview (this is the default). Because our rotation operation will destroy the modelview matrix, we will begin by saving it, by using the command `glPushMatrix()`. (Saving the modelview matrix in this manner is not always required, but it is considered good form.) Then we will compose this matrix with an appropriate rotation matrix. Then we draw the rectangle (in upright form). Since all points are transformed by the modelview matrix prior to projection, this will have the effect of rotating our rectangle. Finally, we will pop off this matrix (so future drawing is not rotated).

To perform the rotation, we will use the command `glRotatef(ang, x, y, z)`. All arguments are `GLfloat`'s. (Or you could use `glRotated()` which takes `GLdouble` arguments.) This command constructs a matrix which performs a rotation in 3-dimensional space counterclockwise by angle *ang* degrees, about the vector (x, y, z). It then *composes* (or multiplies) this matrix with the current modelview matrix. In our case the angle is 10 degrees. To achieve a rotation in the (x, y) plane the vector of rotation would be the z-unit vector, $(0, 0, 1)$. Here is how the code might look (but beware, this conceals a subtle error).

Drawing an Rotated Rectangle (First Attempt)

```
glPushMatrix();                          // save the current matrix
glRotatef(10.0, 0.0, 0.0, 1.0);          // rotate by 10 degrees CCW
glRectf(x-0.1, y-0.1, x+0.1, y+0.1);     // draw the rectangle
glPopMatrix();                           // restore the old matrix
```

The order of the rotation relative to the drawing command may seem confusing at first. You might think, "Shouldn't we draw the rectangle first and then rotate it?". The key is to remember that whenever you draw (using `glRectf()` or `glBegin()...glEnd()`), the points are automatically transformed using the modelview matrix. So, in order to rotate, we first modify the modelview matrix, then draw the points (hence applying the rotation automatically). Popping the matrix at the end is important, otherwise future drawing requests would also be subject to the same modelview matrix and the same rotation.

Although this may seem backwards, it is the way in which almost all object transformations are performed in OpenGL:

(1) Push the matrix stack,

(2) Apply (i.e., multiply) all the desired transformation matrices with the current matrix,

(3) Draw your object (the transformations will be applied automatically), and

(4) Pop the matrix stack.

But something is wrong with this example given above. What is it? The answer is that the rotation is performed about the origin of the coordinate system, not about the center of the rectangle and we want. Fortunately, there is an easy fix. Conceptually, we will draw the rectangle centered at the origin, then rotate it by 10 degrees, and finally *translate* (or move) it by the vector (x, y). To do this, we will need to use the command `glTranslatef(x, y, z)`. All three arguments are `GLfloat`'s. (And there is version with `GLdouble` arguments.) This command creates a matrix which performs a translation by the vector (x, y, z), and then composes (or multiplies) it with the current matrix. Recalling that all 2-dimensional graphics occurs in the $z = 0$ plane, the desired translation vector is $(x, y, 0)$.

So the conceptual order is (1) draw, (2) rotate, (3) translate. But remember that you need to set up the transformation matrix *before* you do any drawing. That is, if $\vec{v}$ represents a vertex of the rectangle, and R is the rotation matrix and T is the translation matrix, and M is the current modelview matrix, then we want to compute the product

$$M(T(R(\vec{v}))) = M \cdot T \cdot R \cdot \vec{v}.$$

Since M is on the top of the stack, we need to first apply translation (T) to M, and then apply rotation (R) to the result, and then do the drawing ($\vec{v}$). Note that the order of application is the exact *reverse* from the conceptual order. This may seems confusing (and it is). But it is easy to remember.

OpenGL's Backwards Transformation Rule: Conceptualize your transformation as first drawing about the origin and then transforming your object to its desired location. Then implement this in OpenGL by applying these transformations and drawing in reverse order.

Drawing an Rotated Rectangle (Correct)

```
glPushMatrix();                       // save the current matrix (M)
glTranslatef(x, y, 0);                // apply translation (T)
glRotatef(10.0, 0.0, 0.0, 1.0);       // apply rotation (R)
glRectf(-0.1, -0.1, 0.1, 0.1);        // draw rectangle at the origin
glPopMatrix();                        // restore the old matrix (M)
```

Projection Revisited: Last time we discussed the use of `gluOrtho2D()` for doing simple 2-dimensional projection. This call does not really do any projection. Rather, it computes the desired projection transformation and multiplies it times whatever is on top of the current matrix stack. So, to use this we need to do a few things. First, set the matrix mode to `GL_PROJECTION`, load an identity matrix (just for safety), and the call `gluOrtho2d()`. Since most transformations are done in modelview mode, we will set the mode back.

Two Dimensional Projection

```
glMatrixMode(GL_PROJECTION);              // set projection matrix
glLoadIdentity();                         // initialize to identity
gluOrtho2D(left, right, bottom top);      // set the drawing area
glMatrixMode(GL_MODELVIEW);               // restore modelview mode
```

If you only set the projection once, then initializing the matrix to the identity is typically redundant (since this is the default value), but it is a good idea to make a habit of loading the identity for safety. If the projection does not change throughout the execution of our program, and so we include this code in our initializations. It might be put in the reshape callback if reshaping the window alters the projection.

How is it done: How does `gluOrtho2D()` and `glViewport()` set up the desired transformation from the idealized drawing window to the viewport? Well, actually OpenGL does this in two steps, first mapping from the window to canonical 2×2 window centered about the origin, and then mapping this canonical window to the viewport. The reason for this intermediate mapping is that the clipping algorithms are designed to operate on this fixed sized window (recall the figure given earlier). The intermediate coordinates are often called *normalized device coordinates*.

As an exercise in deriving linear transformation, let us consider doing this all in one shot. Let W denote the idealized drawing window and let V denote the viewport. Let W_r, W_l, W_b, and W_t denote the left, right, bottom and top of the window (and similarly for V). We wish to derive a linear transformation that maps a point (x, y) in window coordinates to a point (x', y') in viewport coordinates.

The book describes one way of doing this in Chapter 3, so I'll do it in an entirely different way. Let $f(x, y)$ denote this function. Since the function is linear, and clearly it operates on x and y independently, clearly

$$(x', y') = f(x, y) = (ax + c, dy + e),$$

where a, c, d and e, depend on the window and viewport coordinates. Let's derive what a and c are using simultaneous equations. We know that the x-coordinates for the left and right sides of the window (W_l and W_r) should map to the left and right sides of the viewport (V_l and V_r). Thus we have

$$aW_l + c = V_l \qquad aW_r + c = V_r.$$

We can solve these equations simultaneously. By subtracting them to eliminate c we have

$$a = \frac{V_r - V_l}{W_r - W_l}.$$

Plugging this back into to either equation and solving for c we have

$$c = V_l - aW_l$$

A similar derivation for d and e yields

$$d = \frac{V_t - V_b}{W_t - W_b} \qquad e = V_b - dW_b$$

Lecture 6: Geometry and Geometric Programming

(Tuesday, Sep 19, 2000)
Read: Chapter 4 in Hill.

Geometric Programming: There are many areas of computer science that involve computation with geometric entities. This includes not only computer graphics, but also areas like computer-aided design, robotics, computer vision, and geographic information systems. In this and the next few lectures we will consider how this can be done, and how to do this in a reasonably clean and painless way.

Computer graphics deals largely with the geometry of lines and linear objects in 3-space, because light travels in straight lines. For example, here are some typical geometric problems that arise in designing programs for computer graphics.

Geometric Intersections: Given a cube and a ray, does the ray strike the cube? If so which face? If the ray is reflected off of the face, what is the direction of the reflection ray?

Orientation: Three noncollinear points in 3-space define a unique plane. Given a fourth point q, is it above, below, or on this plane?

Transformation: Given unit cube, what are the coordinates of its vertices after rotating it 30 degrees about the vector $(1, 2, 1)$.

Change of coordinates: A cube is represented relative to some standard coordinate system. What are its coordinates relative to a different coordinate system (say, one centered at the camera's location)?

Coordinate-free programming: If you look at almost any text on computer graphics (ours included) you will find that the section on geometric computing begins by introducing coordinates, then vectors, then matrices. Then what follows are many long formulas involving many 4×4 matrices. These formulas are handy, because (along with some procedures for matrix multiplication) we can solve many problems in computer graphics. Unfortunately, from the perspective of software design they are a nightmare, because the intention of the programmer has been lost in all the matrix crunching. The product of a matrix and a vector can have many meanings. It may represent a change of coordinate systems, it may represent a transformation of space, and it may represent a perspective projection.

We will attempt to develop a clean, systematic way of thinking about geometric computations. This method is called *coordinate-free programming* (so named by Tony DeRose, its developer). Rather than reducing all computations to vector-matrix products, we will express geometric computations in the form of high-level geometric operations. These in turn will be implemented using low-level matrix computations, but if you use a good object-oriented programming language (such as C++ or Java) these details are hidden. Henceforth, when the urge to write down an expression involving point coordinates comes to you, ask yourself whether it is possible to describe this operation in a high-level coordinate-free form.

Ideally, this should be the job of a good graphics API. Indeed, OpenGL does provide the some support for geometric operations. For example, it provides procedures for performing basic affine transformations. Unfortunately, a user of OpenGL is still very aware of underlying presence of vectors and matrices in programming. A really well designed API would allow us to conceptualize geometry on a higher level.

Geometries: Before beginning we should discuss a little history. Geometry is one of the oldest (if not the oldest) branches of mathematics. Its origins were in land surveying (and hence its name: geo=earth, and metria=measure). Surveying became an important problem as the advent of agriculture required some way of defining the boundaries between one family's plot and anothers.

Ancient civilizations (the Egyptians, for example) must have possessed a fairly sophisticated understanding of geometry in order to build complex structures like the pyramids. However, it was not until much later in the time of Euclid in Greece in the 3rd century BC, that the mathematical field of geometry was first axiomatized and made formal. Euclid worked without the use of a coordinate system. It was much later in the 17th century when cartesian coordinates were developed (by Descartes), which allowed geometric concepts to be expressed arithmetically.

In the late 19th century a revolutionary shift occured in people's view of geometry (and mathematics in general). Up to this time, no one questioned that there is but one geometry, namely the Euclidean geometry. Mathematicians like Lobachevski and Gauss, suggested that there may be other geometric systems which are just as consistent and valid as Euclidean geometry, but in which different axioms apply. These are called *noneuclidean geometries*, and they played an important role in Einstein's theory of relativity.

We will discuss three basic geometric systems: affine geometry, Euclidean geometry, and projective geometry. Affine geometry is the most basic of these. Euclidean geometry builds on affine geometry by adding the concepts of angles and distances. Projective geometry is more complex still, but it will be needed in performing perspective projections.

Affine Geometry: The basic elements of *affine geometry* are *scalars* (which we can just think of as being real numbers), *points* and *free vectors* (or simply *vectors*). Points are used to specify position. Free vectors are used to specify direction and magnitude, but have no fixed position. The term "free" means that vectors do not necessarily emanate from some position (like the origin), but float freely about in space. There is a special vector called the *zero vector*, $\vec{0}$, that has no magnitude, such that $\vec{v} + \vec{0} = \vec{0} + \vec{v} = \vec{v}$. Note in particular that we did not define a *zero point* or "origin" for affine space. (Although we will eventually have to break down and define something like this in order, simply to be able to define coordinates for our points.)

You might ask, why make a distinction between points and vectors? Both can be represented in the same way as a list of coordinates. The reason is to avoid hiding the intention of the programmer. For example, it makes perfect sense to multiply a vector and a scalar (we stretch the vector by this amount). It is not so clear that it makes sense to multiply a point by a scalar. By keeping these concepts separate, we make it possible to check the validity of geometric operations.

We will use the following notational conventions. Points will be denotes with upper-case Roman letters (e.g., P, Q, and R), vectors will be denoted with lower-case Roman letters (e.g., u, v, and w) and often to emphasize this we will add an arrow (e.g., $\vec{u}$, $\vec{v}$, $\vec{w}$), and scalars will be represented as lower case Greek letters (e.g., α, β, γ). In our programs scalars will be translated to Roman (e.g., a, b, c).

The table below lists the valid combinations of these entities. The formal definitions are pretty much what you would expect. Vector operations are applied in the same way that you learned in linear algebra. For example, vectors are added in the usual "tail-to-head" manner. The difference $P - Q$ of two points results in a free vector directed from Q to P. Point-vector addition $R + \vec{v}$ is defined to be the translation of R by displacement $\vec{v}$. Note that some operations (e.g. scalar-point multiplication, and addition of points) are explicitly not defined.

$vector \leftarrow scalar \cdot vector$, $vector \leftarrow vector/scalar$	scalar-vector multiplication
$vector \leftarrow vector + vector$, $vector \leftarrow vector - vector$	vector-vector addition
$vector \leftarrow point - point$	point-point difference
$point \leftarrow point + vector$, $point \leftarrow point - vector$	point-vector addition

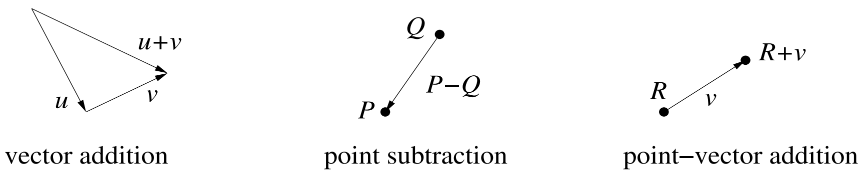


Figure 13: Affine operations.

Affine Combinations: Although the algebra of affine geometry has been careful to disallow point addition and scalar multiplication of points, there is a particular combination of two points that we will consider legal. The operation is called an *affine combination.*

Let's say that we have two points P and Q and want to compute their midpoint R, or more generally a point R that subdivides the line segment $\overline{PQ}$ into the proportions α and $1 - \alpha$, for some $\alpha \in [0, 1]$. (The case $\alpha = 1/2$ is the case of the midpoint). This could be done by taking the vector $Q - P$, scaling it by α, and then adding the result to P. That is,

$$R = P + \alpha(Q - P).$$

Another way to think of this point R is as a *weighted average* of the endpoints P and Q. Thinking of R in these terms, we might be tempted to rewrite the above formula in the following (illegal) manner:

$$R = (1 - \alpha)P + \alpha Q.$$

Observe that as α ranges from 0 to 1, the point R ranges along the line segment from P to Q. In fact, we may allow to become negative in which case R lies to the left of P (see the figure), and if $\alpha > 1$, then R lies to the right of Q. The special case when $0 \le \alpha \le 1$, this is called a *convex combination.*

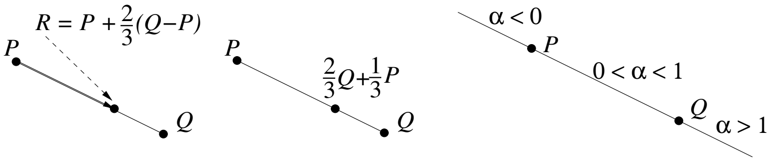


Figure 14: Affine combinations.

In general, we define the following two operations for points in affine space.

Affine combination: Given a sequence of points $P_1, P_2, \ldots, P_n$, an affine combination is any sum of the form

$$\alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n,$$

where $\alpha_1, \alpha_2, \ldots, \alpha_n$ are scalars satisfying $\sum_i \alpha_i = 1$.

Convex combination: Is an affine combination, where in addition we have $\alpha_i \geq 0$ for $1 \leq i \leq n$.

Affine and convex combinations have a number of nice uses in graphics. For example, any three noncollinear points determine a plane. There is a 1–1 correspondence between the points on this plane and the affine combinations of these three points. Similarly, there is a 1–1 correspondence between the points in the triangle determined by the these points and the convex combinations of the points. In particular, the point $(1/3)P + (1/3)Q + (1/3)R$ is the *centroid* of the triangle.

We will sometimes be sloppy, and write expressions of the following sort (which is clearly illegal).

$$R = \frac{P+Q}{2}.$$

We will allow this sort of abuse of notation provided that it is clear that there is a legal affine combination that underlies this operation.

To see whether you understand the notation, consider the following questions. Given three points in the 3-space, what is the union of all their affine combinations? (Ans: the plane containing the 3 points.) What is the union of all their convex combinations? (Ans: The triangle defined by the three points and its interior.)

Euclidean Geometry: In affine geometry we have provided no way to talk about angles or distances. Euclidean geometry is an extension of affine geometry which includes one additional operation, called the *inner product*.

The inner product is an operator that maps two vectors to a scalar. The product of $\vec{u}$ and $\vec{v}$ is denoted commonly denoted $(\vec{u}, \vec{v})$. There are many ways of defining the inner product, but any legal definition should satisfy the following requirements

Positiveness: $(\vec{u}, \vec{u}) \geq 0$ and $(\vec{u}, \vec{u}) = 0$ if and only if $\vec{u} = \vec{0}$.

Symmetry: $(\vec{u}, \vec{v}) = (\vec{v}, \vec{u})$.

Bilinearity: $(\vec{u}, \vec{v} + \vec{w}) = (\vec{u}, \vec{v}) + (\vec{u}, \vec{w})$, and $(\vec{u}, \alpha\vec{v}) = \alpha(\vec{u}, \vec{v})$. (Notice that the symmetric forms follow by symmetry.)

See a book on linear algebra for more information. We will focus on a the most familiar inner product, called the *dot product*. To define this, we will need to get our hands dirty with coordinates. Suppose that the d-dimensional vector $\vec{u}$ is represented by the coordinate vector $(u_0, u_1, \ldots, u_{d-1})$. Then define

$$\vec{u} \cdot \vec{v} = \sum_{i=0}^{d-1} u_i v_i,$$

Note that inner (and hence dot) product is defined only for vectors, not for points.

Using the dot product we may define a number of concepts, which are not defined in regular affine geometry. Note that these concepts generalize to all dimensions.

Length: of a vector $\vec{v}$ is defined to be $|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}$.

Normalization: Given any nonzero vector $\vec{v}$, define the *normalization* to be a vector of unit length that points in the same direction as $\vec{v}$. We will denote this by $\hat{v}$:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}.$$

Distance between points: $\mathrm{dist}(P, Q) = |P - Q|$.

Angle: between two nonzero vectors $\vec{u}$ and $\vec{v}$ (ranging from 0 to π) is

$$\mathrm{ang}(\vec{u}, \vec{v}) = \cos^{-1}\left(\frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}\right) = \cos^{-1}(\hat{u} \cdot \hat{v}).$$

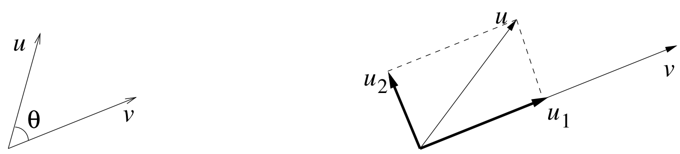
This is easy to derive from the law of cosines.

Orthogonality: $\vec{u}$ and $\vec{v}$ are *orthogonal* (or perpendicular) if $\vec{u} \cdot \vec{v} = 0$.

Orthogonal projection: Given a vector $\vec{u}$ and a nonzero vector $\vec{v}$, it is often convenient to decompose $\vec{u}$ into the sum of two vectors $\vec{u} = \vec{u}_1 + \vec{u}_2$, such that $\vec{u}_1$ is parallel to $\vec{v}$ and $\vec{u}_2$ is orthogonal to $\vec{v}$.

$$\vec{u}_1 = \frac{(\vec{u} \cdot \vec{v})}{(\vec{v} \cdot \vec{v})}\vec{v} \qquad \vec{u}_2 = \vec{u} - \vec{u}_1.$$

(As an exercise, verify that $\vec{u}_2$ is orthogonal to $\vec{v}$.) Note that we can ignore the denominator if we know that $\vec{v}$ is already normalized to unit length. The vector $\vec{u}_1$ is called the *orthogonal projection* of $\vec{u}$ onto $\vec{v}$.



Angle between vectors Orthogonal projection

Figure 15: The dot product and its uses.

Lecture 7: Coordinate Frames and Homogeneous Coordinates

(Thursday, Sep 21, 2000)
Read: Chapter 5 in Hill.

Coordinate Frames and Coordinates: Last time we presented the basic elements of affine and Euclidean geometry: points, vectors, and operations such as affine combinations. However, as of yet we have no mechanism for defining these objects. Today we consider the lower level issues of how these objects are represented using coordinate frames and homogeneous coordinates.

The first question is how to represent points and vectors in affine space. We will begin by recalling how to do this in linear algebra, and generalize from there. We will assume familiarity with concepts from linear algebra. (If any of this seems unfamiliar, please consult any text in linear algebra.) We know from linear algebra that if we have 3-linearly independent vectors, $\vec{u_0}$, $\vec{u_1}$, and $\vec{u_2}$, in 3-space, then we can represent any other vector in 3-space uniquely as a *linear combination*

$$\vec{v} = \alpha_0\vec{u}_0 + \alpha_1\vec{u}_1 + \alpha_2\vec{u}_2,$$

for some choice of scalars α_0, α_1, α_2. Thus, given any such vectors, we can use them to represent any vector in terms of a triple of scalars $(\alpha_0, \alpha_1, \alpha_2)$. In general d linearly independent vectors in dimension d is called a *basis*.

The most familiar basis, called the *standard basis*, is composed of the three *unit vectors* whose coordinates are $(1,0,0)^T$, $(0,1,0)^T$, $(0,0,1)^T$. Recall from matrix algebra that the superscript T indicates matrix transpose, which means that these are to be thought of a column vectors, rather than row vectors. With only one exception this semester (if and when we discuss normals) all vectors will be represented as column vectors, so if the T is missing, it is probably just due to sloppiness. In this case the column vector $(\alpha_0, \alpha_1, \alpha_2)^T$ is called the *Cartesian coordinates* of the vector $\vec{v}$. These vectors have the nice property of being of length 1 and are all mutually orthogonal. Such a basis is called an *orthonormal basis*, and these are generally most popular.

To define a coordinate frame for an affine space we would like to find some way to represent any object (point or vector) as a sequence of scalars. Thus, it seems natural to generalize the notion of a basis in linear algebra to define a basis in affine space. Note that free vectors alone are not enough to define a point (since we cannot define a point by any combination of vector operations). To specify position, we will designate an arbitrary a point, denoted $\mathcal{O}$, to serve as the *origin* of our coordinate frame. Observe that for any point P, $P - \mathcal{O}$ is just some vector $\vec{v}$. Such a vector can be expressed uniquely as a linear combination of basis vectors. Thus, given the origin point $\mathcal{O}$ and any set of basis vectors $\vec{u}_i$, any point P can be expressed uniquely as a sum of $\mathcal{O}$ and some linear combination of the basis vectors:

$$P = \alpha_0\vec{u}_0 + \alpha_1\vec{u}_1 + \alpha_2\vec{u}_2 + \mathcal{O},$$

for some sequence of scalars α_0, α_1, α_2. This is how we will define a coordinate frame for affine spaces. In general we have:

Definition: A *coordinate frame* for a d-dimensional affine space consists of a point, called the *origin* (which we will denote $\mathcal{O}$) of the frame, and a set of d linearly independent *basis vectors*.

In the figure below we show a point P and vector $\vec{w}$. We have also given two coordinate frames, F and G. Observe that P and $\vec{w}$ can be expressed as functions of F and G as follows:

$$\begin{aligned} P &= 3 \cdot F.\vec{e}_0 + 2 \cdot F.\vec{e}_1 + F.\mathcal{O} \\ \vec{w} &= 2 \cdot F.\vec{e}_0 + 1 \cdot F.\vec{e}_1 \\ \\ P &= 1 \cdot G.\vec{e}_0 + 2 \cdot G.\vec{e}_1 + G.\mathcal{O} \\ \vec{w} &= -1 \cdot G.\vec{e}_0 + 0 \cdot G.\vec{e}_1 \end{aligned}$$

Notice that the position of $\vec{w}$ is immaterial, because in affine geometry vectors are free to float where they like.

The Coordinate Axiom and Homogeneous Coordinates: Recall that our goal was to represent both points and vectors as a list of scalar values. To put this on somewhat more formal footing, we introduce the following axiom.

Coordinate Axiom: For every point P in affine space, $0 \cdot P = \vec{0}$, and $1 \cdot P = P$.

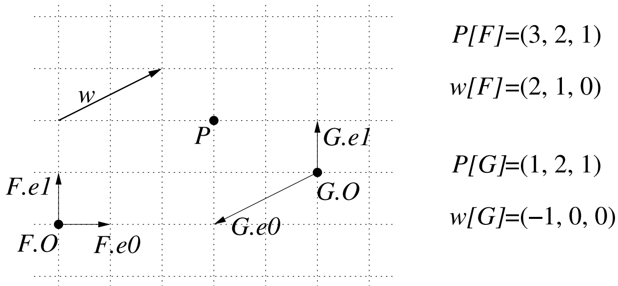


Figure 16: Coordinate Frame.

This is a serious perversion of our rules for affine geometry, but it is allowed just to make the notation easier to understand. Using this notation, we can now write the point and vector of the figure in the following way.

$$\begin{aligned} P &= 3 \cdot F.\vec{e}_0 + 2 \cdot F.\vec{e}_1 + 1 \cdot F.\mathcal{O} \\ \vec{w} &= 2 \cdot F.\vec{e}_0 + 1 \cdot F.\vec{e}_1 + 0 \cdot F.\mathcal{O} \end{aligned}$$

Thus, relative the the coordinate frame $F = \langle F.\vec{e}_0, F.\vec{e}_1, F.\mathcal{O}\rangle$, we can express P and $\vec{w}$ as coordinate vectors relative to frame F as

$$P[F] = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \qquad \text{and} \qquad \vec{w}[F] = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}.$$

We will call these *homogeneous coordinates*. In some linear algebra conventions, vectors are written as row vectors and some as column vectors. We will stick with OpenGL's conventions, of using column vectors, but we may be sloppy from time to time.

Beware: The term "vector" has two meanings: one as an *free vector* in an affine space, and now as a *coordinate vector*, that is, a list of scalar values. The first is a geometric object, and the second is a representation that is used for both free-vectors and for points. Usually, it will be clear from context which meaning is intended.

In general, to represent points and vectors in d-space, we will use coordinate vectors of length $d + 1$. Points have a last coordinate of 1, and vectors have a last coordinate of 0. This representation is called the *homogeneous coordinates* of a point or vector, relative to the frame F. (Some authors put the homogenizing coordinate first rather than last. There are actually good reasons for doing this. But we will stick with standard engineering conventions and place it last.)

Properties of homogeneous coordinates: The choice of appending a 1 for points and a 0 for vectors may seem to be a rather arbitrary choice. Why not just reverse them or use some other scalar values? The reason is that this particular choice has a number of nice properties with respect to geometric operations.

For example, consider two points P and Q whose coordinate representations relative to some frame F are $P[F] = (3, 2, 1)^T$ and $Q[F] = (5, 1, 1)^T$, respectively. Consider the vector

$$\vec{v} = P - Q.$$

If we apply the difference rule that we defined last time for points, and then convert this vector into it coordinates relative to frame F, we find that $\vec{v}[F] = (-2, 1, 0)^T$. Thus, to compute the

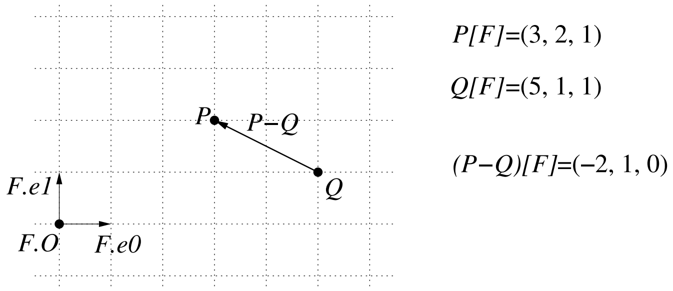


Figure 17: Coordinate arithmetic.

coordinates of $P - Q$ we simply take the component-wise difference of the coordinate vectors for P and Q. The 1-components nicely cancel out, to give a vector result.

In general, a nice feature of this representation is the last coordinate behaves exactly as it should. Let U and V be either points or vectors. After a number of operations of the forms $U + V$ or $U - V$ or αU (when applied to the coordinates) we have:

- If the last coordinate is 1, then the result is a *point*.
- If the last coordinate is 0, then the result is a *vector*.
- Otherwise, this is not a legal affine operation.

This fact can be proved rigorously, but we won't worry about doing so.

This suggests how one might do type checking for a coordinate-free geometry system. Points and vectors are stored using a common base type, which simply consists of a 4-element array of scalars. We allow the programmer to perform any combination of standard vector operations on coordinates. Just prior to assignment, check that the last coordinate is either 0 or 1, appropriate to the type of variable into which you are storing the result. This allows much more flexibility in creating expressions, such as:

$$\mathit{centroid} \leftarrow \frac{P + Q + R}{3},$$

which would otherwise fail type checking. (Unfortunately, this places the burden of checking on the run-time system. One approach is to define the run-time system so that type checking can be turned on and off. Leave it on when debugging and turn it off for the final version.)

Alternative coordinate frames: Any geometric programming system must deal with two conflicting goals. First, we want points and vectors to be represented with respect to some *universal coordinate frame* (so we can operate on points and vectors by just operating on their coordinate lists). But it is often desirable to define points relative to some convenient *local coordinate frame*. For example, latitude and longitude may be a fine way to represent the location of a city, but it is not a very convenient way to represent the location of a character on this page.

What is the most universal coordinate frame? There is nothing intrinsic to affine geometry that will allow us to define such a thing, so we will do so simply by convention. We will fix a frame called the *standard frame* from which all other objects will be defined. It will be an *orthonormal frame*, meaning that its basis vectors are orthogonal to each other and each is of unit length. We will denote the origin by $\mathcal{O}$ and the basis vectors by $\vec{e}_i$. The coordinates of the elements of the standard frame (in 3-space) are defined to be:

$$\vec{e}_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \vec{e}_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad \vec{e}_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \qquad \mathcal{O} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Change of coordinates (example): One of the most important geometric operations in computer graphics is that of converting points and vectors from one coordinate frame to another. Recall from the earlier figure that relative to frame F we have $P[F] = (3,2,1)^T$, and $\vec{w}[F] = (2,1,0)^T$. We derived the coordinates relative to frame G by inspection, but how could we do this computationally? Our goal is to find scalars β_0, β_1, β_2, such that $P = \beta_0 G.e_0 + \beta_1 G.e_1 + \beta_2 G.\mathcal{O}$.

Given that F is a frame, we can describe the elements of G in terms of of F. If we do so we have $G.e_0[F] = (-2,-1,0)^T$, $G.e_1[F] = (0,1,0)^T$, and $G.\mathcal{O}[F] = (5,1,1)^T$. Using this representation, it follows that β_0, β_1, and β_2 must satisfy

$$\begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} = \beta_0 \begin{pmatrix} -2 \\ -1 \\ 0 \end{pmatrix} + \beta_1 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + \beta_2 \begin{pmatrix} 5 \\ 1 \\ 1 \end{pmatrix}.$$

If you break this vector equation into its three components, you get three equations, and three unknowns. If you solve this system of equations (by methods that you learned in linear algebra) then you find that $(\beta_0, \beta_1, \beta_2) = (1,2,1)$. Hence we have

$$\begin{aligned} P[F] &= \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} &= 1 \begin{pmatrix} -2 \\ -1 \\ 0 \end{pmatrix} + 2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + 1 \begin{pmatrix} 5 \\ 1 \\ 1 \end{pmatrix} \\ & &= 1 \cdot G.e_0[F] + 2 \cdot G.e_1[F] + 1 \cdot G.\mathcal{O}[F]. \end{aligned}$$

Therefore, the coordinates of P relative to G are

$$P[G] = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}.$$

As an exercise, see whether you can derive the fact that the coordinates for $\vec{w}$ are $(-1,0,0)^T$.

Change of coordinates (general case): We would like to generalize this for an arbitrary pair of frames. For concreteness, let us assume that F is the standard frame, and suppose that we define G relative to this standard frame by giving the coordinates for the basis vectors $G.e_0$, $G.e_1$ and origin point $G.\mathcal{O}$ relative to frame F:

$$\begin{aligned} G.e_0[F] &= (g_{00}, g_{01}, 0)^T, \\ G.e_1[F] &= (g_{10}, g_{11}, 0)^T, \\ G.\mathcal{O}[F] &= (g_{20}, g_{21}, 1)^T. \end{aligned}$$

Further suppose that we know the coordinate of some point P relative to F, namely $P[F] = (\alpha_0, \alpha_1, \alpha_2)^T$. We know that $\alpha_2 = 1$ since P is a point, but we will leave it as a variable to get an expression that works for free vectors as well.

Our goal is to determine $P[G] = (\beta_0, \beta_1, \beta_2)^T$. Therefore the β values must satisfy:

$$P = \beta_0 G.e_0 + \beta_1 G.e_2 + \beta_2 G.\mathcal{O}.$$

This is an expression in affine geometry. If we express this in terms of F's coordinate frame we have

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{pmatrix} = \beta_0 \begin{pmatrix} g_{00} \\ g_{01} \\ 0 \end{pmatrix} + \beta_1 \begin{pmatrix} g_{10} \\ g_{11} \\ 0 \end{pmatrix} + \beta_2 \begin{pmatrix} g_{20} \\ g_{21} \\ 1 \end{pmatrix} = \begin{pmatrix} g_{00} & g_{10} & g_{20} \\ g_{01} & g_{11} & g_{21} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}.$$

Let M denote the 3×3 matrix above. Note that its columns are the basis elements for G, expressed as coordinate vectors in terms of F.

$$M = \begin{pmatrix} g_{00} & g_{10} & g_{20} \\ g_{01} & g_{11} & g_{21} \\ 0 & 0 & 1 \end{pmatrix} = \left(G.\vec{e}_0[F] \,\middle|\, G.\vec{e}_1[F] \,\middle|\, G.\mathcal{O}[F] \right).$$

Thus, given $P[G] = (\beta_0, \beta_1, \beta_2)^T$ we can multiply this matrix by $P[G]$ to get $P[F] = (\alpha_0, \alpha_1, \alpha_2)^T$.

$$P[F] = M \cdot P[G]$$

But this is not what we wanted. We wanted to get $P[G]$ in terms of $P[F]$. To do this we compute the inverse of M, denoted M^{-1}. We claim that if this is a valid basis (that is, if the basis vectors are linearly independent) then this inverse will exist. Hence we have

$$P[G] = M^{-1} \cdot P[F].$$

In the case of this simple 3×3, this inverse is easy to compute. However, when we will be applying this, we will normally be operating in 3-space, and the matrices will now be 4×4 matrices and the inversion is more involved.

Important Warning: OpenGL stores matrices in *column-major order*. This means that elements of a 4×4 matrix are stored by unraveling them column-by-column.

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

Unfortunately, C and C++ (and most other programming languages other than Fortran) store matrices in *row-major order*. Consequently, if you declare a matrix to be used, say, in `glLoadMatrix()` you might use

```
GLdouble M[4][4];
```

But to access the element in row i and column j, then you need to refer to it by `M[j][i]` (not `M[i][j]` as you normally would). Alternatively, you can declare it `GLdouble M[16]` and perform your own indexing.

Lecture 8: Affine Transformations

(Tuesday, Sep 26, 2000)
Read: Chapter 5 in Hill.

Affine Transformations: So far we have been stepping through the basic elements of geometric programming. We have discussed points, vectors, and their operations, and coordinate frames and how to change the representation of points and vectors from one frame to another. Our next topic involves how to map points from one place to another. Suppose you want to draw an animation of a spinning ball. How would you define the function that maps each point on the ball to its position rotated through some given angle?

We will consider a limited, but interesting class of transformations, called *affine transformations*. These include (among others) the following transformations of space: translations, rotations, uniform and nonuniform scalings (stretching the axes by some constant scale factor), reflections (flipping objects about a line) and shearings (which deform squares into parallelograms). They are illustrated in the figure below.

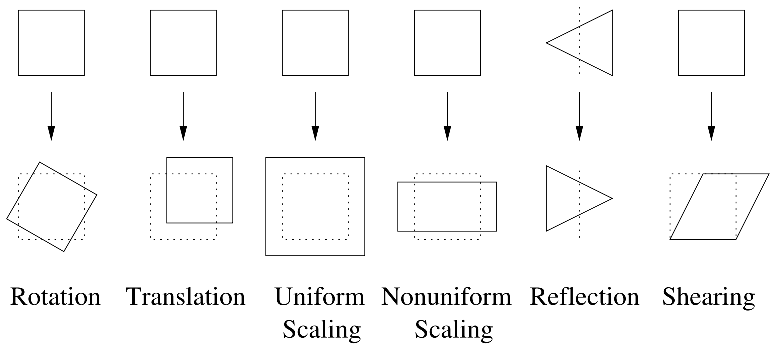


Figure 18: Examples of affine transformations.

These transformations all have a number of things in common. For example, they all map lines to lines. Note that some (translation, rotation, reflection) preserve the lengths of line segments and the angles between segments. Others (like uniform scaling) preserve angles but not lengths. Others (like nonuniform scaling and shearing) do not preserve angles or lengths.

All of the transformation listed above preserve basic affine relationships. (In fact, this is the definition of an affine transformation.) For example, given any transformation T of one of the above varieties, and given two points P and Q, and any scalar α,

$$R = (1-\alpha)P + \alpha Q \qquad \Rightarrow \qquad T(R) = (1-\alpha)T(P) + \alpha T(Q).$$

(We will leave the proof that each of the above transformations is affine as an exercise.) Putting this more intuitively, if R is the midpoint of segment $\overline{PQ}$, before applying the transformation, then it is the midpoint after translation.

Matrix Representation of Affine Transformations: Perhaps a more important consequence of the preservation of affine relations is the following:

$$R = \alpha_0 F.\vec{e}_0 + \alpha_1 F.\vec{e}_1 + \alpha_2 \mathcal{O} \qquad \Rightarrow \qquad T(R) = \alpha_0 T(F.\vec{e}_0) + \alpha_1 T(F.\vec{e}_1) + \alpha_2 T(\mathcal{O}).$$

The equation on the left is how we represent the point (or vector) R in terms of the coordinate frame F. This implication shows that if we know the image of the frame elements under the transformation, then we know the image R under the transformation. Alternatively, if we express R as a homogeneous coordinate column vector $R[F] = (\alpha_0, \alpha_1, \alpha_2)^T$. (Recall that the superscript T means to transpose this row vector into a column vector.) Then we may write the above relationship in matrix form

$$T(R)[F] = \left(T(F.\vec{e}_0)[F] \,\middle|\, T(F.\vec{e}_1)[F] \,\middle|\, T(F.\mathcal{O})[F] \right) \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{pmatrix}.$$

Here the columns of the array are the representation (relative to F) of the images of the elements of the frame under T. This implies that applying an affine transformation (in coordinate form) is equivalent to multiplying the coordinates by a matrix. In dimension d this is a $(d+1) \times (d+1)$ matrix.

If this all seems a bit abstract, here are some concrete applications of this to the basic affine transformations described above. Rather than considering this in the context of 2-dimensional transformations, let's consider it in the more general setting of 3-dimensional transformations. The two dimensional cases can be extracted by just ignoring the rows and columns for the z-coordinates.

Translation: Translation by a fixed vector $\vec{v}$ maps any point P to $P + \vec{v}$. Note that vectors are not altered by translation. (Why not?)

Suppose that relative to the standard frame, $v[F] = (\alpha_x, \alpha_y, \alpha_z, 0)^T$ are the homogeneous coordinates of $\vec{v}$. The three unit vectors are unaffected by translation, and the origin is mapped to $\mathcal{O} + \vec{v}$, whose homogeneous coordinates are $(\alpha_x, \alpha_y, \alpha_z, 1)$. Thus, by the rule given earlier, the homogeneous matrix representation for this translation transformation is

$$T(\vec{v}) = \begin{pmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Scaling: *Uniform scaling* is a transformation which is performed relative to some central fixed point. We will assume that this point is the origin of the standard coordinate frame. Given a scalar β, this transformation maps the object (point or vector) with coordinates $(\alpha_x, \alpha_y, \alpha_z, \alpha_w)^T$ to $(\beta\alpha_x, \beta\alpha_y, \beta\alpha_z, \alpha_w)^T$.

In general, it is possible to specify separate scaling factors for each of the axes. This is called *nonuniform scaling*. The unit vectors are each stretched by the corresponding scaling factor, and the origin is unmoved. Thus, the transformation matrix has the following form:

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

Both points and vectors are scaled.

Reflection: A reflection in the plane is given a line and maps points by flipping the plane about this line. A reflection in 3-space is given a plane, and flips points in space about this plane. In this case, reflection is just a special case of scaling, but where the scale factor is negative. For example, to reflect points about the xy-coordinate plane, we want to scale the z-coordinate by -1. Using the scaling matrix above, we have the following transformation matrix:

$$F_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The cases for the other two coordinate frames are similar.

Rotation: In its most general form, rotation is defined to to take place about some fixed point, and around some fixed vector in space. We will consider the simplest case where the fixed point is the origin of the coordinate frame, and the vector is one of the coordinate axes. There are three basic rotations: about the x, y and z-axes. In each case the rotation is through an angle θ (given in radians). The rotation is assumed to be in accordance with a right-hand rule: if your right thumb is aligned with the axes of rotation, then positive rotation is indicated by your fingers.

Consider the rotation about the z-axis. The z-unit vector and origin are unchanged. The x-unit vector is mapped to $(\cos\theta, \sin\theta, 0, 0)^T$, and the y-unit vector is mapped to $(-\sin\theta, \cos\theta, 0, 0)^T$. Thus the rotation matrix is:

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

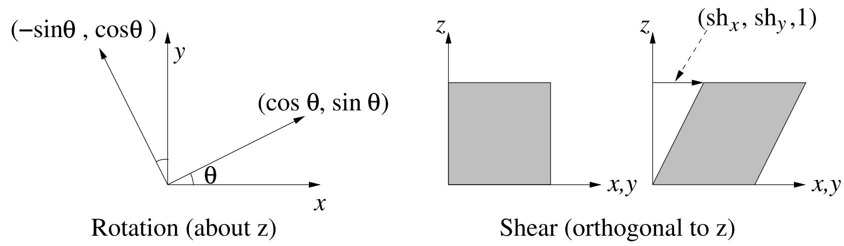


Figure 19: Rotation and shearing.

For the other two axes we have

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Shearing: A shearing transformation is the hardest of the group to visualize. Think of a shear as a transformation that maps a cube into a parallelogram. We will consider the simplest form, in which we start with a unit cube whose lower left corner coincides with the origin. Consider one of the axes, say the z-axis. The face of the cube that lies on the xy-coordinate plane does not move. The face that lies on the plane $z = 1$, is translated by a vector (sh_x, sh_y). In general, a point $P = (p_x, p_y, p_z, 1)$ is translated by the vector $p_z(sh_x, sh_y, 0, 0)$. This vector is orthogonal to the z-axis, and its length is proportional to the z-coordinate of P. This is called an *xy-shear*. (The yz- and xz-shears are defined analogously.)

Under the xy-shear, the origin and x- and y-unit vectors are unchanged. The z-unit vector is mapped to $(sh_x, sh_y, 1, 0)^T$. Thus the matrix for this transformation is:

$$H_{xy}(\theta) = \begin{pmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Shears involving any other pairs of axes are defined similarly.

Composing Affine Transformations: There are many more affine transformations. Affine transformations are closed under composition. (This is not hard to prove.) This means that if T and S are two affine transformations, then the composition $(T \circ S)$, define $(T \circ S)(P) = T(S(P))$ is also an affine transformation. Since T and S can be represented in matrix form as homogeneous matrices M_T and M_S, then it is easy to see that we can express their composition as the matrix product $M_T M_S$. Notice that the last matrix is the one that is applied first to the point or vector.

One way to compute more complex transformation is compose a series of the basic transformations together. For example, suppose that you wanted to rotate about a vertical line (parallel to z) passing through the point P. We could do this by first translating the plane by the vector $\mathcal{O} - P$, so that the (old) point P now coincides with the (new) origin. Then we could apply our rotation about the (new) origin. Finally, we translate space back by $P - \mathcal{O}$ so that the origin is mapped back to P. The resulting sequence of matrices would be

$$R_z(\theta, P) = T(P - \mathcal{O}) \cdot R_z(\theta) \cdot T(\mathcal{O} - P).$$

Lecture 9: More Geometric Operators

(Thursday, Sep 28, 2000)

Read: See Chapter 5 in Hill. The line clipping algorithm is different from the one described in the text, and orientations are not described in the text.

Defining complex transformations: We have already seen that it is possible to define complex affine transformations by composing a number of simple "basic" transformations. We also showed that to construct the matrix for any transformation, it suffices to simply concatenate the images of the basis elements for the standard frame. This suggests an alternative approach for creating complex transformation matrices. Determine the images of the the basis elements of the standard frame, then you can simply create the appropriate transformation matrix by concatenating these images. This is often an easier approach than composing many basic transformations.

To illustrate the idea, consider the 2-dimensional example illustrated below. We want to compute a transformation that maps the square object shown on the left to position P and rotated about P by some angle θ. To do this, we can define two frames F and G, such that the object is in the same position relative to each frame, as shown in the figure.

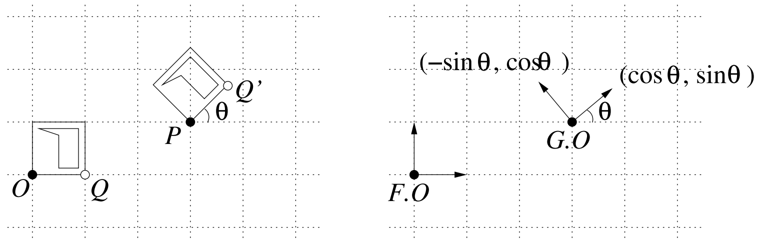


Figure 20: Constructing affine transformations.

For example, suppose for simplicity that F is just the standard frame. (We'll leave the more general case, where neither F nor G is the standard frame as an exercise.) Then the frames G is composed of the elements

$$G.\vec{e}_0 = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} \qquad G.\vec{e}_1 = \begin{pmatrix} -\sin\theta \\ \cos\theta \\ 0 \end{pmatrix} \qquad G.\mathcal{O} = \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}.$$

To compute the transformation matrix A, we express the basis elements of G relative to F, and then concatenate them together. We have

$$A = \begin{pmatrix} \cos\theta & -\sin\theta & 3 \\ \sin\theta & \cos\theta & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

As a check, consider the lower right corner point Q of the original square, whose coordinates relative to F are $(1,0,1)^T$. The product $A \cdot Q[F]$ yields

$$A \cdot Q[F] \;=\; \begin{pmatrix} \cos\theta & -\sin\theta & 3 \\ \sin\theta & \cos\theta & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \;=\; \begin{pmatrix} 3+\cos\theta \\ 1+\sin\theta \\ 1 \end{pmatrix} \;=\; P[F] + \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix}.$$

These are the coordinates of Q', as expected.

More Geometric Operators: So far we have discussed two important geometric operations used in computer graphics, change of coordinate systems and affine transformations. We saw that both operations could be expressed as the product of a matrix and vector (both in homogeneous form). Next we consider two more geometric operations, which are of a significantly different nature.

Cross Product: Here is an important problem in 3-space. You are given two vectors and you want to find a third vector that is orthogonal to these two. This is handy in constructing coordinate frames with orthogonal bases. There is a nice operator in 3-space, which does this for us, called the *cross product*.

The cross product is usually defined in standard linear 3-space (since it applies to vectors, not points). So we will ignore the homogeneous coordinate here. Given two vectors in 3-space, $\vec{u}$ and $\vec{v}$, their *cross product* is defined to be

$$\vec{u} \times \vec{v} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}.$$

A nice mnemonic device for remembering this formula, is to express it in terms of the following symbolic determinant:

$$\vec{u} \times \vec{v} = \begin{vmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}.$$

Here $\vec{e}_x$, $\vec{e}_y$, and $\vec{e}_z$ are the three coordinate unit vectors for the standard basis. Note that the cross product is only defined for a pair of free vectors and only in 3-space. The cross product has the following important properties:

Skew symmetric: $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$. It follows immediately that $\vec{u} \times \vec{u} = 0$ (since it is equal to its own negation).

Nonassociative: Unlike most other products that arise in algebra, the cross product is *not* associative. That is

$$(\vec{u} \times \vec{v}) \times \vec{w} \neq \vec{u} \times (\vec{v} \times \vec{w}).$$

Bilinear: The cross product is linear in both arguments. For example:

$$\begin{aligned} \vec{u} \times (\alpha\vec{v}) &= \alpha(\vec{u} \times \vec{v}), \\ \vec{u} \times (\vec{v} + \vec{w}) &= (\vec{u} \times \vec{v}) + (\vec{u} \times \vec{w}). \end{aligned}$$

Perpendicular: If $\vec{u}$ and $\vec{v}$ are not linearly dependent, then $\vec{u} \times \vec{v}$ is perpendicular to $\vec{u}$ and $\vec{v}$, and is directed according the the right-hand rule.

Angle and Area: The length of the cross product vector is related to the lengths of and angle between the vectors. In particular:

$$|\vec{u} \times \vec{v}| = |u||v| \sin \theta,$$

where θ is the angle between $\vec{u}$ and $\vec{v}$. The cross product is usually not used for computing angles because the dot product can be used to compute the cosine of the angle (in any dimension) and it can be computed more efficiently. This length is also equal to the area of the parallelogram whose sides are given by $\vec{u}$ and $\vec{v}$. This is often useful.

Orientation: Given two real numbers p and q, there are three possible ways they may be ordered: $p < q$, $p = q$, or $p > q$. We may define an orientation function, which takes on the values $+1$, 0, or -1 in each of these cases. That is, $\mathrm{Or}_1(p, q) = \mathrm{sign}(q - p)$, where $\mathrm{sign}(x)$ is either -1, 0, or $+1$ depending on whether x is negative, zero, or positive, respectively. An interesting question is whether it is possible to extend the notion of order to higher dimensions.

The answer is yes, but rather than comparing two points, in general we can define the orientation of $d + 1$ points in d-space. We define the *orientation* to be the sign of the determinant consisting of their homogeneous coordinates (with the homogenizing coordinate given first). For example, in the plane and 3-space the orientation of three points P, Q, R is defined to be

$$\mathrm{Or}_2(P, Q, R) = \mathrm{sign} \begin{vmatrix} 1 & 1 & 1 \\ p_x & q_x & r_x \\ p_y & q_y & r_y \end{vmatrix}, \qquad \mathrm{Or}_3(P, Q, R, S) = \mathrm{sign} \begin{vmatrix} 1 & 1 & 1 & 1 \\ p_x & q_x & r_x & s_x \\ p_y & q_y & r_y & s_y \\ p_z & q_z & r_z & s_z \end{vmatrix}.$$

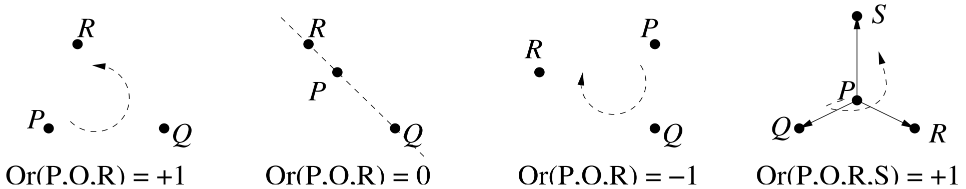


Figure 21: Orientations in 2 and 3 dimensions.

What does orientation mean intuitively? The orientation of three points in the plane is $+1$ if the triangle PQR is oriented counter-clockwise, -1 if clockwise, and 0 if all three points are collinear. In 3-space, a positive orientation means that the points follow a right-handed screw, if you visit the points in the order $PQRS$. A negative orientation means a left-handed screw and zero orientation means that the points are coplanar. Note that the order of the arguments is significant. The orientation of (P, Q, R) is the negation of the orientation of (P, R, Q). As with determinants, the swap of any two elements reverses the sign of the orientation.

You might ask why put the homogeneous coordinate first? The answer a mathematician would give you is that is really where it should be in the first place. If you put it last, then positive oriented things are "right-handed" in even dimensions and "left-handed" in odd dimensions. By putting it first, positively oriented things are always right-handed in orientation, which is more elegant. Putting the homogeneous coordinate last seems to be a convention that arose in engineering, and was adopted later by graphics people. If you stick with the engineering way, then compute the above determinant (with the homogeneous coordinate last) and multiply the final result by -1 if the dimension is odd.

The value of the determinant itself is the area of the parallelogram defined by the vectors $Q - P$ and $R - P$, and thus this determinant is also handy for computing areas. Later we will discuss another method.

Application–Line Clipping: To demonstrate some of the ideas that we have been presenting, we present a coordinate-free algorithm for a clipping a line relative to a convex polygon in the plane. *Clipping* is the process of trimming graphics primitives (e.g., line segments, circles, filled polygons) to the boundaries of some window. (See the figure below.) It is often applied in 2-space with a rectangular window. However, we shall see later that this procedure is also often invoked on nonrectangular windows in 3-space, as part of a more general process called *perspective clipping*.

Because this algorithm is called frequently, it is important to implement it in the most efficient manner. This involves introducing coordinates, and taking advantage of knowledge of the specific structure of the problem. For planar clipping, the resulting algorithm is called the *Liang-Barsky algorithm*. The advantage of the coordinate-free algorithm, which we will discuss, is that it is very easy to derive, and it is very general. It applies to virtually any sort of line segment clipping and in all dimensions. We will use a generalization of this procedure to intersect rays with polyhedra in ray shooting.

In 2-space, define a *halfplane* to be the portion of the plane lying to one side of a line. In general, in dimension d, we define a *halfspace* to the portion of d-space lying to one side of a $(d-1)$-dimensional hyperplane. In any dimension, a halfspace H can be represented by a pair $\langle R, \vec{n} \rangle$, where R is a point lying on the plane and $\vec{n}$ is a normal vector pointing into the halfspace. See the figure below. Observe that a point P lies within the halfspace if and only if the vector $P - R$ forms an angle of at most 90 degrees with respect to $\vec{n}$, that is if

$$((P - R) \cdot \vec{n}) \geq 0.$$

If the dot product is zero, then P lies on the plane that bounds the halfspace.

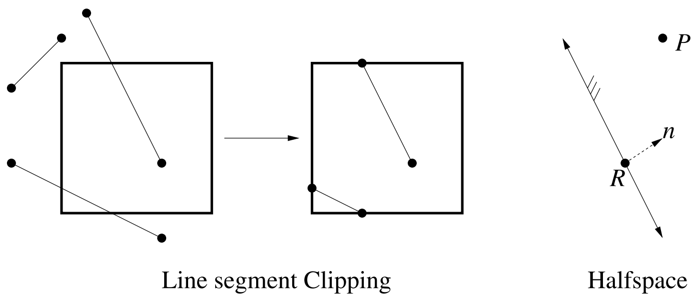


Figure 22: Clipping and Halfspaces.

A *convex polygon* in the plane is the intersection of a finite set of halfplanes. (This definition is not quite accurate, since it allows for unbounded convex polygons, but it is good enough for our purposes.) In general dimensions, a *convex polyhedron* is defined to be the intersection of a finite set of halfspaces. We will discuss the algorithm for the planar case, but we will see that there is nothing in our discussion that precludes generalization to higher dimensions.

The input to the algorithm is a set of halfplanes $H_1, \ldots, H_m$, where $H_i = \langle R_i, \vec{n_i} \rangle$ and a set of line segments, $L_1, \ldots, L_n$, where each line segment is represented by a pair of points, $\overline{P_{i,0}P_{i,1}}$. The algorithm works by clipping each line segment and outputting the resulting clipped segment. Thus it suffices to consider the case of a single segment $\overline{P_0P_1}$. If the segment lies entirely outside the window, then we return a special status flag indicating that the clipped segment is empty.

Parametric line clipper: We will represent each line segment *parametrically*, using convex combinations. In particular, any point on the line segment $\overline{P_0P_1}$ can be represented as

$$P(\alpha) = (1-\alpha)P_0 + \alpha P_1, \qquad \text{where } 0 \le \alpha \le 1.$$

The algorithm will compute two parameter values, α_0 and α_1, and the resulting clipped line segment is $\overline{P(\alpha_0)P(\alpha_1)}$. We will require that $\alpha_0 < \alpha_1$. Initially we set $\alpha_0 = 0$ and $\alpha_1 = 1$. Thus the initial clipped line segment is equal to the original segment. (Be sure you understand why.)

Our approach is to clip the line segment relative to the halfplane of the polygon, one by one. Let us consider how to clip one parameterized segment about one halfplane $\langle R, \vec{n}\rangle$. As the algorithm proceeds, α_0 will increase and α_1 will decrease, depending on where the clips are made. If ever $\alpha_0 > \alpha_1$ then the clipped line is empty, and we may return.

We want to know the value of α (if any) at which the line supporting the line segment intersects the line supporting the halfplane. To compute this, we plug the definition of $P(\alpha)$ into the above condition for lying within the halfplane,

$$((P(\alpha) - R) \cdot \vec{n}) \ge 0,$$

and we solve for α. Through simple affine algebra we have

$$\begin{aligned}
((1-\alpha)P_0 + \alpha P_1) - R) \cdot \vec{n}) &\ge 0 \\
((\alpha(P_1 - P_0) - (R - P_0)) \cdot \vec{n}) &\ge 0 \\
\alpha((P_1 - P_0) \cdot \vec{n}) - ((R - P_0)) \cdot \vec{n}) &\ge 0 \\
\alpha((P_1 - P_0) \cdot \vec{n}) &\ge ((R - P_0)) \cdot \vec{n}) \\
\alpha d_1 &\ge d_r
\end{aligned}$$

where $d_1 = ((P_1 - P_0) \cdot \vec{n})$ and $d_r = ((R - P_0)) \cdot \vec{n})$. From here there are a few cases depending on d_1.

$d_1 > 0$: Then $\alpha \ge d_r/d_1$. We set

$$\alpha_0 = \max(\alpha_0, d_r/d_1).$$

If as a result $\alpha_0 > \alpha_1$, then we return a flag indicating that the clipped line segment is empty.

$d_1 < 0$: Then $\alpha \le d_r/d_1$. We set

$$\alpha_1 = \min(\alpha_1, d_r/d_1).$$

If as a result $\alpha_1 < \alpha_0$, then we return a flag indicating that the clipped line segment is empty.

$d_1 = 0$: Then α is undefined. Geometrically this means that the bounding line and the line segment are parallel. In this case it suffices to check any point on the segment. So, if $(P_0 - R) \cdot \vec{n} < 0$ then we know that the entire line segment is outside of the halfplane, and so we return a special flag indicating that the clipped line is empty. Otherwise we do nothing.

Example: Let us consider an example of this algorithm. In the figure given below we have a convex window bounded by 4-sides, $4 \le x \le 10$ and $2 \le y \le 9$. To derive a halfplane representation for the sides, we create two points with (standard) coordinates $R_0 = (4, 2, 1)^T$

and $R_1 = (10, 9, 1)^T$. Let $\vec{e}_x = (1, 0, 0)^T$ and $\vec{e}_y = (0, 1, 0)^T$ be the coordinate unit vectors. Thus we have the four halfplanes

$$\begin{aligned} H_1 &= \langle R_0, \vec{e}_x \rangle \\ H_2 &= \langle R_0, \vec{e}_y \rangle \\ H_3 &= \langle R_1, -\vec{e}_x \rangle \\ H_4 &= \langle R_1, -\vec{e}_x \rangle. \end{aligned}$$

Let us clip the line segment $P_0 = (0, 8, 1)$ to $P_1 = (16, 0, 1)$.

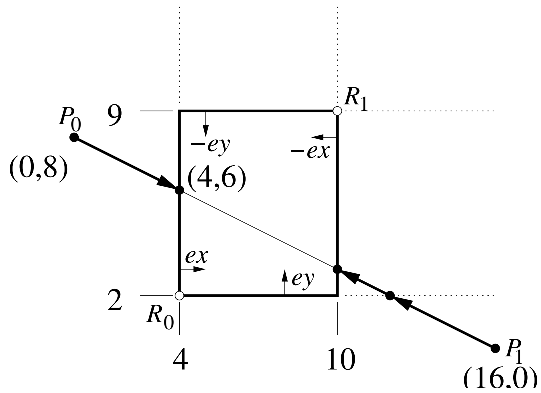


Figure 23: Clipping Algorithm.

Initially $\alpha_0 = 0$ and $\alpha_1 = 1$. First, let us consider the left wall, $\langle R_0, \vec{e}_x \rangle$. Plugging into our equations we have

$$\begin{aligned} d_1 &= ((P_1 - P_0) \cdot \vec{e}_x) \\ &= (((16, 0, 1) - (0, 8, 1)) \cdot (1, 0, 0)) \;=\; ((16, 8, 0) \cdot (1, 0, 0)) \;=\; 16, \\ d_r &= ((R_0 - P_0)) \cdot \vec{e}_x) \\ &= (((4, 2, 1) - (0, 8, 1))) \cdot (1, 0, 0)) \;=\; ((4, -6, 0) \cdot (1, 0, 0)) \;=\; 4. \end{aligned}$$

Since $d_1 > 0$, we let

$$\alpha_0 = \max(\alpha_0, d_r/d_1) = \max(0, 4/16) = 1/4.$$

Observe that the resulting point is

$$P(\alpha_0) = (1 - \alpha_0)P_0 + \alpha_0 P_1 = (3/4)(0, 8, 1) + (1/4)(16, 0, 1) = (4, 6, 1).$$

This is the point of intersection of the left wall with the line segment. The algorithm continues by clipping with respect to the other bounding halfplanes. We will leave the rest of the example as an exercise, but as a hint, from constraint H_2 we get $\alpha_1 \leq 3/4$, from H_3 we get $\alpha_1 \leq 5/8$, and from H_4 we get $\alpha_0 \geq -1/16$. The final values are $\alpha_0 = 1/4$ and $\alpha_1 = 5/8$.

Lecture 10: 3-d Viewing and Orthogonal Projections

(Tuesday, Oct 3, 2000)
Read: Chapter 5 and 7.6.2 in Hill.

Viewing in OpenGL: For the next couple of lectures we will discuss how viewing and perspective transformations are handled for 3-dimensional scenes. In OpenGL, and most similar graphics systems, the process involves the following basic steps, of which the perspective transformation is just one component. We assume that all objects are initially represented relative to a standard 3-dimensional coordinate frame, in what are called *world coordinates*.

Modelview transformation: Maps objects (actually vertices) from their world-coordinate representation to one that is centered around the viewer. The resulting coordinates are called *eye coordinates*.

Perspective projection: This projects points in 3-dimensional eye-coordinates to points on a plane called the *viewplane*. (We will see later that this transformation actually produces a 3-dimensional output, where the third component records depth information.) This projection process consists of three separate parts: the projection transformation (affine part), clipping, and perspective normalization. Each will be discussed below.

Mapping to the viewport: Convert the point from these idealized 2-dimensional coordinates (*normalized device coordinates*) to the viewport (pixel) coordinates.

We have ignored a number of issues, such as lighting and hidden surface removal. These will be considered separately later. The process is illustrated in the figure below. We have already discussed the viewport transformation, so it suffices to discuss the first two transformations.

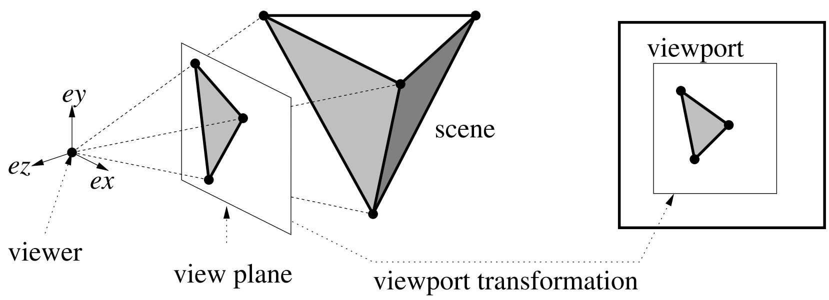


Figure 24: OpenGL Viewing Process.

Converting to Viewer-Centered Coordinate System: As we shall see below, the perspective transformation is simplest when the *center of projection*, the location of the viewer, is the origin and the *view plane* (sometimes called the *projection plane* or *image plane*), onto which the image is projected, is orthogonal to one of the axes, say the z-axis. Let us call these *eye coordinates*. However the user represents points relative to a coordinate system that is convenient for his/her purposes. Let us call these *world coordinates*. This suggests that prior to performing the perspective transformation, we perform a change of coordinate transformation to map points from world-coordinates to eye coordinates.

In OpenGL, there is a nice utility for doing this. The procedure `gluLookAt()` generates the desired transformation to perform this change of coordinates and multiplies it times the transformation at the top of the current transformation stack. (Recall OpenGL's transformation structure from Lecture ?.) This should be done in Modelview mode. Because this is the transformation which should be performed last (just prior to perspective projection), it is the first transformation that should be applied to the Modelview matrix. (Recall that matrices are loaded in the reverse order of their application.) Thus, it is almost always preceded by loading the identity matrix.

```
// assuming: glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz);
```

The arguments are all of type `GLdouble`. The arguments consist of the coordinates of two points and vector, in the standard coordinate system. The point $eye = (eye_x, eye_y, eye_z)$ is the *viewpoint*, that is the location of they viewer (or the camera). To indicate the direction that the camera is pointed, a central point to which the camera is facing is given by $center = (center_x, center_y, center_z)$. The center is significant only that its defines the *viewing vector*, which indicates the direction that the viewer is facing. It is defined to be $center - eye$.

These points define the position and direction of the camera, but the camera is still free to rotate about the viewing direction vector. To fix last degree of freedom, the vector (up_x, up_y, up_z) provides the direction that is "up" relative to the camera. Under typical circumstances, this would just be a vector pointing straight up (which might be $(0, 0, 1)$ in your world coordinate system). In some cases (e.g. in a flight simulator, when the plane banks to one side) you might want to have this vector pointing in some other direction. This vector *need not* be perpendicular to the viewing vector. However, it cannot be parallel to the viewing direction vector.

The Camera Frame: OpenGL uses the arguments to `gluLookAt()` to construct a coordinate frame centered at the viewer. The x- and y-axes are directed to the right and up, respectively, relative to the viewer. It might seem natural that the z-axes be directed in the direction that the viewer is facing, but this is not a good idea. To see why, we need to discuss the distinction between right-handed and left-handed coordinate systems. Consider a orthonormal coordinate system with basis vectors e_x, e_y and e_z. This system is said to be *right handed* if $e_x \times e_y = e_z$, and left handed otherwise ($e_x \times e_y = -e_z$). Right-handed coordinate systems are used by default throughout mathematics. (Otherwise computation of orientations is all screwed up.) Given that the x- and y-axes are directed right and up relative to the viewer, if the z-axis were to point in the direction that the viewer is facing, this would result in left-handed coordinate system. The designers of OpenGL wisely decided to stick to a right-handed coordinate system, which requires that the z-axes is directed opposite to the viewing direction.

Implementing gluLookAt: How does OpenGL implement this change of coordinate transformation? This turns out to be a nice exercise in geometric computation, so let's try it. We want to construct an orthonormal frame whose origin is the point eye, whose $-z$-basis vector is parallel to the view vector, and such that the $\overrightarrow{up}$ vector projects to the up direction in the final projection. (This is illustrated in the following figure, where the x-axis is pointing outwards from the page.)

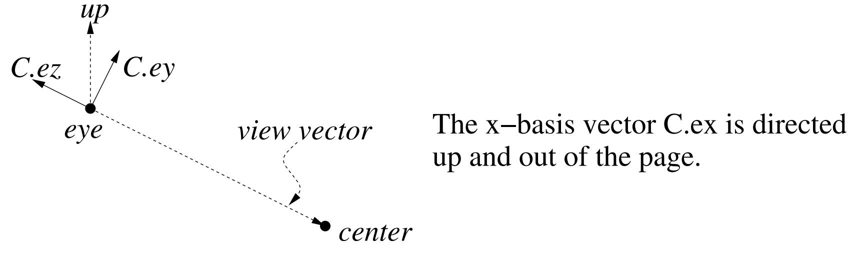


Figure 25: The camera frame.

Let C (for camera) denote this frame. Clearly $C.\mathcal{O} = eye$. As mentioned earlier, the view vector $\vec{v}$ is directed from eye to $center$. The z-basis vector is the normalized negation of this vector.

$$\begin{aligned} \vec{v} &= normalize(center - eye) \\ C.\vec{e}_z &= -\vec{v} \end{aligned}$$

(Recall that normalization divides a vector by its length, thus resulting in a vector having the same direction and unit length.)

Next, we want to select the x-basis vector for our camera frame. It should be orthogonal to the viewing direction, it should be orthogonal to the up vector, and it should be directed to the camera's right. Recall that the cross product will produce a vector that is orthogonal to any pair of vectors, and directed according to the right hand rule. Also, we want this vector to have unit length. Thus we choose

$$C.\vec{e}_x = normalize(\vec{v} \times \overrightarrow{up}).$$

The result of the cross product must be a nonzero vector. This is why we require that the view direction and up vector are not parallel to each other. We have two out of three vectors for our frame. We can extract the last one by taking a cross product of the first two.

$$C.\vec{e}_y = (C.\vec{e}_z \times C.\vec{e}_x).$$

There is no need to normalize this vector, because it is the cross product of two orthogonal vectors, each of unit length. Thus it will automatically be of unit length.

Now, all we need to do is to construct the change of coordinates matrix from the standard frame F to our camera frame C. Recall from our earlier lecture, that the change of coordinate matrix is formed by considering the matrix M whose columns are the basis elements of C relative to F, and then inverting this matrix. The matrix before inversion is:

$$M = \left(C.\vec{e}_x[F] \middle| C.\vec{e}_y[F] \middle| C.\vec{e}_z[F] \middle| C.\mathcal{O}[F] \right) = \begin{pmatrix} C.e_{xx} & C.e_{yx} & C.e_{zx} & C.\mathcal{O}_x \\ C.e_{xy} & C.e_{yy} & C.e_{zy} & C.\mathcal{O}_y \\ C.e_{xz} & C.e_{yz} & C.e_{zz} & C.\mathcal{O}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We can apply a trick to compute the inverse, M^{-1}, efficiently. Normally, inverting a matrix would involve invoking a linear algebra procedure (e.g., based on Gauss elimination). However, because M is constructed from an orthonormal frame, there is a much easier way to construct the inverse.

To see this, consider a 3×3 matrix A whose columns are orthogonal and of unit length. Such a matrix is said to be *orthogonal*. Consider the product $A^T A$ of this matrix and its transpose. Each of the diagonal elements of the product is the dot product of a column of a column of A with itself, which equals 1 (since the columns are of unit length). Each of the off-diagonal elements is the dot-product of two orthogonal vectors, which equals 0. Thus the result is an identity matrix, implying that $A^T = A^{-1}$. The upper-left 3×3 submatrix of M is of this type, but the last column is not. But we can still take advantage of this fact. First, we construct a 4×4 matrix R whose upper left 3×3 submatrix is copied from M:

$$R = \begin{pmatrix} C.e_{xx} & C.e_{yx} & C.e_{zx} & 0 \\ C.e_{xy} & C.e_{yy} & C.e_{zy} & 0 \\ C.e_{xz} & C.e_{yz} & C.e_{zz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Note that M is equal to the product of two matrices, a translation by the vector eye, denoted $T(eye)$ and R. Using the fact that $R^{-1} = R^T$, and $T(eye)^{-1} = T(-eye)$ we have

$$M^{-1} = (T(eye) \cdot R)^{-1} = R^{-1} \cdot T(eye)^{-1} = R^T \cdot T(-eye).$$

Thus, we do not need to invert any matrices to implement `gluLookAt()`. We simply compute the basis elements of the camera-frame (using cross products and normalization as described earlier), then we compute R^T (by copying these elements into the appropriate positions in the final matrix) and compute $T(-eye)$, and finally multiply these two matrices. If you consult the OpenGL Reference Manual you will see that this is essentially how `gluLookAt()` is defined.

Parallel and Orthogonal Projection: The second part of the process involves performing the projection. Projections fall into two basic groups, *parallel projections*, in which the lines of projection are parallel to one another, and *perspective projection*, in which the lines of projection converge a point.

In spite of their superficial similarities, parallel and perspective projections behave quite differently. Parallel projections are affine transformations, and perspective projections are not. (In particular, perspective projections do not preserve parallelism, as is evidenced by a perspective view of a pair of straight train tracks, which appear to converge at the horizon.) So let us start by considering the simpler case of parallel projections and consider perspective later.

There are many different classifications of parallel projections. Among these the simplest one is the *orthographic projection*, in which the lines of projection are all parallel to one of the coordinate axes, the z-axis in particular. See the figure below.

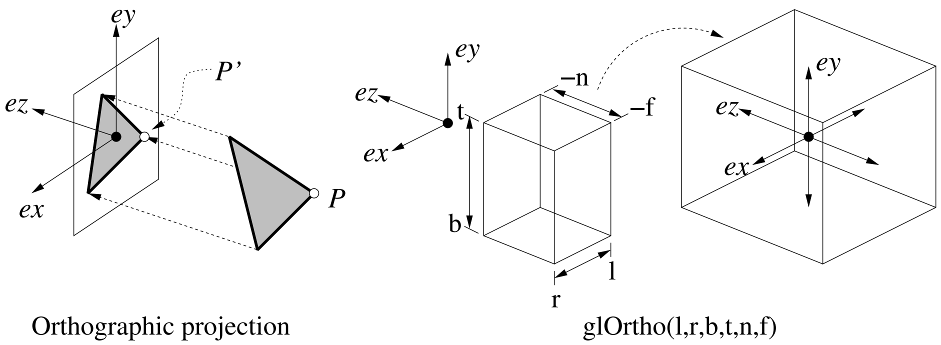


Figure 26: Orthographic Projection and glOrtho.

The transformation maps a point in 3-space to point on the xy-coordinate plane by setting the z-coordinate to zero. Thus a point $P = (p_x, p_y, p_z, 1)^T$ is mapped to the point $P' = (p_x, p_y, 0, 1)$. OpenGL does a few things differently in order to make its later jobs easier. First, the user specifies the window on the xy-plane onto which points are to be projected. This window will then be stretched to fit the viewport. This is done by specifying the minimum and maximum x-coordinates (*left*, *right*) and y-coordinates (*bottom*, *top*). Second, the transformation does not actually set the z-coordinate to zero. Even though the z-coordinate is unneeded for the final drawing, it conveys depth information, which is useful for hidden surface removal. For technical reasons having to do with how hidden surface removal is handled, it is necessary to indicate the range of distances along the z-axis. The user gives the distance along the $-(z)$-axis of the *near* and *far* clipping planes. (The fact that the z-axis points away from the viewing direction is rather unnatural for users. By negating the z-coordinate positive values for near and far are therefore in front of the viewer.) These six values define a rectangle R in 3-space. Points lying outside of this rectangle are clipped away. OpenGL maps R to a

$2 \times 2 \times 2$ hyperrectangle called the *canonical view volume*, which extends from -1 to $+1$ along each coordinate axis. This is done to simplify the clipping and depth buffer processing. The command **glOrtho()** is given these six arguments each as type **GLdouble**. The typical call is:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bottom, top, near, far);
glMatrixMode(GL_MODELVIEW);
```

The matrix that achieves this transformation is easy to derive. We wish to translate the center of the rectangle R to the origin, and then scale each axis so that each of the rectangle widths is scaled to a width of 2. (Note the negation of the z scale factor below.)

$$\begin{aligned} t_x &= (right + left)/2 & t_y &= (top + bottom)/2 & t_z &= (far + near)/2 \\ s_x &= 2/(right - left) & s_y &= 2/(top - bottom) & s_z &= -2/(far - near). \end{aligned}$$

The final transformation is the composition of a scaling and translation matrix.

$$S(s_x, s_y, s_z) \cdot T(-t_x, -t_y, -t_z) = \begin{pmatrix} s_x & 0 & 0 & -t_x s_x \\ 0 & s_y & 0 & -t_y s_y \\ 0 & 0 & s_z & -t_z s_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Lecture 11: Perspective

(Thursday, Oct 5, 2000)
Read: Chapter 7 in Hill.

Basic Perspective: Perspective transformations are the domain of an interesting area of mathematics called *projective geometry*. The basic problem that we will consider is the one of projecting points from a 3 dimensional space onto the 2-dimensional plane, called the *view plane*, centrally through a point (not on this plane) called the *center of projection*. The process is illustrated in the following figure.

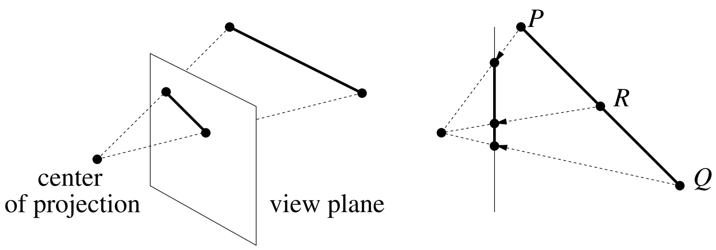


Figure 27: Perspective Transformations.

One nice things about projective transformations is that they map lines to lines. However, projective transformations are not affine, since (except for the special case of parallel projection) do not preserve affine combinations and do not preserve parallelism. For example, consider the perspective projection T shown in the figure. Let R be the midpoint of segment PQ then $T(R)$ is not necessarily the midpoint of $T(P)$ and $T(Q)$.

Projective Geometry: In order to gain a deeper understanding of projective transformations, it is best to start with an introduction to *projective geometry*. Projective geometry was developed in the 17th century by mathematicians interested in the phenomenon of perspective. Intuitively, the basic idea that gives rise to projective geometry is rather simple, but its consequences are somewhat surprising.

In Euclidean geometry we know that two distinct lines intersect in exactly one point, unless the two lines are parallel to one another. This special case seems like an undesirable thing to carry around. Suppose we make the following simplifying generalization. In addition to the *regular points* in the plane (with finite coordinates) we will also add a set of *ideal points* (or *points at infinity* that reside infinitely far away. Now, we can eliminate the special case and say that every two distinct lines intersect in a single point. If the lines are parallel, then they intersect at an ideal point. But there seem to be two such ideal points (one at each end of the parallel lines). Since we do not want lines intersecting more than once, we just imagine that the projective plane *wraps around* so that two ideal points at the opposite ends of a line are equal to each other. This is very elegant, since all lines behave much like closed curves (somewhat like a circle of infinite radius).

For example, in the figure below on the left, the point P is a point at infinity. Since P is infinitely far away it does have a position (in the sense of affine space), but it can be specified by pointing to it, that is, by a direction. All lines that are parallel to one another along this direction intersect at P. In the plane, the union of all the points at infinity forms a line, called the *line at infinity*. (In 3-space the corresponding entity is called the *plane at infinity*.) Note that every other line intersects the line at infinity exactly once. The regular affine plane together with the points and line at infinity define the *projective plane*. It is easy to generalize this to arbitrary dimensions as well.

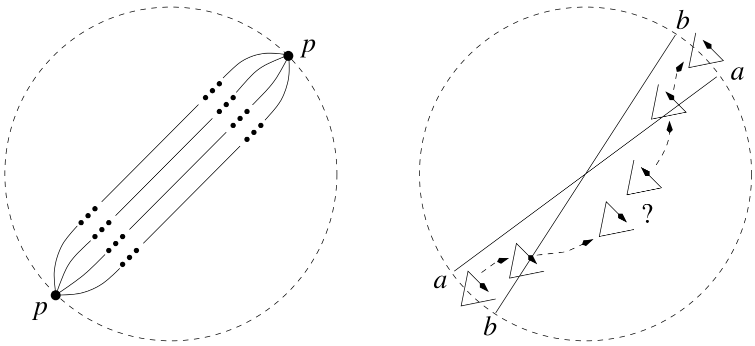


Figure 28: Projective Geometry.

Although the points at infinity seem to be special in some sense, an important tenet of projective geometry is that they are essentially no different from the regular points. In particular, when applying projective transformations we will see that regular points may be mapped to points at infinity and vice versa.

Orientability and the Projective Space: Projective geometry appears to both generalize and simplify affine geometry, so why we just dispensed with affine geometry and use projective geometry instead. The reason is that along with the good comes some rather strange consequences. For example, the projective plane wraps around itself in a rather strange way. In particular, it does not form a sphere as you might expect. (Try cutting it out of paper and gluing the edges together if you need proof.)

The nice thing about lines in the Euclidean plane is that each partitions the plane into two halves, one above and one below. This is not true for the projective plane (since each ideal point is both above and below). Furthermore, orientations such as clockwise and counterclockwise cannot even be defined. The projective plane is a *nonorientable manifold* (like a Moebius strip or Klein bottle). In particular, if you take an object with a counterclockwise orientation, and then translate it through the line at infinity and back to its original position, a strange thing happens. When passing through the line at infinity, its orientation changes. (Note that this does not happen on orientable manifolds like the Euclidean plane or the surface of a sphere).

Intuitively, this is because as we pass through infinity, there is a "twist" in space as is illustrated in the figure above on the right. Notice that the arrow is directed from point a to b, and when it "reappears" on the other side of the plane, this will still be the case. But, if you look at the figure, you will see that the relative positions of a and b are reversed[2].

For these reasons, we choose not to use the projective plane as a domain in which to do most of our geometric computations. Instead, we will briefly enter this domain, just long enough to do our projective transformations, and quickly jump back into the more familiar world of Euclidean space. We will have to take care that when performing these transformations we do not map any points to infinity, since we cannot map these points back to Euclidean space.

New Homogeneous Coordinates: How do we represent points in projective space? It turns out that we can do this by homogeneous coordinates. However, there are some differences. First off, we will not have free vectors in projective space. Consider a regular point P in the plane, with standard (nonhomogeneous) coordinates $(x, y)^T$. There will not be a unique representation for this point in projective space. Rather, it will be represented by any coordinate vector of the form:

$$\begin{pmatrix} w \cdot x \\ w \cdot y \\ w \end{pmatrix}, \qquad \text{for } w \neq 0.$$

Thus, if $P = (4, 3)^T$ are P's cartesian coordinates, the homogeneous coordinates $(4, 3, 1)^T$, $(8, 6, 2)^T$, and $(-12, -9, -3)^T$ are all legal representations of P in projective plane. Because of its familiarity, we will use the case $w = 1$ most often. Given the homogeneous coordinates of a regular point $P = (x, y, w)^T$, the *projective normalization* of P is the coordinate vector $(x/w, y/w, 1)^T$. (This term is confusing, because it is quite different from the process of *length normalization*, which maps a vector to one of unit length. In computer graphics this operation is also referred as *perspective division*.)

How do we represent ideal points? Consider a line passing through the origin with slope of 2. The following is a list of the homogeneous coordinates of some of the points lying on this line:

$$\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 6 \\ 1 \end{pmatrix}, \begin{pmatrix} 4 \\ 8 \\ 1 \end{pmatrix}, \ldots, \begin{pmatrix} x \\ 2x \\ 1 \end{pmatrix}.$$

Clearly these are equivalent to the following

$$\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 1/2 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 1/3 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 1/4 \end{pmatrix}, \ldots, \begin{pmatrix} 1 \\ 2 \\ 1/x \end{pmatrix}.$$

We can see that as x tends to infinity, the limiting point has the homogeneous coordinates $(1, 2, 0)^T$. So, when $w = 0$, the point $(x, y, w)^T$ is the point at infinity, that is pointed to by the vector $(x, y)^T$ (and $(-x, -y)^T$ as well by wraparound).

[2] One way of dealing with this phenomenon, is to define the projective plane differently, as a *two-sided projective plane*. The object starts on the front-side of the plane. When it passes through the line at infinity, it reappears on the back-side of the plane. When it passes again through the line at infinity it reappears on the front-side. Orientations are inverted as you travel from the front to back, and then are corrected from going from back to front.

Perspective Projection Transformations: We will not give a formal definition of projective transformations. (But it is not hard to do so. Just as affine transformations preserve affine combinations, projective transformations map lines to lines and preserve something called a *cross ratio*.) It is generally possible to define a perspective projection using a 4×4 matrix as we did with affine transformations. However, we will need treat projective transformations somewhat differently. Henceforth, we will assume that we will only be transforming points, not vectors. (Typically we will be transforming the endpoints of line segments and vertices of polygonal patches.) Let us assume for now that the points to be transformed are all strictly in front of the eye. We will see that objects behind the eye must eventually be clipped away, but we will consider this later.

Let us consider the following viewing situation. Since it is hard to draw good perspective drawings in 3-space, we will consider just the y and z axes for now (and everything we do with y we will do symmetrically with x later). We assume that the center of projection is located at the origin of some frame we have constructed.

Imagine that the viewer is facing the $-z$ direction. (Recall that this follows OpenGL's convention so that the coordinate frame is right-handed.) The x-axis points to the viewer's right and the y-axis points upwards relative to the viewer. Suppose that we are projecting points onto a projection plane that is orthogonal to the z-axis and is located at distance d from the origin along the $-z$ axis. (Note that d is given as a positive number, not a negative. This is consistent with OpenGL's conventions.) See the following figure.

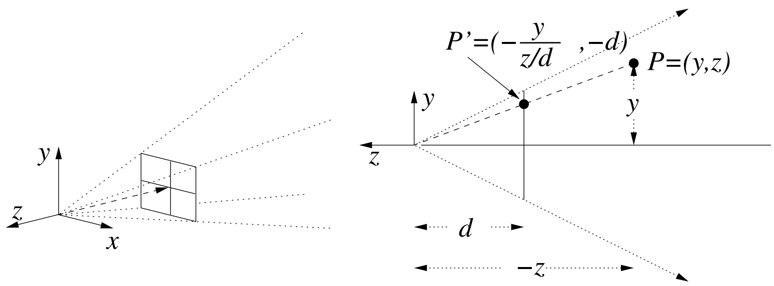


Figure 29: Perspective transformation.

Consider a point $P = (y, z)^T$ in the plane. (Note that z is negative but d is positive.) Where should this point be projected to on the view plane? Let $P' = (y', z')^T$ denote the coordinates of this projection. By similar triangles it is easy to see that the following ratios are equal:

$$\frac{y}{-z} = \frac{y'}{d},$$

implying that $y' = -y/(z/d)$. We also have $z = -d$. Generalizing this to 3-space, the point with coordinates $(x, y, z, 1)^T$ is transformed to the point with homogeneous coordinates

$$\begin{pmatrix} -x/(z/d) \\ -y/(z/d) \\ -d \\ 1 \end{pmatrix}.$$

Unfortunately, there is no 4×4 matrix that can realize this result. (Note that z is NOT a constant and so cannot be stored in the matrix.)

However, there is a 4×4 matrix that will generate the equivalent homogeneous coordinates.

In particular, if we multiply the above vector by $-(z/d)$ we get:

$$\begin{pmatrix} x \\ y \\ z \\ -z/d \end{pmatrix}.$$

This is a linear function of x, y, and z, and so we can write the perspective transformation in terms of the following matrix.

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

After we have the coordinates of a (affine) transformed point $P' = M \cdot P$, we then apply projective normalization (perspective division) to determine the corresponding point in Euclidean space. Notice that if $z = 0$, then we will be dividing by zero. But also notice that the perspective projection maps points on the xy-plane to infinity.

Lecture 12: Perspective in OpenGL

(Tuesday, Oct 10, 2000)
Read: Chapter 7 in Hill.

OpenGL's Perspective Projection: OpenGL provides a couple of ways to specify the perspective projection. The most general method is through `glFrustum()`. We will discuss a simpler method called `gluPerspective()`, which suffices for almost all cases that arise in practice. In particular, this simpler procedure assumes that the viewing window is centered about the view direction vector (the negative z-axis), whereas `glFrustum()` does not.

Consider the following viewing model. In front of his eye, the user holds rectangular window, centered on the view direction, onto which the image is to be projected. The viewer sees any object that lies within a rectangular cone, whose axis is the $-z$-axis, and whose apex is his eye. In order to indicate the height of this cone, the user specifies its angular height, called the *y field-of-view* and denoted *fovy*. It is given in degrees.

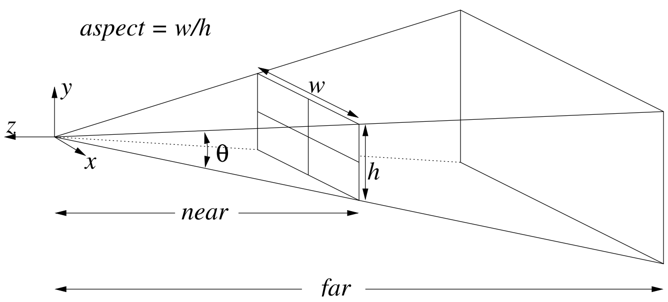


Figure 30: OpenGL's perspective specification.

To specify the angular width of the cone, we could specify the x field-of-view, but the designers of OpenGL decided on a different approach. Recall that the *aspect ratio* is defined to be the width/height ratio of the window. The user presumably knows the aspect ratio of his viewport, and typically users want an undistorted view of the world, so the ratio of the x and y fields-of-view should match the viewport's aspect ratio. Rather than forcing the user to compute the number of degrees of angular width, the user just provides the *aspect ratio* of the viewport, and the system then derives the x field-of-view from this value.

Finally, for technical reasons related to depth buffering, we need to specify a distance along the $-z$-axis to the *near clipping plane* and to the *far clipping plane*. Objects in front of the near plane and behind the far plane will be clipped away. We have a limited number of bits of depth-precision, and supporting a greater range of depth values will limit the accuracy with which we can represent depths. The resulting shape is called the *viewing frustum*. These arguments form the basic elements of the main OpenGL command for perspective.

```
gluPerspective(fovy, aspect, near, far);
```

All arguments are positive and of type **GLdouble**. This command creates a matrix which performs the necessary depth perspective transformation, and multiplies it with the matrix on top of the current stack. This transformation should be applied to the projection matrix stack. So this typically occurs in the following context of calls, usually as part of your initializations.

```
glMatrixMode(GL_PROJECTION);          // projection matrix mode
glLoadIdentity();                     // initialize to identity
gluPerspective(...);
glMatrixMode(GL_MODELVIEW);           // restore default matrix mode
```

Perspective with Depth: The question that we want to consider next is what perspective transformation matrix does OpenGL generate for this call? There is a significant shortcoming with the simple perspective transformation that we described last time. Recall from last time that the point $(x, y, z, 1)^T$ is mapped to the point $(-x/(z/d), -y/(z/d), -d, 1)^T$. The last two components of this vector convey no information, for they are the same, no matter what point is projected.

Is there anything more that we could ask for? In turns out that there is. This is *depth information*. We would like to know how far a projected point is from the viewer. After the projection, all depth information is lost, because all points are flattened onto the projection plane. Such depth information would be very helpful in performing hidden-surface removal. Let's consider how we might include this information.

We will design a projective transformation in which the (x, y)-coordinates of the transformed points are the desired coordinates of the projected point, but the z-coordinate of the transformed point encodes the depth information. This is called *perspective with depth*. The (x, y) coordinates are then used for drawing the projected object and the z-coordinate is used in hidden surface removal. It turns out that this depth information will be subject to a nonlinear distortion. However, the important thing will be that depth-order will be preserved, in the sense that points that are further from the eye (in terms of their z-coordinates) will have greater depth values than points that are nearer.

As a start, let's consider the process in a simple form. As usual we assume that the eye is at the origin and looking down the $-z$-axis. Let us also assume that the projection plane is located at $z = -1$. Consider the following matrix:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

If we apply it to a point P with homogeneous coordinates $(x, y, z, 1)^T$, then the resulting point has coordinates

$$M \cdot P = \begin{pmatrix} x \\ y \\ \alpha z + \beta \\ -z \end{pmatrix} \equiv \begin{pmatrix} -x/z \\ -y/z \\ -\alpha - \beta/z \\ 1 \end{pmatrix}$$

Note that the x and y coordinates have been properly scaled for perspective (recalling that $z < 0$ since we are looking down the $-z$-axis). The *depth value* is

$$z' = -\alpha - \frac{\beta}{z}.$$

Depending on the values we choose for α and β, this is a (nonlinear) monotonic function of z. In particular, depth increases as the z-values decrease (since we view down the negative z-axis), so if we set $\beta < 0$, then the depth value z' will be a monotonically increasing function of depth. In fact, by choosing α and β properly, we adjust the depth values to lie within whatever range of values suits us. We'll see below how these values should be chosen.

Canonical View Volume: In applying the perspective transformation, all points in projective space will be transformed. This includes point that are not within the viewing frustum (e.g., points lying behind the viewer). One of the important tasks to be performed by the system, prior to perspective division (when all the bad stuff might happen) is to clip away portions of the scene that do not lie within the viewing frustum.

OpenGL has a very elegant way of simplifying this clipping. It adjusts the perspective transformation so that the viewing frustum (no matter how it is specified by the user) is mapped to the same canonical shape. Thus the clipping process is always being applied to the same shape, and this allows the clipping algorithms to be designed in the most simple and efficient manner. This idealized shape is called the *canonical view volume*. Clipping is actually performed in homogeneous coordinate (i.e., 4-dimensional) space just prior to perspective division. However, we will describe the canonical view volume in terms of how it appears after perspective division. (We will leave it as an exercise to figure out what it looks like prior to perspective division.)

The canonical view volume (after perspective division) is just a 3-dimensional rectangle. It is defined by the following constraints:

$$-1 \leq x \leq +1, \quad -1 \leq y \leq +1, \quad -1 \leq z \leq +1.$$

The (x, y) coordinates indicate the location of the projected point on the final viewing window. The z-coordinate is used for depth. There is a reversal of the z-coordinates, in the sense that before the transformation, points further from the viewer have smaller z-coordinates (larger in absolute value, but smaller because they are on the negative z side of the origin). Now, the points with $z = -1$ are the closest to the viewer (lying on the near clipping plane) and the points with $z = +1$ are the furthest from the viewer (lying on the far clipping plane). Points that lie on the top (resp. bottom) of the canonical volume correspond to points that lie on the top (resp. bottom) of the viewing frustum.

Returning to the earlier discussion about α and β, we see that we want to map points on the near clipping plane $z = -n$ to $z' = -1$ and points on the far clipping plane $z = -f$ to $z' = +1$. This gives the simultaneous equations:

$$\begin{aligned} -1 &= -\alpha - \frac{\beta}{-n} \\ +1 &= -\alpha - \frac{\beta}{-f}. \end{aligned}$$

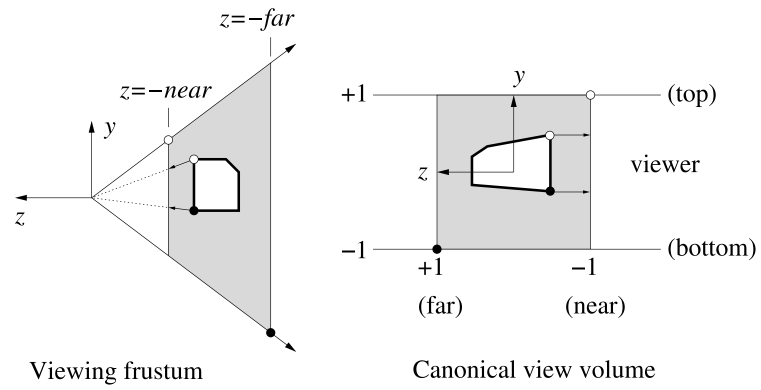


Figure 31: Perspective with depth.

Solving for α and β gives

$$\alpha = \frac{f+n}{n-f} \qquad \beta = \frac{2fn}{n-f}.$$

Perspective Matrix: To see how OpenGL handles this, recall the function `gluPerspective()`. Let $c = \cot(\theta/2)$. We will take a side view as usual (thus ignoring the x-coordinate). Let a denote the aspect ratio, let n denote the distance to the near clipping plane and let f denote the distance to the far clipping plane. (All quantities are positive.) Here is the matrix it constructs to perform the perspective transformation.

$$M = \begin{pmatrix} c/a & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & \dfrac{f+n}{n-f} & \dfrac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Observe that a point P in 3-space with homogeneous coordinates $(x, y, z, 1)^T$ is mapped to

$$M \cdot P \;=\; \begin{pmatrix} cx/a \\ cy \\ ((f+n)z + 2fn)/(n-f) \\ -z \end{pmatrix} \;\equiv\; \begin{pmatrix} -cx/(az) \\ -cy/z \\ (-(f+n) - (2fn/z))/(n-f) \\ 1 \end{pmatrix}.$$

How did we come up with such a strange mapping? Notice that other than the scaling factors, this is very similar to the perspective-with-depth matrix given earlier (given our values α and β plugged in). The diagonal entries c/a and c are present to scale the arbitrarily shaped window into the square (as we'll see later).

To see that this works, we will show that the corners of the viewing frustum are mapped to the corners of the canonical viewing volume (and we'll trust that everything in between behaves nicely). In the figure we show a side view, thus ignoring the x-coordinate. Because the window has the aspect ratio $a = w/h$, it follows that for points on the upper-right edge of the viewing frustum (relative to the viewer's perspective) we have $x/y = a$, and thus $x = ay$.

Consider a point that lies on the top side of the view frustum. We have $-z/y = \cot\theta/2 = c$, implying that $y = -z/c$. If we take the point to lie on the near clipping plane, then we have $z = -n$, and hence $y = n/c$. Further, if we assume that it lies on the upper right corner of the frustum (relative to the viewer's position) then $x = ay = an/c$. Thus the homogeneous coordinates of the upper corner on the near clipping plane (shown as a white dot in the figure) are $(an/c, n/c, -n, 1)^T$. If we apply the above transformation, this is mapped to

$$M\begin{pmatrix} an/c \\ n/c \\ -n \\ 1 \end{pmatrix} = \begin{pmatrix} n \\ n \\ \dfrac{-n(f+n)}{n-f} + \dfrac{2fn}{n-f} \\ n \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 1 \\ \dfrac{-(f+n)}{n-f} + \dfrac{2f}{n-f} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix}.$$

Notice that this is the upper corner of the canonical view volume on the near ($z = -1$) side, as desired.

Similarly, consider a point that lies on the bottom side of the view frustum. We have $-z/(-y) = \cot\theta/2 = c$, implying that $y = z/c$. If we take the point to lie on the far clipping plane, then we have $z = -f$, and so $y = -f/c$. Further, if we assume that it lies on the lower left corner of the frustum (relative to the viewer's position) then $x = -af/c$. Thus the homogeneous coordinates of the lower corner on the far clipping plane (shown as a black dot in the figure) are $(-af/c, -f/c, -f, 1)^T$. If we apply the above transformation, this is mapped to

$$M\begin{pmatrix} -af/c \\ -f/c \\ -f \\ 1 \end{pmatrix} = \begin{pmatrix} -f \\ -f \\ \dfrac{-f(f+n)}{n-f} + \dfrac{2fn}{n-f} \\ f \end{pmatrix} \equiv \begin{pmatrix} -1 \\ -1 \\ \dfrac{-(f+n)}{n-f} + \dfrac{2n}{n-f} \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}.$$

This is the lower corner of the canonical view volume on the far ($z = 1$) side, as desired.

Lecture 13: Lighting and Shading

(Thursday, Oct 12, 2000)
Read: Chapter 8 in Hill.

Lighting and Shading: We will now take a look at the next major element of graphics rendering: light and shading. This is one of the primary elements of generating realistic images. This topic is the beginning of an important shift in approach. Up until now, we have discussed graphics from are purely mathematical (geometric) perspective. Light and reflection brings us to issues involved with the physics of light and color and the physiological aspects of how humans perceive light and color.

An accurate simulation of light energy and how it emanates, reflects off and passes through objects is an amazingly complex, and computationally intractable task. Although the human visual system is quite sensitive to certain errors in accurate rendering, we can be quite tolerant with other sorts of errors. Much of computer graphics involves tricking the eye by producing reasonable approximations, which are good enough to fool the eye (or at least well enough to allow ourselves to suspend belief that what we are seeing is not real).

OpenGL, like most interactive graphics systems, supports a very simple lighting and shading model, and hence can achieve only limited realism. This was done primarily because speed is of the essence in interactive graphics. OpenGL uses something called a *local illumination model*, which means that the light of a point depends only on its relationship to light sources, without considering the other objects in the scene. For example, OpenGL's lighting model does not model shadows, it does not handle indirect reflection from other objects (where light bounces off of one object and illuminates another), it does not handle objects that reflect or refract light (like metal spheres and glass balls). However the designers of OpenGL have included a number of tricks for essentially "faking" many of these effects.

OpenGL's light and shading model was designed to be very efficient and very general (in order to permit the faking alluded to earlier). It contains a number features that seem to bear little or no resemblance to the laws of physics. The lighting model that we will is slightly different from OpenGL's model, but is a bit more meaningful from the perspective of physics.

Light Sources: Before talking about light reflection, we need to discuss where the light originates. In reality, light sources come in many sizes and shapes. They may emit light in varying intensities and wavelengths according to direction. To simplify things, OpenGL assumes that all light sources are points, and that the energy they emit can be modeled as an RGB triple.

Light sources do not necessarily emit white light. They emit light according to a *luminance function*, which can be broken into three components (L_r, L_g, L_b) for the intensities of red, green, and blue light respectively. We will not concern ourselves with the exact units of measurement. Typical units might be watts or lumens, depending on whether you are considering radiometry (the physics of light) or photometry (the study of the perception of light).

Lighting in real environments usually involves a considerable amount of indirect reflection between objects of the scene. If we were to ignore this effect, and simply consider a point to be illuminated only if it can see the light source, then the resulting image in which objects in the shadows are totally black. In indoor scenes we are used to seeing much softer shading, so that even objects that are hidden from the light source are partially illuminated. In OpenGL (and most local illumination models) this is modeled by breaking the light source's intensity into two components: *ambient emission* and *point emission*.

Ambient emission does not come from any one location. Like a heated room, it is scattered uniformly in all locations and directions. A point is illuminated by ambient emission even if it is not visible from the light source. On the other hand, *point emission* originates from the point of the light source. In theory, point emission only affects points that are directly visible to the light source. That is, a point P is illuminate by light source Q if and only if the open line segment $\overline{PQ}$ does not intersect any of the objects of the scene.

Unfortunately, determining whether a point is visible to a light source in a complex scene with thousands of objects can be computationally quite expensive. So OpenGL simply tests whether the surface is facing towards the light or away from the light. Surfaces in OpenGL all all polygons, but let us consider this in a more general setting. Suppose that have a point P lying on some surface. Let $\vec{n}$ denote the normal vector at P, directed *outwards* from the object, and let $\vec{\ell}$ denote the directional vector from P to the light source ($\vec{\ell} = Q - P$), then P will be illuminated if and only if the angle between these vectors is acute, that is, if $\vec{n} \cdot \vec{\ell} > 0$.

For example, in the following figure, the point P is illuminated. In spite of the obscuring triangle, point P' is also illuminated, because other objects in the scene are ignored by the local illumination model. The point P'' is clearly not illuminated, because its normal is directed away from the light.

We may describe the light's intensity in terms of its ambient and point emission strengths:

$$L_a = \begin{pmatrix} L_{ar} \\ L_{ag} \\ L_{ab} \end{pmatrix} \qquad L_p = \begin{pmatrix} L_{pr} \\ L_{pg} \\ L_{pb} \end{pmatrix}.$$

Attenuation: Light is subject to *attenuation*, that is, the decrease in strength of illumination as the distance to the source increases. Physics tells us that the intensity of light falls off as the

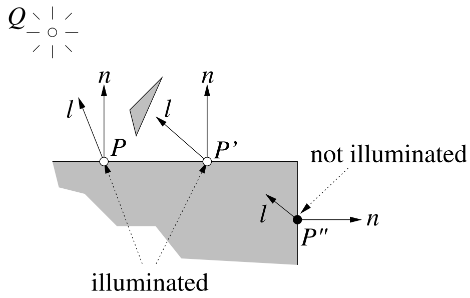


Figure 32: Point light source visibility.

inverse square of the distance. This would imply that the intensity at some (unblocked) point P would be

$$I(P,Q) = \frac{1}{|P-Q|^2} I(Q),$$

where $|P-Q|$ denotes the Euclidean distance from P to Q. However, in OpenGL, our various simplifying assumptions (ignoring indirect reflections, for example) will cause point sources to appear unnaturally dim using the exact physical model of attenuation. Consequently, it is common to use an attenuation function that has constant, linear, and quadratic components. The user specifies constants a, b and c. Let $d = |P-Q|$ denote the distance to the point source. Then the attenuation function is

$$I(P,Q) = \frac{1}{a + bd + cd^2} I(Q).$$

In OpenGL, the default values are $a = 1$ and $b = c = 0$, so there is no attenuation by default.

Directional Sources: A light source can be placed infinitely far away by using the projective geometry convention of setting the last coordinate to 0. Suppose that we imagine that the z-axis points up. At high noon, the sun's coordinates would be modeled by the homogeneous positional vector

$$(0,0,1,0)^T.$$

These are called *directional sources*. There is a performance advantage to using directional sources. Many of the computations involving light sources require computing angles between the surface normal and the light source location. If the light source is at infinity, then all points on a single polygonal patch have the same angle, and hence the angle need be computed only once for all points on the patch.

Sometimes it is nice to have a directional component to the light sources. OpenGL also supports something called a *spotlight*, where the intensity is strongest along a given direction, and then drops off according to the angle from this direction. See the OpenGL function `glLight()` for further information.

Lights in OpenGL: To use lighting in OpenGL, first you must enable lighting, through a call to `glEnable()`. OpenGL allows the user to create up to 8 light sources, named `GL_LIGHT0` through `GL_LIGHT7`. Each light source may either be enabled (turned on) or disabled (turned off). By default they are all disabled. Again this is done using `glEnable()` (and `glDisable()`). The properties of each light source is set by the command `glLight*()`. This command takes three arguments, the name of the light, the property of the light to set, and the value of this property. We will discuss only a simple example for now. Let us enable lighting, and then enable light 0, whose position is $(2,4,5,1)^T$ in homogeneous coordinates, and which has a gray ambient intensity and red diffuse intensity. These intensities are expressed as RGBA values. (We will discuss diffuse intensity below.) The procedure **glLight*()** can be used for setting other things like attenuation.

Setting up a simple lighting situation

```
GLfloat ambientIntensity[4] = {0.5, 0.5, 0.5, 1.0};
GLfloat diffuseIntensity[4] = {1.0, 0.0, 0.0, 1.0};
GLfloat position[4] = {2.0, 4.0, 5.0, 1.0};

glEnable(GL_LIGHTING);                      // enable lighting
glEnable(GL_LIGHT0);                        // enable light 0
                                            // set up light 0 properties
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientIntensity);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseIntensity);
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

There are a number of other commands used for defining how light and shading are done in OpenGL. We will discuss these in greater detail later. They include **glLightModel()** and **glShadeModel()**.

Types of light reflection: The next issue needed to determine how objects appear is how this light is *reflected* off of the objects in the scene and reach the viewer. So the discussion shifts from the discussion of light sources to the discussion of object surface properties. We will assume that all objects are opaque. The simple model that we will use for describing the reflectance properties of objects is called the *Phong model*. The model is some 20 years old, and is based on modeling surface reflection as a combination of the following components:

Emission: This is used to model objects that glow (even when all the lights are off). This is unaffected by the presence of any light sources. However, because our illumination model is local, it does not behave like a light source, in the sense that it does not cause any other objects to be illuminated.

Ambient reflection: This is a simple way to model indirect reflection. All surfaces in all positions and orientations are illuminated equally by this light energy.

Diffuse reflection: The illumination produced by matte (i.e, dull or not shiny) smooth objects, such as foam rubber.

Specular reflection: The bright spots appearing on smooth shiny (e.g. metallic or polished) surfaces. Although specular reflection is related to pure reflection (as with mirrors), for the purposes of our simple model these two are different. In particular, specular reflection only reflects light, not the surrounding objects in the scene.

Let $L = (L_r, L_g, L_b)$ denote the illumination intensity of the light source. OpenGL allows us to break this light's emitted intensity into three components: *ambient* L_a, *diffuse* L_d, and *specular* L_s. Each type of light component consists of the three color components, so, for example, $L_d = (L_{dr}, L_{dg}, L_{db})$, denotes the RGB vector (or more generally the RGBA components) of the diffuse component of light. As we have seen, modeling the ambient component separately is merely a convenience for modeling indirect reflection. It is not as clear why someone would want to turn on or turn off a light source's ability to generate diffuse and specular reflection. (There is no physical justification to this that I know of. It is an object's surface properties, not the light's properties, which determine whether light reflects diffusely or specularly. But, again this is all just a model.) The diffuse and specular intensities of a light source are usually set equal to each other.

An object's color determines how much of a given intensity is reflected. Let $C = (C_r, C_g, C_b)$ denote the object's color. These are assumed to be normalized to the interval $[0, 1]$. Thus we can think of C_r as the fraction of red light that is reflected from an object. Thus, if $C_r = 0$, then no red light is reflected. When light of intensity L hits an object of color C, the amount of reflected light is given by the product

$$LC = (L_r C_r, L_g C_g, L_b C_b).$$

Beware: This is a component-by-component multiplication, and not a vector multiplication or dot-product in the usual sense. For example, if the light is white $L = (1, 1, 1)$ and the color is red $C = (1, 0, 0)$ then the reflection is $LC = (1, 0, 0)$ which is red. However if the light is blue $L = (0, 0, 1)$, then there is no reflection, $LC = (0, 0, 0)$, and hence the object appears to be black.

In OpenGL rather than specifying a single color for an object (which indicates how much light is reflected for each component) you instead specify the amount of reflection for each type of illumination: C_a, C_d, and C_s. Each of these is an RGBA vector. This seems to be a rather extreme bit of generality, because, for example, it allows you to specify that an object can reflect only red light ambient light and only blue diffuse light. Again, I know of no physical explanation for this phenomenon. Note that it is common that the specular color (since it arises by way of reflection of the light source) is usually made the same color as the light source, not the object. In our presentation, we will assume that $C_a = C_d = C$, the color of the object, and that $C_s = L$, the color of the light source.

So far we have laid down the foundation for the Phong Model. Next time we will discuss exactly how the Phong model assigns colors to the points of a scene.

Lecture 14: The Phong Reflection Model

(Tuesday, Oct 17, 2000)
Read: Chapter 8 in Hill.

The Phong Reflection Model: Last time we introduced the Phong reflection model. Recall that this is a local illumination model in which the light coming off of an object is grouped into one of the following categories.

Emission: Light emanating from the object, irrespective of any light sources.

Ambient reflection: Light which reflect uniformly from all objects in all directions.

Diffuse reflection: Models matte (unshiny) reflection.

Specular reflection: Models shiny surface reflection.

The Relevant Vectors: The shading of a point on a surface is a function of the relationship between the viewer, light sources, and surface. (Recall that because this is a local illumination model the other objects of the scene are ignored.) The following vectors are relevant to shading. We can think of them as being centered on the point whose shading we wish to compute. For the purposes of our equations below, it will be convenient to think of them all as being of unit length. They are illustrated in the figure below.

Normal vector: A vector $\vec{n}$ that is perpendicular to the surface and directed outwards from the surface. There are a number of ways to compute normal vectors, depending on the representation of the underlying object. For our purposes, the following simple method is

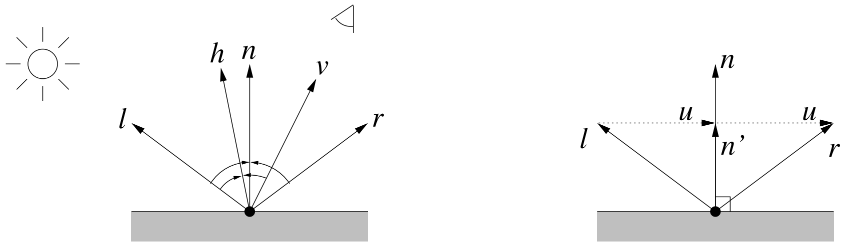


Figure 33: Vectors used in Phong Shading.

sufficient. Given any three noncollinear points, P_0, P_1, P_2, on a polygon, we can compute a normal to the surface of the polygon as a cross product of two of the associated vectors.

$$\vec{n} = normalize((P_1 - P_0) \times (P_2 - P_0)).$$

The vector will be directed outwards if the triple $P_0 P_1 P_2$ has a counterclockwise orientation when seen from outside.

View vector: A vector $\vec{v}$ that points in the direction of the viewer (or camera).

Light vector: A vector $\vec{\ell}$ that points towards the light source.

Reflection vector: A vector $\vec{r}$ that indicates the direction of pure reflection of the light vector. (Based on the law that that the angle of incidence with respect to the surface normal equals the angle of reflection.) The reflection vector computation reduces to an easy exercise in vector arithmetic. First observe that (because all vectors have been normalized to unit length) the orthogonal projection of $\vec{\ell}$ onto $\vec{n}$ is

$$\vec{n}' = (\vec{n} \cdot \vec{\ell})\vec{n}.$$

The vector directed from the tip of $\vec{\ell}$ to the tip of $\vec{n}'$ is $\vec{u} = \vec{n}' - \vec{\ell}$. To get $\vec{r}$ observe that we need add two copies of $\vec{u}$ to $\vec{\ell}$. Thus we have

$$\vec{r} = \vec{\ell} + 2\vec{u} = \vec{\ell} + 2(\vec{n}' - \vec{\ell}) = 2(\vec{n} \cdot \vec{\ell})\vec{n} - \vec{\ell}.$$

Halfway vector: A vector $\vec{h}$ that is midway between $\vec{\ell}$ and $\vec{v}$. Since this is half way between $\vec{\ell}$ and $\vec{v}$, and both have been normalized to unit length, we can compute this by simply averaging these two vectors and normalizing (assuming that they are not pointing in exactly opposite directions). Since we are normalizing, the division by 2 for averaging is not needed.

$$\vec{h} = normalize((\vec{\ell} + \vec{v})/2) = normalize(\vec{\ell} + \vec{v}).$$

Phong Lighting Equations: There almost no objects that are pure diffuse reflectors or pure specular reflectors. The Phong reflection model is based on the simple modeling assumption that we can model any (nontextured) object's surface to a reasonable extent as some mixture of purely diffuse and purely specular components of reflection along with emission and ambient reflection. Let us ignore emission for now, since it is the rarest of the group, and will be easy to add in at the end of the process.

The surface material properties of each object will be specified by a number of parameters, indicating the intrinsic color of the object and its ambient, diffuse, and specular reflectance. Let C denote the RGB factors of the object's base color. As mentioned in the previous lecture, we assume that the light's energy is given by two RGB vectors L_a, its ambient component and L_p its point component (assuming origin at point Q). For consistency with OpenGL, we will assume that we differentiate L_p into two subcomponents L_d and L_s, for the diffuse and specular energy of the light source (which are typically equal to each other). Typically all three will have the same proportion of red to green to blue, since they all derive from the same source.

Ambient light: Ambient light is the simplest to deal with. Let I_a denote the intensity of reflected ambient light. For each surface, let

$$0 \leq \rho_a \leq 1$$

denote the surface's *coefficient of ambient reflection*, that is, the fraction of the ambient light that is reflected from the surface. The ambient component of illumination is

$$I_a = \rho_a L_a C$$

Note that this is a vector equation (whose components are RGB).

Diffuse reflection: Diffuse reflection arises from the assumption that light from any direction is reflected uniformly in all directions. Such an reflector is called a pure *Lambertian reflector*. The physical explanation for this type of reflection is that at a microscopic level the object is made up of *microfacets* that are highly irregular, and these irregularities scatter light uniformly in all directions.

The reason that Lambertian reflectors appear brighter in some parts that others is that if the surface is facing (i.e. perpendicular to) the light source, then the energy is spread over the smallest possible area, and thus this part of the surface appears brightest. As the angle of the surface normal increases with respect to the angle of the light source, then an equal among of the light's energy is spread out over a greater fraction of the surface, and hence each point of the surface receives (and hence reflects) a smaller amount of light.

It is easy to see from the figure below, that as the angle θ between the surface normal $\vec{n}$ and the vector to the light source $\vec{\ell}$ increases (up to a maximum of 90 degrees) then amount of light intensity hitting a small differential area of the surface dA is proportional to the area of the perpendicular cross-section of the light beam, $dA \cos\theta$. The is called *Lambert's Cosine Law.*

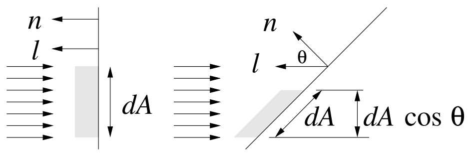


Figure 34: Lambert's Cosine Law.

The key parameter of surface finish that controls diffuse reflection is ρ_d, the surface's *coefficient of diffuse reflection*. Let I_d denote the diffuse component of the light source. If we assume that $\vec{\ell}$ and $\vec{n}$ are both normalized, then we have $\cos\theta = (\vec{n} \cdot \vec{\ell})$. If $(\vec{n} \cdot \vec{\ell}) < 0$, then the point is on the dark side of the object. The diffuse component of reflection is:

$$I_d = \rho_d \max(0, \vec{n} \cdot \vec{\ell}) L_d C.$$

This is subject to attenuation depending on the distance of the object from the light source.

Specular Reflection: Most objects are not perfect Lambertian reflectors. One of the most common deviations is for smooth metallic or highly polished objects. They tend to have *specular highlights* (or "shiny spots"). Theoretically, these spots arise because at the microfacet level, light is not being scattered perfectly randomly, but shows a preference for being reflected according to familiar rule that the angle of incidence equals the angle of reflection. On the other hand, the microfacet level, the facets are not so smooth that we get a clear mirror-like reflection.

There are two common ways of modeling of specular reflection. The Phong model uses the reflection vector (derived earlier). OpenGL instead uses a vector called the *halfway vector*, because it is somewhat more efficient and produces essentially the same results. Observe that if the eye is aligned perfectly with the ideal reflection angle, then $\vec{h}$ will align itself perfectly with the normal $\vec{n}$, and hence $(\vec{n} \cdot \vec{h})$ will be large. On the other hand, if eye deviates from the ideal reflection angle, then $\vec{h}$ will not align with $\vec{n}$, and $(\vec{n} \cdot \vec{h})$ will tend to decrease. Thus, we let $(\vec{n} \cdot \vec{h})$ be the geometric parameter which will define the strength of the specular component. (The original Phong model uses the factor $(\vec{r} \cdot \vec{v})$ instead.)

The parameters of surface finish that control specular reflection are ρ_s, the surface's *coefficient of specular reflection*, and *shininess*, denoted α. As α increases, the specular reflection drops off more quickly, and hence the size of the resulting shiny spot on the surface appears smaller as well. Shininess values range from 1 for low specular reflection up to, say, 1000, for highly specular reflection. The formula for the specular component is

$$I_s = \rho_s \max(0, \vec{n} \cdot \vec{h})^\alpha L_s.$$

As with diffuse, this is subject to attenuation.

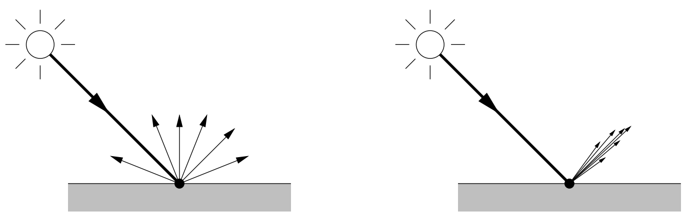


Figure 35: Shininess.

Putting it all together: Combining this with I_e (the light emitted from an object), the total reflected light from a point on an object of color C, being illuminated by a light source L, where the point is distance d from the light source using this model is:

$$\begin{aligned} I &= I_e + I_a + \frac{1}{a + bd + cd^2}(I_d + I_s) \\ &= I_e + \rho_a L_a C + \frac{1}{a + bd + cd^2}(\rho_d \max(0, \vec{n} \cdot \vec{\ell}) L_d C + \rho_s \max(0, \vec{n} \cdot \vec{h})^\alpha L_s), \end{aligned}$$

As before, note that this a vector equation, computed separately for the R, G, and B components of the light's color and the object's color. For multiple light sources, we add up the ambient, diffuse, and specular components for each light source.

Surface Normals: We mentioned one way for computing normals above based on taking the cross product of two vectors on the surface of the object. Here are some other ways.

Normals by Area: The method of computing normals by considering just three points is subject to errors if the points are nearly collinear or not quite coplanar (due to round-off errors). A better method is to consider all the points on the polygon. Suppose we are given a planar polygonal patch, defined by a sequence of n points $P_0, P_1, \ldots, P_{n-1}$. We assume that these points define the vertices of a polygonal patch.

Here is a nice method for determining the plane equation,

$$ax + by + cz + d = 0.$$

Once we have determined the plane equation, the normal vector has the coordinates (a, b, c).

The method makes use of the fact that the coefficients a, b, and c are proportional to the signed areas of the polygon's orthogonal projection onto the yz-, xz-, and xy-coordinate planes, respectively. By a signed area, we mean that if the projected polygon is oriented clockwise the signed area is positive and otherwise it is negative. How to compute the projected area of a polygon? Let us consider the xy-projection for concreteness. The idea is to break the polygon's area into the sum of signed trapezoid areas. See the figure below.

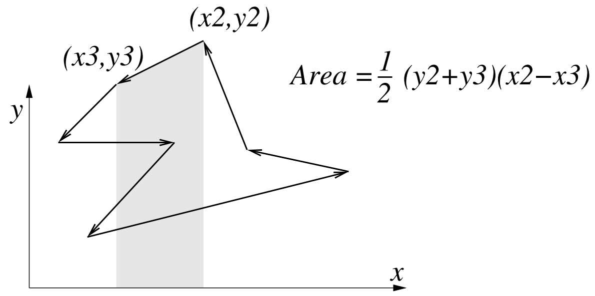


Figure 36: Area of polygon.

Assume that the points are oriented counterclockwise around the boundary. For each edge, consider the trapezoid bounded by that edge and its projection onto the x-axis. The area of the trapezoid will be positive if the edge is directed to the left and negative if it is directed to the right.

The cute observation is that even though the trapezoids extend outside the polygon, its area will be counted correctly. Every point inside the polygon is under one more left edge than right edge and so will be counted once, and each point under the polygon is under the same number of left and right edges, and these areas will cancel.

Summing the areas of the trapezoids yields:

$$\begin{aligned} a &= \frac{1}{2}\sum_{i=1}^{n}(z_i + z_{i+1})(y_i - y_{i+1}) \\ b &= \frac{1}{2}\sum_{i=1}^{n}(x_i + x_{i+1})(z_i - z_{i+1}) \\ c &= \frac{1}{2}\sum_{i=1}^{n}(y_i + y_{i+1})(x_i - x_{i+1}) \end{aligned}$$

Finally, we normalize the vector (a, b, c) to unit length to get the normal vector $\vec{n}$.

Normals for Implicitly Surfaces: Given a surface defined by an *implicit representation*, e.g. the set of points that satisfy some equation, $f(x, y, z) = 0$, then the normal at some point is given by *gradient vector*. This is a vector whose components are the partial derivatives of the function at this point

$$\vec{n} = \begin{pmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{pmatrix}.$$

As usual this should be normalized to unit length. (Recall that $\partial f/\partial x$ is computed by taking the derivative of f with respect to x and treating y and z as though they are constants.) See the text for an example.

Normals for Parametric Surfaces: Surfaces in computer graphics are more often represented parametrically. A *parametric representation* is one in which the points on the surface are defined by three function of 2 variables or *parameters*, say u and v:

$$\begin{aligned} x &= \phi_x(u,v), \\ y &= \phi_y(u,v), \\ z &= \phi_z(u,v). \end{aligned}$$

We will discuss this representation more later in the semester, but for now let us just consider how to compute a normal vector for some point $(\phi_x(u,v), \phi_y(u,v), \phi_z(u,v))$ on the surface.

To compute a normal vector, first compute the gradients with respect to u and v,

$$\frac{\partial \phi}{\partial u} = \begin{pmatrix} \partial \phi_x/\partial u \\ \partial \phi_y/\partial u \\ \partial \phi_z/\partial u \end{pmatrix} \qquad \frac{\partial \phi}{\partial v} = \begin{pmatrix} \partial \phi_x/\partial v \\ \partial \phi_y/\partial v \\ \partial \phi_z/\partial v \end{pmatrix},$$

and then return their cross product

$$\vec{n} = \frac{\partial \phi}{\partial u} \times \frac{\partial \phi}{\partial v}.$$

Lighting and Shading in OpenGL: To describe lighting in OpenGL there are three major steps that need to be performed: setting the lighting and shade model (smooth or flat), defining the lights, their positions and properties, and finally defining object material properties.

Lighting/Shading model: There are a number of parameters that can be set through the command `glLightModel()`, which were mentioned in the previous lecture. One important parameter that we didn't mention is whether polygons are to be drawn with flat shading (every point in the polygon having the same shading) or smooth shading (where shading varies across the surface by interpolating the vertex shading). This is set by the following command, whose argument is either `GL_SMOOTH` (the default) or `GL_FLAT`.

```
glShadeModel(GL_SMOOTH);
```

The shading interplation can be handled in one of two ways. In the classical *Gouraud interpolation* the illumination is computed exactly at the vertices (using the above formula) and the values are interpolated across the polygon. In *Phong interpolation*, the normal vectors are given at each vertex, and the system interpolates these vectors in the interior of the polygon. Then this interpolated normal vector is used in the above lighting equation. This produces more realistic images, but takes considerably more time. OpenGL uses Gouraud shading. Just before a vertex is given (with `glVertex*()`), you should specify its normal vertex (with `glNormal*()`).

Create/Enable lights: We discussed enabling lighting and lights in a previous lecture.

Define surface materials: When lighting is in effect, rather than specifying colors using `glColor()` you do so by setting the material properties of the objects to be rendered. OpenGL computes the color based on the lights and these properties. Surface properties are assigned to vertices (and not assigned to faces as you might think). In smooth shading, this vertex information (for colors and normals) are interpolated across the entire face. In flat shading the information for the first vertex determines the color of the entire face.

Every object in OpenGL is a polygon, and in general every face can be colored in two different ways. In most graphic scenes, polygons are used to bound the faces of solid polyhedra objects and hence are only to be seen from one side, called the *front face*. This is the side from which the vertices are given in counterclockwise order. By default OpenGL, only applies lighting equations to the front side of each polygon and the back side is drawn in exactly the same way. If in your application you want to be able to view polygons from both sides, it is possible to change this default (using `glLightModel()` so that each side of each face is colored and shaded independently of the other. We will assume the default situation.

Recall from the Phong model that each surface is associated with a single color and a coefficient for each type of reflection: emission, ambient, diffuse, and specular. In OpenGL, these two elements are combined into a single vector given as an RGB or RGBA value. For example, in the traditional Phong model, a red object might have a RGB color of $(1, 0, 0)$ and a diffuse coefficient of 0.5. In OpenGL, you would just set the diffuse material to $(0.5, 0, 0)$.

Surface material properties are specified by `glMaterialf()` and `glMaterialfv()`.

```
glMaterialf(GLenum face, GLenum pname, GLfloat param);
glMaterialfv(GLenum face, GLenum pname, const GLfloat *params);
```

It is possible to color the front and back faces separately. The first argument indicates which face we are coloring (`GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`). The second argument indicates the parameter name (`GL_EMISSION`, `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_SHININESS`). The last parameter is the value (either scalar or vector). See the OpenGL documentation for more information.

Other options: You may want to enable a number of GL options using `glEnable()`. This procedure takes a single argument, which is the name of the option. To turn each option off, you can use `glDisable()`. These optional include:

`GL_CULL_FACE`: Recall that each polygon has two sides, and typically you know that for your scene, it is impossible that a polygon can only be seen from its back side. For example, if you draw a cube with six square faces, and you know that the viewer is outside the cube, then the viewer will never see the back sides of the walls of the cube. There is no need for OpenGL to attempt to draw them. This can often save a factor of 2 in rendering time, since (on average) one expects about half as many polygons to face towards the viewer as to face away.

Backface culling is the process by which faces which face away from the viewer (the dot product of the normal and view vector is negative) are not drawn.

By the way, OpenGL actually allows you to specify which face (back or front) that you would like to have culled. This is done with `glCullFace()` where the argument is either `GL_FRONT` or `GL_BACK` (the latter being the default).

`GL_NORMALIZE`: Recall that normal vectors are used in shading computations. You supply these normal to OpenGL. These are assumed to be normalized to unit length in the

Phong model. Enabling this option causes all normal vectors to be normalized to unit length automatically. If you know that your normal vectors are of unit length, then you will not need this. It is provided as a convenience, to save you from having to do this extra work.

Lecture 15: Texture Mapping

(Thursday, Oct 19, 2000)
Read: Chapter 8 in Hill.

Surface Detail: We have discussed the use of lighting as a method of producing more realistic images. This is fine for smooth surfaces of uniform color (plaster walls, plastic cups, metallic objects), but many of the objects that we want to render have some complex surface finish that we would like to model. In theory, it is possible to try to model objects with complex surface finishes through extremely detailed models (e.g. modeling the cover of a book on a character by character basis) or to define some sort of regular mathematical texture function (e.g. a checkerboard or modeling bricks in a wall). But this may be infeasible for very complex unpredictable textures.

Textures and Texture Space : Although originally designed for textured surfaces, the process of *texture mapping* can be used to map (or "wrap") any digitized image onto a surface. For example, suppose that we want to render a picture of the Mona Lisa. We could download a digitized photograph of the painting, and then map this image onto a rectangle as part of the rendering process.

There are a number of common image formats which we might use. We will not discuss these formats. Instead, we will think of an image simply as a 2-dimensional array of RGB values. Let us assume for simplicity that the image is square, of dimensions $N \times N$ (OpenGL requires that N actually be a power of 2 for its internal representation). Images are typically indexed row by row with the upper left corner as the origin. The individual RGB pixel values of the texture image are often called *texels*, short for *texture elements*.

Rather than thinking of the image as being stored in an array, it will be a little more elegant to think of the image as function that maps a point (s, t) in 2-dimensional *texture space* to an RGB value. That is, given any pair (s, t), $0 \le s, t \le 1$, the texture image defines the value of $T(s, t)$ is an RGB value.

For example, if we assume that our image array Im is indexed by row and column from 0 to $N - 1$ starting from the upper left corner, and our texture space $T(s, t)$ is coordinatized by s (horizontal) and t (vertical) from the lower left corner, then we could apply the following function to round a point in image space to the corresponding array element:

$$T(s,t) = Im[\lfloor (1-t)N \rfloor, \lfloor sN \rfloor], \qquad \text{for } s,t \in (0,1).$$

In many cases, it is convenient to imagine that the texture is an infinite function. We do this by imagining that the texture image is repeated cyclically throughout the plane. This is sometimes called a *repeated texture*. In this case we can modify the above function to be

$$T(s,t) = Im[\lfloor (1-t)N \rfloor \bmod N, \lfloor sN \rfloor \bmod N], \qquad \text{for } s,t \in \mathbf{R}.$$

Parameterizations: We wish to "wrap" this 2-dimensional texture image onto a 2-dimensional surface. We need to define a wrapping function that achieves this. The surface resides in 3-dimensional space, so the wrapping function would need to map a point (s, t) in texture space to the corresponding point (x, y, z) in 3-space.

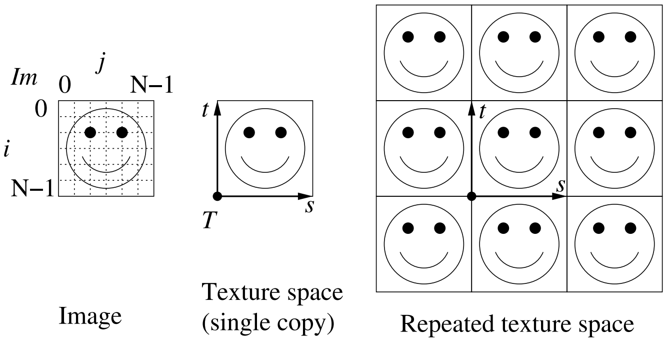


Figure 37: Texture space.

This is typically done by first computing a 2-dimensional *parameterization* of the surface. This means that we associate each point on the object surface with two coordinates (u, v) in *surface space*. Then we have three functions, $x(u, v)$, $y(u, v)$ and $z(u, v)$, which map the parameter pair (u, v) to the x, y, z-coordinates of the corresponding surface point. We then map a point (u, v) in the parameterization to a point (s, t) in texture space.

Let's make this more concrete with an example. Suppose that our shape is the surface of a unit sphere centered at the origin. We can represent any point on the sphere with two angles, representing the point's latitude and longitude. We will use a slightly different approach. Any point on the sphere can be expressed by two angles, ϕ and θ. (These will take the roles of the parameters u and v mentioned above.) Think of the vector from the origin to the point on the sphere. Let ϕ denote the angle in radians between this vector and the z-axis (north pole). (So ϕ is related to but not equal to the latitude.) we have $0 \leq \phi \leq \pi$. Let θ denote the counterclockwise angle of the projection of this vector onto the xy-plane. Thus $0 \leq \theta \leq 2\pi$.

What are the coordinates of the point on the sphere as a function of these two parameters? The z-coordinate is just $\cos\phi$, and clearly ranges from 1 to -1 as ϕ increases from 0 to π. The length of the projection of such a vector onto the x, y-plane will be $\sin\phi$. It follows that the x and y coordinates are related to the cosine and sine of angle θ, respectively, but scaled by this length. Thus we have

$$z(\phi,\theta) = \cos\phi, \qquad x(\phi,\theta) = \cos\theta\sin\phi, \qquad y(\phi,\theta) = \sin\theta\sin\phi.$$

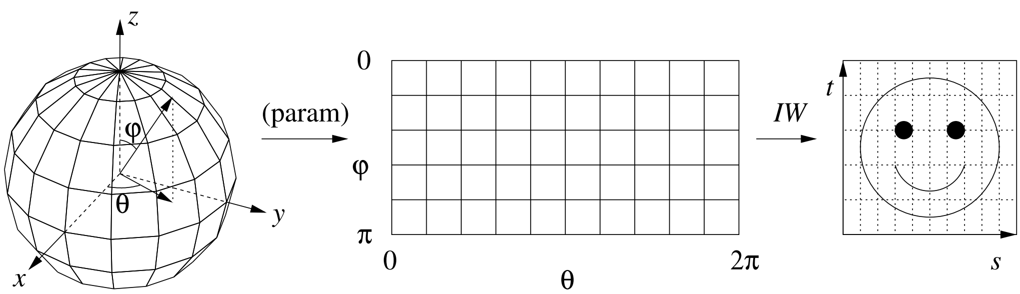


Figure 38: Parameterization of a sphere.

If we wanted to normalize the values of our parameters to the range $[0, 1]$, we could *reparameterize* by letting $u = \phi/\pi$ and $v = \theta/(2\pi)$. (As an exercise, see if you can do this for the traditional latitude and longitude representation, or try this for some other shapes, such as a cone or cylinder.)

If we are given a point (x, y, z) on the surface of the sphere, we could also derive the parameters by inverting this process. In particular

$$\phi = \arccos z \qquad \theta = \arctan(y/x),$$

and hence

$$u = \frac{\arccos z}{\pi} \qquad v = \frac{\arctan(y/x)}{2\pi}.$$

(Note that at the north and south poles there is a singularity in the sense that we cannot derive a unique value for θ.)

The *inverse wrapping* function $IW(u, v)$ maps a point on the parameterized surface to a point (s, t) in texture space. Intuitively, this is an "unwrapping" function, since it unwraps the surface back to the texture, but as we will see, this is what we need. In our simple example, we might just set this function to the identity, that is, $IW(u, v) = (u, v)$.

The Texture Mapping Process: Suppose that the unwrapping function IW, and a parameterization of the surface are given. Here is an overview of the texture mapping process. We will discuss some of the details below.

Project pixel to surface: First we consider a pixel that we wish to draw. We determine the *fragment* of the object's surface that projects onto this pixel, by determining which points of the object project through the corners of the pixel. (We will describe methods for doing this below.) Let us assume for simplicity that a single surface covers the entire fragment. Otherwise we should average the contributions of the various surfaces fragments to this pixel.

Parameterize: We compute the surface space parameters (u, v) for each of the four corners of the fragment. This generally requires a function for converting from the (x, y, z) coordinates of a surface point to its (u, v) parameterization.

Unwrap and average: Then we apply the inverse wrapping function to determine the corresponding region of texture space. Note that this region may generally have curved sides, if the inverse wrapping function is nonlinear. We compute the average intensity of the texels in this region of texture space, by computing a weighted sum of the texels that overlap this region, and then assign the corresponding average color to the pixel.

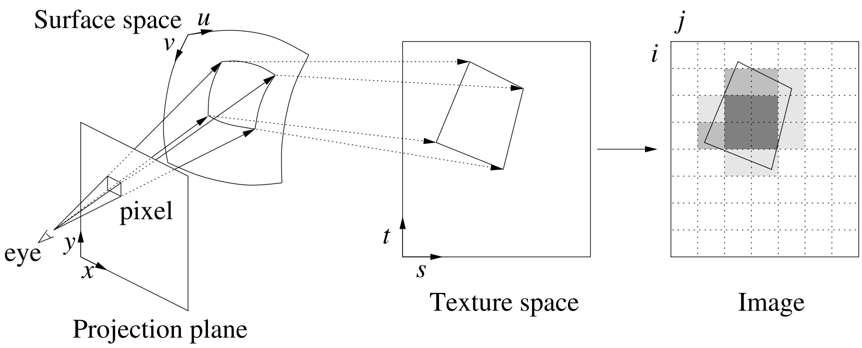


Figure 39: Texture mapping overview.

Texture Mapping Polygons: In OpenGL, all objects are polygons. This simplifies the texture mapping process. For example, suppose that a triangle is being drawn. Typically, when the vertices of the polygon are drawn, the user also specifies the corresponding (s,t) coordinates of these points in texture space. These are called *texture coordinates*. This implicitly defines a linear mapping from texture space to the surface of the polygon. These are specified *before* each vertex is drawn. For example, a texture-mapped object in 3-space with shading is drawn using the following structure.

```
glBegin(GL_POLYGON);
    glNormal3f(nx, ny, nz); glTexCoord2f(tx, ty); glVertex3f(vx, vy, vz);
    ...
glEnd();
```

There are two ways handle texture mapping in this context. The "quick-and-dirty" way (which is by far the faster of the two) is to first project the vertices of the triangle onto the viewport. This gives us three points P_0, P_1, and P_2 for the vertices of the triangle in 2-space. Let Q_0, Q_1 and Q_2 denote the three texture coordinates, corresponding to these points. Now, for any pixel in the triangle, let P be its center. We can represent P uniquely as an affine combination

$$P = \alpha_0 P_0 + \alpha_1 P_1 + \alpha_2 P_2 \qquad \text{for } \alpha_0 + \alpha_1 + \alpha_2 = 1.$$

So, once we compute the α_i's the corresponding point in texture space is just

$$Q = \alpha_0 Q_0 + \alpha_1 Q_1 + \alpha_2 Q_2.$$

Now, we can just apply our indexing function to get the corresponding point in texture space, and use its RGB value to color the pixel.

What is wrong with this approach? There are two problems, which might be very significant or insignificant depending on the context. The first has to do with something called *aliasing*. Remember that we said that after determining the fragment of texture space onto which the pixel projects, we should average the colors of the texels in this fragment. The above procedure just considers a single point in texture space, and does no averaging. In situations where the pixel corresponds to a point in the distance and hence covers a large region in texture space, this may produce very strange looking results, because the color of the pixel is determined entirely by the point in texture space that happens to correspond to the pixel's center.

The second problem has to do with perspective. This approach makes the incorrect assumption that affine combinations are preserved under perspective projection. This is not true. For example, after a perspective projection, the centroid of a triangle in 3-space is in general not mapped to the centroid of the projected triangle. (This is true for parallel projections, but not perspective projections.) Thus, projection followed by wrapping (using affine combinations in 2-space) is not the same as wrapping (using affine combinations in 3-space) and then projecting. The latter is what we should be doing, and the former is what this quick-and-dirty method really does.

There are a number of ways to fix this problem. One requires that you compute the inverse of the projection transformation. For each pixel, we map it back into three space, then compute the wrapping function in 3-space. (See Section 8.5.2 in Hill for a detailed discussion.) The other involve slicing the polygon up into small chunks, such that within each chunk the amount of distortion due to perspective is small.

Texture mapping in OpenGL: OpenGL supports a fairly general mechanism for texture mapping. The process involves a bewildering number of different options. You are refered to the OpenGL documentation for more detailed information. The very first thing to do is to enable texturing.

```
glEnable(GL_TEXTURE_2D);
```

The next thing that you need to do is to input your texture and present it to OpenGL in a format that it can access efficiently. It would be nice if you could just point OpenGL to an image file and have it convert it into its own internal format, but OpenGL does not provide this capability. You need to input your image file into an array of RGB (or possibly RGBA) values, one byte per color component (e.g. 3 bytes per pixel), stored row by row, from upper left to lower right. By the way, OpenGL requires images whose height and widths are powers of 2.

Once the array is input, call the procedure `glTexImage2D()` to have the texture processed into OpenGL's internal format. Here is the calling sequence. There are many different options, which are explained in the documentation.

```
glTexImage2d(GL_TEXTURE_2D, level, internalFormat, width, height, border,
    format, type, image);
glTexImage2d(GL_TEXTURE_2D, 0, GL_RGB, 512, 256, 0,
    GL_RGB, GL_UNSIGNED_BYTE, myImage);
```

Once the image has been input and presented to OpenGL, we need to tell OpenGL how it is to be mapped onto the surface. Again, OpenGL provides a large number of different methods to map a surface. The two most common are `GL_DECAL` which simply makes the color of the pixel equal to the color of the texture, and `GL_MODULATE` (the default) which makes the colors of the pixel the product of the color of the pixel (without texture mapping) times the color of the texture. This latter option is applied when shading is used, since the shading is applied to the texture as well. A example is:

```
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

The last step is to specify the how the texture is to be mapped to each polygon that is drawn. For each vertex drawn by `glVertex*()`, specify the corresponding texture coordinates, as we discussed earlier.

Lecture 16: Bump and Environment Mapping

(Tuesday, Oct 24, 2000)
Read: Chapter 8 in Hill.

Bump mapping: Texture mapping is good for changing the surface color of an object, but we often want to do more. For example, if we take a picture of an orange, and map it onto a sphere, we find that the resulting object does not look realistic. The reason is that there is an interplay between the bumpiness of the orange's peel and the light source. As we move our viewpoint from side to side, the specular reflections from the bumps should move as well. However, texture mapping alone cannot model this sort of effect. Rather than just mapping colors, we should consider mapping whatever properties affect local illumination. One such example is that of mapping surface normals, and this is what *bump mapping* is all about.

What is the underlying reason for this effect? The bumps are too small to be noticed through perspective depth. It is the subtle variations in *surface normals* that causes this effect. At first it seems that just displacing the surface normals would produce a rather artificial effect (for example, the outer edge of the object's boundary will still appear to be perfectly smooth). But in fact, bump mapping produces remarkably realistic bumpiness effects.

Here is an overview of how bump mapping is performed. As with texture mapping we are presented with an image that encodes the bumpiness. Think of this as a monochrome (gray-scale) image, where a large (white) value is the top of a bump and a small (black) value is a valley between bumps. As with texture mapping, it will be more elegant to think of this discrete image as an encoding of a continuous 2-dimensional *bump space*, with coordinates s and t. The gray-scale values encode a function called the *bump displacement function* $b(s,t)$, which maps a point (s,t) in bump space to its (scalar-valued) height. As with texture mapping, there is an *inverse wrapping function* IW which maps a point (u,v) in the object's surface parameter space to (s,t) in bump space.

Perturbing normal vectors: Let us think of our surface as a parametric function in the parameters u and v. That is, each point $P(u,v)$ is given by three coordinate functions $x(u,v)$, $y(u,v)$, and $z(u,v)$. Consider a point $P(u,v)$ on the surface of the object (which we will just call P). Let $\vec{n}$ denote the surface normal vector at this point. Let $(s,t) = IW(u,v)$, so that $b(s,t)$ is the corresponding bump value. The question is, what is the *perturbed normal* $\vec{n}'$ for the point P according to the influence of the bump map? Once we know this normal, we just use it in place of the true normal in our Phong illumination computations.

Here is a method for computing the perturbed normal vector. The idea is to imagine that the bumpy surface has been wrapped around the object. The question is how do these bumps affect the surface normals? This is illustrated in the figure below.

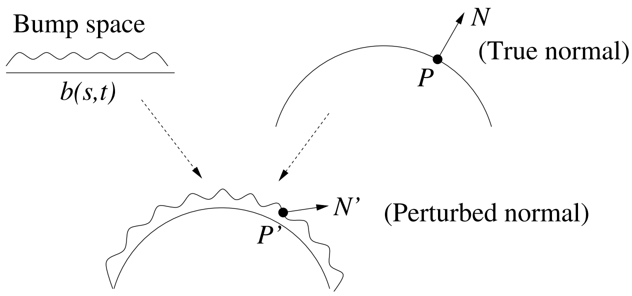


Figure 40: Bump mapping.

Since P is a function of u and v, let P_u denote the partial derivative of P with respect to u and define P_v similarly with respect to v. Since P has three coordinates, P_u and P_v can be thought of as three dimensional vectors. Intuitively, P_u and P_v are tangent vectors to the surface at point P. It follows that the normal vector $\vec{n}$ is (up to a scale factor) given by

$$\vec{n} \;=\; P_u \times P_u \;=\; \begin{pmatrix} \partial x/\partial u \\ \partial y/\partial u \\ \partial z/\partial u \end{pmatrix} \times \begin{pmatrix} \partial x/\partial v \\ \partial y/\partial v \\ \partial z/\partial v \end{pmatrix}.$$

Since $\vec{n}$ may not generally be of unit length, we define $\hat{n} = \vec{n}/|\vec{n}|$ to be the normalized normal.

If we apply our bump at point P, it will be elevated by an amount $b = b(u,v)$ in the direction of the normal. So we have

$$P' = P + b\hat{n},$$

is the elevated point. Note that just like P, the perturbed point P' is really a function of u and v. We want to know what the (perturbed) surface normal should be at P'. But this requires that we know its partial derivatives with respect to u and v. Letting $\vec{n'}$ denote this perturbed normal we have

$$\vec{n}' = P'_u \times P'_v,$$

where P'_u and P'_v are the partials of P' with respect to u and v, respectively. Thus we have

$$P'_u = \frac{\partial}{\partial u}(P + b\hat{n}) = P_u + b_u\hat{n} + b\hat{n}_u.$$

Assuming that the bump b is small, we can neglect the last term, and write

$$P'_u \approx P_u + b_u\hat{n} \qquad P'_v \approx P_v + b_v\hat{n}$$

Taking the cross product we have

$$\begin{aligned}\vec{n}' &\approx (P_u + b_u\hat{n}) \times (P_v + b_v\hat{n}) \\ &\approx (P_u \times P_v) + b_v(P_u \times \hat{n}) + b_u(\hat{n} \times P_v) + b_u b_v(\hat{n} \times \hat{n}).\end{aligned}$$

Since $\hat{n} \times \hat{n} = 0$ and $(P_u \times \hat{n}) = -(\hat{n} \times P_u)$ we have

$$\vec{n}' \approx \vec{n} + b_u(\hat{n} \times P_v) - b_v(\hat{n} \times P_u).$$

The partial derivatives b_u and b_v depend on the particular parameterization of the object's surface. If we assume that the object's parameterization has been constructed in common alignment with the image, then we have the following formula

$$\vec{n}' \approx \vec{n} + b_s(\hat{n} \times P_v) - b_t(\hat{n} \times P_u).$$

If we have an explicit representation for $P(u, v)$ and $b(s, t)$, then these partial derivatives can be computed by calculus. If the surface a polygonal, then P_u and P_v are constant vectors over the entire surface, and are easily computed. Typically we store $b(s, t)$ in an image, and so do not have an explicit representation, but we can approximate the derivatives by taking finite differences.

In summary, for each point P on the surface with (smooth) surface normal $\vec{n}$ we apply the above formula to compute the perturbed normal $\vec{n}'$. Now we proceed as we would in any normal lighting computation, but instead of using $\vec{n}$ as our normal vector, we use $\vec{n}'$ instead. As far as I know, OpenGl does not support bump mapping.

Environment Mapping: Next we consider another method of applying surface detail to model reflective objects. Suppose that you are looking at a shiny waxed floor, or a metallic sphere. We have seen that we can model the shininess by setting a high coefficient of specular reflection in the Phong model, but this will mean that the only light sources will be reflected (as bright spots). Suppose that we want the surfaces to actually reflect the surrounding environment. This sort of reflection of the environment is often used in commercial computer graphics. The shiny reflective lettering and logos that you see on television, the reflection of light off of water, the shiny reflective look of a automobile's body, are all examples.

The most accurate way for modeling this sort of reflective effect is through ray-tracing (which we will discuss later in the semester). Unfortunately, ray-tracing is a computationally intensive technique. To achieve fast rendering times at the cost of some accuracy, it is common to apply an alternative method called *environment mapping* (also called *reflection mapping*).

What distinguishes reflection from texture? When you use texture mapping to "paint" a texture onto a surface, the texture stays put. For example, if you fix your eye on a single point of the surface, and move your head from side to side, you always see the same color (perhaps with variations only due to the specular lighting component). However, if the surface is reflective, as you move your head and look at the same point on the surface, the color changes. This is because reflection is a function of the relationships between the viewer, the surface, and the environment.

Computing Reflections: How can we encode such a complex reflective relationship? The basic question that we need to answer is, given a point on the reflective surface, and given the location of the viewer, determine what the viewer sees in the reflection. Before seeing how this is done in environment mapping, let's see how this is done in the more accurate method called *ray tracing*. In ray tracing we track the path of a light photon backwards from the eye to determine the color of the object that it originated from. When a photon strikes a reflective surface, it bounces off. If $\vec{v}$ is the (normalized) view vector and $\vec{n}$ is the (normalized) surface normal vector, then just as we did in the Phong model, we can compute the *view reflection vector*, $\vec{r}_v$, for the view vector as

$$\vec{r}_v = 2(\vec{n} \cdot \vec{v})\vec{n} - \vec{v}.$$

(See Lecture 14 for a similar derivation of the light reflection vector.)

To compute the "true" reflection, we should trace the path of this ray back from the point on the surface along $\vec{r}_v$. Whatever color this ray hits, will be the color that the viewer observes as reflected from this surface.

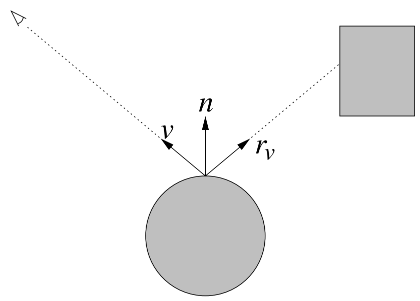


Figure 41: Reflection vector.

Unfortunately, it is expensive to shoot rays through 3-dimensional environments to determine what they hit. (However, this is exactly how the method of ray-tracing works.) We would like to do what we did in texture mapping, and just look the answer up in a precomputed image. To make this tractable, we will make one simplifying assumption.

Distant Environment: The reflective surface is small in comparison with the distances to the objects being reflected in it.

For example, the reflection of a room surrounding a silver teapot would satisfy this requirement. However, if the teapot is sitting on a table, then the table would be too close (resulting in a distorted reflection). The reason that this assumption is important is that the main parameter in determining what the reflection ray hits is the *direction* of the reflection vector, and not the location on the surface from which the ray starts. The space of directions is a 2-dimensional space, implying that we can precompute this information and store it in a 2-dimensional image array.

The environment mapping process: Here is a sketch of how environment mapping can be implemented. The first thing you need to do is to compute the environment map. First off remove the reflective object from your environment. Place a small sphere or cube about the center of the object. It should be small enough that it does not intersect any of the surrounding objects. (You may actually use any convex object for this purpose. Spheres and cubes each have advantages and disadvantages. We will assume the case of a cube in the rest of the discussion.) Project the entire environment onto the six faces of the cube, using the center of the cube as the center of projection. That is, take six separate pictures which together form a complete panoramic picture of the surrounding environment, and store the results in six image files. It may take some time to compute these images, but once they have been computed they can be used to compute reflections from all different viewing positions.

By the way, an accurate representation of the environment is not always necessary. For example, to simulate a shiny chrome-finished surface, a map with varying light and dark regions is probably good enough to fool the eye. This is called *chrome mapping*. But if you really want to simulate a mirrored surface, a reasonably accurate environment map is necessary.

Now suppose that we want to compute the color reflected from some point on the object. As in the Phong model we compute the usual vectors: normal vector $\vec{n}$, view vector $\vec{v}$, etc. We compute the view reflection vector $\vec{r}_v$ from these two. (This is not the same as the light reflection vector, $\vec{r}$, which we discussed in the Phong model, but it is the counterpart where the reflection is taken with respect to the viewer rather than the light source.) To determine the reflected color, we imagine that the view reflection vector $\vec{r}_v$ is shot from the center of the cube and determine the point on the cube which is hit by this ray. We use the color of this point to color the corresponding point on the surface. (We will leave as an exercise the problem of mapping a vector to a point on the surface of the cube.) The process is illustrated below.

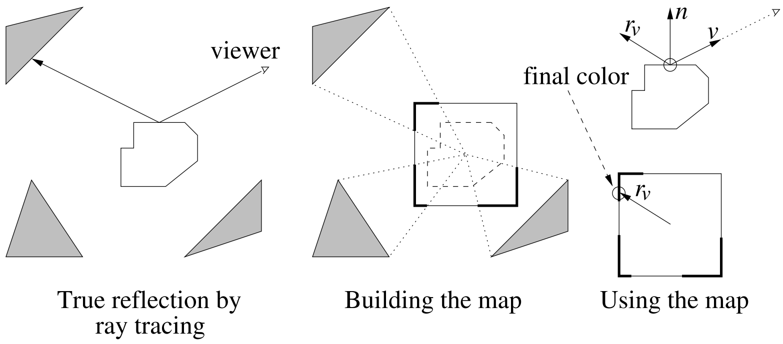


Figure 42: Environment mapping.

Note that the final color returned by the environment map is a function of the contents of the environment image and $\vec{r}_v$ (and hence of $\vec{v}$ and $\vec{n}$). In particular, it is *not* a function of the location of the point on the surface. Wouldn't taking this location into account produce more accurate results? Perhaps, but by our assumption that objects in the environment are far away, the directional vector is the most important parameter in determining the result. (If you really want accuracy, then use ray tracing instead.)

Reflection mapping through texture mapping: OpenGL does not support environment mapping directly, but there is a reasonably good way to "fake it" using texture mapping. Consider a polygonal face to which you want to apply an environment map. They key question is how to compute the point in the environment map to use in computing colors. The solution is to compute this quantities yourself for each vertex on your polygon. That is, for each vertex on the polygon, based on the location of the viewer (which you know), and the location of the vertex (which you know) and the polygon's surface normal (which you can compute), determine the view reflection vector. Use this vector to determine the corresponding point in the environment map. Repeat this for each of the vertices in your polygon. Now, just treat the environment map as though it were a texture map.

What makes the approach work is that when the viewer shifts positions, you will change the texture coordinates of your vertices. In normal texture mapping, these coordinates would be fixed, independent of the viewer's position.

Lecture 17: Review for Midterm

(Thursday, Oct 26, 2000)

The midterm exam will be Tues, Oct 31. The exam is closed-book and closed-notes, but you will be allowed one sheet of notes (front and back).

Read: You are responsible only for the material covered in class, but it is a good idea to consult Hill's book for additional insights on the concepts discussed in class. We have discussed materials from Chapts 1–5 and 7–8 in Hill.

Overview: So far we have covered the main elements to a top-down approach to graphics. The main topics we have discussed include a general introduction to graphics systems, basic OpenGL, affine, euclidean and projective geometry and geometric programming, perspective, and shading. Here is a summary of the topics we discussed.

Graphics Systems and Models: General overview of graphics systems and their structures: raster graphics, RGB color and color maps, basic elements of the graphics pipeline: transformation, projection, clipping, rasterization. When lighting is enabled, it enters in two places, first after the transformations are performed, the system also transforms the normal vectors, and then for each vertex we computes its illumination under the current lighting model. Then when rasterization is performed the colors at the vertices are interpolated to shade the interior of the polygon.

Affine and Euclidean Geometry: Recall that affine geometry has points and free vectors, their associated operations, plus affine combinations. Points and vectors in 3-dimensional affine space can be represented nicely by 4-element homogeneous coordinates, where the last component is either 1 for points or 0 for vectors. Euclidean geometry is an extension to affine geometry, in which we add dot products, and hence the notions of angle and distance. We also discussed orientations and using orientation tests for determining intersections.

Coordinate Systems and Frames: Transformations and matrices are generally used for two different tasks. One is moving points and objects in a space, and the other is for changing between different coordinate systems. Most graphics books tend to blur this distinction, and describe everything in terms of matrices, but it is important to keep the distinction clear in your mind.

A coordinate frame in 3-space is defined by an origin point and 3 linearly independent vectors. Given two coordinate frames in 3-space, there is a matrix that transforms coordinates given in one frame to their representation relative to the other frame. You should know how to construct these matrices.

Affine Transformations: Affine transformations are maps of affine space that preserve affine combinations. Such a transformation in 3-space can be described as a 4×4 matrix (whose last row is always $(0, 0, 0, 1)$). There are two methods for constructing affine transformation, one by constructing two frames, a source F_s and destination F_d, and then determining the matrix that maps from one to the other. You should know both methods for constructing transformations.

Projective Geometry: In projective geometry we do away with free vectors and any notion of orientation, but add points at infinity. Recall that a point at infinity does not have a location, but can be pointed towards. Points in projective 3-space can be represented by 4-element homogeneous coordinates, but the representation is not unique in that all nonzero scalar multiples of a vector represent the same point. Any projective transformation can be expressed as a 4×4 matrix followed by a *normalization*, in which we divide all components of the vector through by its last coordinate (assuming it is not zero).

Perspective: There are two types of projections in computer graphics, parallel and perspective. (An orthogonal projection is a special case of parallel projections where the projection direction is orthogonal to the viewing plane.) Perspective transformations are central to computer graphics. We showed in how to describe projection in terms of a viewing frustum, and to produce a projective transformation (in the form of a 4×4 matrix), which together with perspective division, maps the frustum into a 3-d rectangle. This was called *perspective with depth* because the (x, y)-coordinates of the transformed points are the projection of the point, and the z-coordinate encodes the distance from the viewer.

OpenGL: You are responsible for knowing the basic semantics, but not the exact calling sequences for the OpenGL commands needed for 2-dimensional drawing (Project 1), along with the commands for performing 3-dimensional transformations and perspective. I am more interested in your knowledge of the basic capabilities of OpenGL, rather than the details of the calling sequences. In particular, be sure that you know about the matrix stacks and the following.

Transformations: `glTranslate*()`, `glRotate*()`, `glScale*()` (and the order that they should be performed in relative to drawing commands).

Viewing: `gluLootAt*()`.

Perspective: `glPerspective()`.

Shading and Lighting Models: We introduced a standard local shading model called the *Phong reflection model.* The idea behind this model is that all illumination could be broken down into a mixture of four pure components: emission, ambient, diffuse, and specular. We presented formulas for each of these components, and explained how lighting is handled in OpenGL. You are not responsible for the OpenGL procedures for lighting, but you should understand how OpenGL does lighting.

Surface Mappings: We discussed methods for approximating complex surface finish properties through texture mapping, bump mapping, and environment mapping. Recall that the key is to store the needed information as a function, stored in a large array, and then define the inverse wrapping function for mapping points on the surface of an object to the appropriate point in texture space. The mapping typically involves first parameterizing the surface in terms of two variables (u, v), and then defining the appropriate mapping function from parameter space to the texture space (s, t).

Lecture 18: Midterm

(Tuesday, Oct 31, 2000)
Midterm exam. No lecture today.

Lecture 19: Solid Modeling

(Thursday, Nov 2, 2000)
Read: Chapter 6 and Sect. 14.12 in Hill. Some of the material is not covered in Hill.

Solid Object Representations: We begin discussion of 3-dimensional object models. There is an important fundamental split in the question of how objects are to be represented. Two common choices are between representing the 2-dimensional boundary of the object, called a *boundary representation* or *b-rep* for short, and a volume-based representation, which is sometimes called *CSG* for *constructive solid geometry.* Both have their advantages and disadvantages. Boundary representations seem to be more flexible and general, and so we will start with them.

Polygonal Models: For now, we will consider just the very simplest of modeling techniques based on approximating 3-dimensional objects by piecewise flat (i.e. planar) polygonal surfaces. This representation is called a *(nonconvex) polyhedron* or a *polygonal mesh*. This is the approach that OpenGL assumes.

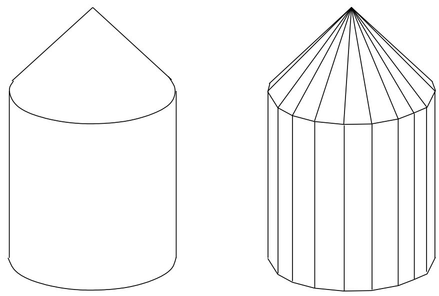


Figure 43: A polygonal mesh representation.

Even smooth objects are assumed to be approximated by a large number of small polygonal patches. It is easy to see from this example that polygonal meshes may not be the most space-efficient way of modeling smooth objects. The alternative that we will discuss later is based on *algebraic surfaces*. However there are a number of properties of polygonal meshes that make them easy to work with. For example, the intersection of two planar surfaces is (ignoring special cases) a line. Generally when intersecting algebraic surfaces of a particular type, the intersection curve may not be of the same type (or may be of higher algebraic degree). Another nice aspect of polygonal patches is that surface normal vectors (which are important for determining shading) are constant throughout the patch and operations such as linear interpolation work nicely.

Boundary Representations of Polygonal Meshes: Suppose we wish to represent a cube using a polygonal mesh. At its simplest we could simply store a set of polygonal faces that bound this object (and for rendering this is often all that is needed). This is typically fine for the most basic graphics applications, since if need just want to draw the object this is all that we require. However, it is often necessary to be able to reason more intelligently about the structure of the object. For example, to automatically compute smooth-shading in a polyhedral object, we might want to interpolate the normal vectors from neighboring faces. In order to perform this interpolation we will need to known which faces are adjacent to a given face.

This suggests that we need some sort of *adjacency information* or as it is often called in solid modeling *topological information*. Generally, a polygonal solid can be broken down into its constituent elements according to the dimension of these elements. There are *vertices* (0-dimensional), *edges* (1-dimensional), and *faces* (2-dimensional). This applies as well to objects bounded by curved sides and surfaces as well, but the edges and faces would not be linear. A complete description of the object's boundary would then consist of the set of its vertices, edges, and faces (and associated geometric information such as coordinates and surface normals), along with adjacency information indicating what is connected to what.

In order to present such a representation, we will need to make some rules about how these elements are joined to one another. For example, it seems natural that each edge should be incident to exactly two vertices and exactly two faces. Well behaved solids are called *2-manifolds*. The main property of a 2-manifold is that for any point on the boundary of the object, the local boundary is topologically equivalent into a 2-dimensional disk. In particular, this means that each edge is incident to exactly two faces, each vertex is incident to a single cycle of faces. However there exist strange sorts of solids where this might not be satisfied.

For example, the figure below left shows an edge that is incident to four faces and on the right we have a vertex incident to two separate "cones" of vertices. We will assume that all objects are 2-manifolds, or have modified so that they are in this form.

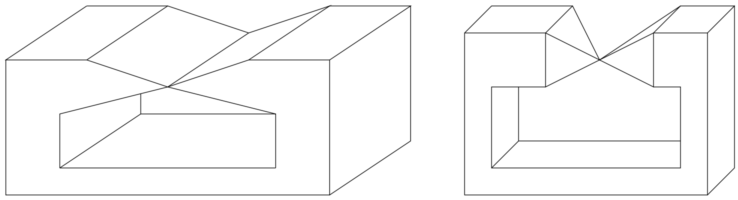


Figure 44: Non-manifold solids.

Given any 2-manifold, there is a relationship between the number of vertices, edges, and faces, which is given by *Euler's formula*. If we let V, E, and F denote the number of vertices edges and faces in an object then for most simple polyhedra we have

$$V - E + F = 2.$$

For example, for a cube we have $V = 8$, $E = 12$, and $F = 6$, and $8 - 12 + 6 = 2$. This can be generalized to polyhedra having "handles" (expressed as the *genus* of the polyhedron) and having multiple components. If G denotes the number of handles, and C denotes the number of components then we have

$$V - E + F = 2(C - G).$$

Doubly-connected Edge List: The representations for 2-manifolds are based on storing the various incidences between entities of one dimension and the next higher and/or lower in dimension. For example, common representations, called the *winged edge*, *half-edge*, *quad-edge* representations, are based on storing the following topological incidence information (in addition to whatever geometric and graphics properties are stored). We will discuss a representative examples, called the *doubly-connected edge list*, or *DCEL*.

The DCEL is an edge-based representation, but vertex and face information is also included for whatever geometric application is using the data structure. There are three sets of records one for each element in the PSLG: *vertex records*, a *edge records*, and *face records*. For the purposes of unambiguously defining left and right, each undirected edge is represented by two directed *half-edges*. See the figure for an example.

Vertex: Each vertex stores its coordinates, along with a pointer to any incident directed edge that has this vertex as its origin, `v.inc_edge`.

Face: Each face f stores a pointer to a single edge for which this face is the incident face, `f.inc_edge`. (See the text for the more general case of dealing with holes.)

Edge: Each undirected edge is represented as two directed edges. Each edge has a pointer to the oppositely directed edge, called its *twin*. Each directed edge has an *origin* and *destination* vertex. Each directed edge is associate with two faces, one to its left and one to its right.

We store a pointer to the origin vertex `e.org`. (We do not need to define the destination, `e.dest`, since it may be defined to be `e.twin.org`.)

We store a pointer to the face to the left of the edge `e.left` (we can access the face to the right from the twin edge). This is called the *incident face*. We also store the next and previous directed edges in counterclockwise order about the incident face, `e.next` and `e.prev`, respectively.

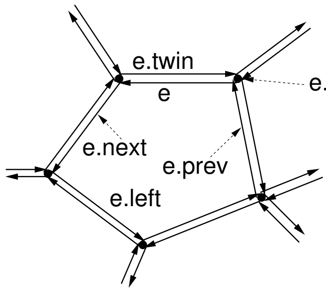


Figure 45: Doubly-connected edge list.

The next and prev pointers provide links around each face of the polygon. The next pointers are directed counterclockwise around each face and the prev pointers are directed clockwise. To get access to the edges of the opposite face for a given edge, access the twin pointer. Of course, in addition the data structure may be enhanced with whatever application data is relevant. In some applications, it is not necessary to know either the face or vertex information (or both) at all, and if so these records may be deleted. See the book for a complete example.

Volume Based Representations: Next, consider volume-based representations, and CSG in particular. One of the most intuitive ways to describe complex objects, especially those arising in manufacturing applications, is as set of *boolean operations* (that is, set union, intersection, difference) applied to a basic set of primitive objects. Manufacturing is an important application of computer graphics, and manufactured parts made by various milling and drilling operations can be described most naturally in this way. For example, consider the object shown in the figure below. It can be described as a rectangular block, minus the central rectangular notch, minus two cylindrical holes, and union with the rectangular block on the upper right side.

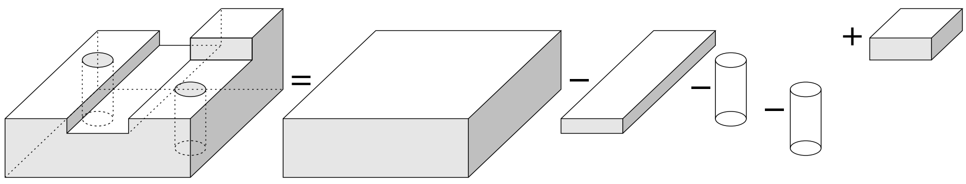


Figure 46: Constructive Solid Geometry.

This idea naturally leads to a tree representation of the object, where the leaves of the tree are certain *primitive object types* (rectangular blocks, cylinders, cones, spheres, etc.) and the internal nodes of the tree are *boolean operations*, union $\cup$, intersection ($\cap$), difference ($-$), etc. For example, the object above might be described with a tree of the following sort. (In the figure we have used $+$ for union.)

The primitive objects stored in the leaf nodes are represented in terms of a primitive *object type* (block, cylinder, sphere, etc.) and a set of defining *parameters* (location, orientation, lengths, radii, etc.) to define the location and shape of the primitive. The nodes of the tree are also labeled by transformation matrices, indicating the transformation to be applied to the object prior to applying the operation. By storing both the transformation and its inverse, as we traverse the tree we can convert coordinates from the world coordinates (at the root of the tree) to the appropriate local coordinate systems in each of the subtrees.

This method is called constructive solid geometry (CSG) and the tree representation is called a CSG tree. One nice aspect to CSG and this hierarchical representation is that once a complex

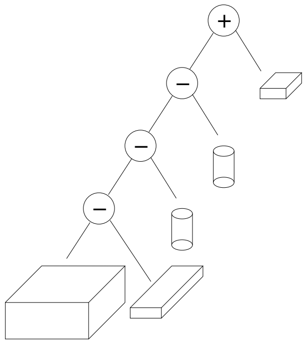


Figure 47: CSG Tree.

part has been designed it can be reused by replicating the tree representing that object. (Or if we share subtrees we get a representation as a directed acyclic graph or DAG.)

Point membership: CSG trees are examples of *unevaluated models*. For example, unlike a b-rep representation in which each individual element of the representation describes a feature that we know is a part of the object, it is generally impossible to infer from any one part of the CSG tree whether a point is inside, outside, or on the boundary of the object. As a ridiculous example, consider a CSG tree of a thousand nodes, whose root operation is the subtraction of a box large enough to enclose the entire object. The resulting object is the empty set! However, you could not infer this fact from any local information in the data structure.

Consider the simple membership question: Given a point P does P lie inside, outside, or on the boundary of an object described by a CSG tree. How would you write an algorithm to solve this problem? For simplicity, let us assume that we will ignore the case when the point lies on the boundary (although we will see that this is a tricky issue below).

The idea is to design the program recursively, solving the problem on the subtrees first, and then combining results from the subtrees to determine the result at the parent. We will write a procedure `Mem(P, T)` where P is the point, and T is pointer to a node in the CSG tree. This procedure returns True if the object defined by the subtree rooted at T contains P and False otherwise. If T is an internal node, let T.left and T.right denote the children of T. The algorithm breaks down into the following cases.

```
bool Mem(P, T)
{
    if (T->isLeaf) return (membership test appropriate to T's type);
    else if (T->isUnion)      return Mem(P, T->left ||  Mem(P, T->right);
    else if (T->isIntersect)  return Mem(P, T->left &&  Mem(P, T->right);
    else if (T->isDifference) return Mem(P, T->left && !Mem(P, T->right);
}
```

Note that the semantics of C operations `||` and `&&` will avoid making recursive calls when they are not needed. For example, in the case of union, if P lies in the right subtree, then the left subtree need not be searched.

This procedure can also be generalized to handle ray shooting intersections. Suppose that R is a ray, and T is a CSG tree. Determine the first intersection of R with any boundary in the object defined by T. The task is complicated slightly by the fact that the CSG object need not be convex, so that a ray may intersect the CSG object in a list of intervals. However the same hierarchical approach may be used to determine the first intersection.

Regularized boolean operations: There is a tricky issue in dealing with boolean operations. This goes back to a the same tricky issue that arose in polygon filling, what to do about object boundaries. Consider the intersection $A \cap B$ shown in the figure below. The result contains a "dangling" piece that has no width. That is, it is locally two-dimensional.

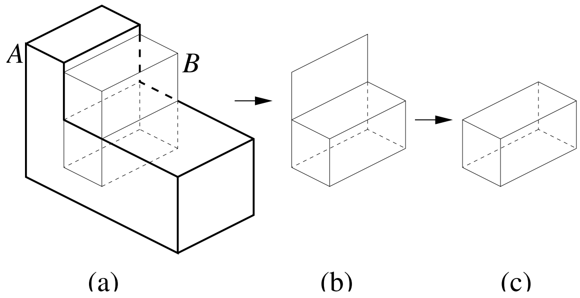


Figure 48: (a) A and B, (b) $A \cap B$, (c) $A \cap^* B$.

These low-dimensional parts can result from boolean operations, and are usually unwanted. For this reason, it is common to modify the notion of a boolean operation to perform a *regularization* step. Given a 3-dimensional set A, the regularization of A, denoted A^*, is the set with all components of dimension less than 3 removed. Topologically, A^* is defined to be the closer of the interior of A

$$A^* = \text{closure}(\text{int}(A)).$$

Note that $\text{int}(A)$ does not contain the dangling element, and then its closure adds back the boundary.

When performing operations in CSG trees, we assume that the operations are all *regularized*, meaning that the resulting objects are regularized after the operation is performed.

$$A \text{ op}^* \ B = \text{closure}(\text{int}(A \text{ op } B)).$$

where op is either $\cap$, $\cup$, or $-$.

Lecture 20: Bézier Curves

(Tuesday, Nov 7, 2000)
Read: Chapter 11 in Hill.

Representations: Today we discuss how curves and surfaces are represented in graphics systems. There are three standard approaches to representing a curve or surface in some dimensional space.

Explicit representation: In this form we represent one variable in terms of another $z = f(x, y)$. This representation is fine if there is only one y-value for each x-value, but it is impossible to represent curves for which this does not hold, spheres for example. The upper cap of a unit sphere could be represented by

$$z = \sqrt{x^2 + y^2}.$$

Because of this limitation, it is rarely used in graphics.

Implicit representation: In this representation a curve in 2-d and a surface in 3-d is represented as the zeros of a formula $f(x, y, z) = 0$. We have seen the representation of a sphere, e.g.

$$x^2 + y^2 + z^2 - 1 = 0.$$

It is common to place some restrictions on the possible classes of functions. A polynomial function is any function which can be expressed as a linear combination of integer powers of x, y, and z. We say that a curve or surface is *algebraic* if it can be expressed in this way as the zeroes of a polynomial function. The *degree* of an algebraic function is the highest sum of its powers (e.g. the term xy^2z is of degree $1 + 2 + 1 = 4$).

Implicit representation are fine for surfaces in 3-space, and in general for $(d-1)$-dimensional surfaces in d-dimensional space. But to represent a lower dimensional object, say a curve in 3-space we would need to compute the intersection of two such surfaces. This involves solving a of algebraic equations. These are generally messy to work with, limiting the popularity of this method.

Parametric representation: In this representation a curve in 2-d is given as three functions of one parameter $(x(u), y(u))$ and a surface in 3-d is given as function of two parameters $(x(u, v), y(u, v), z(u, v))$. An example is the parametric representation of a sphere

$$\begin{aligned} x(\theta, \phi) &= \cos\phi\cos\theta \\ y(\theta, \phi) &= \cos\phi\sin\theta \\ z(\theta, \phi) &= \sin\phi, \end{aligned}$$

for $0 \le \theta \le 2\pi$ and $-\pi/2 \le \phi \le \pi/2$. Notice that although the sphere has an algebraic implicit representation, it does not seem to have an algebraic parametric representation. (The one above involves trigonometric functions, which are not algebraic.)

Which representation is the best? It depends on the application. Implicit representations are nice, for example, for computing the intersection of a ray with the surface, or determining whether a point lies inside, outside, or on the surface. On the other hand, parametric representations are nice if you want to break the surface up into small polygonal elements for rendering. Parametric representations are nice because they are easy to subdivide into small patches for rendering, and hence they are popular in graphics. Sometimes (but not always) it is possible to convert from one representation to another. (It is easier to convert parametric representations to implicit, than the other way around.) We will concentrate on parametric representations in this lecture.

Continuity: Consider a parametric curve $P(u) = (x(u), y(u), z(u))^T$. An important condition that we would like our curves (and surfaces) to satisfy is that they should be as smooth as possible. How can we formalize this mathematically as follows. We would like the curves themselves to be continuous (that is not making sudden jumps in value). If the first k derivatives (as function of u) exist and are continuous, we say that the curve has *kth order parametric continuity*, denoted *C^k continuity*. Thus, 0th order continuity just means that the curve is continuous, 1st order continuity means that tangent vectors vary continuously, and so on.

Note that this definition is dependent on the particular parametric representation used. Since the same curve may be parameterized in different, some people suggest that a more appropriate definition in some circumstances is *geometric continuity*, denoted G^k, which depends solely on the shape of the curve, and not on the parameterization used. One way to achieve this is to assume that the curve has been parameterized by arc length (that is u is equal to the distance traveled along the curve) and then C^k continuity to this parameterization.

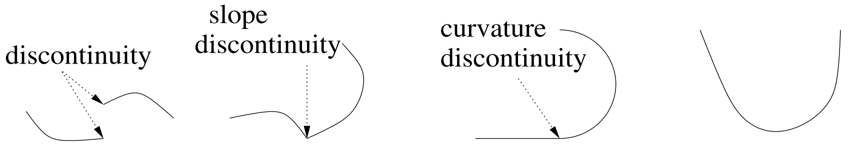


Figure 49: Degrees of continuity.

Generally we will want as high a continuity as we can get, but higher continuity generally comes with a higher computational cost. C^2 continuity is usually an acceptable goal.

Interactive Curve Design: For a designer who wishes to design a curve or surface, a symbolic representation of a curve as a mathematical formula is not very easy representation to deal with. A much more natural method to define a curve is to provide a sequence of *control points*, and to have a system which automatically generates a curve which approximates this sequence. Such a procedure inputs a sequence of points, and outputs a parametric representation of a curve. (This idea can be generalized to surfaces as well, but let's study it first in the simpler context of curves.)

It might seem most natural to have the curve pass through the control points, that is to *interpolate* between these points. There exists such an interpolating polygon, called the *Lagrangian interpolating polynomial.* However there are a number of difficulties with this approach. For example, suppose that the designer wants to interpolate a nearly linear set of points. To do so he selects a sequence of points that are very close to lying on a line. However, polynomials tend to "wiggle", and as a result rather than getting a line, we get a wavy curve passing through these points.

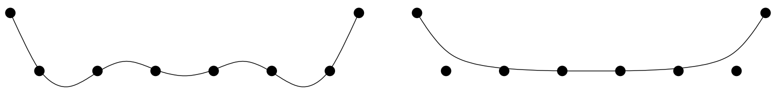


Figure 50: Interpolation versus approximation.

Instead our approach will be to merely *approximate* the control points. We will discuss two methods for doing this, called Bézier and B-spline curves.

The de Casteljau Algorithm: Let us continue to consider the problem of defining a smooth curve that approximates a sequence of control points, $\langle \mathbf{p}_0, \mathbf{p}_1, \ldots \rangle$. We begin with the simple idea on which these curves will be bsaed. Let us start with the simplest case of two control points. The simplest "curve" which approximates them is just the line segment $\overline{\mathbf{p}_0, \mathbf{p}_1}$. The function mapping a parameter u to a points on this segment involves a simple affine combination:

$$\mathbf{p}(u) = (1-u)\mathbf{p}_0 + u\mathbf{p}_1 \qquad \text{for } 0 \le u \le 1.$$

Observe that this is a weighted average of the points, and for any value of u, the two weighting or *blending functions* u and $(1-u)$ are nonnegative and sum to 1.

Now, let us consider how to generalize this to three points. We want a smooth curve approximating them. Consider the line segments $\overline{\mathbf{p}_0\mathbf{p}_1}$ and $\overline{\mathbf{p}_1\mathbf{p}_2}$. From linear interpolation we know how to interpolate a point on each, say:

$$\mathbf{p}_{01}(u) \;=\; (1-u)\mathbf{p}_0 + u\mathbf{p}_1 \qquad \mathbf{p}_{11}(u) \;=\; (1-u)\mathbf{p}_1 + u\mathbf{p}_2.$$

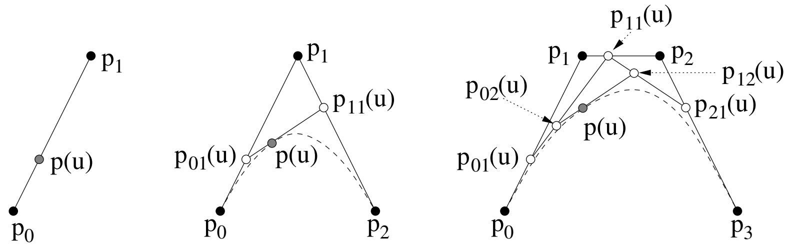


Figure 51: Repeated interpolation.

Now that we are down to two points, let us apply the above method to interpolate between them:

$$\begin{aligned}\mathbf{p}(u) &= (1-u)\mathbf{p}_{01}(u) + u\mathbf{p}_{11}(u) \\ &= (1-u)((1-u)\mathbf{p}_0 + u\mathbf{p}_1) + u((1-u)\mathbf{p}_1 + u\mathbf{p}_2) \\ &= (1-u)^2\mathbf{p}_0 + (2u(1-u))\mathbf{p}_1 + u^2\mathbf{p}_2.\end{aligned}$$

An example of the resulting curve is shown in the figure on the left.

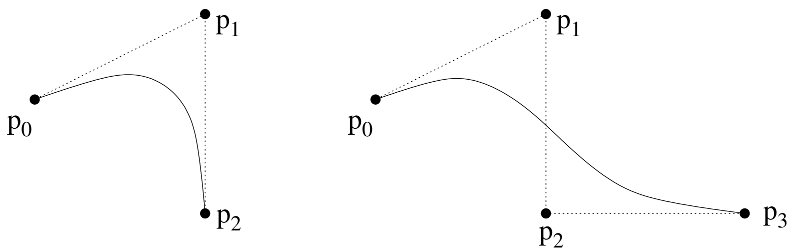


Figure 52: Bézier curves for three and four control points.

This is a algebraic parametric curve of degree 2, called a *Bézier curve* of degree 2. Observe that the function involves a a weighted sum of the control points using the following *blending functions*:

$$b_{02}(u) = (1-u)^2 \qquad b_{12}(u) = 2u(1-u) \qquad b_{22}(u) = u^2.$$

As before, observe that for any value of u the blending functions are all nonnegative and all sum to 1, and hence each point on the curve is a convex combination of the control points.

Let's carry this one step further. Consider four control points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$. First use linear interpolation between each pair yielding the points $\mathbf{p}_{01}(u)$ and $\mathbf{p}_{11}(u)$ and $\mathbf{p}_{21}(u)$ as given above. Then compute the linear interpolation between each pair of these giving

$$\mathbf{p}_{02}(u) = (1-u)\mathbf{p}_{01}(u) + u\mathbf{p}_{11}(u) \qquad \mathbf{p}_{12}(u) = (1-u)\mathbf{p}_{11}(u) + u\mathbf{p}_{21}(u).$$

Finally interpolate these $(1-u)\mathbf{p}_{02}(u) + u\mathbf{p}_{12}(u)$. Expanding everything yields

$$\mathbf{p}(u) = (1-u)^3\mathbf{p}_0 + (3u(1-u)^2)\mathbf{p}_1 + (3u^2(1-u))\mathbf{p}_2 + u^3\mathbf{p}_3.$$

This process of repeated interpolation is called the *de Casteljau algorithm*, named after a CAGD (computer-aided geometric design) designer working for a French automobile manufacturer. The final result is a Bézier curve of degree 3. Again, observe that if you plug in any value for u, these blending functions are all nonnegative and sum to 1. In this case, the blending functions are

$$\begin{aligned} b_{03}(u) &= (1-u)^3 \\ b_{13}(u) &= 3u(1-u)^2 \\ b_{23}(u) &= 3u^2(1-u) \\ b_{33}(u) &= u^3. \end{aligned}$$

Notice that if we write out the coefficients for the bending functions (adding a row for the degree 4 functions, which you can derive on your own), we get the following familiar pattern.

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

This is just the famous Pascal's triangle. In general, the ith blending function for the degree k Bézier curve has the general form

$$b_{ik}(u) = \binom{k}{i}(1-u)^{k-i}u^i.$$

These polynomial functions are important in mathematics, and are called the *Bernstein polynomials.*

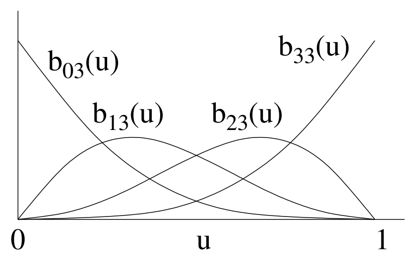


Figure 53: Bézier blending functions (Bernstein polynomials) of degree 3.

In the case of a cubic parametric curve we can express the final result in matrix form as

$$\mathbf{p}(u) = [1, u, u^2, u^3]\ \mathbf{M}_B \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix},$$

where $\mathbf{M}_B$ is called the *Bézier geometry matrix*

$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}.$$

The $\mathbf{p}_i$'s are interpreted as row vectors here.

Bézier curve properties: Bézier curves have a number of interesting properties. Because each point on a Bézier curve is a convex combination of the control points, the curve lies entirely within the convex hull of the control points. (This is not true of interpolating polynomials which can wiggle outside of the convex hull.) Observe that all the blending functions are 0 at $u = 0$ except the one associated with $\mathbf{p}_0$ which is 1 and so the curve starts at $\mathbf{p}_0$ when $u = 0$. By a symmetric observation, when $u = 1$ the curve ends at the last point. By evaluating the derivatives at the endpoints, it is also easy to verify that the curve's tangent at $u = 0$ is collinear with the line segment $\mathbf{p}_0\mathbf{p}_1$. A similar fact holds for the ending tangent and the last line segment.

If you compute the derivative of the curve with respect to u, you will find that it is itself a Bézier curve. Thus, the parameterized tangent vector of a Bézier curve is a Bézier curve. Finally the Bézier curve has the following *variation diminishing property.* Consider the polyline connecting the control points. Given any line ℓ, the line intersects the Bézier curve no more times than it intersects this polyline. Hence the sort of "wiggling" that we saw with interpolating polynomials does not occur with Bézier curves.

Lecture 21: Bezier Surfaces and B-splines

(Thursday, Nov 9, 2000)
Read: Chapter 11 in Hill.

Subdividing Bézier curves: Last time we introduced the mathematically elegant Bézier curves. Before going on to discuss surfaces, we need to consider one more issue. In order to render curves or surfaces using a system like OpenGL, which only supports rendering of flat objects, we need to approximate the curve by a number of small linear segments. Typically this is done by computing a sufficiently dense set of points along the curve or surface, and then approximating the curve or surface by a collection of line segments or polygonal patches, respectively. Bézier curves (and surfaces) lend themselves to a very elegant means of recursively subdividing them into smaller pieces. This is nice, because if we want to render a curve at varying resolutions, we can perform either a high number or low number of subdivisions. Furthermore, if part of the surface is visible and part is not, we can adaptively subdivide the surface where it visible, and leave the other part alone.

Here is a simple subdivision scheme works for these curves. Let $\langle \mathbf{p}_0, \ldots, \mathbf{p}_3 \rangle$ denote the original sequence of control points (this can be adapted to any number of points). Relabel these points as $\langle \mathbf{p}_{00}, \ldots, \mathbf{p}_{03} \rangle$. Perform the repeated interpolation construction using the parameter $u = 1/2$. Label the vertices as shown in the figure below. Now, consider the sequences $\langle \mathbf{p}_{00}, \mathbf{p}_{01}, \mathbf{p}_{02}, \mathbf{p}_{03} \rangle$ and $\langle \mathbf{p}_{03}, \mathbf{p}_{12}, \mathbf{p}_{21}, \mathbf{p}_{30} \rangle$. Each of these sequences defines its own Bézier curve. Amazingly, the concatenation of these two Bézier curves is equal to the original curve. (We will leave the proof of this as an exercise.)

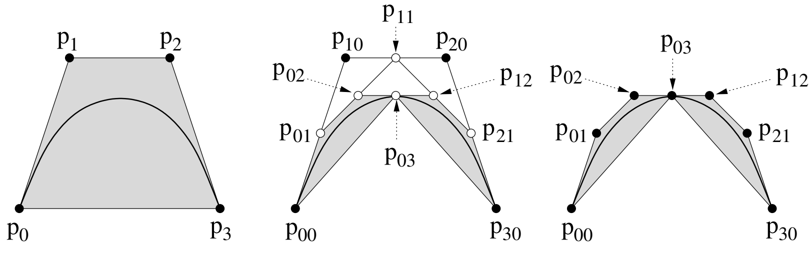


Figure 54: Bézier subdivision.

Repeating this subdivision allows us to split the curve into as small a set of pieces as we would like, and at all times we are given each subcurve in exactly the same form as the original, represented as a set of four control points. Typically this is done until each of the pieces is sufficiently close to being flat, or is sufficiently small.

Bézier Surfaces: Last time we defined Bézier curves. It is an easy matter to extend this notion to Bézier surfaces. Recall that Bézier curves were defined by a process of repeated interpolation. We can extend the notion of interpolation along a line to interpolation along two dimensions. This is called *bilinear interpolation*. Suppose that we are given four control points p_{00}, p_{01}, p_{10}, and p_{11}. (Note that the indexing has changed here relative to the previous section.) We use two parameters u and v. We interpolate between p_{00} and p_{01} using u, between p_{10} and p_{11} using u, and then interpolate between these two values using v.

$$\begin{aligned} p(u,v) &= (1-v)((1-u)p_{00} + up_{01}) + v((1-u)p_{10} + up_{11}) \\ &= (1-v)(1-u)p_{00} + (1-v)up_{01} + v(1-u)p_{10} + vup_{11} \end{aligned}$$

Note that this is not a linear interpolation (because u and v are multiplied times each other). Recalling that $(1-u)$ and u are the first-degree Bézier blending functions $b_{0,1}(u)$ and $b_{1,1}(u)$, we see that this can be written as

$$p(u,v) = b_{01}(v)b_{01}(u)p_{00} + b_{01}(v)b_{11}(u)p_{01} + b_{11}(v)b_{01}(u)p_{10} + b_{11}(v)b_{11}(u)p_{11}.$$

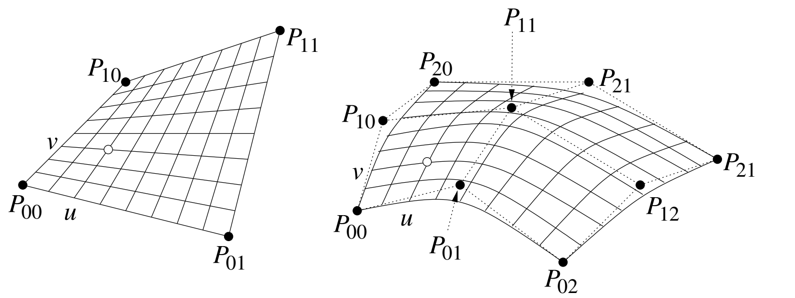


Figure 55: Bézier surfaces.

Generalizing this to higher degree, say cubic Bézier surfaces, we have we have a 4×4 array of control points, $\mathbf{p}_{ij}$, $0 \le i, j \le 3$, and the resulting parametric formula is

$$\mathbf{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_{i,3}(v) b_{j,3}(u) \mathbf{p}_{i,j}.$$

This is sometimes called a *tensor product* construction. (By the way, there are other ways of mapping curves to surfaces, which do not have quite such a restrictive row-column structure, but they are quite a bit more involved to explain.)

Observe that if we fix the value of v, then as u varies we get a Bézier curve. Similarly if we fix u and let v vary then it traces out a Bézier curve. The final surface is this combination of curves. It has the same convex hull and tangent properties that Bézier curves have.

How are Bézier surfaces rendered in OpenGL? We can generalize the subdivision process for curves in a straightforward manner (using $u = v = 1/2$). This will result in four sets of control points, where the union of the resulting surface patches is equal to the original surface. Again, the subdivision process may be repeated until each patch is sufficiently close to being flat or is sufficiently small, after which the resulting control points define the vertices of a polygon.

Cubic B-splines: Although Bézier curves are very elegant, they do have some shortcomings. The main problem is that if we want to define a single complex curve with many variations and wiggles, we need to have a large number of control points. But this leads to a high degree polynomial, hence more complex calculations. The fact that the Bézier blending functions are all nonzero over the entire range $u \in (0,1)$ means that these functions have *global support*. This means that the movement of even one control point has an effect on the entire curve (although it is most noticeable only in the region of the point). A system that provides for local support would be prefered, where each control point only affects a local portion of the curve.

One solution would be to link together a many low degree (e.g. cubic) Bézier curves end to end. Getting the joints to link with C^2 continuity (recall that this means that the function and its first two derivatives are continuous) is a bit tricky. (We will leave as an exercise the conditions on the control points that would guarantee this.) What we would like is a method of stringing many points together so that we get the best of all worlds: low degree, many control points, and C^2 (or higher) continuity.

B-splines were developed to address these shortcomings. The idea is that we will still use smooth blending functions multiplied times the control points, but these functions will have the property that these blending functions are nonzero only over a small amount of the parameter range. Thus these functions have only *local support.* Over the nonzero range, they will consist of the concatenation of smooth polynomials. As before each point on the curve will be given by blending the control points

$$\mathbf{p}(u) = \sum_{i=0}^{m} B_i(u)\mathbf{p}_i,$$

where $B_i(u)$ denotes the ith blending function. The figure below left gives a crude rendering of B-splines blending functions of order 2. We will not discuss B-spline surfaces explicitly, since they follow from exactly the same *tensor product* construction used with Bézier surfaces.

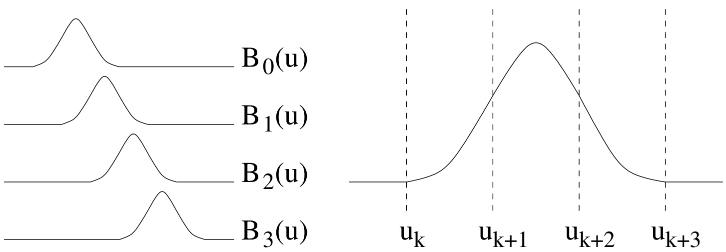


Figure 56: B-spline basis function.

Note that it is impossible to define a single polynomial that is zero on some range and nonzero on some other. So to define the B-spline blending functions we will need to subdivide the parameter space u into a set of intervals, and define a different polynomial over each interval. The result is a *piecewise polynomial function.* If we join the pieces with sufficiently high continuity then the resulting spline will have the same continuity. In the figure above right, each interval contains a different polynomial function.

The B-spline blending functions are a generalization of the Bézier blending functions. Let's suppose that we want to generate a curve of degree d. (The standard cubic B-spline will be the case $d = 3$.) Also let us assume that we have $m+1$ data points $\mathbf{p}_0, \ldots, p_m$. Rather than work over the interval $0 \le u \le 1$ as we did for Bézier curves, it will be notationally convenient to extend the range of u to a set of intervals:

$$u_{\min} = u_0 \le u_1 \le u_2 \le \ldots \le u_n = u_{\max}.$$

These parameter values are called *knot points.* (Note that the term "point" does not refer to a point in space, as with control points. These are just scalar values.) Each of the blending functions will consist of the concatenation of polynomial functions, with one polynomial over each *knot interval*, $[u_{i-1}, u_i]$. For simplicity you might think of these as being intervals of unit length for the time being, but we will see later that there are advantages to making intervals of different sizes. There will be a relationship between the number of intervals n and the number of points m, which we will consider later.

How do we define the B-spline blending functions? There are two ways to do this. The first is to write down the requirements that the blending functions must be C^2 continuous at the joint points, and that they satisfy the convex hull property. Together these constraints completely define B-splines. (We leave this as an exercise.)

Instead, as with the Bézier blending functions, we will do this by recursively applying linear interpolation to the blending functions of the next lower degree. An elegant recursive expression of the blending function (but somewhat difficult to understand) is given by the *Cox-deBoor recursion.* Let $B_{i,d}(u)$ denote the ith blending function for a B-spline of degree d.

$$\begin{aligned} B_{k,0}(u) &= \begin{cases} 1 & \text{if } u_k \le u < u_{k+1}, \\ 0 & \text{otherwise.} \end{cases} \\ B_{k,d}(u) &= \frac{u - u_k}{u_{k+d} - u_k} B_{k,d-1}(u) + \frac{u_{k+d+1} - u}{u_{k+d+1} - u_{k+1}} B_{k+1,d-1}(u). \end{aligned}$$

This is quite hard to comprehend at first sight. However observe that as with Bézier curves, the curve at each degree is expressed as the weighted average of two curves and the next lower degree. It can be proved (by induction) that irrespective of the knot spacing the blending functions sum to 1.

If you grind through the definitions, then you will see that $B_{k,0}(u)$ is a step function that is 1 in the interval $[u_k, u_{k+1})$. $B_{k,1}(u)$ spans two intervals and is a piecewise linear function that goes from 0 to 1 and then back to 0. $B_{k,2}(u)$ spans three intervals and is a piecewise quadratic that grows from 0 to 1/4, then up to 3/4 in the middle of the second interval, back to 1/4, and back to 0. Finally $B_{k,3}(u)$ is a cubic that spans four intervals growing from 0 to 1/6 to 2/3, then back to 1/6 and to 0. Thus, successively higher degrees are successively smoother.

Notice that only the basis case of the recursion is defined in a piecewise manner, but all the other functions inherit their piecewise nature from this.

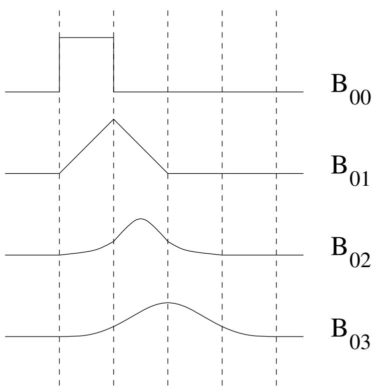


Figure 57: B-spline blending functions.

Knot Selection: We have left the issue of how to select the knot points unspecified. The simplest way to select the knots is to space them evenly. These are called *uniform B-splines.* However if you do this you will notice that you will not have the nice property that we had with Bézier curves, that the curve starts at the first point and ends with the last. The reason is that the Bézier blending functions summed to 1 for all parameter values and in particular the first blending function was equal to 1 for $u = 0$. However with B-splines, we do not naturally have this property.

If our curve is a closed curve, then we can define the blending function cyclically. If the curve is open we need a method to allow us to tie down the endpoints. This is done by allowing the knot points to vary nonuniformly with u. These are called *nonuniform B-splines*. In particular, if you repeat a knot value multiple times, then it has the effect of increasing the weight of the blending function at this parameter value. (In the Cox-deBoor formula, you will get the fraction 0/0 which should be interpreted as 1.) For example, if you repeat a knot point $d + 1$ times, then the blending function will grow to 1 at the corresponding parameter value, and the curve will pass through the corresponding data point at this parameter value.

So to tie our B-spline to its endpoints we apply the following procedure. Given $m + 1$ data points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_m$, create $m + 1 + 2d$ knot points. For example, for a cubic ($d = 3$) spline we use the knot values

$$\langle 0, 0, 0, 0, 1, 2, \ldots, m - 1, m, m, m, m\rangle.$$

(Adding 3 extra 0's and 3 extra m's.) Notice that this does not alter the number of control points, it simply causes the Cox-deBoor recursion to give added weight in the blending function to the first and last points when u is near $u_{\min}$ and $u_{\max}$, respectively.

B-splines curves and surfaces are a popular in geometric design applications.

The subdivision method for Bézier curves can also be generalized to B-splines. However there is another generalization of both that represents the state of the art in the field, which we discuss next.

NURBS: It turns out that there is one more class of splines that encompasses both Bézier curves and surfaces as well as B-splines, with the strange name *NURBS*, which stands for *nonuniform rational B-splines*. We will not discuss NURBS in detail, but just present a high-level intuition as to their origin.

A *rational function* is the ratio of two polynomials. There are curves that can be expressed as rational functions that cannot be expressed exactly using simple polynomials. For example, we gave a trigonometric parameterization of a circle. There is no known exact polynomial parameterization. However there is a rational parameterization of a unit circle on the xy-plane.

$$\begin{aligned} x(u) &= \frac{1 - u^2}{1 + u^2} \\ y(u) &= \frac{2u}{1 + u^2} \\ z(u) &= 0 \end{aligned}$$

Observe that as u tends from 0 to $+\infty$ this traces out the upper half of the circle (with $u = 1$ corresponding to 90 degrees) and from 0 to $-\infty$ it traces out the bottom half.

Here is one last clever use of homogeneous coordinates. Rather than limiting ourselves to computing splines in 3-space, let us put our control points in projective space (using homogeneous coordinates). For example the above curve could be given as a regular quadratic parameterization using homogeneous coordinate:

$$x(u) = 1 - u^2$$

$$\begin{aligned} y(u) &= 2u \\ z(u) &= 0 \\ w(u) &= 1 + u^2 \end{aligned}$$

Now, when we perform perspective normalization, the w-coordinate is divided through and we get a rational parameterization for free! This is essential idea behind NURBS: 4 dimensional nonuniform B-splines, followed by normalization.

A nice property of NURBS is that they are preserved under projective transformations. Thus, to render the projection of a NURB curve, project the control points and then render the NURB resulting from the projected control points.

Lecture 22: Ray Tracing

(Tuesday, Nov 14, 2000)
Read: Chapter 14 in Hill.

Ray Tracing: Ray tracing is among the conceptually simplest methods for synthesizing very realistic images. Unlike the simple polygon rendering methods used by OpenGL, ray tracing can easily produce shadows, and it can model reflective and transparent objects. Ray tracing also forms the basis of many approaches to more complex types of image generation (for example, it is used in Monte Carlo radiosity). In spite of its conceptual simplicity, ray tracing can be computationally quite intensive, and hence it is not usually used in interactive contexts. Today we will discuss the basic elements of ray tracing, and next time we will discuss the details of handling ray intersections in greater detail.

The Basic Idea: Consider our standard perspective viewing scenario. There is a viewer located at some position, and in front of the viewer is the view plane, and on this view plane is a window. We want to render the scene that is visible to the viewer through this window. Consider an arbitrary point on this window. The color of this point is determined by the light ray that passes through this point and hits the viewer's eye.

More generally, light travels in rays that are emitted from the light source, and hit objects in the environment. When light hits a surface, some of its energy is absorbed, and some is reflected in different directions. (If the object is transparent, light may also be transmitted through the object.) The light may continue to be reflected off of other objects. Eventually some of these reflected rays find their way to the viewer's eye, and only these are relevant to the viewing process.

If we could accurately model the movement of all light in a 3-dimensional scene then in theory we could produce very accurate renderings. Unfortunately the computational effort needed for such a complex simulation would be prohibitively large. How might we speed the process up? Observe that most of the light rays that are emitted from the light sources never even hit our eye. Consequently the vast majority of the light simulation effort is wasted. This suggests that rather than tracing light rays as they leave the light source (in the hope that it will eventually hit the eye), instead we reverse things and trace backwards along the light rays that hit the eye. This is the idea upon which *ray tracing* is based.

Imagine that the viewing window is replaced with a fine mesh of horizontal and vertical grid lines, so that each grid square corresponds to a pixel in the final image. We shoot rays out from the eye through the center of each grid square in an attempt to trace the path of light backwards toward the light sources. Consider the first object that such a ray hits. We want to know the intensity of reflected light at this surface point. This depends on a number of things, principally the reflective and color properties of the surface, and the amount of light reaching this point from the various light sources. The amount of light reaching this surface point is the hard to compute accurately. This is because light from the various light sources might be blocked by other objects in the environment and it may be reflected off of others.

A purely local approach to this question would be to use the model we discussed in the Phong model, namely that a point is illuminated if the angle between the normal vector and light vector is acute. In ray tracing it is common to use a somewhat more global approximation. We will assume that the light sources are points. We shoot a ray from the surface point to each of the light sources. For each of these rays that succeeds in reaching a light source before being blocked another object, we infer that this point is illuminated by this source, and otherwise we assume that it is not illuminated, and hence we are in the shadow of the blocking object. (Can you imagine a situation in which this model will fail to correctly determine whether a point is illuminated?) This model is illustrated on the left in the following figure.

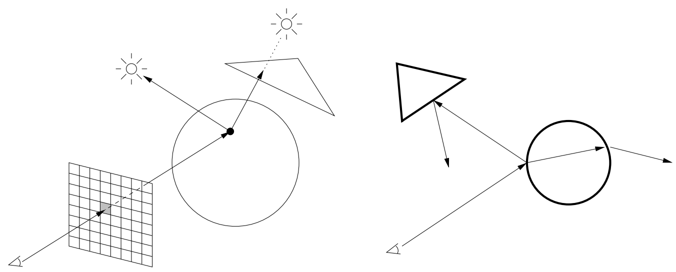


Figure 58: Ray Tracing.

Given the direction to the light source and the direction to the viewer, and the surface normal (which we can compute because we know the object that the ray struck), we have all the information that we need to compute the reflected intensity of the light at this point, say, by using the Phong model and information about the ambient, diffuse, and specular reflection properties of the object. We use this model to assign a color to the pixel. We simply repeat this operation on all the pixels in the grid, and we have our final image.

Even this simple ray tracing model is already better than what OpenGL supports (because, for example, OpenGL's local lighting model does not compute shadows). The ray tracing model can easily be extended to deal with reflective objects (such as mirrors and shiny spheres) and transparent objects (glass balls and rain drops). For example, when the ray hits a reflective object, we compute the reflection ray and shoot it into the environment. We invoke the ray tracing algorithm recursively. When we get the associated color, we blend it with the local surface color and return the result. The generic algorithm is outlined below.

Generic Ray Tracer Program

```
main() {
    for (i = 0; i < nRows; i++) {
        for (j = 0; j < nCols; j++) {
            Ray R = ray from eye through window row i and column j;
            image[i][j] = trace(R);
        }
    }
}
```

```
RGBColor trace(Ray R) {
    Shoot ray R into the scene and let X be the first object hit and
        P be the point of contact with this object;
    if (X is reflective) {
        Ray R1 = reflection ray at P;
        Color C1 = trace(R1);
    }
    if (X is transparent) {
        Ray R2 = refraction ray at P;
        Color C2 = trace(R2);
    }
    C = result of Phong model at P combining the effects of C1 and C2
        and the surface color at P;
    return C;
}
```

There are two questions to be considered. How to determine what object the ray intersects (which we will consider next time) and how to use this information to determine the reflected color. We will concentrate on this latter item today.

Reflection: Recall the Phong reflection model. Each object is associated with a color, and its coefficients of ambient, diffuse, and specular reflection, denoted ρ_a, ρ_d and ρ_s. To model the reflective component, each object will be associated with an additional parameter called the *coefficient of reflection*, denoted ρ_r. As with the other coefficients this is a number in the interval $[0, 1]$. Let us assume that this coefficient is nonzero. We compute the view reflection ray (which equalizes the angle between the surface normal and the view vector). Let $\vec{v}$ denote the normalized vector that points backwards along the viewing ray. (This is essentially the same as the view vector used in the Phong model, but it may not point directly back to the eye because of reflections.) Let $\vec{n}$ denote the outward pointing normalized surface normal vector. The normalized *view reflection vector*, denoted $\vec{r}_v$ was derived earlier this semester:

$$\vec{r}_v = 2(\vec{n} \cdot \vec{v})\vec{n} - \vec{v}.$$

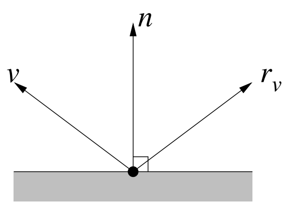


Figure 59: Reflection.

Next we shoot the ray emanating from the surface contact point along this direction and apply the above ray-tracing algorithm recursively. Eventually, when the ray hits a nonreflective object, the resulting color is returned. This color is then factored into the Phong model, as will be described below. Note that it is possible for this process to go into an infinite loop, if say you have two mirrors facing each other. To avoid such looping, it is common to have a maximum recursion depth, after which some default color is returned, irrespective of whether the object is reflective.

Transparent objects and refraction: To model refraction, sometimes called *transmission*, we maintain a coefficient of transmission, denoted ρ_t. We also need to associate each surface with two additional parameters, the *indices of refraction*[3] for the incident side η_i and the transmitted side, η_t. Recall from physics that the index of refraction is the ratio of the speed of light through a vacuum versus the speed of light through the material. Typical indices of refraction include water: 1.333, glass: 1.5, and diamond: 2.47. *Snell's law* says that if a ray is incident with angle θ_i (relative to the surface normal) then it will transmitted with angle θ_t (relative to the opposite normal) such that

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i}.$$

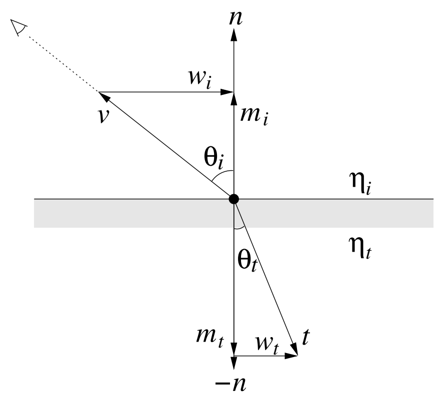


Figure 60: Refraction.

Let us work out the direction of the transmitted ray from this. As before let $\vec{v}$ denote the normalized view vector, directed back along the incident ray. Let $\vec{t}$ denote the unit vector along the transmitted direction, which we wish to compute. The orthogonal projection of $\vec{v}$ onto the normalized normal vector $\vec{n}$ is

$$\vec{m}_i = (\vec{v}\cdot\vec{n})\vec{n} = (\cos\theta_i)\vec{n}.$$

Consider the two parallel horizontal vectors $\vec{w}_i$ and $\vec{w}_t$ in the figure. We have

$$\vec{w}_i = \vec{m}_i - \vec{v}.$$

Since $\vec{v}$ and $\vec{t}$ are each of unit length we have

$$\frac{\eta_t}{\eta_i} = \frac{\sin\theta_i}{\sin\theta_t} = \frac{|\vec{w}_i|/|\vec{v}|}{|\vec{w}_t|/|\vec{t}|} = \frac{|\vec{w}_i|}{|\vec{w}_t|}.$$

Since $\vec{w}_i$ and $\vec{w}_t$ are parallel we have

$$\vec{w}_t = \frac{\eta_i}{\eta_t}\vec{w}_i = \frac{\eta_i}{\eta_t}(\vec{m}_i - \vec{v}).$$

The projection of $\vec{t}$ onto $-\vec{n}$ is $\vec{m}_t = -(\cos\theta_t)\vec{n}$, and hence the desired refraction vector is:

$$\begin{aligned}\vec{t} &= \vec{w}_t + \vec{m}_t \;=\; \frac{\eta_i}{\eta_t}(\vec{m}_i - \vec{v}) - (\cos\theta_t)\vec{n} \;=\; \frac{\eta_i}{\eta_t}((\cos\theta_i)\vec{n} - \vec{v}) - (\cos\theta_t)\vec{n} \\ &= \left(\frac{\eta_i}{\eta_t}\cos\theta_i - \cos\theta_t\right)\vec{n} - \frac{\eta_i}{\eta_t}\vec{v}.\end{aligned}$$

[3] To be completely accurate, the index of refraction depends on the wavelength of light being transmitted. This is what causes white light to be spread into a spectrum as it passes through a prism. But since we do not model light as an entire spectrum, but only through a triple of RGB values (which produce the same color visually, but not the same spectrum physically) we will not get realistic results. For simplicity we assume that all wavelengths have the same index of refraction.

We have already computed $\cos\theta_i = (\vec{v}\cdot\vec{n})$. We can derive $\cos\theta_t$ from Snell's law and basic trigonometry:

$$\begin{aligned}\cos\theta_t &= \sqrt{1-\sin^2\theta_t} = \sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2\sin^2\theta_i} = \sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2(1-\cos^2\theta_i)} \\ &= \sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2(1-(\vec{v}\cdot\vec{n})^2)}.\end{aligned}$$

What if the term in the square root is negative? This is possible if $(\eta_i/\eta_t)\sin\theta_i > 1$. In particular, this can only happen if $\eta_i/\eta_t > 1$, meaning that you are already inside an object with an index of refraction greater than 1. Notice that when this is the case, Snell's law breaks down, since it is impossible to find θ_t whose sine is greater than 1. This is a situation where *total internal reflection* takes place. The light source is not refracted at all, but is reflected within the object. (For example, this is one of the important elements in the physics of rainbows.) When this happens, the refraction reduces to reflection and so we set $\vec{t} = \vec{r}_v$, the view reflection vector.

Illumination Equation Revisited: We can combine the familiar Phong illumination model with the reflection and refraction computed above. We assume that we have shot a ray, and it has hit an object at some point P.

Light sources: Let us assume that we have a collection of light source $L_1, L_2, \ldots$. Each is associated with an RGB vector of intensities (any nonnegative values). Let L_a denote the global RGB intensity of ambient light.

Visibility of Light Sources: The function $\text{Vis}(P, i)$ returns 1 if light source i is visible to point P and 0 otherwise. If there are no transparent objects, then this can be computed by simply shooting a ray from P to the light source and seeing whether it hits any objects.

Material color: We assume that an object's material color is given by C. This is an RGB vector, in which each component is in the interval $[0, 1]$. We assume that the specular color is the same as the light source, and that the object does not emit light. Let ρ_a, ρ_d, and ρ_s denote the ambient, diffuse, and specular coefficients of illumination, respectively. These coefficients are typically in the interval $[0, 1]$. Let α denote the specular shininess coefficient.

Vectors: Let $\vec{n}$, $\vec{h}$, and $\vec{l}$ denote the normalized normal, halfway-vector, and light vectors. See the lecture on the Phong model for how they are computed.

Attenuation: We assume the existence of general quadratic light attenuation, given by the coefficients a, b, and c, as before. Let d_i denote the distance from the contact point P to the ith light source.

Reflection and refraction: Let ρ_r and ρ_t denote the reflective and transmitted (refracted) coefficients of illumination. If $\rho_t \neq 0$ then let η_i and η_t denote the indices of refraction, and let $\vec{r}_v$ and $\vec{t}$ denote the normalized view reflection and transmission vectors.

Let the pair $(P, \vec{v})$ denote a ray originating at point P and heading in direction $\vec{v}$. The complete ray-tracing reflection equation is:

$$\begin{aligned}I &= \rho_a L_a C + \sum_i \text{Vis}(P, i)\frac{L_i}{a + bd_i + cd_i^2}[\rho_d C\max(0, \vec{n}\cdot\vec{l}) + \rho_s\max(0, (\vec{n}\cdot\vec{h}))^\alpha] \\ &\quad + \rho_r\,\text{trace}(P, \vec{r}_v) + \rho_t\,\text{trace}(P, \vec{t}).\end{aligned}$$

Note that if ρ_r or ρ_t are equal to 0 (as is often the case) then the corresponding ray-trace call need not be made. Observe that attenuation and lighting are not applied to results of reflection and refraction. This seems to behave reasonably in most lighting situations, where lights and objects are relatively close to the eye.

Lecture 23: Ray Intersections

(Thursday, Nov 14, 2000)
Read: Chapter 14 in Hill.

Rays and Intersections: The main question that we left unresolved in ray tracing is how to actually perform intersections between rays and objects in our scene. We will discuss this today. First off, how is a ray represented? An obvious method is to represent it by its origin point P and a directional vector $\vec{u}$. Points on the ray can be described *parametrically* using a scalar t:

$$R = \{P + t\vec{u} \mid t > 0\}.$$

Notice that our ray is *open*, in the sense that it does not include its endpoint. This is done because in many instances (e.g., reflection) we are shooting a ray from the surface of some object. We do not want to consider the surface itself as an intersection. In implementing a ray tracer, it is also common to store some additional information as part of a *ray object*. For example, you might want to store the value t_0 at which the ray hits its first object (initially, $t_0 = \infty$) and perhaps a pointer to the object that it hits.

Given an object in the scene, a *ray intersection procedure* determines whether the ray intersects and object, and if so, returns the value $t' > 0$ at which the intersection occurs. (This is a natural use of object-oriented programming, since the intersection procedure can be made a member function of the object.) Otherwise, if t' is smaller than the current t_0 value, then t_0 is set to t'. Otherwise the trimmed ray does not intersect the object.

Ray-Sphere Intersection: Let us consider one of the most popular nontrivial intersection tests for rays, intersection with a sphere in 3-space. We represent a ray R by giving its origin point P and a normalized directional vector $\vec{u}$. Suppose that the sphere is represented by giving its center point C and radius r (a scalar). Our goal is to determine the value of t for which the ray strikes the sphere, or to report that there is no intersection. In our development, we will try to avoid using coordinates, and keep the description as *coordinate-free* as possible.

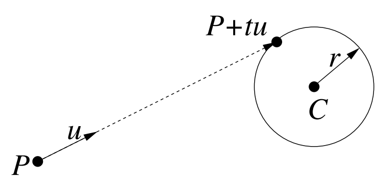


Figure 61: Ray-sphere intersection.

We know that a point Q lies on the sphere if its distance from the center of the sphere is r, that is if $|Q - C| = r$. So the ray intersects at the value of t such that

$$|(P + t\vec{u}) - C| = r.$$

Notice that the quantity inside the $|.|$ above is a vector. Let $\vec{v} = C - P$. This gives us

$$|t\vec{u} - \vec{v}| = r.$$

We know u, v, and r and we want to find t. By the definition of length using dot products we have

$$(t\vec{u} - \vec{v}) \cdot (t\vec{u} - \vec{v}) = r^2.$$

Observe that this equation is scalar valued (not a vector). We use the fact that dot-product is a linear operator, and so we can manipulate this algebraically into:

$$t^2(\vec{u} \cdot \vec{u}) - 2t(\vec{u} \cdot \vec{v}) + (\vec{v} \cdot \vec{v}) - r^2 = 0$$

This is a quadratic equation $at^2 + bt + c = 0$, where

$$\begin{aligned} a &= (\vec{u} \cdot \vec{u}) = 1 \qquad \text{(since } \vec{u} \text{ is normalized)}, \\ b &= -2(\vec{u} \cdot \vec{v}), \\ c &= (\vec{v} \cdot \vec{v}) - r^2 \end{aligned}$$

We can solve this using the quadratic formula

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Numerical Issues: There are some numerical instabilities to beware of. If r is small relative to $|\vec{v}|$ then we may lose the effect of r in the discriminant. It is suggested that rather than computing this in the straightforward way, instead use the following algebraically equivalent manner. The discriminant is $D = b^2 - 4ac$. First observe that we can express the determinant as

$$D = 4(r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2).$$

(We will leave it as an exercise to verify this.) If D is negative then there is no solution, implying that the ray misses the sphere. If it is positive then there are two real roots:

$$t = \frac{-b \pm \sqrt{D}}{2a} = (\vec{u} \cdot \vec{v}) \pm \sqrt{r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2}.$$

Which root should we take? Recall that $t > 0$ and increases as we move along the ray. Therefore, we want the smaller positive root. If neither root is positive then there is no intersection. Consider

$$t^- = (\vec{u} \cdot \vec{v}) - \sqrt{r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2} \qquad t^+ = (\vec{u} \cdot \vec{v}) + \sqrt{r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2}.$$

If $t^- > 0$ then take it, otherwise if $t^+ > 0$ then take it. Otherwise, there is no intersection (since it intersects the negative extension of the ray).

Note that it is a not a good idea to compare floating point numbers against zero, since floating point errors are always possible. A good rule of thumb is to do all of these 3-d computations using doubles (not floats) and perform comparisons against some small value instead, e.g. `SMALL = 1E-3`. The proper choice of this parameter is a bit of "hokus-pokus". It is usually adjusted until the final image looks okay.

More Care with Roundoff Errors: There is still a possibility of roundoff error if we simply use the formulas given above for solving the quadratic equation. The problem is that when two very similar numbers are subtracted we may lose many significant digits. Recall the basic equation $at^2 + bt + c = 0$. Rather than applying the quadratic formula directly, numerical analysts recommend that you first compute the root with the larger absolute value, and then use the identity $t^- t^+ = c/a$, to extract the other root. In particular, if $b \le 0$ then use:

$$\begin{aligned} t^+ &= \frac{-b+\sqrt{D}}{2a} = (\vec{u}\cdot\vec{v}) + \sqrt{r^2 - |\vec{v} - (\vec{u}\cdot\vec{v})\vec{u}|^2}, \\ t^- &= \frac{c}{at^+} \qquad \text{(only if } t^+ > 0\text{)}. \end{aligned}$$

Otherwise, if $b > 0$, then we use

$$\begin{aligned} t^- &= \frac{-b-\sqrt{D}}{2a} = (\vec{u}\cdot\vec{v}) - \sqrt{r^2 - |\vec{v} - (\vec{u}\cdot\vec{v})\vec{u}|^2}, \\ t^+ &= \frac{c}{at^-} \qquad \text{(only if } t^- < 0\text{)}. \end{aligned}$$

As before, select the smaller positive root as the solution. In typical applications of ray tracing, this extra care does not seem to be necessary, but it is good thing to keep in mind if you really want to write a robust ray tracer.

Normal Vector: In addition to computing the intersection of the ray with the object, it is also necessary to compute the normal vector at the point of intersection. In the case of the sphere, note that the normal vector is directed from the center of the sphere to point of contact. Thus, if t is the parameter value at the point of contact, the normal vector is just

$$\vec{n} = normalize(P + t\vec{u} - C).$$

Ray-Triangle Intersection: Suppose that we wish to intersect a ray with a polyhedral object. There are two standard approaches to this problem. The first works only for convex polyhedra. In this method, we represent a polyhedron as the intersection of a set of halfspaces. In this case, we can easily modify the 2-d line segment clipping algorithm presented in Lecture 9 to perform clipping against these halfspaces. We will leave this as an exercise. The other method involves representing the polyhedron by a set of polygonal faces, and intersecting the ray with these polygons. We will consider this approach here.

There are two tasks which are needed for ray-polygon intersection tests. The first is to extract the equation of the (infinite) plane that supports the polygon, and determine where the ray intersects this plane. The second step is to determine whether the intersection occurs within the bounds of the actual polygon. This can be done in a 2-step process. We will consider a slightly different method, which does this all in one step.

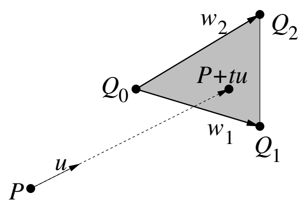


Figure 62: Ray-triangle intersection.

Let us consider the simplest case of a triangle. Let Q_0, Q_1, and Q_2 be the vertices of the triangle in 3-space. Any point Q' that lies on this triangle can be described by a convex combination of these points

$$Q' = \alpha_0 Q_0 + \alpha_1 Q_1 + \alpha_2 Q_2,$$

where $\alpha_i \geq 0$ and $\sum_i \alpha_i = 1$. From the fact that the α_i's sum to 1, we can set $\alpha_0 = 1 - \alpha_1 - \alpha_2$ and do a little algebra to get

$$Q' = Q_0 + \alpha_1(Q_1 - Q_0) + \alpha_2(Q_2 - Q_0),$$

where $\alpha_i \geq 0$ and $\alpha_1 + \alpha_2 \leq 1$. Let

$$\vec{w}_1 = Q_1 - Q_0, \qquad \vec{w}_2 = Q_2 - Q_0,$$

giving us the following

$$Q' = Q_0 + \alpha_1\vec{w}_1 + \alpha_2\vec{w}_2.$$

Recall that our ray is given by $P + t\vec{u}$ for $t > 0$. We want to know whether there is a point Q' of the above form that lies on this ray. To do this, we just substitute the parametric ray value for Q' yielding

$$\begin{aligned} P + t\vec{u} &= Q_0 + \alpha_1\vec{w}_1 + \alpha_2\vec{w}_2 \\ P - Q_0 &= -t\vec{u} + \alpha_1\vec{w}_1 + \alpha_2\vec{w}_2. \end{aligned}$$

Let $\vec{w}_P = P - Q_0$. This is an equation, where t, α_1 and α_2 are unknown (scalar) values, and the other values are all 3-element vectors. Hence this is a system of three equations with three unknowns. We can write this as

$$\left(-\vec{u} \;\middle|\; \vec{w}_1 \;\middle|\; \vec{w}_2 \right) \begin{pmatrix} t \\ \alpha_1 \\ \alpha_2 \end{pmatrix} = \left(\vec{w}_P \right).$$

To determine t, α_1 and α_2, we need only solve this system of equations. Let M denote the 3×3 matrix whose columns are $-\vec{u}$, $\vec{w}_1$ and $\vec{w}_2$. We can do this by computing the inverse matrix M^{-1} and then we have

$$\begin{pmatrix} t \\ \alpha_1 \\ \alpha_2 \end{pmatrix} = M^{-1} \left(\vec{w}_P \right).$$

There are a number of things that can happen at this point. First, it may be that the matrix is singular (i.e., its columns are not linearly independent) and no inverse exists. This happens if $\vec{t}$ is parallel to the plane containing the triangle. In this case we will report that there is no intersection. Otherwise, we check the values of α_1 and α_2. If either is negative then there is no intersection and if $\alpha_1 + \alpha_2 > 1$ then there is no intersection.

Normal Vector: In addition to computing the intersection of the ray with the object, it is also desirable to compute the normal vector at the point of intersection. In the case of the triangle, this can be done by computing the cross product

$$\vec{n} = \mathit{normalize}((Q_1 - Q_0) \times (Q_2 - Q_0)) = \mathit{normalize}(\vec{w}_1 \times \vec{w}_2).$$

But which direction should we take for the normal, $\vec{n}$ or $-\vec{n}$? This depends on which side of the triangle the ray arrives. The normal should be directed opposite to the directional ray of the vector. Thus, if $\vec{n} \cdot \vec{u} > 0$, then negate $\vec{n}$.

Lecture 24: More Issues in Ray Tracing

(Tuesday, Nov 21, 2000)
Read: Chapter 14 in Hill. (The material on Bezier surfaces is not covered in our text.)

Issues in Ray Tracing: Today we consider a number of miscellaneous issues in the ray tracing process.

Ray and Bezier Surface Intersection: Let us consider a more complex but more realistic ray intersection problem, namely that of intersecting a ray with a Bezier surface. One possible approach would be to derive an implicit representation of infinite algebraic surface on which the Bezier patch resides, and then determine whether the ray hits the portion of this infinite surface corresponding to the patch. This leads to a very complex algebraic task.

Instead, our approach is based on using circle-ray and triangle-ray intersection tests (which we have already discussed) and the deCasteljau procedure. To do so, we will describe the process for a Bezier curve, and then consider the generalization to surfaces. First, we could use the convex hull property to provide a simple heuristic test for whether the ray misses the convex hull of the control points. If so, it misses the curve. If not, then we'll see later what to do. This would require computing the convex hull and an intersection test for convex hulls. We will apply a simpler test, by finding an enclosing circle for the curve. We do this by computing an approximate center point C for the curve. This can be done, for example, as the centroid of the control points or the midpoint between the first and last control points. Given the point C, we then compute the distance from each control point and C. Let $d_{\max}$ denote the largest such distance. The circle with center C and radius $d_{\max}$ encloses all the control points, and hence it encloses the convex hull of the control points, and hence it encloses the entire curve. We test the ray for intersection with the circle. If it does not hit the circle, then we may safely say that it does not hit the Bezier curve.

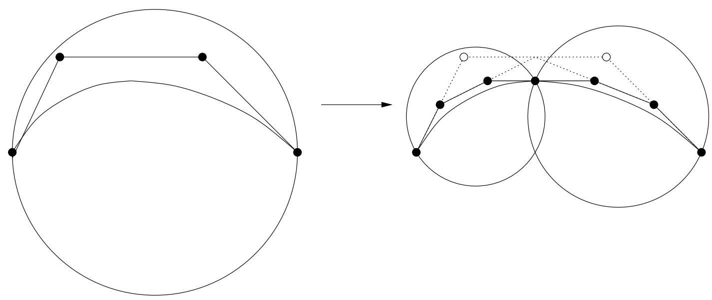


Figure 63: Subdivision.

If the ray does hit the circle, it still may miss the curve. Here we apply the deCasteljau algorithm to subdivide the Bezier curve into two Bezier subcurves. (We will leave the generalization of the curve subdivision to surfaces as an exercise.) Then we apply the ray intersection algorithm recursively to the two subcurves. If both return misses, then we miss. If either or both returns a hit, then we take the closer of the two hits. We need some way to keep this recursive procedure from looping infinitely. To do so, we need some sort of stopping criterion. Here are a few possibilities:

Fixed level decomposition: Fix an integer k, and decompose the curve to a depth of k levels (resulting in 2^k) subcurves in all. This is certainly simple, but not a very efficient approach. It does not consider the shape of the curve or its distance from the viewer.

Decompose until flat: For each subcurve, we can compute some function that measures how *flat*, that is, close to linear, the curve is. For example, this might be done by considering the ratio of the length of the line segment between the first and last control points and distance of the furthest control point from this line. At this point we reduce the ray intersection to line segment intersection problem.

This can be generalized to surfaces. When a surface patch is sufficiently flat, we can approximate by two triangles. Although this works reasonably well for curves, not that we may have problems with surfaces because two adjacent patches with different degrees of curvature may be subdivided to a different levels. As a result, we may produce a small "crack" between these two surfaces, because they no longer share the same control points along their common boundary. These cracks may be small, but they can be quite noticeable visibly.

Decompose to pixel width: We continue to subdivide the curve until each subcurve, when projected back to the viewing window, overlaps a region of less than one pixel. Clearly it is unnecessary to continue to subdivide such a curve. This solves the crack problem (since cracks are smaller than a pixel) but may produce an unnecessarily high subdivision for nearly flat curves. Also notice that this the notion of back projection is easy to implement for rays emanating from the eye, but this is much harder to determine for reflection or refracted rays.

The most reasonable approach is probably a hybrid between the latter two.

Procedural Textures: We can apply texture mapping in ray tracing just as we did in OpenGL's rendering model. Given the intersection of the ray with an object, we need to map this intersection point to some sort of 2-dimensional parameterization (u, v). From this parameterization, we can then apply the inverse wrapping function to map this point into texture coordinates (u, v).

In ray tracing there is another type of texture, which is quite easy to implement. The idea is to create a function $f(x, y, z)$ which maps a point in 3-space to a color. This is called a *procedural texturing*.

We usually think of textures as 2-dimensional "wallpapers" that are wrapped around objects. The notion here is different. We imagine that the texture covers all of 3-space and that the object is cut out of this infinite texture. This is actually quite realistic in some cases. For example, a wood grain texture arises from the cylindrical patterns of light and dark wood that results from the trees varying rates of growth between the seasons. When you cut a board of wood, the pattern on the surface is the intersection of a single plane and this cylindrical 3-dimensional texture. Here are some examples.

Checker: Let C_0 and C_1 be two RGB colors. Imagine that we tile all of three dimensional space with a collection of unit cubes each of side length s and of alternating colors C_0 and C_1. This might be easiest to see in the 1-dimensional case first. Given an x-coordinate, we divide it by s and take its floor. If the resulting number is even then we assign the color C_0 and if odd we assign C_1.

```
checker(x) = (floor(x/s) % 2) == 0 : C_0 ? C_1);
```

Note that `floor()`, as used here, is not a built-in C++ function. (There is one but it works for floats, not integers). Note that this is different from casting function `int()`. Whereas `floor()` always rounds down to the next smaller value, `int()` rounds to the next smaller absolute value.

To generalize this to three space, we simply apply this idea separately to each coordinate and sum the results.

```
checker(P) = ((floor(P.x/s)+floor(P.y/s)+floor(P.z/s))%2) == 0 : C_0 ? C_1);
```

Note that if we intersect an axis-orthogonal plane with the resulting 3-dimensional checker pattern, then the result will be a 3-dimensional checker. If we intersect it with a non-axis aligned object (like a sphere) then the result takes on a decidedly different appearance.

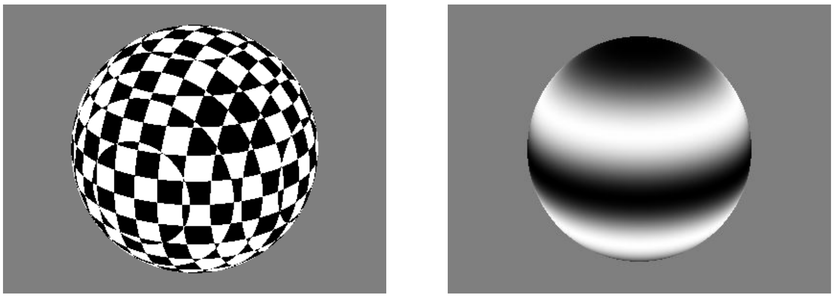


Figure 64: 3-d checkerboard and gradient textures.

Gradient: A gradient is a texture that alternates smoothly between two colors C_0 and C_1, relative to some directional vector $\vec{v}$ and an origin point Q. The idea is that for each point P, consider the relative length of the projection of $P - Q$ onto $\vec{v}$

$$\beta = \frac{((P - Q) \cdot \vec{v})}{(\vec{v} \cdot \vec{v})}.$$

We use the value of β to blend smoothly between C_0 and C_1. In particular, the color of point P is the convex combination,

$$\text{gradient}(P) = (1 - \alpha)C_0 + \alpha C_1,$$

where $\alpha = (1 - \cos(\beta\pi))/2$. Observe that as the projection of P onto $\vec{v}$ moves along the length of the vector, β increases from 0 to 1 to 2, and the cosine term varies from 1 to -1 to 1. The value of α varies from 0 to 1 to 0, and hence the final color varies from C_0 to C_1 and back to C_0. Thus we get a smooth blend of these two colors. The figure above shows an example of applying a gradient pattern to a sphere, where the center point Q is the center of the sphere, $\vec{v}$ is vertical, and the length of $\vec{v}$ is half the radius of the sphere.

When combined with a random number generator, procedural textures can produce quite complex natural textures, such as wood grains and marble textures. See the book for further examples.

Ray Generation: Let us return to the basic issue of how to generate rays. Let us assume that we are given essentially the same information that we use in `gluLookAt()` and `gluPerspective()`. In particular, let E denote the eye point, C denote the center point, and $\vec{up}$ denote the up vector for `gluLookAt()`. Let θ denote the y-field of view. Let $nRows$ and $nCols$ denote the number of rows and columns in the final image, and let $\alpha = nCols/nRows$ denote the window's aspect ratio. In gluPerspective we also need to give the distance to the near and far clipping planes. But for ray tracing these are not really needed, so to make our life simple, let us assume that the window is 1 unit in front of the eye. (The distance is not important, since the aspect ratio and the field-of-view really determine everything up to a scale factor.)

The height and width of view window relative to its center point are

$$h = \tan(\theta/2) \qquad w = h\alpha.$$

So, the window extends from $-h$ to $+h$ in height and $-w$ to $+w$ in width. Now, we proceed to compute the viewing coordinate frame as we did in Lecture 10. The origin is E, the location of the eye.

$$\begin{aligned}
\vec{e}_z &= -normalize(C - E), \\
\vec{e}_x &= normalize(\vec{up} \times \vec{e}_z), \\
\vec{e}_y &= \vec{e}_z \times \vec{e}_x.
\end{aligned}$$

We will follow the (somewhat strange) convention used in .bmp files and assume that rows are indexed from bottom to top and columns are indexed from left to right. Every point on the view window has $\vec{e}_z$ coordinate of -1. Now, suppose that we want to shoot a ray for row r and column c, where $0 \leq r < nRows$ and $0 \leq c < nCols$. Observe that $r/nRows$ is in the range from 0 to 1. Multiplying by $2h$ and subtracting h maps us linearly to the interval $-h$ to h, as desired. Applying this to row and column indices we have

$$\begin{aligned}
u_r &= 2h\frac{r}{nRows} - h, \\
u_c &= 2w\frac{c}{nCols} - w.
\end{aligned}$$

The location of the corresponding point on the viewing window is

$$P(r, c) = E + u_c\vec{e}_x + u_r\vec{e}_y - \vec{e}_z.$$

Thus, the desired ray $R(r, c)$ has the origin E and the directional vector

$$\vec{v}(r, c) = normalize(P(r, c) - E).$$

Lecture 25: Scan Conversion

(Tuesday, Nov 28, 2000)
Read: Chapter 10 in Hill.

Scan Conversion: We turn now to a number of miscellaneous issues involved in the implementation of computer graphics systems. In our top-down approach we have concentrated so far on the high-level view of computer graphics. In the next few lectures we will consider how these things are implemented. In particular, we consider the question of how to map 2-dimensional geometric objects (as might result from projection) to a set of pixels to be colored. This process is called *scan conversion* or *rasterization*. We begin by discussing the simplest of all rasterization problems, drawing a single line segment.

Let us think of our raster display as an integer grid, in which each pixel is a circle of radius 1/2 centered at each point of the grid. We wish to illuminate a set of pixels that lie on or close to the line. In particular, we wish to draw a line segment from $q = (q_x, q_y)$ to $r = (r_x, r_y)$, where the coordinates are integer grid points (typically by a process of rounding). Let us assume further that the slope of the line is between 0 and 1, and that $q_x < r_x$. This may seem very restrictive, but it is not difficult to map any line drawing problem to satisfy these conditions. For example, if the absolute value of the slope is greater than 1, then we interchange the roles of x and y, thus resulting in a line with a reciprocal slope. If the slope is negative, the algorithm is very easy to modify (by decrementing rather than incrementing). Finally, by swapping the endpoints we can always draw from left to right.

Bresenham's Algorithm: We will discuss an algorithm, which is called *Bresenham's algorithm*. It is one of the oldest algorithms known in the field of computer graphics. It is also an excellent example of how one can squeeze every but of efficiency out an algorithm. We begin by considering an *implicit* representation of the line equation. (This is used only for deriving the algorithm, and is not computed explicitly by the algorithm.)

$$f(x, y) = ax + by + c = 0.$$

If we let $d_x = r_x - q_x$, $d_y = r_y - q_y$, it is easy to see (by substitution) that $a = d_y$, $b = -d_x$, and $c = -(q_x r_y - r_x q_y)$. Observe that all of these coefficients are all integers. Also observe that $f(x, y) > 0$ for points that lie below the line and $f(x, y) < 0$ for points above the line. For reasons that will become apparent later, let us use an equivalent representation by multiplying by 2

$$f(x, y) = 2ax + 2by + 2c = 0.$$

Here is the intuition behind Bresenham's algorithm. For each integer x value, we wish to determine which integer y value is closest to the line. Suppose that we have just finished drawing a pixel (p_x, p_y) and we are interested in figuring out which pixel to draw next. Since the slope is between 0 and 1, it follows that the next pixel to be drawn will either be the pixel to our East ($E = (p_x + 1, p_y)$) or the pixel to our NorthEast ($NE = (p_x + 1, p_y + 1)$). Let q denote the exact y-value (a real number) of the line at $x = p_x + 1$. Let $m = p_y + 1/2$ denote the y-value midway between E and NE. If $q < m$ then we want to select E next, and otherwise we want to select NE. IF $q = m$ then we can pick either, say E. See the figure.

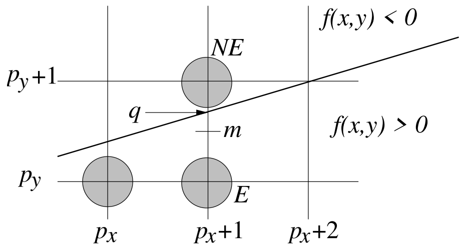


Figure 65: Bresenham's midpoint algorithm.

To determine which one to pick, we have a *decision variable* D which will be the value of f at the midpoint. Thus

$$\begin{aligned} D &= f(p_x + 1, p_y + (1/2)) \\ &= 2a(p_x + 1) + 2b\left(p_y + \frac{1}{2}\right) + 2c \\ &= 2ap_x + 2bp_y + (2a + b + 2c). \end{aligned}$$

If $D > 0$ then m is below the line, and so the NE pixel is closer to the line. On the other hand, if $D \leq 0$ then m is above the line, so the E pixel is closer to the line. (Note: We can see now why we doubled $f(x, y)$. This makes D an integer quantity.)

The good news is that D is an integer quantity. The bad news is that it takes at least at least two multiplications and two additions to compute D (even assuming that we precompute the part of the expression that does not change). One of the clever tricks behind Bresenham's algorithm is to compute D *incrementally.* Suppose we know the current D value, and we want to determine its next value. The next D value depends on the action we take at this stage.

We go to E next: Then the next midpoint will have coordinates $(p_x + 2, p_y + (1/2))$ and hence the new D value will be

$$\begin{aligned} D_{new} &= f(p_x + 2, p_y + (1/2)) \\ &= 2a(p_x + 2) + 2b\left(p_y + \frac{1}{2}\right) + 2c \\ &= 2ap_x + 2bp_y + (4a + b + 2c) \\ &= 2ap_x + 2bp_y + (2a + b + 2c) + 2a \\ &= D + 2a = D + 2d_y. \end{aligned}$$

Thus, the new value of D will just be the current value plus $2d_y$.

We go to NE next: Then the next midpoint will have coordinates $(p_x + 2, p_y + 1 + (1/2))$ and hence the new D value will be

$$\begin{aligned} D_{new} &= f(p_x + 2, p_y + 1 + (1/2)) \\ &= 2a(p_x + 2) + 2b\left(p_y + \frac{3}{2}\right) + 2c \\ &= 2ap_x + 2bp_y + (4a + 3b + 2c) \\ &= 2ap_x + 2bp_y + (2a + b + 2c) + (2a + 2b) \\ &= D + 2(a + b) = D + 2(d_y - d_x). \end{aligned}$$

Thus the new value of D will just be the current value plus $2(d_y - d_x)$.

Note that in either case we need perform only one addition (assuming we precompute the values $2d_y$ and $2(d_y - d_x)$. So the inner loop of the algorithm is quite efficient.

The only thing that remains is to compute the initial value of D. Since we start at (q_x, q_y) the initial midpoint is at $(q_x + 1, q_y + 1/2)$ so the initial value of D is

$$\begin{aligned} D_{init} &= f(q_x + 1, q_y + 1/2) \\ &= 2a(q_x + 1) + 2b\left(q_y + \frac{1}{2}\right) + 2c \\ &= (2aq_x + 2bq_y + 2c) + (2a + b) \\ &= 0 + 2a + b \qquad \text{Since } (q_x, q_y) \text{ is on line} \\ &= 2d_y - d_x. \end{aligned}$$

We can now give the complete algorithm. Recall our assumptions that $q_x < r_x$ and the slope lies between 0 and 1. Notice that the quantities $2d_y$ and $2(d_y - d_x)$ appearing in the loop can be precomputed, so each step involves only a comparison and a couple of additions of integer quantities.

Bresenham's midpoint algorithm

```
void bresenham(IntPoint q, IntPoint r) {
    int dx, dy, D, x, y;
    dx = r.x - q.x;                // line width and height
    dy = r.y - q.y;
    D = 2*dy - dx;                 // initial decision value
    y = q.y;                       // start at (q.x,q.y)
    for (x = q.x; x <= r.x; x++) {
        writePixel(x, y);
```

```
            if (D <= 0) D += 2*dy;      // below midpoint - go to E
            else {                      // above midpoint - go to NE
                D += 2*(dy - dx); y++;
            }
        }
    }
```

Bresenham's algorithm can be modified for drawing other sorts of curves. For example, there is a Bresenham-like algorithm for drawing circular arcs. The generalization of Bresenham's algorithm is called the *midpoint algorithm*, because of its use of the midpoint between two pixels as the basic discriminator.

Filling Regions: In most instances we do not want to draw just a single curve, and instead want to fill a region. There are two common methods of defining the region to be filled. One is polygon-based, in which the vertices of a polygon are given. We will discuss this later. The other is *pixel based*. In this case, a boundary region is defined by a set of pixels, and the task is to fill everything inside the region. We will discuss this latter type of filling for now, because it brings up some interesting issues.

The intuitive idea that we have is that we would like to think of a set of pixels as defining the *boundary* of some region, just as a closed curve does in the plane. Such a set of pixels should be connected, and like a curve, they should split the infinite grid into two parts, an *interior* and an *exterior*. Define the *4-neighbors* of any pixel to be the pixels immediately to the north, south, east, and west of this pixel. Define the *8-neighbors* to be the union of the 4-neighbors and the 4 closest diagonal pixels. There are two natural ways to define the notion of being connected, depending on which notion of neighbors is used.

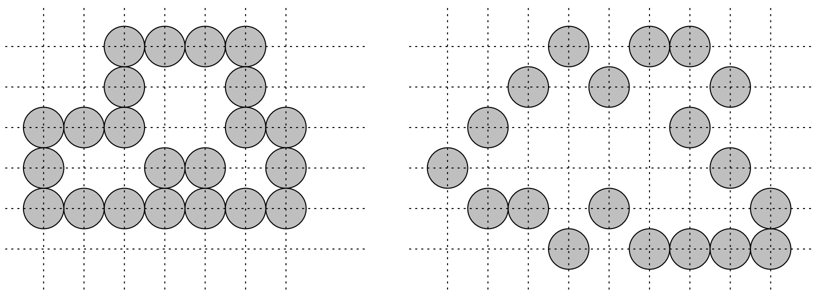


Figure 66: 4-connected (left) and 8-connected (right) sets of pixels.

4-connected: A set is 4-connected if for any two pixels in the set, there is path from one to the other, lying entirely in the set and moving from one pixel to one of its 4-neighbors.

8-connected: A set is 8-connected if for any two pixels in the set, there is path from one to the other, lying entirely in the set and moving from one pixel to one of its 8-neighbors.

Observe that a 4-connected set is 8-connected, but not vice versa. Recall from the Jordan curve theorem that a closed curve in the plane subdivides the plane into two connected regions, and interior and an exterior. We have not defined what we mean by a closed curve in this context, but even without this there are some problems. Observe that if a boundary curve is 8-connected, then it is generally not true that it separates the infinite grid into two 8-connected regions, since (as can be seen in the figure) both interior and exterior can be joined to each other by a 8-connected path. There is an interesting way to fix this problem. In particular, if we require that the boundary curve be 8-connected, then we require that the region it define be 4-connected. Similarly, if we require that the boundary be 4-connected, it is common to assume that the region it defines be 8-connected.

Recursive Flood Filling: Irrespective of how we define connectivity, the algorithmic question we want to consider is how to fill a region. Suppose that we are given a starting pixel $p = (p_x, p_y)$. We wish to visit all pixels in the same *connected component* (using say, 4-connectivity), and assign them all the same color. We will assume that all of these pixels initially share some common *background color*, and we will give them a new *region color*. The idea is to walk around, as whenever we see a 4-neighbor with the background color we assign it color the region color. The problem is that we may go down dead-ends and may need to backtrack. To handle the backtracking we can keep a stack of unfinished pixels. One way to implement this stack is to use recursion. The method is called *flood filling*. The resulting procedure is simple to write down, but it is not necessarily the most efficient way to solve the problem. See the book for further consideration of this problem.

Recursive Flood-Fill Algorithm (4-connected)

```
void floodFill(intPoint p) {
    if (getPixel(p.x, p.y) == backgroundColor) {
        setPixel(p.x, p.y, regionColor);
        floodFill(p.x - 1, p.y);          // apply to 4-neighbors
        floodFill(p.x + 1, p.y);
        floodFill(p.x, p.y - 1);
        floodFill(p.x, p.y + 1);
    }
}
```

Lecture 26: Scan Conversion for Polygons

(Thursday, Nov 30, 2000)
Read: Chapter 10 in Hill.

Filling Polygons: One of the fundamental low-level operations that is critical to the efficiency of graphics systems is that of rasterizing polygons. We will consider the case where the polygon is filled with a single solid color, but polygon filling typically involves producing additional shading or mapping complex textures as well. However, the basic task of determining which pixels are part of the filled region is fundamental to these other tasks.

Tie breaking rules: The task of polygon filling is one that is really much trickier than what one might imagine at first. It is very important, for example, that patches that share a common edge should be drawn in such a way that there are no "gaps" between the adjacent regions. Overlap (writing the same pixel twice) also represents a danger, since there are some pixel writing modes (e.g. using an accumulation buffer) where writing a pixel twice results in a different color.

Our text suggests the following criteria for determining exactly which pixels to draw as part of the polygon. Briefly, the rule can be called *ILB* for *interior-left-bottom*, since this describes which pixels the algorithm considers to be "part" of the filled polygon. A pixel is drawn if it is either in the interior of the polygon, along a left-side edge, or along a bottom-side edge. Unfortunately, it is rather hard to describe the exact conditions under which the rule applies. Instead, I prefer to use a slightly different rule. It is called *LRsU* for "Look Right and slightly Up".

Here is the idea. Suppose you have a pixel and you want to determine whether it is part of the filled polygon or not. Consider a point that infinitessimally to the right of the point, but not directly right since there might be a horizontal edge there. Instead look to the right and very slightly up. If this point is in the interior of the polygon then consider the pixel to be inside and hence filled, and if this point is exterior to the polygon, consider it to not be filled. An example of an application of this rule is shown in the figure below.

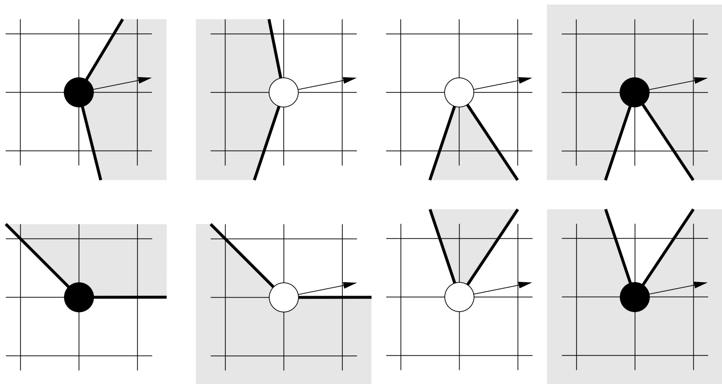


Figure 67: The LRsU Rule.

You might ask how this notion of an "infinitessimal" is computed. It is important to note that we never actually compute infinitessimal values (say, using floating point). Instead, the algorithm simply makes its discrete decisions *as if* an infinitessimal perturbation was applied.

Scan Line Algorithm: The standard algorithm for filling polygons is called the *scan-line* algorithm. It is one example of general paradigm of geometric algorithms, which are usually called *sweep-line algorithms* in computational geometry.

It works by imagining a scanning line that scans row-by-row from the bottom of the polygon to the top, and working from left to right along each row. When the scan line first comes to an edge of the polygon, it starts drawing pixels. When it next comes to an edge it stops. It simply repeats this process until the entire polygon has been scanned out.

However, this simple idea hides some nasty special cases that need to be handled carefully. For example, what happens if the scan line hits vertex? Our text suggests a fairly elegant way to handle the various special cases, by choosing to consider or ignore certain edges and vertices.

Horizontal edges: Are ignored entirely. We will see that this has a magical way of working out.

Edge coincides with pixel: If the interior of a nonhorizontal edge coincides exactly with a pixel, then we invert the drawing mode at this pixel. That is, if we were not drawing then we start with this pixel, and if we were drawing then we stop drawing at this pixel (i.e. it is not drawn).

Vertex coincides with pixel: We treat each nonhorizontal edge as though it is *open* on top and *closed* on the bottom. That is, the topmost pixel does not belong to the edge, but the bottommost pixel does. Here are various cases that we might encounter:

One edge up and one edge down: In this case because this vertex belongs to exactly one of the edges, the drawing mode is inverted at this pixel.

Both edges down: In this case because this pixel does not belong to either edge, the drawing mode is unaffected.

Both edges up: In this case because this pixel belongs to both edges, the drawing mode is inverted twice, and as a result it does not change.

One edge horizontal: Because we have chosen to ignore horizontal edges entirely, the mode will change only if the other vertex is a bottom vertex.

The claim is that these rules are exactly what you need to satisfy the LRsU rules for drawing. In particular, notice that we never introduce any infinitessimal values. An example is shown in the figure below.

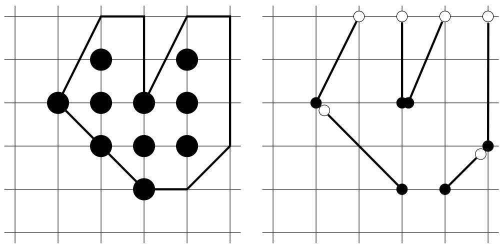


Figure 68: The Scan-Line Algorithm.

Implementation: There are a couple aspects to the efficient implementation of the scan-line algorithm.

Which edge do we hit next? We need a data structure which tells us which line the scan line hits next. This will be done by a structure called the *active edge table* (or *AET*).

When should edges be added to the AET? We need to know at the start of each new scan-line which edges have ceased to be active, and which new edges have started to be active. This we do with a structure called the *edge table*.

At what pixel does the scan line hit an edge? For each edge in the AET, we keep track of its x-coordinate along the current scan line. However, this quantity is a floating point number in general. Can we update this number using strictly integer arithmetic?

The book describes the last component of the algorithm in some detail (especially how to avoid floating point arithmetic) but we will not worry about this for now. The main thing that we need to know for each edge is what is its current x-coordinate along the scan line. The book observes that as we increment the y-coordinate in the scan line by 1, the corresponding x-coordinate of each line changes by $1/m$, where m is the slope of the line. To make it easy to update the x-coordinates of each edge, we will store both the current x coordinate, and the value $1/m$ for each edge that is currently active. Let us concentrate on the first two issues, since these issues are common to any sort of scan-line (or sweep-line) algorithm.

Edge table: First we consider the *edge table*. (In general sweep-line algorithms we need a *priority queue* which tells us what events are coming up.) Because the main events for the scan line algorithm involve the introduction of new edges, the edge table contains a list edges sorted according to their bottom y-coordinates. (Recall that horizontal edges are simply ignored at not stored in the table.) Each entry of the edge table (and the active edge table) has the following structure:

```
struct E_Node {
    int     y_max;          // y-coordinate of edge's top vertex
    double  x_curr;         // x-coordinate of edge's bottom vertex
```

```
        double  recip_slope;    // reciprocal of edges slope (dx/dy)
        E_Node  *next;          // next entry in edge table
    };
```

The `x_curr` entry initially contains the x-coordinate of the bottom vertex of the edge. We will see later that its value is changed when the edge becomes active. The entries in the table are bucket sorted by `ymin`, the y-coordinate of the edge's bottom vertex. We have an array `ET` that contains one entry for each scan line. `ET[i]` points to a linked list of `E_Nodes` for which `ymin = i`.

Initially all nonhorizontal edges are inserted into the edge table. As the scan line visits row i, the contents of `ET[i]` are inserted in the active edge table.

Active Edge Table: The active edge table (AET) is a linked list that contains the edges that are currently intersected by the scan line. They are stored in ascending order according to the x-coordinate of the intersection of the edge with the current scan line. (Notice that if we assume that edges of the polygon are not self-intersecting, we do not need to sort this list except when new edges are inserted.)

Each entry has exactly the same `E_Node` structure as the edge table. Indeed, when an edge becomes active, the entry is simply copied from the ET and inserted in the appropriate place in the AET. When the scan-line reaches the top of an edge, it is deleted from the AET. Otherwise, the scan-line algorithm simply moves from one AET entry to the next, inverting the drawing mode between each consecutive pair. Note the the AET should always have an even number of entries.

Scan-line algorithm: Here is the scan-line algorithm.

(1) Bucket sort all nonhorizontal edges into the edge table ET according to their bottom y-coordinate.

(2) Set y to the smallest y-coordinate in the edge table.

(3) Initialize the AET to empty.

(4) Repeat until the AET is empty:

  (a) Move from ET bucket y to AET those edges for which `y_min = y`, keeping the AET sorted by x-coordinate.

  (b) Remove from the AET those entries for which `y_max = y`.

  (c) Visit the AET in pairs, filling in the pixels in between.

  (d) Increment y by 1.

  (e) For each edge in the AET, update `x_curr` by the reciprocal slope.

Lecture 27: Hidden Surface Removal

(Tuesday, Dec 5, 2000)
Read: Chapter 13 in Hill.

Hidden-Surface Removal: We continue our discussion of implementation issues in computer graphics by talking about hidden surface removal. We are given a collection of objects (represented, say, by a set of polygons) in 3-space, and a viewing situation, and we want to render only the visible surfaces. Each polygon face is assumed to be flat (although extensions to hidden-surface elimination of curved surfaces is an important problem) and opaque. We may assume that each polygon is represented by a cyclic listing of the (x, y, z) coordinates of their vertices, so that from the "front" the vertices are enumerated in counterclockwise order.

One question that arises right away is what do we want as the output of a hidden-surface procedure. There are generally two options.

Object precision: The algorithm computes its results to machine precision (the precision used to represent object coordinates). The resulting image may be enlarged many times without significant loss of accuracy. The output is a set of visible object faces, and the portions of faces that are only partially visible.

Image precision: The algorithm computes its results to the precision of a pixel of the image. Thus, once the image is generated, any attempt to enlarge some portion of the image will result in reduced resolution.

Although image precision approaches have the obvious drawback that they cannot be enlarged without loss of resolution, the fastest and simplest algorithms usually operate by this approach.

The hidden-surface elimination problem for object precision is interesting from the perspective of algorithm design, because it is an example of a problem that is rather hard to solve in the worst-case, and yet there exists a number of fast algorithms that work well in practice. As an example of this, consider a patch-work of n thin horizontal strips in front of n thin vertical strips. If we wanted to output the set of visible polygons, observe that the complexity of the projected image with hidden-surfaces removed is $O(n^2)$. Hence, it is impossible to beat $O(n^2)$ in the worst case. However, almost no one in graphics uses worst-case complexity as a measure of how good an algorithm is, because these worst-case scenarios do not happen often in practice. (By the way there is an "optimal" $O(n^2)$ algorithm, which is never used in practice.)

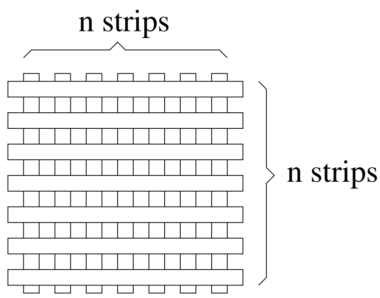


Figure 69: Worst-case example for hidden-surface elimination.

Back-face culling: This is not a general hidden surface removal algorithm, but rather just a heuristic for eliminating obviously invisible faces from consideration. It can eliminate roughly half of the faces from consideration. Assuming that the viewer is never inside any of the objects of the scene, then the back sides of objects are never visible to the viewer, and hence they can be eliminated from consideration. For each polygonal face, we assume an outward pointing normal can be computed (e.g. by the area method described earlier this semester). If this normal is directed *away* from the viewpoint, that is, if its dot product with a vector directed towards the viewer is negative, then the face can be immediately discarded from consideration. On average this quick test can eliminate about one half of the faces from further consideration.

Depth-Sort Algorithm: A fairly simple hidden-surface algorithm is based on the principle of painting objects from back to front, so that more distant polygons are overwritten by closer polygons. This is called the *depth-sort algorithm.* This suggests the following algorithm:

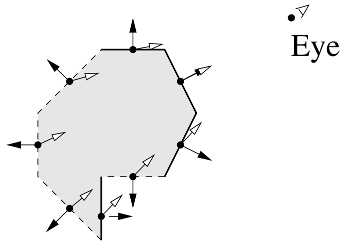


Figure 70: Back-face culling.

sort all the polygons according to increasing distance from the viewpoint, and then scan convert them in reverse order (back to front). This is sometimes called the *painter's algorithm* because it mimics the way that oil painters usually work (painting the background before the foreground). The painting process involves setting pixels, so the algorithm is an image precision algorithm.

There is a very quick-and-dirty technique for hidden-surface elimination, which unfortunately does not generally work. Compute a *representative point* on each polygon (e.g. the centroid or the furthest point to the viewer). Sort the objects by decreasing order of distance from the viewer to the representative point (or using the pseudodepth which we discussed in discussing perspective) and draw the polygons in this order. Unfortunately, just because the representative points are ordered, it does not imply that the entire polygons are ordered. Worse yet, it may be *impossible* to order polygons so that this type of algorithm will work. The following figure shows such an example, in which the polygons overlap one another cyclically.

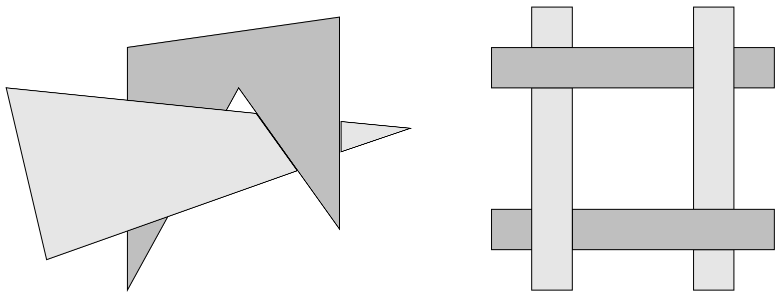


Figure 71: Hard cases to depth-sort.

In these cases we may need to *cut* one or more of the polygons into smaller polygons so that the depth order can be uniquely assigned. Also observe that if two polygons do not overlap in x, y space, then it does not matter what order they are drawn in.

Here is a snapshot of one step of the depth-sort algorithm. Given any object, define its *z-extents* to be an interval along the z-axis defined by the object's minimum and maximum z-coordinates. We begin by sorting the polygons by depth using furthest point as the representative point, as described above. Let's consider the polygon P that is currently at the end of the list. Consider all polygons Q whose z-extents overlaps P's. This can be done by walking towards the head of the list until finding the first polygon whose maximum z-coordinate is less than P's minimum z-coordinate. Before drawing P we apply the following tests to each of these polygons Q. If any answers is "yes", then we can safely draw P before Q.

(1) Are the x-extents of P and Q disjoint?

(2) Are the y-extents of P and Q disjoint?

(3) Consider the plane containing Q. Does P lie entirely on the opposite side of this plane from the viewer?

(4) Consider the plane containing P. Does Q lie entirely on the same side of this plane from the viewer?

(5) Are the projections of the polygons onto the view window disjoint?

In the cases of (1) and (2), the order of drawing is arbitrary. In cases (3) and (4) observe that if there is any plane with the property that P lies to one side and Q and the viewer lie to the other side, then P may be drawn before Q. The plane containing P and the plane containing Q are just two convenient planes to test. Observe that tests (1) and (2) are very fast, (3) and (4) are pretty fast, and that (5) can be pretty slow, especially if the polygons are nonconvex.

If all tests fail, then are only hope to resolve the situation is to split one or both of the polygons. Before doing this, we first see whether this can be avoided by putting Q at the end of the list, and then applying the process on Q. To avoid going into infinite loops, we mark each polygon once it is moved to the back of the list. Once marked, a polygon is never moved to the back again. If a marked polygon fails all the tests, then we need to split. To do this, we use P's plane like a knife to split Q. We then take the resulting pieces of Q, compute the furthest point for each and put them back into the depth sorted list.

In theory this partitioning could generate $O(n^2)$ individual polygons, but in practice the number of polygons is much smaller. The depth-sort algorithm needs no storage other than the frame buffer and a linked list for storing the polygons (and their fragments). However, it suffers from the deficiency that each pixel is written as many times as there are overlapping polygons.

Depth-buffer Algorithm: The depth-buffer algorithm is one of the simplest and fastest hidden-surface algorithms. Its main drawbacks are that it requires a lot of memory, and that it only produces a result that is accurate to pixel resolution and the resolution of the depth buffer. Thus the result cannot be scaled easily and edges appear jagged (unless some effort is made to remove these effects called "aliasing"). It is also called the *z-buffer* algorithm. This algorithm assumes that for each pixel we store two pieces of information, (1) the color of the pixel (as usual), and (2) the depth of the object that gave rise to this color. This is called the *depth-buffer* (or *z-buffer*, since z is the axis used to store depth information). Initially the depth-buffer values are set to the maximum depth value.

Suppose that we have a k-bit depth buffer, implying that we can store integer depths ranging from 0 to $D = 2^k - 1$. After applying the perspective-with-depth transformation, we know that all depth values have been scaled to the range $[-1, 1]$. We scale the depth value to the range of the depth-buffer and convert this to an integer, e.g. $\lfloor (z+1)/(2D) \rfloor$. If this depth is less than or equal to the depth at this point of the buffer, then we store its RGB value in the color buffer. Otherwise we do nothing.

This algorithm is favored for hardware implementations because it is so simple and essentially reuses the same algorithms needed for basic scan conversion.

Scan Conversion for the Depth-Buffer Algorithm: Consider the process of scan-converting a triangle shown in the figure below using a depth-buffer.

Let P_0, P_1, and P_2 be the vertices of the triangle after the perspective-plus-depth transformation has been applied, and the points have been scaled to the screen size. Let $P_i = (x_i, y_i, z_i)$ be the coordinates of each vertex, where (x_i, y_i) are the final screen coordinates and z_i is the depth of this point.

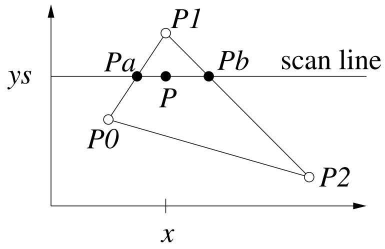


Figure 72: Depth-buffer scan conversion.

Scan-conversion takes place by scanning along each row of pixels that this triangle overlaps. Based on the y-coordinates of the current scan line y_s and the y-coordinates of the vertices of the triangle, we can interpolate the depth of at the endpoints P_a and P_b of the scan-line. For example, given the configuration in the figure, we have:

$$\rho_a = \frac{y_s - y_0}{y_1 - y_0}$$

is the ratio into which the scan line subdivides the edge P_0P_1. The depth of point P_a, can be interpolated by the following affine combination

$$z_a = (1 - \rho_a)z_0 + \rho_a z_1.$$

(Is this really an accurate interpolation of the depth information? Remember that the projection transformation is nonlinear, but the result may surprise you. We'll leave this question as an exercise.) We can derive a similar expression for z_b.

Then as we scan along the scan line, for each value of y we have

$$\alpha = \frac{x - x_a}{x_b - x_a},$$

and the depth of the scanned point is just the affine combination

$$z = (1 - \alpha)z_a + \rho z_b.$$

It is more efficient (from the perspective of the number of arithmetic operations) to do this by computing z_a accurately, and then adding a small incremental value as we move to each successive pixel on the line. The scan line traverses $x_b - x_a$ pixels, and over this range, the depth values change over the range $z_b - z_a$. Thus, the change in depth per pixel is

$$\Delta_z = \frac{z_b - z_a}{x_b - x_a}.$$

Starting with z_a, we add the value Δ_z to the depth value of each successive pixel as we scan across the row. An analogous trick may be used to interpolate the depth values along the left and right edges.

Lecture 28: Light and Color

(Thursday, Dec 7, 2000)
Read: Chapter 12 in Hill. Our book does not discuss Gamma correction.

Achromatic Light and Gamma Correction: Light and its perception are important to understand for anyone interested in computer graphics. Before considering color, we begin by considering some issues in the perception of light intensity and the generation of light on most graphics devices. Let us consider color-free, or *achromatic light*, that is gray-scale light. It is characterized by one attribute: *intensity* which is a measure of energy, or *luminance*, which is the intensity that we perceive. Intensity affects brightness, and hence low intensities tend to black and high intensities tend to white. Let us assume for now that each intensity value is specified as a number from 0 to 1, where 0 is black and 1 is white.

You would think that intensity and luminance are directly proportional to each other. Twice the intensity is perceived as being twice as bright. However, the human perception of luminance is nonlinear. For example, suppose we want to generate 10 different intensities, producing a uniform continuous variation from black to white on a typical CRT display. It would seem logical to use equally spaced intensities: $0.0, 0.1, 0.2, \ldots, 0.9$. However our eye does not perceive these intensities as varying uniformly. The reason is that the eye is sensitive to *ratios* of intensities, rather than absolute differences. Thus, 0.2 appears to be twice as bright as 0.1, but 0.6 only appears to be 20% brighter than 0.5. Therefore, to achieve *perceptual* uniformity, intensities should be chosen on a *logarithmic*, rather than a linear scale.

To make things more complicated, there is not a linear relation between the voltage supplied to the electron gun of the CRT and the intensity of the resulting phosphor. The relationship between voltage and brightness of the phosphors is more closely approximated by:

$$I = V^\gamma,$$

where I denotes the intensity of the pixel and V denotes the voltage on the signal (which is proportional to the RGB values you store in your frame buffer), and γ is a constant that depends on physical properties of the display device (which ranges typically from 1.5 to 2.5 for CRT monitors). The term *gamma* refers to the nonlinearity of the transfer function. *Gamma correction* is the process of altering the pixel values the inverse of this function. In a system that does not do gamma correction, the problem is that low voltages produce unnaturally intensities compared to high voltages. The result is that dark colors appear unnaturally dark. In order to correct this effect, modern monitors provide the capability of gamma correction. In order to achieve a desired intensity I, we instead aim to produce a corrected intensity:

$$I' = I^{1/\gamma}.$$

Thus, when the gamma effect is taken into account, we will get the desired intensity.

Many high-end graphics displays (like SGI's) provide a form of automatic gamma correction. In most PC's the gamma can be adjusted manually. However, even with gamma correction, do not be surprised if the same RGB values will produce exactly the same colors on different systems. As an experiment, run the `xv` program on your favorite image. Open the color editor window and try entering different values of gamma to see what the effect is on the final image.

Light and Color: Light as we perceive it is *electromagnetic radiation* from a narrow band of the complete spectrum of electromagnetic radiation called the *visible spectrum*. The physical nature of light has elements that are like particle (when we discuss photons) and as a wave. Recall that wave can be described either in terms of its *frequency*, measured say in cycles per second, or the inverse quantity of *wavelength*. The electro-magnetic spectrum ranges from very low frequency (high wavelength) radio waves (greater than 10 centimeter in wavelength) to microwaves, infrared, visible light, ultraviolet and x-rays and high frequency (low wavelength) gamma rays (less than 0.01 nm in wavelength). Visible light lies in the range of wavelengths from around 400 to 700 nm. Recall that nm denotes a nanometer which is 10^{-7} of a centimeter.

Physically, the light energy that we perceive as color can be described in terms of a function of wavelength λ, called the *spectral distribution function* or simply *spectral function*, $f(\lambda)$. As we walk along the wavelength axis (from short to long wavelengths), the associated colors that we perceive varying along the colors of the rainbow from violet to blue, green, yellow, orange and red.

The Eye and Color Perception: Light and color are complicated in computer graphics for a number of reasons. The first is that the *physics* of light is very complex. Secondly, our *perception* of light is a function of our optical systems, which perform numerous unconscious corrections and modifications to the light we see.

The retina of the eye is a light sensitive membrane, which contains two types of light-sensitive receptors, *rods* and *cones*. Cones are color sensitive. There are three different types, which are selectively more sensitive to red, green, or blue light. There are from 6 to 7 million cones concentrated in the *fovea*, which corresponds to the center of your view. The *tristimulus theory* states that we perceive color as a mixture of these three colors.

Blue cones: peak response around 440 nm with about 2% of light absorbed by the cone.

Green cones: peak response around 545 nm with about 20% of light absorbed by the cone.

Red cones: peak response around 580 nm, with about 19% of light absorbed by the cone.

The different absorption rates comes from the fact that we have far fewer blue sensitive cones in the fovea as compared with red and green. Rods in contrast occur in lower density in the fovea, and do not distinguish color. However they are sensitive to low light and motion, and hence serve a function for vision at night.

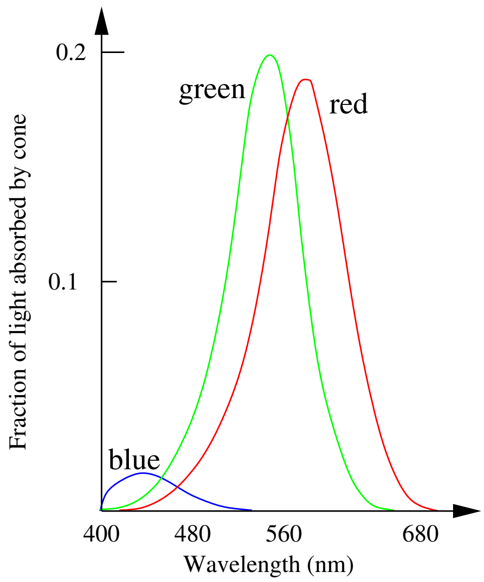


Figure 73: Spectral response curves for cones (adapted from Foley, vanDam, Feiner and Hughes).

It is possible to produce light within a very narrow band of wavelengths using lasers. Note that because of our limited ability to sense light of different colors, there are many different spectra that appear to us to be the same color. These are called *metamers*. Thus, spectrum and color are not in 1–1 correspondence. Most of the light we see is a mixture of many wavelengths combined at various strengths. For example, shades of gray varying from white to black all correspond to fairly flat spectral functions.

Describing Color: Throughout this semester we have been very lax about defining color carefully. We just spoke of RGB values as if that were enough. However, we never indicated what RGB means, independently from the notion that they are the colors of the phosphors on your display. How would you go about describing color precisely, so that, say, you could unambiguously indicate exactly what shade you wanted in a manner that is independent of the display device? Obviously you could give the spectral function, but that would be overkill (since many spectra correspond to the same color) and it is not clear how you would find this function in the first place.

There are three components to color, which seem to describe color much more predictably than does RGB. These are hue, saturation, and lightness. The *hue* describes the dominant wavelength of the color in terms of one of the pure colors of the spectrum that we gave earlier. The *saturation* describes how pure the light is. The red color of a fire-engine is highly saturated, whereas pinks and browns are less saturated, involving mixtures with grays. Gray tones (including white and black) are the most highly unsaturated colors. Of course lightness indicates the intensity of the color. But although these terms are somewhat more intuitive, they are hardly precise.

The tristimulus theory suggests that we perceive color by a process in which the cones of the three types each send signals to our brain, which sums these responses and produces a color. This suggests that there are three "primary" spectral distribution functions, $R(\lambda)$, $G(\lambda)$, and $B(\lambda)$, and every saturated color that we perceive can be described as a linear combination of these three:

$$C = rR + gG + bB.$$

(This means that we weight these three functions by the scalars r, g, and b and integrate over the entire spectrum, then C is the color that we perceive.)

Extensive studies with human subjects have shown that it is indeed possible to define saturated colors as a combination of three spectra, but the result has a very strange outcome. Some colors can only be formed by allowing some of the coefficients r, g, or b to be negative. E.g. there is a color C such that

$$C = 0.7R + 0.5G - 0.2B.$$

We know what it means to form a color by adding light, but we cannot subtract light that is not there. The way that this equation should be interpreted is that we cannot form color C from the primaries, but we can form the color $C + 0.2B$ by combining $0.7R + 0.5G$. When we combine colors in this way they are no longer pure, or saturated. Thus such a color C is in some sense *super saturated*, since it cannot be formed by a purely additive process.

The CIE Standard: In 1931, a commission was formed to attempt to standardize the science of colorimetry. This commission was called the Commission Internationale de l'Éclairage, or CIE.

The results described above lead to the conclusion that we cannot describe all colors as positive linear combinations of three primary colors. So, the commission came up with a standard for describing colors. They defined three special *super saturated* X, Y, and Z, which do not correspond to any real colors, but they have the property that every real color can be represented as a positive linear combination of these three.

The resulting 3-dimensional space, and hence is hard to visualize. A common way of drawing the diagram is to consider a single 2-dimensional slice, by normalize by cutting with the plane $X + Y + Z = 1$. We can then project away the Z component, yielding the *chromaticity coordinates*:

$$x = \frac{X}{X + Y + Z} \qquad y = \frac{Y}{X + Y + Z}$$

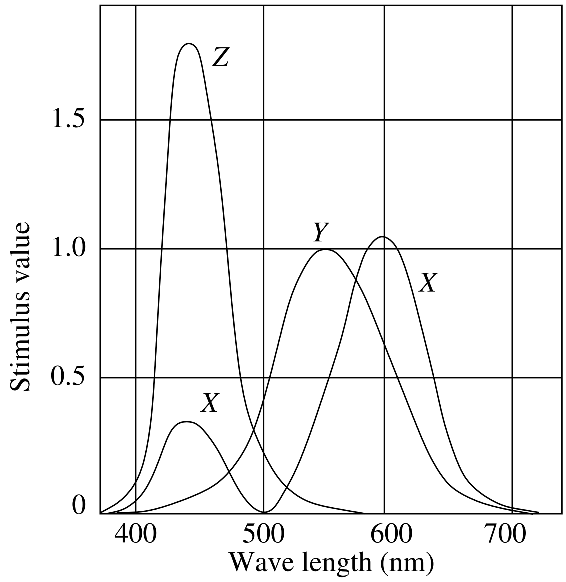


Figure 74: CIE primary colors (adapted from Hearn and Baker).

(and z can be defined similarly). These components describe just the color of a point. Its brightness is a function of the Y component. (Thus, an alternative, but seldom used, method of describing colors is as xyY.)

If we plot the various colors in this (x, y) coordinates produce a 2-dimensional "shark-fin" convex shape shown in the figure below. Let's explore this figure a little. Around the curved top of the shark-fin we see the colors of the spectrum, from the long wavelength red to the short wavelength violet. The top of the fin is green. Roughly in the center of the diagram is white. The point C corresponds nearly to "daylight" white. As we get near the boundaries of the diagram we get the purest or most *saturated* colors (or *hues*). As we move towards C, the colors become less and less saturated.

The CIE model is useful for providing formal specifications of any color as a 3-element vector, however it is not the easiest way to produce color in hardware. Typical hardware devices like CRT's, televisions, and printers use other standards that are more convenient for generation purposes. Unfortunately, neither CIE nor these models is particularly intuitive from a user's perspective.

Lecture 29: Final Review

(Tuesday, Dec 12, 2000)

Read: Since the midterm we have covered (parts of) Chapters 6, and 10–14 in Hill.

Overview: This semester we have presented an introduction to computer graphics. The main topics that we covered included the basic elements of 3-dimensional rendering (including basic geometry, affine transformations, the OpenGL API, projective geometry and perspective), lighting and shading using the Phong model, providing surface details through texture and bump mapping, basic solid modeling (including CSG and Bezier curves and surfaces), highly realistic rendering through ray tracing, and basic implementation issues including (scan conversion and hidden-surface removal), and finally light and color modeling.

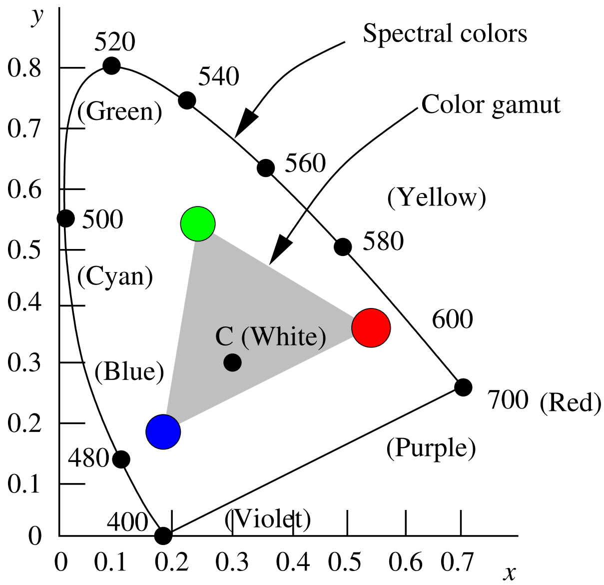


Figure 75: CIE chromaticity diagram and color gamut (adapted from Hearn and Baker).

What we didn't cover: There is just too much material about computer graphics to squeeze into one graphics course. Some issues include:

Anti-aliasing: A important problem in graphics is the presense of jagged-edges. These are quite noticeable if you look closely at the ray-traced images from the last project. What techniques can be used to reduce these jagged effects? We did not cover this, since a good treatment of the subject would require a deeper understanding of sampling theory.

More global illumination models: We have seen numerous times this semester where we "faked it" for the sake of speed, even though we knew that our models might not be completely realistic. However there are much more realistic global illumination models that have been constructed, which model indirect illumination. They work on the assumption that most surfaces reflect a certain amount of the light that shines upon them, and so all objects act as light sources. One such method is called *radiosity*.

More on modeling: We could have spent a whole semester discussing aspects of solid modeling. We did not discuss the various types of spline curves and surfaces and NURBS (nonuniform rational B-splines) and we did not discuss fractal models. There are also a number of issues involved in representing and simplifying large complex surface models.

More issues with color: We just introduced color, but we did not discuss many of the issues that are involved here. One involves conversion between different color systems. Another is the issue of approximating a large range of colors using a small look-up table of color values, through the process of dithering.

Animation: Once you know how to make a single image, the next question is how to put them together to form an animation. The main issues in animation involve topics such as how animation is specified (frame by frame or though some parametrically defined process), smooth interpolation between frames and morphing, temporal antialiasing issues, and kinematics and dynamics. We also could have discussed modeling more complex phenomena, for example, using particle systems to model movements of multiple objects. We would also need to discuss modeling 3-dimensional rotations and quaternions.

Topics: The final exam will be comprehensive but will emphasize the material since the midterm. Here are the main topics that we have covered since the midterm.

Old material: Do not neglect going over basic affine geometry. Although this was covered in the first part of the semester, it is something that will be needed everywhere in graphics.

Surface mappings: This was covered on the midterm, but there may be a couple of residual questions on the topics of texture mapping and bump mapping.

Solid Modeling: We discussed a number of issues related to the representation of solid objects.

Polygonal Meshes and DCEL: We discussed representing objects as a patches of polygons, and the DCEL (double-connect edge list) data structure for representing the topological connections between patches.

CSG Trees: Constructive solid geometry involves the representation of objects as boolean operations (union, intersection, subtraction) of primitive shapes.

Curves and Surfaces: We introduced the notion of C^k continuity. We discussed Bezier curves and surfaces, their geometric properties, and subdivision methods. (B-splines will not be covered.)

Ray Tracing: We discussed ray tracing in depth as a method for generating very realistic images, especially in the presence of reflection and refraction effects. We also discussed the related issues of:

Ray intersection: Compute the intersection between a ray and various types of objects. We considered spheres, triangles, and Bezier-surfaces.

Procedural 3-d texture: As part of the project we introduced the notion of a 3-dimensional texture, a function that maps a point in 3-space to a color (as opposed to the texture map method which first reduces the point to a 2-dimensional point in parameter space).

Scan conversion: We discussed Bresenham's algorithm for scan converting lines. We also discussed how to scan convert polygons, using a carefully designed rule for handling degenerate situations so that pixels that fall on the edges or vertices of a polygon are only rendered once.

Hidden-Surface algorithms: The main algorithms we considered were the following.

Back-face culling: Not a general hidden-surface algorithm, but a quick and dirty heuristic for removing faces on the back sides of objects from consideration.

Depth sort: Sorting the objects by depth and painting from back to front.

Depth-buffer: Scan-converting polygons but storing depth in addition to color.

Light and Color: We introduced the basic elements of light and color and how they can be modeled. We discussed gamma correction, the tristimulus theory (color is formed from the mixture of three primary colors) and the CIE color model.

Lecture X01: More on Graphics Systems and Models

(Supplemental)

Read: This material is not covered in our text. See the OpenGL Programming Guide Chapt 3 for discussion of the general viewing model.

Image Synthesis: In a traditional bottom-up approach to computer graphics, at this point we would begin by discussing how pixels are rendered to the screen to form lines, curves, polygons, and eventually build up to 2-d and then to 3-d graphics.

Instead we will jump directly into a discussion 3-d graphics. We begin by considering a basic model of viewing, based on the notion of a viewer holding up a *synthetic-camera* to a model of the scene that we wish to render. This implies that our graphics model will involve the following major elements:

Objects: A description of the 3-dimensional environment. This includes the geometric structure of the objects in the environment, their colors, reflective properties (texture, shininess, transparency, etc).

Light sources: A description of the locations of light sources, their shape, and the color and directional properties of their energy emission.

Viewer: A description of the location of the viewer and the position and properties of the synthetic camera (direction, field of view, and so on).

Each of these elements may be described to a greater or lesser degree of precision and realism. Of course there are trade-offs to be faced in terms of the efficiency and realism of the final images. Our goal will be to describe a model that is as rich as possible but still fast enough to allow real time animation (say, at least 20 frames per second) on modern graphics workstations.

Geometric Models: The first issue that we must consider is how to describe our 3-dimensional environment in a manner that can be processed by our graphics API. As mentioned above, such a model should provide information about geometry, color, texture, and reflective properties for these objects. Models based primarily around simple mathematical structures are most popular, because they are easy to program with. (It is much easier to render a simple object like a sphere or a cube or a triangle, rather than a complex object like a mountain or a cloud, or a furry animal.)

Of course we would like our modeling primitives to be flexible enough that we can model complex objects by combining many of these simple entities. A reasonably flexible yet simple method for modeling geometry is through the use of *polyhedral models*. We assume that the solid objects in our scene will be described by their 2-dimensional boundaries. These boundaries will be assumed to be constructed entirely from flat elements (points, line segments, and planar polygonal faces). Later in the semester we will discuss other modeling methods involving curved surfaces (as arise often in manufacturing) and bumpy irregular objects (as arise often in nature).

The boundary of any polyhedral object can be broken down into its boundary elements of various dimensions:

Vertex: Is a (0-dimensional) point. It is represented by its (x, y, z) coordinates in space.

Edge: Is a (1-dimensional) line segment joining two vertices.

Face: Is a (2-dimensional) planar polygon whose boundary is formed by a closed cycle of edges.

The way in which vertices, edges and faces are joined to form the surface of an object is called its *topology*. An object's topology is very important for reasoning about it properties. (For example, a robot system may want to know whether an object has an handles which it can use to pick the object up with.) However, in computer graphics, we are typically only interested in what we need to render the object. These are its faces.

Faces form the basic rendering elements in 3-dimensional graphics. Generally speaking a face can be defined by an unlimited number of edges, and in some models may even contain polygonal *holes*. However, to speed up the rendering process, most graphics systems assume that faces consist of simple convex polygons. A shape is said to be *convex* if any line intersects the shape in a single line segment. Convex polygons have internal angles that are at most 180 degrees, and contain no holes.

Since you may want to have objects whose faces are not convex polygons, many graphics API's (OpenGL included) provide routines to break complex polygons down into a collection of convex polygons, and triangles in particular (because all triangles are convex). This process is called *triangulation* or *tesselation*. This increases the number of faces in the model, but it significantly simplifies the rendering process.

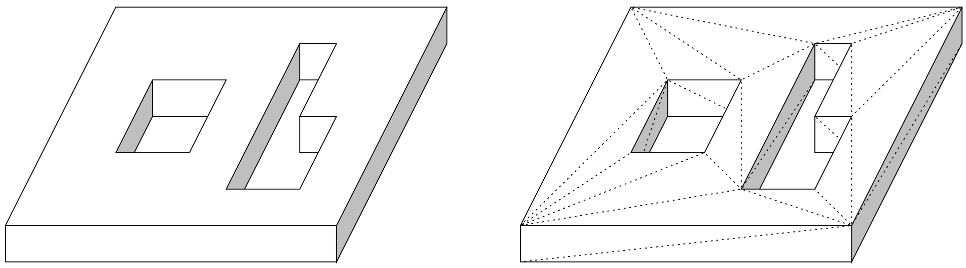


Figure 76: A polyhedral model and one possible triangulation of its faces.

In addition to specifying geometry, we also need to specify color, texture, surface finish, etc in order to complete the picture. These elements affect how light is reflected, giving the appearance of dullness, shininess, bumpiness, fuzziness, and so on. We will discuss these aspects of the model later. This is one of the most complex aspects of modeling, and good surface modeling may require lots of computational time. In OpenGL we will be quite limited in our ability to affect surface finishes.

Light and Light Sources: The next element of the 3-dimensional model will be the light sources. The locations of the light sources will determine the *shading* of the rendered scene (which surfaces are light and which are dark), and the location of shadows. There are other important elements with light sources as well. The first is shape of the light source. Is it a point (like the sun) or does it cover some area (like a florescent light bulb). This affects things like the sharpness of shadows in the final image. Also objects (like a brightly lit ceiling) can act as indirect reflectors of light. In OpenGL we will have only point light sources, and we will ignore indirect reflection. We will also pretty much ignore shadows, but there are ways of faking them. These models are called *local illumination models*.

The next is the color of the light. Incandescent bulbs produce light with a high degree of red color. On the other hand florescent bulbs produce a much bluer color of light. Even the color of the sun is very much dependent on location, time of year, time of day. It is remarkable how sensitive the human eye is to even small variations.

The light that is emitted from real light sources is a complex spectrum of electromagnetic energy (over the visible spectrum, wavelengths ranging from 350 to 780 nanometers). However to simplify things, in OpenGL (and almost all graphics systems) we will simply model emitted light as some combination of red, green and blue color components. (This simple model cannot easily handle some phenomenon such as rainbows.)

Just how light reflects from a surface is a very complex phenomenon, depending on the surface qualities and microscopic structure of object's surface. Some objects are smooth and shiny and others are matte (dull). OpenGL models the reflective properties of objects by assuming that each object reflects light in some combination of these extremes. Later in the semester we will discuss shiny or *specular reflection*, and dull or *diffuse reflection*. We will also model indirect reflection (light bouncing from other surfaces) by assuming that there is a certain amount of *ambient light*, which is just floating around all of space, without any origin or direction.

Later we will provide an exact specification for how these lighting models work to determine the brightness and color of the objects in the scene.

Camera Model: Once our 3-dimensional *scene* has been modeled, the next aspect to specifying the image is to specify the location and orientation of a synthetic camera, which will be taking a picture of the scene.

Basically we must *project* a 3-dimensional scene onto a 2-dimensional imaging window. There a number of ways of doing this. The simplest is called a *parallel projection* where all objects are projected along parallel lines, and the other is called *perspective projection* where all objects are projected along lines that meet at a common point. Parallel projection is easier to compute, but perspective projections produce more realistic images.

One simple camera model is that of a *pin-hole camera*. In this model the camera consists of a single point called the *center of projection* on one side of a box and on the opposite side is the *imaging plane* or *view plane* onto which the image is projected.

Let us take a moment to consider the equations that define how a point in 3-space would be projected to our view plane. To simplify how perspective views are taken, let us imaging that the camera is pointing along the positive z-axis, the center of projection is at the origin, and the imaging plane is distance d behind the center of projection (at $z = -d$). Let us suppose that the box is h units high (along the y-axis) and w units wide (along the x-axis).

A side view along the yz-plane is shown below. Observe that, by similar triangles, a point with coordinates (y, z) will be projected to the point

$$y_p = -\frac{y}{z/d},$$

and by a similar argument the x-coordinate of the projection will be

$$x_p = -\frac{x}{z/d}.$$

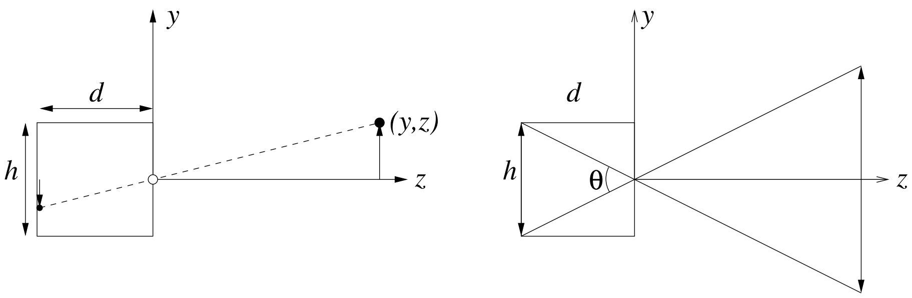


Figure 77: Pinhole camera.

Thus once we have transformed our points into this particular coordinate system, computing a perspective transformation is a relatively simple operation.

$$(x, y, z) \Rightarrow \left(-\frac{x}{z/d}, -\frac{y}{z/d}, -d\right).$$

The z-coordinate of the result is not important (since it is the same for all projected points) and may be discarded.

Finally observe that this transformation is not defined for all points in 3-space. First off, if $z = 0$, then the transformation is undefined. Also observe that this transformation has no problem projecting points that lie behind the camera. For this reason it will be important to *clip* away objects that lie behind plane $z = 0$ before applying perspective.

Even objects that lie in front of the center of projection may not appear in the final image, if their projection does not lie on the rectangular portion of the image plane. By a little trigonometry, it is easy to figure out what is the angular diameter θ of the cone of visibility. Let us do this for the yz-plane. This is called the *field of view* (for y). (A similar computation could be performed for the xz-plane). The first rule of computing angles is to reduce everything to right triangles. If we bisect θ by the z-axis, then we see that it lies in a right triangle whose opposite leg has length $h/2$ and whose adjacent leg has length d, implying that $\tan(\theta/2) = h/(2d)$. Thus, the field of view is

$$\theta = 2 \arctan \frac{h}{2d}.$$

Observe that the image has been inverted in the projection process. In real cameras it is not possible to put the film in from of the lens, but there is not reason in our mathematical model that we should be limited in this way. Consequently, when we introduce the perspective transformation later, we assume that the view plane is in front of the center of projection, implying that the image will not be inverted.

Before moving on we should point out one important aspect of this derivation. Reasoning in 3-space is very complex. We made two important assumptions to simplify our task. First, we selected a convenient frame of reference (by assuming that the camera is pointed along the z-axis, and the center of projection is the origin. The second is that we projected the problem to a lower dimensional space, where it is easier to understand. First we considered the yz-plane, and reasoned by analogy to the xy-plane. Remember these two ideas. They are fundamental to getting around the complexities of geometric reasoning.

But what if you camera is not pointing along the z-axis? Later we will learn how to perform transformations of space, which will map objects into a coordinate system that is convenient for us.

Camera Position: Given our 3-dimensional scene, we need to inform the graphics system where our camera is located. This usually involves specifying the following items:

Camera location: The location of the center of projection.

Camera direction: What direction (as a vector) is the camera pointed in.

Camera orientation: What direction is "up" in the final image.

Focal length: The distance from the center of projection to the image plane.

Image size: The size (and possibly location) of the rectangular region on the image plane to be taken as the final image.

There are a number of ways of specifying these quantities. For example, rather than specifying focal length and image size, OpenGL has the user specify the field of view and the image *aspect ratio*, the ratio of its width (x) to height (y).

At this point, we have outlined everything that must be specified for rendering a 3-dimensional scene (albeit with a considerable amount of simplification and approximation in modeling). Next time we will show how to use OpenGL to turn this abstract description into a program, which will render the scene.

Lecture X02: X Window System

(Supplemental)
Read: Chapter 1 in Hill.

X Window System: Although Window systems are not one of the principal elements of a graphics course, some knowledge about how typical window systems works is useful. We will discuss elements of the X-window system, which is typical of many window systems, such as Windows95.

X divides the *display* into rectangular regions called *windows*. (Note: The term *window* when used later in the context of graphics will have a different meaning.) Each window acts as an input/output area for one or more processes. Windows in X are organized hierarchically, thus each window (except for a special one called the *root* that covers the entire screen) has a *parent* and may have one or more *child* windows that it creates and controls. For example, menus, buttons, scrollbars are typically implemented as child windows.

Window systems like X are quite complex in structure. The following are components of X.

X-protocol: The lowest level of X provides routines for communicating with graphics devices (which may reside elsewhere on some network).

One of the features of X is that a program running on one machine can display graphics on another by sending graphics (in the form of messages) over the network. Your program acts like a *client* and sends commands to the *X-server* which talks to the display to make the graphics appear. The server also handles graphics resources, like the color map.

Xlib: This is a collection of library routines, which provide low-level access to X functions. It provides access to routines for example, creating windows, setting drawing colors, drawing graphics (e.g., lines, circles, and polygons), drawing text, and receiving input either through the keyboard or mouse.

Another important aspect of Xlib is maintaining a list of user's preferences for the appearance of windows. Xlib maintains a database of desired window properties for various applications (e.g., the size and location of the window and its background and foreground colors). When a new application is started, the program can access this database to determine how it should configure itself.

Toolkit: Programming at the Xlib level is extremely tedious. A toolkit provides a higher level of functionality for user interfaces. This includes objects such as menus, buttons, and scrollbars.

When you create a button, you are not concerned with how the button is drawn or whether its color changes when it is clicked. You simply tell it what text to put into the button, and the system takes care drawing the button, and informing your program when a the button has been selected. The X-toolkit functions translate these requests into calls to Xlib functions. We will not be programming in Xlib or the X-toolkit this semester. We will discuss GLUT later. This is a simple toolkit designed for use with OpenGL.

Graphics API: The toolkit supplies tools for user-interface design, but it does little to help with graphics. Dealing with the window system at the level of drawing lines and polygons is very tedious when designing a 3-dimensional graphics program. A graphics API (application programming interface) is a library of functions which provide high-level access to routines for doing 3-dimensional graphics. Examples of graphics API's include PHIGS, OpenGL (which we will be using), and Java3D.

Window Manager: When you are typing commands in a Unix system, you are interacting with a program called a *shell*. Similarly, when you resize windows, move windows, delete windows,

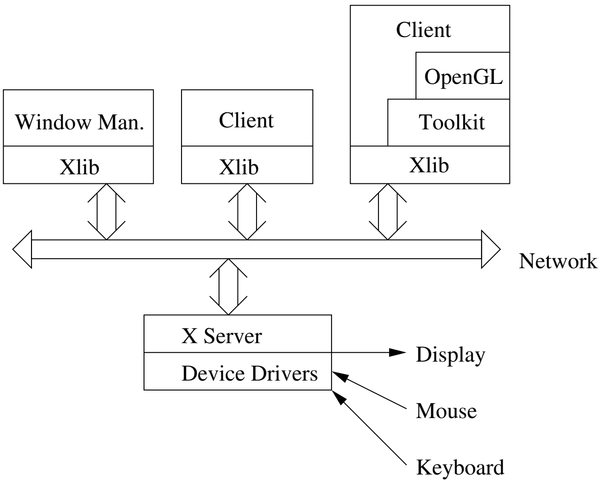


Figure 78: X-windows client-server structure.

you are interacting with a program called a *window manager*. A window manager (e.g. twm, ctwm, fvwm) is just an application written in X. It's only real "privilege" with respect to the X system is that it has final say over where windows are placed. Whenever a new window is created, the window-manager is informed of its presence, and informs approves (or determines) its location and placement.

It is the window manager's job to control the layout of the various windows, determine where these windows are to be placed, and which windows are to be on top. Neither the window manager, nor X, is responsible for saving the area of the screen where a window on top obscures a window underneath. Rather, when a window goes away, or is moved, X informs the program belonging to the obscured window that it has now been "exposed". It is the job of the application program to redraw itself.

Lecture X03: Ray-Polyhedron Intersectino

(Supplementary)

Comment: The representation of a plane as a homogeneous equation here should probably be replaced by a representation as a point and an outward pointing normal. See the lecture on the Liang-Barsky clipping algorithm.

Ray-Polyhedron Intersection: We present an intersection algorithm for a ray and a convex polyhedron. The convex polyhedron is defined as the intersection of a collection of halfspaces in 3-space. (This algorithm is a variation of the Liang-Barsky line segment clipping algorithm, which is used for clipping line segments against the 3-dimensional view volume, which was introduced when we were discussing perspective.) As before, we represent the ray parametrically as $P + t\vec{u}$, for scalar $t > 0$. Let $H_1, H_2, \ldots, H_k$ denote the halfspaces defining the polyhedron. We will compute the intersection of the ray with each halfspace in turn. The final result will be the intersection of the ray with the entire polyhedron.

An important property of convex bodies (of any variety) is that a line intersects a convex body in at most one line segment. Thus the intersection of the ray with the polyhedron can be specified entirely by an interval of scalars $[t_0, t_1]$, such that the intersection is defined by the set of points

$$P + t\vec{u} \qquad \text{for } t_0 \leq t \leq t_1.$$

Initially, let this interval be $[0, \infty]$. (For line segment intersection the only change is that the initial value of t_1 is set so that we end and the endpoint of the segment. Otherwise the algorithm is identical.)

Suppose that we have already performed the intersection with some number of the halfspaces. It might be that the intersection is already empty. This will be reflected by the fact that $t_0 > t_1$. When this is so, we may terminate the algorithm at any time. Otherwise, let $H = (a, b, c, d)$ be the coefficients of the current halfspace.

We want to know the value of t (if any) at which the ray intersects the plane. Plugging in the representation of the ray into the halfspace inequality we have

$$a(p_x + t\vec{u}_x) + b(p_y + t\vec{u}_y) + c(p_z + t\vec{u}_z) + d \leq 0,$$

which after some manipulations is

$$t(a\vec{u}_x + b\vec{u}_y + c\vec{u}_z) \leq -(ap_x + bp_y + cp_z + d).$$

If P and $\vec{u}$ are given in homogeneous coordinates, this can be written as

$$t(H \cdot \vec{u}) \leq -(H \cdot P).$$

This is not really a legitimate geometric expression (since dot product should only be applied between vectors). Actually the halfspace H should be thought of as a special geometric object, a sort of *generalized normal vector*. For example, when transformations are applied, normal vectors should be multiplied by the inverse transpose matrix to maintain orthogonality.

We consider three cases.

$(H \cdot \vec{u}) > 0$: In this case we have the constraint

$$t \leq \frac{-(H \cdot P)}{(H \cdot \vec{u})}.$$

Let t^* denote the right-hand side of this inequality. We trim the high-end of the intersection interval to $[t_0, \min(t_1, t^*)]$.

$(H \cdot \vec{u}) < 0$: In this case we have

$$t \geq \frac{-(H \cdot P)}{(H \cdot \vec{u})}.$$

Let t^* denote the right-hand side of this inequality. In this case, we trim the low-end of the intersection interval to $[\max(t_0, t^*), t_1]$.

$(H \cdot \vec{u}) = 0$: In this case the ray is parallel to the plane. Either entirely above or below. We check the origin. If $(H \cdot P) \leq 0$ then the origin lies in (or on the boundary of) the halfspace, and so we leave the current interval unchanged. Otherwise, the origin lies outside the halfspace, and the intersection is empty. To model this we can set t_1 to any negative value, e.g., -1.

After we repeat this on each face of the polyhedron, we have the following possibilities:

$t_1 < t_0$: In this case the ray does not intersect the polyhedron.

$0 = t_0 \leq t_1$: In this case, the origin is within the polyhedron. If $t_1 = \infty$, then the polyhedron must be unbounded (e.g. like a cone) and there is no intersection. Otherwise, the first intersection point is the point $P + t_1\vec{u}$.

$0 < t_0 \leq t_1$: In this case, the origin is outside the polyhedron, and the first intersection is at $P + t_0\vec{u}$.

As with spheres it is a good idea to check against a small positive number, rather than 0 exactly, because of floating point errors. For ray tracing applications, when we set the value of either t_0 or t_1, it is a good idea to also record which halfspace we intersected. This will be useful if we want to know the normal vector at the point of intersection (which will be (a, b, c) for the current halfspace).

Lecture X04: Scan Conversion of Circles

(Supplementary)

Read: Section 3.3 in Foley, vanDam, Feiner and Hughes.

Midpoint Circle Algorithm: Let us consider how to generalize Bresenham's midpoint line drawing algorithm for the rasterization of a circle. We will make a number of assumptions to simplify the presentation of the algorithm. First, let us assume that the circle is centered at the origin. (If not, then the initial conditions to the following algorithm are changed slightly.) Let R denote the (integer) radius of the circle.

The first observations about circles is that it suffices to consider how to draw the arc in the positive quadrant from $\pi/4$ to $\pi/2$, since all the other points on the circle can be determined from these by *8-way symmetry.*

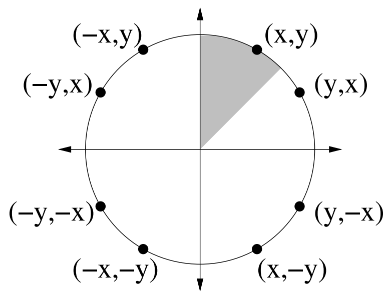


Figure 79: 8-way symmetry for circles.

What are the comparable elements of Bresenham's midpoint algorithm for circles? As before, we need an implicit representation of the function. For this we use

$$F(x, y) = x^2 + y^2 - R^2 = 0.$$

Note that for points *inside* the circle (or under the arc) this expression is negative, and for points *outside* the circle (or above the arc) it is positive.

Let's assume that we have just finished drawing pixel (x_p, y_p), and we want to select the next pixel to draw (drawing clockwise around the boundary). Since the slope of the circular arc is between 0 and -1, our choice at each step our choice is between the neighbor to the east E and the neighbor to the southeast SE. If the circle passes above the midpoint M between these pixels, then we go to E next, otherwise we go to SE.

Next, we need a decision variable. We take this to be the value of $F(M)$, which is

$$\begin{aligned} D &= F(M) = F(x_p + 1, y_p - \frac{1}{2}) \\ &= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2. \end{aligned}$$

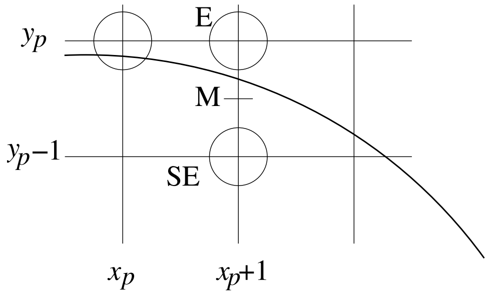


Figure 80: Midpoint algorithm for circles.

If $D < 0$ then M is *below* the arc, and so the E pixel is closer to the line. On the other hand, if $D \geq 0$ then M is *above* the arc, so the SE pixel is closer to the line.

Again, the new value of D will depend on our choice.

We go to E next: Then the next midpoint will have coordinates $(x_p + 2, y_p - (1/2))$ and hence the new d value will be

$$\begin{aligned} D_{new} &= F(x_p + 2, y_p - \frac{1}{2}) \\ &= (x_p + 2)^2 + (y_p - \frac{1}{2})^2 - R^2 \\ &= (x_p^2 + 4x_p + 4) + (y_p - \frac{1}{2})^2 - R^2 \\ &= (x_p^2 + 2x_p + 1) + (2x_p + 3) + (y_p - \frac{1}{2})^2 - R^2 \\ &= (x_p + 1)^2 + (2x_p + 3) + (y_p - \frac{1}{2})^2 - R^2 \\ &= D + (2x_p + 3). \end{aligned}$$

Thus, the new value of D will just be the current value plus $2x_p + 3$.

We go to NE next: Then the next midpoint will have coordinates $(x_p + 2, Sy_p - 1 - (1/2))$ and hence the new D value will be

$$\begin{aligned} D_{new} &= F(x_p + 2, y_p - \frac{3}{2}) \\ &= (x_p + 2)^2 + (y_p - \frac{3}{2})^2 - R^2 \\ &= (x_p^2 + 4x_p + 4) + (y_p^2 - 3y_p + \frac{9}{4}) - R^2 \\ &= (x_p^2 + 2x_p + 1) + (2x_p + 3) + (y_p^2 - y_p + \frac{1}{4}) + (-2y_p + \frac{8}{4}) - R^2 \\ &= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2 + (2x_p + 3) + (-2y_p + 2) \\ &= D + (2x_p - 2y_p + 5) \end{aligned}$$

Thus the new value of D will just be the current value plus $2(x_p - y_p) + 5$.

The last issue is computing the initial value of D. Since we start at $x = 0$, $y = R$ the first midpoint of interest is at $x = 1$, $y = R - 1/2$, so the initial value of D is

$$\begin{aligned} D_{init} &= F(1, R - \frac{1}{2}) \\ &= 1 + (R - \frac{1}{2})^2 - R^2 \\ &= 1 + R^2 - R + \frac{1}{4} - R^2 \\ &= \frac{5}{4} - R. \end{aligned}$$

This is something of a pain, because we have been trying to avoid floating point arithmetic. However, there is a very clever observation that can be made at this point. We are only interested in testing whether D is positive or negative. Whenever we change the value of D, we do so by a integer increment. Thus, D is always of the form $D' + 1/4$, where D' is an integer. Such a quantity is positive if and only if D' is positive. Therefore, we can just ignore this extra 1/4 term. So, we initialize $D_{init} = 1 - R$ (subtracting off exactly 1/4), and the algorithm behaves *exactly* as it would otherwise!

Lecture X05: Cohen-Sutherland Line Clipping

(Supplemantal)

Cohen-Sutherland Line Clipper: Let us consider the problem of clipping a line segment with endpoint coordinates $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$, against a rectangle whose top, bottom, left and right sides are given by WT, WB, WL and WR, respectively. We will present an algorithm called the *Cohen-Sutherland* clipping algorithm. The basic idea behind almost all clipping algorithms is that it is often the case that many line segments require only very simple analysis to determine either than they are entirely visible or entirely invisible. If either of these tests fail, then we need to invoke a more complex intersection algorithm.

To test whether a line segment is entirely visible or invisible, we use the following (imperfect but efficient) heuristic. Let be the endpoints of the line segment to be clipped. We compute a 4 bit code for each of the endpoints P_0 and P_1. The code of a point (x, y) is defined as follows.

Bit 1: 1 if point is above window, i.e. $y > WT$.

Bit 2: 1 if point is below window, i.e. $y < WB$.

Bit 3: 1 if point is right of window, i.e. $x > WR$.

Bit 4: 1 if point is left of window, i.e. $x < WL$.

This subdivides the plane into 9 regions based on the values of these codes. See the figure.

Now, observe that a line segment is entirely visible if and only if both of the code values of its endpoints are equal to zero. That is, if $C_0 \vee C_1 = 0$ then the line segment is visible and we draw it. If both line segments lie entirely above, entirely below, entirely right or entirely left of the window then the segment can be rejected as completely invisible. In other words, if $C_0 \wedge C_1 \neq 0$ then we can discard this segment as invisible. Note that it is possible for a line to be invisible and still pass this test, but we don't care, since that is a little extra work we will have to do to determine that it is invisible.

Otherwise we have to actually clip the line segment. We know that one of the code values must be nonzero, let's assume that it is (x_0, x_1). (Otherwise swap the two endpoints.) Now,

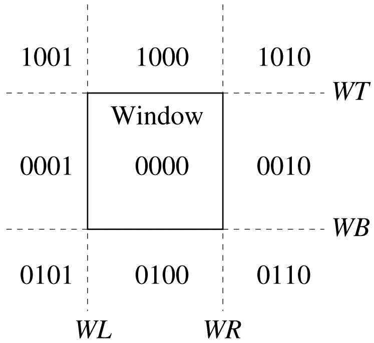


Figure 81: Cohen-Sutherland region codes.

we know that some code bit is nonzero, let's try them all. Suppose that it is bit 4, implying that $x_0 < WL$. We can infer that $x_1 \geq WL$ for otherwise we would have already rejected the segment as invisible. Thus we want to determine the point (x_c, y_c) at which this segment crosses WL. Clearly

$$x_c = WL,$$

and using similar triangles we can see that

$$\frac{y_c - y_0}{y_1 - y_0} = \frac{WL - x_0}{x_1 - x_0}.$$

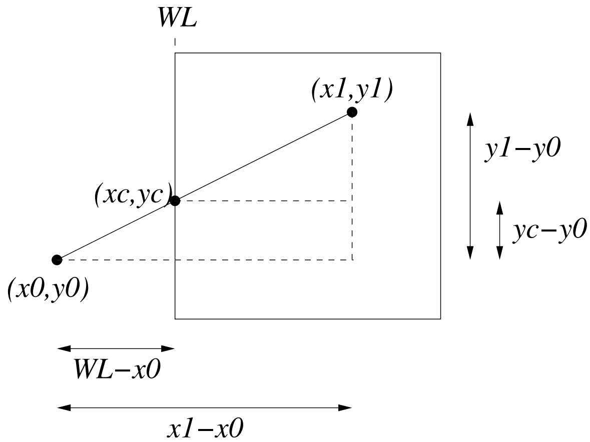


Figure 82: Clipping on left side of window.

From this we can solve for y_c giving

$$y_c = \frac{WL - x_0}{x_1 - x_0}(y_1 - y_0) + y_0.$$

Thus, we replace (x_0, y_0) with (x_c, y_c), recompute the code values, and continue. This is repeated until the line is trivially accepted (all code bits = 0) or until the line is completely rejected. We can do the same for each of the other cases.

Lecture X06: Halftone Approximation

(Supplemental)

Read: Chapter 10 in Hill.

Halftone Approximation: Not all graphics devices provide a continuous range of intensities. Instead they provide a discrete set of choices. The most extreme case is that of a monochrom display with only two colors, black and white. Inexpensive monitors have look-up tables (LUT's) with only 256 different colors at a time. Also, when images are compressed, e.g. as in the gif format, it is common to reduce from 24-bit color to 8-bit color. The question is, how can we use a small number of available colors or shades to produce the perception of many colors or shades? This problem is called *halftone approximation*.

We will consider the problem with respect to monochrome case, but the generalization to colors is possible, for example by treating the RGB components as separate monochrome subproblems.

Newspapers handle this in reproducing photographs by varying the dot-size. Large black dots for dark areas and small black dots for white areas. However, on a CRT we do not have this option. The simplest alternative is just to round the desired intensity to the nearest available gray-scale. However, this produces very poor results for a monochrome display because all the darker regions of the image are mapped to black and all the lighter regions are mapped to white.

One approach, called *dithering*, is based on the idea of grouping pixels into groups, e.g. 3×3 or 4×4 groups, and assigning the pixels of the group to achieve a certain affect. For example, suppose we want to achieve 5 halftones. We could do this with a 2×2 dither matrix.

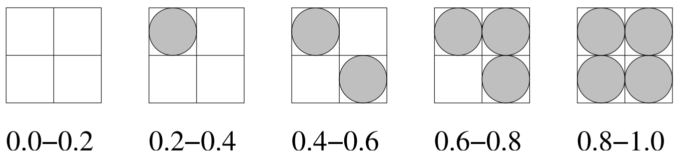


Figure 83: Halftone approximation with dither patterns.

This method assumes that our displayed image will be twice as large as the original image, since each pixel is represented by a 2×2 array. (Actually, there are ways to adjust dithering so it works with images of the same size, but the visual effects are not as good as the error-diffusion method below.)

If the image and display sizes are the same, the most popular method for halftone approximation is called *error diffusion*. Here is the idea. When we approximate the intensity of a pixel, we generate some approximation error. If we create the same error at every pixel (as can happen with dithering) then the overall image will suffer. We should keep track of these errors, and use later pixels to correct for them.

Consider for example, that we a drawing a 1-dimensional image with a constant gray tone of $1/3$ on a black and white display. We would round the first pixel to 0 (black), and incur an error of $+1/3$. The next pixel will have gray tone $1/3$ which we add the previous error of $1/3$ to get $2/3$. We round this to the next pixel value of 1 (white). The new accumulated error is $-1/3$. We add this to the next pixel to get 0, which we draw as 0 (black), and the final error is 0. After this the process repeats. Thus, to achieve a $1/3$ tone, we generate the pattern $010010010010\ldots$, as desired.

We can apply this to 2-dimensional images as well, but we should spread the errors out in both dimensions. Nearby pixels should be given most of the error and further away pixels be given less. Furthermore, it is advantageous to distribute the errors in a somewhat random way to avoid annoying visual effects (such as diagonal lines or unusual bit patterns). The Floyd-Steinberg method distributed errors as follows. Let (x, y) denote the current pixel.

Right: 7/16 of the error to $(x + 1, y)$.

Below left: 3/16 of the error to $(x - 1, y - 1)$.

Below: 5/16 of the error to $(x, y - 1)$.

Below right: 1/16 of the error to $(x + 1, y - 1)$.

Thus, let $S[x][y]$ denote the shade of pixel (x, y). To draw $S[x][y]$ we round it to the nearest available shade K and set $err = S[x][y] - K$. Then we compensate by adjusting the surrounding shades, e.g. $S[x + 1][y] += (7/16)err$.

There is no strong mathematical explanation (that I know of) for these magic constants. Experience shows that this produces fairly good results without annoying artifacts. The disadvantages of the Floyd-Steinberg method is that it is a serial algorithm (thus it is not possible to determine the intensity of a single pixel in isolation), and that the error diffusion can sometimes general "ghost" features at slight displacements from the original.

The Floyd-Steinberg idea can be generalized to colored images as well. Rather than thinking of shades as simple scalar values, let's think of them as vectors in a 3-dimensional RGB space. First, a set of *representative colors* is chosen from the image (either from a fixed color palette set, or by inspection of the image for common trends). These can be viewed as a set of, say 256, points in this 3-dimensional space. Next each pixel is "rounded" to the nearest representative color. This is done by defining a distance function in 3-dimensional RGB space and finding the nearest neighbor among the representatives points. The difference of the pixel and its representative is a 3-dimensional error vector which is then propagated to neighboring pixels as in the 2-dimensional case.

Lecture X07: 3-d Rotation and Quaternions

(Supplemental)

Read: This material is covered only briefly in Foley (Sect. 21.1.3). My presentation is coming mostly from the book "Advanced Animation and Rendering Techniques" by A. Watt and M. Watt (1992).

Rotation and Orientation in 3-Space: One of the trickier problems 3-d geometry is that of parameterizing rotations and the orientation of frames. We have introduced the notion of orientation before (e.g., clockwise or counterclockwise). Here we mean the term in a somewhat different sense, as a directional position in space. Describing and managing rotations in 3-space is a very difficult task (at least conceptually), compared with the relative simplicity of rotations in the plane.

Why do we care about rotations? Suppose that you are an animation programmer for a computer graphics studio. The object that you are animating is to be moved smoothly from one location to another. If the object is in the same directional orientation before and after, we can just translate from one location to the other. If not, we need to find a way of interpolating between its two orientations. This usually involves rotations in 3-space. But how should these rotations be performed so that the animation looks natural? Another example is one in which the world is stationary, but the camera is moving from one location and viewing situation to another. Again, how can we move smoothly and naturally from one to the other?

Since smoothly interpolating positions by translation is pretty easy to understand, let us ignore the issue of position, and just focus on orientations and rotations about the origin. Let F denote

Figure 84: Floyd-Steinberg Algorithm (Source: Poulbére and Bousquet, 1999).

the standard coordinate frame, and consider another orthonormal frames G. We want some way to represent G concisely, relative to F. Furthermore, given two such orthonormal frames, G and H, we would like a way to interpolate smoothly between these two (say if we want to produce a smooth animation from one to the other). We could just represent G and H by their orthonormal basis vectors. But if we were to interpolate (linearly) between corresponding pairs of basis vectors, the intermediate vectors would not necessarily be orthonormal. We will explore two methods for dealing with rotation, *Euler angles* and *quaternions*.

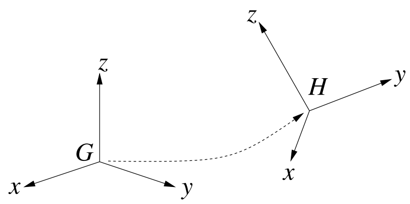


Figure 85: Smooth Interpolation of Frames.

Euler Angles: Euler was a famous mathematician who lived in the 18th century. He proved many important theorems, among which is one that states that the composition any number of rotations in three-space is just a single rotation in 3-space (about an appropriately chosen vector). Euler also showed that any rotation in 3-space could be broken down into exactly three rotations, one about each of the coordinate axes. These are sometimes called *rolls*.

For example, consider the orthonormal frame G, described earlier. Suppose that we want to rotate the standard frame F so that it coincides with G. Let us consider the process in reverse. We will see how to rotate G so that it coincides with the standard frame. Perhaps the easiest way to see this is to consider the three rotations that bring G into alignment with F, and then reverse these rotations.

This process is easier to describe than it is to visualize. Suppose that we label G's basis vectors $\vec{u}$, $\vec{v}$ and $\vec{w}$. Let $\vec{w}'$ denote the projection of w onto the yz-coordinate plane. First rotate about the x-axis by some angle θ_x, until the vector $\vec{w}'$ coincides with the z-axis. (See part (a) of the figure below.) The original vector $\vec{w}$ will now lie on the xz-coordinate plane. Next, rotate about the y-axis (thus keeping the xz-coordinate plane fixed) until $\vec{w}$ coincides with the z-axis. (See part (b) of the figure.) Call this angle θ_y. When this happens, the two vectors $\vec{u}$ and $\vec{v}$ (being orthogonal to $\vec{w}$) must lie on the xy-plane. Finally, we rotate about the z-axis until $\vec{u}$ coincides with the x-axis. (See part (c) of the figure.) Call this angle θ_z. Assuming that G is orthonormal and right-handed, it follows that $\vec{v}$ will coincide with the y-axis. Thus, by these three rotations, one about each of the axes, we can bring G into alignment with F.

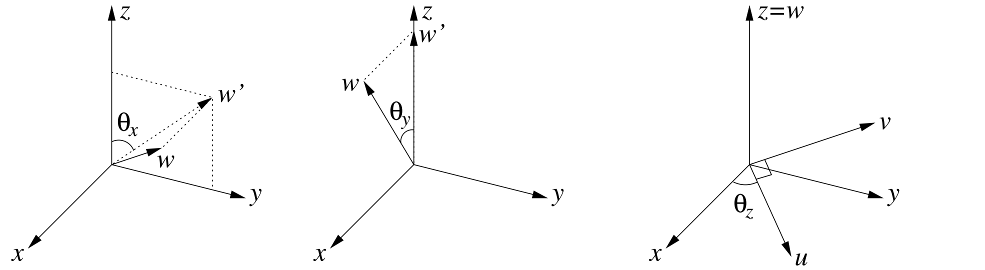
It follows that it is possible to perform any change of orientation with three rotations, one about each of the coordinate axes, by a triple of three angles, $(\theta_x, \theta_y, \theta_z)$. These define a general rotation matrix, by composing the three basic rotations:

$$R(\theta_x, \theta_y, \theta_z) = R_z(\theta_z)R_y(\theta_y)R_x(\theta_x).$$

These three angles are called the *Euler angles* for the rotation. Thus, we can parameterize *any* rotation in 3-space as triple of numbers, each in the range $[0, 2\pi]$.

Now, given two orientations in space, say given by the Euler angles $\Theta = (\theta_x, \theta_y, \theta_z)$ and $\Phi = (\phi_x, \phi_y, \phi_z)$, we can interpolate between then, say by taking convex combinations. Given any $\alpha \in [0, 1]$, we can define

$$R(\alpha) = R((1 - \alpha)\Theta + \alpha\Phi),$$



(a) Rotate about x so that w lies in xz–plane.

(b) Rotate about y so that w coincides with z.

(c) Rotate about z so that u and v coincide with x and y.

Figure 86: Rotating a frame to coincide with the standard frame.

for example. As α varies from 0 to 1, this will smoothly rotate from one orientation to the other.

There are some problems with Euler angles. The major problem is the fact that this representation depends on the choice of coordinate system. In the plane, a 30 degree rotation is the same, no matter what direction the axes are pointing (as long as they are orthonormal and right-handed). However, the result of an Euler angle orientation depends very much on where the frame is, and even the order in which you name the axes. It would be nice to have a method of describing rotations that is independent of the choice of coordinate systems. This is what we consider next time.

Angular Displacement: Last time we discussed Euler angles as a means of expressing general rotations in 3-space. However, we noted that the result of an Euler angle transformation depends on the positioning of the axes, and on the order in which the axes are labeled. Today we discuss an approach to rotation that is invariant under rigid changes of the coordinate system.

Perhaps a somewhat more natural way to express rotations (about the origin) in 3-space is in terms of two quantities, $(\theta, \vec{u})$, consisting of an angle θ, and an axis of rotation $\vec{u}$. Let's consider how we might do this. First consider a vector $\vec{v}$ to be rotated. Let us assume that $\vec{u}$ is of unit length.

Our goal is to describe the image of $\vec{v}$ under this rotation as a function of θ and $\vec{u}$. Let $R(\vec{v})$ denote this image. In order to derive this, we begin by decomposing $\vec{v}$ as the sum of its components that are parallel to and orthogonal to $\vec{u}$, respectively.

$$\vec{v}_{\|} = (\vec{u} \cdot \vec{v})\vec{u} \qquad \vec{v}_{\perp} = \vec{v} - \vec{v}_{\|} = \vec{v} - (\vec{u} \cdot \vec{v})\vec{u}.$$

Note that $\vec{v}_{\|}$ is unaffected by the rotation, but $\vec{v}_{\perp}$ is rotated to a new position $R(\vec{v}_{\perp})$. To determine this rotated position, we will first construct a vector that is orthogonal to $\vec{v}_{\perp}$ lying in the plane of rotation.

$$\vec{w} = \vec{u} \times \vec{v}_{\perp} = \vec{u} \times (\vec{v} - \vec{v}_{\|}) = (\vec{u} \times \vec{v}) - (\vec{u} \times \vec{v}_{\|}) = \vec{u} \times \vec{v}.$$

The last step follows from the fact that $\vec{u}$ and $\vec{v}_{\|}$ are parallel, and so the cross product is zero. Clearly $\vec{w}$ is orthogonal to both $\vec{v}_{\perp}$ and $\vec{u}$. Furthermore, because $\vec{v}_{\perp}$ is orthogonal to the unit vector $\vec{u}$, it follows that $\vec{w}$ is of the same length as $\vec{v}_{\perp}$.

Now, consider the plane spanned by $\vec{v}_{\perp}$ and $\vec{w}$. We have

$$R(\vec{v}_{\perp}) = (\cos\theta)\vec{v}_{\perp} + (\sin\theta)\vec{w}.$$

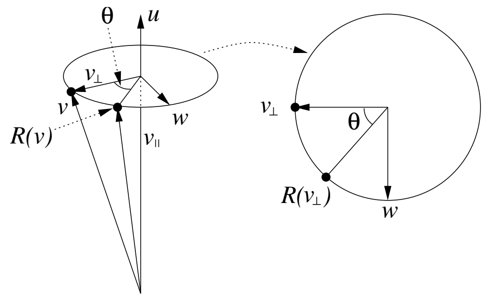


Figure 87: Angular displacement.

From this we have

$$\begin{aligned}R(\vec{v}) &= R(\vec{v}_{\|}) + R(\vec{v}_{\perp}) \\ &= R(\vec{v}_{\|}) + (\cos\theta)\vec{v}_{\perp} + (\sin\theta)\vec{w} \\ &= (\vec{u}\cdot\vec{v})\vec{u} + (\cos\theta)(\vec{v} - (\vec{u}\cdot\vec{v})\vec{u}) + (\sin\theta)\vec{w} \\ &= (\cos\theta)\vec{v} + (1-\cos\theta)\vec{u}(\vec{u}\cdot\vec{v}) + (\sin\theta)(\vec{u}\times\vec{v}).\end{aligned}$$

This last expression is the image of $\vec{v}$ under the rotation. Notice that, unlike Euler angles, this is expressed entirely in terms of geometric quantities, which do not depend on the choice of coordinates. This is an advantage over Euler angles. But it is rather hard to handle in this form.

Quaternions: We will now delve into a subject, which at first may seem quite unrelated. But keep the above expression in mind, since it will reappear in most surprising way.

This story begins in the early 19th century, when the great mathematician Hamilton was searching for a generalization of the complex number system. Imaginary numbers can be thought of as linear combinations of two basis elements, 1 and i, which satisfy the multiplication rules $1^2 = 1$, $i^2 = -1$ and $1 \cdot i = i \cdot 1 = i$. (The interpretation of $i = \sqrt{-1}$ arises from the second rule.) Hamilton searched for a generalization involving two imaginary basis values, i and j, but couldn't make it work. After many years he hit upon the trick, which was to consider three imaginary values i, j, and k, which behave as follows:

$$i^2 = j^2 = k^2 = -1 \qquad ij = k,\ jk = i,\ ki = j.$$

Combining these, it follows that $ji = -k$, $kj = -i$ and $ik = -j$. A *quaternion* is defined to be a generalized complex number of the form

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k.$$

We will see that quaternions bear a striking resemblance to our notation for angular displacement. In particular, we can rewrite the quaternion notation in terms of a scalar and vector as

$$\mathbf{q} = (s, \vec{u}) = s + u_x i + u_y j + u_z k.$$

Given the rules above for multiplication, it is easy to derive the rules for multiplying quaternions.

$$\mathbf{q}_1\mathbf{q}_2 = (s_1 s_2 - (\vec{u}_1 \cdot \vec{u}_2), s_1\vec{u}_2 + s_2\vec{u}_1 + \vec{u}_1 \times \vec{u}_2).$$

(We will leave this derivation as an exercise. If you ignore the cross-product term, this bears a striking superficial resemblance to the rule for complex multiplication.) Quaternion multiplication is associative but not commutative.

Define the *conjugate* of a quaternion $\mathbf{q} = (s, \vec{u})$ to be

$$\overline{\mathbf{q}} = (s, -\vec{u}).$$

It is easy to show that the product of a quaternion and its conjugate has a zero vector component, and hence may be thought of as a scalar. Define the *magnitude* of a quaternion to be the square root of this product

$$|\mathbf{q}|^2 = \mathbf{q}\overline{\mathbf{q}} = s^2 + |\vec{u}|^2.$$

(Notice that we are abusing notation a bit here.) A *unit quaternion* is one of unit magnitude, $|\mathbf{q}| = 1$. A *pure quaternion* is one with a 0 scalar component

$$\mathbf{p} = (0, \vec{v}).$$

Any quaternion of nonzero magnitude has a multiplicative *inverse*, which is

$$\mathbf{q}^{-1} = \frac{1}{|\mathbf{q}|^2}\overline{q}.$$

(Try multiplying $\mathbf{q}\mathbf{q}^{-1}$ to see why this is so.) Observe that if $\mathbf{q}$ is a unit quaternion, then $\mathbf{q}^{-1} = \overline{\mathbf{q}}$.

Quaternions and Rotation: What do quaternions have to do with rotation in 3-space? We will represent a rotation by a unit quaternion $\mathbf{q}$ (we will see exactly how later). Given any point $P = (v_x, v_y, v_z)$ in 3-space, we will represent it by mapping it to a pure quaternion, $\mathbf{p} = (0, \vec{v})$. The image of $\mathbf{p}$ under the rotation $\mathbf{q}$ will again be a pure quaternion (and so is easy to map back to a point in 3-space).

Define the *rotation operator*

$$R_{\mathbf{q}}(\mathbf{p}) = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}.$$

By applying the multiplication rule, and using the fact that $\mathbf{q}^{-1} = \overline{\mathbf{q}}$ for unit quaternions, it is easy to derive that

$$R_{\mathbf{q}}(\mathbf{p}) = (0,\ (s^2 - (\vec{u}\cdot\vec{u}))\vec{v} + 2\vec{u}(\vec{u}\cdot\vec{v}) + 2s(\vec{u}\times\vec{v})).$$

(Again, we leave the derivation as an exercise.)

So what does this have to do with rotation? Let us see if we can express this in a more suggestive form. Since $\mathbf{q}$ is of unit magnitude, we can express it as

$$\mathbf{q} = (\cos\theta, (\sin\theta)\vec{u}), \qquad \text{where } |\vec{u}| = 1.$$

Plugging this into the above expression, we have

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= (0,\ (\cos^2\theta - \sin^2\theta)\vec{v} + 2(\sin^2\theta)\vec{u}(\vec{u}\cdot\vec{v}) + 2\cos\theta\sin\theta(\vec{u}\times\vec{v})) \\ &= (0,\ (\cos 2\theta)\vec{v} + (1-\cos 2\theta)\vec{u}(\vec{u}\cdot\vec{v}) + \sin 2\theta(\vec{u}\times\vec{v})). \end{aligned}$$

Now, recall the rotation displacement equation presented earlier. The vector part of this quaternion is identical, except that we have 2θ in place of θ.

Thus, in summary, we encode points in 3-space as pure quaternions

$$\mathbf{p} = (0, \vec{v}),$$

and we encode a rotation by angle θ about a unit vector $\vec{u}$ as a unit quaternion

$$\mathbf{q} = (\cos(\theta/2), \sin(\theta/2)\vec{u}),$$

then the image of the point under this rotation is given by the vector part of the result of the quaternion rotation operator $R_{\mathbf{q}}(\mathbf{p})$.

For example, consider the 3-d rotation shown in the figure below. This rotation can be achieved by performing a rotation about the y-axis by -90 degrees. Thus $\theta = -\pi/2$, and $\vec{u} = (0,1,0)$. Thus the quaternion that encodes this rotation is

$$\mathbf{q} = (\cos(-\pi/4), \sin(-\pi/4)(0,1,0)) = \left(\frac{1}{\sqrt{2}}, \left(0, -\frac{1}{\sqrt{2}}, 0\right)\right).$$

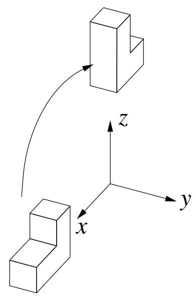


Figure 88: Rotation example.

The image of the point P on the tip of the x-unit vector with homogeneous coordinates $(1,0,0,1)$, comes by representing P by a vector $\vec{v} = (1,0,0)$, and then encoding $\vec{v}$ as a pure quaternion $\mathbf{p} = (0,(1,0,0))$. Then we apply the rotation operator

$$\begin{aligned}
R_{\mathbf{q}}(\mathbf{p}) &= (0,\ (1/2 - 1/2)(1,0,0) + 2(0,1,0)0 + (2/\sqrt{2})((0,-1/\sqrt{2},0) \times (1,0,0))) \\
&= (0,(0,0,0) + (0,0,0) + (-1)(0,0,-1)) \\
&= (0,(0,0,1)).
\end{aligned}$$

Thus P is mapped to a point on the z-axis, as expected.

Composing Rotations: We have shown that each unit quaternion corresponds to a rotation in 3-space. This is an elegant representation, but can we manipulate rotations through quaternion operations? The answer is yes. In particular, the action of multiplying two unit quaternions results in another unit quaternion. Furthermore, the resulting product quaternion corresponds to the composition of the two rotations. In particular, given two unit quaternions $\mathbf{q}$ and $\mathbf{q}'$, a rotation by $\mathbf{q}$ followed by a rotation by $\mathbf{q}'$ is equivalent to a single rotation by the product $\mathbf{q}'' = \mathbf{q}'\mathbf{q}$. That is,

$$R_{\mathbf{q}'} R_{\mathbf{q}} = R_{\mathbf{q}''} \qquad \text{where } \mathbf{q}'' = \mathbf{q}'\mathbf{q}.$$

This follows from the associativity of quaternion multiplication, and the fact that $(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}^{-1}\mathbf{q}'^{-1}$, as shown below.

$$\begin{aligned}
R_{\mathbf{q}'}(R_{\mathbf{q}}(\mathbf{p})) &= \mathbf{q}'(\mathbf{q}\mathbf{p}\mathbf{q}^{-1})\mathbf{q}'^{-1} \\
&= (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}^{-1}\mathbf{q}'^{-1}) \\
&= (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}\mathbf{q}')^{-1} \\
&= \mathbf{q}''\mathbf{p}\mathbf{q}''^{-1} \\
&= R_{\mathbf{q}''}(\mathbf{p}).
\end{aligned}$$

Matrices and Quaternions: Quaternions provide a very elegant way of representing rotations in 3-space. Returning to the problem of interpolating smoothly between two orientations, we can see that we can describe the before and after orientations of any object by two quaternions, $\mathbf{q}$ and $\mathbf{p}$. Then, to interpolate smoothly between these two orientations, we just interpolate between $\mathbf{q}$ and $\mathbf{p}$ in quaternion space. This is not really a linear interpolation, because the quaterions must be of unit length. It is more like interpolating between two points on the surface of a sphere.

However, once we have a quaternion representation, we need a way to inform our (quaternion challenged?) graphics API (like OpenGL) about the actual transformation. In particular, given a unit quaternion

$$\mathbf{q} \;=\; (\cos(\theta/2), \sin(\theta/2)\vec{u}) \;=\; (w, (x, y, z)),$$

what is the corresponding affine transformation (expressed as a rotation matrix). By simply expanding the definition of of $R_{\mathbf{q}}(\mathbf{p})$, it is not hard to show that the following (homogeneous) matrix is equivalent

$$\begin{pmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy & 0 \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx & 0 \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Thus, given your quaternion interpolant, you generate this matrix, and invoke `glLoadMatrix()`, and all subsequently drawn points will be rotated in accordance with the quaternion.

Lecture X08: Radiosity

(Supplemental)
Read: Section 16.13 in Foley.

Radiosity: The whole philosophy of our previous lectures on illumination were based on what we called "quick-and-dirty" methods: efficient approaches that manage to "fool the eye". This philosophy represents the more applied branch of computer graphics, and we have seen that very realistic images can be produced in this way, under various assumptions about lighting and reflection (e.g. that light sources are points, that there is no indirect illumination, that surfaces are Lambertian, etc.) In particular, the idea that illumination is a *local* phenomenon is central to the efficiency of these approaches. This means that the illumination of a point depends only on that point, and its relationship to a small number of point light sources. The more theoretical branch of graphics insists that to achieve realism without these assumptions, a *global* illumination model must be considered. Such an illumination model would take into account the fact that light is not just coming from a few point light sources, but that light is arriving indirectly from many different directions.

Consider the following example. Imagine a room with white walls. Shine a single spot light upwards from the floor onto one of the walls. Is the wall the only object in the room to be lit? No, the ceiling will be illuminated indirectly from this light, and the light from the ceiling will dimly illuminate much of the rest of the room. A ray traced solution will only find the direct illumination of the light on the wall. The other indirect illuminations could only be modeled using ambient light. Furthermore, if the wall that the light hits is colored red, then the indirect illumination will have a red cast to it, even though the light source is white. Ray tracing cannot account for this phenomenon.

What are the elements of a global illumination model? The basic idea is that rather than viewing the world as a small set of light sources and a large number of nonradiating objects, we think of each object as being a potential light source. Some objects (light sources) radiate light directly, but others (nonblack surfaces) can radiate light indirectly. The illumination at any point generally depends on *all* of the objects in the environment (at least those that are visible from this object), and furthermore the object has an influence on the illumination of *all* the other objects. *Radiosity* is an example of a global illumination model.

Radiosity Overview: Before plunging into the details of the radiosity equations, let us first consider the problem at a high level. For each point on the surface of some object in our environment, we want to know the intensity of this point, how bright it appears. This intensity (called the *radiosity*) of the point P is a function of (1) the emittance of light from this point (if it is a light source), and (2) the reflection of light coming from other surfaces in the environment. The second component is quite complicated, because it depends on the radiosity of points on surfaces throughout the environment, whether these points are visible from P, and how reflective the surface is that P lies on.

Sampling: As one might imagine, radiosity computations are quite expensive from a computational standpoint. The idea is that for every point in the environment, we need to know the illumination of all the surface elements that this point can see. Thus, it is as if we are solving a hidden surface removal problem for *every* point in the environment. Of course, there are infinitely many points in the environment, so to make this computation tractable, it is common to solve this hidden surface problem for some sampled points in the environment.

How are these points selected? There are a few approaches. The most common is based on a generalization of the *finite element method.* We subdivide each of the object surfaces into a number of small polygonal patches. Such a subdivision is called a *surface mesh.* For each patch, we will compute an approximation of the radiosity of this patch. For example, this could be done by computing the radiosities at each of its vertices and then averaging these.

How do you construct these patches? This is a hard question. If you make patches very small you get good accuracy, but the execution time will be slow. If you make the patches large, you gain speed, but at the cost of accuracy. The best methods use an *adaptive* approach. First start with a coarse mesh, determining in which areas the radiosity is varying most rapidly, and then refining these areas and trying again. When the radiosity values are fairly constant in the neighborhood of a patch of the mesh, or when the patches are deemed to be "small enough" then we do not need to refine further. The figure below gives an example using a quadtree-based meshing algorithm where a shadow has caused a sudden change in radiosity.

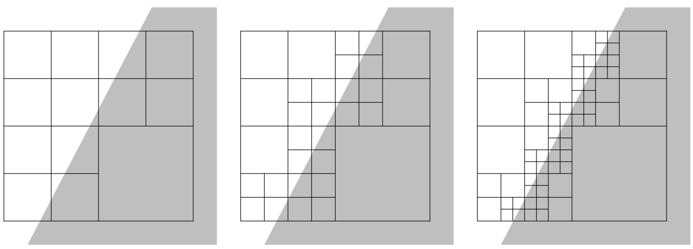


Figure 89: Adaptive Meshing for Radiosity.

More sophisticated methods, like *discontinuity meshing* actually attempt to align the edges of the mesh with sharp changes in radiosity (e.g. as happens along the edge of a shadow).

Who comes first? So let us just assume that we have a set of points at which we will perform radiosity computations. How do we compute the radiosity of a single point? There is a real problem here. The radiosity at point A depends on the radiosity from all visible points B. But similarly the radiosity at B depends on the radiosity at the visible point A. How can we compute radiosities, when each seems to depend on all the others?

We will see that there are two general approaches. One is based on defining a large linear system of equations, that "encodes" all of the radiosity dependencies. By solving this equation, we can determine all the radiosities at all the points. The problem is that the size of this linear equation is enormous. If you have n surface patches in your mesh, the matrix storing the equation is of size $n^2 \times n^2$. This is because each variable in the equation involves the transmission of light between two surface patches.

The other method is based on the idea of starting with the brightest light source and shooting its radiation around to the entire scene. Next we move to the next brightest light source and repeat this process. Note that as we do this, surfaces that were initially black start picking up more and more intensity. Eventually a nonemitting light source can start accumulating more and more intensity, until it becomes the brightest light source, and then it shoots its intensity to the surrounding scene. This is called *progressive refinement radiosity.*

Basics of Radiance: The most basic concept of radiosity is radiance. We define *radiance*, denoted L, as the amount of energy per unit time (or equivalently power) emitted from a point x in a given direction. We can define the direction relative to a surface by giving two angles θ the angle with respect to the surface normal, and ϕ the angle of the projection onto the surface. We will use ω to denote the resulting directional vector. Thus we will express radiance as either $L(x, \theta, \phi)$ or more succinctly as $L(x, \omega)$.

Since points and directions have no real size, we define radiance in terms of small differential quantities. We can think of a small differential area as a small square or circular patch on the surface of an object. We can think of a differential direction as a small *solid angle*, that is a small cone centered around this direction. Solid angles are measured in terms of radians squared, or *steradians.*

In computer graphics, since we are assuming that radiance arises from a surface, we will apply Lambert's law to take into account the fact that for a piece of surface, the amount of energy directed along the normal is greater than the amount of energy at a large angle to the normal. Thus energy will be scaled by the cosine of the angle with the normal. Thus the power radiating from a small patch in some small solid angle can be expressed as:

$$L(x, \theta, \phi) dx \cos\theta d\omega.$$

By the way, radiance is measured in watts per square meter per steradian.

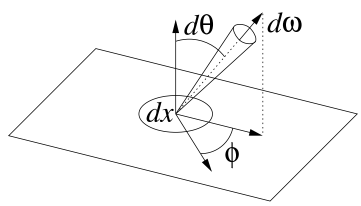


Figure 90: Radiance.

We would like to express $d\omega$ in terms of θ and ϕ. This is given by

$$d\omega = (\sin\theta) d\theta d\phi.$$

The reason for the $\sin\theta$ term is that since θ and ϕ are related to latitude and longitude, and θ becomes small, the longitude lines become closer and closer near the poles. This factor accounts for this.

Radiance is a directional quantity, in that it depends not only on position but also on direction. The *radiosity*, denoted B, of a point is defined to be the total power leaving a point on a surface per unit area of the surface (in all directions). We can define radiosity in terms of the more basic quantity radiance by integrating over the entire hemisphere lying above the surface. The surface of this hemisphere is denoted Ω. Thus we can define

$$B(x) = \int_\Omega L(x, \theta, \phi) \cos\theta d\omega.$$

Simple Radiosity Equation: The most general illumination models are based on the idea of a *bidirectional reflection function* which indicates how the strength and direction of the reflected radiance depends on the strength and direction incoming radiance (called *irradiance*). However, it will simplify things greatly to assume that all surfaces are Lambertian, that is, ideal diffuse reflectors. This has the advantage of eliminating all the directional elements of radiance.

If surfaces are Lambertian, then we can simplify $L(x, \theta, \phi)$ and just write $L(x)$. The radiosity at the point x is given by

$$\begin{aligned} B(x) &= \int_\Omega L(x, \theta, \phi) \cos\theta d\omega \\ &= L(x) \int_\Omega \cos\theta d\omega \\ &= L(x) \int_0^\pi \int_0^{2\pi} \cos\theta \sin\theta d\theta d\phi \\ &= \pi L(x). \end{aligned}$$

This means simply that radiosity depends only on the radiance, the light power, at the point.

At this point we can state the *radiosity equation* (for Lambertian reflectors). It states that the the radiosity of a point is equal to the amount of energy emitted from this point (this happens when the point lies on a light source) plus the total reflection of all incoming light. Let $\rho_d(x)$ denote the coefficient of diffuse reflection for the object at point x (earlier we had written this k_d). We can write this as

$$L(x) = L_e(x) + \frac{\rho_d(x)}{\pi} \int_\Omega L_i(x, \theta, \phi) \cos\theta d\omega.$$

where L_e denote emitted radiance and L_i denotes the incoming irradiance. We cannot eliminate the directional component from the L_i term, because we still need to consider Lambert's law for incoming radiation. If we define

$$H(x) = \int_\Omega L_i(x, \theta, \phi) \cos\theta d\omega.$$

and let $E(x)$ denote the emitted radiosity $\pi L_e(x)$, and recall that $B(x) = \pi L(x)$ then we can write this as

$$B(x) = E(x) + \rho_d(x) H(x).$$

The term $H(x)$ essentially describes how much illumination energy is arriving from all other points in the scene.

To simplify $H(x)$ we can use the Lambertian assumption. Rather than integrating over the angular space surrounding x, instead we will integrate over the set of points on all surfaces, denoted S. Let $y \in S$ be such a surface point visible from x in direction ω. Let θ' denote the angle between the surface normal at y and the line-of-sight vector from y to x $(-\omega)$, and let ϕ' be defined similar to ϕ but for y. Let r denote the distance from x to y.

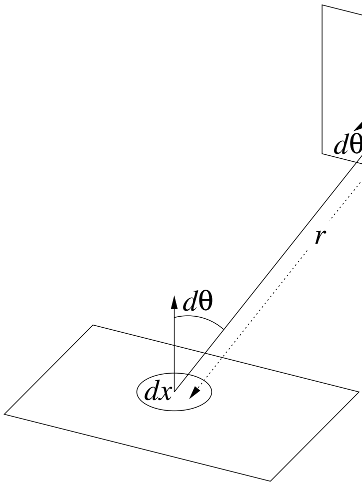


Figure 91: Illumination from another surface.

First we observe that by symmetry of radiance (the energy sent from y to x equals the energy received from x to y), we have $L(x, \theta, \phi) = L(y, \theta', \phi')$. Since we assume that all surfaces are Lambertian, we have

$$L(y, \theta', \phi') = \frac{B(y)}{\pi}.$$

We can express the differential angle $d\omega$ in terms of a differential area of the surface near y as

$$d\omega = \frac{\cos\theta' \, dy}{r^2}.$$

(The r^2 term arises because as we move further away the differential angle sweeps out a larger area, and the cosine term arises because as y's surface is slanted with respect to the line of sight, we sweep out a larger area.)

Because we will integrate over all visible elements, we will include a *visibility function* $V(x, y) = 1$ if x can see y and 0 otherwise. Putting these together, we can now define $H(x)$ in terms of an integral over surface points:

$$H(x) = \int_{y \in S} B(y) \frac{\cos\theta \cos\theta'}{\pi r^2} V(x, y) dy.$$

Form Factors: In practice, we cannot expect to be able to solve this integral equation. As mentioned before, most radiosity methods are based on subdividing space into small patches, and assuming that the radiosity is constant for each patch. Thus, in the equation for $H(x)$ above, we can assume that $B(y)$ is constant for all points y in a surface patch.

For each pair of patches, P_i and P_j in our mesh, we define the *form factor* $F_{i,j}$ to be the fraction of light energy leaving P_i that arrives at patch P_j.

$$F_{i,j} = \frac{1}{A_j} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta \cos\theta'}{\pi r^2} V(x, y) dy dx.$$

$F_{i,j}$ is a dimensionless quantity. If patches are close, large, and facing one another, Fi, j will be large.

From this we can rewrite the radiosity equation as a system of linear equations:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{j,i} \frac{A_j}{A_i}.$$

Here B_i is the radiosity of patch i (the amount of light reflected per unit area), E_i is the amount of light emitted from this patch per unit area, ρ_i is the reflectivity of patch i ($\rho \approx 0$ means a dark nonreflecting object and $\rho \approx 1$ means a bright highly reflecting object). Finally A_i and A_j are the areas of patches P_i and P_j, respectively. These terms are necessary because all quantities were stated in terms of energy per unit area, so larger patches contribute more energy.

Observe that this equation is circular, because it defines the light leaving a patch i in terms of the light leaving a patch j, but the light leaving patch j may be affected by light leaving a patch i. The way to think about this equation is a set of linear constraints (in the variables B_i) which must all be satisfied simultaneously. In other words, this is a system of linear equations. Unlike the simple systems of 2 or 3 equations we have seen so far, this is a system of perhaps thousands of equations, since we have one equation for each surface patch (and that equation contains thousands of terms). Clearly solving such a system is a task requiring heavy computational resources. Fortunately, most small patches have virtually no effect on distant patches, so the linear system is sparse. Iterative techniques from numerical analysis, such as Gauss-Seidel, can be used to solve this type of system.

To make the connection with linear systems clearer, observe that since we assume that light can travel equally well in any direction it follows that

$$A_i F_{i,j} = A_j F_{j,i}.$$

We can simplify the above equation as

$$\begin{aligned} B_i &= E_i + \rho_i \sum_{j=1}^{n} B_j F_{i,j} \\ E_i &= B_i - \rho_i \sum_{j=1}^{n} B_j F_{i,j} \end{aligned}$$

which can be written in matrix form as

$$\begin{pmatrix} 1-\rho_1 F_{1,1} & -\rho_1 F_{1,2} & \cdots & -\rho_1 F_{1,n} \\ -\rho_2 F_{2,1} & 1-\rho_2 F_{2,2} & \cdots & -\rho_2 F_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ -\rho_n F_{n,1} & -\rho_n F_{n,2} & \cdots & 1-\rho_n F_{n,n} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}.$$

The values ρ_i are dependent on the surface types. The hard thing to compute are the values of $F_{i,j}$. It can be shown that there is a fairly simple geometric interpretation of $F_{i,j}$. We break the i-th patch into small differential elements. For each element we consider a hemisphere surrounding this element, and project patch j onto this hemisphere through its center. We then project this projection orthographically onto the base circle of the hemisphere. The value of $F_{i,j}$ is the area of this projection, divided by the area of the circle. Thus intuitively patches that occupy a larger field of view contribute more to $F_{i,j}$ and patches that are more nearly orthogonal to the surface contribute more.

Computing this orthogonal projection of a spherical projection is somewhat tricky (considering that it must be repeated for every tiny element of every patch), so it is important to speed this computation up, at the cost of the introduction of approximation errors. We can approximate the hemisphere by a hemicube, and discretize the surface of the hemicube into square (pixel-like) elements. We project all the surrounding patches onto each of the faces of the hemicube. (Note that this is essentially a visible surface elimination task, which can be solved with hardware assistance, e.g. using a z-buffer algorithm.) Each cell of the hemicube is now associated with a patch, and we apply a weighting factor that depends on the square of the hemicube, and sum these up.

Needless to say, this process is extremely computationally intensive. We are basically solving a visible surface determination problem at *every* point on the surface of our objects. Much of the research in radiosity is devoted to mechanisms to save computations, without sacrificing realism.