CMSC 427/828M: Computer Graphics[1]
Spring 1997
Dave Mount

Lecture 1: Course Introduction

(Thursday, Jan 30, 1997)
Review of syllabus. No lecture today.

Lecture 2: Graphics Systems and Models

(Tuesday, Feb 4, 1997)
Read: Chapter 1 in Angel.

Computer Graphics: Computer graphics is concerned with producing images using a computer. 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.

Our approach to teaching computer graphics will be based on a *top-down approach*. In a traditional *bottom-up approach* we would begin by discussing primitive 2-dimensional graphics from the level of illuminating individual pixels on a raster graphics devices, and work up successively to complex 3-dimensional graphics. Instead a top-down approach begins by asking the question of how to model 3-dimensional graphics at a very high level, and then later considering what sorts of low-level tools will be needed to implement this model.

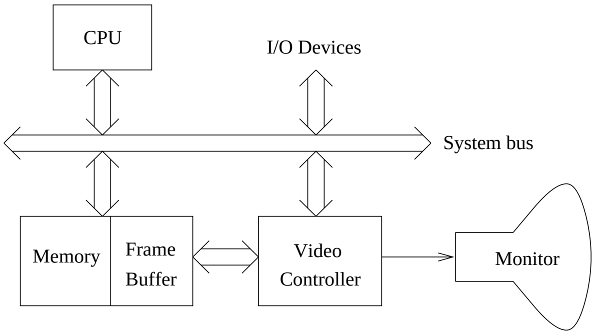
In this first lecture, we begin with the basics of 3-dimensional graphics: graphics display systems and basic models for 3-d graphics.

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 dots, called *pixels*. The screen has 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.

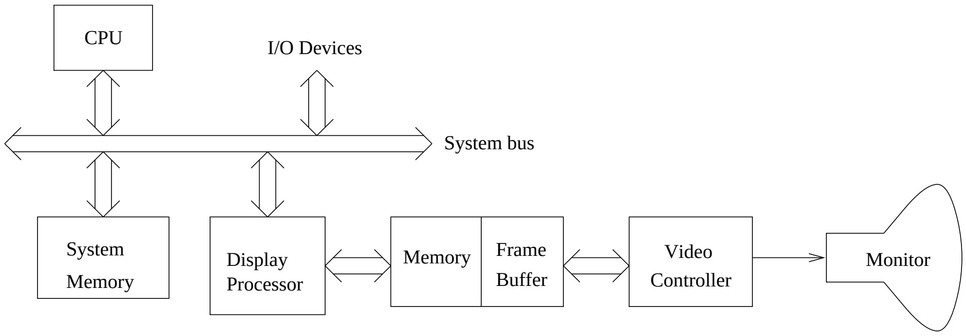
In *color* raster devices each pixel is a composite of (typically) three components: red, green, and blue, that are combined in varying proportions to achieve different colors (and there are three electron beams). The number of bits used for each pixel is called the *depth* of the display. Simple color graphics systems, like the ones we will be using will have typically 8-bit depth. Higher-end graphics systems will have at least 24-bit depth, and often much more. 8-bit depth is quite limiting, since it implies there are only 256 different colors that can be displayed at any one time. We will discuss how it is that these systems still manage to produce realistic looking images.

The image is stored as a two-dimensional array of pixel values (called a *frame buffer*), and the display hardware produces the image line-by-line (called *raster lines*). For each pixel on each line the intensity of the pixel is varied by varying the intensity of the electron beam as it scans across the line. Since the phosphor only holds its color for a short period of time, the display must be continuously updated (from 30 to 60 times a second) to avoid a "flickering" affect.

A hardware device called a *video controller* reads the frame buffer and produces the image on the display. An example of this type of configuration is shown below.



Simple Raster Graphics System



Raster Graphics with Display Processor

Figure 1: Raster display architectures.

The frame buffer is not a special device. It is simply a chunk of the computer's 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. The high-level process of taking a graphics model and producing an image is called *rendering*. The lower-level process of drawing individual lines or filling polygonal regions of the image by modifying the frame buffer is called *scan conversion* or *rasterization*.

Complex graphics systems achieve great speed by providing separate hardware support, in the form of a *display processor*, to relieve the computer's main processor from much of the mundane effort involved in maintaining the frame buffer. A good 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 transformation.

Rasterization: Involves a number of tasks related to mapping a polygon to a set of illuminated pixels. This involves determining which pixels overlap the polygonal region, providing shading and color, and perhaps performing hidden surface removal and other advanced operations.

These operations are typically *pipelined*, where each processor on the pipeline performs its task and passes the results to the next phase.

Color: As mentioned earlier, color is achieved by having three electron guns mixing the colors red, green, and blue. When all are illuminated fully you get white, and when all are off you get black. You can get arbitrary colors by mixing these (although it is not usually easy to figure out the proper proportions to achieve a certain color).

The intensity of each beam is specified by a number that is stored in the frame buffer for each pixel. Typically each color is specified by an 8-bit value, requiring a total of $8 \cdot 3 = 24$ bits to represent each pixel. 24-bit color provides more varieties of color than the typical human eye can distinguish, and so this is felt to be adequate. This is sometimes refered to as *true color*.

Because storing 24 bits of information for each pixel of a (say 1000×1000) eats up a fair chunk of memory (3 megabytes), many PC's and low-end graphics workstations use a more economical approach. They use only 8 bits per pixel. Each 8-bit entry is an index into a 256 entry array called a *colormap*. Each entry in the colormap is a 24-bit value, containing the R, G, and B components of the color. This is called *indexed color* or *pseudo-color*.

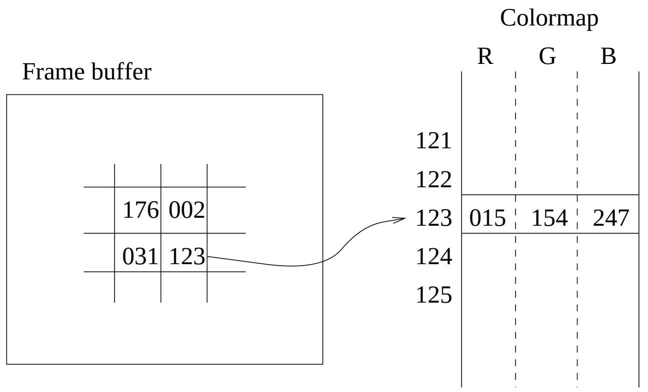


Figure 2: Colormaps and indexed color.

For most simple uses (such as window interfaces) 256 colors at any one time is quite adequate. However, when displaying images with lots of smooth shading, 256 is woefully small. In this case systems resort to methods of approximating colors. We will discuss these later. As memory costs fall, in the not too distant future, all frame buffers will likely support at least 24 bits for color.

Window Systems: Once the eventual widespread availability of graphics workstations became apparent, the idea of graphics-based user interfaces quickly surfaced. One of the earliest such systems was Xerox Star's window-based workstation environment. This system introduced the idea of windows and menus.

This idea was quickly adopted and extended by Apple in their Macintosh machines. Many concepts that are familiar in most present-day window systems (icons for example) were introduced here. These concepts were then adapted by others, including Microsoft and X, which we now consider the further detail.

X Window System: Recognizing the need for platform-independent window systems, a consortium of universities and corporations came up with the X window system, which is what we see primarily on the Unix systems around campus. We will not describe X in great detail, but we will present an overview of some of the important ideas of X (and some of these common to many window systems).

X divides the display into rectangular regions called *windows*. 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.

Client and Server: One of X's major contributions is that machines running programs can display their output on displays (supporting X-windows) without even knowing what the display hardware is. The application program that is generating graphics is called the *client*. It may be (and typically is) the same machine that the display is associated with. However, it may also be any machine that is networked to the machine with the display. The software that controls the display is called the *server*. The client and server communicate through the network, with the client sending requests (e.g. create a window, draw a line, draw some text) and the server responding to these requests and sending back information such as information about the display and windows (e.g. its size and whether it has color) and events (e.g. whether the mouse button was pressed). Data structures may be shared between the client and server to avoid the need to constantly send information back and forth through the network.

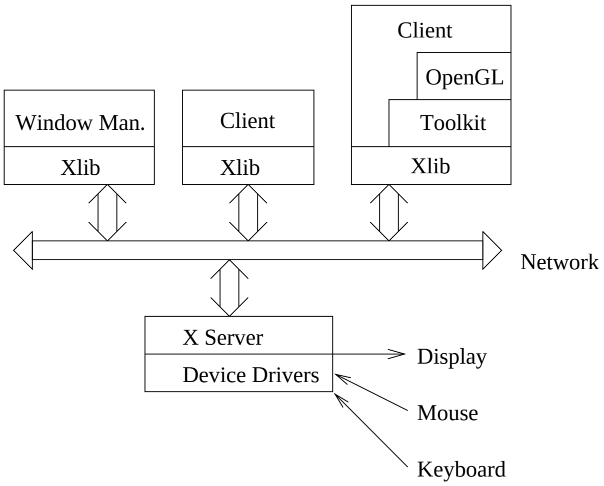


Figure 3: X-windows client-server structure.

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, expose new windows, iconify windows, you are interacting with a program called a *window manager*. The window manager (e.g. `twm`) is just an application written in X. It's only real "privilege" with respect to the X system is that only one can be running at once, and whenever a new window is opened up, the window-manager is informed, so it can decide where the window should be placed.

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.

Event driven programming: 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, or expose events from X).

This can be handled in a number of ways. One is by *polling* the various input devices. This involves setting up an infinite loop that repeatedly queries each device. However it is quite wasteful of computational resources. Instead, the typical approach is 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 that it is interested.

In an animation, the user may not be providing any input at all. In these cases the program can register for a *timer event*. For example, the program may request that it be "wakened-up" by an event every 1/50 of a second, say, to update the next image of the animation.

Lecture 3: More on Graphics Systems and Models

(Thursday, Feb 6, 1997)
Read: Chapter 1 in Angel.

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 at least two 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), and the light sources. These are all elements that are independent of the location of the viewer.

Viewer: A description of the location of the viewer and the properties of the synthetic camera (its direction, field of view, projection properties.

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 (e.g. 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 programs. As mentioned above, such a model should provide information about geometry, color, texture, and reflective properties for these objects. Because these models are mathematically based, synthetic objects (cubes, spheres, cylinders) are relatively easy to model, whereas natural objects (trees, mountains, skin, hair, clouds, water) are much harder. Much of the effort in computer graphics research over the last ten years has dealt with developing realistic models for natural objects.

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). 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.

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.* To speed up the rendering process, many graphics systems provide routines to break complex polygons down into a collection of triangles. This process is called *triangulation* or *tesselation.*

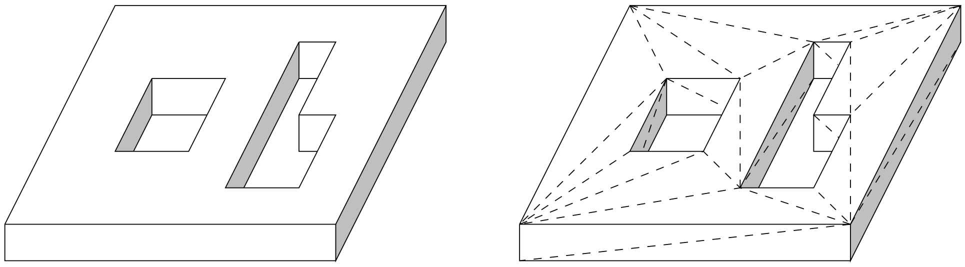


Figure 4: A polyhedral model and its tesselation.

In some polyhedral models full *topological* information will be provided, indicating the incidences between vertices, edges, and faces. In graphics systems it often suffices just to specify the faces in terms of a list of vertices. This is the approach used in OpenGL.

In addition to specifying geometry, we also need to specify color, texture, surface finish, etc in order to complete the picture. We will discuss these aspects of the model later.

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. First, is the light source essentially a point (like the sun) or does it cover some area (like a florescent light buld). This affects things like the sharpness of shadows in the final image.

The next is the color of the light. Some light sources (such as incandescent bulbs) produce light with a high degree of red color whereas others (florescent bulbs) produce proportionately less. What we see in an object is not its color, but the color of the light that is reflected from that object toward our eye. If an object reflects only blue light, but the light sources emit only red light, then that object will appear to be black.

The light that is emitted from a light source is a complex spectrum of electromagnetic energy (over the visible spectrum). However to simplify things, we will simply model it as some combination of red, green and blue color components. Once we have done this, we can further think of each color component separately, and assume that our light sources are *monochromatic* (single colored). After computing each of these monochromatic components separately, we combine the three results to get the final RGB values.

To produce the most realistic images a good renderer needs to model as much of the interplay and interreflection of light in a scene. Light is emitted from the light source, strikes various objects in the scene, being absorbed all or partially by some, bouncing off of others at various angles depending on their surface characteristics. What we see is a function of those light rays that eventually enter our eyes. However only a very tiny fraction of the emitted light does eventually reach our eyes.

Lighting models based on simulating this complex interaction of light and objects are called *global illumination models*. Although these models result in the most realistic images, they are very complex to compute. Consequently most commercial interactive graphics systems compute much simpler *local illumination models* where we make the simplifying assumption that the light rays that are emitted from an object come directly from the light sources.

Lighting Models: Just how light reflects from a surface is a very complex phenomenon, depending on the surface qualities and microscopic structure of object's surface. We will simplify the modeling of light reflection by considering two extremes. The first is *diffuse reflection* in which light is assumed to be scattered equally in all directions. The second is *specular reflection* in which light is assumed to be reflected primarily according to Snell's law (angle of incidence equals angle of refraction). The diffuse reflection model works well for nonshiny objects like cotton fabric or typing paper. The specular reflection model works well for shiny objects such as metallic surfaces. The compromise model is to assume that each object has some degree of diffuse reflection and some degree of specular reflection. By varying the proportions of these two aspects, it is possible to mimic the surface properties of many different objects.

Even with these two components of light reflection, there is still something that is unaccounted for. The space underneath a table may be concealed from all light sources, and yet it is not entirely black. The reason is that it is illuminated indirectly from other surfaces. However, local illumination models do not allow for the possibility of indirect reflection. Consequently, another source of light called *ambient light* is usually added to these models. Ambient light just refers to the amount nondirectional light that is present at every point in a scene. Ambient light is like random "noise". It is uniformly present everywhere and has no directional components.

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 model has been constructed, the next aspect to specifying the image is to specify the synthetic camera that will be taking a picture of the scene. 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.

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

$$x_p = -\frac{x}{z/d}.$$

Thus once we have transformed our points into this particular coordinate system, computing a perspective transformation is a relatively simple operation.

$$(x, y, z) \Rightarrow \left(-\frac{y}{z/d}, -\frac{x}{z/d}, -d\right).$$

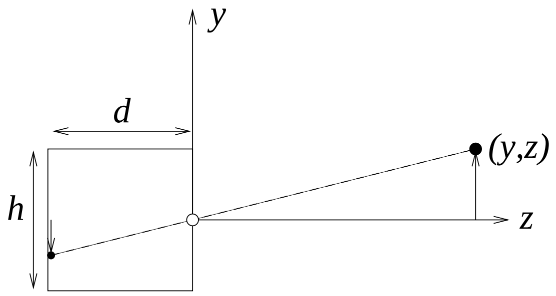


Figure 5: Pinhole camera.

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 see that a point will only be visible if it lies within a cone centered at the origin with angle θ where

$$\theta = 2\tan^{-1}\frac{h}{2d}.$$

This angle is called the *field of view* (for y). A similar formula holds for x.

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.

Finally, observe that this transformation was based on the assumption that objects have been represented using this camera-centered coordinate system. In general the camera moves around, and so this is not a convenient method for describing the location of objects. It will be necessary for the system to apply a transformation to map coordinates in the world coordinate system into the camera coordinate system.

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).

Lecture 4: Introduction to OpenGL

(Tuesday, Feb 11, 1997)
Read: Chapter 2 in Angel. This lecture was presented by the TA.

OpenGL: OpenGL is a general purpose application programmer's interface (API) for computer graphics. It main features are the facts that it is portable, flexible, and designed with efficient implementation on typical graphics workstations in mind. When we speak of OpenGL, we will actually be referring to three separate (but related) libraries: basic OpenGL (GL), the OpenGL utilities (GLU), and the OpenGL utility toolkit (GLUT).

This is not reference material, but hopefully it will provide enough information to point you in the right direction for reference material. For detailed reference information on GL and GLU consult the *OpenGL Reference Manual* (2nd edition) and for GLUT consult *OpenGL: Programming for the X Window System*, by M. J. Kilgard. Rather than buying these books, this information is available through the web from the following sites (the first for GL and GLU, and the second for GLUT):

```
http://www.digital.com:80/pub/doc/opengl/
http://www.sgi.com/Technology/OpenGL/glut3doc/spec3.html
```

Links to these sites have bee set up on the class web page.

Important information: Here is some important information to know before reading the subsequent sections.

Generic Function Naming: Many OpenGL procedures come in a variety of forms, depending on the argument types. For example, `glVertex3f()` takes 3 float arguments, `glVertex4d()` takes 4 double arguments, and `glVertex3iv()` takes a single pointer to an array of 3 integers. These procedures are lumped under the single name `glVertex*()` in the description below. See the reference manual for the actual calling sequences.

GL Data Types: OpenGL defines a number of data types. Most of these translate into their expected types in C or C++, for example, `GLdouble` is the same as `double`, and `GLuint` is an `unsigned int`. The type `GLClampf` is a `float` that has been "clamped" to the interval $[0, 1]$, meaning that values less than 0 are set to 0 and values greater than 1 are set to 1. See the file `Mesa/include/GL/gl.h` for exact type definitions.

RGBA: OpenGL colors are typically defined using RGB (red, green, blue) components and a special A (or alpha) component The A component has varying interpretations depending on context. For example, it can be used to blend colors. When no special effects are desired, set $A = 1$.

Errors: OpenGL and Mesa are "quiet" about errors. When an error has been committed, the system does not display any error message. Rather it sets an error code, which the user can query through `glGetError()`. It is the programmer's responsibility to check the error code after each call to an OpenGL procedure. However, if you do not want to be troubled by this, set the environment variable `MESA_DEBUG`. This will cause Mesa to output a message whenever an error is detected (and produces a few annoying warning messages that you may not care about). This can be done by adding the command "`setenv MESA_DEBUG`" to your login initialization file.

Initializations: The following procedures are typically invoked once in the main program, before the graphics processing begins.

`glutInit(int *argcp, char **argv):`
Initialize GLUT and OpenGL. Pass in command-line arguments.

`glutInitDisplayMode(unsigned int mode):`
Set GLUT's display mode. The mode is the logical-or "|" of one or more of the following:

`GLUT_RGBA`	use RGBA colors
`GLUT_INDEX`	use color-mapped colors (not recommended)
`GLUT_SINGLE`	use single-buffering
`GLUT_DOUBLE`	allow double-buffering
`GLUT_DEPTH`	allow depth-buffering
`GLUT_ACCUM`	allow accumulation buffer
`GLUT_ALPHA`	allow alpha (A) components of colors
`GLUT_STENCIL`	allow stencil buffer

`glutInitWindowSize(int width, int height):`
Set the window size. Also see: `glutInitWindowPosition()`.

`glutCreateWindow(char *title):`
Create window with given title (typically argv[0]).

`glutDisplayFunc(void (*)(void)):`
Call the given function to redisplay everything (required).

`glutReshapeFunc(void (*)(int width, int height)):`
Call the given function when window is resized (and originally created). See the function `glViewport()`, which typically should be called after each resizing.

`glutIdleFunc(void (*)(void)):`
Call the given function with each refresh cycle of the display.

`glutMouseFunc(void (*)(int button, int state, int x, int y)):`
Call the given function when a mouse button is clicked. The `button` argument is any of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHT_BUTTON`, and the `state` is either `GLUT_DOWN` or `GLUT_UP`.

`glutMotionFunc(void (*)(int x, int y)):`
Call the given function when the mouse is dragged (with button held down). The arguments (x, y) give the mouse coordinates relative to the upper right corner of the window.

`glutPassiveMotionFunc(void (*)(int x, int y)):`
Call the given function when the mouse is moved.

`glutKeyboardFunc(void (*)(unsigned char key, int x, int y)):`
Call the given function when a keyboard key is hit.

`glutMainLoop(void):`
This starts the main event loop. Control returns only through one of the above callback functions.

Miscellaneous display control functions: Here are some miscellaneous routines that affect how drawing and displaying take place.

`glutPostRedisplay(void):`
Request that the image be redrawn (e.g. after resize).

`glutSwapBuffers(void):`
Swap the front and back buffers (used in double buffering).

`glClearColor(GLclampf R, GLclampf G, GLclampf B, GLclampf A):`
Specifies the color to be used to erase the image before each new drawing sequence is initiated.

glClear(GLbitfield mask):
Clears one or more of the existing buffers. The mask is a logical or ("|") of any of GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_ACCUM_BUFFER_BIT, or GL_STENCIL_BUFFER_BIT.

glFlush(void):
OpenGL normally saves or "buffers" drawing commands for greater efficiency. If you want the display to be updated this instant, this procedure forces drawing to commence.

glFinish(void):
This is like `glFlush()`, but it waits until all drawing has been finished before it returns.

glHint(GLenum target, GLenum mode):
Provides a hint as to how OpenGL should behave. Normally this is to suggest whether speed or accuracy is more important. The `mode` may be GL_DONT_CARE, GL_FASTEST, or GL_NICEST. Some `target` values include

GL_FOG_HINT	accuracy of fog computation
GL_PERSPECTIVE_CORRECTION_HINT	accuracy of interpolation in texture mapping

OpenGL Basic Drawing: With the exception of `glRect()`, most drawing is done by creating geometric objects using `glBegin()...glEnd()` combination, where between each such pair you specify a collection of vertices that define the object. Along with specifying vertices, you can specify a number of attributes that affect how the vertices will appear. These include things such as color, surface normals (for shading), and texture coordinates (for texture mapping).

glRect*(...):
Draw a rectangle on the $z = 0$ plane. The arguments give the (x, y) coordinates of the lower left and upper right corners of the rectangle.

glBegin(GLenum mode)...glEnd(void):
Define an object, where mode may be any of:

GL_POINTS	set of points
GL_LINES	set of line segments (not connected)
GL_LINE_STRIP	chain of connected line segments
GL_LINE_LOOP	closed polygon (may self intersect)
GL_TRIANGLES	set of triangles (not connected)
GL_TRIANGLE_STRIP	linear chain of triangles
GL_TRIANGLE_FAN	fan of triangles (joined to one point)
GL_QUADS	set of quadrangles (not connected)
GL_QUAD_STRIP	linear chain of quadrangles
GL_POLYGON	a convex polygon

There are a limited set of commands that can be issued between `glBegin()..glEnd()` pairs. These include the following: `glVertex*()`, `glColor*()`, `glNormal*()`, `glTexCoord*()`, `glMaterial*()`, and `glCallList()`. See the reference manual for the exact specification of vertex order and orientation.

glVertex*(...):
Specify the coordinates of a vertex.

glNormal*(...):
Specify the surface normal for subsequent vertices. The normal should be of unit length *after* the modelview transformation is applied. Call `glEnable(GL_NORMALIZE)` to let OpenGL take care of this automatically.

`glColor*(...)`:
Specify the color of subsequent vertices. This is normally used if lighting is *not* enabled. Otherwise, colors are defined by `glMaterial*()`, defined below under lighting.

GLUT utilities for complex objects: GLUT provides a number of procedures for drawing complex objects. These objects can be further modified by setting transformations for translating, rotating, scaling, and so on.

`glutBitmapCharacter(void *font, int character)`:
Draw a given character of a given `font` at the current raster position and advance the current raster position. Fonts include `GLUT_BITMAP_9_BY_15`, `GLUT_BITMAP_8_BY_13`, `GLUT_BITMAP_TIMES_ROMAN_10`, `GLUT_BITMAP_TIMES_ROMAN_24`. See `glRasterPos*()` to set the current raster position.

`glRasterPos*(...)`:
Set the current raster position (in 3D window coordinates) for `glutBitmapCharacter()` and some other pixel copying operations.

`glutSolidSphere(GLdouble radius, GLint slices, GLint stacks)`:
Draw a solid sphere of the given `radius`, subdivided into `slices` longitudinal strips, and `stacks` latitudinal strips. Normal vectors are also defined for proper illumination. Other shapes that are supported include `glutSolidCone()`, `glutSolidCube()`, `glutSolidTorus()`, `glutSolidDodecahedron()`, `glutSolidTeapot()`, `glutSolidOctahedron()`, `glutSolidTetrahedron()`, and `glutSolidIcosahedron()`. All of these also have wire-frame versions that draw only the edges of the shape.

Display lists: Because of its structure, OpenGL requires lots of procedure calls to render a complex scene (defining vertices, multiplying matrices, assigning normal vectors). This in turn generates lots of overhead for procedure calls, multiplying matrices, computing lighting parameters, and so on. To improve efficiency, OpenGL allows you to generate an object, called a *display list*, into which OpenGL commands are stored in an efficient internal format, which can be called back again in the future.

`GLuint glGenLists(GLsizei range)`:
Allocates *range* new display lists. It returns a number n, where the new lists are numbered $n, n+1, \ldots, n + range - 1$.

`glNewList(GLuint list, GLenum mode)...glEndList(void)`:
All OpenGL commands between such a pair are stored in display list `list`. The mode is either `GL_COMPILE`, to compile the list, or `GL_COMPILE_AND_EXECUTE` to both compile and execute. Note that some OpenGL commands (mostly those having to do with accessing the current state) are executed immediately and not compiled. See the reference manual for details.

`glCallList(GLuint list)`:
Invokes display list `list`. Note that the system does not make any effort to save its current state, thus the display list should make use of functions such as `glPushMatrix()` and `glPopMatrix()` to save and restore the current state.

Lecture 5: More on OpenGL

(Thursday, Feb 13, 1997)
Read: Chapter 3 in Angel. This lecture was presented by the TA.

Transformations and Perspective: OpenGL provides three different matrices: modelview, perspective, and texture (used in texture mapping). Actually, these matrices are maintained in three stacks. This allows you to push (save) the existing matrix on the stack, apply some new local transformations, draw some vertices under this transformation, and then pop the stack thus restoring the existing transformation. At any time one of the three matrices is active and is set by `glMatrixMode()`. Note that the transformation that has been multiplied last is the one that is applied first to the vertices.

`glMatrixMode(GLenum mode):`
Set the current matrix mode to one of `GL_MODELVIEW`, `GL_PROJECTION`, or `GL_TEXTURE`. The default is `GL_MODELVIEW`.

`glLoadIdentity(void):`
Set the current matrix to the identity.

`glPushMatrix(void):`
Make a copy of the current matrix and push it on the stack.

`glPopMatrix(void):`
Pop the top of the current matrix.

`glLoadMatrixd(const GLdouble *M):`

`glLoadMatrixf(const GLfloat *M):`
Sets the current matrix to `M`. Beware: Matrices are stored in column-major order, not row-major order as in most programming languages.

`glMultMatrixd(const GLdouble *M):`

`glMultMatrixf(const GLfloat *M):`
Postmultiply the given matrix M by the current matrix C, yielding $C \cdot M$ as the new current matrix.

`glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z):`

`glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z):`
Generate a rotation matrix about the specified vector and postmultiply it by the current matrix.

`glScaled(GLdouble x, GLdouble y, GLdouble z):`

`glScalef(GLfloat x, GLfloat y, GLfloat z):`
Generate a scale matrix and postmultiply it by the current matrix.

`glTranslated(GLdouble x, GLdouble y, GLdouble z):`

`glTranslatef(GLfloat x, GLfloat y, GLfloat z):`
Generate a translation matrix and postmultiply it by the current matrix.

`gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz):`
All arguments are of type `GLdouble`. Generates a matrix with the eye at the specified location, looking at the given center point, with the given up directional vector, and postmultiplies by the current (typically modelview) matrix.

`glOrtho(left, right, bottom, top, near, far):`
All arguments are of type `GLdouble`. Generates an orthogonal projection matrix and postmultiplies by the current (typically perspective) matrix. The parameters specify the bounds of the 3-d rectangular viewing volume.

`gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top):`
A two dimensional version of `glOrtho3d()`, for when the z-coordinate does not matter.

`glFrustum(left, right, bottom, top, near, far):`
All arguments are of type `GLdouble`. Generates a perspective projection matrix and postmultiplies by the current (typically perspective) matrix. The parameters specify the bounds of the viewing frustum. The arguments `near` and `far` are the distances to the near and far clipping planes along the $-z$-axis. The other arguments specify the bounds of the viewing rectangle on the near clipping plane.

`gluPerspective(fovy, aspect, zNear, zFar):`
All arguments are of type `GLdouble`. Generates a matrix with the given up-down field of view (in degrees), for a window with the given aspect ratio (width/height) and the given near and far clipping planes, and multiplies by the current (typically perspective) matrix.

`glViewport(GLint x, GLint y, GLsizei width, GLsizei height):`
Sets the current viewport, that is, the portion of the window to which everything will be clipped. Typically this is the entire window: `glViewport(0, 0, win_width, win_height)`, and hence this should be called whenever the window is reshaped.

Options that can be enables or disabled: OpenGL offers a number of capabilities that may be enabled or disabled.

`glEnable(GLenum cap):`

`glDisable(GLenum cap):`
Enable/Disable some option. Some common options include the following. By default, all are initially disabled.

`GL_CULL_FACE`	back-face culling
`GL_DEPTH_TEST`	depth-buffering
`GL_FOG`	fog
`GL_BLEND`	color blending
`GL_LIGHTING`	lighting
`GL_LIGHTi`	turn on/off ith light source (0-7)
`GL_NORMALIZE`	automatic normalization of normals
`GL_TEXTURE_2D`	texture mapping

Note that options may be enabled and disabled for individual polygons. For example, texture mapping may be turned on before drawing one polygon, and then turned off for others. Only this polygon will be texture mapped.

Lighting: In OpenGL there may be up to 8 light sources (numbered 0 through 7). If lighting is enabled (see `glEnable()`) then the shading of each object depends on which light sources are turned on (enabled) and the materials and surface normals of each of the objects in the scene. Note that when lighting is enabled, it is important that each vertex be associated with a proper normal vector (by calling `glNormal*()` prior to generating the vertex).

`glShadeModel(GLenum mode):`
The `mode` may be either `GL_FLAT` or `GL_SMOOTH`. In flat shading every point on a polygon is shaded according to its first vertex. In smooth shading the shading from each of the various vertices is interpolated.

`glLightModelf(GLenum pname, GLfloat param):`

`glLightModelfv(GLenum pname, const GLfloat *params):`
Defines lighting model parameters. The first version is for defining scalar parameters, and the second is for vector parameters. There is also an integer version. One important parameter is the global intensity of ambient light (independent of any light sources). It has the `pname` of `GL_LIGHT_MODEL_AMBIENT` and `params` is a pointer to an RGBA vector intensities.

glLightf(GLenum light, GLenum pname, GLfloat param):
glLightfv(GLenum light, GLenum pname, const GLfloat *params):
Defines parameters for a single light source. The first version is for defining scalar parameters, and the second is for vector parameters. There is also an integer version. The first argument is one of GL_LIGHT0 through GL_LIGHT7, indicating which light source this applies to. The argument pname gives one of the properties to be assigned. These include the following:

GL_POSITION	(x, y, z, w) of position of light
GL_EMISSION	RGBA of intensity of emitted light
GL_AMBIENT	RGBA of intensity of ambient light
GL_DIFFUSE	RGBA of intensity of diffuse light
GL_SPECULAR	RGBA of intensity of specular light

By default, illumination intensity does not decrease, or attenuate, with distance. In general, if d is the distance from the light source to the object, and the light source is not a point at infinity, then the intensity attenuation is given by $1/(a + bd + cd^2)$ where a, b, and c are specified by the following parameters:

GL_CONSTANT_ATTENUATION	a-coefficient
GL_LINEAR_ATTENUATION	b-coefficient
GL_QUADRATIC_ATTENUATION	c-coefficient.

Normally light sources send light uniformly in all directions. To define a spotlight, set the following parameters.

GL_SPOT_CUTOFF	maximum spread angle of spotlight
GL_SPOT_DIRECTION	(x, y, z, w) direction of spotlight
GL_SPOT_EXPONENT	exponent of spotlight distribution

glMaterialf(GLenum face, GLenum pname, GLfloat param):

glMaterialfv(GLenum face, GLenum pname, const GLfloat *params):
Defines surface material parameters for subsequently defined objects. The first version is for defining scalar parameters, and the second is for vector parameters. There is also an integer version. Polygonal objects in OpenGL have two sides. You can assign properties either to the front, back, or both sides. (The front side is the one from which the vertices appear in counterclockwise order.) The first argument indicates the side. The possible values are GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK. The second argument is the specific property. Possbilities include:

GL_AMBIENT	RGBA of the ambient coefficients
GL_DIFFUSE	RGBA of the diffuse coefficients
GL_SPECULAR	RGBA of the specular coefficients
GL_SHININESS	single number in the range $[0, 128]$ that indicates degree of shininess.

Blending and Fog: Blending and fog are two OpenGL capabilities that allow you to produce interesting lighting and coloring affects. When a pixel is to be drawn on the screen, it normally overwrites any existing pixel color. When blending is enabled (by calling glEnable(GL_BLEND)) then the new (source) pixel is blended with the existing (destination) pixel in the frame buffer.

glBlendFunc(GLenum sfactor, GLenum dfactor):
Determines how new pixel values are blended with existing values. The arguments indicate what factors are multiplied by the source and destination RGBA values, respectively, before they are added together. Some choices for the factors include the following, where the subscript s refers to source and d to destination.

GL_ZERO	$(0, 0, 0, 0)$
GL_ONE	$(1, 1, 1, 1)$
GL_SRC_COLOR	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)$
GL_DST_COLOR	(R_d, G_d, B_d, A_d)
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)$
GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$

Note that `SRC` factors may only be used for `sfactor` and `DST` factors may only be used for `dfactor`. For example, to overlay 3 semitransparent drawings, set the A components of the object's color to 0.3333, and draw each using `glBlendFunc(GL_SRC_ALPHA, GL_ONE)`.

Fog produces an effect whereby more distant objects are blended increasingly with a *fog color*, typically some shade of gray. It is enabled by calling `glEnable(GL_FOG)`.

`glFogf(GLenum pname, GLfloat param):`
`glFogfv(GLenum pname, const GLfloat *params):`
Specifies the parameters that define how fog is computed. The first version is for defining scalar parameters, and the second is for vector parameters. There is also an integer version. The fog mode determines how distances are attenuated according to the fog. We will only mention linear attenuation. See the reference manual for other ones.

GL_FOG_COLOR	RGBA of fog (default = $(0, 0, 0, 0)$)
GL_FOG_MODE	`GL_LINEAR` for linear attenuation
GL_FOG_START	depth at which attenuation starts
GL_FOG_END	depth beyond which there is only fog

Texture Mapping: Texture mapping is the process of taking an image, presented typically as a 2-dimensional array of RGB values and mapping it onto a polygon. Setting up texture mapping involves the following steps: define a texture by specifying the image and its format (through `glTexImage2d()`, and specify how object vertices correspond to points in the texture, and finally enabling texture mapping. Of course, first the texture must be input or generated by the program. OpenGL provides a wide variety of other features, but we will only summarize a few here, which are sufficient for handling a single 2-dimensional texture. To handle multiple textures, see the commands `glGenTextures()` and `glBindTexture()`.

`glTexImage2D(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, GLvoid *pixels)`: Defines a texture stored in the array `pixels`. The target is typically `GL_TEXTURE_2D` (unless you are using 1-dimensional texture mapping). The `level` is typically 0 (unless you are using multiresolution textures, called "mipmaps"). The next argument specifies the internal formal in which OpenGL will store the map (in Mesa the value `GL_RGB` is recommended). The width and height of the texture image are given next and must each be a power of 2 (plus 2b if b is the value of `border`). The value `border` indicates the width of the border and is either 0 or 1. (Borders are used for smoother blending near the boundary when one texture image is replicated many times to fill a large area. If this not a concern then set border to 0.) The last three arguments specify the actual texture. The format and type arguments specify how you store the texture image. Typical values for `format` are `GL_RGB` or `GL_RGBA` (meaning 3 or 4 color components are stored for each pixel) and values for type include `GL_UNSIGNED_BYTE`, in which each RGB color component is scaled to the range [0...255] and stored in a single byte, or `GL_FLOAT`, in which each RGB component is scaled to the interval $[0, 1]$ and stored in a single float. See the reference manual for complete information. The array containing the texture image (stored row by row, left to right, and top to bottom), which is pointed to by `pixels`.

`glTexEnvf(GLenum target, GLenum pname, GLfloat param):`
Specifies texture mapping environment parameters. The target must be `GL_TEXTURE_ENV`. The pname must be `GL_TEXTURE_ENV_MODE`. This determines how a color from the texture image is to be merged with an existing color on the surface of the polygon. The param may be any of the following:

`GL_MODULATE`	multiply color components together
`GL_BLEND`	linearly blend color components
`GL_DECAL`	use the texture color
`GL_REPLACE`	use the texture color

There are subtle differences between `GL_DECAL` and `GL_REPLACE` when different formats are used or when the A component of the RGBA color is not 1. See the reference manual. The default is `GL_MODULATE`.

`glTexParameterf(GLenum target, GLenum pname, GLfloat param):`
`glTexParameterfv(GLenum target, GLenum pname, const GLfloat *params):`
Specify how texture interpolation is to be performed. The first version is for defining scalar parameters, and the second is for vector parameters. There is also an integer version. Assuming 2-dimensional textures, the target is `GL_TEXTURE_2D`, the `pname` is either:

`GL_TEXTURE_MAG_FILTER`	magnification filter
`GL_TEXTURE_MIN_FILTER`	minification filter

Magnification is used when a pixel of the texture is smaller than the corresponding pixel of the screen onto which it is mapped and minification applies in the opposite case. Typical values are either

`GL_NEAREST`	take the nearest texture pixel
`GL_LINEAR`	take the weighted average of the 4 surrounding texture pixels

This procedure may also be invoked to specify other properties of texture mapping.

`glTexCoord*(...):`
Similar to `glNormal()` but specifies the texture coordinates of subsequently defined vertices for texture mapping. For a standard 2-dimensional textures, the texture coordinates are a pair (s, t) in the interval $[0, 1] \times [0, 1]$. The texture coordinate specifies the point on the image that are to be mapped to this vertex. OpenGL interpolates the mapping of intermediate points of the polygon.

Lecture 6: Geometry and Geometric Programming

(Tuesday, Feb 18, 1997)
Read: Chapt 4, and Appendix B in Angel.

Geometric Programming: Computer graphics (as well as many other areas such as computer-aided design, robotics, computer vision, geographic information systems) basically involves computation with geometric entities. In 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. Consider the following problems that might arise in a computer graphics application:

Ray tracing: 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? If we have a sphere instead of a cube, what is the lattitude and longitude of the point that the ray hits?

Hidden surfaces: Given two triangles and a view point, are both triangles entirely visible or does one obscure the other (either totally or partially)? Which triangle is closer? What portion of the hidden triangle is visible?

Sidedness: A triangle in 3-space defined by its three vertices defines a unique plane. Given a point q does it lie above, below, or on this plane? Any 4 points define a sphere in 3-space. Given a point q does it lie inside, outside, or on this sphere?

Change of coordinates: A cube is represented relative to a standard coordinate system. What are its coordinates relative to a coordinate system centered at the camera's location?

Given the basic analytic geometry and linear algebra that you (should) know already, you could probably answer each of these questions, although it might take a little time to come up with the necessary formulas. Basically, you would describe the basic geometric objects in terms of various coordinates and parameters, then set up the defining equations and relations from linear algebra or analytic geometry, and solve them. Unfortunately, the resulting code will be quite messy and very difficult for anyone else to understand or modify.

Instead we will try learn a more systematic way of performing geometric programming. This method is called a *coordinate-free programming* because it attempts to perform virtually all operations at a high purely geometric level (vectors, points, lines, coordinate frames, polygons) rather than at a lower analytic-geometry level (in terms of coordinates). The method is not purely coordinate-free, since we will need coordinates for raw input and graphics output, but the goal will be to implement a set of geometric abstract data types, and hide all the coordinate processing from the programmer.

Geometries: Before beginning we should discuss a little history. Geometry is one of the oldest (if not the oldest) branches of mathmatics. It's origin was in land surveying (hence its name: geo=earth, and metry=measure), which became an important problem when people started forming societies, farmland was broken up into parcels, and boundary disputes started to erupt.

It was Euclid in ancient Greece who came up with the idea of axiomatizing geometry, and thus turning a great deal of intuitive understanding into a formal mathematical system. Many centuries later, the idea of coordinatizing space using (x, y)-coordinates was developed by Descartes and others. This made the connections between geometry and algebra much closer.

Around the time of Gauss, there was a rather revolutionary shift in people's view of geometry. Up to this time the sense was that the familiar Euclidean was the only geometry. But there was one question that was unsolved for many years: could Euclid's fifth postulate (given a line and a point that is not on the line, there is exactly one line passing through the point that is parallel to the line) be proven from his other axioms. After many failed attempts to do so, it was discovered that the answer was in fact no, and in fact there existed perfectly consistent geometric systems (hyperbolic for example) in which this axiom fails. This leads to the viewpoint that there is not one "true" geometry, but that one is free to define geometry in any manner that happens to suit the current purpose. This gave rise to a number of different geometry systems, and these *noneuclidean* geometries turned out to be important to Einstein's theory of relativity. Around 100 years ago, Klein stated that basically different geometries can be characterized by a set of algebraic transformations, and what properties are preserved under these transformations. Looking at geometric systems in terms of their transformations has been central to the mathematical theory of geometry ever since.

We will discuss 4 basic types of geometry: linear algebra, affine geometry, Euclidean geometry, and projective geometry. The first three build upon one another and the main innovation over your 7th-grade knowledge of these fields will be the element of how coordinatize these systems. Projective geometry is the geometry that will be needed for perspective transformations. However, we will see that very strange things happen in projective spaces (e.g. familiar concepts such as left and right, clockwise and counterclockwise become impossible to define). Consequently, we will try to do most of our work within affine and Euclidean geometry, and only use projective geometry for perspective.

Affine Geometry: We begin with *affine geometry*, a subset of Euclidean geometry that deals essentially with the simple objects (vectors, points, lines, etc) and simple line-preserving transformations (such as translation, rotation, scaling). Affine geometry is a subset of Euclidean geometry mainly because of the absence of any formal notion of distances or angles.

The basic elements of affine spaces 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. The notion of point is the major difference between affine geometry and linear algebra. In linear algebra there are no points (only vectors).

We will use the following notational conventions. Points will be refered to with upper-case letters (e.g., P, Q, and R), vectors will be denotes with lower-case letters (e.g., u, v, and w) and sometimes 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 these will be translated to roman (e.g., a, b, c).

What operations can be performed on vector and points? Here are the basic ones. Vector operations apply in the same way you learned in linear algebra. The difference $P - Q$ of two points results in a vector directed from Q to P. Point-vector addition $R + w$ results in the point which is translated by w from R. The following are the legal set of operations:

```
Vector = Scalar * Vector     Vector = Vector / Scalar
Vector = -Vector
Vector = Vector + Vector     Vector = Vector - Vector
Vector = Point - Point
Point  = Point + Vector      Point  = Point - Vector
```

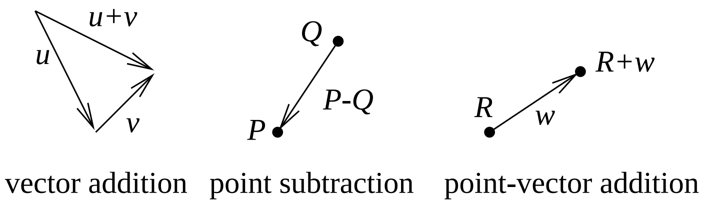


Figure 6: Affine operations.

Note that some operations (e.g. addition of points) are explicitly not defined. In every space there is a special vector called the *zero vector*, $\vec{0}$, that has no magnitude, such that $\vec{v} + \vec{0} = \vec{v}$.

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 the two, that we will consider legal. The operation is called an *affine combination*. Let's say that we want to compute the midpoint of a line segment, or more generally a point P on a line segment that break the segment into subsegments whose lengths are in the proportions α and $1-\alpha$ (the case $\alpha = 1/2$ is the case of the midpoint). The line segment might be specified by its endpoints P_0 and P_1. This could be done by first scaling the vector defined by the difference of the endpoints to the proper length, and then adding this to an endpoint:

$$P = P_0 + \alpha(P_1 - P_0).$$

Another way to think of this point P is as a *weighted average* of the endpoints P_0 and P_1. Thus we might be tempted to write the following (nonsensical) statement

$$P = (1-\alpha)P_0 + \alpha P_1$$

or to put this into a more symmetric form

$$P = \alpha_0 P_0 + \alpha_1 P_1 \qquad \text{where } \alpha_0 + \alpha_1 = 1.$$

See the figure below.

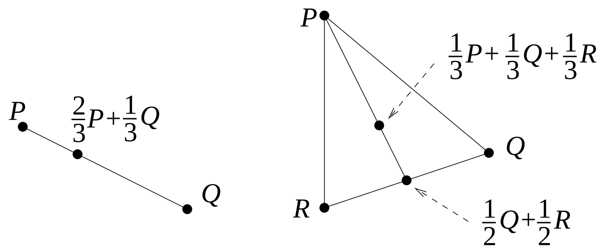


Figure 7: Affine combinations.

Because this notation is nice (even though it is not properly defined, because we cannot add points, nor can we multiply points by scalars) we will allow this special case. In particular, any arithmetic combination of points is legal, as long as the sum of coefficients is 1. For example, we would allow the following sort of operation, even though point addition is not generally legal.

$$Q = \frac{P_0 + P_1}{2};$$

In general it is possible to generalize the definition of affine combinations to k points

$$\alpha_0 P_0 + \alpha_1 P_1 + \ldots + \alpha_{k-1} P_{k-1},$$

where $\alpha_0 + \alpha_1 + \ldots + \alpha_{k-1} = 1$. For example, if P_0, P_1 and P_2 form a triangle in the plane, then the affine combination $(P_0 + P_1 + P_2)/3$ is the *centroid* (center of gravity) of the triangle.

Euclidean Geometry: The main difference between affine geometry and Euclidean geometry is that in Euclidean geometry we can talk about the notions of distances, lengths, and angles. A Euclidean geometry is defined to be an affine geometry together with an operator called an *inner product*. An inner product is an operator that maps two vectors to a scalar. We will not review the basic axioms behind inner products. (See your linear algebra text.) Instead we will focus on a the most familiar inner product, the *dot product*. 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 product is defined only for vectors, not for points.

Using the inner product we can define a number of familiar concepts that do not exist in affine geometry:

Length: $|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}$.

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

Angle between two vectors:

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

Orthogonality: $\vec{u}$ and $\vec{v}$ are orthogonal if $\vec{u} \cdot \vec{v} = 0$.

Orthogonal projection: The orthogonal projection of $\vec{w}$ onto $\vec{v}$ is given by

$$\vec{w'} = \frac{(\vec{w} \cdot \vec{v})}{(\vec{v} \cdot \vec{v})}\vec{v}.$$

(Note: There is an error in the book's formula.) The orthogonal remainder u is $w - w'$.

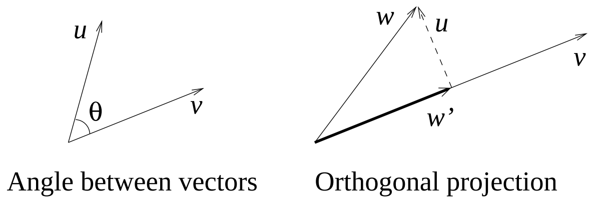


Figure 8: Inner product applications.

We can associate every nonzero vector with a vector of length 1 in the same direction, called a *unit vector*:

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

Replacing $\vec{v}$ with $\hat{v}$ is called *normalization.*

Lecture 7: Coordinate Systems and Frames

(Thursday, Feb 20, 1997)

Read: Chapt 4 and Appendix B in Angel.

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. We also consider one of the messiest problems in computer graphics, how to perform transformations between coordinate systems.

The first question is how to represent points and vectors in affine space. We know from linear algebra that if we have 3-linearly independent vectors, u_0, u_1, and u_2, in 3-space, then we can represent any other vector in 3-space 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. When the three vectors correspond to the three unit vectors $(1,0,0)$, $(0,1,0)$, $(0,0,1)$, then $(\alpha_0, \alpha_1, \alpha_2)$ are just the *Cartesian coordinates* of the vector $\vec{v}$. Such a set of vectors defines a *basis* for the space. Generally basis vectors need only be linearly independent. In most cases it will be convenient to assume that the basis vectors are of equal length and are mutually orthogonal.

Do define a coordinate system we need something that acts like a set of basis vectors? But vectors alone are not enough (since we cannot define a point from vectors). To specify position, we will add a point, denoted O, to serve as the origin of our coordinate frame. It is easy to see that for any point P, since $P - O$ is just some vector $\vec{v}$, P can be expressed as a sum of Oand some linear comination of basis vectors:

$$P = \alpha_0 \vec{u_0} + \alpha_1 \vec{u_1} + \alpha_2 \vec{u_2} + O,$$

for some α_1, α_2, α_3.

Definition: A *coordinate frame* in 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 see that P and w can be expressed in terms of u_0, u_1, and $\mathcal{O}$ as

$$\begin{aligned} P &= 2 \cdot u_0 + 2 \cdot u_1 + \mathcal{O} \\ w &= 2 \cdot u_0 + 1 \cdot u_1. \end{aligned}$$

Notice that the location of w is immaterial, and it would probably make things clearer to draw it as if it is originating from O.

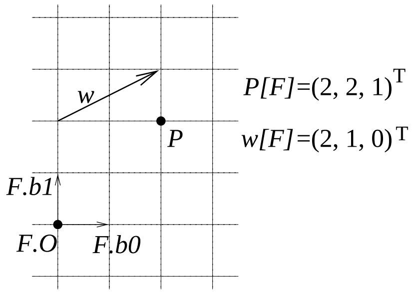


Figure 9: Coordinate Frame.

How do we formalize this into a coordinate system? To do this, we introduce the following axiom.

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

This is rather a strange axiom, but it serves a nice purpose. Using this notation, we can now write the point and vector of the figure in the following way.

$$\begin{aligned} P &= 2 \cdot u_0 + 2 \cdot u_1 + 1 \cdot \mathcal{O} \\ w &= 2 \cdot u_0 + 1 \cdot u_1 + 0 \cdot \mathcal{O} \end{aligned}$$

Thus, relative the the coordinate frame $F = \{u_0, u_1, O\}$, we can write P as the vector $P[F] = (2, 2, 1)^T$ and w as the vector $w[F] = (2, 1, 0)^T$. The superscript T means that we are taking the transpose of this row-vector, since (following the book and OpenGL conventions) we will represent points as column-vectors. Since this will be the normal case, we will often drop the superscript. Beware: The term "vector" now has two meanings, one as an free-vector in an affine space, and now as a column-vector 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.

Note that to represent points and vectors in d-space, we will generally use coordinate lists of length $d + 1$. Points have a last coordinate of 1, and vectors have a last coordinate of 0. We will refer to this as the *homogeneous coordinates* of a point or vector, relative to the frame F.

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 $P[F] = (1, 2, 1)$ and $Q[F] = (4, -1, 1)$, respectively. Consider the vector

$$v = P - Q.$$

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 $v[F] = (-3, 3, 0)$. Thus, to compute the coordinates of $P - Q$ we simply take the component-wise difference of the coordinate vectors for P and Q.

In general, a nice feature of this representation is the last coordinate behaves exactly as it should. Let x and y be either points or vectors. After the operation $x + y$ or $x - y$ or αx (when applied to the coordinates):

- If the last coordinate is 1 the result is a point.
- If the last coordinate is 0 the result is a vector.
- Else, this was an illegal 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. Allow the programmer to perform any combination of standard vector operations on coordinates, but 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:

```
u = (P + Q + R)/3;
```

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

Alternative coordinate systems: Any geometric programming system must deal with two conflicting goals. First, we want points and vectors to be represented with respect to some common coordinate system (so we can operate on points and vectors by just operating on their coordinate lists). But it is often more convenient to define points relative to some local coordinate frame.

For example, suppose an interior decorator wants to produce a CAD model of the A. V. Williams buiding, and specify the location of furniture. Where should the coordinate frame be positioned? It would be logical to place it within the room. However each room would then have its own coordinate frame. These coordinate frames would have to be related to one global coordinate frame for the entire building. To specify the location of a book on my desk, I might use a coordinate frame that is local to the desk, but this would require that I relate this frame to the coordinate frame for the room. In general, complex CAD models are presented in terms of numerous local coordinate frames, that are defined in terms of more global coordinate frames.

What is the most global coordinate frame? There is nothing in affine geometry that suggests where such a frame should go, since every point is the same as every other point, and every direction is the same as every other. 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 of unit length.

Change of coordinates: One of the most important geometric operations in computer graphics is that of converting points and vectors from one coordinate system to another. Let us consider a two dimensional example first.

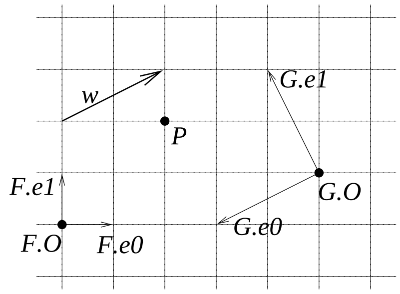


Figure 10: Change of Coordinates.

As we have seen, relative to frame F we have $P[F] = (2, 2, 1)$, and $w[F] = (2, 1, 0)$. But what is the coordinates of P and w relative to frame G? Our goal is to find constants β_0, β_1, β_2, such that $P = \beta_0 G.e_0 + \beta_1 G.e_1 + \beta_2 G.O$.

Observe that if we convert all coordinates into frame F we have $G.O = (5, 1, 1)$, $G.e_0 = (-2, -1, 0)$, and $G.e_1 = (-1, 2, 0)$. First, observe that

$$P - G.O = (-3, 1, 0).$$

Because $G.e_0$ and $G.e_1$ are linearly indenpendent, we can express $P - G.O$ uniquely as a linear combination of these two vectors. In particular if we solve a system of equations we have

$$(-3, 1, 0) = 1 \cdot (-2, -1, 0) + 1 \cdot (-1, 2, 0).$$

Hence we have

$$\begin{aligned} P = (2, 2, 1) &= 1 \cdot (-2, -1, 0) + 1 \cdot (-1, 2, 0) + 1 \cdot (5, 1, 1) \\ &= 1 \cdot G.e_0 + 1 \cdot G.e_1 + 1 \cdot G.O. \end{aligned}$$

Therefore $P[G] = (1, 1, 1)$ are the coordinates of P relative to G. As an exercise, see whether you can derive the fact that the coordinates for w are $(-1, 0, 0)$.

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 $G.O$ relative to frame F:

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

Further suppose that we know the coordinate of P relative to F, namely $P[F] = (\alpha_0, \alpha_1, \alpha_2)$. (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 vectors as well.)

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

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

This is an expression in affine geometry. If we write everything out using F as a coordinate system 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:

$$M = \begin{pmatrix} g_{00} & g_{10} & g_{20} \\ g_{01} & g_{11} & g_{21} \\ 0 & 0 & 1 \end{pmatrix} = \left(G.e_0 \,\middle|\, G.e_1 \,\middle|\, G.O \right).$$

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

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

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 matrixed are stored in the following order:

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

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:

```
double M[4][4];
```

if you want to access the element in row i and column j, you need to refer to it by `M[j][i]` not `M[i][j]`. Alternatively, you can declare it `double M[16]` and perform your own indexing.

Lecture 8: Affine Transformations

(Tuesday, Feb 25, 1997)
Read: Chapt 4 and Appendix B in Angel.

Transformations: So far we have been stepping through the basic elements of our geometric system. 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.

Now we consider transformations of space. Although it is generally posible to define very complex functions applied to space (mapping points points and vectors to vectors), we will be interested in a rather limited class of transformations that have the property of mapping lines to lines. Transformations are classified according to what they preserve.

Rigid (Euclidean) transformations: These transformations preserve angles and lengths. Examples include translations and rotations.

Orthogonal (Homothetic) transformations: These transformations preserve angles but not necessarily lengths. In addition to rigid transformations, uniform scaling is allowed.

Affine transformations: These transformations preserve affine combinations. For example, of R is the midpoint of points P and Q, then after applying any affine transformation T, $T(R)$ is the midpoint of $T(P)$ and $T(Q)$. Affine transformations preserve parallelism, but the angles between nonparallel lines may be altered.

Projective transformations: These transformations include all affine transformations and preserve something called a *cross ratio*. These transformations are not actually defined over Euclidean space, but over an extended space called *projective space*. We wil not be concerned with the complete general class projective transformations, but consider the special cases that we will need for perspective viewing.

Affine Transformations: We bein by discussing *affine transformations*. These are transformations that preserve affine combinations. This includes transformations such as translations, rotations, uniform and nonuniform scalings (stretching the axes by some constant scale factor), and reflections (flipping objects about a line) and shearings (which we will discuss later). These transformations are generally invertible (except for scalings by a factor of 0).

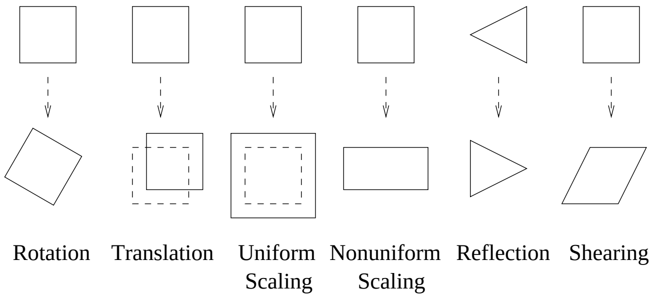


Figure 11: Examples of affine transformations.

An important operation on affine transformations is *composition*. For example, we may want an affine map that first rotates space, and then translates it. This could be achieved by deriving both of these transformations and them composing them to form the combined transformation. An important aspect of affined transformations (and all the classes defined above) is that they are closed under composition.

How are transformations defined? We will see that any affine transformation can be described by a 4×4 matrix, which is then multiplied by the coordinates of either points or vectors to perform the transformation. However, people fine it easier to think in terms of high-level geometric operations (such as "rotate" and "translate") rather than the specific entries made in a matrix. Nonetheless, in the design of a package of geometric routines, you may have to be aware of how these matrices are constructed and manipulated.

We have already seen one application of matrix multiplication, namely for effecting a change of coordinate representations. An important aspect of good geometric computing is making clear the intent of each matrix multiplication (change of coordinates, affine transformation, perspective transformation, etc).

There are generally two methods that are used for defining complex affine transformations. The first is by defining a mapping between coordinate frames, and the second is by composing a set of basic transformations. Both methods have advantages and disadvantages, so we will discuss them both.

Mapping coordinate frames: In general given any two frames F_1 and F_2, there is a unique affine transformation that maps between them. If the coordinate frames are both orthonormal then the resulting transformation is a rigid transformation, and if the frames are orthogonal but the unit vectors one frame are uniformly longer than the other frame then the resulting transformation will be rigid. The construction method has the advantage that it can be used to define mappings between one space and a space of lower dimension.

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_1 and F_2, such that the object is in the same position relative to each frame, as shown on the right.

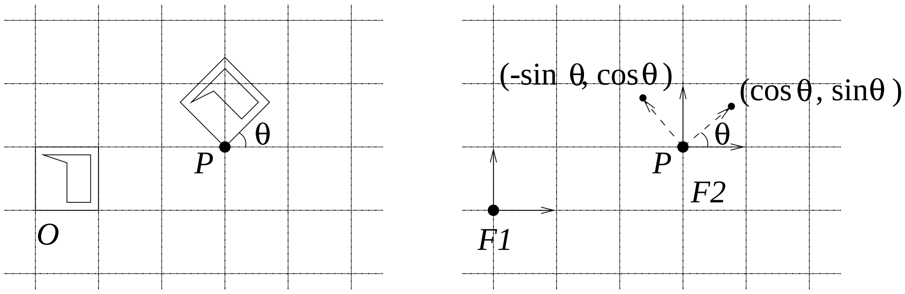


Figure 12: Example of constructing affine transformations.

For example, suppose for simplicity that F_1 is just the standard frame. Then the frames F_2 is composed of the elements

$$\begin{aligned} u_0 &= (\cos\theta, \sin\theta, 0)^T \\ u_1 &= (-\sin\theta, \cos\theta, 0)^T \\ P &= (3, 1, 1)^T. \end{aligned}$$

The corresponding matrix for the transformation would just consist of the concatenation of these elements:

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

As a check, consider the lower right corner point Q of the original square, whose coordinates are $(1,0,1)^T$. The product AQ yields

$$(3+\cos\theta, 1+\sin\theta, 1) = P + (\cos\theta, \sin\theta, 0),$$

as expected. Notice that the matrix A is the same as the matrix F_2.toStdFrame, which we discussed last time.

This example was made simpler by the fact that F_1 was the standard frame. If this were not the case, then we would first have to convert coordinates of the point from the standard frame into their representation relative to frame F_1. This is done by the transformation F_1.fromStdFrame. Recall that this matrix comes about by first taking the elements of frame F_1, concatenating them together, and then computing the inverse of the resulting matrix. It is not hard to show (but it is generally a little tricky to believe) that given any two frames F_1 and F_2 the matrix representation of the affine transformation that maps the elements of F_1 to the elements of F_2 is

$$A = (F_2.\text{toStdFrame}) \cdot (F_1.\text{fromStdFrame}).$$

Pay close attention: This matrix assumes that points and vectors are given in terms of their representation *in standard coordinates* and returns points and vectors in this same representation. It may seem strange that we have used matrices whose function is to convert between representations to perform an entirely different function (that of mapping points) but it just turns out that this is the correct matrix by basic matrix algebra.

Basic Transformations: The above method is good when you know the initial and final positions of some object. In many cases it is more natural to think of the transformation in terms of a series of elemental transformations (first rotate, then translate, etc.) Complex transformations can be built up from these basic transformations and matrix multiplication.

Translation: Translation by a fixed vector $\vec{a}$ maps point P to $P+\vec{a}$. Note that vectors are not altered by translation. (Why not?) Suppose that $(\alpha_x, \alpha_y, \alpha_z, 0)^T$ are the homogeneous coordinates of $\vec{a}$. The translation matrix is

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

Scaling: Scaling the axes by the scale factors β_x, β_y, β_z, is given by the matrix

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

Both points and vectors are scaled.

Rotation: There are three basic rotations: about the x, y and z-axes. In each case the rotation is about the origin and the angle θ gives the amount of the rotation according to a right-hand rule. The rotation matrices are

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

and

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

and

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

Shearing: A shearing transformation applies a stretch orthogonal to some axis that depends on the distance along this axis. Intuitively a shear turns cubes into parallelograms. For example, suppose that the shearing axis is the z-axis and we want to shear along the x-direction by some angle θ then the resulting shearing matrix would be

$$H_{xz}(\theta) = \begin{pmatrix} 1 & 0 & \tan\theta & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Shears involving any other pairs of axes are defined similarly.

Lecture 9: Angles, Areas, and Applications

(Thursday, Feb 27, 1997)

Read: Chapt 4 and Appendix B in Angel. Some of this material is not covered in the text.

Cross Product: Let us review the cross-product operator. Given two vectors in 3-space, $\vec{u}$ and $\vec{v}$, their *cross product* is define to be

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

This is easier to remember as the symbolic determinant:

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

where $\vec{e}_x$, $\vec{e}_y$, and $\vec{e}_z$ are the three coordinate unit vectors.

Note that the cross product is only defined for vectors in 3-space (the generalization would require d vectors in d-space). It has the following nice properties:

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. Thus, this is a nice device for constructing orthogonal vectors for coordinate frames.

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

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

where θ is the angle between $\vec{u}$ and $\vec{v}$.

Orientation: Given two real numbers a and b, there are three possible ways they may be ordered: $a < b$, $a = b$, or $a > b$. Equivalently we may define $\mathrm{Or}_1(a, b) = \mathrm{sign}(a - b)$, where $\mathrm{sign}(x)$ is either -1, 0, or $+1$ depending on whether x is negative, zero, or positive, respectively.

In general given any $d + 1$ points in d dimensional space, we may define their *orientation* to be the sign of the determinant consisting of their homogeneous coordinates. For example, in the plane, the orientation of three points A, B, C is defined to be

$$\mathrm{Or}_2(A, B, C) = \begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{vmatrix}. \qquad \mathrm{Or}_3(A, B, C, D) = \begin{vmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 1 & 1 & 1 & 1 \end{vmatrix}.$$

(Note that this generalizes the 1-dimensional definition.) The orientation of 3 points will be $+1$ if the triangle ABC is oriented counter-clockwise, -1 if clockwise, and 0 if they are collinear.

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

Orientation is a nice operation to keep in mind. For example, suppose you want to know whether two line segments AB and CD intersect in the plane. Although this sounds like a simple problem, it (like many simple geometry primitives) is fraught with special cases and potential pitfalls. For example, you need to consider the possibility of lines being parallel, collinear, or intersecting in a T-junction, etc.

However, if you assume that no three of the points are collinear, then observe that the segments intersect if and only if A and B lie on the opposite sides of the line CD, and if C and D lie on opposite sides of the line AB. To determine these you could try to extract the line equations for AB and CD, but this process may run into problems. (E.g. what if you try to compute the slope, and the line is vertical?) The cleanest approach is to use the orientation test (e.g. test that $\mathrm{Or}_2(C, D, A) \neq \mathrm{Or}_2(C, D, B)$ and $\mathrm{Or}_2(A, B, C) \neq Or_2(A, B, D)$.

Application: General Rotation: As an application of what we have learned about affine transformations, let us consider a pretty typical example of applying these methods: rotation about an arbitrary vector in 3-space. We are given a nonzero vector $\vec{u}$ and an angle θ and we want to derive the matrix for an affine transformation that rotates through angle θ around vector $\vec{u}$ (rotating say, around the origin).

The book describes how to do this through the method of composing basic transformations. In particular, they show that the final matrix can be expressed as the product of 5 basic rotations (two rotations to align $\vec{u}$ with the x-axis, one to rotate around the x-axis, and two to restore $\vec{u}$ to its original placement).

We will give a way to do this using the method of mapping coordinate frames, which I think is intuitively simpler. The idea is to create an orthonormal frame in which u takes the role of the x-axis. Then we transform to this coordinate system, apply a rotation about the x-axis, and transform back.

To begin, define the *normalization* of a nonzero vector $\vec{u}$, denote $\hat{u}$, to be the vector divided by its length

$$\hat{u} = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}}.$$

We want to construct a basis consisting of three vectors, $\vec{u}$, $\vec{v}$, and $\vec{w}$, that are mutually orthogonal and of unit length. The best way to do this from a numerical standpoint would be to apply the Gram-Schmidt procedure (described in Appendix A) to the set consisting of $\vec{u}$, and any two of the coordinate vectors. (To avoid numerical instabilities, take the unit vectors corresponding to the coordinates of u with the smallest absolute values.)

However, a quick and dirty solution would be the following. If u is parallel to the x-coordinate vector, $\vec{e}_x$, then just to a rotation about the x-axis. Otherwise, use cross-products to compute:

$$\begin{aligned} \vec{v} &= \vec{u} \times \vec{e}_x \qquad \text{(perpendicular to } u\text{)} \\ \vec{w} &= \vec{u} \times \vec{v}. \qquad \text{(perpendicular to } u\text{, and } v\text{)} \end{aligned}$$

Then define a frame F consisting of the standard origin O, and the normalized vectors $\hat{u}$, $\hat{v}$, and $\hat{w}$.

In this frame the transformation is just a rotation about the x-axis (because it is aligned with u). The final transformation is:

$$F.\text{toStdFrame} \cdot R_x(\theta) \cdot F.\text{fromStdFrame}.$$

Application: Ray-Shooting in the plane: We will study ray-shooting later as a method for producing highly realistic images. The basic question that is answered over and over in ray shooting, is which object does a ray hit? We will considered a simplified problem in the plane: Given a ray and a circle, where (if at all) does the ray hit the circle?

We represent a ray R by giving its origin point P and a directional vector $\vec{u}$. Any point on the ray can be described by the combination

$$R(t) = P + t\vec{u},$$

where $t \geq 0$ is a scalar. Suppose that the circle 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 circle, or to report that there is no intersection.

We know that a point Q lies on the circle if its distance from the center of the circle is r, that is if $|Q - C| = r$. So the ray intersects at the value of t such that

$$|P + t\vec{u} - C| = r.$$

Let $\vec{v} = P - C$, from which we have

$$|t\vec{u} + \vec{v}| = r.$$

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 is a scalar equation (not a vector equation). 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, where

$$\begin{aligned} a &= (\vec{u} \cdot \vec{u}) \\ b &= 2(\vec{u} \cdot \vec{v}) \\ c &= (\vec{v} \cdot \vec{v}) - r^2 \end{aligned}$$

We can solve this using the quadratic formula. The discriminant is $D = b^2 - 4ac$. If this is negative then there is no solution (implying that the ray missing the circle). If it is zero then the ray just grazes the circle. If it is positive then there are two roots:

$$t = \frac{-b \pm \sqrt{D}}{2a}.$$

Which do we take? We should take the smaller positive root (smaller because it represents the first intersection, and positive because $t \geq 0$ is a constraint for lying on the ray.)

Lecture 10: Damage Control

(Tuesday, Mar 4, 1997)
This lecture addresses some errors found Chapt. 4 of the text.

Changing Frames Example: Some students have been trying to read through the book to get a better understanding of how rotations and change of frames works. Unfortunately, the book appears to contain a number of critical errors (which is unfortunate, given that this is already a confusing topic).

The first example arises in Section 4.3.4 but the problem originates in Section 4.3.3. In Sections 4.3.2 and 4.3.3 the author introduces change of coordinates in standard linear space (not affine space), and then introduces homogeneous coordinates. For some reason, he switches in Section 4.3.3 from writing vectors and points as column-vectors, to writing them as row-vectors. This means that all the matrices he generates need to be transposed before they can be used. Thus, the matrix given at the top of page 132 should be transposed before being used.

To add to the confusion, on page 132 in the middle he gives the equation:

$$u = M^T v,$$

which is in error. It should be

$$v = M^T u$$

Please make a note of this.

The problems continue in Section 4.3.4, because he now starts mixing the transposed matrix M from Section 4.3.4 with the untransposed matrix A from Section 4.3.3 and winds up with complete garbage.

Let's try the example from scratch. I am going to use his notation, so you can see where the error occurs. First, we have a frame in 3-space whose basis vectors are $\vec{e}_1$, $\vec{e}_2$, and $\vec{e}_3$, and whose origin is the point Q_0. Let us call this frame Q. Second we have a frame with basis vectors $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ and origin is the point P_0. Call this frame P.

Frame P has been defined in terms of frame Q. (For example, we might think of Q as being the standard frame and P as being the frame of the camera.) In particular,

$$\begin{aligned} \vec{v}_1 &= \vec{e}_1, \\ \vec{v}_2 &= \vec{e}_1 + \vec{e}_2, \\ \vec{v}_3 &= \vec{e}_1 + \vec{e}_2 + \vec{e}_3. \\ P_0 &= -\vec{e}_1 - 2\vec{e}_2 - 3\vec{e}_3 + Q_0 \end{aligned}$$

Thus, the coordinates of these objects relative to frame Q are:

$$v_1[Q] = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad v_2[Q] = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad v_3[Q] = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \qquad P[Q] = \begin{pmatrix} -1 \\ -2 \\ -3 \\ 1 \end{pmatrix}.$$

We want to find the matrices M and M^{-1} that convert between the representations of a point or vector in these two frames. In particular, suppose the $u[Q]$ are the coordinates of object u relative to frame Q and $u[P]$ are the coordinates of u relative to frame P. Then we want to find matrix M such that $u[Q] = Mu[P]$ and $u[P] = M^{-1}u[Q]$. (If we think of Q as being the standard frame, then M is P.toStdFrame and M^{-1} is P.fromStdFrame.

The derivation of the matrix is explained (in transpose form) in Section 4.3.4, but the bottom line is that the matrix that converts from P's representation to Q's representation is formed by just concatenating the frame elements of frame P. (In the book he incorrectly concatenates the column-vectors for $\vec{v}_i[Q]$ with the row-vector of $P[Q]$ and gets garbage as a result.) Thus the final matrix is

$$M = \begin{pmatrix} 1 & 1 & 1 & -1 \\ 0 & 1 & 1 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This converts $u[P]$ to $u[Q]$. With a little work we can invert this, yielding

$$M^{-1} = \begin{pmatrix} 1 & -1 & 0 & -1 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This converts $u[Q]$ to $u[P]$.

Let's consider an example. Consider the point

$$R = 2\vec{v}_1 - 3\vec{v}_2 + P.$$

Thus, $R[P]$ is the column-vector $(2, -3, 0, 1)^T$. To get $R[Q]$, we multiply M times $R[P]$ giving:

$$M \cdot R[P] = \begin{pmatrix} 1 & 1 & 1 & -1 \\ 0 & 1 & 1 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ -3 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ -5 \\ -3 \\ 1 \end{pmatrix} = R[Q].$$

As a check, we could work this out algebraically as well. From the definition of $R[P]$ and then substituting in the definitions of $\vec{v}_i$ and P_0 we have:

$$\begin{aligned} R &= 2\vec{v}_1 + (-3)\vec{v}_2 + 0\vec{v}_3 + P_0 \\ &= 2(\vec{e}_1) + (-3)(\vec{e}_1 + \vec{e}_2) + 0(\vec{e}_1 + \vec{e}_2 + \vec{e}_3) + (-\vec{e}_1 - 2\vec{e}_2 - 3\vec{e}_3 + Q_0) \\ &= (2 - 3 - 1)\vec{e}_1 + (-3 - 2)\vec{e}_2 - 3\vec{e}_3 + Q_0 \\ &= (-2)\vec{e}_1 + (-5)\vec{e}_2 + (-3)\vec{e}_3 + Q_0 \end{aligned}$$

implying that the $R[Q] = (-2, -5, -3, 1)^T$.

3-d Rotation Example: Last time we saw how to use the method of "mapping between coordinate frames" to affect a general rotation in 3-space. I refered you to the text book for an explanation of how to accomplish this using the method of "composing basic transformations". A number of conscientious students undertook this exercise, and found that the book's answer was in error. I will work this example out as well.

Let us suppose that we have a vector $\vec{u}$ about which we want to rotate through an angle of θ. We assume that we *normalize* $\vec{u}$ as vector $\vec{v}$ where

$$\vec{v} = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}} = \begin{pmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{pmatrix}.$$

The idea is to rotate space so that this vector coincides with one of the coordinate axes. This generally requires two rotations, and is somewhat difficult to visualize (unless you have a good virtual reality system). The key is to consider carefully chosen 2-dimensional projections.

We first rotate space so that vector $\vec{v}$ lies on one of the coordinate planes. (We'll use the xz-plane.) Once the vector in on this plane, then we can rotate this plane around the y-axis, until it coincides with either the x- or z-axis. (We'll use the z-axis). Once the vector is aligned with the z-axis, we can just do a z-rotation, and then rotate back to the original situation (by multiplying by the inverse rotations).

Our first rotation will be around the x-axis until the vector lies in the xz-plane. Observe that any such rotation will rotate the yz-plane about the origin. Consider the *orthogonal projection* of vector $\vec{v}$, denoted $\vec{v}'$, onto the yz-plane. Its two-dimensional coordinates are (α_y, α_z). If we rotate $\vec{v}$ about the x-axis far enough that it hits the xz-plane, notice that its "shadow" $\vec{v}'$ will rotate until it hits the z-axis (because we have projected out x).

What is this angle? From the figure in the book it is easy to see that $\vec{v}'$ lies on the hypotenuse of a right triangle whose base (on the z-axis) is of length α_z, whose other side is of length α_y, and hence the hypotenuse is of length $\sqrt{\alpha_y^2 + \alpha_z^2}$. We may assume that this length is nonzero. (Why? If not then $\vec{v}$ coincides with the x-axis, and we are done.) The angle that $\vec{v}'$ forms with the z-axis is clearly

$$\theta_x = \cos^{-1} \frac{\alpha_z}{d},$$

where d is the length $\sqrt{\alpha_y^2 + \alpha_z^2}$ of the hypotenuse. The sine of θ_x is α_y/d, implying that the desired rotation matrix is

$$R_x(\theta_x) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z/d & -\alpha_y/d & 0 \\ 0 & \alpha_y/d & \alpha_z/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

(It is a good idea to check at this point that the angle of rotation has the proper sign. It is easy to make mistakes here. According to the figure, this rotation is being performed according to a right-hand rule with respect to the x-axis, and so the sign of the angle is positive. If the direction was left-handed, then we would need to negate the sines, α_y/d in the matrix.)

In this case we could compute the cosine and sine of the angle explicitly. But, if you are computing the angle in order for it to be passed to some other computation, you should be careful about deriving angles from their sines and cosines, since these functions are not generally 1–1 for angles over the range from 0 to 2π (e.g. $\cos\theta = \cos\theta$). There are also problems with potentially dividing by 0. Normally you should avoid computing the angle if all you need is the sine and cosine (since this is an extra step that can introduce errors) but if you want the angle, then a safer way to compute is using the built-in function `atan2(y,x)`. It returns the angle over the range 0 to 2π whose tangent is y/x. This works even if $x = 0$.

After this rotation, the rotated vector $\vec{v}''$ lies in the xz-plane, and so we can rotate it into the z-axis by rotating about the y-axis. The vector is of length 1, its projection onto the x-axis has not changed from the previous rotation, and its projection on the z-axis is just d. Thus the angle formed between the vector and the z-axis is

$$\theta_y = \cos^{-1}\frac{d}{1} \qquad (??)$$

But this is wrong! We must be careful now, since the rotation is being performed left-handed about y, thus the angle should be negated.

$$\theta_y = -\cos^{-1}\frac{d}{1}.$$

The sine of this angle is $-\alpha_x$. The resulting rotation matrix is

$$R_y(\theta_y) = \begin{pmatrix} d & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Now, we can finally rotate about the z-axis through the desired original angle θ. But then we have to put things back the way we found them. Thus we "unrotate" around y by angle $-\theta_y$ and around x by angle $-\theta_x$. The final string of matrices is

$$R_{\vec{v}}(\theta) = R_x(-\theta_x)R_y(-\theta_y)R_z(\theta)R_y(\theta_y)R_x(\theta_x).$$

In the text an example is worked through in detail. They consider rotating by an angle of 45 degrees around the vector with homogeneous coordinates $(1, 2, 3, 0)^T$. Unfortunately, the results are all wrong. They correctly normalize the vector to

$$\vec{v} = \left(\frac{1}{\sqrt{14}}, \frac{2}{\sqrt{14}}, \frac{3}{\sqrt{14}}, 0\right)^T$$

but when they compute the length of $\vec{v}' = (2/\sqrt{14}, 3/\sqrt{14})$ they get $d = \sqrt{14}$ rather than $d = \sqrt{13}$ as they should. In all the matrices involving θ_x the denominator should be changed to $\sqrt{13}$. The error affects the rest of the calculations.

Lecture 11: Perspective

(Thursday, March 6, 1997)

Read: Chapt 5. Before reading, be sure to check the errata list. Half of the z-coordinates in the chapter should be negated.

`http://www.cs.unm.edu/~angel/CS433/errata/errata.html`.

Basic Perspective: Now let's consider the problem we have been building up to, namely performing perspective transformations. 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 d dimensional space onto a $d-1$ dimensional plane (called the *projection plane*, centrally through a point (not on this plane) called the *center of projection*.

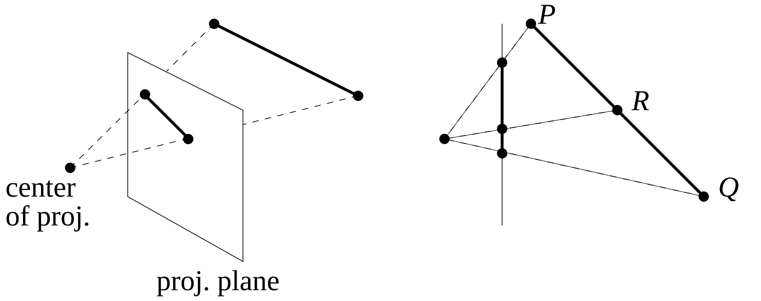


Figure 13: Perspective Transformations.

One nice things about projective transformations is that they map lines to lines. The main difficulty with projective transformations is that (except for the special case of parallel projection) they are not affine transformations, in the sense that they do not preserve affine combinations. For example, consider the perspective projection T shown in the figure. If R is the midpoint of segment PQ, and A is a projective transformation, then $T(R)$ is not the midpoint of $T(P)$ and $T(Q)$.

Projective Geometry: Projective transformations can be rather tricky things to deal with. In part, this is because of some of the strange aspects of *projective geometry*, of which they are part. 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 unpredictable.

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 meet each other.

This is very elegant. Now all lines behave much like closed curves (somewhat like a circle of infinite radius). This seems to be a very nice generalization of circles and lines.

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 this point. 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 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.

Although the points at infinity seem to be special in some sense, an important tenent of projective geometry is that they are 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.

This system seems to be so nice and general, that it seems as though we should have just

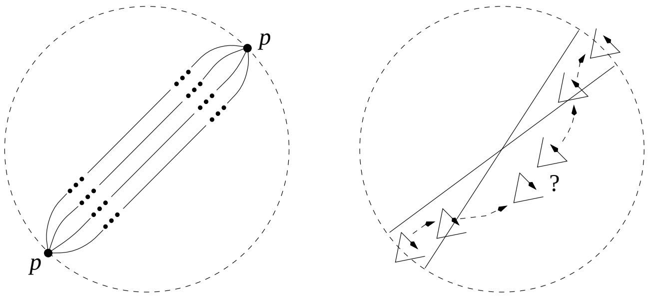


Figure 14: Projective Geometry.

dispensed with affine geometry, and used projective geometry instead. But 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). That is, it is possible to take an object with a counterclockwise orientation, and then translate it to the same position with a clockwise orientation. (Note that this cannot be done on orientable manifolds like the Euclidean plane or the surface of a sphere). The reason is that as we pass through infinity, there is a "twist" in space as is illustrated in the figure on the right.

For these reasone, we choose not to use the projective plane as a domain in which to do most of our geometric comptutations. Instead, we will briefly enter this domain, just long enough to do our projective transformations, and quickly jump back into "familiar" 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. (Free vectors will be used to construct the geometric objects, such as points, lines, and polygons that will be transformed with our perspective transformation, but they will not be projected themselves.)

Consider a regular point P in the plane, with standard (nonhomogeneous) coordinates (x, y). 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:

$$\begin{pmatrix} ax \\ ay \\ a \end{pmatrix}, \qquad \text{for } a \neq 0.$$

Thus, if $P = (4, 3)$ are P's nonhomogeneous coordinates, the homogeneous coordinates $(4, 3, 1)$, $(8, 6, 2)$, and $(2, 1.5, 1/2)$ 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 *normalization* of P is the coordinate vector $(x/w, y/w, 1)^T$. (Do not confuse this with the normalization of a vector, which has a completely difference meaning.)

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

$$(1,2,1),(2,4,1),(3,6,1),(4,8,1),\ldots$$

These are equivalent to the following

$$(1,2,1),(1,2,1/2),(1,2,1/3),(1,2,1/4),\ldots$$

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

Projective Transformations: 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. (This follows OpenGL's convention. We look along $-z$ rather than $+z$ so that our coordinate frame will be 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. This causes some confusion (see our book for example).

This is shown in the following figure.

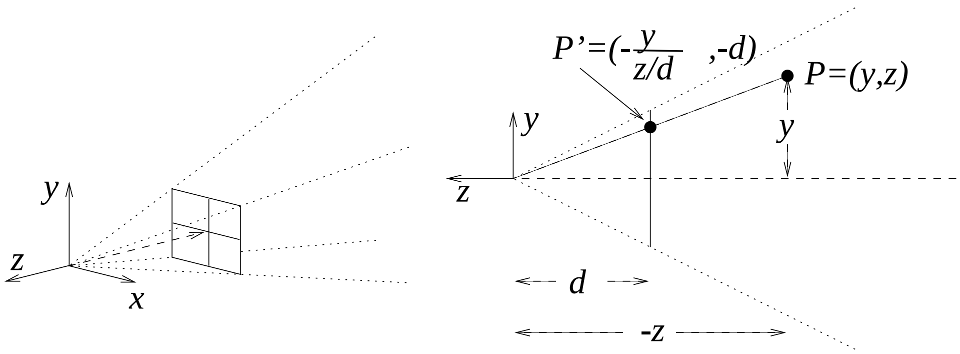


Figure 15: Perspective transformation.

Consider a point (y,z) in the plane. Note that z is negative but d is positive. Where should this point be projected to on the projection plane? Let (y_p,z_p) denote its projection onto the projection plane. By similar triangles it is easy to see that we have the ratios:

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

implying that $y_p = -y/(z/d)$. We also have $z = -d$. Generalizing this to 3-space we have that the point $(x, y, z, 1)$ should be transformed to

$$\left(\frac{x}{-(z/d)}, \frac{y}{-(z/d)}, -d, 1\right)^T.$$

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

$$\left(x, y, z, -\frac{z}{d}\right)^T.$$

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 transformed point $P' = M \cdot P$, we normalize 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.

In general it is a fact (which we will not prove) that every projective transformation can be expressed in two indivisible steps:

Linear part: multiply the homogeneous coordinates of the point by a $(d+1) \times (d+1)$ matrix.

Projection part: normalize, by dividing through by the last coordinate.

Lecture 12: More Perspective

(Tuesday, March 11, 1997)
Read: Chapt 5. (Check errata list on web.)

Depth Information: Last time we introduced the notion of projective geometry, and homogeneous coordinates for projective geometry. Recall that in this system, there are no vectors and that any nonzero scalar multiple of the coordinates of a point all represent the same point. We assumed that point coordinates are given relative to the camera's coordinate frame (e.g. through a change of coordinates transformation). We then showed that the perspective projection could be achieved by first multiplying the point's coordinates by an appropriate 4×4 matrix, and then *normalizing* by dividing through by the last (homogeneous) coordinate.

The problem with the perspective matrix we defined above is that information regarding the distance of the point from the viewer (which is proportional to the z-coordinate) is lost in the projection transformation. Recall that the point $(x, y, z, 1)^T$ is mapped to the point $(x/z, y/z, 1, 1)^T$. We would like to capture the depth information, because it is needed in hidden-surface elimination. In particular, among the many objects project onto the same point of the projection plane, the closest one to the eye is the only one that is visible (assuming opaque objects).

Today we consider the perspective transformation in greater detail and show how to extract this information.

Perspective with Depth: There is a very nice trick for adding depth 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 contains depth information. It turns out that this depth information will be subject to a nonlinear distortion. The important thing is 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.

There are a number of ways of doing this. 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 $z = -1$ (rather than $z = -d$ as last time). Consider the following matrix:

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

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

$$M_{depth}P = \begin{pmatrix} x \\ y \\ \alpha z + \beta \\ -z \end{pmatrix} \equiv \begin{pmatrix} -x/z \\ -y/z \\ -(\alpha z + \beta)/z \\ 1 \end{pmatrix}.$$

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

$$z' = -\frac{\alpha z + \beta}{z} = -\alpha - \frac{\beta}{z},$$

which, depending on the values of α and β, is a monotonic function of z. Thus, by choosing α and β properly, we can use this coordinate to store depth information.

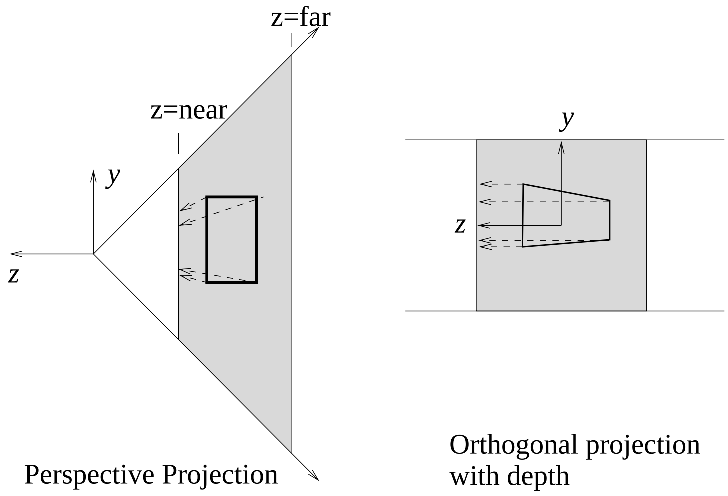


Figure 16: Perspective with depth.

Viewing in OpenGL: OpenGL performs perspective by passing points through a series of transformations. After each transformation is applied, the points have been mapped into a new coordinate system. This process is illustrated below.

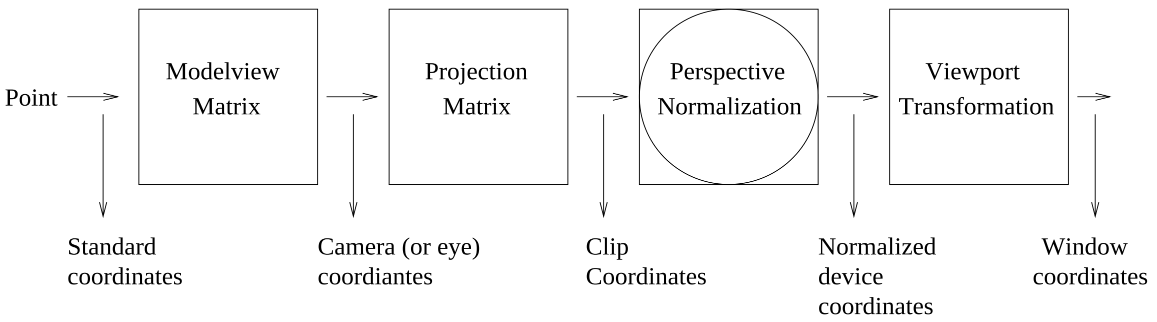


Figure 17: OpenGL Viewing Process.

First points are given in the standard coordinate frame. They are multiplied by a matrix called the *modelview matrix* which converts to camera coordinates. This is the coordinate system we described yesterday (x is left, y is up, and $-z$ is the direction of the camera). The modelview matrix may be constructed by composing a series of basic transformations (rotations, translations, etc), by creating a frame C for the camera, and then loading in the matrix C.fromStdFrame, or by the procedure `gluLootAt()` given below.

The next matrix in the chain is called the *projection matrix*. This matrix is responsible for performing the projection (either orthogonal or perspective). In OpenGL, the commands for defining matrices work by (1) specifying the current matrix you want to modify, and (2) then either loading a specific matrix or multiplying it (on the right) by a new matrix.

The matrix being modified is set by the call

```
glMatrixMode(GLenum mode);
```

where `mode` is either `GL_MODELVIEW`, the modelview matrix, `GL_PROJECTION`, the projection matrix, or `GL_TEXTURE`, the texture matrix (which we will discuss later this semester). Once the current matrix is set, you can load an identity matrix using

```
glLoadIdentity();
```

You can also load your own matrix with

```
glLoadMatrix..(..);
```

I have omitted the argument list and exact specification, because (as with many OpenGL commands) you can specify the matrix in many different ways. Beware: OpenGL stores matrices in column-major and not the usual row-major order. There are also procedures for multiplying the matrix, and performing various basic transformations (translate, rotate, etc.).

The simplest way to set the modelview matrix for a particular viewing situation is through the following procedure:

```
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_x, eye_y, eye_z) is the location of they eye (or the camera), it is directed so that it is pointing at the point $(center_x, center_y, center_z)$ (thus the vector $center - eye$ is the viewing direction).

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 this the vector (up_x, up_y, up_z) provides the direction that is "up" relative to the camera. Under normal circumstances, this would just be a vector pointing straight up. 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, but it cannot be parallel to the viewing direction vector.

This transformation should be applied to the OpenGL modelview matrix. OpenGL simply constructs this matrix and mutiplies it times whatever matrix is currently active. Since the modelview matrix is the default matrix, and it is initialized to the identity, then normally you can just issue this command. However, if you want to be safe, you may proceed it with either of the following two commands, which select the modelview matrix and initialize it to the identity:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(...);
```

Specifying the projection in OpenGL: OpenGL supports both orthogonal projection (a parallel projection where the projection plane is orthogonal to the direction of projection) and perspective projection. To specify an orthogonal parallel projection, use the function

```
glOrtho(left, right, bottom, top, near, far);
```

The arguments are all of type `GLdouble`. This defines the boundary of a 3-dimensional rectangle, bounded by the constraints:

$$\text{left} \leq x \leq \text{right} \qquad \text{bottom} \leq y \leq \text{top} \qquad \text{near} \leq -z \leq \text{far}.$$

Warning: Consistent with OpenGL's convention of looking along the negative z-axis, positive near and far values correspond to negative z-coordinates. The projection is parallel along the z-axis. All geometric objects are clipped at the boundary of this rectangle before being displayed.

This transformation should be applied to the OpenGL projection matrix. So, before invoking this procedure, you should set things up with:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(...);      -or-     glPerspective(...)
```

There are a couple of ways to specify a perspective projection. The most general is

```
glFrustum(left, right, bottom, top, near, far);
```

The arguments are all of type `GLdouble`. This defines a *viewing frustum*, which is defined as follows. Imagine the plane $z = -near$. On this plane, draw a rectangle with lower-left corner $(left, bottom, -near)$ and upper-right corner $(right, top, -near)$. Now, extend a pyramid whose apex is the origin and whose edges pass through the corners of this rectangle. Extend this pyramid as far as $z = -far$. This truncated pyramid (or frustum) is called the *viewing frustum* (see Fig. 5.25 on page 190 in our book, but note that they should negate all the z-coordinates in the figure).

The procedures `glOrtho()` and `glFrustum()` essentially create and multiply a particular matrix times the current perspective matrix. Next time we will see exactly how this matrix is constructed.

Lecture 13: Perspective Implementation

(Thursday, March 13, 1997)

Read: Chapt 5. (Check errata list on web. Warning: I could not reproduce the results of Section 5.8.2, so I have derived the results here by a slightly different approach.)

Canonical View Volume: Last time we presented the basics of *perspective-with-depth*, a perspective transformation that retains depth information and OpenGL's procedures for orthogonal and perspective projections. Today we will see how to implement these as matrices.

The goal of these transformations is to map the visible portion of the 3-dimensional scene to a fixed shape, called the *canonical view volume*. As we saw yesterday, the canonical view volume can be thought of as a 3-dimensional rectangle. (This is the natural state of affairs in an orthogonal transformation, and in the case of a perspective transformation, this is the result of the perspective-with-depth transformation.) The canonical view volume will be the rectangle bounded by the inequalities:

$$\begin{aligned} -1 \leq & \quad x & \leq +1, \\ -1 \leq & \quad y & \leq +1, \\ -1 \leq & \quad z & \leq +1. \end{aligned}$$

The (x, y) coordinates are used to indicate the location of the projected point on the final viewing window. The z-coordinate is used for depth. In the canonical view volume, we finally get rid of the constant negation of z-coordinates. In particular lower z-values correspond to nearer points and higher z-values correspond to further points.

Orthogonal Transformation: Consider the OpenGL call that defines an orthogonal projection:

```
glOrtho(l, r, b, t, n, f);
```

which defines a viewing volume with the constraints:

$$\begin{aligned} l \leq & \quad x & \leq r, \\ b \leq & \quad y & \leq t, \\ n \leq & \; -z & \leq f. \end{aligned}$$

Note the negation of z.

Mapping this to the canonical view volume involves a simple scaling and translation of the various intervals to $[-1, +1]$. To map x from, say $[l, r]$ to $[-1, +1]$ can be achieved by first translating the interval $[l, r]$ so that its midpoint coincides with the origin, and then scaling by an appropriate amount. Since the midpoint is at $(r + l)/2$ the translation is by $-(r + l)/2$, and since the original length is $(r - l)$ and the final length is 2, the scaling factor is $2/(r - l)$. We apply the same idea to each of the coordinates yielding two matrices, one for translation and one for scaling. In the case of the z-coordinate, we want to map $[n, f]$ to $[-1, +1]$, but sinze the z-coordinates are negated, we need to scale by a negative amount. It is easy to see that the resulting matrix is:

$$M_{ortho} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

(Note the -2 in the 3rd row and column, due to the negation of z.) The function `glOrtho()` simply computes this matrix and multiplies it by the current matrix.

See the book for handling of *oblique* (nonorthogonal parallel) projections. Basically the approach is the same, but a shear is applied based on the angles of the projection.

Perspective Transformation: Next consider the OpenGL call that defines a perspective projection

```
glFrustum(l, r, b, t, n, f);
```

which defines a viewing frustum whose near face (let's call this the *projection window*) is defined by

$$\begin{aligned} l \leq \quad & x & \leq r, \\ b \leq \quad & y & \leq t, \\ & z & = -n. \end{aligned}$$

and whose far face lies at $z = -f$. Plane $z = -n$ is the *near (or hither) clipping plane* and $z = -f$ is the *far (or yon) clipping plane*. This defines a viewing frustum.

In order to apply the perspective-with-depth transformation from last time, we will compose the following basic transformations:

Shear: Shear the viewing frustum so that the projection window is centered along the z-axis. The (x, y, z) coordinates of the center of the near face is

$$\left(\frac{r+l}{2}, \frac{t+b}{2}, -n\right)^T.$$

Thus the cotangents of the x and y shearing angles are $(r + l)/(2n)$ and $(t + b)/(2n)$, respectively. This yields the matrix

$$H = \begin{pmatrix} 1 & 0 & \frac{r+l}{2n} & 0 \\ 0 & 1 & \frac{t+b}{2n} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

It is easy to verify that after applying this, the lower-left and upper-right corners of the projection window are mapped to

$$\begin{pmatrix} -\frac{r-l}{2} \\ -\frac{t-b}{2} \\ -n \\ 1 \end{pmatrix} \quad \begin{pmatrix} \frac{r-l}{2} \\ \frac{t-b}{2} \\ -n \\ 1 \end{pmatrix}.$$

Scale: The transformation we presented last time was based on the assumption that the projection plane was located at $z = -1$. We need to scale z by $1/n$ to achieve this. After this the z-coordinates of projection window become -1. We will also want to scale the projection rectangle so that its x and y extents each range over the interval $[-1, 1]$. We do this by scaling x by $2/(r - l)$ and y by $2/(t - b)$, respectively. This yields the scaling matrix:

$$S = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{1}{n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Because scaling by an arbitrary nonzero constant does not change the effective homogeneous coordinates, we can apply a uniform scaling by n, by multiplying this matrix times

$$U = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & 0 & n \end{pmatrix}.$$

(The only reason I can see for doing this is for compatibility with the OpenGL specifications, but there may be good reasons to avoid dividing by n, for example if the user sets $n = 0$.)

Let $S' = SU$. After applying $S'H$, the corners of the projection rectangle (after normalization) are mapped to $(-1, -1, -1, 1)^T$ and $(1, 1, -1, 1)^T$ respectively. The viewing frustum is now centered along the $-z$-axis and has slopes of its faces are ± 1. The near clipping plane has been mapped to $z = -1$ and the far clipping plane has been mapped to $z = -f/n$.

Perspective Depth: Finally we apply the perspective depth transformation given last time. Recall that this matrix had two unspecified constants, α and β. To derive the desired values of α and β, recall that we want the interval $[n, f]$ to be mapped to the interval $[-1, +1]$. From the existing shearing and scaling transformations, this interval has been mapped to $[-1, -f/n]$. The perspective depth transformation maps z to $-\alpha - \beta/z$. This gives the following constraints to be satisfied:

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

Solving for α and β we have

$$\begin{aligned} \alpha &= -\frac{f+n}{f-n}, \\ \beta &= -\frac{2f}{f-n}. \end{aligned}$$

Thus this gives the perspective depth matrix:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2f}{f-n} \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

Finally we compute the product $MS'H$ resulting in the final general perspective depth projection matrix

$$M_{pers} = MS'H = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

The function `glFrustum()` computes this matrix and multiplies it by the current matrix.

Simpler Perspective Specification: The `glFrustum()` procedure is very general, but somewhat more general than is needed for many viewing situations. Typically, the projection rectangle is centered along the z-axis. When this is the case, there is a simpler OpenGL specification that can be used instead:

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

All arguments are of type `GLdouble`. This procedure defines a viewing frustum whose near and far clipping planes are at $z = -\text{near}$ and $z = -\text{far}$, respectively, whose projection window is centered about the z-axis, and has *aspect ratio* of `aspect`. The aspect ratio is the defined to be width/height for the window. To specify the size of the projection window, the argument `fovy` specifies the *y-field-of-view*, that is, the total angle in the yz-plane of the viewing frustum. For this particular case, the near and far values should be positive.

Viewport specification: Once the perspective transformation has been applied, the results are mapped onto the OpenGL graphics window on the screen. Normally the aspect ratio of the projection window should match the aspect ratio of the window, otherwise distortion will be noticeable. When the window is reshaped, the common approach is to reevaluate the projection to match.

Sometimes it is desirable to map the projected image to only a rectangular portion within the OpenGL window. This is done by specifying a *viewport*.

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

The (x, y) values indicate the lower-left corner of the viewport (in pixels). Width and height indicate the size of the viewport on the window. Unlike the previous operators, that were applied relative to world coordinates, this applies to window coordinates.

Lecture 14: Shading and Lighting Models

(Tuesday, March 18, 1997)
Read: Chapt. 6.

Lighting: One of the primary elements of realism, is the generation of realistic looking lighting and shading. The fact that many computer generated images look "computer generated" has to do with the fact that the human eye is very sensitive to interpreting surface characteristics from the subtleties of how light is reflected.

The lighting models that are used today in computer graphics are quite a bit better than what was available a decade ago. The most realistic global illumination models can rival photography for realism. These are called *global illumination models* because they take into account the interaction of light between the various surfaces of the scene. Howerver, they are enormously expensive to compute.

In contrast, *local illumination models* typically only consider the effect of light arriving directly from light sources, and use crude approximations for indirect lighting effects. The images are not as realistic, but in computer graphics, ground truth is often less important than being good enough to "fool the eye". An active area of research in computer graphics is how to produce the most realistic image for a given number of computational cycles.

The propagation of light is an extraordinarily complex physical phenomenon, and modeling it accurately is quite difficult. The main property of objects that determines their appearance is called *reflectance*. When light strikes an opaque surface it may be *absorbed* into the material or *reflected*. To simplify things, let us break light into its separate wavelengths, and consider each individually. Or, putting it in graphics terms, let us consider the red, green, and blue components of light individually. For each color, the intensity of the reflected light determines what we see. This intensity is a function of the intensity of the incoming light and the directions along which the light enters and leaves the surface of the object.

To achieve the most general lighting model, one needs to derive something called the *bidirectional reflectance distribution function* or *BRDF* for the object. This is a function, which given an incoming and outgoing direction, indicates the fraction of incoming light that is emitted along a certain differential reflection direction. For example, an object with a coarse surface finish (e.g. a piece of cloth) tends to scatter light equally in all directions, irrespective of the direction that it comes from. On the other hand, a purely reflective object (like a mirror) reflects almost the entire intensity along a single direction. Most objects lie somewhere between these two extremes.

Specifying such a function for each surface element in a scene would be an enormously complicated task. On top of this, there are other issues such as texture and bumpiness that make the derivation of such a function an extremely complicated task.

In order to achieve realistic rendering times and to make the task of surface and light modeling feasible, it is necessary to make some significant simplifications to arrive at a shading model that is simple enough work with. Today we will consider the most standard illumination model (the one used in OpenGL and most other real-time graphics packages). It applies well to smooth (i.e. nonbumpy) object surfaces that arise frequently from man-made objects.

Light Sources: Before talking about reflection, we need to discuss where the light originates from. Light sources come in many sizes and shapes. They may emit light in varying intensities and wavelengths according to direction. To simplify things, we will assume that light sources are points.

Light sources need not emit white light. They emit light according to a *luminance* function, which has three components (I_r, I_g, I_b) for the intensities of red, green, and blue light respectively. We will not concern ourselves with the exact units of measurement (typically watts or lumens, depending on whether you are considering radiometry or photometry).

In many graphics systems, OpenGL for example, it is common to extend this 3-element vector to a 4-element vector, where the last component is used to express certain "special effects". In OpenGL, color and intensities are typically expressed in terms of something called RGBA. The 4th (A) component is usually set to 1, but we will see examples where this value can be altered to achieve so special effects, such as transparency.

Lighting in typical environments involves a considerable amount of indirect reflection. In simple local illumination models, this is handled by breaking the light's intensity into two components, *ambient emission*, which is uniform in all locations and directions, and its *point emission*, which originates from a given point location. We may describe the light's intensity in terms of these two components:

$$I_a = \begin{pmatrix} I_{ar} \\ I_{ag} \\ I_{ab} \end{pmatrix} \qquad I(p_0) = \begin{pmatrix} I_r(p_0) \\ I_g(p_0) \\ I_b(p_0) \end{pmatrix},$$

where p_0 is the point indicating the location of the light source.

What is the intensity of light received at some point p? In the case of ambient intensity, it is always I_a, irrespective of the location of p. Light emitted from a point source is received at p only if there is no object blocking p from the light source. (Otherwise p is in the shadow of the source.) Normally, when dealing with area light sources, it is possible that p is only partially blocked from the light. The term *umbra* is the locus of points that are totally blocked, and *penumbra* is the locus of points that are partially blocked.

In addition to shadows, illumination is subject to *attenuation* due to the distance from the source. Physics tells us that the intensity of light falls off as the inverse square of the distance. This would imply that the intensity at some (unblocked) point p would be

$$I(p, p_0) = \frac{1}{|p - p_0|^2} I(p_0).$$

However, in graphics systems, our various simplifying assumptions (ignoring indirect reflections, for example) cause point sources to appear unnaturally dim using the exact physical model of attenuation. Consequently, in most systems (like OpenGL) it is common to use an attenuation function that has constant, linear, and quadratic components. The user specifies constants a, b and c, and if $d = |p - p_0|^2$, then the attenuation function is

$$I(p, p_0) = \frac{1}{a + bd + cd^2} I(p_0).$$

A light source can be placed infinitely far away by using the projective geometry convention of setting the last coordinate to 0. Thus, at high noon, the sun's coordinates might be

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

that is, a point at infinity along the positive y-axis. There is a performance advantage to using light sources at infinity. 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 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 book and the OpenGL function `glLight()` for further information.

The Phong Reflection Model: 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: To model objects that glow (even when all the lights are off) we can assign a certain amount of emitted illumination. 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 illuminate other objects.

Ambient reflection: A simple way to model indirect reflection. All surfaces in all positions and orientations are illuminated equally by this light intensity.

Diffuse reflection: The shading produced by dull (non-shiny) smooth objects.

Specular reflection: The bright spots appearing on smooth shiny (e.g. metallic or polished) surfaces.

An illumination model stores these basic elements in the surface description of every object. In the case of OpenGL, when these values are set through the `glMaterial()` call, and they apply to all objects that are defined until the values are changed.

The question that we are interested in is what is the amount of light that is transmitted (either by emission or reflection) from each point in the direction of the viewer.

The formulas that we will give will be stated in terms of *monochromatic* light, but you should really imagine that each involves 3 separate components, for red, green, and blue light. (The separation of color components is practically expedient, but not necessarily physically correct. For example, some surfaces act like prisms causing different wavelengths to scatter differently. In these cases, components of one primary can contribute to components of a different primary.)

In addition to the properties of the light sources and the surface material, the model is determined by the following four vectors which you can think of originating from the surface point whose reflectance we wish to compute. All these vectors are assumed to be normalized to unit length.

Normal vector: A vector $\vec{n}$ that is perpendicular to the surface at this point.

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

Light vector: A vector $\vec{l}$ that points towards the light source. (If there are multiple light sources, then we will repeat the shading computation for each light source individually, and sum the results.)

Reflection vector: A vector $\vec{r}$ that indicates the direction of pure reflection of the light vector. This vector lies in the same plane as the normal and view vector, and such that the angle of incidence equals the angle of reflection. We will discuss how to compute it from n and l later.

Let $L = (L_r, L_g, L_b)$ denote the intensity of the light source. OpenGL allows us to break this light's emitted intensity into 3 components: ambient L_a, diffuse L_d, and specular L_s. Each type of light component consists of 3 color components, so, for example, L_{rd} denotes the intensity of the red component of diffuse illumination.

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 behind this (that I know of). Thus, 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. We can think of these as being 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. 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

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

Warning: 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 not reflection, $LC = (0, 0, 0)$, and hence the object appears to be black.

As with lights, in OpenGL it is possible to assign separate components of color for ambient, diffuse, and specular reflection: C_a, C_d, C_s. Each of these is a RGBA vector. Again, I know of know justification for varying an objects color depending on the nature of the type of reflection, but OpenGL provides this capability. Typically these values are set equal to each other.

Next lecture we will see how the Phong illumination model combines all these elements.

Lecture 15: More Shading and Lighting

(Thursday, March 20, 1997)

Read: Chapt. 6. The material on computing normals by areas is not presented in our book, but is covered in Foley, et al.

The Phong Illumination Model: Last time we introduced the Phong illumination model. Let us consider it in greater detail. Recall that the model treated light illumination from each object as being composed of the following components:

Emission: To model objects that glow.

Ambient reflection: A simple way to model indirect reflection. All surfaces in all positions and orientations are illuminated equally by this light intensity.

Diffuse reflection: The shading produced by dull (non-shiny) smooth objects.

Specular reflection: The bright spots appearing on smooth shiny (e.g. metallic or polished) surfaces.

Relevant Vectors: The shading of a point on a surface is a function of the relationship between the viewer, light sources, and surface. The following vectors are relevant to shading. We can think of them as being centered on the point whose shading we wish to compute. All are assumed to be normalized to unit length.

Normal vector: A vector $\vec{n}$ that is perpendicular to the surface and directed outwards from the surface.

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

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

Reflection vector: A vector $\vec{r}$ that indicates the direction of pure reflection of the light vector.

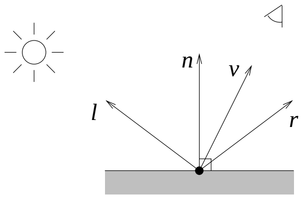


Figure 18: Vectors used in Phong Shading.

There are a few ways to compute normal vectors, depending on the representation of the underlying object. Before discussing the Phong model, let us consider this in greater detail.

Normals by Cross Products: 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 through a cross product:

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

The vector will be directed outwards if the triple $P_0P_1P_2$ has a counterclockwise orientation when seen from outside.

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.

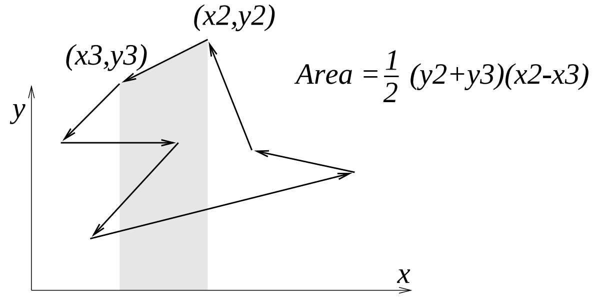


Figure 19: Area of polygon.

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

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

See the text for an example.

The Reflection Vector: The last issue is that of computing the reflection vector. This reduces to a little exercise in vector arithmetic. First observe that (because all vectors have been normalized to unit length) the orthogonal projection of $\vec{l}$ onto $\vec{n}$ is

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

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

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

Phong Lighting Equations: Let L_a, L_d and L_s denote the RGB intensities of the light source's ambient, diffuse and specular components. Recall from last time that L_d and L_s are typically equal, and L_a is scaled according to the amount of indirect reflection to be modeled. Let C_a, C_d, and C_s denote the RGB factors of the object's ambient, diffuse and specular components. Recall from last time that these are typically set equal to each other.

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

$$0 \leq k_a \leq 1$$

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

$$I_a = k_a L_a C_a$$

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

Diffuse reflection: Diffuse reflection arises from the assumption that light from any direction is reflectied 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 $\vec{l}$ 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.*

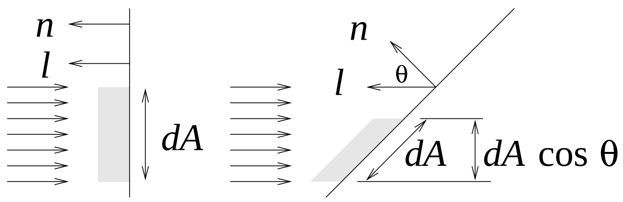


Figure 20: Lambert's Cosine Law.

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

$$I_d = k_d \max(0, \vec{n} \cdot \vec{l}) L_d C_d.$$

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 models of specular reflection, and some debate as to which one produces the most realistic results. The Phong model is one common model, but OpenGL uses another, called the *halfway vector model.*

Define $\vec{h}$ to be the *halfway vector*, the normalized vector which is halfway (i.e. bisects) $\vec{l}$ and $\vec{v}$. Thus, $\vec{h} = \text{Normalize}(\vec{l} + \vec{v})$. 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 is 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. (In the Phong model the corresponding factor is $(\vec{r} \cdot \vec{v})$ instead.)

The parameters of surface finish that control specular reflection are k_s, the surface's *coefficient of specular reflection*, and *shininess*, denoted α. As $\alpha > 1$ increases, the angle of specular reflection decreases, 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

$$I_s = k_s(\vec{n} \cdot \vec{h})^{\alpha} L_s C_s.$$

As with diffuse, this is subject to attenuation.

Combining everything: Putting this all together together with I_e the light emitted from an object, the total illumination of a point using our model is:

$$\begin{aligned} I &= I_e + I_a + \frac{1}{a + bd + cd^2}(I_d + I_s) \\ &= I_e + k_a L_a C_a + \frac{1}{a + bd + cd^2}(k_d \max(0, \vec{n} \cdot \vec{l}) L_d C_d + k_s(\vec{n} \cdot \vec{h})^{\alpha} L_s C_s), \end{aligned}$$

where d is the distance from the object to the light source. For multiple light sources, we add up the ambient, diffuse, and specular components for each light source.

Lecture 16: Review for Midterm

(Tuesday, April 1, 1997)

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. (See Chapt 1.)

OpenGL: You are responsible for knowing the basic semantics, but not the exact calling sequences for the OpenGL commands needed for Project 1, along with the commands for performing transformations and perspective. I am more interested in your knowledge of the basic capabilities of OpenGL, rather than the details of how to get something done. You should be familiar with the function calls made in the sample skeleton program (`skel.c`) along with the following:

Transformations: `glTranslate*()`, `glRotate*()`, `glScale*()` (see Section 4.9.2 in Angel and Lecture 8).

Viewing: `gluLootAt*()` (see Section 5.2.3 and Lecture 12).

Perspective: `glOrtho()`, `glFrustum()`, `glPerspective()` (see Lectures 12 and 13).

(See Chapts 2 and 3, a smatterings throughout the book.)

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 some applications of geometric programming, for example for computing areas of polygons, rotations about arbitrary vectors, ray-shooting, etc. (See Chapt 4 and Appendices B and C.)

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. In particular, the two matrices F.toStdFrame and F.fromStdFrame maps coordinate from F to the standard frame and from the standard frame to F, respectively. You should know how to construct these matrices. (This was not discussed in the text.)

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. (See Chapt 4.)

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. (Orthogonal projections are 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) that 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. (See Chapt 5.)

Shading and Lighting Models: We introduced a standard local shading model called the *Phong illumination 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. (See Chapt 6, through to 6.4.3).

Lecture 17: Midterm Exam

(Thursday, April 3, 1997)

Midterm exam. No lecture today.

Lecture 18: Lighting and Viewing in OpenGL

(Tuesday, April 8, 1997)

Read: Sections 6.7 and 6.8 for information on lighting.

Motion Simulation: The next project is a simple 3-d motion simulator. This is a very common application in computer graphics (including many video games, driving simulations, flight simulation, and virtual-reality systems). In any motion simulation the graphics rendering process is complicated by the fact that the viewer is constantly moving, so the perspective and viewing transformations need to be updated with each display cycle.

The main question is how do we put together the various elements of OpenGL relating to viewing and transformation to achieve the desired effects? During each idle event you should do two things, first advance the simulation, and then request that the image be updated (`glutPostRedisplay()`.

Initialization: Initialization involves all the things that were used in the first project (initializing GL, setting up the event callbacks, for example). When you initialize the display mode `glutInitDisplayMode()` be sure to add depth buffering in addition to RGB color and double buffering

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

You will/may also want to enable a number of GL options using `glEnable()`. These include:

GL_DEPTH_TEST: Perform depth buffering for hidden surface removal. (Required.)

GL_CULL_FACE: Remove faces on the back sides of objects. (Optional: for efficiency in rendering, but you do not see the insides of the clouds.)

GL_NORMALIZE: Normalize normal vectors automatically. You will probably not need this unless you perform transformations other that rotation and translation. The reason for needing this option is that normal vectors are not automatically scaled to unit length before lighting computations are performed. If you define normals of unit length, but then you scale the object, the normals will be scaled as well (actually by the inverse transformation). This may cause their lengths to change.

Others: Including lighting, texture mapping, and fog. We will discuss these later.

In addition you will need to initialize the lighting, fog and texture mapping (for 828M students), and enable a number of other options. We will discuss these later. I also enabled backface culling (for greater efficiency in hidden surface removal),

Advance: To advance the state of the simulation, you need to move the viewer and any mobile objects (i.e. the two ships) incrementally. This involves determining the amount of translation and rotation. Note that you cannot make use of OpenGL's transformations functions for this; they can only be used for transforming objects for drawing. However, because of the limited nature of the motion, you will not really need the full capabilities of matrix multiplication. Each ship object is associated with its current position, its current heading, and its current translational and rotational velocities. I'll leave the geometric details to you.

After advancing, the next issue is redisplaying everything.

Display: Begin by clearing the frame buffer. For this project, this will require that clearing both the color and depth of the frame buffer. (Using `GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT`). The drawing process breaks up into the following components:

Perspective: First set the perspective. Based on the location of the current ship, invoke `gluLookAt()` to set the camera location. Then, change the matrix mode to `GL_PERSPECTIVE`, invoke `gluPerspective()` to set the viewing perspective. For grad students I have stated that the x-field of view should be 60 degrees, and you should perform the necessary trigonometry to determine what to pass to `gluPerspective()` for the y-field of view (based on the current window aspect ratio). Then set the matrix mode back to `GL_MODELVIEW`.

For undergrads, the y-field of view is around 47 degrees, assuming you use the 400×300 window I suggested.

Light positions: Although you can set up many of the light properties in the initialization, the light position must be transformed according to the camera position. We will discuss how to do this later.

Draw objects: Drawing objects generally involves the following steps:

Material: Set the material properties of the current object. These remain in effect until they are changed.

Local transformations: It is often convenient to transform into a convenient frame before drawing. For example, the routine for drawing a sphere `glutSolidSphere()` draws a sphere only about the origin. To draw an object about an arbitrary point and at an arbitrary orientation, the approach would be:

```
glPushMatrix();            (save the current matrix)
glTranslatef(...);         (transform to convenient frame)
glRotatef(...);
...draw as if object located at origin...
glPopMatrix();             (restore transformation)
```

Draw object: Issue the appropriate drawing commands. For complex objects like the sphere and torus, you can invoke the appropriate glut procedure. For example, to draw a torus centered at the origin with axis along the z-axis use:

```
glutSolidTorus(innerRad, outerRad, nsides, nrings);
```

The arguments are illustrated in the following figure.

Normals: In addition to defining vertices, you need to define surface normals when lighting is being used. Procedures like `glutSolidSphere()` and `glutSolidTorus()` define their own normals. If you define your own object, say using `glBegin()..glEnd()` you will need to assign normals to each vertex for accurate lighting. This is done by calling `glNormal()` prior to defining the vertex. The normal remains in effect for all vertices, until you define a new one. Thus, if a number of vertices have the same normal, then you can issue one `glNormal()` call for all of them.

For this project, the main objects to draw are the ground, spheres and ships.

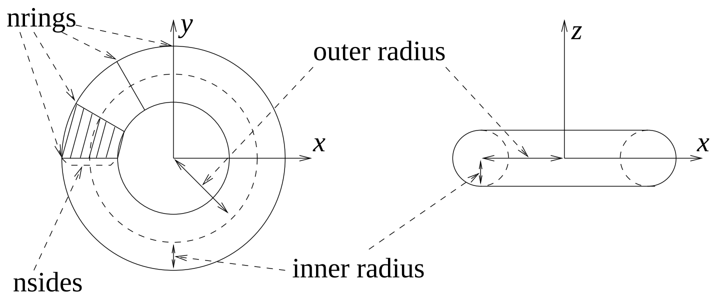


Figure 21: glutSolidTorus parameters.

Lighting in OpenGL: To describe lighting in OpenGL there are three major steps that need to be performed.

Ligting model: Define the parameters of the general lighting model. There are a number of parameters that can be set here, but the main one for this project is

```
glShadeModel(GL_FLAT) (or GL_SMOOTH)
```

This sets up smooth or flat shading. There is a function `glLightModel*()`, which is used for setting other global lighting parameters (such as the global ambient light). However, in the project I let them all go to their default values.

Create/Enable lights: OpenGL allows 8 different light sources. For each source you can set its position and its ambient, diffuse, and specular intensities. (Note: There is an error in the handout: light sources do not have `GL_EMISSION`, since they emit light by definition. This should have been listed under surface material properties instead.)

This is done through the procedures `glLightf()` and `glLightfv()`.

```
glLightf(GLenum light, GLenum pname, GLfloat param);
glLightfv(GLenum light, GLenum pname, const GLfloat *params);
```

Each procedure is given the name of a light `GL_LIGHTi` where i ranges from 0 to 7, the name of a property (e.g., `GL_DIFFUSE`), and the property itself. For vector properties (such as color specification), you need the "fv" form of the command. The the last argument to the procedure is the 4-element array of RGBA values. (The "A" value is typically set to 1.0.) See the text for more information.

When the lighting parameters have been set up, the light only becomes lit, once you enable it, e.g. using

```
glEnable(GL_LIGHT0);
```

Define surface materials: When lighting is used you do not define the "color" of each object, as we did in the first project. Rather you define its surface material properties, and OpenGL computes the color for you.

It may seem odd that surface properties are not assigned to faces, but rather the individual vertices. OpenGL takes the surface properties associated with each vertex and applies the to the face. In flat shading it just takes the first vertex of the object and assume that its normal vector applies to the entire face. In smooth shading, it computes the shading at the corner points are performs some sort of smooth interpolation across the shape.

Beware of using very large polygons that are very close to the viewer (e.g., the ground in the project) since the vertices are so far apart relative to the viewer that the interpolation may not yield accurate results. When this is a concern, these polygons can be broken up into smaller pieces.

Recall from the Phong model that surfaces are associated with colors and coefficients for each of emission, ambient, diffuse, and specular reflections. In OpenGL, these are combined into a single value RGBA value. For example, if you had a bright red object with the RGBA color $(1, 0, 0, 1)$, and you wanted its diffuse component to be scaled down, then you would just scale down this color, e.g., $(.5, 0, 0, 1)$, and give it to OpenGL as the diffuse color of the object.

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 front face is generally the one from which vertices appear to be in counterclockwise order. The first argument indicates which face we are coloring front, back, or both. The second argument indicates the parameter name (for emission, ambient, diffuse, or specular), and the last parameter is the value. See the text for more information.

Lecture 19: Surface Mappings

(Thursday, April 10, 1997)
Read: Sections 10.1 through 10.4 in Angel.

Surface Detail: Today we discuss a topic that would normally appear later in the semester, but because of its relevance to the current project we consider it now. We have discussed how to provide some element of surface realism through lighting and shading. 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.

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 (such as grass the rocky texture on a mountain side). The quick and dirty way is to digitize an image of such a texture, and then to map this texture onto the surface of the object, as you might apply wallpaper to the wall of a room. This is called texture mapping.

Texture mapping: The process of "wrapping" a texture image around an object is called *texture mapping*, and it is one of the most popular techniques for creating realistic images quickly.

Suppose that we are given a digitized image stored in some graphics file format. Such formats include GIF (graphics interchange format), JPEG (joint photographic experts group), or PPM (potable pixel map). Different formats offer different tradeoffs between the degree of compression, color fidelity, and simplicity. We will not discuss the internals of particular formats.

We may input such an image, and store it in a large 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 usually row by row with the upper left corner as the origin. The individual RGB pixel values of the texture image are called *texels*.

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. We can achieve this through a simple scaling of the parameters:

$$T(s,t) = Im[\lfloor(1-t)N\rfloor][\lfloor sN\rfloor],$$

where Im is the 2-dimensional image array. (I am not being careful in the cases where $t=0$ or $s=1$, since these will generate subscripts out of bounds, but let's not worry about this for now.) In some cases it is more convenient to define 3-dimensional textures (e.g. to model the grain in wood, which is naturally formed from concentric cylinders) or even a 4-dimensional projective texture (e.g. to model a texture which is projected onto a surface). But 2-dimensional textures are the most common, so we consider them.

We wish to "wrap" this 2-dimensional texture 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 (x,y,z) in 3-space to a point (s,t) in texture space. It is often more common to take an indirect route. First, we construct a 2-dimensional *parameterization* of the surface. That is, we associate each point on the object surface with two coordinates (u,v) in the *surface space*. The *inverse wrapping* function $IW(u,v)$ maps a point on the surface to a point (s,t) in texture space. This is an "unwrapping" function, since it unwraps the surface back to the texture, but as we will see, this is what we need.

Suppose that the unwrapping function IW, and a parameterization of the surface are known. Here is an overview of the texture mapping process. First we consider a pixel that we wish to draw. We determine the *fragment* of the object's surface that projects on to this pixel, by determining which points of the object project through the corners of the pixel. (Most graphics systems, OpenGL included, are rather sloppy in doing this. They may represent a fragment by a single point on the surface, e.g. the projection of the center of the pixel.) 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.) We compute the surface space parameters (u,v) for each of the 4 corners of the fragment. Then we apply the unwrapping function to determine the corresponding region of texture space. Finally we compute the average intensity of this region of texture space, by computing a weighted sum of the texels that overlap this region, and assign the corresponding blended color to the pixel. (Again in OpenGL, a quicker but less accurate approach is used. The center of the pixel is projected into surface space, and is then transformed to texture space. A weighted average of the 4 nearest texels are used for the final RGB value.)

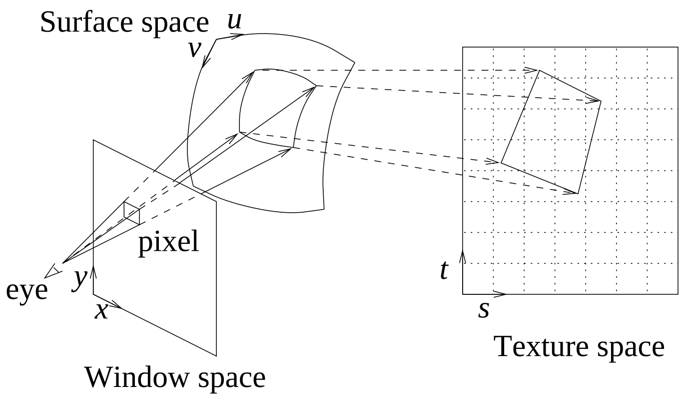


Figure 22: Texture mapping.

The main question is how are these two functions, the surface parameterization and the inverse wrapping function, defined? The surface parameterization depends on the nature of the surface.

For example, if you are modeling a sphere then we can represent any point on a sphere with two angles, representing the point's latitude and longitude. For example, if the sphere is located at the origin and has radius r, and if we think of the z-axis as being the north pole, and the 0-longitude passing through the x-axis, then any point on the sphere can be expressed by two angles, ϕ and θ. Think of the vector from the origin to the point on the sphere. Let ϕ denote the angle between this vector and the z-axis (north pole). Thus $0 \le \phi \le \pi$. Let θ denote the CCW angle of the projection of this vector onto the xy-plane. Thus $0 \le \theta \le 2\pi$. Then it follows that the point on the sphere has the coordinates

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

We can let $u = \phi/\pi$ and $v = \theta/(2\pi)$.

In OpenGL, all objects are polygons. If we are given a triangle, then we can use any 2 of the barycentric coordinates as a 2-dimensional parameter system. This leads to a linear interpolation function.

If we are given a convex quadrangle (e.g. formed from `GL_QUADS`) then the most common approach is to use *bilinear interpolation*. Consider a quadrangle whose corners are $P_{00}, P_{01}, P_{10}, P_{11}$. We can associate any pair (u, v) with a point in the quadrangle as follows. Using u compute an affine combination of P_{i0} and P_{i1}:

$$\begin{aligned} P_0(u) &= (1-u)P_{00} + uP_{01}, \\ P_1(u) &= (1-u)P_{10} + uP_{11}. \end{aligned}$$

Then using v take an affine combination of these two points

$$\begin{aligned} P(u,v) &= (1-v)P_0(u) + vP_1(u) \\ &= (1-v)((1-u)P_{00} + uP_{01}) + v((1-u)P_{10} + uP_{11}) \\ &= (1-v)(1-u)P_{00} + u(1-v)P_{01} + (1-u)vP_{10} + uvP_{11}. \end{aligned}$$

(I am not sure what OpenGL does for polygons with more than 4 sides. One possibility would be to triangulate the polygon and then apply individual linear interpolations.)

In both cases we have shown how to map coordinates onto points in 3-space. In practice, what is needed is to map the 3-dimensional coordinates onto the surface parameter space. This is done by the (sometimes messy) task of inverting these functions.

The second task is to compute the inverse wrapping function. Once we have determined the surface space parameters of the object, this is often accomplished by a very simple scaling transformation. In OpenGL the method that is used is to specify the texture coordinates for each vertex of the polygon. Then we can use the surface parameters (u, v) of the point to perform either a linear or bilinear interpolation into texture space.

Texture mapping in OpenGL: In OpenGL, then main things that need to be done for texture mapping are to input the RGB values of the texture into an array. Then call the procedure `glTexImage2D()` to have the texture processed into OpenGL's internal format. (The procedure has an incredible array of options. See the example in Section 10.2.2 for a typical no-frills calling sequence.)

Next use `glTexEnvf()` to specify how the image is to be mapped. Is it just to be painted on (`GL_DECAL`) or blended in some manner with the existing colors (`GL_MODULATE` or `GL_BLEND`). Then just prior to drawing a texture mapped object, enable texture mapping with

```
glEnable(GL_TEXTURE_2D);
```

Remember to turn it off afterwards if it is not desired. The last step is to specify the texture coordinates for each vertex of your texture mapped object, by calling `glTexCoord2f()` just prior to drawing the vertex.

Lecture 20: More Surface Mappings

(Tuesday, April 15, 1997)
Read: Sections 10.1 through 10.4 in Angel.

Reflection Mapping: Last time we presented texture mapping as a method for providing surface detail to objects. Today we consider some variants of this idea to other sorts of surface details.

First, suppose that you are looking at a shiny waxed floor, or a metallic sphere. We can model the shininess by setting a high coefficient of specular reflection, 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. 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 very computationally intensive technique. People who need fast rendering times, but are willing to cheat a little on accuracy, use an alternative method, called *reflection mapping* (also called *environmental 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.

How can we encode such a complex 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. Computing the reflection vector is analogous to what we did in the Phong model (but there we computed the refection from the light source). For a physically accurate solution, this would involve computing the reflection ray and shooting it through the environment to see what is hit. If we hit a red object, then the viewer sees red at this point. (To verify this, just trace the ray backwards. Reflections work the same in either direction.)

Unfortunately, it is expensive to shoot rays through 3-dimensional environments to determine what they hit. Typical graphics environments may involve thousands to millions of polygonal elements, and this process is slow even with very efficient data structures. We would like to do what we did in texture mapping, and just look the answer up in a precomputed image. However this is not possible, unless we make one very strong simplifying assumption.

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

Without this assumption it is still possible to perform reflection mapping, but the applicability and accuracy of the results will degrade. This is not really a reasonable assumption in many instances, but the remarkable fact is that our eye is not very good at checking the accuracy of reflections. So the eye is usually fooled into thinking that this is an honest reflection.

The reason that this assumption is important is that the main parameter in determining what the reflection ray hits is its *direction* of the reflection ray, and not the spot on the surface that the ray starts from. 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.

Here is how reflection mapping is actually implemented. First off remove the reflective object from your environment. Place a small sphere or cube about the center of the object (small enough that it does not intersect any of the surrounding objects). You may use any convex object, but these seem to be the simplest to work with. We will discuss the case of a cube below. Project the entire environment onto the 6 faces of the cube, using the center of the cube as the center of projection. That is, take 6 separate pictures which together form a complete panoramic picture of the surrounding environment, and store them in 6 image files. It takes some time to compute these images, but once they have been computed they can be reused from all different viewing positions.

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 n, view vector v, etc. We compute the view reflection vector vr from these two. To determine the reflected color, we use the vr vector to identify the appropriate side of the cube (from the coordinate with the maximum absolute value) and a point on this side of the cube indicated by this vector. We make this the color of the object at this point.

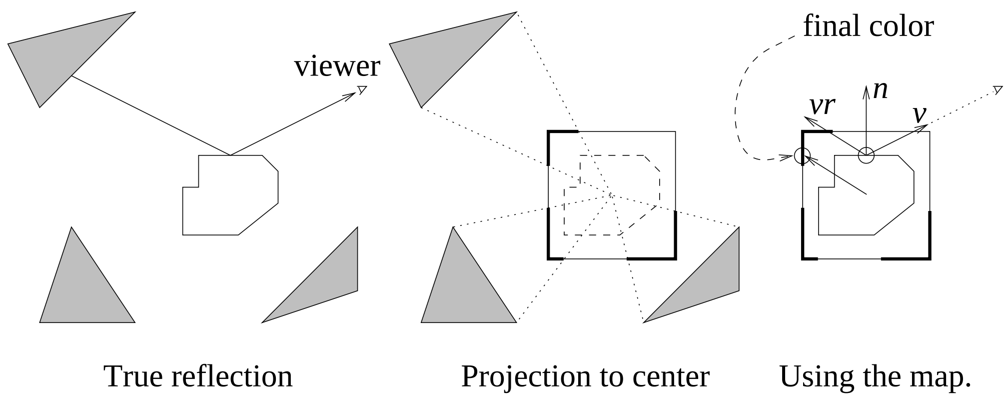


Figure 23: Reflection mapping.

Reflection mapping through texture mapping: This description of reflection mapping assumes that you have access to the view reflection vector at each point. If you are using a system like OpenGL to do your rendering, this information is hidden down in the implementation. However, it is still possible to handle this by reducing the problem to a texture mapping problem. The catch is that the viewpoint must be known in advance.

Suppose that you know your viewpoint in advance. Then for each point on the surface of your object you can compute the reflection vector vr, and given vr you can determine the corresponding point in your image. This point becomes the corresponding "texture coordinates" for your texture map.

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 current lighting. As we move our viewpoint from side to side, the specular reflections from the bumps move as well. However, texture mapping alone cannot model this sort of effect.

What is the underlying reason for this effect? It turns out that 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 edge of the object will still appear to be perfectly smooth). But in fact, bump mapping produces remarkably realistic bumpiness effects, especially when combined with texture mapping. (There is a variant of bump mapping, called *displacement mapping*, in which not just the normals, but the actual surface is displaced, but this affects depth-buffering and so is a more expensive option.)

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.

Here is how bump mapping is performed. As with texture mapping, there is an equivalent to texture space, called the *bump space*, and a separate object space. We are given a *bump displacement function* $b(s,t)$, which maps a point (s,t) in bump space to a displacement. Think of $b(s,t)$ as being the height of the bump at this location (it is a scalar quantity). The function $b(s,t)$ may be given as an equation, or taken from an image file as in texture mapping. As with texture mapping, there is an *inverse wrapping function* IW which maps a point (u,v) in the object's surface space to (s,t) in bump space.

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 N denote the surface normal 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* 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 illumination computations.

Here is a simple approximation for 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.

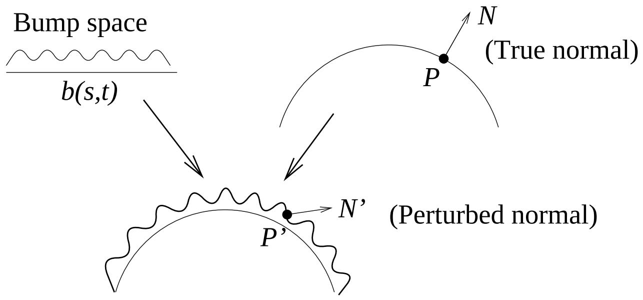


Figure 24: Bump mapping.

The derivation of N' makes use of the fact from differential geometry that if P_u denotes the partial derivative of P with respect to u, and P_v denotes the partial derivative of P with respect to v, then

$$N = P_u \times P_u.$$

Using our parametric representation of the surface, we can compute these as

$$P_u = \begin{pmatrix} \partial x(u,v)/\partial u \\ \partial y(u,v)/\partial u \\ \partial z(u,v)/\partial u \end{pmatrix} \qquad P_v = \begin{pmatrix} \partial x(u,v)/\partial v \\ \partial y(u,v)/\partial v \\ \partial z(u,v)/\partial v \end{pmatrix}.$$

(Recall from calculus that the partial derivative is computed by treating one parameter or the other as being constant.) N may not generally be of unit length, so we define $\hat{N} = N/|N|$ to be the normalized normal.

Clearly, if we apply the 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. We want to know what the surface normal should be at P'. But this requires that we know its partial derivatives with respect to u and v. That is,

$$N' = P'_u \times P'_v,$$

where P'_u is the partial of P' with respect to u and P'_v is the partial of P' with respect to v. We have

$$\begin{aligned} P'_u &= \frac{\partial}{\partial u}(P + b\hat{N}) \\ &= P_u + b_u\hat{N} + b\hat{N}_u. \end{aligned}$$

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

$$\begin{aligned} P'_u &\approx P_u + b_u\hat{N} \\ P'_v &\approx P_v + b_v\hat{N} \end{aligned}$$

Taking the cross product we have

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

Since $\hat{N} \times \hat{N} = 0$ we have

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

The partials 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

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

If we have a symbolic representation for $P(u, v)$ and $b(s, t)$, then these partial derivatives can be computed. 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 a symbolic representation, but we can approximate the derivatives by taking finite differences.

So how do we use this? We proceed as we would in any normal lighting computation, but instead of using N as our normal vector, we use N', the perturbed normal, instead. It is the use of N' that creates the desired bumpy impression.

Lecture 21: Hidden Surface Removal

(Thursday, April 17, 1997)
Read: Section 7.7 in Angel.

Implementation Issues: Up to now in our top-down coverage of computer graphics, we have seen the basics of generating a perspective image of a 3-dimensional scene and how to determine the color of points based on lighting and texture considerations. There are a number of low-level implementation issues that we have not discussed. These include the following:

Clipping: How are lines and polygons that are not visible in the view-frustum clipped away?

Hidden surface remove: How do we determine which surfaces are visible to the viewer, and which are hidden from those in front?

Scan conversion: How do we digitize lines, text, and polygons on the screen, as an collection of pixels?

Antialiasing: How do we avoid jagged lines and smooth out undesirable discontinuities in the resulting digitization process.

Dithering: If the monitor cannot support full 24-bit true color, how do we approximate colors?

We will spend some time discussing some of these issues over the next few lectures, but not necessarily in this order. Today we consider the hidden-surface removal problem.

Hidden-Surface Removal: Some authors refer to this as *visible-surface determination*. 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). 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. The first is that we generate the set of *pixels* that form the final image. The second is that we generate a collection of *polygons* in the plane, whose interiors do not intersect, and which together form the final image. The fastest algorithms usually operate by the former approach (since we have to perform scan conversion anyway, we might as well combine it with hidden-surface removal). However, the advantage of the latter approach is that the problem can be solved exactly, independent of the resolution of the final image, and hence can be scaled and transformed. We will consider both instances.

The hidden-surface elimination problem is interesting from the perspective of algorithm design, because it is an example of a problem that is fairly 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, 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 no one ever uses.)

Back-face culling: One simplest method for removing roughly half of the faces from consideration is to observe that, 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

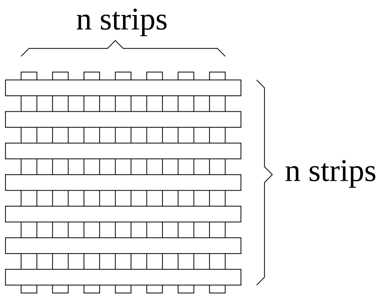


Figure 25: Worst-case example for hidden-surface elimination.

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 1/2 of the faces from further consideration.

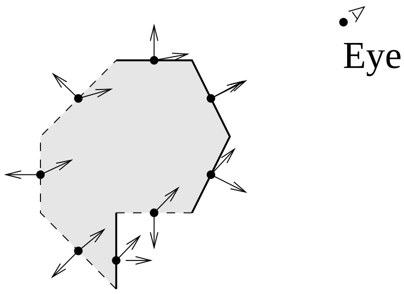


Figure 26: Back-face culling.

In OpenGL, the command `glEnable(GL_CULL_FACE)` can be used to enable this test.

Depth-Sort Algorithm: A fairly simple hidden-surface algorithm is called the depth-sort algorithm. It is based on the following simple principle. If we scan convert all the polygons, with more distant polygons first and closer polygons later, the closer ones will hide the more distant ones. This suggests the following algorithm: 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).

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 nearest point to the viewer), project these points saving the $-1/z$ distance information we used in our transformation, and then sort by decreasing order of distances, and draw in this order.

Unfortunately, just because the centroids or nearest points are ordered, it does not imply that the entire polygons are ordered. Worse yet, it may not be possible to order polygons, as shown in the following figure.

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 much what order they are drawn in.

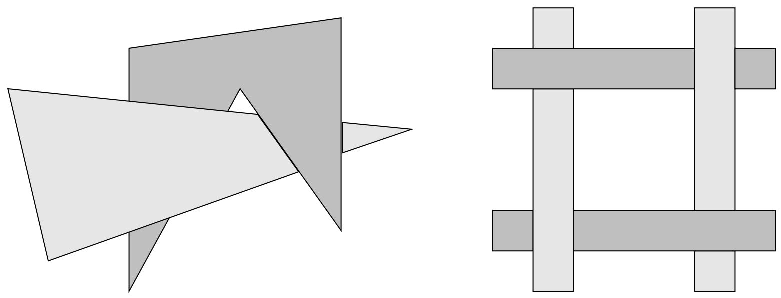


Figure 27: Hard cases to depth-sort.

Here is a snapshot of one step of the depth-sort algorithm. We begin by sorting the polygons by depth. For example, the initial "depth" estimate of a polygon may be taken to be the closest z value of any vertex of the polygon. Let's take the polygon P at the end of the list. Consider all polygons Q whose z-extents overlaps P's. Before drawing P we make the following tests. If any test is passed, then we can assume P can be drawn before Q.

(1) Do the x-extents not overlap?

(2) Do the y-extents not overlap?

(3) Is P entirely on the opposite side of Q's plane from the viewpoint?

(4) Is Q entirely on the same side of P's plane as the viewpoint?

(5) Do the projections of the polygons overlap not overlap?

If all the tests fail then we split either P or Q using the plane of the other. The new cut polygons are inserting into the depth order and the process continues.

In theory this partitioning could generate $O(n^2)$ individual polygons, in practice the number of polygons is much smaller.

Depth-buffer Algorithm: The depth-buffer algorithm is one of the simplest and fastest hidden-surface algorithms. Its main drawback is that it requires a lot of memory. 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. All that needs to be added is interpolation of depth values. Consider the process of scan-converting a triangle shown in the figure below.

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 $P_i = (x_i, y_i, z_i)$

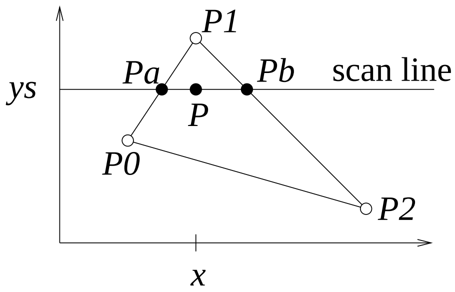


Figure 28: Depth-buffer interpolation.

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.

Although we have not yet discussed it in detail, 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, is just the affine combination

$$z_a = (1 - \rho_a)z_0 + \rho z_1.$$

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

$$z = (1 - \alpha)z_a + \rho z_b.$$

Lecture 22: Ray Tracing

(Tuesday, April 22, 1997)
Read: Section 6.10.1 in Angel. Most of the material in today's lecture is not covered in the text.

Ray Tracing: Ray tracing is among the simplest methods (but computationally intensive) for achieving realistic images. Furthermore, ray tracing forms the basis of many approaches to more complex types of image generation (for example, it is used in Monte Carlo radiosity).

The Basic Idea: Imagine that you are taking a picture of a scene, and you want to determine the color of a point on the projection plane. The color of this point is determined by the light ray that passes through this point and hits the eye. Light leaves a light source, and hits an object. Some its energy is absorbed. Some is reflected in different directions. If the object absorbs mostly blue and green light, then the reflected light will contain mostly red components of light. The light may continue to be reflected off of objects. It may hit transparent objects, and be refracted. We see the light rays that finally enter our eye.

If we could accurately model the movement of light around a 3-dimensional scene then we could produce very stunningly accurate images. But the time to do this would be prohibitive.

The main problem is that most of the light rays that leave the light sources never even hit our eye. Consequently the vast majority of the effort is wasted.

Ray tracing is a method that is based on correcting this problem by *reversing* optics. Imagine that the viewer is holding a wire screen in front of his eye. Each grid square in the screen corresponds to a pixel to be rendered in the final image. We shoot rays out from the eye through the center of each grid square (in an attempt to model the light coming in from this ray).

Consider the first object that this ray hits. We want to know the intensity of reflected light at this point. To do this, we need to know which light sources are visible from here. Shoot a ray from the contact point of the ray with the object to each of the light sources (which we will assume to be point light sources for simplicity). Each of these rays that succeeds in reaching a light source before hitting another object is illuminated by this light source.

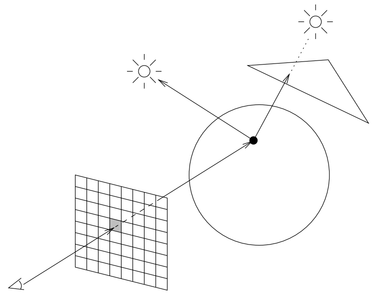


Figure 29: Ray Tracing.

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. We use this to color the pixel. We simply repeat this operation on all the pixels in the grid, and we have our image. All that we needed to make this process work are the following things:

- A model of our 3-dimensional scene.
- A procedure that traces a ray through the scene and determines the first object that was hit.
- Some procedures for computing the Phong illumination model (and implicitly utility routines for computing things like reflection vectors).

Reflections and refractions: Already this is better than what OpenGL does (because OpenGL does not compute shadows, and implicitly this procedure does by checking whether a light source is visible to a point).

However, this description is not fully correct. The main problem is that this description has made the simplifying assumption that the light that illuminates a point on an object arrives directly from the light sources. This ignores the possibility that light might be arriving indirectly from other objects in the scene. Since indirect illumination could arrive from anywhere in the scene, this is quite difficult to model (e.g., where should we shoot the next ray?). Typically the ambient light component of the Phong model is used to model indirect reflections.

A nice feature of ray tracing is that we can model highly reflective objects (such as mirrors, shiny sphere) and transparent objects (glass balls, rain drops). In particular, in addition to the traditional ambient, diffuse, and specular components of a point's color, let us also consider adding a reflective and transparent component.

To model the reflective component, each object will be associated with a coefficient of reflection, k_r. As with the coefficients of ambient, diffuse, and specular reflection, this is a number in the interval $[0, 1]$. If the coefficient is nonzero, then we compute the view reflection ray (which equalizes the angle between the surface normal and the view vector) and shoot this ray. Let $\vec{u}$ denote the unit vector that points backwards along the viewing ray. (This is analogous to the view vector used in the Phong model, but it may not point directly back to the eye because of reflections.) Following the same formula that we used for computing the light reflection vector the view reflection vector, denoted $\vec{ur}$, is

$$\vec{ur} = 2(\vec{n} \cdot \vec{u})\vec{n} - \vec{u}.$$

(This is already normalized.) We take the ray along this direction and apply the ray-tracing algorithm recursively. (To avoid infinite looping, it is common to have a maximum recursion depth, after which a default background color, e.g. black, is returned.) When the reflected ray-trace color is returned, it is factored into the color exactly as the other colors were in the Phong model.

To model refraction, sometimes called *transmission*, we need to associate each surface with two additional parameters, the *indices of refraction* 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) where

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i}.$$

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.

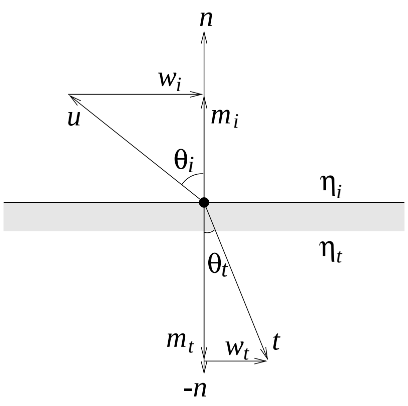


Figure 30: Refraction.

Let us work out the direction of the transmitted ray from this. As before let $\vec{u}$ denote the unit vector directed backwards along the incident ray. Let $\vec{t}$ denote the unit vector along the transmitted direction. The orthogonal projection of $\vec{u}$ onto $\vec{n}$ is

$$\vec{m}_i = (\vec{u} \cdot \vec{n})\vec{n} = (\cos\theta_i)\vec{n}.$$

Consider the two parallel horizontal vectors $\vec{w}_i$ and $\vec{w}_t$ in the figure. We have

$$\vec{w}_i = \vec{m}_i - \vec{u}.$$

Since $\vec{u}$ and $\vec{t}$ are each of unit length we have

$$\frac{\eta_t}{\eta_i} = \frac{\sin\theta_i}{\sin\theta_t} = \frac{|\vec{w}_i|/|\vec{u}|}{|\vec{w}_t|/|\vec{t}|} = \frac{|\vec{w}_i|}{|\vec{w}_t|}.$$

Thus,

$$\vec{w}_t = \frac{\eta_i}{\eta_t}\vec{w}_i = \frac{\eta_i}{\eta_t}(\vec{m}_i - \vec{u})$$

The projection of $\vec{t}$ onto $-\vec{n}$ is $\vec{m}_t = -(\cos\theta_t)\vec{n}$, and hence $\vec{t}$ is the sum

$$\begin{aligned}
\vec{t} &= \vec{w}_t + \vec{m}_t \\
&= \frac{\eta_i}{\eta_t}(\vec{m}_i - \vec{u}) - (\cos\theta_t)\vec{n} \\
&= \frac{\eta_i}{\eta_t}((\cos\theta_i)\vec{n} - \vec{u}) - (\cos\theta_t)\vec{n} \\
&= \left(\frac{\eta_i}{\eta_t}\cos\theta_i - \cos\theta_t\right)\vec{n} - \frac{\eta_i}{\eta_t}\vec{u}.
\end{aligned}$$

We have already computed $\cos\theta_i = (\vec{u}\cdot\vec{n})$. We can derive $\cos\theta_t$ from Snell's law and basic trigonometry:

$$\cos\theta_t \;=\; \sqrt{1-\sin^2\theta_t} \;=\; \sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2\sin^2\theta_i} \;=\; \sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2(1-\cos^2\theta_i)}.$$

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 why you can see a rainbow.) When this happens, set $\vec{t} = \vec{ur}$, the view reflection vector.

Illumination Equation Revisited: So we can combine the Phong illumination model with the reflection and refraction computed above. Let us consider an illumination model which is similar, but somewhat simpler than the model that we presented for OpenGL (but in which reflection and refraction are considered). 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. (Unlike OpenGL, individual lights do not have their own ambient light intensities.)
- **Visibility of Light Sources:** The function $\mathrm{Vis}(P, i)$ 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. When transparent objects are present this is considerably harder (since we need to consider all the refracted light beams that hit the light source). A simplifying (but unrealistic) assumption is that transparent objects never block illumination light (or that they always do).

Material color: We assume that an object's ambient and diffuse material colors are both the same, and 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 k_a, k_d, and k_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, half-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, c, as before. Let d_i denote the distance from the contact point P to the ith light source.

Reflection and refraction: Let k_r and k_t denote the reflective and transmitted (refracted) coefficients of illumination. Let $\vec{ur}$ 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 illumination equation is:

$$\begin{aligned} I &= k_a L_a C + \sum_i \mathrm{Vis}(P,i) \frac{L_i}{a + bd_i + cd_i^2} [k_d C \max(0, \vec{n} \cdot \vec{l}) + k_s \max(0, (\vec{n} \cdot \vec{h}))^\alpha] \\ &\quad + k_r \mathrm{trace}(P, \vec{ur}) + k_t \mathrm{trace}(P, \vec{t}). \end{aligned}$$

Note that if k_r or k_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 (since they already carry their own light intensities), and they are not attenuated (for the same reasons described on the midterm problem).

Lecture 23: Intersections and Clipping

(Thursday, April 24, 1997)
Read: Sections 7.3 through 7.6.

Rays and Intersections: The main question that we left unresolved yesterday is how to actually perform ray intersections. We will discuss this today. While we are at it, we will also consider the related issue of line and polygon clipping. This latter issue is not used in ray tracing, but the basic geometric processing and mathematical issues are the same.

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 $t > 0$ as $P + t\vec{u}$. It is a good idea to store t as part of the representation, so that the ray is actually a "trimmed ray". (Initially t is set to some huge value, e.g. `MAXFLOAT`.) Ray intersection procedures compute the value of t at which the ray intersects the object. If this is smaller than the current t value, then the new value is used. Otherwise the trimmed ray does not intersect the object.

Sphere intersection testing: Earlier this semester we considered the question of determining whether a ray in the plane intersects a circle. The test is easy to generalize for testing whether a ray intersects a sphere in 3-space.

As above, 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).

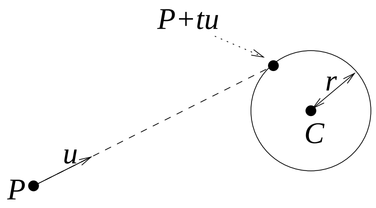


Figure 31: Sphere intersection.

Our goal is to determine the value of t for which the ray strikes the sphere, or to report that there is no 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

$$|P + t\vec{u} - C| = r.$$

Let $\vec{v} = C - P$ (this is opposite in sign from what we did earlier in the semester, but it works out a little cleaner in the end). 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, 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. The discriminant is $D = b^2 - 4ac$. There are some numerical instabilities to watch out for. 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, that it be computed in the following algebraically equivalent manner

$$D/4 = r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2.$$

If this 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 do 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. First consider

$$t^- = (\vec{u} \cdot \vec{v}) - \sqrt{r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2}.$$

If $t^- > 0$ then take it. Otherwise we should take the larger root

$$t^+ = (\vec{u} \cdot \vec{v}) + \sqrt{r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2}.$$

If this larger root is not positive either, then the sphere does not intersect the ray (it intersects the negative extension of the ray).

It is a not a good idea to compare floating point numbers against zero, since floating point errors are always possible. You should define some small value, e.g. `EPS = 1.0E-3`, and perform all comparisons against this value instead.

There is a possibility of roundoff error if we simply use the formulas given above (due to loss of floating precision when subtracting numbers). Instead it is recommended that you first compute the root with the larger absolute value, and then use the identity $t^-t^+ = c/a$, to get the other root. In particular, if $b \leq 0$ then use:

$$\begin{aligned} t^+ &= \frac{-b + \sqrt{D}}{2a} = (\vec{u} \cdot \vec{v}) + \sqrt{r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2}, \\ t^- &= \frac{c}{at^+} \qquad \text{(only if } t^+ > 0). \end{aligned}$$

Otherwise, if $b > 0$, then we use

$$\begin{aligned} t^- &= \frac{-b - \sqrt{D}}{2a} = (\vec{u} \cdot \vec{v}) - \sqrt{r^2 - |\vec{v} - (\vec{u} \cdot \vec{v})\vec{u}|^2}, \\ t^+ &= \frac{c}{at^-} \qquad \text{(only if } t^- < 0). \end{aligned}$$

As before, select the smaller positive root as the solution.

Plane Intersections: Next we consider the intersection of a ray with a plane. We have seen that a plane in 3-space can be represented by its four coefficients (a, b, c, d), such that a point $P = (p_x, p_y, p_z)$ lies on the plane if and only if

$$ap_x + bp_y + cp_z + d = 0.$$

We have already seen in earlier lectures how to compute the plane's coefficients from a set of point. Note that this is a homogeneous representation, in the sense that the plane is invariant under any nonzero scalar multiple of (a, b, c, d).

A plane partitions space into two regions, called *halfspaces*, one lying above the plane and the other lying below. If we think of the normal vector (a, b, c) as being directed outwards (away) from the halfspace, then a point P lies in the (closed) halfspace if and only if

$$ap_x + bp_y + cp_z + d \leq 0.$$

To represent the opposite halfspace, we can just negate the coefficients $(-a, -b, -c, -d)$. This representation is invariant under positive scalar multiples.

The intersection of any number of halfspaces defines a *convex polyhedron* in 3-space. This one of two common ways of defining a polyhedron. (The other is by specifying its vertices and edges. This latter form is more general, since it can also be used to define nonconvex polyhedra, but the intersection of halfspaces is simple and quite useful.)

The question we will consider is, given a convex polyhedron represented by a collection of halfspaces, how do we determine the first intersection of the ray with the boundary of the polyhedron. The origin of the ray may either be inside or outside the polyhedron.

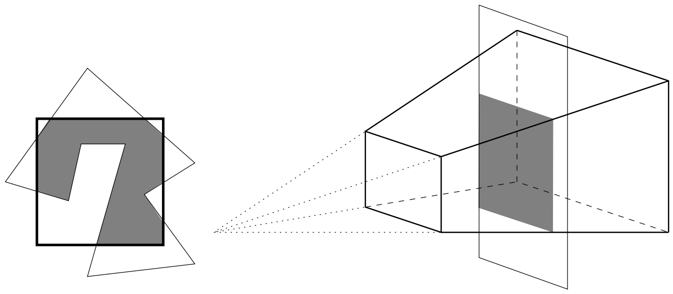


Figure 32: Clipping and convex intersection.

Clipping: Another motivation for studying the intersection problem comes from a completely different origin, clipping. *Clipping* is the process of trimming graphics primitives (e.g., line segments or polygons) to some shape. In the plane the shape is typically the window. In 3-space the viewing volume is the viewing frustum.

The simplest object to clip is a line segment. The process of line segment clipping against the view volume is quite similar to ray intersection with a polyhedron. This is particularly true of the Liang-Barsky algorithm, which uses a parametric representation for line segments, that is analogous to the parametric representation we use for rays. The 6-faces of the view volume define a convex polyhedron in 3-space. The same formulas that we will derive for rays can be adapted easily for clipping line segments as well.

The process of clipping a general polygon is a little trickier, because in general the clipping process may result in the creation or deletion of vertices. However, it is a fairly easy process to modify the list of vertices on the fly, and then just pass them through the graphics pipeline. This is how the Sutherland-Hodgman polygon clipping algorithm works.

Note that there is some difference in different graphics systems as to whether clipping is done before or after the projection transformation is applied. The approach described here assumes that it is done before the projection transformation is applied. (I believe that OpenGL does it after projection but before normalization. This is a little more elegant, but requires some care due to issues of projective geometry.)

Ray-Polyhedron Intersection: We will present an intersection algorithm for a ray and a polyhedron bounded by a set of halfspaces. This is a variation of the Liang-Barsky line segment clipping algorithm. As before, we represent the ray parametrically as $P + t\vec{u}$, for scalars $t \geq 0$. 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 24: Fractals

(Tuesday, April 29, 1997)
Read: Section 8.8 in Angel.

Fractals: One of the most important aspects of any graphics system is how objects are modeled. Most man-made (manufactured) objects are fairly simple to describe, largely because the plans for these objects are be designed "manufacturable". However, objects in nature (e.g. mountainous terrains, plants, and clouds) are often much more complex. These objects are characterized by a nonsmooth, chaotic behavior. The mathematical area of *fractals* was created largely to better understand these complex structures.

One of the early investigations into fractals was a paper written on the length of the coastline of Scotland. The contention was that the coastline was so "jagged" that its length seemed to constantly increase as the length of your measuring device (mile-stick, yard-stick, etc.) got smaller. Eventually, this phenomenon was identified mathematically by the concept of the *fractal dimension*. The other phenomenon that characterizes fractals is *self similarity*, which means that features of the object seem to reappear in numerous places but with smaller and smaller size.

In nature, self similarity does not occur exactly, but there is often a type of *statistical* self similarity, where features at different levels exhibit similar statistical characteristics, but at different scales.

Julia Sets: One of the examples of fractals arising in mathematics involves sets called *attractors*. The idea is to consider some function of space and to see where points are mapped under this function. There are many ways of defining functions of the plane or 3-space. One way that is popular with mathematicians is to consider the complex plane. Each coordinate (a, b) in this space is associated with the complex number $a + bi$, where $i = \sqrt{-1}$. Adding and multiplying complex numbers follows the familiar rules:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

and

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

Define the *modulus* of a complex number $a + bi$ to be length of the corresponding vector in the complex plane, $\sqrt{a^2 + b^2}$. This is a generalization of the notion of absolute value with reals. Observe that the numbers of given fixed modulus just form a circle centered around the origin in the complex plane.

Now, consider any complex number z. If we repeatedly square this number,

$$z \rightarrow z^2,$$

then the number will tend to fall towards zero if its modulus is less than 1, it will tend to grow to infinity if its modulus is greater than 1. And numbers with modulus 1 will stay at modulus 1. In this case, the set of points with modulus 1 is said to be an *attractor* of this iterated function. The *attractor set* is the set that remains fixed under the mapping, i.e. it is the *fixed-point* of the function. (Note that it is the set as a whole that is fixed, NOT the points themselves.)

Suppose we modify the complex function so that instead of simply squaring the point we apply the iterated function

$$z \rightarrow z^2 + c$$

where c is any complex constant. Now as before, under this function, some points will tend toward ∞ and others towards finite numbers. However there will be a set of points that will tend toward neither. Altogether these latter points form the *attractor* of the function system. This is called the *Julia set* for the point c. An example for $c = -0.62 - 0.44i$ is shown in the following figure.

A common method for approximately rendering Julia sets is to iterate the function until the modulus of the number exceeds some prespecified threshold. If the number diverges, then we display one color, and otherwise we display another color. How many iterations? It really depends on the desired precision. Points that are far from the boundary of the attractor will diverge quickly. Points that very close, but just outside the boundary may take much longer to diverge. Consequently, the longer you iterate, the more accurate your image will be.

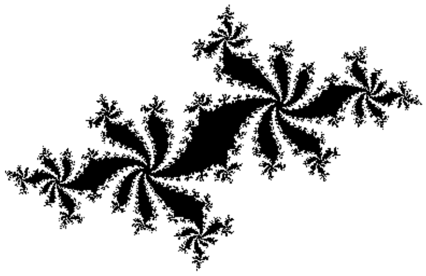


Figure 33: A Julia Set.

The Mandelbrot Set: For some values of c the Julia set forms a connected set of points in the complex plane. For others it is not. For each point c in the complex plane, if we color it black if Julia(c) is connected, and color it white otherwise, we will a picture like the one shown below. (This is not quite correct, there is a "halo" around the set that is an artifact of my fractal drawing program.) This set is called the *Mandelbrot set.*

One way of approximating whether a complex point d is in the Mandelbrot set is to start with $z = (0, 0)$ and successively iterate the function $z \rightarrow z^2 + d$, a large number of times. If after a large number of iterations the modulus exceeds some threshold, then the point is considered to be outside the Mandelbrot set, and otherwise it is inside the Mandelbrot set. As before, the number of iterations will generally determine the accuracy of the drawing.

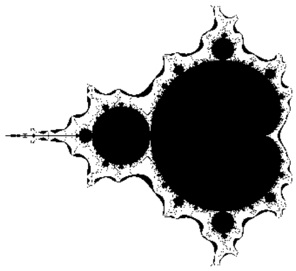


Figure 34: The Mandelbrot Set.

Fractal Dimension: One of the important elements that characterizes fractals is the notion of *fractal dimension.* Fractal sets behave strangely in the sense that they do not seem to be 1-, 2-, or 3-dimensional sets, but seem to have noninteger dimensionality.

What do we mean by the *dimension* of a set of points in space? Intuitively, we know that a point is zero-dimensional, a line is one-dimensional, and plane is two-dimensional and so on. If you put the object into a higher dimensional space (e.g. a line in 5-space) it does not change its dimensionality. If you continuously deform an object (e.g. deform a line into a circle or a plane into a sphere) it does not change its dimensionality.

How do you determine the dimension of an object? There are various methods. Here is one, which is called *fractal dimension*. Suppse we have a set in d-dimensional space. Define a d-dimensional *ϵ-ball* to the interior of a d-dimensional sphere of radius ϵ. An ϵ-ball is an open set (it does not contain its boundary) but for the purposes of defining fractal dimension this will not matter much. In fact it will simplify matters (without changing the definitions below) if we think of an ϵ-ball to be a solid d-dimensional hypercube whose side length is 2ϵ (an ϵ-square).

The dimension of an object depends intuitively on how the number of balls its takes to cover the object varies with ϵ. First consider the case of a line segment. Suppose that we have covered the line segment, with ϵ-balls, and found that it takes some number of these balls to cover to segment. Suppose we cut the size of the balls exactly by $1/2$. Now how many balls will it take? It will take roughly twice as many to cover the same area. (Note, this does not depend on the dimension in which the line segment resides, just the line segment itself.) More generally, if we reduce the ball radius by a factor of $1/a$, it will take roughly a times as many balls to cover the segment.

On the other hand, suppose we have covered a planar region with ϵ-balls. Now, suppose we cut the radius by $1/2$. How many balls will it take? It will take 4 times as many. Or in general, if we reduce the ball by a radius of $1/a$ it will take roughly a^2 times as many balls to cover the same planar region. Similarly, one can see that with a 3-dimensional object, reducing by a factor of $1/2$ will require 8 times as many, or a^3.

This suggests that the nature of a d-dimensional object is that the number of balls of radius ϵ that are needed to cover this object grows as $(1/\epsilon)^d$. To make this formal, given an object A in d-dimensional space, define

$$N(A, \epsilon) = \text{smallest number of } \epsilon\text{-balls needed to cover } A.$$

It will not be necessary to the absolute minimum number, as long as we do not use more than a constant factor times the minimum number. We claim that an object A has dimension d if $N(A, \epsilon)$ grows as $C(1/\epsilon)^d$, for some constant C. This applies in the limit, as ϵ tends to 0. How do we extract this value of d? Observe that if we compute $\ln N(A, \epsilon)$ (any base logarithm will work) we get $\ln C + d\ln(1/\epsilon)$. As ϵ tends to zero, the constant term C remains the same, and the $d\ln(1/\epsilon)$ becomes dominant. If we divide this expression by $\ln(1/\epsilon)$ we will extract the d.

Thus we define the *fractal dimension* of an object to be

$$d = \lim_{\epsilon \to 0} \frac{\ln N(A, \epsilon)}{\ln(1/\epsilon)}.$$

Formally, an object is said to be a *fractal* if it has a noninteger fractal dimension.

Now suppose we try to apply this to fractal object. Consider first the *Sierpinski triangle*, defined as the limit of the following process.

How many ϵ-squares does it take to cover this figure. It takes one 1-square to cover it, three $(1/2)$-squares, nine $(1/4)$-squares, and in general 3^k, $(1/2^k)$-squares to cover it. Letting $\epsilon = 1/2^k$, we find that the fractal dimension of the Sierpinski triangle is

$$D = \lim_{\epsilon \to 0} \frac{\ln N(A, \epsilon)}{\ln(1/\epsilon)}$$

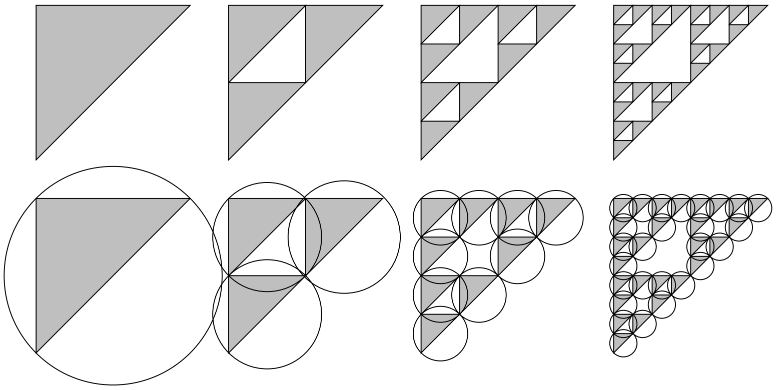


Figure 35: The Sierpinski triangle.

$$
\begin{aligned}
&= \lim_{k\to\infty} \frac{\ln N(A,(1/2^k))}{\ln(1/(1/2^k))} \\
&= \lim_{k\to\infty} \frac{\ln 3^k}{\ln 2^k} \;=\; \lim_{k\to\infty} \frac{k\ln 3}{k\ln 2} \\
&= \lim_{k\to\infty} \frac{\ln 3}{\ln 2} \;=\; \frac{\ln 3}{\ln 2} \;\approx\; 1.58496\ldots.
\end{aligned}
$$

Thus although the Sierpinski triangle resides in 2-dimensional space, it is essentially a 1.58 dimensional object, with respect to fractal dimension. Although this definition is general, it is sometimes easier to apply the following formula for fractals made through repeated subdivision. Suppose we form an object by repeatedly replacing each "piece" of length x by b nonoverlapping pieces of length x/a each. Then it follows that the fractal dimension will be

$$D = \frac{\ln b}{\ln a}.$$

As another example, consider the limit of the process shown in the following figure. The area of the object does not change, and it follows that the fractal dimension of the interior is the same as a square, which is 2 (since the balls that cover the square could be rearranged to cover the object, more or less). However, if we consider the boundary, observe that with each iteration we replace one segment of length x with 4 subsegments each of length $\sqrt{2}/4$. It follows that the fractal dimension of the boundary is

$$\frac{\ln 4}{\ln(4/\sqrt{2})} = 1.3333\ldots.$$

The the shape is not a fractal (by our definition), but its boundary is.

Lecture 25: Solid Modeling

(Thursday, May 1, 1997)
Read: Chapt. 8 in Angel.

Solid Object Representations: Today we continue our 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

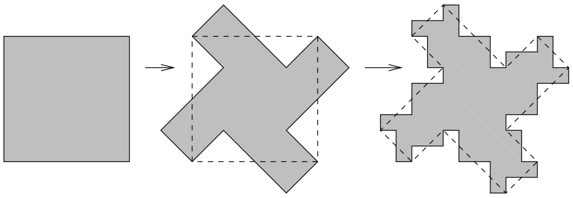


Figure 36: An object with a fractal boundary.

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.

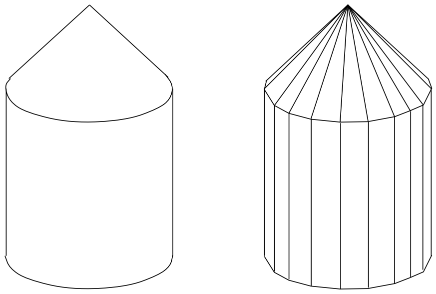


Figure 37: A polygonal mesh representation.

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. 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.

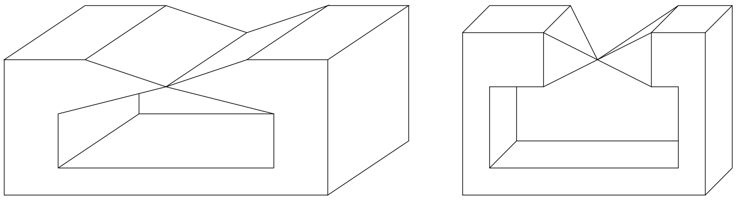


Figure 38: Non-manifold solids.

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

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

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* and *quad-edge* representations, are based on storing the following topological incidence information (in addition to whatever geometric and graphics properties are stored).

For each vertex: A pointer to at least one incident edge.

For each face: A pointer to at least one bounding edge.

For each edge: We store the following:

- **Vertices:** Pointers to the two incident vertices.
- **Faces:** Pointers to the two incident faces.
- **Edges:** Pointers to each of the two neighboring edges (clockwise and counterclockwise) along each of the incident faces.

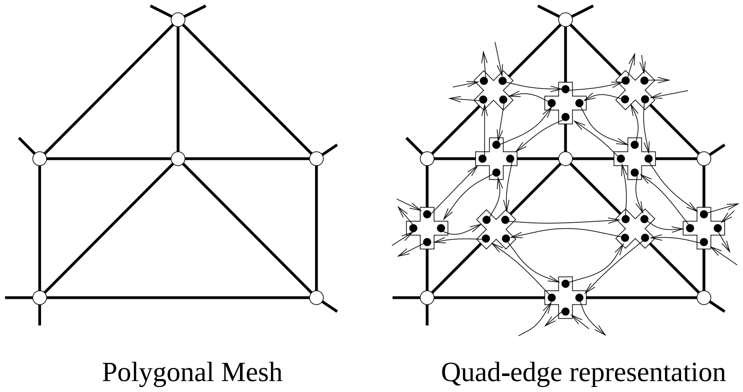


Figure 39: Quad-edge representation.

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.

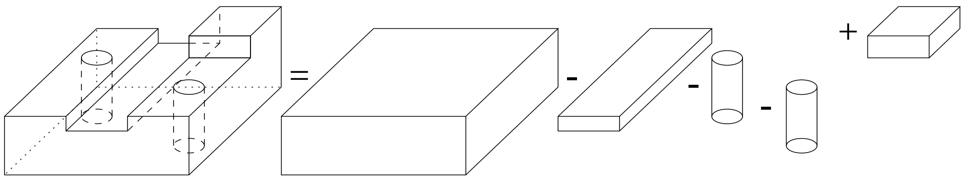


Figure 40: Constructive Solid Geometry.

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 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

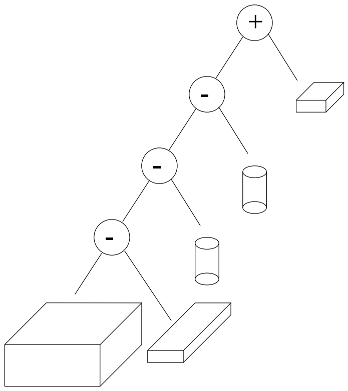


Figure 41: CSG Tree.

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.

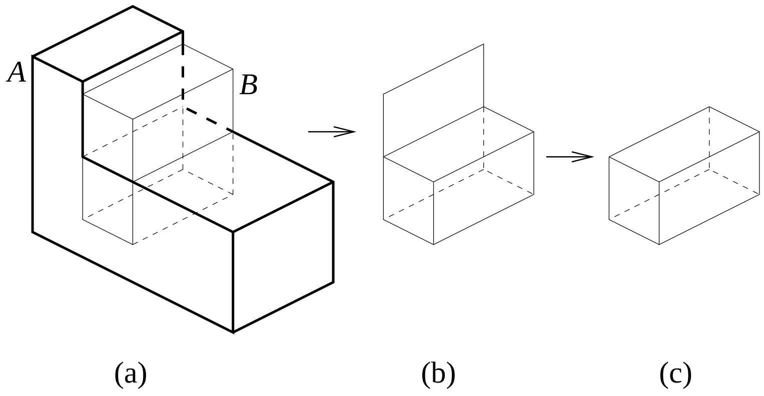


Figure 42: (a) A and B, (b) $A \cap B$, (c) $A \cap^* B$.

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 26: Curves and Surfaces

(Tuesday, May 6, 1997)
Read: Chapt. 9 in Angel.

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.

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

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

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 $1 + 2 + 1 = 4$).

An implicit representation works fine for surfaces. To represent a curve in 3-space we would need to compute the intersection of 2 surfaces, that is, provide a system of algebraic equations. These are generally messy to work with.

Parametric representation: In this representation a curve is given as three functions of one parameter $(x(u), y(u), z(u))$ and a surface is given as function of two parameters $(x(u,v), y(u,v), z(u,v))$. For example, here is a 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 $0 \leq \theta \leq 2\pi$ and $-\pi/2 \leq \phi \leq \pi/2$. (Notice that this is somewhat different from the parameterization we gave earlier for spheres, since the range of ϕ values differs.) Notice that although the ellipse has an algebraic implicit representation, it does not seem to have an algebraic parametric representation. There are alternative parametric representations (e.g. one is in the form of a ratio of polynomials, and is called a *rational parameterization.*)

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 the most popular in computer graphics, so we will concentrate on them.

Continuity: An important condition that we would like our curves as surfaces to satisfy is that they should be as smooth as possible. What does this mean? We would like the curves themselves to be continuous (that is not making sudden jumps in value). This is called *zeroth order* (or C^0) *continuity.* If the first derivative is also continuous, then we have C^1 continuity. In this case tangents vary continuously along the curve. In C^2 continuous curves, the radius of curvature varies continuously. In general, if the first k derivatives are continuous, then we have C^k continuity.

Since the same function can be parameterized in potentially many ways, this seems like the wrong criterion to be using for continuity. For example, when two parametric curves are joined together at a common point, suppose that one has been parameterized with a parameter value that is changing slowly for a unit distance along the curve and the other is changing rapidly for a unit distance along the curve. Even if the curves joined smoothly at their common joining point, they are considered discontinuous because their parameterizations are not continuous. For this reason, people often separate the notions of *parametric continuity* from the weaker notion of *geometric continuity.* In geometric continuity, we consider the smoothness of the underlying curve, not its parameterization. We let G^k denote kth order geometric 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 considered good enough.

Blending Polynomials: The basic problem for a designer is how to construct a polynomial to achieve or approximate a certain desired shape. Let us begin with curves, and then we will generalize to surfaces later. Consider the case of a cubic parametric curve. Such a curve has the following representation.

We can express a cubic parametric curve as

$$\begin{aligned} x(u) &= c_{0x} + c_{1x}u + c_{2x}u^2 + c_{3x}u^3 \\ y(u) &= c_{0y} + c_{1y}u + c_{2y}u^2 + c_{3y}u^3 \\ z(u) &= c_{0z} + c_{1z}u + c_{2z}u^2 + c_{3z}u^3, \end{aligned}$$

or in vector notation as

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

Beware of the notation here. We are treated the quantities $\mathbf{c}_i$ as row-vectors in this equation.

Trying to figure out how to set the coefficients to achieve a certain desired shape is not an easy task. An important concept in geometric design was the idea of using a set of *control points*, whose placement determines the coefficients.

For example, we could specify four points in the plane and determine the unique cubic polynomial that passes through these points. Generally, it takes $d+1$ points to determine a polynomial of degree d. This is called a *interpolating polynomial*. Although the equations are easy to derive, this method was discovered to produce polynomials that tended to "wiggle" too much between the control points. Consequently people looked for methods in which the curve did not necessarily pass through the control points, but was influenced by them.

This suggests the following approach, which can be thought of as a generalization some of the interpolation schemes we have seen earlier this semester. Consider cubic polynomials. Define four *blending functions* $b_i(u)$, $1 \le i \le 4$, each a cubic polynomial, such that given any four points $\mathbf{p}_i$, $0 \le i \le 3$, we take the *weighted average* of these blending functions times the points:

$$\mathbf{p}(u) = \sum_{i=0}^{3} b_i(u)\mathbf{p}_i \qquad \text{for } 0 \le u \le 1.$$

Intuitively, if we want the curve to start near $\mathbf{p}_0$ then go towards $\mathbf{p}_1$, then $\mathbf{p}_2$ then $\mathbf{p}_3$, we want $b_0(u)$ to be the largest of the four when $u = 0$, $b_3(u)$ should be the largest $u = 1$, and $b_1(u)$ and $b_2(u)$ would be largest for some intermediate values of u, e.g. $u = 1/3$ and $u = 2/3$, respectively. As u varies from 0 to 1, the curve will be attracted to each successive point. The question is how to design a set blending functions that have this property. There are many solutions, but we will discuss one that is particularly elegant.

Bézier Curves: Bézier curves and surfaces are based on the simple idea of repeated linear interpolation. We will derive the cubic Bézier curve based on four control points, but it is easy to see how to generalize this idea for any number of control points.

We begin by analogy. Suppose we want to interpolate between just two points. This can be done using a simple linear function:

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

Observe that this is a weighted average of the points, and for any value of u, the two weighting functions u and $(1-u)$ are nonnegative and sum to 1. In other words, $p(u)$ is a convex combination of $\mathbf{p}_0$ and $\mathbf{p}_1$ for all values of u.

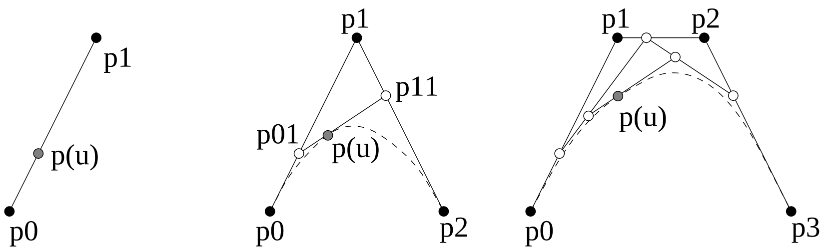


Figure 43: Repeated interpolation.

Suppose we now have three points and we want a smooth quadratic interpolation between them. Consider the line segments $\mathbf{p}_0\mathbf{p}_1$ and $\mathbf{p}_1\mathbf{p}_2$. From the linear interpolation we know how to interpolate a point on each, say

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

Now suppose that we interpolate between these two in the same way

$$\begin{aligned}
\mathbf{p}(u) &= (1-u)\mathbf{p}_{00} + u\mathbf{p}_{10} \\
&= (1-u)((1-u)\mathbf{p}_0 + u\mathbf{p}_1) + u((1-u)\mathbf{p}_1 + u\mathbf{p}_2) \\
&= (1-u)^2\mathbf{p}_0 + (2u(1-u))\mathbf{p}_1 + u^2\mathbf{p}_2.
\end{aligned}$$

This is the Bézier curve of order 2. Again, notice 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. The blending functions are:

$$\begin{aligned}
b_{02}(u) &= (1-u)^2 \\
b_{12}(u) &= 2u(1-u) \\
b_{22}(u) &= u^2.
\end{aligned}$$

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

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

Finally interpolate these $(1-u)\mathbf{p}_{02} + u\mathbf{p}_{12}$. 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 is the Bézier curve of order 3. Again, observe that if you plug in any value for u, these functions are all nonnegative and sum to 1. The blending functions are

$$\begin{aligned}
b_{03}(u) &= (1-u)^3 = 1 - 3u + 3u^2 - u^3 \\
b_{13}(u) &= 3u(1-u)^2 = 3u - 6u^2 + 3u^3 \\
b_{23}(u) &= 3u^2(1-u) = 3u^2 - 3u^3 \\
b_{33}(u) &= u^3.
\end{aligned}$$

We can express the final cubic parametric curve in matrix form as

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

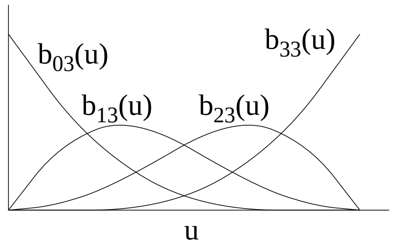


Figure 44: Bézier blending functions.

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.

Clearly we could continue this idea for any number of points. It is not hard to extrapolate the general form of the blending functions. In particular, each blending function arises by applying linear interpolation to two blended points. We have the following recursive rule for $d \geq 0$ and $0 \leq i \leq d$:

$$b_{i,d}(u) = \begin{cases} 1 & \text{if } i = d = 0, \\ 0 & \text{if } i < 0 \text{ or } i > d, \\ b_{i-1,d-1}(u) \cdot u + b_{i,d-1}(u) \cdot (1-u) & \text{otherwise.} \end{cases}$$

Based on the analogy with Pascal's triangle, this can also be expressed in the following form

$$b_{i,d} = \binom{d}{i} u^i (1-u)^{d-i} = \frac{d!}{i!(d-i)!} u^i (1-u)^{d-i},$$

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.

Bézier Surfaces: Now that we have defined Bézier curves, it is an easy matter to define surfaces. We simply generalize the bilinear interpolation idea that we saw earlier. For concreteness, suppose that we are dealing with cubic surfaces. We will use a 4×4 array of control points, $\mathbf{p}_{ij}$, $0 \leq i, j \leq 3$, and the resulting parametric formula is

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

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 Béziercurve. The final surface is this combination of curves. It has the same convex hull and tangent properties that Bézier curves have.

Lecture 27: More on Curves and Surfaces

(Thursday, May 8, 1997)
Read: Chapt. 9 in Angel.

Subdividing Bézier curves: Last time we introduced the mathematically elegant Bézier curves and surfaces, but there was one important issue that we failed to discuss: how to render them. Typically this is done by computing a sufficiently dense set of points along the curve or surface, and then approximate 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 curves and surfaces. 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 how the subdivision scheme works for curves. Perform the repeated interpolation construction using the parameter $u = 1/2$. Label the vertices as shown in the figure below. Now, consider the quadruples $\langle \mathbf{p}_{00}, \mathbf{p}_{01}, \mathbf{p}_{02}, \mathbf{p}_{03} \rangle$ and $\langle \mathbf{p}_{03}, \mathbf{p}_{12}, \mathbf{p}_{21}, \mathbf{p}_{30} \rangle$. Each of these quadruples defines a Bézier curve. Amazingly, the first quadruple generates the first half of the curve for $u \in [0, 1/2]$ and the second quadruple generates the second half of the curve for $u \in [1/2, 1]$.

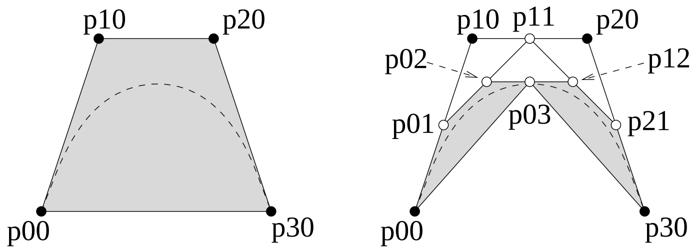


Figure 45: Bézier subdivision.

Thus 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.

The generalization to surfaces is straightforward. First the four curves associated with varying u and fixed v are subdivided, and then the 8 curves curves associated with varying v and fixed u are subdivided, resulting in 16 control points.

Cubic B-splines: We observed last time that Bézier curves 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, and global unpredictability since the movement of even one control point has an effect on the entire curve. On the other hand, we could contend ourselves to link together a many cubic Bézier curves, but getting the joints to link with C^2 continuity (recall that this means that the function and its first two derivatives are continuous) can be quite hard. 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 point, and C^2 continuity.

B-splines were developed to address these shortcomings. Intuitively the idea is that we will still use smooth blending functions multiplied times the control points, but these functions will have the property that they are 0 over most of the range of parameter values, and consist of nonzero smooth polynomials over a small interval. 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.

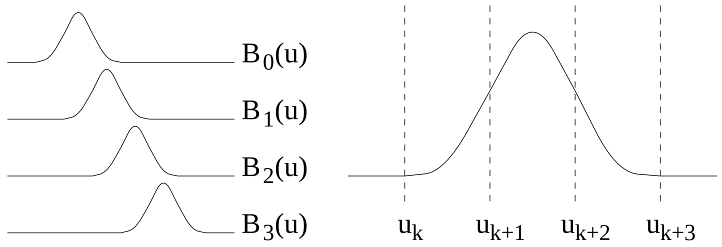


Figure 46: B-spline basis function.

The problem is 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:

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

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? 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.* (Warning: There is an error in the book's formula.) 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 a mouthfull. First off, observe that if the knot intervals are equally spaced, e.g. $u_i = i$, then the two blending coefficients sum to 1. What is more important is that irrespective of the knot spacing the blending functions sum to 1 (provided that we have an infinite sequence of knot values). This can be proved by induction.

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 2 intervals and is a piecewise linear function that goes from 0 to 1 and then back to 0. $B_{k,2}(u)$ spans 3 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 4 intervals growing from 0 to 1/6 to 2/3, then back to 1/6 and to 0. 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.

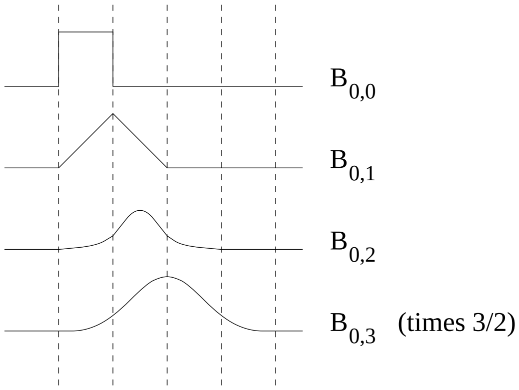


Figure 47: B-spline blending functions.

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:

$$\begin{aligned} x(u) &= 1-u^2 \\ y(u) &= 2u \\ z(u) &= 0 \\ w(u) &= 1+u^2 \end{aligned}$$

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 28: Final Review

(Tuesday, May 13, 1997)

Overview: This semester we have covered introductory computer graphics. The main topics that we covered included basic geometry and transformations, projective geometry and perspective, illumination and shading, surface mappings, hidden-surface removal, and object modeling. Much of this was done in the context of OpenGL. This was based on a top-down approach. Compared with previous semesters we spent less time with the low-level details: rasterization, clipping, dithering, and spent more time on higher-level issues: perspective transformations, surface mappings, curves and surfaces.

What we didn't cover: There is just too much material about computer graphics to squeeze into one graphics course. Some issues include:

Rasterization: How are lines drawn? How are polygons filled? The algorithms are pretty simple, but there are some interesting issues to consider. For example, can you define a triangle rasterization, such that given two triangles that share a common edge, each pixel in their union will be drawn at least once (i.e. no cracks between them), or better that no pixel is drawn more than once.

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.

Issues with color: Throughout the semester we assumed that we could render RGB values. There are two significant issues. First, how do we render many colors approximately when we have a display that only supports a color map with 64 colors. This task is called *dithering*. There are a number of other problems with color perception. The first has to do with the fact that the human eye does not perceives light in a linear manner. For this reason, illumination is often scaled by a process called *gamma correction* to make RGB values appear as we think they should. Another issue involves the use of other color models (e.g. subtractive models such as CMY or CMYK for printed colors).

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 highly 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. The method is called *radiosity*.

Here is a simple way to think about how radiosity works. First emit light from all the light sources and determine the intensities of incoming light, and hence the intensity of outgoing light (based on the object's reflective properties). Then, for each object that reflects light, treat it as a new light source and repeat the process. (Note that if an object is red, then it will reflect only red light. This phenonemon is called *color bleeding*, where the color of a given point is influenced by nearby objects.) Repeat the process until the system arrives at some steady state.

This is one of the topics covered in an advanced graphics course.

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.

Topics: The final exam will be comprehensive but will emphasize the material since the midterm (actually since Lecture 14, on lighting and shading). Here are the main topics that we have covered since the midterm.

Lighting and Shading: Although lighting and shading was covered just prior to the midterm, you should treat it as a major topic for the final. In particular the issues involve understanding the Phong illumination model, some understanding of its strengths and weaknesses, and the nature of diffuse and specular reflection. You should be aware of OpenGL's approach to lighting (as much as we encountered in Project 2), but you are not responsible for knowing the exact calling sequences.

Surface Mappings: We discussed methods for approximating complex surface finish properties such as textured color, bumpiness, and reflection. Recall that the key is to store the needed information as a function, stored in a large array, and then define the necessary functions for mapping points on the surface of an object to the appropriate function value. The mapping typically involved first parameterizing the surface in terms of 2 variables, and then defining the appropriate mapping function from parameter space to the texture space.

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.

Ray Tracing: We discussed ray tracing in depth as a method for generating very realistic images, especially in the presence of sharp reflection and refraction effects. We also discussed the related issues of computing intersections between rays and various types of objects. 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).

Solid Modeling: We discussed a number of issues related to how to model and represent 3-dimensional objects. In particular we discussed the following:

Fractals: as a method of modeling irregular objects. (We introduced the notion of fractal dimension and showed that fractal objects have noninteger dimension).

Constructive Solid Geometry: A hierarchical tree-structured representation of objects through the use of boolean operations.

Boundary Representations: We discussed representing objects by their boundary patches and Euler's formula.

Curves and Surfaces: We discussed Bézier curves and B-spline curves and their properties. We introduced the notion of C^k smoothness. In both cases the important element was that we viewed the curve as a weighted average of a given set of control points and derived a particular weighting function that guaranteed certain smoothness properties.

Lecture X01: Scan Conversion for Lines and Circles

(Supplemental)

Midpoint Line Algorithm: We continue with the development of Bresenham's Algorithm (also called the Midpoint Algorithm) for scan converting lines. Recall that we are drawing the line segment from (x_0, y_0) to (x_1, y_1). Furthermore, we have assumed that the slope of the line is between 0 and 1, and that $x_0 < x_1$. (Last time we showed how to convert the other cases to this case.)

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 $dx = x_1 - x_0$, $dy = y_1 - y_0$, it is easy to see (by substitution) that $a = dy$, $b = -dx$, and $c = -(x_0 y_1 - x_1 y_0)$. Observe that all of these coefficients are all integers. Observe that that $F(x, y) > 0$ for points that lie below the line and $F(x, y) < 0$ for points above the line. (To see this, consider what happens to $F(x, y)$ as y goes to either $+\infty$ or $-\infty$. Because $dx > 0$ we have $b < 0$.)

Actually, 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. As in the previous 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 (x_p, y_p) 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 ($E = (x_p + 1, y_p)$) or the pixel to our NorthEast ($NE = (x_p + 1, y_p + 1)$). Let Q denote the exact y-value (a real number) of the line at $x = x_p + 1$. Let M denote the midpoint between y_p and $y_p + 1$, $M = y_p + 1/2$. 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.

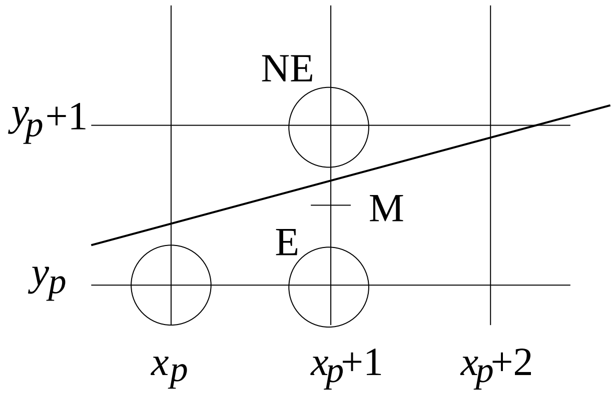


Figure 48: Midpoint algorithm for lines.

To determine which one to pick, we have a *decision variable* D which will be the value of $F(M)$. Thus

$$\begin{aligned} D &= F(M) = F(x_p + 1, y_p + (1/2)) \\ &= 2a(x_p + 1) + 2b(y_p + \frac{1}{2}) + c \\ &= 2ax_p + 2by_p + (2a + b + c). \end{aligned}$$

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 2 multiplications and 2 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 $(x_p + 2, y_p + (1/2))$ and hence the new D value will be

$$D_{new} = F(x_p + 2, y_p + (1/2))$$

$$
\begin{aligned}
&= 2a(x_p+2) + 2b(y_p+\frac{1}{2}) + c \\
&= 2ax_p + 2by_p + (4a+b+c) \\
&= 2ax_p + 2by_p + (2a+b+c) + 2a \\
&= D + 2a = D + 2dy.
\end{aligned}
$$

Thus, the new value of D will just be the current value plus $2dy$.

We go to NE next: Then the next midpoint will have coordinates $(x_p+2, y_p+1+(1/2))$ and hence the new D value will be

$$
\begin{aligned}
D_{new} &= F(x_p+2, y_p+1+(1/2)) \\
&= 2a(x_p+2) + 2b(y_p+\frac{3}{2}) + c \\
&= 2ax_p + 2by_p + (4a+3b+c) \\
&= 2ax_p + 2by_p + (2a+b+c) + (2a+2b) \\
&= D + 2(a+b) = D + 2(dy-dx).
\end{aligned}
$$

Thus the new value of D will just be the current value plus $2(dy-dx)$.

Note that in either case we need perform only one addition (assuming we precompute the values $2dy$ and $2(dy-dx)$. 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 (x_0, y_0) the initial midpoint is at $(x_0+1, y_0+1/2)$ so the initial value of D is

$$
\begin{aligned}
D_{init} &= F(x_0+1, y_0+1/2) \\
&= 2a(x_0+1) + 2b(y_0+\frac{1}{2}) + c \\
&= (2ax_0 + 2by_0 + c) + (2a+b) \\
&= 0 + 2a + b \qquad \text{Since } (x_0, y_0) \text{ is on line} \\
&= 2dy - dx.
\end{aligned}
$$

We can now give the complete algorithm (again, under the assumptions that $x_0 < x_1$ and the slope lies between 0 and 1.

```
// Bresenham's midpoint algorithm
// Scan convert a line segment from (x0,y0) to (x1,y1)
//
void MidPointLine(int x0, int y0, int x1, int y1)
{
    int dx, dy, incE, incNE, D, x, y;

    dx = x1 - x0;                   // line coefficients
    dy = y1 - y0;
    D = 2*dy - dx;                  // initial decision value
    incE = 2*dy;                    // precompute decision increments
    incNE = 2*(dy - dx);
    x = x0;                         // start at (x0,y0)
    y = y0;
    WritePixel(x,y);
    while (x < x1) {
        if (D <= 0) D += incE;      // below midpoint - go to E
```

```
            else {                          // above midpoint - go to NE
                D += incNE;
                y++;
            }
            x++;
            WritePixel(x,y);
        }
    }
```

Midpoint Circle Algorithm: To illustrate that this idea can be applied to other types of curves, let us consider how to apply the same idea to drawing circles. As before 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*.

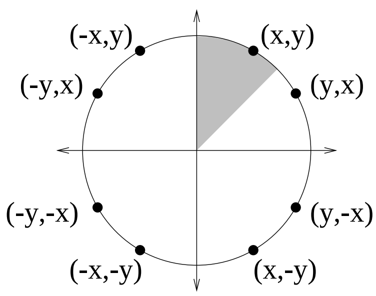


Figure 49: 8-way symmetry for circles.

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.

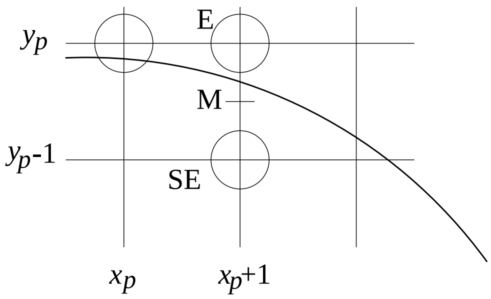


Figure 50: Midpoint algorithm for circles.

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

$$\begin{aligned} D &= F(M) = F(x_p + 1, y_p - \frac{1}{2}) \\ &= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2. \end{aligned}$$

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

$$D_{init} = F(1, R - \frac{1}{2})$$

$$
\begin{aligned}
&= 1 + (R - \frac{1}{2})^2 - R^2 \\
&= 1 + R^2 - R + \frac{1}{4} - R^2 \\
&= \frac{5}{4} - R.
\end{aligned}
$$

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 X02: Scan Conversion for Polygons

(Supplemental)

Filling Polygons: Although line drawing is fundamental to graphics (and especially printed graphics where color is not always readily available), with the wide availability of color raster displays, it is much more important to be able to draw entire regions. The regions are usually modeled using polygonal patches (we will see more about this later), and so polygon filling is an important primitive in computer graphics. We will consider the case where the polygon is filled with a single solid color, but polygon filling often involves producing additional shading or mapping complex textures as well. However, the basic task of determining which pixels are part of the filled region and which pixels are not is fundamental.

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, for example, if the polygons are written in XOR write mode, since a pixel that is written twice will generally have an undesired value.

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, which is functionally equivalent. 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. Note that the rule is very slightly different from the ILB rule given in the book (e.g. the bottommost pixel in Fig 3.17 on page 90 in the book would NOT be drawn by the LRsU rule).

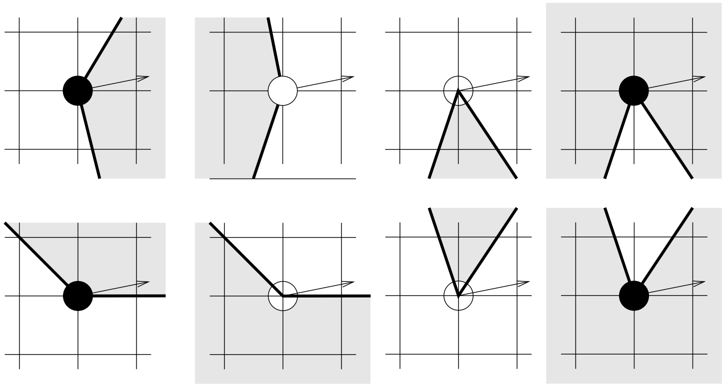


Figure 51: The LRsU Rule.

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. There are the types of vertices we can hit:

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. An example is shown in the figure.

Implementation: There are a couple aspects to the efficient implementation of the scan-line algorithm.

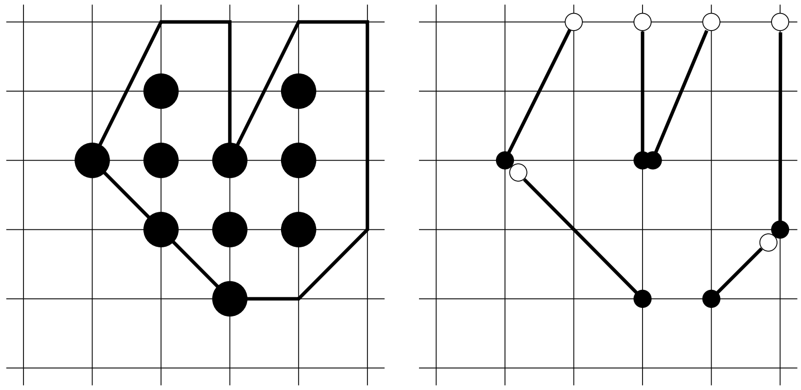


Figure 52: The Scan-Line Algorithm.

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.

0. Bucket sort all nonhorizontal edges into the edge table ET according to their bottom y-coordinate.
1. Set y to the smallest y-coordinate in the edge table.
2. Initialize the AET to empty.
3. Repeat until the AET is empty:
   - 3.1 Move from ET bucket y to AET those edges for which `y_min = y`, keeping the AET sorted by x-coordinate.
   - 3.2 Remove from the AET those entries for which `y_max = y`.
   - 3.3 Visit the AET in pairs, filling in the pixels in between.
   - 3.4 Increment y by 1.
   - 3.5 For each edge in the AET, update `x_curr` by the reciprocal slope.

Lecture X03: 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.

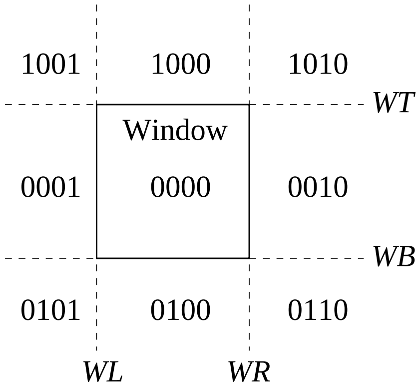


Figure 53: Cohen-Sutherland region codes.

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, 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

$$\frac{y_c - y_0}{y_1 - y_0} = \frac{WL - x_0}{x_1 - x_0}.$$

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 X04: Light and Color

(Supplemental)

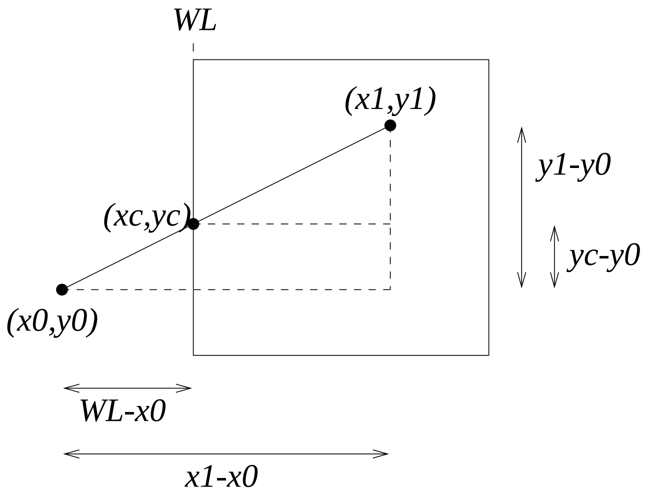


Figure 54: Clipping on left side of window.

Light: Light and its perception is a very deep area of study. In the next couple of lectures we will explore some of the issues that arise with light and color as they relate to computer graphics.

Light as we perceive it is *electromagnetic radiation* from a narrow band of the complete spectrum of electromagnetic radiation called the *visible spectrum*. The spectrum ranges from very low frequency radio waves to microwaves, infrared, visible light, ultraviolet and high frequency x-rays. As with a sine wave, the *frequency* of radiation is inversely proportional to its *wavelength*. Visible light lies in the range of wavelengths from around 400- to 700-nm (where 1 nanometer $= 10^{-7}$ centimeters).

Light travels from some source (e.g. the sun) where it strikes an object. The object absorbs some of the light, and reflects the rest in various directions. The light that enters our eye is what we perceive. This light is a combination of many wavelengths. The dominant wavelength determines the color that we see. The intensity of the radiation determines the brightness of what we see.

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. For example, we will see that equal energy intensities of colored light are perceived as being of different brightnesses depending on the color. I will use terms like *intensity* to describe the physical amount of energy, and *brightness* to describe our perception of this energy.

Another complicating factor is that *hardware* devices can often only approximate colors to a limited precision. Much of the "art" of computer graphics involves using this hardware to produce an image which can "fool the eye" into perceiving a realistic image.

Gamma Correction: *Achromatic light* is light as we would perceive from a black-and-white television. It is characterized by one attribute: *intensity* which is a measure of energy. 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. (These are not absolute phyical values, but usually ranges applied to hardware devices.)

One of the problems we will see with light and color is that the human perception system is nonlinear. For example, suppose we want to generate 11 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: $0.0, 0.1, 0.2, \ldots, 1.0$. 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. For example, we perceive the 10% difference between 0.1 and 0.11 the same as we would perceive the 10% difference between 0.5 and 0.55. 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 pixel. For this reason, most graphics servers provide a form of automatic correction, called *gamma correction*, so that a linear varition in the user-supplied intensity values (e.g. RGB) generate a perceived linear variation in brightness. However, even with gamma correction, do not be surprised if the same RGB values will produce exactly the same colors on two different systems.

Halftone Approximation: As mentioned earlier, graphics devices do not usually provide a continuous range of intensities but instead provide a descrete set of choices. The most extreme case is that of a monochrom display with only 2 colors, black and white. However, even with the 256 different colors that are available on a typical graphics workstation with a color map, this is completely inadequate for displaying a color image with smooth variations in shading. 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.

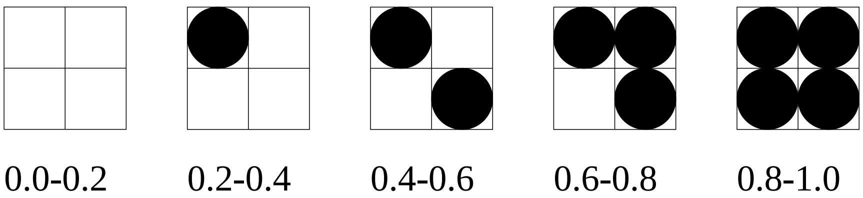


Figure 55: Halftone approximation with dither patterns.

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 X05: Color and Color Models

(Supplemental)

What is light and color? We mentioned last time that how we perceive and reproduce light and color depends on a complex interplay between the physics of light, the biology of the human perception system, and the nature of the hardware devices (CRT monitors, photographic and video cameras, offset printers, etc) that are used produce images.

We mentioned last time that the visible portion of light can be described in terms of a *spectral power distribution* (SPD) which describes the amount of each wavelength of light in the visible spectrum. For some interval of the electromagnetic spectrum, we can describe the physical quantity of *intensity* of light as the physical amount of energy that is incident to some surface. This is sometimes called *linear light.*

We sense light through three types of photoreceptive cells in our retina called *cones*. (There is a fourth type of cell called *rods* which are used only in very dim lighting situations, but they do not play a significant role in color perception.) When presented with an SPD, our visual system takes the combination of these 3 types of cones and what we perceive is called *color*. Because our perception of color is dependent on a combination of 3 values, most color systems describe color as a point in a 3-dimensional space. However, there is quite a bit of variety in how this is done.

Lightness (or *brightness*) are not physical quantities, but perceptual quantities. It is a complex function of linear light, but there are two components that can be separated out from the equation. First, we are more sensitive to light in some spectral ranges than others. For example, given equal amounts of red, green, and blue light, we will perceive the green as being the brightest, red second, and blue a distant third. Secondly, as mentioned last time, our perception system is *nonlinear* in that it responds logarithmically to increases in intensity, that is, we perceive differences in terms of *ratios* of intensities rather than absolute differences.

Luminance: 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.

Since lightness or brightness are hard quantities to pin down, they defined a concept called *luminance* which accounts for the sensitivity of our eye to different spectral ranges. Thus, given equal intensities of blue and green light, the blue light will have much lower luminance than the green light. Basically, luminance is just intensity multiplied by a *luminance-efficiency function* for the human eye. This function is 0 outside the visible spectrum (400 to 700 nm) and peaks near green at around 555 nm.

The luminance (linear brightness) of an SPD is a scalar quantity. To compute this you integrate the SPD using the luminous-efficiency function as a weighting function. (So that green light is given the strongest weight in deciding brightness.) This quantity is denoted Y, by the CIE. (Note: Y does NOT stand for "yellow" like R, G, and B stand for red, green and blue. Think of Y as the grayscale of a colored object.)

Note that luminance is still a linear quantity, in that it does not take into account the logarithmic nature of perception. Thus, gamma correction (described last time) would have to be applied to scale the light to a perceptually smooth range of values. If you were to compute Y values for each point in a scene, and apply gamma correction, the image would be a what you would expect from a black-and-white photograph.

CIE Chromaticity Diagram: We now have a way of describing the lightness of a color in terms of the more formal quantity, luminance. But how do we describe a color mathematically? (without giving the whole SPD.) The idea is to define 3 *primary colors*, from which all other colors can be produced by taking appropriate additive combinations. What should these primaries be? For reasons that we will see shortly, it is not generally possible to completely span the color space using additive combinations of three standard colors (such as red, green, and blue). In particular, no matter which colors you use, some subtraction is needed. However, devices like CRT's can only produce colors by addition (i.e. by making a phosphor brighter).

The CIE solved the problem by defining three standard primaries named X, Y and Z, so that colors can be made through a strictly additive process. The value Y is just the luminance defined above. The values X and Z are shown in the graph on page 406 of our text. Think of these functions as weighting functions which can be applied to an SPD for each wavelength λ, denoted $\overline{x}(\lambda)$, $\overline{y}(\lambda)$, and $\overline{z}(\lambda)$.

To map an SPD, given by $P(\lambda)$ into a 3-element vector, we integrate using these weighting functions:

$$X = k \int P(\lambda)\overline{x}(\lambda)d\lambda \qquad Y = k \int P(\lambda)\overline{y}(\lambda)d\lambda \qquad Z = k \int P(\lambda)\overline{z}(\lambda)d\lambda.$$

where k is a scaling factor. Thus, any color can be described as a 3-element vector in (X, Y, Z) space by adding these primaries together in the specified proportions.

This is a 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*:

$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z}$$

(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 showed on page 408 in our text. 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. As we get near the boundaries of the diagram we get the purest or most *saturated* colors (or *hues*). As we move towards the white point, the colors become less and less saturated.

Different groups define colors slightly differently, but the CIE coordinate space is widely regarded as an international standard. A nice feature of the XYZ-space is that we can predict the effect of additively combining various colors. For example, once red, green, and blue have been defined in terms of these quantities (e.g. red is $(0.64, 0.33, 0.03)$ in terms of (x, z, y)), then this defines 3 points in the chromaticity diagram. We can form any point in the triangle by taking appropriate linear combinations of these points. This is exactly how the RGB system works.

However, we can now see why RGB is not a universal system for producing colors. If we assume only additive combinations, then we can only produce points in the triangle spanned by the points R, G, and B. Because the shark-fin is rounded, no matter which three visible colors we select, we cannot span the entire fin. What we lose are some of the very highly saturated colors around the edges of the diagram.

Note that points on opposite sides of white are additive *complementary* colors. That is, by combining both colors we can produce white.

One shortcoming of CIE is that equal distances between colors do not produce perceptually equal changes. An alternative system called CIE LUV attempts to correct this.

Raster Graphics Color Models: 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.

RGB Color Model: The RGB model uses the additive primaries red, green, and blue. So (assuming our normalization from 0 to 1), red is $(1, 0, 0)$, green is $(0, 1, 0)$ and blue is $(0, 0, 1)$. These colors can be added to make other colors, black is $(0, 0, 0)$, white is $(1, 1, 1)$, and complementary colors for the primaries are cyan (light greenish blue), $(0, 1, 1)$, magenta (a purplish red), $(1, 0, 1)$, and yellow, $(1, 1, 0)$.

Note that the RGB vector (R, G, B) is a linear function of the (X, Y, Z) vector, and so the conversion between the two can be accomplished by a matrix multiplication

$$(X, Y, Z) = (R, G, B)M$$

for some matrix M (which depends on the exact color characteristics of the CRT.

CMY Color Model: RGB is an additive color system. However, most printing devices use a subtractive system. This means that the ink absorbs a particular color, and hence what you see is what is reflected (not absorbed). When inks are combined, they absorb a combination of colors, and hence the reflected colors are reduced, or subtracted.

When dealing with subtractive colors, the natural set of primaries to use are the *subtractive primaries*: cyan, magenta, and yellow. It is very easy to convert from RGB to CMY (and vice versa), by simply subtracting each component from 1

$$(C, M, Y) = (1 - R, 1 - G, 1 - B).$$

Thus in the CMY model, black is $(1, 1, 1)$ and white is $(0, 0, 0)$.

Although theoretically correct, this simple conversion does not always produce exactly true results. The reason has to do with spectral overlap, where inks that appear pure in one color actually absorb combinations of other colors. Some small correction factors may be needed to get really accurate conversions.

A variation of CMY is CMYK, where an extra redundant component K has been added to represent black. Many printing systems use black ink, rather than combining C, M, and Y to produce black. This is simply for economical reasons (black is cheaper ink) and because the use of black in character printing is so common.

YIQ Color Model: The YIQ model is used in the US for color television broadcasting. It is a recoding of RGB with some tricks for broadcasting efficiency and compatibility with black-and-white. The Y component is equal to the Y component of the CIE standard. This is basically a measure of luminance, so black-and-white sets simply display this one component. The I and Q components encode hue and saturation. Since our visual system is more sensitive to variations in luminance, the I and Q components can be transmitted with lower bandwidth that what is used for Y.

User oriented systems: The models RGB, CMY, and YIQ are tailored for particular devices (CRT's, printers, and video). User's find these systems (as well as XYZ difficult to work with. For this reason there are systems that describe colors in terms that are more natural. For example, the HSV system defines color in terms of hue (color), saturation (purity), and value (lightness).

Lecture X06: Radiosity

(Supplemental)

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.

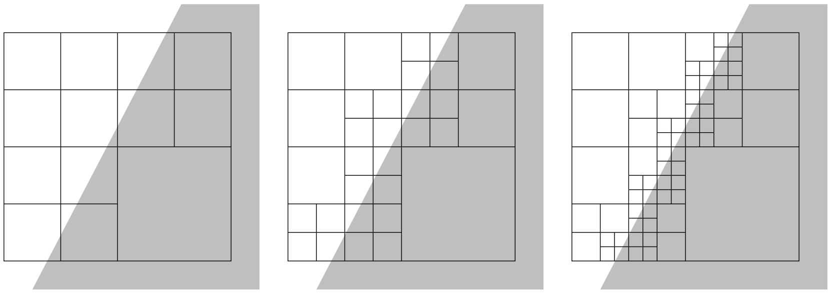


Figure 56: Adaptive Meshing for Radiosity.

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.

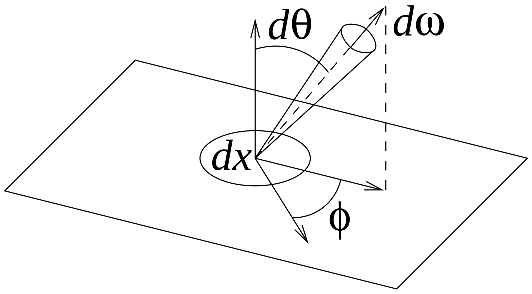


Figure 57: Radiance.

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.

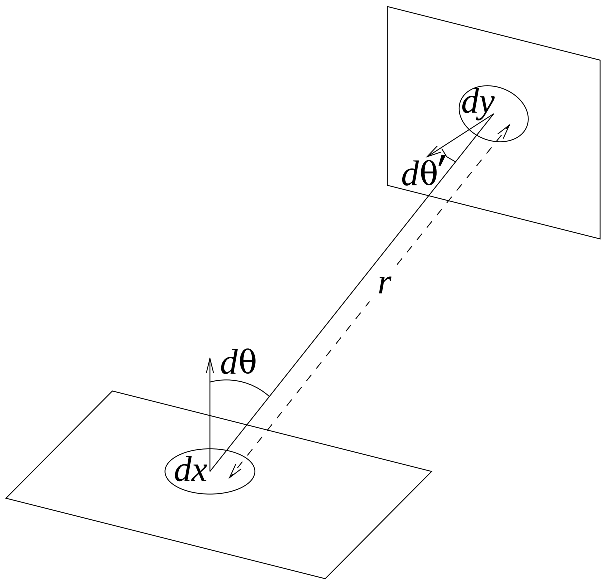


Figure 58: Illumination from another surface.

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.